

---

# Workforce Behaviours in Healthcare Systems

---



Michalis Panayides

School of Mathematics

Cardiff University

A thesis submitted for the degree of

*Doctor of Philosophy*

June 2023



# Abstract

This thesis investigates the behavioural dynamics that emerge at the interface of Emergency Departments (EDs) and the Emergency Medical Service (EMS). The focus is on the impact that time-targets may have on staff behaviour and patient well-being. This research is structured into two main parts: the first part is the development of a queueing theoretic representation of an ED and the second part is the development of a game theoretic model between two EDs and the EMS that distributes ambulances to them.

This thesis uses a variety of mathematical and computational fields such as linear algebra, game theory, queueing theory, graph theory, optimisation, probability theory, agent-based simulation and reinforcement learning.

The queueing model is developed using both a discrete event simulation and a Markov chain approach. The queueing network consists of two queueing nodes where there is some strategic managerial behaviour that relates to how two types of individuals are routed between the two nodes. The first node acts as a buffer for one type of individuals before moving to the second node, while the second node consists of a waiting room and a service centre. Both approaches are used to obtain performance measures of the queueing system and explicit formulas are derived for the mean waiting time, the mean blocking time and the proportion of individuals within a given target time. In addition, some numeric results are presented that compare the Markov chain and discrete event simulation approaches.

Consequently, this thesis describes the development and application of a 3-player game theoretic model between two such queueing networks and a service that distributes individuals to them. In particular the game is then reduced to a 2-player normal-form game. The resultant model is used to explore dynamics between all players. A backwards induction technique is used to get the utilities of the normal-form game between the two queueing systems. The particular game is then applied to a healthcare scenario to capture the emergent behaviour between the EMS and two EDs. The results and outcomes that are produced

by various instances of the game are then analysed and discussed. The learning algorithm replicator dynamics is used to explore the evolutionary behaviours that emerge in the game. In particular, the behaviour that naturally emerges from the game seems to be one that causes more blockage and includes less cooperation. Several ways to escape this learned inefficient behaviour are discussed.

Finally, the thesis explores an extension of the queueing theoretic model that allows servers to choose their own service speed. This is implemented using an agent-based simulation approach. The agent-based model is then used in conjunction with a reinforcement learning algorithm to explore the effect that the servers' behaviour has on the overall performance of the system.



# Acknowledgements

First of all I would like to express my sincerest and greatest gratitude to my incredible supervisors, Dr. Vince Knight and Prof. Paul Harper whose guidance and support helped me grow as a researcher and as a person. Their tolerance, and great sense of humour, helped me through the ups and downs that come with a PhD and I am very grateful for that. I truly believe I could not have had a better pair of supervisors. Thanks to the both of you.

I would also like to thank The Healthcare Improvement Studies Institute (THIS) for funding this research and for providing me with this wonderful experience. In particular I would like to thank the mentor I was assigned by THIS, Prof. Sonya Crowe, for her advice and support. Those couple of meetings were really helpful and valuable. I would also like to thank Prof. Davina Allen for her comments and ethnographic insights that she provided that helped guide my research.

On a more personal note I would like to thank my parents, Anastasia and Panayiotis and my brother Stavros for their unconditional love and support throughout my PhD. A big thank you to my friends in Cardiff as well; my flatmates Nikolas and Vasilis for all the PhD talk they had to endure; Athena for the countless Thursday lunch breaks; and Panos for all the necessary board game nights.

I would also like to thank all of my academic buddies and in particular Elizabeth Williams for her constant help with any random problem I had through my PhD and for constantly looking out for my plant Trev; Matthew Howells for never failing to make the right pun and for the countless nerf gun battles we had; and Michela Corradini for being the most annoying individual I ever met, and for that one thing she helped me fix on Figure D.11 in page 297 of this thesis and kept saying I should include her in my acknowledgements for it. Lastly, thank you to the School of Mathematics staff for maintaining the friendly and supportive environment that I have enjoyed throughout my PhD.

# Dissemination of Work

## Publications

1. Michalis Panayides, Vince Knight, and Paul Harper. **A game theoretic model of the behavioural gaming that takes place at the EMS - ED interface.** *European Journal of Operational Research*, 305(3):1236-1258, 2023.

## Talks & Posters

All copies of the slides and posters are publicly available on GitHub.

- **A game theoretic model of the behavioural gaming that takes place at the EMS-ED interface.** *THIS Space Event, 2020*
- **Using Python to measure the expected wait in a queue with two waiting rooms.** *PyCon Namibia, 2020*
- **How to make an awesome package in python.** *SIAM/IMA PGR seminar series, 2021*
- **The Ambulance Decision Game.** *Poster for School of Mathematics poster day, 2021*
- **A 3-player game theoretic model of a choice between two queueing systems with strategic managerial decision making.** *EURO, 2021*
- **A 3-player game theoretic model of a choice between two queueing systems with strategic managerial decision making.** *Wales mathematics Colloquium, Gregynog Hall, 2022*
- **Recovering from inefficiencies in queueing systems with two consecutive waiting zones.** *CORS/INFORMS conference, 2022*

- A game theoretic model between two Emergency Departments and the Emergency Medical Services. *ORAHs/CHOIR seminar series, University of Twente, 2022*
- A 3-player game theoretic model of a choice between two queueing systems with strategic managerial decision making. *EURO, 2022*
- A game theoretic model between two Emergency Departments and the Emergency Medical Services. *SWORDS, 2022*
- Playing games - An introduction to game theory. *Cardiff University, School outreach day, 2023*

## Software development

- Ambulance\_game, an open source python library for the development of a game theoretic model between two Emergency Departments and the Emergency Medical Services. **Contribution:** Main developer.
- Ciw, a discrete event simulation python library for open queueing networks. **Contributions:** Implementation of server dependent distributions, implementation of server priorities.
- Nashpy, a Python library used for the computation of equilibria in 2 player strategic form games. **Contributions:** Implementation of the learning algorithm, asymmetric replicator dynamics.

# Contents

<b>Abstract</b>	<b>i</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>Dissemination of Work</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 History of Operational Research . . . . .	2
1.2 Congestion in healthcare . . . . .	3
1.3 Research questions and thesis structure . . . . .	4
1.4 Software development and best practices . . . . .	7
1.4.1 Version control . . . . .	8
1.4.2 Virtual environments . . . . .	10
1.4.3 Summary of written software . . . . .	11
1.4.4 Testing and code quality checkers . . . . .	11
1.5 Chapter summary . . . . .	14
<b>2 Literature review</b>	<b>15</b>
2.1 Introduction . . . . .	15
2.2 Operational Research models and Healthcare . . . . .	16
2.3 Game theory and Queueing theory . . . . .	18
2.4 Game theory in Healthcare . . . . .	20
2.5 Behavioural Operational Research . . . . .	21
2.5.1 Agent-based modelling . . . . .	21
2.5.2 Agent-based modelling and healthcare . . . . .	21
2.5.3 Reinforcement learning . . . . .	22
2.6 Chapter summary . . . . .	24
<b>3 Queueing theoretic model</b>	<b>25</b>
3.1 Introduction . . . . .	25
3.2 Discrete Event Simulation . . . . .	26

3.2.1	Implementation . . . . .	28
3.2.2	Custom node class . . . . .	29
3.2.3	Performance Measures . . . . .	32
3.3	Markov chain model . . . . .	33
3.3.1	Steady state probability vector . . . . .	36
3.3.2	Numerical integration approach . . . . .	37
3.3.3	Linear algebraic approach . . . . .	38
3.3.4	Least squares approach . . . . .	39
3.4	Performance measures . . . . .	40
3.4.1	Waiting time . . . . .	42
3.4.2	Blocking time . . . . .	64
3.4.3	Proportion of individuals within target . . . . .	70
3.5	Numeric results and timings . . . . .	85
3.5.1	Markov chain waiting time approaches comparison . . . . .	85
3.5.2	Accuracy of steady state probability calculations . . . . .	87
3.5.3	Computation time of DES and Markov chain . . . . .	91
3.5.4	Truncation effect on performance measures . . . . .	93
3.6	ED-EMS application . . . . .	99
3.7	Chapter summary . . . . .	101
<b>4</b>	<b>Game theoretic model</b>	<b>103</b>
4.1	Introduction . . . . .	103
4.2	Game theory concepts . . . . .	103
4.2.1	Normal form games . . . . .	104
4.2.2	Nash Equilibrium . . . . .	105
4.2.3	Learning Algorithms . . . . .	109
4.2.4	Perfect-information extensive form game . . . . .	115
4.2.5	Imperfect-information extensive form game . . . . .	116
4.3	Formulation . . . . .	117
4.3.1	Players and parameters . . . . .	117
4.3.2	Strategies . . . . .	119
4.3.3	Payoffs . . . . .	123
4.3.4	Imperfect information extensive form game . . . . .	126
4.4	Methodology . . . . .	127
4.4.1	Distribution service and Brent's algorithm . . . . .	127
4.4.2	Routing Matrix . . . . .	142
4.4.3	Queueing systems and normal form games . . . . .	146
4.4.4	Solving the game . . . . .	154
4.5	ED-EMS application . . . . .	158

4.6	Chapter summary . . . . .	159
<b>5</b>	<b>Numerical Results</b>	<b>161</b>
5.1	Introduction . . . . .	161
5.2	Data Collection . . . . .	161
5.3	Dataset description . . . . .	164
5.4	What if scenarios . . . . .	168
5.4.1	Example 1 . . . . .	169
5.4.2	Example 2 . . . . .	173
5.5	Chapter summary . . . . .	184
<b>6</b>	<b>Extending to agent-based model</b>	<b>186</b>
6.1	Introduction . . . . .	186
6.2	State-dependent variation . . . . .	187
6.2.1	Implementation . . . . .	188
6.3	Server-dependent variation . . . . .	190
6.3.1	Implementation . . . . .	190
6.4	State and server-dependent model . . . . .	192
6.4.1	Implementation . . . . .	193
6.4.2	Game theoretic model . . . . .	196
6.5	The agent-based model . . . . .	199
6.5.1	Utility functions . . . . .	200
6.5.2	Case study . . . . .	202
6.6	Reinforcement learning algorithm . . . . .	205
6.6.1	Numeric results . . . . .	208
6.7	Chapter summary . . . . .	215
<b>7</b>	<b>Conclusions</b>	<b>217</b>
7.1	Research Overview . . . . .	217
7.2	Contributions . . . . .	219
7.3	Future Work . . . . .	221
	<b>Appendices</b>	<b>235</b>
<b>A</b>	<b>The ambulance_game Python library</b>	<b>236</b>
A.1	Installation . . . . .	237
A.2	Tutorial . . . . .	237
A.3	How-to guides . . . . .	241
A.3.1	Discrete Event Simulation . . . . .	242
A.3.2	Markov Chains . . . . .	246
A.3.3	Game Theoretic Model . . . . .	251

---

A.4	Reference . . . . .	253
A.5	Explanation . . . . .	257
A.5.1	Additional information . . . . .	257
A.5.2	Other libraries . . . . .	258
<b>B</b>	<b>Game theoretic model - Numerical results</b>	<b>260</b>
B.1	Changing time targets . . . . .	261
B.2	Multiple values of “weight” parameter . . . . .	263
B.3	Other numerical results . . . . .	266
<b>C</b>	<b>Reinforcement Learning - Numerical results</b>	<b>268</b>
<b>D</b>	<b>Steady state probabilities (closed-form)</b>	<b>282</b>
D.1	Graph Theory . . . . .	282
D.2	A graph theoretic model underlying the Markov chain . . . . .	283
D.3	Spanning trees . . . . .	285
D.4	Spanning Trees rooted at (0,0) . . . . .	286
D.5	Conjecture of adding rows . . . . .	288
D.6	The effect of increasing N . . . . .	290
D.7	Unknown terms . . . . .	292
D.8	DRL arrays . . . . .	292
D.9	Examples of mappings of directed spanning trees to permutation arrays . . . . .	295
D.10	Closed-form approach for state probabilities . . . . .	296
D.11	Example of the permutation algorithm . . . . .	299

# List of Figures

1.1	Pull request example on GitHub. . . . .	9
1.2	Conda environment example. . . . .	10
1.3	Python test example. . . . .	12
1.4	GitHub Actions workflow example. . . . .	13
3.1	A diagrammatic representation of the queueing network. The threshold $T$ only applies to type 2 individuals. If the number of individuals in node 1 is greater than or equal to $T$ , only individuals of type 1 are accepted (at a rate $\lambda_1$ ) and individuals of type 2 (arriving at a rate $\lambda_2$ ) are blocked in node 2. . . . .	25
3.2	An equivalent model to the one described in Figure 3.1. The difference between the two diagrams is the formulation of node 2. The original diagram uses a node with no servers and a queueing capacity of $M$ while this one uses $M$ servers with no queueing capacity. . . . .	27
3.3	Generic case of Markov chain model. The diagram shows the two disjoint sets of states $S_1$ and $S_2$ and the transition rates between the states. . . . .	35
3.4	Adjusted case of the Markov chain model. The diagram makes use of the truncated state space $\tilde{S}$ where the state space $S$ is bounded by $N$ and $M$ . . . . .	36
3.5	Variation of Markov chain model where all arrivals are removed. This diagram is used as a visualisation aid to illustrate how the recursive algorithm works. . . . .	43
3.6	Markov chain example with $C = 1, T = 2, N = 3, M = 1$ . . . . .	54
3.7	Variation of Markov chain model where type 2 arrivals are removed (i.e. all arrows pointing down with a rate of $\lambda_1$ are removed). This diagram is used as a visualisation aid for the blocking time formula. . . . .	65
3.8	Example of Markov model with $C = 2, T = 3, N = 6, M = 2$ . . . . .	67
3.9	Example of Markov model with $C = 1, T = 2, N = 4, M = 2$ . . . . .	70



3.10	Example of Markov model with $C = 2, T = 2, N = 4, M = 2$ . . .	72
3.11	Waiting times of the three waiting time approaches for different values of $N$ (left) and the maximum difference in waiting time among the three approaches over different values of $N$ (right). . .	85
3.12	Waiting times of the three waiting time approaches for different values of $M$ (left) and the maximum difference in waiting time among the three approaches over different values of $M$ (right). . .	85
3.13	Computation time of the recursive, direct and closed-form waiting time approaches for different values of $N$ . . . . .	86
3.14	Computation time of the recursive, direct and closed-form waiting time approaches for different values of $M$ . . . . .	86
3.15	Computation time of the recursive, direct and closed-form waiting time approaches for different values of $N$ and $M$ . . . . .	87
3.16	Heatmaps for $\mu = 0.03$ of the state probabilities using the DES approach, the Markov chain approach and the differences between the two. . . . .	88
3.17	Heatmaps for $\mu = 0.09$ of the state probabilities using the DES approach, the Markov chain approach and the differences between the two. . . . .	88
3.18	Heatmaps for $\mu = 0.15$ of the state probabilities using the DES approach, the Markov chain approach and the differences between the two. . . . .	88
3.19	Heatmaps for $\mu = 0.21$ of the state probabilities using the DES approach, the Markov chain approach and the differences between the two. . . . .	89
3.20	Heatmaps for $\mu = 0.27$ of the state probabilities using the DES approach, the Markov chain approach and the differences between the two. . . . .	89
3.21	Heatmaps for $C = 1$ of the state probabilities using the DES approach, the Markov chain approach and the differences between the two. . . . .	90
3.22	Heatmaps for $C = 2$ of the state probabilities using the DES approach, the Markov chain approach and the differences between the two. . . . .	90
3.23	Heatmaps for $C = 3$ of the state probabilities using the DES approach, the Markov chain approach and the differences between the two. . . . .	90

3.24	Heatmaps for $C = 4$ of the state probabilities using the DES approach, the Markov chain approach and the differences between the two. . . . .	90
3.25	Heatmaps for $C = 5$ of the state probabilities using the DES approach, the Markov chain approach and the differences between the two. . . . .	91
3.26	Example 1 - Comparison of overall mean waiting time between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation. . . . .	93
3.27	Example 1 - Comparison of type 1 individuals mean waiting time between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation. . . . .	94
3.28	Example 1 - Comparison of type 2 individuals mean waiting time between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation. . . . .	94
3.29	Example 1 - Comparison of mean blocking time between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation. . . . .	95
3.30	Example 1 - Comparison of overall proportion of individuals within target between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation. . . . .	95
3.31	Example 1 - Comparison of proportion of type 1 individuals within target between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation. . . . .	96
3.32	Example 1 - Comparison of proportion of type 2 individuals within target between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation. . . . .	96
3.33	Example 2 - Comparison of overall mean waiting time between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation. . . . .	97

3.34	Example 2 - Comparison of type 1 individuals mean waiting time between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation. . . . .	97
3.35	Example 2 - Comparison of type 2 individuals mean waiting time between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation. . . . .	97
3.36	Example 2 - Comparison of mean blocking time between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation. . . . .	98
3.37	Example 2 - Comparison of overall proportion of individuals within target between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation. . . . .	98
3.38	Example 2 - Comparison of proportion of type 1 individuals within target between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation. . . . .	98
3.39	Example 2 - Comparison of proportion of type 2 individuals within target between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation. . . . .	99
3.40	A diagrammatic representation of the Emergency Department. Similar to Figure 3.1, the threshold $T$ only applies to type 2 individuals (i.e. patients arriving via an ambulance that can be blocked). If the number of individuals in the hospital's waiting room is greater than or equal to $T$ , only type 1 patients are accepted while patients of type 2 are blocked in the parking space. .	100
4.1	Example of fictitious play algorithm run for 1000 iterations that converges to a Nash equilibrium. . . . .	111
4.2	Example of stochastic fictitious play algorithm for 1000 iterations that converges to a Nash equilibrium. . . . .	112
4.3	Example of asymmetric replicator dynamics algorithm that does not converge. . . . .	114
4.4	Example of asymmetric replicator dynamics algorithm that converges	115

4.5	Example of a perfect information extensive form game with 2 players and 4 terminal nodes. . . . .	116
4.6	An example of an imperfect information extensive form game with 2 players and 4 terminal nodes. . . . .	116
4.7	A diagrammatic representation of the game theoretic model. Individuals arrive at the distribution service at a rate of $\lambda_2$ and then a proportion of them are distributed to Queueing system $A$ ( $p_A$ ) and the remaining proportion to Queueing system $B$ ( $p_B$ ) so that $p_A + p_B = 1$ . The corresponding arrival rates of type 2 individuals to Queueing systems $A$ and $B$ are thus given by: $p_A\lambda_2$ and $p_B\lambda_2$ . . . . .	118
4.8	The Markov chain model that will be generated when queueing system $A$ chooses to play a strategy of $T_A = 1$ . . . . .	119
4.9	The Markov chain model that will be generated when queueing system $A$ chooses to play a strategy of $T_A = 2$ . . . . .	120
4.10	The Markov chain model that will be generated when queueing system $A$ chooses to play a strategy of $T_A = 3$ . . . . .	120
4.11	The Markov chain model that will be generated when queueing system $A$ chooses to play a strategy of $T_A = 4$ . . . . .	121
4.12	The Markov chain model that will be generated when queueing system $A$ chooses to play a strategy of $T_A = 5$ . . . . .	121
4.13	The Markov chain model that will be generated when queueing system $A$ chooses to play a strategy of $T_A = 6$ . . . . .	122
4.14	A diagrammatic representation of the game theoretic model listing the performance measures that correspond to each player's utilities. . . . .	123
4.15	Imperfect information extensive form game between the distribution service and the 2 queueing systems . . . . .	126
4.16	Decision values for queueing system $A$ and queueing system $B$ where ( <i>decision value 1</i> ) = $\alpha L_A(p_A) + (1 - \alpha)B_A(p_A)$ and ( <i>decision value 2</i> ) = $\alpha L_B(p_B) - (1 - \alpha)B_B(p_B)$ . . . . .	129
4.17	Visualisation of Brent's algorithm showing the differences between the two decision values and the point at which the function crosses the $x$ -axis . . . . .	130
4.18	Brent's algorithm example where the service parameter of queueing system $A$ is $\mu^A = 1$ . . . . .	130
4.19	Brent's algorithm example where the service parameter of queueing system $A$ is $\mu^A = 1.5$ . . . . .	131
4.20	Brent's algorithm example where the service parameter of queueing system $A$ is $\mu^A = 2$ . . . . .	131

4.21	Brent's algorithm example where the service parameter of queueing system A is $\mu^A = 2.5$ . . . . .	132
4.22	Brent's algorithm example where the service parameter of queueing system A is $\mu^A = 3$ . . . . .	132
4.23	Decision values for queueing system A and queueing system B (top) and the differences between them (bottom). . . . .	133
4.24	Violinplots of the duration of Brent's algorithm for different values of $N_A$ for the first parameter set. . . . .	140
4.25	Violinplots of the duration of Brent's algorithm for different values of $N_A$ for the second parameter set. . . . .	141
4.26	Line plots of the duration of Brent's algorithm for different values of $N_A$ over different values of the absolute tolerance parameter $\text{xtol}$ for the first parameter set. . . . .	141
4.27	Line plots of the duration of Brent's algorithm for different values of $N_A$ over different values of the absolute tolerance parameter $\text{xtol}$ for the second parameter set. . . . .	142
4.28	Fictitious play algorithm run on the strategies of the players. . . .	156
4.29	Stochastic fictitious play algorithm on the strategies of the players.	157
4.30	Asymmetric replicator dynamics on the strategies of the players. .	158
5.1	Structure of the <b>data</b> directory . . . . .	164
5.2	Number of times each value of $\lambda_2$ is used in the dataset . . . . .	165
5.3	Number of times each value of $\lambda_2$ is used in the dataset for $\lambda_2 \in [0, 10]$	166
5.4	Explored combinations of the values of $C, N$ and $M$ for player A .	167
5.5	Explored combinations of the values of $C, N$ and $M$ for player B .	168
5.6	Example 1: Asymmetric replicator dynamics . . . . .	170
5.7	Example 1: Asymmetric replicator dynamics (what if $t = 1.7$ ) . .	172
5.8	Example 1: Asymmetric replicator dynamics (what if $t = 1.5$ ) . .	172
5.9	Example 2: Asymmetric replicator dynamics . . . . .	176
5.10	Example 2: Compartmentalised <i>PoA</i> . . . . .	177
5.11	Example 2: Asymmetric replicator dynamics and compartmentalised <i>PoA</i> with increased hospital capacity $N$ . . . . .	178
5.12	Example 2: Asymmetric replicator dynamics and compartmentalised <i>PoA</i> with increased parking capacity $M$ . . . . .	179
5.13	Example 2: Asymmetric replicator dynamics and compartmentalised <i>PoA</i> with increased arrival of type 2 patients $\lambda_2$ . . . . .	180
5.14	Example 2: Asymmetric replicator dynamics and compartmentalised <i>PoA</i> with increased number of staff $C$ . . . . .	181

5.15	Example 2: Asymmetric replicator dynamics and compartmentalised $PoA$ with incentivisation. . . . .	184
6.1	Markov chain example with $C = 1, T = 1, N = 2, M = 1$ . . . . .	188
6.2	Asymmetric replicator dynamics algorithm run on the game obtained from the Markov chain model. . . . .	197
6.3	Asymmetric replicator dynamics algorithm run on the game obtained from the DES model using a runtime of 300 time units. . .	197
6.4	Asymmetric replicator dynamics algorithm run on the game obtained from the DES model using a runtime of 500 time units. . .	198
6.5	Asymmetric replicator dynamics algorithm run on the game obtained from the DES model using a runtime of 1000 time units. .	198
6.6	Asymmetric replicator dynamics algorithm run on the game obtained from the DES model using a runtime of 3000 time units and a state and server dependent service rate. . . . .	199
6.7	Example policy for server 1 and server 2 at iteration 1 . . . . .	207
6.8	Example policy for server 1 and server 2 at iteration 2 . . . . .	207
6.9	Example policy for server 1 and server 2 at iteration 3 . . . . .	207
6.10	Utilities (left) and mean service rate (right) of servers from the reinforcement learning run using utility function $U_k^{(3)}$ with $e = 0.1$ and 100,000 iterations . . . . .	209
6.11	Utilities (left) and mean service rate (right) of servers from the reinforcement learning run using utility function $U_k^{(3)}$ with $e = 0.1$ and 500,000 iterations . . . . .	209
6.12	Utilities (left) and weighted mean service rate (right) of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.1$ . . . . .	210
6.13	Utilities (left) and weighted mean service rate (right) of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.1$ and an initial service rate of 0.2 for all servers. . . . .	211
6.14	Utilities (left) and weighted mean service rate (right) of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.1$ and an initial service rate of 1.5 for all servers. . . . .	211
6.15	Utilities (left) and weighted mean service rate (right) of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.5$ . . . . .	212
6.16	Utilities (left) and weighted mean service rate (right) of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.5$ and increased arrival rates of $\lambda_1 = 2$ and $\lambda_2 = 2.5$ . . . . .	212

6.17	Utilities (left) and weighted mean service rate (right) of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.5$ and no upper bound on the service rate. . . . .	213
6.18	Utilities (left) and weighted mean service rate (right) of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.5$ and changing arrival rates throughout the run. . . . .	215
A.1	Structure of the modules in the <code>ambulance_game</code> library . . . . .	253
B.1	Asymmetric replicator dynamics run and PoA of the game theoretic model with time target 1.0 and parameters: $\alpha = 0.97, \lambda_2 = 0.1, \lambda_1^{(1)} = 3.0, \lambda_1^{(2)} = 4.5, \mu^{(1)} = 2.0, \mu^{(2)} = 3.0, C^{(1)} = 3, C^{(2)} = 2, N^{(1)} = 6, N^{(2)} = 7, M^{(1)} = 5, M^{(2)} = 4$ . . . . .	261
B.2	Asymmetric replicator dynamics run and PoA of the game theoretic model with time target 3.0 and parameters: $\alpha = 0.97, \lambda_2 = 0.1, \lambda_1^{(1)} = 3.0, \lambda_1^{(2)} = 4.5, \mu^{(1)} = 2.0, \mu^{(2)} = 3.0, C^{(1)} = 3, C^{(2)} = 2, N^{(1)} = 6, N^{(2)} = 7, M^{(1)} = 5, M^{(2)} = 4$ . . . . .	261
B.3	Asymmetric replicator dynamics run and PoA of the game theoretic model with time target 5.0 and parameters: $\alpha = 0.97, \lambda_2 = 0.1, \lambda_1^{(1)} = 3.0, \lambda_1^{(2)} = 4.5, \mu^{(1)} = 2.0, \mu^{(2)} = 3.0, C^{(1)} = 3, C^{(2)} = 2, N^{(1)} = 6, N^{(2)} = 7, M^{(1)} = 5, M^{(2)} = 4$ . . . . .	262
B.4	Asymmetric replicator dynamics run and PoA of the game theoretic model with time target 7.0 and parameters: $\alpha = 0.97, \lambda_2 = 0.1, \lambda_1^{(1)} = 3.0, \lambda_1^{(2)} = 4.5, \mu^{(1)} = 2.0, \mu^{(2)} = 3.0, C^{(1)} = 3, C^{(2)} = 2, N^{(1)} = 6, N^{(2)} = 7, M^{(1)} = 5, M^{(2)} = 4$ . . . . .	262
B.5	Asymmetric replicator dynamics run and PoA of the game theoretic model with time target 9.0 and parameters: $\alpha = 0.97, \lambda_2 = 0.1, \lambda_1^{(1)} = 3.0, \lambda_1^{(2)} = 4.5, \mu^{(1)} = 2.0, \mu^{(2)} = 3.0, C^{(1)} = 3, C^{(2)} = 2, N^{(1)} = 6, N^{(2)} = 7, M^{(1)} = 5, M^{(2)} = 4$ . . . . .	263
B.6	Asymmetric replicator dynamics run and PoA of the game theoretic model with $\alpha = 0.0$ and parameters: $\lambda_2 = 32.05, \lambda_1^{(1)} = 0.0, \lambda_1^{(2)} = 0.0, \mu^{(1)} = 4.2, \mu^{(2)} = 6.6, C^{(1)} = 1, C^{(2)} = 3, N^{(1)} = 2, N^{(2)} = 6, M^{(1)} = 7, M^{(2)} = 4, t = 2.0$ . . . . .	263
B.7	Asymmetric replicator dynamics run and PoA of the game theoretic model with $\alpha = 0.3$ and parameters: $\lambda_2 = 32.05, \lambda_1^{(1)} = 0.0, \lambda_1^{(2)} = 0.0, \mu^{(1)} = 4.2, \mu^{(2)} = 6.6, C^{(1)} = 1, C^{(2)} = 3, N^{(1)} = 2, N^{(2)} = 6, M^{(1)} = 7, M^{(2)} = 4, t = 2.0$ . . . . .	264

B.8	Asymmetric replicator dynamics run and PoA of the game theoretic model with $\alpha = 0.6$ and parameters: $\lambda_2 = 32.05, \lambda_1^{(1)} = 0.0, \lambda_1^{(2)} = 0.0, \mu^{(1)} = 4.2, \mu^{(2)} = 6.6, C^{(1)} = 1, C^{(2)} = 3, N^{(1)} = 2, N^{(2)} = 6, M^{(1)} = 7, M^{(2)} = 4, t = 2.0$ . . . . .	264
B.9	Asymmetric replicator dynamics run and PoA of the game theoretic model with $\alpha = 0.9$ and parameters: $\lambda_2 = 32.05, \lambda_1^{(1)} = 0.0, \lambda_1^{(2)} = 0.0, \mu^{(1)} = 4.2, \mu^{(2)} = 6.6, C^{(1)} = 1, C^{(2)} = 3, N^{(1)} = 2, N^{(2)} = 6, M^{(1)} = 7, M^{(2)} = 4, t = 2.0$ . . . . .	265
B.10	Asymmetric replicator dynamics run and PoA of the game theoretic model with $\alpha = 1$ and parameters: $\lambda_2 = 32.05, \lambda_1^{(1)} = 0.0, \lambda_1^{(2)} = 0.0, \mu^{(1)} = 4.2, \mu^{(2)} = 6.6, C^{(1)} = 1, C^{(2)} = 3, N^{(1)} = 2, N^{(2)} = 6, M^{(1)} = 7, M^{(2)} = 4, t = 2.0$ . . . . .	265
B.11	Asymmetric replicator dynamics run and PoA of the game theoretic model with parameters: $\alpha = 0.95, \lambda_2 = 36.04, t = 6.0, \lambda_1^{(1)} = 15.24, \lambda_1^{(2)} = 0.0, \mu^{(1)} = 6.77, \mu^{(2)} = 2.22, C^{(1)} = 9, C^{(2)} = 9, N^{(1)} = 10, N^{(2)} = 9, M^{(1)} = 4, M^{(2)} = 3$ . . . . .	266
B.12	Asymmetric replicator dynamics run and PoA of the game theoretic model with parameters: $\alpha = 0.96, \lambda_2 = 21.4, t = 1.0, \lambda_1^{(1)} = 4.2, \lambda_1^{(2)} = 19.8, \mu^{(1)} = 4.2, \mu^{(2)} = 6.6, C^{(1)} = 1, C^{(2)} = 3, N^{(1)} = 2, N^{(2)} = 6, M^{(1)} = 7, M^{(2)} = 4$ . . . . .	266
B.13	Asymmetric replicator dynamics run and PoA of the game theoretic model with parameters: $\alpha = 0.97, \lambda_2 = 18.7, t = 2.0, \lambda_1^{(1)} = 4.5, \lambda_1^{(2)} = 3.0, \mu^{(1)} = 2.0, \mu^{(2)} = 3.0, C^{(1)} = 3, C^{(2)} = 2, N^{(1)} = 6, N^{(2)} = 7, M^{(1)} = 5, M^{(2)} = 4$ . . . . .	267
B.14	Asymmetric replicator dynamics run and PoA of the game theoretic model with parameters: $\alpha = 1.0, \lambda_2 = 24.0, t = 5.0, \lambda_1^{(1)} = 6.0, \lambda_1^{(2)} = 4.5, \mu^{(1)} = 2.0, \mu^{(2)} = 3.0, C^{(1)} = 3, C^{(2)} = 2, N^{(1)} = 6, N^{(2)} = 7, M^{(1)} = 5, M^{(2)} = 4$ . . . . .	267
C.1	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(3)}$ with $e = 0$ and 100,000 time steps . . . . .	268
C.2	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(3)}$ with $e = 0.5$ and 100,000 time steps . . . . .	269
C.3	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(3)}$ with $e = 1$ and 100,000 time steps . . . . .	269



C.4	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(3)}$ with $e = 0$ and 500,000 time steps . . . . .	269
C.5	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(3)}$ with $e = 0.2$ and 500,000 time steps . . . . .	270
C.6	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(3)}$ with $e = 0.5$ and 500,000 time steps . . . . .	270
C.7	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(3)}$ with $e = 1$ and 500,000 time steps . . . . .	270
C.8	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(3)}$ with $e = 0.1$ , 500,000 time steps and an initial service rate of $\mu = 1$ for all servers . . . . .	271
C.9	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(3)}$ with $e = 0.1$ , 500,000 time steps and an initial service rate of $\mu = 0.1$ for all servers . . . . .	271
C.10	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(3)}$ with $e = 0.1$ , 500,000 time steps and an initial service rate of $\mu = 0.5$ for all servers . . . . .	271
C.11	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(3)}$ with $e = 1$ and 500,000 time steps decreasing $\lambda_2$ to 0.5 . . . . .	272
C.12	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(3)}$ with $e = 1$ and 500,000 time steps increasing $\lambda_2$ to 1.5 . . . . .	272
C.13	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0$ . . . . .	272
C.14	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.1$ . . . . .	273
C.15	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.2$ . . . . .	273
C.16	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.3$ . . . . .	273
C.17	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.4$ . . . . .	274
C.18	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.5$ . . . . .	274

C.19 Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.6$ . . . . .	274
C.20 Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.7$ . . . . .	275
C.21 Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.8$ . . . . .	275
C.22 Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.9$ . . . . .	275
C.23 Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 1$ . . . . .	276
C.24 Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.1$ (only the early iterations) . . . . .	276
C.25 Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.1$ (only the late iterations) . . . . .	276
C.26 Utilities and approximation of the weighted mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.1$ . Note that the approximation uses the Markov chain model to get the state probabilities instead of the DES state probabilities. . . . .	277
C.27 Utilities and approximation of the weighted mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.5$ . Note that the approximation uses the Markov chain model to get the state probabilities instead of the DES state probabilities. . . . .	277
C.28 Utilities and approximation of the weighted mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.9$ . Note that the approximation uses the Markov chain model to get the state probabilities instead of the DES state probabilities. . . . .	277
C.29 Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.1$ and increased arrival rates of $\lambda_1 = 1.0$ and $\lambda_2 = 1.5$ . . . . .	278
C.30 Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.5$ and increased arrival rates of $\lambda_1 = 1.0$ and $\lambda_2 = 1.5$ . . . . .	278

C.31	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.5$ and increased arrival rates of $\lambda_1 = 3.0$ and $\lambda_2 = 3.5$ . . . . .	278
C.32	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.5$ and increased arrival rates of $\lambda_1 = 4.0$ and $\lambda_2 = 4.5$ . . . . .	279
C.33	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.9$ and increased arrival rates of $\lambda_1 = 1.0$ and $\lambda_2 = 1.5$ . . . . .	279
C.34	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.9$ . . . . .	279
C.35	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.1$ and no upper bound on the service rate. . . . .	280
C.36	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.9$ and no upper bound on the service rate. . . . .	280
C.37	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.5$ and increased arrival rates of $\lambda_1 = 1.0$ and $\lambda_2 = 1.5$ . . . . .	280
C.38	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.5$ and increased arrival rates of $\lambda_1 = 3.0$ and $\lambda_2 = 3.5$ . . . . .	281
C.39	Utilities and mean service rate of servers from the reinforcement learning run using utility function $U_k^{(7)}$ with $e = 0.5$ and increased arrival rates of $\lambda_1 = 4.0$ and $\lambda_2 = 4.5$ . . . . .	281
D.1	An example of a simple undirected graph with 5 vertices and 7 edges.	282
D.2	An example of a directed graph with 5 vertices and 7 edges. . . .	283
D.3	An example of a weighted directed graph with 5 vertices and 7 edges.	283
D.4	Example of a Markov model with $C = 1, T = 3, N = 5, M = 2$ . .	284
D.5	Example of one of the spanning trees rooted at vertex 3 of the directed graph. Note that the graph corresponds to a Markov model with parameters $T = 1, N = 3, M = 1$ . . . . .	285
D.6	Examples of Markov chain models with different values of $M$ , where $M = 1$ (left), $M = 2$ (middle) and $M = 3$ (right). . . . .	289
D.7	Markov chain model with $C = 1, T = 1, N = 3, M = 1$ with all of its equivalent spanning trees rooted at $(0, 0)$ . . . . .	293

---

D.8	Example of a permutation that does not produce a valid spanning tree based on Rule 1 . . . . .	295
D.9	Example of a permutation that does not produce a valid spanning tree based on Rule 2 . . . . .	295
D.10	The two sets of permutations that should not be considered in the calculation of $R(p_1, p_2, p_3)$ . The left-most set consists of all the permutations that are excluded by Rule 1 and the right-most are excluded by Rule 2 . . . . .	297
D.11	All permutations that are excluded by Rule 1 and Rule 2 partitioned into four disjoint subsets. . . . .	297

# List of Tables

1.1	List of tools used to check the code quality. . . . .	12
3.1	Parameter values for steady state probabilities accuracy example 1	88
3.2	Parameter values for steady state probabilities accuracy example 2	91
3.3	Relative timings for the computational time needed to get performance measures using the DES and Markov chain models. Note that these timings are all relative to the DES run with a single trial.	92
3.4	Relative timings for the computational time needed to get performance measures using the DES model and the Markov chain model with the smarter approach. . . . .	92
3.5	Parameter values for truncation effect example 1 . . . . .	93
3.6	Parameter values for truncation effect example 2 . . . . .	97
4.1	A game theoretic matrix representation of the Prisoner's Dilemma game . . . . .	104
4.2	Parameter values for Brent's method examples. . . . .	129
4.3	Parameter sets that were used for the tolerance sensitivity analysis.	140
4.4	Parameter values for routing matrix example. . . . .	143
4.5	Parameter values for game formulation example . . . . .	148
4.6	Parameter values for example on solving the game . . . . .	154
5.1	Data collection: Parameters . . . . .	162
5.2	Contents of <code>_parameters/main.csv</code> file . . . . .	163
5.3	Descriptive statistics of the dataset . . . . .	165
5.4	Parameter values for the first example of the what if scenarios. . .	169
5.5	Example 1: Strategies played and performance measures . . . . .	173
5.6	Parameter values for the second example of the what if scenarios.	173
6.1	Parameter values for game theoretic model example to observe the differences of running the game with DES and Markov chains. . .	197
6.2	Performance measures that could affect each agent's utility . . . .	200

6.3	Utility functions that can be used to measure each server's happiness	201
6.4	Parameter values used in the case study. . . . .	203
6.5	Each server's performance measure for a run of the simulation. . .	204
6.6	Utility function 1 ( $U_k^{(1)}$ ) for the 4 servers and different values of $e$	204
6.7	Utility function 2 ( $U_k^{(2)}$ ) for the 4 servers and different values of $e$	204
6.8	Utility function 3 ( $U_k^{(3)}$ ) for the 4 servers and different values of $e$	204
6.9	Utility function 4 ( $U_k^{(4)}$ ) for the 4 servers and different values of $e$	204
6.10	Utility function 5 ( $U_k^{(5)}$ ) for the 4 servers and different values of $e$	205
6.11	Utility function 6 ( $U_k^{(6)}$ ) for the 4 servers and different values of $e$	205
6.12	Utility function 7 ( $U_k^{(7)}$ ) for the 4 servers and different values of $e$	205
6.13	Parameter values for the reinforcement learning algorithm experiments. . . . .	208
A.1	Parameters of the game . . . . .	237

# Listings

3.1	Using the <code>ciw</code> library to create a queueing network with two queues and two types of individuals and the particular structure defined in Figure 3.2. . . . .	28
3.2	Function that contains the custom node class that blocks individuals to node 1 when the number of individuals in node 2 exceeds the threshold. . . . .	29
3.3	Building and simulating the network model using the custom node	31
3.4	Function that extracts performance measures from the simulation records . . . . .	32
3.5	Using the function defined in 3.4 . . . . .	32
3.6	Steady state probabilities calculation using numeric integration using the <code>odeint</code> function. . . . .	37
3.7	Steady state probabilities calculation using numeric integration using the <code>solve_ivp</code> function. . . . .	38
3.8	Steady state probabilities calculation using linear algebraic approach with <code>numpy.linalg.solve</code> . . . . .	39
3.9	Steady state probabilities calculation using least squares approach with <code>numpy.linalg.lstsq</code> . . . . .	39
3.10	Code snippet for getting the set of all states and the steady state probabilities. . . . .	40
3.11	Code snippet for calculating the mean number of individuals in the system. . . . .	41
3.12	Code snippet for calculating the mean number of individuals in Node 1. . . . .	41
3.13	Code snippet for calculating the mean number of individuals in Node 2. . . . .	42
3.14	Function that checks if a state is a waiting state. . . . .	46
3.15	Function for the expected waiting time in a state. . . . .	47
3.16	Function for the overall expected waiting time in a state using recursion. . . . .	48

3.17	Function to check if a state is an accepting state. . . . .	49
3.18	Function to get the mean waiting time recursively for a specific individual type. . . . .	50
3.19	Function for the overall waiting time formula. . . . .	52
3.20	Function to get a row of the coefficients matrix and the corresponding value of the right hand side vector. . . . .	57
3.21	Function that formulates (but does not solve) the linear system needed to get the waiting time. . . . .	58
3.22	Functions to solve the linear system and get waiting time for each state . . . . .	59
3.23	Function to calculate the mean waiting time for a given individual type using the direct approach . . . . .	60
3.24	Function to calculate the mean waiting time for a given individual type using the closed-form approach . . . . .	62
3.25	Usage of the function to calculate the mean blocking time. . . . .	69
3.26	Function for the simplified version of $\Psi_{k,\lambda}(t)$ . . . . .	79
3.27	Function for the cumulative distribution of the Hypoexponential distribution . . . . .	80
3.28	Function for the cumulative distribution of the Erlang distribution . . . . .	81
3.29	Function for deciding which distribution to use to calculate the probability of an individual being within a given time target. . . . .	81
3.30	Function for calculating the probability of spending less time than the target in the system for a given individual type. . . . .	83
3.31	Combining all functions to calculate the proportion of type 1 individuals within a time target of 1 time unit. . . . .	84
4.1	Lemke-Howson python code for the pig and piglet game. . . . .	107
4.2	Lemke-Howson python code for the Rock-Paper-Scissors game. . . . .	107
4.3	Support enumeration python code for the coordination game. . . . .	109
4.4	Fictitious play python code for a 2-player game. . . . .	110
4.5	Stochastic fictitious play python code for a 2-player game. . . . .	111
4.6	Asymmetric replicator dynamics python code for a 2-player game. . . . .	113
4.7	Asymmetric replicator dynamics run on a game that is able to reach steady state . . . . .	114
4.8	Function that gets the mean blocking difference using the Markov chain model . . . . .	134
4.9	Using the function defined in 4.8 to calculate the difference between the decision values of the two systems at $p_A = 0.5$ . . . . .	135
4.10	Using Brent's algorithm to find the point where the differences of the two decision values are zero. . . . .	136



4.11	Calculating the optimal split between type 1 and type 2 individuals. Note that the outcome of the function is the value of $p_A$ while the value of $p_B$ is given by $1 - p_A$ . . . . .	138
4.12	Function that returns the routing matrix for a given set of game parameters. . . . .	144
4.13	Using the function defined in the code snippet 4.12 to get the routing matrix . . . . .	146
4.14	Variables that correspond to the parameter set from Table 4.5 . .	148
4.15	Function that takes as inputs the given strategies (thresholds) of the players and calculates the corresponding utilities of the players.	148
4.16	Example of using the function defined in 4.15 . . . . .	150
4.17	Function that returns the payoff matrices and the routing matrix	151
4.18	Example of code that gets the values of the payoff matrices and the routing matrix for example 1 . . . . .	152
4.19	Build the <code>nashpy</code> object that consists of the game . . . . .	153
4.20	Variables that correspond to the parameter set from Table 4.6 . .	154
4.21	Using the <code>ambulance_game</code> library to build the game . . . . .	155
4.22	Support enumeration and Lemke-Howson computation on the payoff matrices of the two players . . . . .	155
4.23	Fictitious play algorithm on the payoff matrices of the two players	156
4.24	Stochastic fictitious play algorithm on the payoff matrices of the two players . . . . .	157
4.25	Asymmetric replicator dynamics algorithm on the payoff matrices of the two players . . . . .	157
5.1	Example of saving and loading compressed data . . . . .	163
6.1	State-dependent service time distribution class. . . . .	188
6.2	Example of the state-dependent variation of the queueing system.	189
6.3	Server-dependent service time distribution class. . . . .	190
6.4	Example of the server-dependent variation of the queueing system.	191
6.5	Example of the state and server-dependent variation of the queueing system . . . . .	193
6.6	Example code of the state and server-dependent variation of the queueing system . . . . .	195
6.7	Example of using fair allocation of individuals to servers . . . . .	195
A.1	Python code that defines the parameters. . . . .	238
A.2	Python code that defines the game instance. . . . .	238
A.3	Python code that finds a Nash equilibrium of the game instance. .	239
A.4	Python code that runs the asymmetric replicator dynamics algorithm. . . . .	239

A.5	Python code for the strategy of the third player. . . . .	240
A.6	Python code for the waiting time of patients at each hospital. . .	240
A.7	Python code for the mean total time of patients at each hospital.	241
A.8	The average steady state probabilities of the model based on the simulation. . . . .	245
A.9	Examples of how to define the state-dependent, server-dependent and state and server dependent service rates. . . . .	245
A.10	The simulation using the state and server dependent service rate.	246
A.11	Python code for the parameters used in the Markov chain model.	247
A.12	Python code for building the states of the Markov chain model. .	247
A.13	Python code for visualising the Markov chain model. . . . .	247
A.14	Python code for building the transition matrix of the Markov chain model. . . . .	248
A.15	Python code for building the steady state probabilities of the Markov chain model. . . . .	249
A.16	Python code for getting the expected number of patients in the Markov chain model. . . . .	249
A.17	Python code for getting the expected waiting time of individuals in the Markov chain model. . . . .	249
A.18	Python code for getting the expected blocking time of type 2 in- dividuals in the Markov chain model. . . . .	250
A.19	Python code for getting the proportion of individuals within target using the Markov chain model. . . . .	250
A.20	Parameters for the game theoretic model. . . . .	251
A.21	Calculating the best response of the ambulance service given the strategies of the hospitals $(T_1, T_2)$ . . . . .	251
A.22	Routing matrix for the ambulance service. . . . .	252
A.23	Building the game and getting the Nash equilibrium of the game.	252

# Chapter 1

## Introduction

This thesis focuses on the application of Operational Research (OR) to healthcare systems and in particular at the interface of Emergency Departments (EDs) and Emergency Medical Services (EMS). OR models have been used a lot in the past in conjunction with healthcare systems to better inform the decision-making of healthcare managers and policy makers. The use of OR in healthcare systems can be particularly useful in situations where improvements in the system are needed to cope with the demand for healthcare services.

OR is a field of study that makes use of mathematical and computational techniques to solve problems in a wide range of areas, such as healthcare, business, engineering, and the military. The core of OR has been to help decision makers make better decisions by providing them with the tools to analyse and understand the problem at hand and the possible solutions to it. Such mathematical tools include queueing theory, game theory, decision theory, statistical analysis, machine learning and many more.

This research was funded by The Healthcare Improvement Studies (THIS) Institute of the University of Cambridge. The THIS Institute is a research institute that aims to create a world-leading scientific asset for the National Health Service (NHS) about how to improve quality and safety in healthcare.

The introductory Chapter is structured in the following way:

- Section 1.1 provides an overview of the history of OR and its development, along with a brief introduction to queueing theory and game theory.
- Section 1.2 introduces the problem of congestion in healthcare systems and in particular in EDs.

- Section 1.3 presents the research questions and objectives of this thesis, along with the overall structure of the thesis.
- Section 1.4 provides an overview of the software development tools and best practices that were used in this research.

## 1.1 History of Operational Research

The history of OR can be traced back to the 1940s during World War II when the UK forces started to use OR techniques to optimise the use of their aircrafts. Patrick Blackett was a British physicist and a mathematician and was one of the first people to use OR techniques to solve problems in the military and was later referred to as the *father of OR* [19]. At the time, many strategic problems were too complicated to solve by any one person or a single discipline. Scientists and mathematicians from different fields were brought together to solve problems such as finding the best strategies of air defence, minimising losses from submarines and radar deployment. These teams were called *Operational Research teams* or *Operations Research teams* and the collection of techniques that they used were later used to form the discipline of OR [120].

After the war, the discipline of OR started to grow and spread to other industries and fields. Universities started advancing the field by developing new techniques and methods and later started to offer undergraduate and postgraduate courses on it. Additionally, the introduction of electronic computers allowed OR techniques to be used in more complex problems [120]. The first formal definition of OR was given by the British Operational Research Society:

“Operational research is the application of the methods of science to complex problems arising in the direction and management of large systems of men, machines, materials and money in industry, business, government, and defence. The distinctive approach is to develop a scientific model of the system, incorporating measurement of factors such as chance and risk, with which to predict and compare the outcomes of alternative decisions, strategies or controls. The purpose is to help management determine its policy and actions scientifically” [134].

There are numerous methods and techniques that form the discipline of OR. Some of the most common ones are mathematical programming, queueing theory, game theory, decision theory, data mining and statistical analysis. The main OR techniques that are used in this thesis are queueing theoretic models and game theoretic models.

Queueing theory is a branch of OR that studies queues like those found in banks, supermarkets, hospitals and many more. A queueing model is an abstract representation of a system whose purpose is to understand the underlying behaviour of the system so that informed and intelligent decisions can be made. Queueing theory was first introduced in 1909 by A.K. Erlang, who was a Danish mathematician and engineer, where it was applied to the problem of telecommunications [3]. Erlang placed the foundation for the Poisson distribution in queueing theory which then led to the development of the Exponential distribution. The motivation for the work around queueing theory between 1920 and 1930 has been the practical problem of congestion. In the early 1950s applications of queueing theory started to develop outside the field of telecommunications and in 1960s queueing theory was utilised for the performance evaluation of computer systems. Ever since, queueing theory and its applications have been growing and expanding to many different areas [137].

Another branch of OR that is used in this thesis is game theory. Game theory focuses on the study of strategic interaction between players and the outcomes of such interactions. It is widely used to study behavioural patterns of players in a game and explore strategies that players can use to maximise their payoff. Although mathematicians have been studying strategic games for a long time now, it was not until 1940s that game theory started receiving more attention with the publication of John von Neumann and Oskar Morgenstern [153]. The book introduced the mathematical theory of economic and social organisation, based on a theory of games of strategy. Around 1950, mathematician John Nash developed a criterion for mutual consistency of players' strategies, which is a concept now known as the *Nash equilibrium*. Nash proved that for every finite  $n$ -player, non-cooperative game there exists a Nash equilibrium in mixed strategies. In the next few decades, game theory started to be applied to many different areas such as economics, biology, politics, psychology and many more. In the last few years, game theory has also been applied to healthcare systems [30, 82].

## 1.2 Congestion in healthcare

EDs are one of the most important parts of a hospital and are the first point of contact for individuals seeking medical attention. It is also one of the most congested areas of a hospital and is often the cause of long waiting times for patients. EDs are under increasing pressure to meet patient waiting time targets and satisfy regulation targets [61]. It is widely reported that ED congestion severely impacts not only patients in the ED but also the EMS [75, 89, 94]. One

of the major concern for the ambulance service is that the ambulances are held waiting parked outside the EDs to offload (dispatch) their patients when the ED is particularly busy [32]. As a result, ambulance blocking not only impacts patients that are waiting for ED service, but has a major knock-on effect on the ability of ambulances to respond to new EMS calls, and thus placing lives at risk [38].

There are numerous news articles that address the complications that arise when ambulances stay blocked outside of hospitals for a long amount of time [4, 35]. Most such news reports comment on the long idle time of ambulances when being blocked outside of hospitals and not being utilised as best as they could be [144]. In addition, there are several reports of examples where this became an issue for new patients that needed to be transported to a hospital but were unable to do so due to the ambulance taking too long to reach them [97]. Some even mention the negative effect that this has on the morale of ambulance paramedics [33].

In the United Kingdom, the NHS sets some regulations on ED performance. One of these regulations is that 95% of patients that arrive at the ED should be admitted, transferred or discharged within four hours. This is where gaming behaviour might be observed between the EDs and the EMS.

### 1.3 Research questions and thesis structure

This thesis aims to explore behavioural patterns that emerge at the ED-EMS interface using a game theoretic model that is informed by an underlying queueing network. A model is developed that describes the situation where an ambulance service would have to distribute its patients between two EDs. The two EDs are thought of as two queueing systems and the EMS as a distributor that decides how to distribute patients to them, aiming to minimise some performance measure. The patients that are distributed by the EMS arrive at the EDs via an ambulance and are then either offloaded at the ED or stay blocked outside in the ambulance. This is where gaming behaviour is incorporated into the model. The managerial decision of whether or not to offload a patient is a strategic decision that is made by the two EDs. This decision is mapped to a parameter that will be referred to as the *threshold* of the ED. A high threshold indicates that the ED accepts ambulance patients more often, while a low threshold means that the ED blocks ambulances more frequently. The model is then used to explore the emergent behaviour of the system.

The main research questions of the research presented in this thesis are:

- How can queueing theory be used to model an Emergency Department that

can accept and block patients from an ambulance service?

- How can one extract performance measures from such a queueing model?
- How can game theory be used to model the interaction between the EMS and two EDs?
- How can the developed model be used to explore the emergent behaviour of the system?
- How can agent-based modelling be used to model how staff at the EDs may choose the speed at which they serve patients in order to maximise some utility?

Specifically, the focus is on the construction of a 3-player game theoretic model between two queueing systems and a service that distributes individuals to them. The resultant model is used to explore the emergent dynamics between the three players. This study explores two new concepts: obtaining performance measures for a new queueing theoretic model with a service centre and a buffer space and, and using a learning algorithm to model the emergence of behaviour. The developed theoretical model is illustrated through the application to a healthcare system of two EDs and the EMS, exploring the inefficiencies that emerge and ways to apply some incentive mechanisms to improve them. The EDs are modelled as two queueing systems each with a tandem buffer and a service centre. The performance measures are then used as the utilities of the game. The novelty of the queueing model here is a contribution not only to game theoretic literature but also to the queueing theoretic literature, since no such model of a tandem queueing model with a pair of parameters for the buffer has been previously considered.

This thesis aims to explore the behaviour that emerges from an interactive situation between two queueing systems that aim to maximise their own utility. In addition, this research focuses on the impact that this may have on the ambulance's blocking time and patients' waiting time.

The research presented on this thesis is based on the following assumptions:

- Some managerial decision making is involved in choosing when to start the blockage of ambulances.
- The ambulance service decides how patients are distributed to different hospitals rather than choosing the nearest hospital for each patient.

- The waiting time of a patient, arriving by an ambulance in the ED, is measured from the time they are offloaded to the ED itself, rather than from the time the ambulance arrives at the parking space.

The research presented in this thesis consists of four main chapters. That is a chapter on the queueing theory model that is presented introduced in this thesis, a chapter on the game theoretic model that uses the queueing model in its construction, a chapter on the numerical results of the game theoretic model and a chapter on the agent-based model that is used to explore some additional emergence of behaviour. Overall, the thesis is structured as follows:

- Chapter 1 introduces the problem and motivation behind the research presented in this thesis. A background of OR as well as an overview of the software development process and best practices are also presented.
- Chapter 2 presents a literature review of the relevant research. This includes a review of the literature on OR models applied to healthcare systems, a review of the conjunction of queueing theory and game theory and a review of the literature on game theoretic models applied to healthcare systems. Moreover, a brief review on behavioural OR is also presented to provide some context for the agent-based model that is presented in this thesis.
- Chapter 3 introduces a queueing network model that accepts two types of individuals and has two waiting spaces. Two modelling approaches are discussed; Discrete Event Simulation (DES) and the Markov chain (MC) approach. The chapter mainly focuses on the MC approach and presents how the steady state probabilities and certain performance measures can be obtained from it. This system is then used to describe an ED that accepts patients arriving by ambulance and patients that arrive by other means.
- Chapter 4 presents a game theoretic model that is informed by the queueing network model. The chapter starts by giving a brief overview of game theoretic concepts that are utilised in this chapter. The formulation of the game theoretic model is then presented along with the methodology that was used to solve it. The model is then mapped to a 3-player game between the EMS and two EDs.
- Chapter 5 presents an overview of the numerical results of the game theoretic model. The chapter gives an overview of the data collection process as well as a brief description of the data parameters explored. The results of the numerical experiments are then presented and discussed.



- Chapter 6 presents an extension to the queueing model discussed in Chapter 3. The queueing model is extended to use state-dependent and server-dependent service times instead of the constant service times that were previously used. An agent-based model is constructed where there are different service times for each server and each state of the system. A reinforcement learning algorithm is then used to observe the learning that takes place when servers choose the speed at which they serve individuals in order to maximise some utility.
- Chapter 7 presents an overall summary of the research presented in this thesis. The chapter also lists the main contributions of this thesis and presents some suggestions for future work.

## 1.4 Software development and best practices

Scientists and researchers are increasingly using software development tools to conduct their research. In fact the use of software is becoming so prevalent that it is now considered a fundamental part of the scientific process. Yet software is not always developed following practices that ensure its reproducibility and sustainability. The best practices discussed in this section promote better quality software, which in turn improves the research's reproducibility and reusability [70].

Open Source Software (OSS) is software with source code that everyone can access, modify and improve. OSS improves accessibility, reproduction, transparency and innovation in scientific research [101]. Moreover, OSS development encourages developer-user communities and builds trust among users [99].

The following list is a summary of best practices as described in [158]:

1. Write programs for people, not computers.
2. Let the computer do the work.
3. Make incremental changes.
4. Don't repeat yourself (or others).
5. Plan for mistakes.
6. Optimise software only after it works correctly.
7. Document design and purpose, not mechanics.

## 8. Collaborate.

This is not an exhaustive list of best practices, rather it is a list of principles that can be used to guide the development of software. Software tools that are used together with these principles are version control, code reviews, automated testing, code formatting and documentation. These principles were used to guide the development of the software that is presented in this thesis. These concepts are also discussed in more detail in the following sections.

This thesis relies heavily on the use of software. All code written for this thesis is written in the open source programming language Python [148]. In particular, all software developed for this thesis is publicly available on GitHub and is licensed under the MIT license. The MIT license is a permissive license that allows users to inspect, modify and redistribute the software. The repository for the software developed for this thesis has also been archived using Zenodo [111].

### 1.4.1 Version control

A *version control system* (VCS) is a system that manages the evolution of an ongoing object. It records any changes made by the software developers and allows them to revert to previous versions of the software. The adoption of VCS has become widespread in software development and has empowered software developers to work effectively and collaboratively on large projects.

Version control usually consists of three main components: a repository, a working directory and a staging area. The *repository* is the place where the software is stored. The *working directory* is the place where the software is being developed. The *staging area* is the place where the changes made to the software are stored before they are committed to the history of the repository [20].

There are two main types of VCS: centralized VCS and distributed VCS. Centralized VCS is a VCS where all the software developers work with a single central repository. The central repository is the only place where the software is stored and where all the changes are made. The most commonly used centralized VCS is Subversion and CVS. The distributed VCS is a VCS where all the software developers work with a local copy of the software. The local copy of the software is then synchronised with a central repository where the changes are made [160]. Some examples of distributed VCS are Git, Mercurial and Bazaar. For the purposes of this thesis, the distributed VCS Git [130] is used.

Git is a distributed VCS that is used to track changes in source code during software development. There are a number of services that host online Git servers

that can be used to store Git repositories. The importance of using a Git server is that it allows for collaboration and makes, not only the code, but also the history of the code available to everyone. Some examples of Git servers are GitHub, GitLab and BitBucket. For the purposes of this thesis, the Git server GitHub is used.

GitHub has several features that make it a good choice for hosting Git repositories. It is a web-based Git repository hosting service that allows users to collaborate on projects. Every new feature or bug fix that was developed for this thesis was developed in a separate branch of the repository. GitHub allows users to create branches that are used to develop features independently of the main code base. Once the feature is complete, the branch is merged into the main code base. This process is called *branching* and it is a core concept of Git. Branching allows developers to work on multiple features at the same time and to collaborate with other developers. It also allows developers to work on a feature without affecting the main code base.

GitHub also offers users the ability to comment on each other's code, to review each other's code and raise issues on each other's repositories. This is often done through a process called a *pull request* (PR). A PR is a mechanism for a developer to notify team members that they have completed a feature. The team members review the changes, discuss potential improvements and eventually approve and merge the changes into the main code base. Figure 1.1 shows an example of a PR on GitHub.

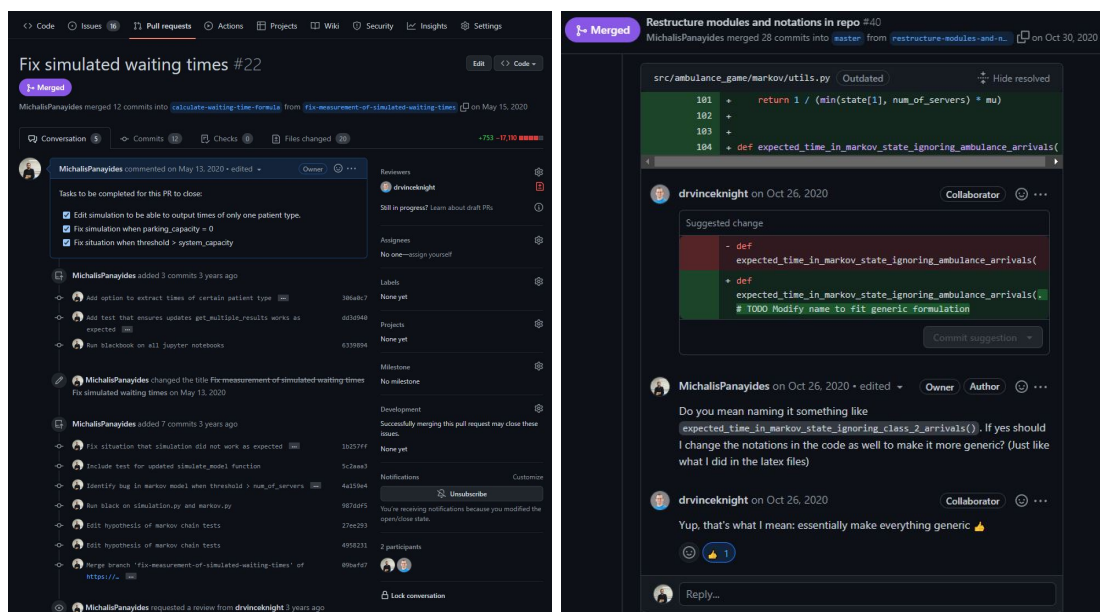


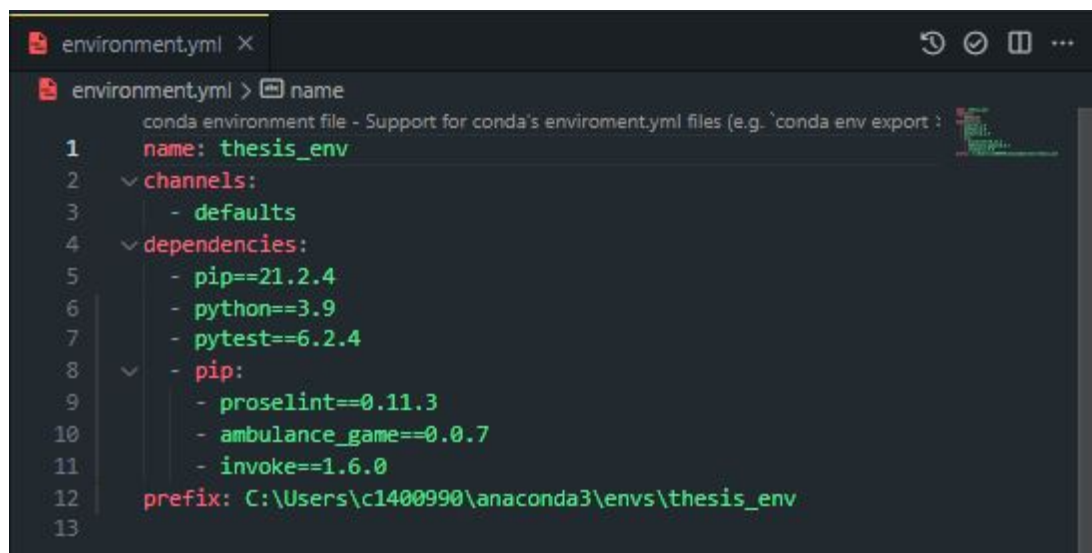
Figure 1.1: Pull request example on GitHub.

### 1.4.2 Virtual environments

A virtual environment is a tool that is used to create an isolated Python environment on a user's computer for a specific project. Virtual environments are used to isolate the dependencies of different projects and to ensure that the dependencies of one project do not interfere with the dependencies of another project.

There are several tools that can be used for creating virtual environments and managing dependencies. For the purposed of this thesis, the *Anaconda* package manager is used. Anaconda is an open source Python distribution platform that is used for scientific computing [123]. It offers a number of tools that are used in conjunction with Python. The most important of these tools is the *Conda* package manager. Conda is an open source package and environment management system that is used to install, update and manage packages and their dependencies. Conda was originally developed for Python programs but it can package and distribute software for any language.

For every repository that was created for this thesis, a *conda environment* was created. Every repository contains a `.yaml` file that contains the list of dependencies and the versions of the dependencies that are required to run the software. The `.yaml` file is recognised by Conda and is used to create the virtual environment. Figure 1.2 shows the contents of the `.yaml` file associated with the repository of this thesis.



```
environment.yaml
conda environment file - Support for conda's enviroment.yaml files (e.g. `conda env export`)
1  name: thesis_env
2  channels:
3    - defaults
4  dependencies:
5    - pip==21.2.4
6    - python==3.9
7    - pytest==6.2.4
8    - pip:
9      - proselint==0.11.3
10     - ambulance_game==0.0.7
11     - invoke==1.6.0
12  prefix: C:\Users\c1400990\anaconda3\envs\thesis_env
13
```

Figure 1.2: Conda environment example.

### 1.4.3 Summary of written software

The software written for this thesis is written in the open source programming language Python [148]. Python is a general-purpose, object-oriented programming language that is used for a wide range of applications. It is both beginner-friendly and powerful with a large community of developers and users.

For the purposes of this thesis a python library called `ambulance_game` was developed. The library contains a number of classes and functions used to formulate and solve the problem described in this thesis. More information about the installation and usage of the library can be found in Appendix A.

One of the most important, and often neglected, aspects of software development is documentation. Software documentation is a set of instructions that are used to explain how software works. Documentation should explain how users can install and use the software. All repositories associated with this thesis contain a `README.md` file that contains a brief description of the repository and instructions on how to clone the repository or install the software associated with it. The source code of the software has been written in a modular and readable manner with variable and function names that focus on readability. The source code of the software also contains docstrings that are used to explain the purpose of the functions and classes created. Docstrings are a form of documentation that is written directly in the source code.

### 1.4.4 Testing and code quality checkers

Code testing is a process that is used to ensure that the software works as expected. A *test* is a piece of code that is used to check the correctness of another piece of code. Tests are used to ensure that the software works as expected and that it produces the correct results.

“Testing is the process of executing a program with the intent of finding errors.” [102]

Throughout the development of the software associated with this thesis, a Test Driven Development (TDD) approach was used. TDD is a software development process where the tests are written before the code. This ensures that all the code is well tested and allows the programmer to change the program in small steps, increasing overall confidence in the program’s quality [9].

The Python code was tested using the *pytest* library. Pytest is a testing framework that is used to run tests. Figure 1.3 shows an example of a test written that

was written as part of the `ambulance_game` library.

```

38 @given(
39     threshold=integers(min_value=0, max_value=100),
40     system_capacity=integers(min_value=1, max_value=100),
41     buffer_capacity=integers(min_value=1, max_value=100),
42 )
43 def test_build_states(threshold, system_capacity, buffer_capacity):
44     """
45     Test to ensure that the build_states function returns the correct number of
46     states, for different integer values of the threshold, system and buffer capacities
47     """
48     states = build_states(
49         threshold=threshold,
50         system_capacity=system_capacity,
51         buffer_capacity=buffer_capacity,
52     )
53
54     if threshold > system_capacity:
55         assert len(states) == system_capacity + 1 # +2
56     else:
57         states_after_threshold = system_capacity - threshold + 1
58         size_of_s2 = states_after_threshold if states_after_threshold >= 0 else 0
59         all_states_size = size_of_s2 * (buffer_capacity + 1) + threshold
60         assert len(states) == all_states_size
61

```

Figure 1.3: Python test example.

Apart from testing the code, it is also important to ensure that the code is well written and that it follows a set of coding standards. This can be done using a set automated tools that are used to check the code for common errors and to ensure that the code follows a set of coding standards. Table 1.1 shows a list of tools that were used throughout the development of software and writing of this thesis.

Tool	Description
<code>black</code>	A code formatter that is used to ensure that the code is compliant with the PEP8 coding standard [149].
<code>flake8</code>	A tool for style guide enforcement.
<code>pylint</code>	A code linter that statically analyses your code.
<code>mypy</code>	A Python static type checker.
<code>aspell</code>	A grammar checker that catches spelling errors.
<code>alex</code>	A checker for insensitive and inconsiderate writing.
<code>proselint</code>	A linter for prose.

Table 1.1: List of tools used to check the code quality.

The tools listed in Table 1.1 were able to be automated using the `tox` library. Tox is an automation tool that aims to automate and standardise testing in Python.

It is a generic virtual environment management and test command line tool that can be used to check if a package installs correctly and run tests. It also acts as a frontend to Continuous Integration (CI) servers such as **GitHub Actions**. GitHub Actions is a service hosted on GitHub that is used to automate software development workflows. It is used to build, test and deploy software. GitHub Actions was used to automate the testing and code quality checks of the software associated with this thesis. Figure 1.4 shows an example of a GitHub Actions workflow that was used to automate the testing and code quality checks of the `ambulance_game` library.

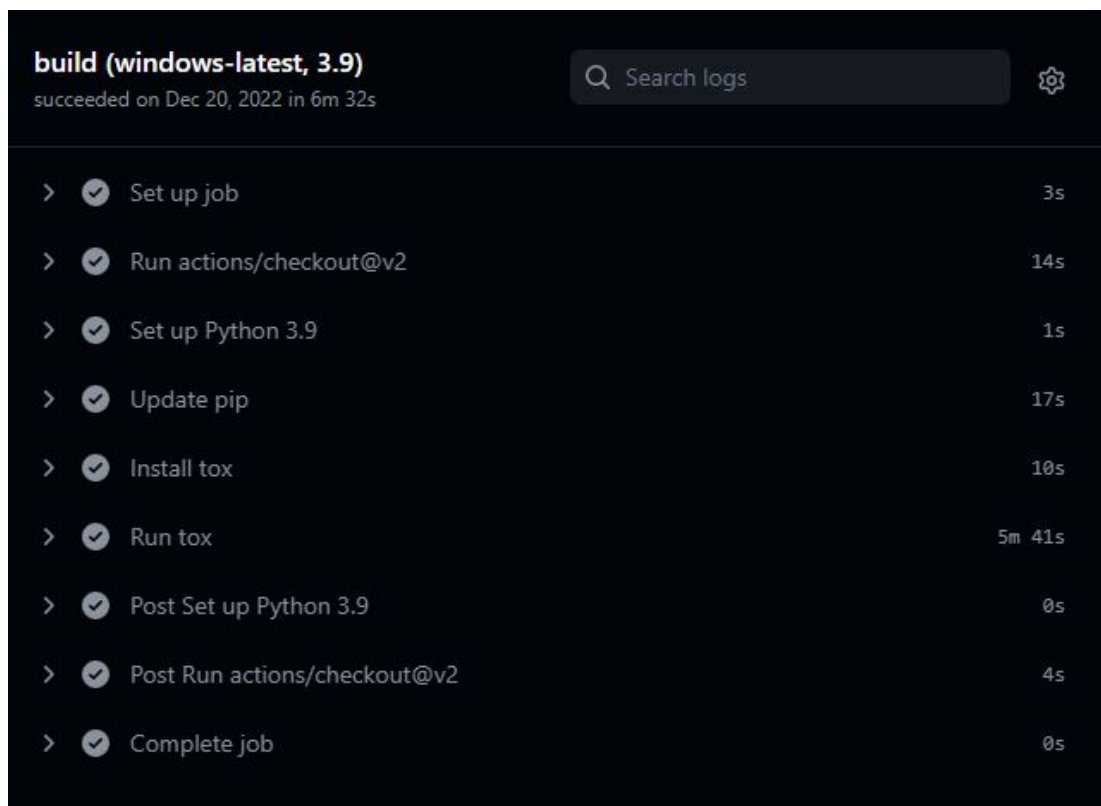


Figure 1.4: GitHub Actions workflow example.

## 1.5 Chapter summary

This chapter has given an introduction to OR and a brief overview of the history of OR along with a discussion on queuing theory and game theory. Additionally, the motivation for the research problem has also been discussed and the thesis objectives have been defined.

The chapter has also provided an overview of the best practices that were adopted throughout the development of the software associated with this thesis. Tools such as tox and GitHub Actions were used to automate the testing and code quality checks of the software.



# Chapter 2

## Literature review

### 2.1 Introduction

Chapter 1 introduced the problem of congestion in emergency departments (EDs) and the need for a model that can be used to understand the impact of different policies on the system. This chapter provides a review of the current relevant literature in the field of queueing theory and game theory as well as the combination of the two and their application to healthcare systems. This is achieved by partitioning the literature in four different sections each reviewing a different aspect of research. The literature review is structured in the following way:

- Section 2.2 provides a review of the literature of the techniques used in this thesis along with their application to healthcare systems. This includes Markov chain models and simulation models.
- Section 2.3 provides a review of the literature on the combination of game theory and queueing theory.
- Section 2.4 gives an overview of some examples of game theoretic models applied in healthcare systems.
- Section 2.5 discusses the general literature on behavioural modelling and some of its applications.

While the literature review is not exhaustive, it provides a good overview of the field of queueing theory and game theory as well as their application to healthcare systems. This literature review builds on the literature review provided in [114].

## 2.2 Operational Research models and Healthcare

This section aims to provide a review of Operational Research (OR) techniques and some applications to healthcare systems. OR is a discipline that consists of numerous mathematical tools and techniques that can be used to solve problems in a variety of fields. Some of these fields are healthcare, transportation, logistics, manufacturing, finance and many more.

Markov chains were originally developed by Russian mathematician Andrey Markov in 1906 who is known for their work in probability theory, analysis and number theory. They were originally developed to model the distribution of vowels and consonants in Pushkin's poem *Eugeny Onegin* [118]. Markov extended the weak law of large numbers and the central limit theorem to certain sequences of dependent random variables that were then known as Markov chains [14].

Markov chains are a mathematical tool that can be used to model a system and how it evolves over time. A markov chain is a stochastic model that consists of a set of states and a transition probability matrix that describes the probability of moving from one state to another. The following papers are examples of Markov chains being used in healthcare systems.

Even specifically in healthcare systems, Markov chains can be used to model a range of different scenarios. For example in [98] a Markov reward model is developed for a healthcare system to model the movement of patients between hospital states where patients arrive at a constant rate. An additional Markov model is also developed to determine patient numbers and costs at any time where arrivals are taken from a waiting list and a fixed growth of arrivals that is slowly declining to zero is introduced in the waiting list. The model is then applied to geriatric patients to determine costs over time. In [63] a Markov chain model is created to analyse the elderly people flow in the French Healthcare system (FHS) and model their pathway in hospital. The model is then applied to a French Hospital to understand the dynamics of elderly patients flow. The authors in [56] present a Markov-chain model to analyse the progression of opioid addiction in order to develop treatments. The Markov chain model is used to predict the proportion of patients in a given stage of intervention. In addition, in [91] a Markov Decision Process (MDP) framework on Discrete time Markov chain (DTMC) is developed to optimise medical equipment repair and replacement decisions. The model is used to determine the optimal repair and replacement decision based on the product life cycle and status. The authors in [91] also use a

dataset of 24,516 repair and maintenance records that reveal the most common reasons of fault and the most economically viable repair options. In [151] the optimal staffing problem is addressed for a non-preemptive priority queue with two customer classes and a time-dependent arrival rate. The authors use mixed discrete-continuous time Markov chains (MDCTMCs) to evaluate the behaviour of the system and generate the minimum staffing levels required. The applications of interest here were systems where the customers can be categorised into priority classes, such as emergency departments and call centres.

Another queueing theory technique that has been traditionally utilised to represent healthcare systems is Discrete Event Simulation (DES). DES is a technique that is used to model the behaviour of a system by representing it as a set of events that occur over time. More details on DES can be found in Section 3.2.

The authors of [132] compare Markov modelling with DES to assess if any of the two may change some healthcare resource allocation decisions. The authors compare the two approaches in a systematic review and state that DES is suitable for modelling systems with limited resources and are able to better capture complexity and uncertainty in the system along with the ability to capture individual patient histories. On the other hand, some disadvantages of DES over Markov modelling is that it is computationally more expensive, requires more data and is more difficult to validate. The authors conclude that DES may be preferred over Markov modelling when individual patient history is an important driver of future events. In [156] the authors use a discrete event simulation model to determine the optimal number of critical care beds required for a hospital. Discrete event simulation is utilised to help resource planning and simulate different what-if scenarios. The authors in [42] use a discrete event simulation model to model the emergency department of a hospital in Canada. The aim of the paper was to reduce patient waiting times, improve the overall service delivery and system throughput. Additionally, the authors in [53] use a discrete event simulation model to predict the progression of Alzheimer's disease through correlated changes in cognition, behavioural disturbance and function. Individuals in the models are assigned unique demographic and clinical characteristics and were the severity of the disease was tracked throughout. The simulation results suggest that donepezil leads to health benefits and cost savings for patients with mild to moderate Alzheimer's disease and is even more beneficial when patients are in the mild stages of the disease.

## 2.3 Game theory and Queueing theory

A number of papers have been published that touch upon the use of queueing models together with game theoretic concepts. In [28] the authors study a simultaneous price competition between two firms that are modelled as two distinct queueing systems with a fixed capacity and a combined arrival rate. They calculate the Nash equilibrium both for identical and heterogeneous firms and show that for the former a pure Nash equilibrium always exist and for the latter a unique equilibrium exists where only one firm operates. The authors have also extended their model in [29] by allowing the players (firms) to choose capacities. A main result from this paper was that when both firms operate independently as a monopoly, the equilibria are socially optimal, but this is not the case when the firms operate together. Another extension of [28] was introduced in [31] where a long-run version of the competition was considered that also had capacity as a decision variable. An additional paper that focuses on competition is [45] where the authors created a competition between two sellers where seller 1 supplies a product instantly and seller 2 is modelled as a make-to-order M/M/1 queue. The game that is played requires the two sellers to make a choice on the price of the product and then seller 2 to set a capacity that guarantees a maximum expected delay. In our work, while giving some consideration to equilibrium behaviour, similar to the work of [28, 29], emergent behaviour is more precisely addressed by considering learning algorithms like asymmetric replicator dynamics [52]. More details on learning algorithms can be found in Section 4.2.3.

Another specific part of our research, as described later in the thesis, is the construction of a queueing system with a tandem buffer and a single service centre. There are several examples from literature that touch upon queueing models with tandem queues. In [37] the authors explore threshold joining strategies in a Markov model that has two tandem queues. Another example is the one described in [25] where they investigated a network of multiple tandem queues where customers decide which queue to attend before joining. Similarly, in [13] the authors examine a network of  $N$  tandem M/M/1 queues and with multi-type customers. The customers in this paper react to a price  $p$  by picking demand rates that maximise utility. In [150] a profit maximisation problem is studied that has two servers; an M/M/1 queue and a parking service providing complementary service while the customer is in the first service. The providers gain a reward when customers complete both services and no reward when they finish one of them. One of the main conclusions of this study is that by increasing the general demand both providers lower their prices to compensate for the increase in wait. The problem was later extended by [135] where they considered arrivals of batches

that can share the parking service. Finally, [2] examines a tandem network of two  $M/M/1$  queues that are ran by two different profit-maximising service providers. The network receives three types of customers; those requiring both services, customers requiring the first service and customers requiring the second service. The authors showed that optimal prices also maximise social utility and that removing two types of customers that don't need both services leads to higher profit and lower demand rate. In our work, the concepts described in [13, 25, 37] are extended by introducing a threshold parameter that determines when individuals can progress from one queue to the other.

Additionally, in [60] the authors explore combining queueing theory, agent-based simulation and game theory to study the impact of ambulance diversion. They consider overcrowded emergency departments (ED) and the use of ambulance diversion (AD) during which a hospital is not accepting patients by ambulance. The formulated games are analysed to explore the potential of cooperation in this setting. The authors conclude that in such a setting cooperation is not something that emerges naturally in the presence of strategic behaviour and propose a centralised form of ambulance routing. The lack of cooperation in healthcare settings is something that will be further explored in the work of this thesis. In particular Section 4 will describe a game that is formulated to study the impact of ambulance blockage outside the emergency department. In [26] the authors study a queueing model in which two strategic servers may choose their own capacities and service rates where the faster a server works, the more cost it incurs. The buyer chooses to allocate demand based on the performance of the servers where faster servers are allocated more demand. The authors investigate the trade-off between efficiency and incentives and find that it is possible to design an allocation policy that is both efficient and can also incentivise the servers to work quickly. This paper shares some similarities with the work of this thesis. In particular, this thesis formulates a game to investigate the trade-off between ambulance blockage and overall efficiency of the ED which is a similar concept to the one described in [26]. In [72] the authors study the behaviour of vendors in competition. Similar to the work of [26], the authors consider a queueing model where servers choose their own service rates at a cost. The servers are also rewarded for each customer that they serve and based on that cost a two-player game between the two servers is formulated.

## 2.4 Game theory in Healthcare

In this section a review of the literature is provided that is relevant to game theoretic models used in healthcare systems. Game theory is a mathematical tool that is used to model strategic interactions between players in a system. The players are assumed to be rational and make decisions based on their own self-interest and the information they have about the other players' decisions and the system's state. A more formal definition of the game theory concepts used in this thesis is given in Section 4.2.

In the above models, the players are attempting to increase their share of individuals choosing to queue. In public healthcare type settings, this is not necessarily the case. Rational usage of public services will not necessarily lead to a socially optimal outcome. Rather, the overall service needs to be considered as players aim to minimise their experienced congestion. In [125] a healthcare application was studied where patients could choose between two hospitals, where a utility function is derived that is based on patients' perceived quality of life. In [79] the authors place the individuals' choices between different public services within the formulation of routing games and measure inefficiencies using a concept known as the price of anarchy (PoA) [82]. They show that the price of anarchy increases with worth of service and that is low for systems with insufficient capacities. In [30] a two-tier healthcare system with a capacity constrained is studied where patients can choose between two systems to receive their service. The first system is labelled as the free system (public government-funded hospital) which offers service without seeking any profit and the second one is the toll system (private hospital) that aims to maximise its own profit. The authors, also compare the two-tier system with its one-tier equivalent, where only the free system exists. In [77] a normal form game is built that is informed by a two-dimensional Markov chain in order to model interactions between critical care units. In [154] a queueing-game-theoretical model is introduced where there are two types of service providers; a high quality high-congested hospital and a low quality low-congested hospital. The authors study a two-stage Stackelberg game where the government is the *leader* and the arriving patients are the *followers*. In [40] the authors study the network effect of ambulance diversion by proposing a non-cooperative game between two EDs that are modelled as a queueing network. Each ED's objective is to minimise its own waiting time and chooses a diversion threshold based on the patients it has. In equilibrium both EDs choose to divert ambulances in order to avoid getting arrivals from the other ED. In this thesis this concept is extended by allowing the ambulance service to decide how to distribute its patients among the two EDs. The players of the game are both the

hospitals and the customers of the hospitals, as opposed to the previous models which are one or the other. Thus, the novelty of our work is combining both these aspects.

## 2.5 Behavioural Operational Research

Behavioural OR seeks to (i) advance our understanding of how behavioural factors affect the conduct of, and interact with, model-based processes that support problem solving and decision making [85], and (ii) to leverage this understanding for improving outcomes [51, 62]. Moreover, behavioural OR is a sensitive discipline and is subject to the individual studying it, which means that it is interpreted differently from researchers to researcher [59].

### 2.5.1 Agent-based modelling

When a modeller looks into implementing behaviour into a certain model, agent-based modelling is one of the most commonly used techniques. Agent-based modelling is a micro-abstracted modelling technique that is capable of modelling complex systems that are composed of many interacting agents [57, 69]. It is also capable of dealing with both deterministic and stochastic problems.

In such models each simulated item is considered an agent. An agent represents a person within a group of interest and is modelled in such a way that it has its own perception of the system and the environment in general. Therefore, these autonomous agents may be able to make decisions based on their own memory and experience. Additionally they can, not only interact with the environment, but they can also interact among themselves individually (or with the whole population) and make certain choices based on these interactions. Essentially in every time step all agents observe the environment and choose to move based on that observation. Some traditional examples of agent-based models are segregation models, predator-prey models, and forest fire models [69].

### 2.5.2 Agent-based modelling and healthcare

There are numerous examples of simulation models that have been used in literature to model healthcare systems but only a few of them are agent-based models. As stated in [44] healthcare systems are based on human interactions which is what makes them so complex. Agent-based simulation modelling is capable of capturing both human intention and human interaction which is why it makes it one of the most suitable candidate for modelling such healthcare systems. This

section provides a brief overview of the literature around agent-based models applied to healthcare.

The authors of [60], that was also mentioned in Section 2.3, also make use of agent-based modelling to model the setting of ambulance diversion. They used an agent-based model to provide insights into how spatial structure, number of hospitals and different policies contribute to cooperation in avoiding ambulance diversion. In essence, the agent-based model was used to assess whether partial diversion of ambulances would be a viable option for the ambulance service where it was shown at the end that it was not. Furthermore, the authors of [54] also make use of agent-based modelling to model the different queuing strategies in the youth health care setting. The agent-based simulation model is parameterised with actual market data and different queuing strategies are investigated. The model is structured to incorporate additional complexities that a queueing theoretic model would fail to capture and thus, the model is able to provide insights on the queueing strategy decision that would not be possible otherwise.

In [131] the authors make use of agent-based simulation to model the setting of hospital emergency departments. The paper describes the system analysis and the preliminary model that was developed for the simulation these emergency departments along with the advantages of using agent-based modelling. The authors claim that agent-based models could be a better fit to model situations where the human aspect is important. The human element is part of the system and thus, it should be included in the model to describe the fundamental concepts of the system. The authors of [18] present two ongoing projects of agent-based models that are applied to the healthcare setting. The first discussed project is an agent-based simulation model that is used to model the long term monitoring of Chronic Obstructive Pulmonary Disease (COPD) which is a major public health problem. The second project is applying agent-based simulation to visualise and explore informal social networks amongst staff at the Akdeniz University Hospital.

### 2.5.3 Reinforcement learning

Reinforcement learning is a machine learning technique that is used to train agents to make decisions in an uncertain environment. Reinforcement learning is formulated in such a way that the agent may choose between a set of policies and aims to maximise the expected reward. The agent is able to learn from its own experience and is able to make decisions based on the reward it receives. There are numerous applications of reinforcement learning in the literature and it is used in many different fields such as robotics, finance, and gaming [136]. This



Section provides a brief overview of the literature around reinforcement learning.

In [68] the authors combine agent-based modelling with reinforcement learning to model the setting of price negotiations. The authors study the ability of agents to perform price negotiations and propose a new model that is based on the ability of agents to distinguish between different words. The words correspond to the agent's demand and the agents use these words to negotiate. A reinforcement learning algorithm is used to train the agents to distinguish between the words and use them to negotiate. At the end it is shown that these words become meaningful in the process of negotiations and certain strategies are learned by the reinforcement learning algorithm. In [161] the authors combine agent-based modelling with reinforcement learning to model the electricity market. An agent-based simulation model is developed to compare market characteristics of different pricing methods. The authors use reinforcement learning to train the generators to improve their bidding strategies in a repeated bidding game where generators aim to maximise their profit. The authors in [141] also make use of reinforcement learning in the power market setting. Specifically, they use Q-learning to train the suppliers' bidding strategies based on maximising their profit and their utilisation rate. Through Q-learning's exploitation and exploration trade-off the suppliers are able to learn the most profitable action to take under different market conditions. At the end of the paper the authors discuss the outcomes of four test cases with three suppliers under different demand values.

In [140] the authors compare three different learning mechanisms in a multi-agent based simulation and analyse the results in a bargaining game. The authors used reinforcement learning to validate and verify the results of the simulation. The results show that the learning mechanisms that enable agents to acquire their rational behaviours differ according to the knowledge representation of the agents. A similar idea is extended in [138] where a multi-agent based simulation is introduced to explore agents who can reproduce human-like behaviours in the repeated bargaining game. The authors compare the results of Roth's learning agents and Q-learning agents in the sequential bargaining game. The authors conclude that reinforcement learning agents cannot learn consistent behaviours in the repeated bargaining game while Q-learning agents can learn such behaviour but cannot reproduce the decreasing trend found in subject experiments. The concepts from [138, 140] are extended in [139] where the authors explore agents that can reproduce human-like behaviours and human-like thinking in the sequential bargaining game. The authors compare the results of Q-learning agents with different action selection mechanisms and conclude that only Q-learning agents with Boltzmann distribution selection can reproduce both human-like behaviours

and thinking.

Although agent-based modelling has been somewhat used in the literature to model healthcare systems, this has not been the case for reinforcement learning. In this thesis reinforcement learning is used to train the servers of a queueing system to pick their service rates in order to maximise their utility.

## 2.6 Chapter summary

This chapter has provided an overview of the literature around OR in healthcare systems. More specifically, Section 2.2 provides a review of the literature of the techniques used in this thesis along with their application to healthcare systems. In particular the main techniques that has been reviewed are Markov modelling and discrete event simulation. Chapter 3 of this thesis introduces a novel queueing network that is modelled using both discrete event simulation and Markov chains. In Section 3.6 the proposed model is applied to a healthcare scenario to model the setting of an emergency department that receives patients from an ambulance service.

In Sections 2.3 and 2.4 the literature around the combination of queueing theory and game theory and the literature around game theoretic techniques applied in healthcare is reviewed. Chapter 4 of this thesis introduces a novel game theoretic model that is applied to the queueing network model introduced in Section 3. The game theoretic model is used to model the scenario where two queueing systems and a patient distribution service compete in a 3-player game to maximise their own utilities. Subsequently, Section 4.5 describes how the game theoretic model is applied to the healthcare scenario where the players are an ambulance service, and two emergency departments.

Section 2.5 of this chapter provides a brief overview of the literature around behavioural modelling. One of the main techniques that has been reviewed is agent-based modelling and reinforcement learning. These two techniques are used in Chapter 6.5 to extend the model introduced in Section 3 and observe the model from a more behavioural perspective.

## Chapter 3

# Queueing theoretic model

### 3.1 Introduction

One of the main outcomes of this research is the creation of a queueing network model that consists of two queueing nodes and accepts two types of individuals.

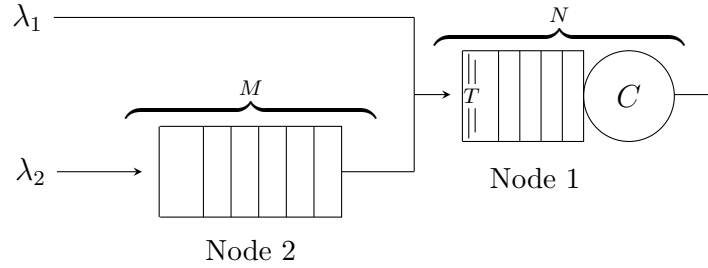


Figure 3.1: A diagrammatic representation of the queueing network. The threshold  $T$  only applies to type 2 individuals. If the number of individuals in node 1 is greater than or equal to  $T$ , only individuals of type 1 are accepted (at a rate  $\lambda_1$ ) and individuals of type 2 (arriving at a rate  $\lambda_2$ ) are blocked in node 2.

The model consists of two types of individuals; type 1 and type 2. Type 1 individuals arrive instantly at node 1 and wait to receive their service. Type 2 individuals arrive at node 2 and wait there until they are allowed to move to node 1. They are allowed to proceed only when the number of individuals in node 1 **and** in service is less than a pre-determined threshold  $T$ . When the number of individuals is equal to or exceeds this threshold, all type 2 individuals that arrive will stay *blocked* in node 2 until the number of people in node 1 falls below  $T$ . This is shown diagrammatically in Figure 3.1. The parameters of the described queueing model are:

- $\lambda_i$ : The arrival rate of type  $i$  individuals where  $i \in \{1, 2\}$

- $\mu$ : The service rate for individuals receiving service at node 1
- $C$ : The number of servers
- $T$ : The threshold at which individuals of the second type are blocked
- $N$ : The capacity of node 1 (i.e  $N = C + (\text{Queue capacity})$ )
- $M$ : The capacity of node 2

Note that the parameters  $N$  and  $M$  are defined as the capacities of nodes 1 and 2 respectively. In the case where these capacities are infinite, the queueing network can only be modelled using the Discrete Event Simulation (DES) approach. The Markov chain approach can only use  $N$  and  $M$  as artificial truncation parameters to approximate the behaviour of the system. In addition, when type 1 individuals arrive at node 1 and it is at full capacity, they become lost from the system. Similarly, when type 2 individuals arrive at node 2 and it is at full capacity, they are also lost from the system.

In Section 3.6 this queueing network will be used to model the structure of an Emergency Department (ED) that receives patients in ambulances from the Emergency Medical Services (EMS). This chapter extends the concepts described in [114] and consists of the following sections:

- Section 3.2 gives an overview of the discrete event simulation model.
- Section 3.3 gives an overview of the Markov chain model.
- Section 3.4 describes the Markov chain model is used to extract performance measures of the system.
- Section 3.5 presents some numerical results and timings experiments for the queueing network, along with a comparison between the DES and Markov chain approach.
- Section 3.6 describes how the queueing network can be used to model the emergent behaviour between EDs and the EMS.

## 3.2 Discrete Event Simulation

Discrete Event Simulation (DES) is a method for modelling the behaviour of real-world systems in which the system is made up of discrete events, each of which has a certain duration [122]. It can be used to understand complex situations in order to make predictions and thus provide improvements [78]. The three

main approaches to building DES models are the activity scanning approach, the event scheduling approach and the process interaction approach [95]. Under the scope of this study only the event scheduling approach is considered. This section describes the DES model used to represent the queueing network of Section 3.

In order to use DES on the queueing network shown in Figure 3.1 an equivalent queueing network must be constructed. The current queueing network is a two-node queueing system that accepts two types of individuals, where type 1 individuals arrive at node 1 and type 2 individuals arrive at node 2. The modification that is required revolves around the mechanisms of node 2. Node 2 is defined as a non-service node where there is only a queueing space for individuals to wait there until they are allowed to node 1. From an implementation perspective there is an equivalent system that can be used where instead of a node with no service and queueing capacity  $M$ , there are  $M$  servers each serving with a deterministic service time of 0 and no queueing capacity, as shown in Figure 3.2.

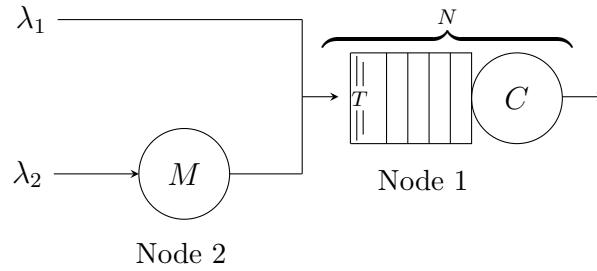


Figure 3.2: An equivalent model to the one described in Figure 3.1. The difference between the two diagrams is the formulation of node 2. The original diagram uses a node with no servers and a queueing capacity of  $M$  while this one uses  $M$  servers with no queueing capacity.

The arrival times for both nodes are exponentially distributed with mean  $\lambda_1$  and  $\lambda_2$  corresponding to type 1 and type 2 individuals respectively. Node 1 has an exponentially distributed service time with rate  $\mu$ , a total of  $C$  servers and a queueing capacity of  $N - C$  (making the overall capacity  $N$ ). Node 2 has a deterministic service time of 0, a total of  $M$  servers and a queueing capacity of 0. Note here that, similar to Figure 3.1, parameters  $N$  and  $M$  are used to approximate the real world system and in fact can be taken to be infinite in the DES. Finally the routing parameter is defined as an array that probabilistically routes individuals from all nodes to all other nodes. For this particular system the routing parameter needs only to route individuals from node 2 to node 1.

$$\text{routing parameter} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \quad (3.1)$$

The routing parameter is a matrix that represents how nodes are connected to each other. For example the element in the first row and second column of the routing parameter corresponds to the probability of an individual leaving the first node and arriving at the second node. In addition, when a row of the routing parameter contains only zeros, the equivalent node routes individuals to the exit of the system. Note that, in the particular routing parameter the nodes are reversed because in the implementation node 2 is assigned id 1 and node 1 is assigned id 2.

### 3.2.1 Implementation

The python library `ciw` [110, 142] was used to implement the DES model. The library treats queues as distinct nodes in the network where each node has an arrival distribution, a service distribution, a number of available servers and a queue capacity.

The following code can be used to generate a queueing network with two queues and two types of individuals, where type 1 individuals arrive at node 1 with an arrival rate of `lambda_1` and type 2 individuals arrive at node 2 with an arrival rate of `lambda_2`. Node 2 has a deterministic fixed service rate of 0 (since there is no service involved in the buffer centre) and node 1 has an exponential service rate of `mu`.

```
>>> import ciw
>>> lambda_1 = 1.0
>>> lambda_2 = 2.0
>>> mu = 0.5
>>> num_of_servers = 3
>>> system_capacity = 10
>>> buffer_capacity = 5
>>> model = ciw.create_network(
...     arrival_distributions=[
...         ciw.dists.Exponential(lambda_2),
...         ciw.dists.Exponential(lambda_1)
...     ],
...     service_distributions=[
...         ciw.dists.Deterministic(0), ciw.dists.Exponential(mu)
...     ],
...     routing=[
...         [0.0, 1.0],
...         [0.0, 0.0]
...     ],
...     number_of_servers=[buffer_capacity, num_of_servers],
...     queue_capacities=[0, system_capacity - num_of_servers],
... )
```

Code snippet 3.1: Using the `ciw` library to create a queueing network with two queues and two types of individuals and the particular structure defined in Figure 3.2.

As described earlier in Section 3 and as shown in Figure 3.1, type 1 individuals arrive at node 1 and exit the system after their service finishes, but type 2 individuals arrive at node 2 and then proceed to node 1 after leaving node 2. This logic is implemented in the queueing network using the `routing` parameter that consists of the routing probabilities between different nodes. For the current implementation the routing matrix is a  $2 \times 2$  array that routes individuals from node 2 to node 1 with a probability of 1.0. Furthermore, the server availability for nodes 1 and 2 are set to the `num_of_servers` and `buffer_capacity` respectively and the queue capacities are set to `system_capacity - num_of_servers` and 0. Note that for node 2 queue capacity is set to 0 and its number of servers is set to the buffer capacity. From `ciw`'s data records perspective this made more sense since individuals are recorded as blocked this way. If the queue capacity was non-zero, individuals could also have a waiting time but no waiting should take place in node 2, only blockage.

### 3.2.2 Custom node class

Another specific feature of the particular model is that type 2 individuals need to stay blocked in node 2 whenever the number of individuals in node 1 reaches a certain threshold  $T$ . As opposed to the rest of the model, this portion of the queueing network requires more effort to build. `Ciw` allows users to get more custom behaviour by creating their own node class that inherits from the original one. By inheriting the original `ciw.Node` the general behaviour of all nodes can be altered. Note that node ids are assigned in the order they are created so in the `ciw` implementation node 2 is assigned id 1 and node 1 is assigned id 2.

```
>>> import numpy as np
>>> def build_custom_node(threshold=float("inf")):
...     """
...     Build a custom node to replace the default ciw.Node. Inherits from
...     the original ciw.Node class and replaces methods
...     release_blocked_individual and finish_service.
...     The methods are modified in such a way that all individuals that
...     are in the buffer space (node 1) stay blocked as long as the
...     number of individuals in the service area node (node 2) exceeds the
...     threshold.
...
...     Parameters
...     -----
...     threshold : int, optional
...         The capacity threshold to be used by the method
...     Returns
...     -----
...     class
...         A custom node class that inherits from ciw.Node
...     """
...
...     class CustomNode(ciw.Node):
```

```

...     """
...     Overrides the default release_blocked_individual and
...     finish_service methods of the ciw.Node class
...     """
...
...     def __init__(self, id_, simulation):
...         """
...         Initializes the node with the given id and simulation using
...         the initialisation of ciw's Node object with the addition of
...         the threshold parameter.
...         """
...         super().__init__(id_, simulation)
...         self.simulation.threshold = threshold
...
...     def release_blocked_individual(self):
...         """
...         Releases an individual who becomes unblocked when
...         another individual is released:
...         - check if individual in node 2 and should stay blocked
...           i.e. if the number of individuals in that
...             node > threshold
...         - check if anyone is blocked by this node
...         - find the individual who has been blocked the longest
...         - remove that individual from blocked queue
...         - check if that individual had their service interrupted
...         - release that individual from their node
...         """
...         continue_blockage = (
...             self.number_of_individuals >= threshold
...             and self.id_number == 2
...         )
...         if (
...             self.len_blocked_queue > 0
...             and self.number_of_individuals < self.node_capacity
...             and not continue_blockage
...         ):
...             receiving_node = (
...                 self.simulation.nodes[self.blocked_queue[0][0]]
...             )
...             individual_to_receive_index = [
...                 ind.id_number
...                 for ind in receiving_node.all_individuals
...             ].index(self.blocked_queue[0][1])
...             individual_to_receive = (
...                 receiving_node.all_individuals[
...                     individual_to_receive_index
...                 ]
...             )
...             self.blocked_queue.pop(0)
...             self.len_blocked_queue -= 1
...             if individual_to_receive.interrupted:
...                 individual_to_receive.interrupted = False
...                 receiving_node.interrupted_individuals.remove(
...                     individual_to_receive
...                 )
...                 receiving_node.number_interrupted_individuals -= 1
...                 receiving_node.release(individual_to_receive_index, self)
...
...     def finish_service(self):

```



```

...         """
...         The next individual finishes service:
...         - finds the individual to finish service
...         - check if they need to change class
...         - find their next node
...         - release the individual if there is capacity at destination,
...           otherwise cause blockage
...         - Note that blockage also occurs when we are at node 1 and
...           the number of individuals on node 2 are more than the
...           'threshold'
...         """
...         (
...             next_individual,
...             next_individual_index
...         ) = self.find_next_individual()
...         self.change_customer_class(next_individual)
...         next_node = self.next_node(next_individual)
...         next_individual.destination = next_node.id_number
...         if not np.isinf(self.c):
...             next_individual.server.next_end_service_date=float("Inf")
...         blockage = (
...             next_node.number_of_individuals >= threshold
...             and self.id_number == 1
...         )
...         if (
...             next_node.number_of_individuals < next_node.node_capacity
...         ) and not blockage:
...             self.release(next_individual_index, next_node)
...         else:
...             self.block_individual(next_individual, next_node)
...
...         return CustomNode

```

Code snippet 3.2: Function that contains the custom node class that blocks individuals to node 1 when the number of individuals in node 2 exceeds the threshold.

The class `CustomNode` inherits from `ciw's Node` class and changes two of the methods (`release_blocked_individual` and `finish_service`) so that the additional logic of the threshold is incorporated. In the `release_blocked_individual` method an additional check is added before releasing a potentially blocked individual from node 2 to node 1. This essentially checks whether the id number of the node is 2 and the number of individuals in it are more than or equal to the `threshold` so that it can accept a blocked individual. Similarly `finish_service` is called once an individual finishes their service. The additional check that was added checks whether the id of the node is 1 and the number of individuals in the next node (i.e. node 1) is more than the threshold, which would result in blockage. Finally, the simulation object can be created and simulated for a specific `threshold` and `runtime` by running:

```

>>> threshold = 4
>>> runtime = 1000

```

```
>>> custom_node = build_custom_node(threshold)
>>> ciw.seed(0)
>>> simulation = ciw.Simulation(model, node_class=custom_node)
>>> simulation.simulate_until_max_time(runtime)
```

Code snippet 3.3: Building and simulating the network model using the custom node

### 3.2.3 Performance Measures

Having run the simulation using `ciw` all necessary performance measures can be calculated. Calculating all performance measure that are related to the duration of time is not too difficult. The code snippet in 3.4 gets all waiting times, service times and blocking times for all individuals that have passed through the model.

```
>>> def extract_times_from_records(simulation_records, warm_up_time):
...     """Get the required times (waiting, service, blocking) out of ciw's
...     records where all individuals are treated the same way. This function
...     can't distinguish between class 1 and class 2 individuals. It returns
...     the aggregated waiting time, service times BUT only blocking times of
...     class 2 individuals.
...     """
...     waiting_times = [
...         r.waiting_time
...         for r in simulation_records
...         if r.arrival_date > warm_up_time and r.node == 2
...     ]
...     serving_times = [
...         r.service_time
...         for r in simulation_records
...         if r.arrival_date > warm_up_time and r.node == 2
...     ]
...     blocking_times = [
...         r.time_blocked
...         for r in simulation_records
...         if r.arrival_date > warm_up_time and r.node == 1
...     ]
...     return waiting_times, serving_times, blocking_times
```

Code snippet 3.4: Function that extracts performance measures from the simulation records

Using the earlier variable `simulation`, the waiting times, service times and blocking times can be extracted from the simulation records.

```
>>> warm_up_time = 100
>>> all_records = simulation.get_all_records()
>>> waiting_times, serving_times, blocking_times = extract_times_from_records(
...     all_records, warm_up_time
... )
>>> np.mean(waiting_times), np.mean(serving_times), np.mean(blocking_times)
(1.7038750111337655, 2.041619227158985, 8.227702587974997)
```

Code snippet 3.5: Using the function defined in 3.4

The performance measures discussed so far are the ones that are related to the amount of time that individuals spend in the system. Some additional performance measures that could be of interest are the ones that are related to the number of individuals, such as the mean number of individuals in the system, the mean number of individuals in Node 1 and the mean number of individuals in Node 2. These metrics can also be calculated using an additional functionality of `ciw`. There is an additional argument that can be passed to the `ciw.Simulation` object called `tracker` that makes the simulation object track the state probabilities of the system throughout the simulation. Thus, at the end of the simulation, the probability distribution of the number of individuals in each node can be extracted. Section 3.4 gives a more detailed explanation of how to calculate these performance measures using only the state probabilities of the system.

### 3.3 Markov chain model

A Markov chain is a stochastic model that is the primary analytical tool to study queues. Under the assumption that all rates (arrival and service) are Markovian the queueing system can be represented by a Markov chain model [74]. The states of the Markov chain are denoted by  $(u, v)$  where:

- $u$  is the number of individuals blocked in node 2
- $v$  is the number of individuals either in node 1 or in the service centre

The set of all possible combination of pairs  $(u, v)$  form all the possible states that the system can visit. The state space of the Markov chain is denoted as the set  $S = S(T)$  which can be written as the disjoint union:

$$\begin{aligned} S(T) &= S_1(T) \cup S_2(T) \text{ where:} \\ S_1(T) &= \{(0, v) \in \mathbb{N}_0^2 \mid v < T\} \\ S_2(T) &= \{(u, v) \in \mathbb{N}_0^2 \mid v \geq T\} \end{aligned} \tag{3.2}$$

$S_1$  consists of the set of states where the number of individuals in node 1 is less than  $T$  (i.e.  $v < T$ ) and subsequently the number of individuals in node 2 is zero (i.e.  $u = 0$ ). Similarly,  $S_2$  consists of the set of states where the number of individuals in node 1 is greater than or equal to  $T$  (i.e.  $v \geq T$ ) and hence it is possible for individuals to be at node 2 (i.e.  $u \geq 0$ ). This is illustrated diagrammatically in Figure 3.3.

Having defined the set of states of the Markov chain model, the generator matrix can also be obtained. The generator matrix  $Q$  of the Markov chain consists of the rates between the numerous states of the model. Every entry  $Q_{ij} = Q_{(u_i, v_i), (u_j, v_j)}$  represents the rate from state  $i = (u_i, v_i)$  to state  $j = (u_j, v_j)$  for all  $(u_i, v_i), (u_j, v_j) \in S$ . The entries of  $Q$  can be calculated using the state-mapping function described in equation (3.3). Here  $\Lambda$  denotes the overall arrival rate in the model for both types of individuals (i.e.  $\Lambda = \lambda_1 + \lambda_2$ ).

$$Q_{ij} = \begin{cases} \Lambda, & \text{if } (u_i, v_i) - (u_j, v_j) = (0, -1) \text{ and } v_i < T \\ \lambda_1, & \text{if } (u_i, v_i) - (u_j, v_j) = (0, -1) \text{ and } v_i \geq T \\ \lambda_2, & \text{if } (u_i, v_i) - (u_j, v_j) = (-1, 0) \\ v_i \mu, & \text{if } (u_i, v_i) - (u_j, v_j) = (0, 1) \text{ and } v_i \leq C \text{ or} \\ & (u_i, v_i) - (u_j, v_j) = (1, 0) \text{ and } v_i = T \leq C \\ C \mu, & \text{if } (u_i, v_i) - (u_j, v_j) = (0, 1) \text{ and } v_i > C \text{ or} \\ & (u_i, v_i) - (u_j, v_j) = (1, 0) \text{ and } v_i = T > C \\ -\sum_{j=1}^{|Q|} Q_{ij} & \text{if } i = j \\ 0, & \text{otherwise} \end{cases} \quad (3.3)$$

Note that for large values of  $N$  and  $M$  most of the entries of the transition matrix will be zero. In order to speed up the computation of the transition matrix, instead of considering every possible pair of states in the state space a new function that maps a state to every possible destination state can be used. Function  $\mathcal{M}$  from equation (3.4) takes a state  $(u, v)$  and maps it to the set of all possible destination states that the system can go to when on that state.

$$\mathcal{M}(u, v) = \begin{cases} \{(u, v+1), (u, v-1)\} & \text{if } v < T \\ \{(u+1, v), (u, v+1), (u, v-1)\} & \text{if } v = T \text{ and } u = 0 \\ \{(u+1, v), (u, v+1), (u-1, v)\} & \text{if } v = T \text{ and } u > 0 \\ \{(u, v+1), (u+1, v), (u, v-1)\} & \text{if } v > T \end{cases} \quad (3.4)$$

A visualisation of how the transition rates relate to the states of the model can be seen in the general Markov chain model shown in Figure 3.3.

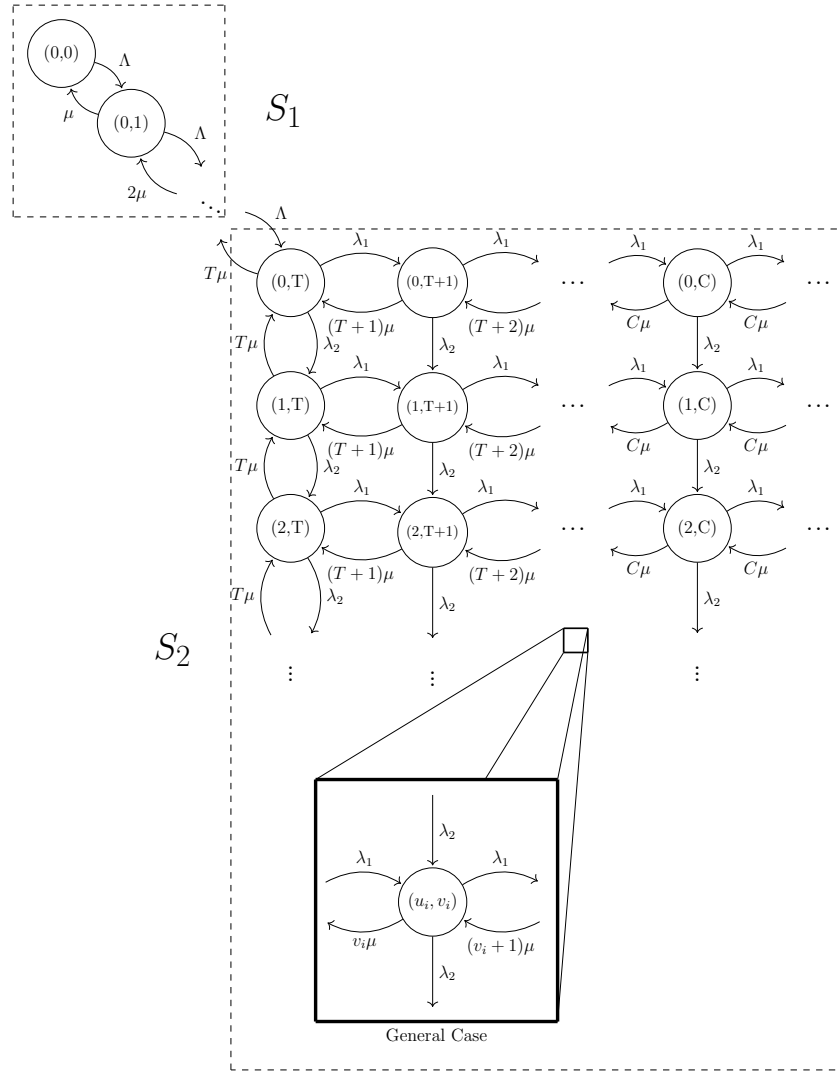


Figure 3.3: Generic case of Markov chain model. The diagram shows the two disjoint sets of states  $S_1$  and  $S_2$  and the transition rates between the states.

In order to consider this model numerically an adjustment needs to be made. The problem defined above assumes no upper boundary to the number of individuals that can wait for service or for the ones that are blocked in node 2. Therefore, a different state space  $\tilde{S}$  is constructed where  $\tilde{S} \subseteq S$  and there is a maximum allowed number of individuals  $N$  that can be in node 1 and a maximum allowed number of individuals  $M$  that can be blocked in node 2:

$$\tilde{S} = \{(u, v) \in S \mid u \leq M, v \leq N\} \quad (3.5)$$

The adjusted Markov chain model with states  $\tilde{S}$  can be seen in Figure 3.4.

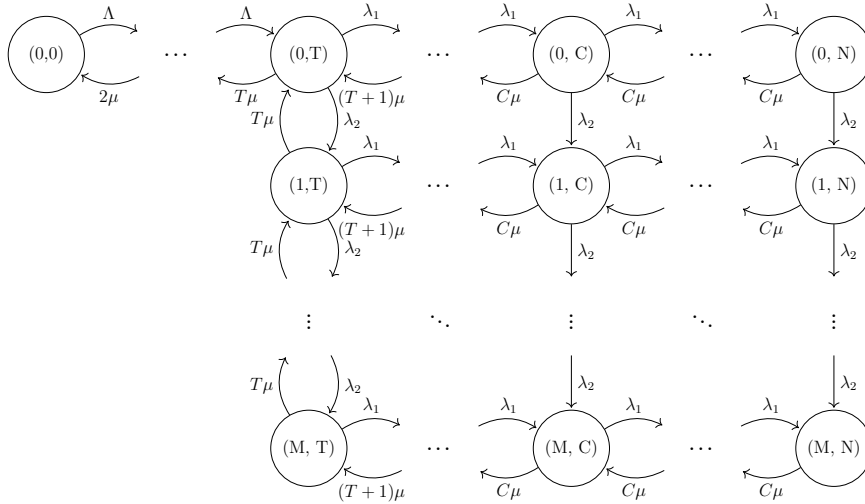


Figure 3.4: Adjusted case of the Markov chain model. The diagram makes use of the truncated state space  $\tilde{S}$  where the state space  $S$  is bounded by  $N$  and  $M$ .

The parameters  $N$  and  $M$  represent the capacities of Node 1 and Node 2 respectively. In order to define the state space  $\tilde{S}$ , these two parameters need to be finite and greater than zero. However, there is also the situation where  $N$  and  $M$  may be infinite. In that case, the Markov chain cannot model such infinite state space, but can consider these two parameters as the truncation points of the state space.

### 3.3.1 Steady state probability vector

The generator matrix  $Q$  defined in (3.3) can be used to get the probability vector  $\pi$  that contains the steady state probabilities of the Markov chain model. The vector  $\pi$  is commonly used to study stochastic systems and its main purpose is to keep track of the probability of being at any given state of the Markov chain model.  $\pi_i$  is the steady state probability of being in state  $(u_i, v_i) \in \tilde{S}$  which is the  $i^{\text{th}}$  state of  $\tilde{S}$  for some ordering of  $\tilde{S}$ . The term **steady state** refers to the instance of the vector  $\pi$  where the probabilities of being at any state becomes stable over time. Thus, by considering the steady state vector  $\pi$  the relationship between it and  $Q$  is given by:

$$\frac{d\pi}{dt} = \pi Q = \vec{0} \quad (3.6)$$

The following parameters of the Markov model will act as a running example for all approaches:

Parameter	$\lambda_1$	$\lambda_2$	$\mu$	$C$	$T$	$N$	$M$
Value	1.0	2.0	2.0	2.0	3.0	4.0	2.0

### 3.3.2 Numerical integration approach

A method that can be used to get the steady state probability vector is to solve the differential equation (3.6) numerically. Two methods of solving the differential equation were considered. Both methods observe the value of  $\pi$  over time until it reaches the steady state based on some initial starting value  $\pi_0$ :

$$\begin{aligned} \frac{d\pi}{dt} &= \pi Q \\ \pi(t_0) &= \pi_0 \\ \text{where } \pi_0 &= \left[ \frac{1}{|\pi|}, \frac{1}{|\pi|}, \dots, \frac{1}{|\pi|} \right] \end{aligned} \quad (3.7)$$

Two types of methods were considered to solve the differential equation numerically. The first method uses a combination of Adams' method [15] and the backward differentiation formula (BDF) [36]. This method is generally used to solve systems of the form  $\frac{dy}{dt} = f$  with a dense or banded Jacobian when the problem is stiff, which then uses the BDF algorithm, while when the problem is non-stiff it uses Adams' method. This was implemented using `scipy.integrate.odeint` from the python library SciPy [152] that uses the `lsoda` [116] integration method.

```
>>> import ambulance_game as abg
>>> import scipy as sci
>>> Q = abg.markov.get_transition_matrix(
...     lambda_1=1,
...     lambda_2=2,
...     mu=2,
...     num_of_servers=2,
...     threshold=3,
...     system_capacity=4,
...     buffer_capacity=2
... )
>>> pi = abg.markov.get_steady_state_numerically(
...     Q, integration_function=sci.integrate.odeint
... )
>>> pi
array([0.17596013, 0.2639402 , 0.19795515, 0.14846636, 0.08660538,
       0.05464387, 0.02474439, 0.02268236, 0.02500215])
```

Code snippet 3.6: Steady state probabilities calculation using numeric integration using the `odeint` function.

The second approach uses the explicit Runge-Kutta integration method of order

5 by controlling the error assuming accuracy of order 4 [41, 86]. The general recursive formula for the explicit family of Runge-Kutta methods is given by:

$$y_{n+1} = y_n + h \sum_{i=1}^s b_i k_i \quad (3.8)$$

$$\begin{aligned} k_1 &= f(t_n, y_n), \\ k_2 &= f(t_n + c_2 h, y_n + h(a_{21} k_1)), \\ k_3 &= f(t_n + c_3 h, y_n + h(a_{31} k_1 + a_{32} k_2)), \\ &\vdots \\ k_s &= f(t_n + c_s h, y_n + h(a_{s1} k_1 + a_{s2} k_2 + \cdots + a_{s,s-1} k_{s-1})) \end{aligned}$$

where  $y_0$  is the given initial value,  $s$  is the number of stages and  $h$  is the step size. The coefficients  $b_i$ ,  $c_i$ , and  $a_{ij}$  are usually arranged in a mnemonic device known as the Butcher's tableau. This was implemented using `scipy.integrate.solve_ivp` from the python library SciPy [152].

```
>>> pi = abg.markov.get_steady_state_numerically(
...     Q, integration_function=sci.integrate.solve_ivp
... )
>>> pi
array([0.17596012, 0.26394019, 0.19795515, 0.14846637, 0.08660539,
       0.05464388, 0.02474439, 0.02268236, 0.02500215])
```

Code snippet 3.7: Steady state probabilities calculation using numeric integration using the `solve_ivp` function.

### 3.3.3 Linear algebraic approach

The steady state probability vector  $\pi$  can be obtained by solving the linear equation:

$$Q^T \pi = \vec{0} \quad \text{such that} \quad \sum_i \pi_i = 1 \quad (3.9)$$

The two equations can be combined into one by augmenting the matrix  $Q^T$  in such a way that it includes the extra constraint  $\sum_i \pi_i = 1$ . The new augmented matrix  $\tilde{Q}$  is defined as  $Q$  with the final column replaced with a vector of ones and vector  $\vec{b}$  is defined as a column vector of 0s apart from the final element which is 1. Note that,  $\tilde{Q}$  needs to be a square matrix in order to solve the equation using linear algebra (i.e. the matrix needs to be invertible). Thus, the steady state



probability vector can be calculated by solving the linear equation:

$$\tilde{Q}^T \pi = \vec{b} \quad (3.10)$$

Using LU decomposition with partial pivoting and row interchanges, matrix  $Q^T$  can be expressed of the form  $P \times L \times U$ , where  $P$  is a permutation matrix,  $L$  is a unit lower triangular matrix, and  $U$  is an upper triangular matrix [133]. The factored form of  $Q^T$  can then be used to solve the system. This was implemented using `numpy.linalg.solve` from the python library `numpy` [8, 65].

```
>>> import numpy as np
>>> pi = abg.markov.get_steady_state_algebraically(
...     Q, algebraic_function=np.linalg.solve
... )
>>> pi
array([0.17596013, 0.2639402 , 0.19795515, 0.14846636, 0.08660538,
       0.05464387, 0.02474439, 0.02268236, 0.02500215])
```

Code snippet 3.8: Steady state probabilities calculation using linear algebraic approach with `numpy.linalg.solve`.

### 3.3.4 Least squares approach

Another approach that is considered is the least squares method. As the problem becomes more complex (i.e. as the artificial parameters  $N$  and  $M$  defined in equation (3.5) increase) the computational time required to solve it increases by a lot. Thus, one may obtain a good approximation of the steady state vector  $\pi$  by solving the following equation:

$$\pi = \operatorname{argmin}_{\pi \in \mathbb{R}^{|\pi|}} \|\tilde{Q}^T \pi - b\|_2^2 \quad (3.11)$$

The above expression gets the vector  $\pi$  that approximately solves equation  $\tilde{Q}^T \pi = b$ . This was implemented using `numpy.linalg.lstsq` from the python library `numpy` [65].

```
>>> pi = abg.markov.get_steady_state_algebraically(
...     Q, algebraic_function=np.linalg.lstsq
... )
>>> pi
array([0.17596013, 0.2639402 , 0.19795515, 0.14846636, 0.08660538,
       0.05464387, 0.02474439, 0.02268236, 0.02500215])
```

Code snippet 3.9: Steady state probabilities calculation using least squares approach with `numpy.linalg.lstsq`.

An additional approach that was considered but not completed was a closed-form formula for the steady state probabilities. The work that was done on this approach is described in Appendix D.

### 3.4 Performance measures

Using vector  $\pi$  there are numerous performance measures of the model that can be calculated. The following equations utilise  $\pi$  to get performance measures on the average number of individuals in node 1 and in node 2:

- Mean number of individuals in the entire system:

$$L_S = \sum_{i=1}^{|\pi|} \pi_i (u_i + v_i) \quad (3.12)$$

- Mean number of individuals in node 1:

$$L_1 = \sum_{i=1}^{|\pi|} \pi_i v_i \quad (3.13)$$

- Mean number of individuals in node 2:

$$L_2 = \sum_{i=1}^{|\pi|} \pi_i u_i \quad (3.14)$$

The python code for these functions may be obtained quite quickly using the set of all states and the steady state probability.

```
>>> import ambulance_game as abg
>>> import numpy as np
>>> all_states = abg.markov.build_states(
...     threshold=3 ,
...     system_capacity=4,
...     buffer_capacity=2
... )
>>> Q = abg.markov.get_transition_matrix(
...     lambda_1=1,
...     lambda_2=2,
...     mu=2,
...     num_of_servers=2,
...     threshold=3,
...     system_capacity=4,
...     buffer_capacity=2
... )
>>> pi = abg.markov.get_steady_state_algebraically(
...     Q, algebraic_function=np.linalg.lstsq
... )
```

Code snippet 3.10: Code snippet for getting the set of all states and the steady state probabilities.

Having the set of all states and the steady state probabilities, the mean number of individuals in the system, in node 1 and in node 2 can be calculated as shown in code snippets 3.11, 3.12 and 3.13.

```
>>> def get_mean_number_of_individuals_in_system(pi, states):
...     """Gets the mean number of individuals in the system
...     Parameters
...     -----
...     pi : numpy.ndarray
...         steady state vector
...     states : list
...         list of tuples that contains all states
...     Returns
...     -----
...     float
...         Mean number of individuals in the whole model
...     """
...     states = np.array(states)
...     mean_inds_in_system = np.sum((states[:, 0] + states[:, 1]) * pi)
...     return mean_inds_in_system
>>>
>>> round(get_mean_number_of_individuals_in_system(pi, all_states), 10)
2.0872927227
```

Code snippet 3.11: Code snippet for calculating the mean number of individuals in the system.

```
>>> def get_mean_number_of_individuals_in_node_1(pi, states):
...     """Mean number of individuals in node 1
...     Parameters
...     -----
...     pi : numpy.ndarray
...         steady state vector
...     states : list
...         list of tuples that contains all states
...     Returns
...     -----
...     float
...         Mean number of individuals
...     """
...     states = np.array(states)
...     mean_inds_in_node_1 = np.sum(states[:, 1] * pi)
...     return mean_inds_in_node_1
>>>
>>> round(get_mean_number_of_individuals_in_node_1(pi, all_states), 10)
1.8187129478
```

Code snippet 3.12: Code snippet for calculating the mean number of individuals in Node 1.

```

>>> def get_mean_number_of_individuals_in_node_2(pi, states):
...     """Mean number of class 2 individuals blocked
...     Parameters
...     -----
...     pi : numpy.ndarray
...         steady state vector
...     states : list
...         list of tuples that contains all states
...     Returns
...     -----
...     float
...         Mean number of blocked class 2 individuals
...     """
...     states = np.array(states)
...     mean_blocked = np.sum(states[:, 0] * pi)
...     return mean_blocked
>>>
>>> round(get_mean_number_of_individuals_in_node_2(pi, all_states), 10)
0.2685797749

```

Code snippet 3.13: Code snippet for calculating the mean number of individuals in Node 2.

Consequently, there are some additional performance measures of interest that are more complex to calculate. Such performance measures are the mean waiting time in the system (for both type 1 and type 2 individuals), the mean time blocked in node 2 (only valid for type 2 individuals) and the proportion of individuals that wait in node 1 within a predefined time target (for both types). Under the scope of this study three approaches have been considered to calculate these performance measures; a recursive algorithm, a direct approach and a closed-form equation. Furthermore, different formulas arise for type 1 individuals and different ones for type 2 individuals.

### 3.4.1 Waiting time

Waiting time is the amount of time that individuals wait in node 1 before they start their service. For a given set of parameters there are three different performance measures around the mean waiting time that need to be considered. The mean waiting time of type 1 individuals, the mean waiting time of type 2 individuals, and the overall mean waiting time.

#### 3.4.1.1 Recursive approach

The first approach to be considered is a recursive approach to getting the mean waiting time [12]. To calculate the mean waiting time of type 1 individuals one must first identify the set of states  $(u, v)$  where a wait can occur. For this particular Markov chain, these are all states that satisfy  $v > C$  i.e. all states

where the number of individuals in the node 1 exceed the number of servers. The set of such states is defined as the *waiting states* and can be denoted as a subset of all the states, where:

$$S_w = \{(u, v) \in S \mid v > C\} \quad (3.15)$$

Moreover, another element that needs to be considered is the expected waiting time spent in each state for type  $i$  individuals. In order to do so a variation of the Markov model has to be considered where arrivals are removed. Figure 3.5 shows this new Markov chain model.

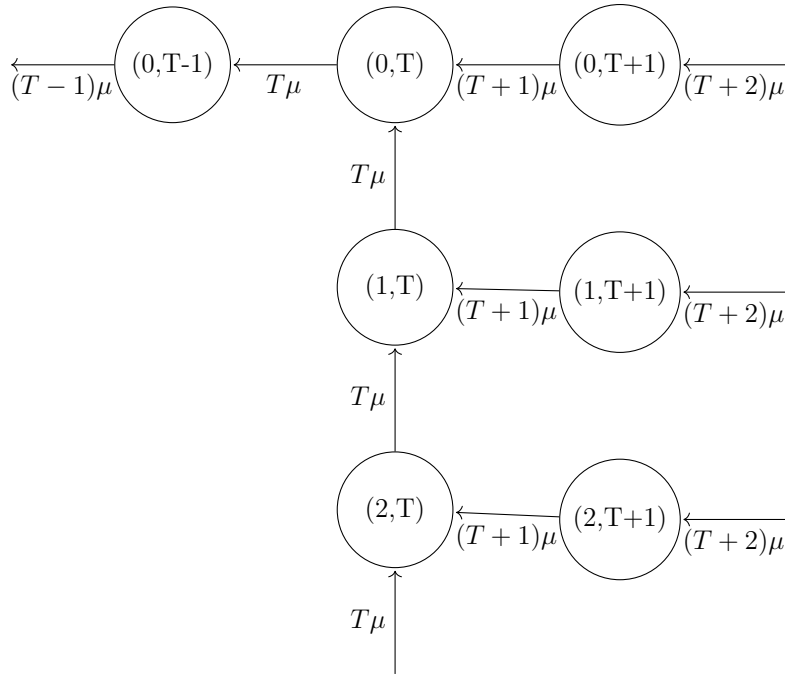


Figure 3.5: Variation of Markov chain model where all arrivals are removed. This diagram is used as a visualisation aid to illustrate how the recursive algorithm works.

For this particular Markov chain variation the expected time spent at each state  $(u, v)$  for type  $i$  individuals is denoted by  $c_w^i(u, v)$ . From a type 1 individual's perspective, when they arrive at the system, no matter how many other individuals of either type arrive after them it will not affect their own time. The desired time spent at each state can be acquired by calculating the inverse of the sum of the out-flow rate of that state. Therefore by eliminating the arrival rates of both individuals, the time spent at each state for type 1 individuals can be expressed as:

$$c_w^{(1)}(u, v) = \begin{cases} 0, & \text{if } u > 0 \text{ and } v = T \\ \frac{1}{\min(v, C)\mu}, & \text{otherwise} \end{cases} \quad (3.16)$$

Note here that whenever any type 1 individual is at a state  $(u, v)$  where  $u > 0$  and  $v = T$  (i.e. all states  $(1, T), (2, T) \dots, (M, T)$ ) the expected waiting time is set to 0. This is done to capture the trip through the Markov chain from the perspective of type 1 individuals, meaning that individuals visit all states of the threshold column but only the one in the first row will return a non-zero time. Additionally, in equation (3.16) the service rate  $\mu$  is multiplied by the minimum of  $v$  and  $C$  since, when the system is at a state  $(u, v)$  where  $v \geq C$ , the maximum out-flow service rate of  $C\mu$  is reached.

Similarly from a type 2 individual's perspective the same logic holds. The only difference is that type 2 individuals cannot have a waiting time when they are blocked in node 2. From the Markov chain model's perspective, type 2 individuals cannot have a wait whenever they are at state  $(u, v)$  where  $u > 0$ . Thus, the time at each state for type 2 individuals can be expressed as:

$$c_w^{(2)}(u, v) = \begin{cases} 0, & \text{if } u > 0 \\ \frac{1}{\min(v, C)\mu}, & \text{otherwise} \end{cases} \quad (3.17)$$

Using the set of waiting states defined in (3.15) and equations (3.16) and (3.17) the following recursive formula can be used to get the mean waiting time spent in each state. The formula goes through all states from right to left recursively and adds the total expected waiting time of all these states together until it reaches a state that is not in the set of waiting states. Thus, the expected waiting time of a type  $i$  individual when they arrive at state  $(u, v)$  is given by:

$$w^{(i)}(u, v) = \begin{cases} 0, & \text{if } (u, v) \notin S_w \\ c_w^{(i)}(u, v) + w^{(i)}(u - 1, v), & \text{if } u > 0 \text{ and } v = T \\ c_w^{(i)}(u, v) + w^{(i)}(u, v - 1), & \text{otherwise} \end{cases} \quad (3.18)$$

Whenever the system is at state  $(u, v)$  and an individual arrives, depending on the type of the individual, the system will move to a different state. The state that the Markov chain will transition to when a type  $i$  individual arrives is defined as the *arriving state*  $\mathcal{A}_i(u, v)$ . Using Figure 3.4 as reference, an arrival of a type 1 individual makes the system transition to the state on the right. Similarly

an arrival of a type 2 individual makes the system transition to the right if the threshold hasn't been reached and transition down if the threshold has been reached. This can be expressed mathematically as:

$$\mathcal{A}_1(u, v) = (u, v + 1) \quad (3.19)$$

$$\mathcal{A}_2(u, v) = \begin{cases} (u, v + 1), & \text{if } v < T \\ (u + 1, v), & \text{if } v \geq T \end{cases} \quad (3.20)$$

Additionally, there are certain states in the model where arrivals cannot occur. A type 1 individual cannot arrive whenever the model is at any state  $(u, N)$  for all  $u$ , where  $N$  is the capacity of node 1. Therefore the set of all such states that an arrival may occur are defined as *accepting states*. The set of accepting states for type 1 individuals is denoted as:

$$S_A^{(1)} = \{(u, v) \in S \mid v < N\} \quad (3.21)$$

Similarly, an arrival of a type 2 individual cannot occur whenever the model is at state  $(M, v)$  for all  $v$ , where  $M$  is the capacity of node 2. The set of accepting states for type 2 individuals is denoted as:

$$S_A^{(2)} = \{(u, v) \in S \mid u < M\} \quad (3.22)$$

Finally, the total mean waiting time can be calculated by summing over all expected waiting times of accepting states multiplied by the probability of being at that state. The different approaches that are used to get the state probabilities are described in Section 3.3.1. The mean waiting time in the system for type  $i$  individuals is given by:

$$W^{(i)} = \frac{\sum_{(u,v) \in S_A^{(i)}} \pi(u,v) w^{(i)}(\mathcal{A}_i(u,v))}{\sum_{(u,v) \in S_A^{(i)}} \pi(u,v)} \quad (3.23)$$

Consequently, using both the mean waiting time for type 1 individuals  $W^{(1)}$  and the mean waiting time for type 2 individuals  $W^{(2)}$ , the overall mean waiting time in the system is a linear combination of the 2. The overall waiting time can be then given by the following equation where  $\theta_1$  and  $\theta_2$  are the coefficients of the waiting time for each type of individual:

$$W = \theta_1 W^{(1)} + \theta_2 W^{(2)} \quad (3.24)$$

The two coefficients represent the proportion of individuals of each type that traversed through the model. Theoretically, determining these percentages should be as quick as looking at the arrival rates of each type  $\lambda_1$  and  $\lambda_2$ , but in practise if either node 1 or node 2 are full, some individuals may become lost to the system. Thus, one should account for the probability that an individual is lost to the system. This probability can be calculated by using the two sets of accepting states  $S_A^{(2)}$  and  $S_A^{(1)}$  defined earlier in (3.21) and (3.22). Let us define the probability that an individual of type  $i$  is not lost to the system as  $P(L'_i)$ :

$$P(L'_1) = \sum_{(u,v) \in S_A^{(1)}} \pi(u, v) \quad P(L'_2) = \sum_{(u,v) \in S_A^{(2)}} \pi(u, v)$$

Having defined these probabilities one may combine them with the arrival rates of each individual type in such a way to get the expected proportions of type 1 and type 2 individuals in the model.

$$\theta_1 = \frac{\lambda_1 P(L'_1)}{\lambda_2 P(L'_2) + \lambda_1 P(L'_1)}, \quad \theta_2 = \frac{\lambda_2 P(L'_2)}{\lambda_2 P(L'_2) + \lambda_1 P(L'_1)} \quad (3.25)$$

Thus, by using these values as the coefficient of equation (3.24) the resultant equation can be used to get the overall waiting time.

$$W = \frac{\lambda_1 P(L'_1)}{\lambda_2 P(L'_2) + \lambda_1 P(L'_1)} W^{(1)} + \frac{\lambda_2 P(L'_2)}{\lambda_2 P(L'_2) + \lambda_1 P(L'_1)} W^{(2)} \quad (3.26)$$

#### 3.4.1.1.1 Implementation

Implementing the recursive approach for the waiting time in python uses the same logical structure as described in Section 3.4.1.1. The first function needed is one that checks if a state belongs in the set of waiting states that corresponds to the set defined in (3.15).

```
>>> def is_waiting_state(state, num_of_servers):
...     """Checks if waiting occurs in the given state. In essence, all
...     states (u,v) where v > C are considered waiting states.
...     Parameters
...     -----
...     state : tuple
...         a tuples of the form (u,v)
...     num_of_servers : int
...         the number of servers = C
```



```

...     Returns
...     -----
...     Boolean
...         An indication of whether or not any wait occurs on the given
...         state
...     """
...     return state[1] > num_of_servers
>>>
>>> is_waiting_state(state=(1, 4), num_of_servers=2)
True

```

Code snippet 3.14: Function that checks if a state is a waiting state.

Similarly a function that calculates the expected wait in each state is needed that corresponds to equations (3.16) and (3.17). Note here that the following function takes the individuals type as an argument and thus only one function is needed for both expressions.

```

>>> def expected_time_in_markov_state_ignoring_arrivals(
...     state,
...     class_type,
...     num_of_servers,
...     mu,
...     threshold,
... ):
...     """Get the expected waiting time in a Markov state when ignoring any
...     subsequent arrivals. When considering the waiting time of class 2
...     individuals, and when these individuals are in a blocked state
...     ( $v > 0$ ) then by the definition of the problem the waiting time in
...     that state is set to 0. Additionally, all states where  $u > 0$  and
...      $v = T$  automatically get a waiting time of 0 because class 1
...     individuals only pass one of the states of that column (only state
...      $(0, T)$  is not zero).
...
...     Parameters
...     -----
...     state : tuple
...         a tuples of the form  $(u, v)$ 
...     class_type : int
...         A string to distinguish between class 1( $=0$ ) and class 2( $=1$ )
...         individuals
...     num_of_servers : int
...         The number of servers =  $C$ 
...     mu : float
...         The service rate =  $\mu$ 
...
...     Returns
...     -----
...     float
...         The expected waiting time in the given state
...     """
...     if state[0] > 0 and (state[1] == threshold or class_type == 1):
...         return 0
...     return 1 / (min(state[1], num_of_servers) * mu)
>>>
>>> expected_time_in_markov_state_ignoring_arrivals(
...     state=(3, 4),

```

```

...     class_type=0,
...     num_of_servers=1,
...     mu=4,
...     threshold=2,
... )
0.25

```

Code snippet 3.15: Function for the expected waiting time in a state.

The following block of code is the implementation of equation (3.18), where it returns the waiting time of an individual when they arrive at a given state until they leave that particular state. Note that this function uses both of the functions defined earlier.

```

>>> def get_waiting_time_for_each_state_recursively(
...     state,
...     class_type,
...     lambda_2,
...     lambda_1,
...     mu,
...     num_of_servers,
...     threshold,
...     system_capacity,
...     buffer_capacity,
... ):
...     """Performs a recursive algorithm to get the expected waiting time of
...     individuals when they enter the model at a given state. Given an
...     arriving state the algorithm moves down to all subsequent states
...     until it reaches one that is not a waiting state.
...
...     Class 1:
...     - If (u,v) not a waiting state: return 0
...     - Next state s_d = (0, v - 1)
...     - w(u,v) = c(u,v) + w(s_d)
...
...     Class 2:
...     - If (u,v) not a waiting state: return 0
...     - Next state: s_n = (u-1, v), if u >= 1 and v=T
...       s_n = (u, v - 1), otherwise
...     - w(u,v) = c(u,v) + w(s_n)
...
...     Note: For all class 1 individuals the recursive formula acts in a
...     linear manner meaning that an individual will have the same waiting
...     time when arriving at any state of the same column e.g (2, 3) or
...     (5, 3).
...
...     Parameters
...     -----
...     state : tuple
...     class_type : int
...     lambda_2 : float
...     lambda_1 : float
...     mu : float
...     num_of_servers : int
...     threshold : int
...     system_capacity : int
...     buffer_capacity : int

```

```

...
...     Returns
...     -----
...     float
...     The expected waiting time from the arriving state of an
...     individual until service
...     """
...     if not is_waiting_state(state, num_of_servers):
...         return 0
...     if state[0] >= 1 and state[1] == threshold:
...         next_state = (state[0] - 1, state[1])
...     else:
...         next_state = (state[0], state[1] - 1)
...
...     wait = expected_time_in_markov_state_ignoring_arrivals(
...         state=state,
...         class_type=class_type,
...         num_of_servers=num_of_servers,
...         mu=mu,
...         threshold=threshold,
...     )
...     wait += get_waiting_time_for_each_state_recursively(
...         state=next_state,
...         class_type=class_type,
...         lambda_2=lambda_2,
...         lambda_1=lambda_1,
...         mu=mu,
...         num_of_servers=num_of_servers,
...         threshold=threshold,
...         system_capacity=system_capacity,
...         buffer_capacity=buffer_capacity,
...     )
...     return wait
>>>
>>> get_waiting_time_for_each_state_recursively(
...     state=(3, 4),
...     class_type=0,
...     lambda_2=1,
...     lambda_1=1,
...     mu=4,
...     num_of_servers=1,
...     threshold=2,
...     system_capacity=4,
...     buffer_capacity=3,
... )
0.75

```

Code snippet 3.16: Function for the overall expected waiting time in a state using recursion.

Additionally, before getting the mean waiting time for each type of individuals, a function for the set of accepting states described in (3.21) and (3.22) needs to be constructed.

```

>>> def is_accepting_state(
...     state, class_type, threshold, system_capacity, buffer_capacity
... ):

```

```

... """
...     Checks if a state given is an accepting state. Accepting states are
...     defined as the states of the system where arrivals may occur. In
...     essence these states are all states apart from the one when the system
...     cannot accept additional arrivals. Because there are two types of
...     arrivals though, the set of accepting states is different for class 1
...     and class 2 individuals:
...
...     Parameters
...     -----
...     state : tuple
...         a tuples of the form (u,v)
...     class_type : int
...         A string to distinguish between class 1 (=0) and class 2
...         individuals (=1)
...     system_capacity : int
...         The capacity of the system (hospital) = N
...     buffer_capacity : int
...         The capacity of the buffer space = M
...
...     Returns
...     -----
...     Boolean
...         An indication of whether or not an arrival of the given type
...         (class_type) can occur
...     """
...     if class_type == 1:
...         condition = (
...             (state[0] < buffer_capacity)
...             if (threshold <= system_capacity)
...             else (state[1] < system_capacity)
...         )
...     if class_type == 0:
...         condition = state[1] < system_capacity
...     return condition

```

Code snippet 3.17: Function to check if a state is an accepting state.

The only thing left to do is to find the weighted average of the waiting times for all states using the steady state probabilities. The function defined in 3.18 corresponds to the expression for  $W^{(i)}$  defined in equation (3.23).

```

>>> import ambulance_game as abg
>>> import numpy as np
>>> def mean_waiting_time_formula_using_recursive_approach(
...     all_states,
...     pi,
...     class_type,
...     lambda_2,
...     lambda_1,
...     mu,
...     num_of_servers,
...     threshold,
...     system_capacity,
...     buffer_capacity,
...     **kwargs,
... ):
...     """

```

```

...     Get the mean waiting time by using a recursive formula.
...     All  $w(u,v)$  terms are calculated recursively by going through
...     the waiting times of all previous states.
...
...     Parameters
...     -----
...     all_states : list
...     pi : array
...     class_type : int
...     lambda_2 : float
...     lambda_1 : float
...     mu : float
...     num_of_servers : int
...     threshold : int
...     system_capacity : int
...     buffer_capacity : int
...
...     Returns
...     -----
...     float
...     """
...     mean_waiting_time = 0
...     probability_of_accepting = 0
...     for u, v in all_states:
...         if is_accepting_state(
...             state=(u, v),
...             class_type=class_type,
...             threshold=threshold,
...             system_capacity=system_capacity,
...             buffer_capacity=buffer_capacity,
...         ):
...             arriving_state = (u, v + 1)
...             if class_type == 1 and v >= threshold:
...                 arriving_state = (u + 1, v)
...
...             current_state_wait = get_waiting_time_for_each_state_recursively
...     (
...         state=arriving_state,
...         class_type=class_type,
...         lambda_2=lambda_2,
...         lambda_1=lambda_1,
...         mu=mu,
...         num_of_servers=num_of_servers,
...         threshold=threshold,
...         system_capacity=system_capacity,
...         buffer_capacity=buffer_capacity,
...     )
...     mean_waiting_time += current_state_wait * pi[u, v]
...     probability_of_accepting += pi[u, v]
...     return mean_waiting_time / probability_of_accepting
>>>
>>> all_states = abg.markov.build_states(
...     threshold=2,
...     system_capacity=4,
...     buffer_capacity=3,
... )
>>> Q = abg.markov.get_transition_matrix(
...     lambda_1=1,
...     lambda_2=1,

```

```

...     mu=4,
...     num_of_servers=1,
...     threshold=2,
...     system_capacity=4,
...     buffer_capacity=3
... )
>>> pi = abg.markov.get_markov_state_probabilities(
...     abg.markov.get_steady_state_algebraically(
...         Q, algebraic_function=np.linalg.solve
...     ),
...     all_states,
... )
>>> round(
...     mean_waiting_time_formula_using_recursive_approach(
...         all_states=all_states,
...         pi=pi,
...         class_type=0,
...         lambda_2=1,
...         lambda_1=1,
...         mu=4,
...         num_of_servers=1,
...         threshold=2,
...         system_capacity=4,
...         buffer_capacity=3,
...     ), 10
... )
0.192140129

```

Code snippet 3.18: Function to get the mean waiting time recursively for a specific individual type.

Finally the overall waiting time for both individuals can be calculated by taking the weighted average of the waiting times for each type of individual as described in equation (3.26).

```

>>> def overall_waiting_time_formula(
...     all_states,
...     pi,
...     lambda_2,
...     lambda_1,
...     mu,
...     num_of_servers,
...     threshold,
...     system_capacity,
...     buffer_capacity,
...     waiting_formula,
...     **kwargs,
... ):
...     """
...     Gets the overall waiting time for all individuals by calculating both
...     class 1 and class 2 waiting times. Thus, considering the probability
...     that an individual is lost to the system (for both classes)
...     calculates the overall waiting time.
...
...     Parameters
...     -----
...     all_states : list

```

```

...     pi : array
...     lambda_1 : float
...     lambda_2 : float
...     mu : float
...     num_of_servers : int
...     threshold : int
...     system_capacity : int
...     buffer_capacity : int
...     waiting_formula : function
...
...     Returns
...     -----
...     float
...         The overall mean waiting time by combining class 1 and class 2
...         individuals
...     """
...     mean_waiting_times_for_each_class = [
...         waiting_formula(
...             all_states=all_states,
...             pi=pi,
...             class_type=class_type,
...             lambda_2=lambda_2,
...             lambda_1=lambda_1,
...             mu=mu,
...             num_of_servers=num_of_servers,
...             threshold=threshold,
...             system_capacity=system_capacity,
...             buffer_capacity=buffer_capacity,
...         )
...         for class_type in range(2)
...     ]
...     prob_accept = [
...         np.sum(
...             [
...                 pi[state]
...                 for state in all_states
...                 if is_accepting_state(
...                     state=state,
...                     class_type=class_type,
...                     threshold=threshold,
...                     system_capacity=system_capacity,
...                     buffer_capacity=buffer_capacity,
...                 )
...             ]
...         )
...         for class_type in range(2)
...     ]
...     class_rates = [
...         prob_accept[class_type]
...         / ((lambda_2 * prob_accept[1]) + (lambda_1 * prob_accept[0]))
...         for class_type in range(2)
...     ]
...     class_rates[0] *= lambda_1
...     class_rates[1] *= lambda_2
...     mean_waiting_time = np.sum(
...         [
...             mean_waiting_times_for_each_class[class_type]
...             * class_rates[class_type]
...             for class_type in range(2)

```

```

...     ]
...     )
...     return mean_waiting_time
>>>
>>> round(overall_waiting_time_formula(
...     all_states=all_states,
...     pi=pi,
...     lambda_2=1,
...     lambda_1=1,
...     mu=4,
...     num_of_servers=1,
...     threshold=2,
...     system_capacity=4,
...     buffer_capacity=3,
...     waiting_formula=mean_waiting_time_formula_using_recursive_approach,
... ), 10)
0.1572461886

```

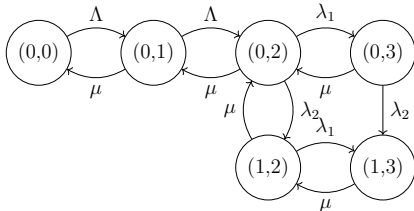
Code snippet 3.19: Function for the overall waiting time formula.

### 3.4.1.2 Direct approach

The direct approach uses similar concepts to the recursive approach of Section 3.4.1.1. Instead of using recursion, a linear system of the set of equations generated by equation (3.18) for every state  $(u, v)$  is solved. The set of equations that need to be solved for individuals of type  $i$  are all  $w^{(i)}(u, v)$  for all possible states  $(u, v) \in S$ .

$$w^{(i)}(u, v) = \begin{cases} 0, & \text{if } (u, v) \notin S_w \\ c_w^{(i)}(u, v) + w^{(i)}(u - 1, v), & \text{if } u > 0 \text{ and } v = T \\ c_w^{(i)}(u, v) + w^{(i)}(u, v - 1), & \text{otherwise} \end{cases}$$

Consider a relatively small model where  $C = 1, T = 2, N = 3, M = 1$ . All possible equations  $w^{(i)}(u, v)$  are given by equations (3.29) - (3.32).



$$w^{(i)}(0, 0) = 0 \quad (3.27)$$

$$w^{(i)}(0, 1) = 0 \quad (3.28)$$

$$w^{(i)}(0, 2) = c_w^{(i)}(0, 2) + w^{(i)}(0, 1) \quad (3.29)$$

$$w^{(i)}(0, 3) = c_w^{(i)}(0, 3) + w^{(i)}(0, 2) \quad (3.30)$$

$$w^{(i)}(1, 2) = c_w^{(i)}(1, 2) + w^{(i)}(0, 2) \quad (3.31)$$

$$w^{(i)}(1, 3) = c_w^{(i)}(1, 3) + w^{(i)}(1, 2) \quad (3.32)$$

Figure 3.6: Markov chain example with  $C = 1, T = 2, N = 3, M = 1$

Additionally, the above equations can be transformed into a linear system of the



form  $Zx = y$  where:

$$Z = \begin{pmatrix} -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \end{pmatrix}, x = \begin{pmatrix} w^{(i)}(0, 0) \\ w^{(i)}(0, 1) \\ w^{(i)}(0, 2) \\ w^{(i)}(0, 3) \\ w^{(i)}(1, 2) \\ w^{(i)}(1, 3) \end{pmatrix}, y = \begin{pmatrix} 0 \\ 0 \\ -c_w^{(i)}(0, 2) \\ -c_w^{(i)}(0, 3) \\ -c_w^{(i)}(1, 2) \\ -c_w^{(i)}(1, 3) \end{pmatrix} \quad (3.33)$$

A more generalised form of the equations in (3.33) can be given for any value of  $C, T, N, M$  by:

$$w^{(i)}(0, 0) = 0 \quad (3.34)$$

$$w^{(i)}(0, 1) = c_w^{(i)}(0, 1) + w^{(i)}(0, 0) \quad (3.35)$$

$$w^{(i)}(0, 2) = c_w^{(i)}(0, 2) + w^{(i)}(0, 1) \quad (3.36)$$

$$\vdots$$

$$w^{(i)}(0, T-1) = c_w^{(i)}(0, T-1) + w^{(i)}(0, T-2) \quad (3.37)$$

$$w^{(i)}(0, T) = c_w^{(i)}(0, T) + w^{(i)}(0, T-1) \quad (3.38)$$

$$w^{(i)}(0, T+1) = c_w^{(i)}(0, T+1) + w^{(i)}(0, T) \quad (3.39)$$

$$w^{(i)}(0, T+2) = c_w^{(i)}(0, T+2) + w^{(i)}(0, T+1) \quad (3.40)$$

$$\vdots$$

$$w^{(i)}(0, N) = c_w^{(i)}(0, N) + w^{(i)}(0, N-1) \quad (3.41)$$

$$w^{(i)}(1, T) = c_w^{(i)}(1, T) + w^{(i)}(0, T) \quad (3.42)$$

$$w^{(i)}(1, T+1) = c_w^{(i)}(1, T+1) + w^{(i)}(1, T) \quad (3.43)$$

$$\vdots$$

$$w^{(i)}(M, N) = c_w^{(i)}(M, N) + w^{(i)}(M, N-1) \quad (3.44)$$

The equivalent matrix form of the linear system of equations (3.34) - (3.44) is given by  $Zx = y$ , where:

$$Z = \begin{pmatrix}
 -1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\
 1 & -1 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\
 0 & 1 & -1 & \dots & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\
 \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
 0 & 0 & 0 & \dots & -1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\
 0 & 0 & 0 & \dots & 1 & -1 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\
 0 & 0 & 0 & \dots & 0 & 1 & -1 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\
 0 & 0 & 0 & \dots & 0 & 0 & 1 & -1 & \dots & 0 & 0 & 0 & \dots & 0 \\
 \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & -1 & 0 & 0 & \dots & 0 \\
 0 & 0 & 0 & \dots & 0 & 1 & 0 & 0 & \dots & 0 & -1 & 0 & \dots & 0 \\
 0 & 0 & 0 & \dots & 0 & 0 & 1 & 0 & \dots & 0 & 0 & -1 & \dots & 0 \\
 \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & -1
 \end{pmatrix}$$

$$x = \begin{pmatrix}
 w^{(i)}(0, 0) \\
 w^{(i)}(0, 1) \\
 w^{(i)}(0, 2) \\
 \vdots \\
 w^{(i)}(0, T-1) \\
 w^{(i)}(0, T) \\
 w^{(i)}(0, T+1) \\
 w^{(i)}(0, T+2) \\
 \vdots \\
 w^{(i)}(0, N) \\
 w^{(i)}(1, T) \\
 w^{(i)}(1, T+1) \\
 \vdots \\
 w^{(i)}(M, N)
 \end{pmatrix}, y = \begin{pmatrix}
 0 \\
 -c_w^{(i)}(0, 1) \\
 -c_w^{(i)}(0, 2) \\
 \vdots \\
 -c_w^{(i)}(0, T-1) \\
 -c_w^{(i)}(0, T) \\
 -c_w^{(i)}(0, T+1) \\
 -c_w^{(i)}(0, T+2) \\
 \vdots \\
 -c_w^{(i)}(0, N) \\
 -c_w^{(i)}(1, T) \\
 -c_w^{(i)}(1, T+1) \\
 \vdots \\
 -c_w^{(i)}(M, N)
 \end{pmatrix} \quad (3.45)$$

Thus, solving for  $x$  gets the values of all  $w^{(i)}(u, v)$  for all states  $(u, v) \in S$ . These values can then be used with equation (3.23) to compute the mean waiting time for type  $i$  individuals  $W^{(i)}$ . Now, having  $W^{(1)}$  and  $W^{(2)}$ , equation (3.26) can be utilised once more to compute the overall mean waiting time for both individual types.

### 3.4.1.2.1 Implementation

Similar to the implementation of the recursive approach from Section 3.4.1.1.1 the functions that correspond to equations (3.15), (3.16), (3.17), (3.21), (3.22)

will be used again. For the implementation of the direct approach the aim is to construct matrix  $Z$  and vector  $y$  as described in Section 3.4.1.2 in order to solve the system of linear equations described in (3.4.1.2).

The block of code in 3.20 returns the values of one row of matrix  $Z$  that corresponds to the state  $(u, v)$  and the value of vector  $y$  that correspond to the state  $(u, v)$ .

```
>>> import itertools
>>> def get_coefficients_row_of_array_for_state(
...     state,
...     class_type,
...     mu,
...     num_of_servers,
...     threshold,
...     system_capacity,
...     buffer_capacity
... ):
...     """
...     For direct approach: Constructs a row of the coefficients matrix.
...     The row to be constructed corresponds to the waiting time equation
...     for a given state (u,v)
...     """
...     lhs_coefficient_row = np.zeros(
...         [buffer_capacity + 1, system_capacity + 1]
...     )
...     lhs_coefficient_row[state[0], state[1]] = -1
...     for (u, v) in itertools.product(
...         range(1, buffer_capacity + 1), range(threshold)
...     ):
...         lhs_coefficient_row[u, v] = np.NaN
...
...     rhs_value = 0
...     if is_waiting_state(state, num_of_servers):
...         if state[0] >= 1 and state[1] == threshold:
...             next_state = (state[0] - 1, state[1])
...         else:
...             next_state = (state[0], state[1] - 1)
...
...         lhs_coefficient_row[next_state[0], next_state[1]] = 1
...         rhs_value = -expected_time_in_markov_state_ignoring_arrivals(
...             state=state,
...             class_type=class_type,
...             mu=mu,
...             num_of_servers=num_of_servers,
...             threshold=threshold,
...         )
...     vectorised_array = np.hstack(
...         (
...             lhs_coefficient_row[0, :threshold],
...             lhs_coefficient_row[:, threshold:].flatten("F"),
...         )
...     )
...     return vectorised_array, rhs_value
>>>
>>> get_coefficients_row_of_array_for_state(
...     state=(2,3),
```

```

...     class_type=0,
...     mu=4,
...     num_of_servers=1,
...     threshold=2,
...     system_capacity=3,
...     buffer_capacity=2
... )
(array([ 0.,  0.,  0.,  0.,  1.,  0.,  0., -1.]), -0.25)

```

Code snippet 3.20: Function to get a row of the coefficients matrix and the corresponding value of the right hand side vector.

In code snippet 3.20, the function returns a tuple with two elements; the row of matrix  $Z$  and the value of vector  $y$  that corresponds to state (2,3). Using the function defined in 3.20 the matrix  $Z$  and vector  $y$  can be constructed by considering all states of the Markov chain.

```

>>> def get_waiting_time_linear_system(
...     class_type,
...     mu,
...     num_of_servers,
...     threshold,
...     system_capacity,
...     buffer_capacity
... ):
...     """
...     For direct approach: Obtain the linear system  $Z X = y$  by finding
...     the array  $Z$  and the column vector  $y$  that are required. Here  $Z$  is
...     denoted as "all_coefficients_array" and  $y$  as "constant_column".
...     The function stacks the outputs of
...     get_coefficients_row_of_array_for_state() for all states. In
...     essence all outputs are stacked together to form a square matrix
...     (/) and equivalently a column vector ( $y$ ) that will be used to find
...      $X$  s.t.  $Z * X = y$ 
...     """
...     all_coefficients_array = np.array([])
...     all_states = abg.markov.build_states(
...         threshold=threshold,
...         system_capacity=system_capacity,
...         buffer_capacity=buffer_capacity,
...     )
...     for state in all_states:
...         lhs_vector, rhs_value = get_coefficients_row_of_array_for_state(
...             state=state,
...             class_type=class_type,
...             mu=mu,
...             num_of_servers=num_of_servers,
...             threshold=threshold,
...             system_capacity=system_capacity,
...             buffer_capacity=buffer_capacity,
...         )
...         if len(all_coefficients_array) == 0:
...             all_coefficients_array = [lhs_vector]
...             constant_column = [rhs_value]
...         else:
...             all_coefficients_array = np.vstack(
...                 [all_coefficients_array, lhs_vector]
...             )

```

```

...         )
...         constant_column.append(rhs_value)
...     return all_coefficients_array, constant_column
>>>
>>> Z, y = get_waiting_time_linear_system(
...     class_type=0,
...     mu=4,
...     num_of_servers=1,
...     threshold=2,
...     system_capacity=3,
...     buffer_capacity=2
... )
>>> Z
array([[ -1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0., -1.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  1., -1.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1., -1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1., -1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0., -1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.,  0., -1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.,  0.,  0., -1.]])
>>> y
[0, 0, -0.25, 0, 0, -0.25, -0.25, -0.25]

```

Code snippet 3.21: Function that formulates (but does not solve) the linear system needed to get the waiting time.

The piece of code in 3.22 solves the linear system  $ZX = y$  to obtain the vector  $X$  containing the waiting times for all states of the Markov chain. After solving the linear system for vector  $X$  it also converts the 1-dimensional array into a 2-dimensional array where the entry at row  $u$  and column  $v$  corresponds to the expected waiting time that an individual will have to wait when arriving and the Markov chain is at state  $(u, v)$ .

```

>>> def convert_solution_to_correct_array_format(
...     array, all_states, system_capacity, buffer_capacity
... ):
...     """
...     For direct approach: Convert the solution into a format that matches
...     the state probabilities array. The given array is a one-dimensional
...     array with the waiting times of each state
...     """
...     array_with_correct_shape = np.zeros(
...         [buffer_capacity + 1, system_capacity + 1]
...     )
...     for index, (u, v) in enumerate(all_states):
...         array_with_correct_shape[u, v] = array[index]
...     return array_with_correct_shape
>>>
>>>
>>> def get_waiting_times_of_all_states_using_direct_approach(
...     class_type,
...     all_states,
...     mu,
...     num_of_servers,
...     threshold,

```

```

...     system_capacity,
...     buffer_capacity,
... ):
...     """
...     For direct approach: Solve  $M \cdot X = b$  using numpy.linalg.solve() where:
...      $M$  = The array containing the coefficients of all  $w(u,v)$  equations
...      $b$  = Vector of constants of equations
...      $X$  = All  $w(u,v)$  variables of the equations
...     """
...     M, b = get_waiting_time_linear_system(
...         class_type=class_type,
...         mu=mu,
...         num_of_servers=num_of_servers,
...         threshold=threshold,
...         system_capacity=system_capacity,
...         buffer_capacity=buffer_capacity,
...     )
...     state_waiting_times = np.linalg.solve(M, b)
...     state_waiting_times = convert_solution_to_correct_array_format(
...         array=state_waiting_times,
...         all_states=all_states,
...         system_capacity=system_capacity,
...         buffer_capacity=buffer_capacity,
...     )
...     return state_waiting_times
>>>
>>> all_states = abg.markov.build_states(
...     threshold=2,
...     system_capacity=3,
...     buffer_capacity=2,
... )
>>>
>>> get_waiting_times_of_all_states_using_direct_approach(
...     class_type=0,
...     all_states=all_states,
...     mu=4,
...     num_of_servers=1,
...     threshold=2,
...     system_capacity=3,
...     buffer_capacity=2
... )
array([[ -0.    , -0.    ,  0.25,  0.5 ],
       [ 0.    ,  0.    ,  0.25,  0.5 ],
       [ 0.    ,  0.    ,  0.25,  0.5 ]])

```

Code snippet 3.22: Functions to solve the linear system and get waiting time for each state

Finally, similar to Section 3.4.1.1.1, using equation (3.23) the mean waiting time for either type of individuals can be calculated as shown in 3.23.

```

>>> def mean_waiting_time_formula_using_direct_approach(
...     all_states,
...     pi,
...     class_type,
...     lambda_2,
...     lambda_1,
...     mu,

```

```

...     num_of_servers,
...     threshold,
...     system_capacity,
...     buffer_capacity,
...     **kwargs,
... ):
...     """
...     Get the mean waiting time by using a direct approach.
...     """
...     wait_times = get_waiting_times_of_all_states_using_direct_approach(
...         class_type=class_type,
...         all_states=all_states,
...         mu=mu,
...         num_of_servers=num_of_servers,
...         threshold=threshold,
...         system_capacity=system_capacity,
...         buffer_capacity=buffer_capacity,
...     )
...     mean_waiting_time, probab_accept_class_2_ind = 0, 0
...     for (u, v) in all_states:
...         if is_accepting_state(
...             state=(u, v),
...             class_type=class_type,
...             threshold=threshold,
...             system_capacity=system_capacity,
...             buffer_capacity=buffer_capacity,
...         ):
...             arriving_state = (u, v + 1)
...             if class_type == 1 and v >= threshold:
...                 arriving_state = (u + 1, v)
...             mean_waiting_time += wait_times[arriving_state] * pi[u, v]
...             probab_accept_class_2_ind += pi[u, v]
...     return mean_waiting_time / probab_accept_class_2_ind
>>>
>>> all_states = abg.markov.build_states(
...     threshold=2,
...     system_capacity=4,
...     buffer_capacity=3,
... )
>>> Q = abg.markov.get_transition_matrix(
...     lambda_1=1,
...     lambda_2=1,
...     mu=4,
...     num_of_servers=1,
...     threshold=2,
...     system_capacity=4,
...     buffer_capacity=3
... )
>>> pi = abg.markov.get_markov_state_probabilities(
...     abg.markov.get_steady_state_algebraically(
...         Q, algebraic_function=np.linalg.solve
...     ),
...     all_states,
... )
>>> round(
...     mean_waiting_time_formula_using_direct_approach(
...         all_states=all_states,
...         pi=pi,

```

```

...     class_type=0,
...     lambda_2=1,
...     lambda_1=1,
...     mu=4,
...     num_of_servers=1,
...     threshold=2,
...     system_capacity=4,
...     buffer_capacity=3,
... ), 10
... )
0.192140129

```

Code snippet 3.23: Function to calculate the mean waiting time for a given individual type using the direct approach

### 3.4.1.3 Closed-form approach

The final approach for getting the mean waiting time is to use a closed-form approach. This approach is an immediate simplification of the recursive approach described in Section 3.4.1.1.

$$W^{(1)} = \frac{\sum_{\substack{(u,v) \in S_A^{(1)} \\ v \geq C}} \frac{1}{C\mu} \times (v - C + 1) \times \pi(u, v)}{\sum_{(u,v) \in S_A^{(1)}} \pi(u, v)} \quad (3.46)$$

The mean waiting time of type 2 individuals:

$$W^{(2)} = \frac{\sum_{\substack{(u,v) \in S_A^{(2)} \\ \min(v, T) \geq C}} \frac{1}{C\mu} \times (\min(v + 1, T) - C) \times \pi(u, v)}{\sum_{(u,v) \in S_A^{(2)}} \pi(u, v)} \quad (3.47)$$

Having  $W^{(1)}$  and  $W^{(2)}$ , equation (3.26) can then be used to compute  $W$ , the overall mean waiting time for both types.

#### 3.4.1.3.1 Implementation

The closed-form method can be implemented in one function shown in code snippet 3.24. The function is broken down in two parts for the case of each individual type.

```

>>> def mean_waiting_time_formula_using_closed_form_approach(
...     all_states,
...     pi,
...     class_type,
...     mu,
...     num_of_servers,
...     threshold,
...     system_capacity,

```



```

...     buffer_capacity,
...     **kwargs,
... ):
...     """
...     Get the mean waiting time by using a closed-form method.
...     """
...     sojourn_time = 1 / (num_of_servers * mu)
...     if class_type == 0:
...         mean_waiting_time = np.sum(
...             [
...                 (state[1] - num_of_servers + 1) * pi[state]
...                 * sojourn_time
...                 for state in all_states
...                 if is_accepting_state(
...                     state=state,
...                     class_type=class_type,
...                     threshold=threshold,
...                     system_capacity=system_capacity,
...                     buffer_capacity=buffer_capacity,
...                 )
...                 and state[1] >= num_of_servers
...             ]
...         ) / np.sum(
...             [
...                 pi[state]
...                 for state in all_states
...                 if is_accepting_state(
...                     state=state,
...                     class_type=class_type,
...                     threshold=threshold,
...                     system_capacity=system_capacity,
...                     buffer_capacity=buffer_capacity,
...                 )
...             ]
...         )
...     if class_type == 1:
...         mean_waiting_time = np.sum(
...             [
...                 (min(state[1] + 1, threshold) - num_of_servers)
...                 * pi[state]
...                 * sojourn_time
...                 for state in all_states
...                 if is_accepting_state(
...                     state=state,
...                     class_type=class_type,
...                     threshold=threshold,
...                     system_capacity=system_capacity,
...                     buffer_capacity=buffer_capacity,
...                 )
...                 and min(state[1], threshold) >= num_of_servers
...             ]
...         ) / np.sum(
...             [
...                 pi[state]
...                 for state in all_states
...                 if is_accepting_state(
...                     state=state,
...                     class_type=class_type,

```

```

...         threshold=threshold,
...         system_capacity=system_capacity,
...         buffer_capacity=buffer_capacity,
...     )
...     ]
... )
... return mean_waiting_time
>>>
>>> all_states = abg.markov.build_states(
...     threshold=2,
...     system_capacity=4,
...     buffer_capacity=3,
... )
>>> Q = abg.markov.get_transition_matrix(
...     lambda_1=1,
...     lambda_2=1,
...     mu=4,
...     num_of_servers=1,
...     threshold=2,
...     system_capacity=4,
...     buffer_capacity=3
... )
>>> pi = abg.markov.get_markov_state_probabilities(
...     abg.markov.get_steady_state_algebraically(
...         Q, algebraic_function=np.linalg.solve
...     ),
...     all_states,
... )
>>> round(
...     mean_waiting_time_formula_using_closed_form_approach(
...         all_states=all_states,
...         pi=pi,
...         class_type=0,
...         mu=4,
...         num_of_servers=1,
...         threshold=2,
...         system_capacity=4,
...         buffer_capacity=3,
...     ), 10
... )
0.192140129

```

Code snippet 3.24: Function to calculate the mean waiting time for a given individual type using the closed-form approach

A numeric comparison of the 3 approaches used to compute the mean waiting time can be found in Section 3.5.1.

### 3.4.2 Blocking time

Unlike the waiting time in Section 3.4.1, the blocking time is only calculated for type 2 individuals. That is because type 1 individuals cannot be blocked. Thus, one only needs to consider the pathway of type 2 individuals to get the mean blocking time of the system.

For the waiting time formula described in equation (3.23) in Section 3.4.1.1 the expected waiting time for each state was considered by ignoring all arrivals. Here, the same approach is used but ignoring only arrivals of type 2 individuals. That is because for the waiting time formula, once an individual enters node 1 (i.e. starts waiting) any individual arriving after them will not affect their pathway. That is not the case for the blocking time. When a type 2 individual is blocked, any type 1 individual that arrives will cause the blocked individual to stay blocked for more time. Therefore, unlike Figure 3.5, type 1 arrivals are considered here. Once again a variation of the already existing Markov chain model described in Figure 3.4 can be seen in Figure 3.7 where type 2 arrivals are ignored.

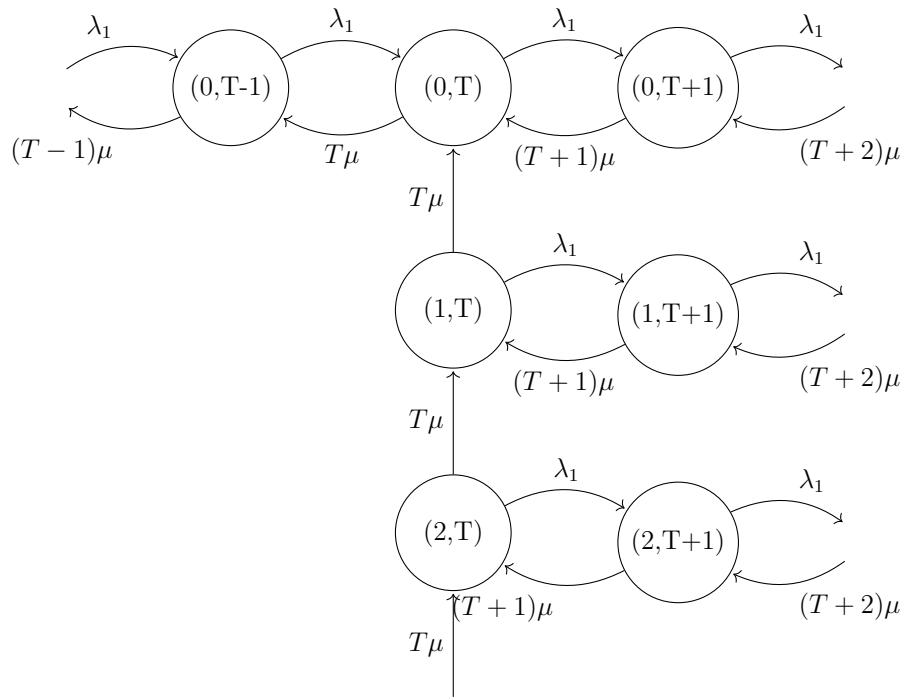


Figure 3.7: Variation of Markov chain model where type 2 arrivals are removed (i.e. all arrows pointing down with a rate of  $\lambda_1$  are removed). This diagram is used as a visualisation aid for the blocking time formula.

By the nature of this new Markov chain variation a similar recursive approach to the waiting time cannot be used here. Since both service completions and new arrivals can occur, the path of an individual from arrival to departure is not fixed. For example, for a particular Markov chain model with a threshold of  $T = 2$ , an individual arriving at state  $(2, 3)$  may have multiple different pathways. Both of these are valid paths:

- $(2, 3) \rightarrow (2, 2) \rightarrow (1, 2) \rightarrow (0, 2)$
- $(2, 3) \rightarrow (2, 4) \rightarrow (2, 3) \rightarrow (2, 2) \rightarrow (1, 2) \rightarrow (0, 2)$

Similar to equations (3.16) and (3.17) the expected time spent in each state here is denoted as:

$$c_b(u, v) = \begin{cases} \frac{1}{\min(v, C)\mu}, & \text{if } v \leq C \\ \frac{1}{\min(v, C)\mu + \lambda_1}, & \text{otherwise} \end{cases} \quad (3.48)$$

In equation (3.48), both service completions and type 1 arrivals are considered. Thus, from a blocked individual's perspective whenever the system moves from one state  $(u, v)$  to another state it can either be:

- because of a service being completed: we will denote the probability of this happening by  $p_s(u, v)$ .
- because of an arrival of an individual of type 1: denoting such probability by  $p_o(u, v)$ .

These probabilities are given by:

$$p_s(u, v) = \frac{\min(v, C)\mu}{\lambda_1 + \min(v, C)\mu}, \quad p_o(u, v) = \frac{\lambda_1}{\lambda_1 + \min(v, C)\mu} \quad (3.49)$$

The set of states where blocking can occur is defined as the *blocking states* and consists of all states  $(u, v)$  where  $u$  is non-zero. In essence, the set of blocking state  $S_b$  is defined as:

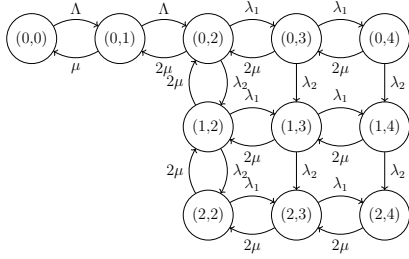
$$S_b = \{(u, v) \in S \mid u > 0\} \quad (3.50)$$

From Figure 3.7 the set  $S_b$  consists of all states below the first line of Markov chain. In addition, in order to not consider individuals that will be lost to the system, the set of accepting states needs to be taken into account. As defined in Section 3.4.1.1, the set of accepting states  $S_A^{(2)}$  is given by equation (3.22).

Having defined  $c_b(u, v)$  and  $S_b$  a formula for the expected blocking time at each state can be given by:

$$b(u, v) = \begin{cases} 0, & \text{if } (u, v) \notin S_b \\ c_b(u, v) + b(u-1, v), & \text{if } v = N = T \\ c_b(u, v) + b(u, v-1), & \text{if } v = N \neq T \\ c_b(u, v) + p_s(u, v)b(u-1, v) + p_o(u, v)b(u, v+1), & \text{if } u > 0 \text{ and } v = T \\ c_b(u, v) + p_s(u, v)b(u, v-1) + p_o(u, v)b(u, v+1), & \text{otherwise} \end{cases} \quad (3.51)$$

Unlike equation (3.23), equation (3.51) cannot be solved recursively. Only a direct approach will be used to solve this equation. By enumerating all possible equations generated by (3.51) for all states  $(u, v)$  that belong in  $S_b$  a system of linear equations arises where the unknown variables are all the  $b(u, v)$  terms. For instance, let us consider a Markov model where  $C = 2, T = 3, N = 6, M = 2$ . The Markov model is shown in Figure 3.8 and the equivalent equations are (3.52) - (3.57). The equations considered here are only the ones that correspond to the blocking states.



$$b(1, 2) = c_b(1, 2) + p_o b(1, 3) \quad (3.52)$$

$$b(1, 3) = c_b(1, 3) + p_s b(1, 2) + p_o b(1, 4) \quad (3.53)$$

$$b(1, 4) = c_b(1, 4) + b(1, 3) \quad (3.54)$$

$$b(2, 2) = c_b(2, 2) + p_s b(1, 2) + p_o b(2, 3) \quad (3.55)$$

$$b(2, 3) = c_b(2, 3) + p_s b(2, 2) + p_o b(1, 4) \quad (3.56)$$

$$b(2, 4) = c_b(2, 4) + b(2, 3) \quad (3.57)$$

Figure 3.8: Example of Markov model with  $C = 2, T = 3, N = 6, M = 2$ .

Additionally, the above equations can be transformed into a linear system of the form  $Zx = y$  where:

$$Z = \begin{pmatrix} -1 & p_o & 0 & 0 & 0 & 0 \\ p_s & -1 & p_o & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ p_s & 0 & 0 & -1 & p_o & 0 \\ 0 & 0 & 0 & p_s & -1 & p_o \\ 0 & 0 & 0 & 0 & 1 & -1 \end{pmatrix}, x = \begin{pmatrix} b(1, 2) \\ b(1, 3) \\ b(1, 4) \\ b(2, 2) \\ b(2, 3) \\ b(2, 4) \end{pmatrix}, y = \begin{pmatrix} -c_b(1, 2) \\ -c_b(1, 3) \\ -c_b(1, 4) \\ -c_b(2, 2) \\ -c_b(2, 3) \\ -c_b(2, 4) \end{pmatrix} \quad (3.58)$$

A more generalised form of the linear system of (3.58) can thus be given for any value of  $C, T, N, M$  by:

$$b(1, T) = c_b(1, T) + p_o b(1, T + 1) \quad (3.59)$$

$$b(1, T + 1) = c_b(1, T + 1) + p_s(1, T) + p_o b(1, T + 1) \quad (3.60)$$

$$b(1, T + 2) = c_b(1, T + 2) + p_s(1, T + 1) + p_o b(1, T + 3) \quad (3.61)$$

$$\vdots \quad (3.62)$$

$$b(1, N) = c_b(1, N) + b(1, N - 1) \quad (3.63)$$

$$b(2, T) = c_b(2, T) + p_s b(1, T) + p_o b(2, T + 1) \quad (3.64)$$

$$b(2, T + 1) = c_b(2, T + 1) + p_s b(2, T) + p_o b(2, T + 2) \quad (3.65)$$

$$\vdots \quad (3.66)$$

$$b(M, T) = c_b(M, T) + b(M, T - 1) \quad (3.67)$$

The equivalent matrix form of the linear system of equations (3.59) - (3.67) is given by  $Zx = y$ , where:

$$Z = \begin{pmatrix} -1 & p_o & 0 & \dots & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ p_s & -1 & p_o & \dots & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & p_s & -1 & \dots & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & -1 & 0 & 0 & 0 & \dots & 0 & 0 \\ p_s & 0 & 0 & \dots & 0 & 0 & -1 & p_o & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & p_s & -1 & p_o & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & 0 & \dots & 1 & -1 \end{pmatrix}, \quad (3.68)$$

$$x = \begin{pmatrix} b(1, T) \\ b(1, T + 1) \\ b(1, T + 2) \\ \vdots \\ b(1, N) \\ b(2, T) \\ b(2, T + 1) \\ \vdots \\ b(M, T) \end{pmatrix}, y = \begin{pmatrix} -c_b(1, T) \\ -c_b(1, T + 1) \\ -c_b(1, T + 2) \\ \vdots \\ -c_b(1, N) \\ -c_b(2, T) \\ -c_b(2, T + 1) \\ \vdots \\ -c_b(M, T) \end{pmatrix} \quad (3.69)$$

Thus, having calculated the mean blocking time  $b(u, v)$  for every blocking state individually, a similar formula to equation (3.23) can be derived. The resultant blocking time formula is given by:

$$B = \frac{\sum_{(u,v) \in S_A} \pi(u,v) b(\mathcal{A}_2(u, v))}{\sum_{(u,v) \in S_A} \pi(u,v)} \quad (3.70)$$

Note here that  $\pi(u, v)$  is the steady state probability that the Markov chain model is at state  $(u, v)$  described in Section 3.3.1.

### 3.4.2.1 Implementation

The mean blocking time is only calculated using a direct approach similar to the one described in Section 3.4.1.2.1. Since this implementation is the same as the waiting time one, the code snippet shown in 3.25 shows only the usage of the function rather than the function itself.

```
>>> import ambulance_game as abg
>>> import numpy as np
>>>
>>> all_states = abg.markov.build_states(
...     threshold=2,
...     system_capacity=4,
...     buffer_capacity=3,
... )
>>> Q = abg.markov.get_transition_matrix(
...     lambda_1=1,
...     lambda_2=1,
...     mu=4,
...     num_of_servers=1,
...     threshold=2,
...     system_capacity=4,
...     buffer_capacity=3
... )
>>> pi = abg.markov.get_markov_state_probabilities(
...     abg.markov.get_steady_state_algebraically(
...         Q, algebraic_function=np.linalg.solve
...     ),
...     all_states,
... )
>>> round(abg.markov.mean_blocking_time_formula_using_direct_approach(
...     all_states=all_states,
...     pi=pi,
...     lambda_1=1,
...     mu=4,
...     num_of_servers=1,
...     threshold=2,
...     system_capacity=4,
...     buffer_capacity=3,
... ), 10)
0.1287843179
```

Code snippet 3.25: Usage of the function to calculate the mean blocking time.

### 3.4.3 Proportion of individuals within target

Another performance measure that needs to be taken into consideration is the proportion of individuals whose waiting and service times are within a specified time target. In order to consider such a measure though one would need to obtain the distribution of time in the system for all individuals. The complexity of such a task comes from the fact that different individuals arrive at different states of the Markov model. Consider the case when an arrival occurs when the model is at a specific state.

#### 3.4.3.1 Distribution of time at a specific state (with 1 server)

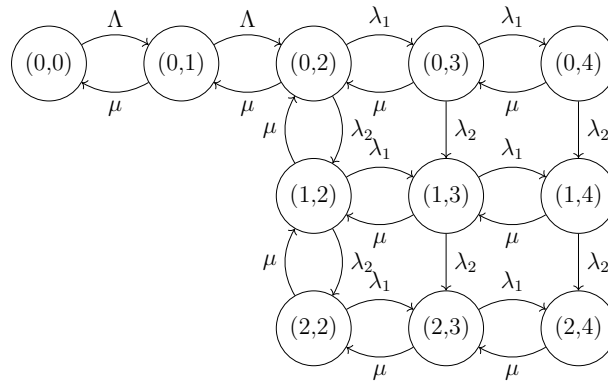


Figure 3.9: Example of Markov model with  $C = 1, T = 2, N = 4, M = 2$

Consider the Markov model of Figure 3.9 with one server (i.e. the rate of service completion is  $\mu$  throughout the Markov model) and a threshold of two individuals. Assume that a type 1 individual arrives when the model is at state  $(0, 3)$ , thus forcing the model to move to state  $(0, 4)$ . The distribution of the time needed for the specified individual to exit the system from state  $(0, 4)$  is given by the sum of exponentially distributed random variables with the same parameter  $\mu$ . The sum of such random variables form the Erlang distribution which is defined by the number of random variables  $k$  that are added together and their exponential parameter  $\mu$ .

$$\begin{aligned}
 X_i &\sim \text{Exp}(\mu) \\
 X_1 + X_2 + \dots + X_k &\sim \text{Erlang}(k, \mu)
 \end{aligned}
 \tag{3.71}$$

Note here that these random variables represent the individual's pathway from the perspective of the individual. Thus,  $X_i$  represents the random variable of the



time that it takes for an individual to move from the  $i^{\text{th}}$  position of the queue to the  $(i - 1)^{\text{th}}$  position (i.e. for someone in front of them to finish their service) and  $X_0$  is the time it takes that individual from starting their service to exiting the system.

$$\begin{aligned}
 (0, 4) &\Rightarrow X_3 \sim \text{Exp}(\mu) \\
 (0, 3) &\Rightarrow X_2 \sim \text{Exp}(\mu) \\
 (0, 2) &\Rightarrow X_1 \sim \text{Exp}(\mu) \\
 (0, 1) &\Rightarrow X_0 \sim \text{Exp}(\mu) \\
 S = X_3 + X_2 + X_1 + X_0 &= \text{Erlang}(4, \mu)
 \end{aligned} \tag{3.72}$$

Thus, the waiting and service time of an individual in the model of Figure 3.9 can be captured by an Erlang distributed random variable. The general CDF of the Erlang distribution  $\text{Erlang}(k, \mu)$  is given by:

$$P(S < t) = 1 - \sum_{i=0}^{k-1} \frac{1}{i!} e^{-\mu t} (\mu t)^i \tag{3.73}$$

Unfortunately, the Erlang distribution can only be used for the sum of identically distributed random variables from the exponential distribution. Therefore, this approach cannot be used when one of the random variables has a different parameter than the others. In fact the only case where this can be use is only when the number of servers are  $C = 1$ , similar to the explored example, or when an individual arrives and goes straight to service (i.e. when there is no other individual waiting and there is an empty server).

### 3.4.3.2 Distribution of time at a specific state (with multiple servers)

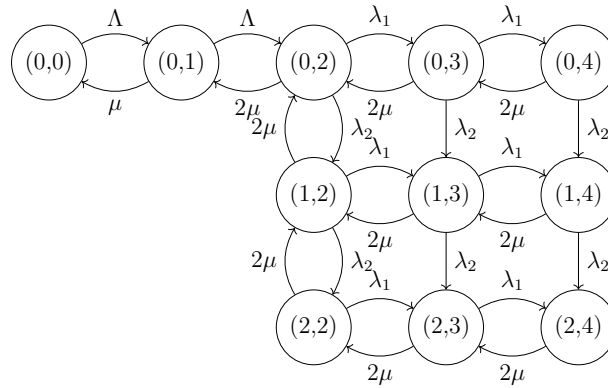


Figure 3.10: Example of Markov model with  $C = 2, T = 2, N = 4, M = 2$

Figure 3.10 represents the same Markov model as Figure 3.9 with the only exception that there are 2 servers here. By applying the same logic, assuming that an individual arrives at state  $(0, 4)$ , the sum of the following random variables arises.

$$\begin{aligned}
 (0, 4) &\Rightarrow X_2 \sim \text{Exp}(2\mu) \\
 (0, 3) &\Rightarrow X_1 \sim \text{Exp}(2\mu) \\
 (0, 2) &\Rightarrow X_0 \sim \text{Exp}(\mu)
 \end{aligned} \tag{3.74}$$

Since these exponentially distributed random variables do not share the same parameter, an Erlang distribution cannot be used. In fact, the problem can now be viewed as the sum of exponentially distributed random variables with different parameters, which is in turn the sum of Erlang distributed random variables. The sum of Erlang distributed random variables is said to follow the hypoexponential distribution. The hypoexponential distribution is defined with two vectors of size equal to the number of Erlang random variables that are added together [5, 127]. For this particular example:

$$\left. \begin{aligned}
 X_2 &\sim \text{Exp}(2\mu) \\
 X_1 &\sim \text{Exp}(2\mu) \\
 X_0 &\sim \text{Exp}(\mu) \Rightarrow X_0 = S_2 \sim \text{Erlang}(1, \mu)
 \end{aligned} \right\} \begin{aligned}
 X_1 + X_2 &= S_1 \sim \text{Erlang}(2, 2\mu) \\
 S_1 + S_2 &= H \sim \text{Hypo}((2, 1), (2\mu, \mu))
 \end{aligned} \tag{3.75}$$

The random variable  $H$  from equation (3.75) follows a hypoexponential distribu-

tion with two vector parameters  $((2, 1)$  and  $(2\mu, \mu)$ ). The CDF of this distribution can be therefore used to get the probability of the time spent in the system being less than a given target. The CDF of the general hypoexponential distribution  $Hypo(\vec{r}, \vec{\lambda})$ , is given by the (3.76), where vector  $\vec{r}$  contains all  $k$ -values of the Erlang distributions defined in (3.71) and  $\vec{\lambda}$  is a vector of the distinct parameters [46].

$$P(H < t) = 1 - \left( \prod_{j=1}^{|\vec{r}|} \lambda_j^{r_j} \right) \sum_{k=1}^{|\vec{r}|} \sum_{l=1}^{r_k} \frac{\Psi_{k,l}(-\lambda_k) t^{r_k-l} e^{-\lambda_k t}}{(r_k - l)!(l-1)!}$$

**where**  $\Psi_{k,l}(t) = -\frac{\partial^{l-1}}{\partial t^{l-1}} \left( \prod_{j=0, j \neq k}^{|\vec{r}|} (\lambda_j + t)^{-r_j} \right)$

**and**  $\lambda_0 = 0, r_0 = 1$  (3.76)

The computation of the derivative makes equation (3.76) computationally expensive. In [87] an alternative linear version of that CDF is explored via matrix analysis, and is given by the following formula:

$$F(x) = 1 - \sum_{k=1}^n \sum_{l=0}^{k-1} (-1)^{k-1} \binom{n}{k} \binom{k-1}{l} \sum_{j=1}^n \sum_{s=1}^{j-1} e^{-x\lambda_s} \prod_{l=1}^{s-1} \left( \frac{\lambda_l}{\lambda_l - \lambda_s} \right)^{k_s}$$

$$\times \sum_{s < a_1 < \dots < a_{l-1} < j} \left( \frac{\lambda_s}{\lambda_s - \lambda_{a_1}} \right)^{k_s} \prod_{m=s+1}^{a_1-1} \left( \frac{\lambda_m}{\lambda_m - \lambda_{a_1}} \right)^{k_m} \prod_{n=a_1}^{a_2-1} \left( \frac{\lambda_n}{\lambda_n - \lambda_{a_2}} \right)^{k_n}$$

$$\dots \prod_{r=a_{l-1}}^{j-1} \left( \frac{\lambda_r}{\lambda_r - \lambda_{a_j}} \right)^{k_r} \sum_{q=0}^{k_s-1} \frac{((\lambda_s - \lambda_{a_1})x)^q}{q!},$$

for  $x \geq 0$  (3.77)

Although equation (3.77) is a simplified version of equation (3.76) it still has some unnecessary complexity. The described expressions are general expressions used to get the CDF of the hypoexponential distribution, which is in turn the sum of multiple Erlang distributed random variables. However, the random variable  $H$ , described in (3.75), is the sum of only two different erlang distributed random variables. Thus, perhaps a more simplified version of the above expressions can be derived that is specific to the case of two Erlang distributed random variables.

### 3.4.3.3 Specific CDF of hypoexponential distribution

Equations (3.76) and (3.77) refer to the general CDF of the hypoexponential distribution where the size of the vector parameters can be of any size [46]. In the Markov chain models described in Figures 3.9 and 3.10 the parameter vectors of the hypoexponential distribution are of size two, and in fact, for any possible version of the investigated Markov chain model the vectors can only be of size two. This is true since for any dimensions of this Markov chain model there will always be at most two distinct exponential parameters; the parameter for finishing a service ( $\mu$ ) and the parameter for moving forward in the queue ( $C\mu$ ). For the unique case of  $C = 1$  the hypoexponential distribution will not be used as this is equivalent to an Erlang distribution. Therefore, by fixing the sizes of  $\vec{r}$  and  $\vec{\lambda}$  to 2, the following specific expression for the CDF of the hypoexponential distribution arises, where the derivative is removed:

$$\begin{aligned}
 P(H < t) &= 1 - \left( \prod_{j=1}^{|\vec{r}|} \lambda_j^{r_j} \right) \sum_{k=1}^{|\vec{r}|} \sum_{l=1}^{r_k} \frac{\Psi_{k,l}(-\lambda_k) t^{r_k-l} e^{-\lambda_k t}}{(r_k - l)!(l - 1)!} \\
 \text{where} \quad \Psi_{k,l}(t) &= \begin{cases} \frac{(-1)^l (l-1)!}{\lambda_2} \left[ \frac{1}{t^l} - \frac{1}{(t+\lambda_2)^l} \right], & k = 1 \\ -\frac{1}{t(t+\lambda_1)^{r_1}}, & k = 2 \end{cases} \\
 \text{and} \quad \lambda_0 = 0, r_0 = 1 & \quad (3.78)
 \end{aligned}$$

Note here that the only difference between equation (3.76) and (3.78) is the  $\Psi$ , where it is now only computed for  $k = 1, 2$ . The following subsection proves the following expression:

$$-\frac{\partial^{l-1}}{\partial t^{l-1}} \left( \prod_{j=0, j \neq k}^{|\vec{r}|} (\lambda_j + t)^{-r_j} \right) = \begin{cases} \frac{(-1)^l (l-1)!}{\lambda_2} \left[ \frac{1}{t^l} - \frac{1}{(t+\lambda_2)^l} \right], & k = 1 \\ -\frac{1}{t(t+\lambda_1)^{r_1}}, & k = 2 \end{cases} \quad (3.79)$$

### 3.4.3.4 Proof of specific hypoexponential distribution (eq. (3.79))

This section aims to show that there exists a simplified expression of equation (3.76) that is specific to the proposed Markov model. Function  $\Psi$  is defined using the parameter  $t$  and the variables  $k$  and  $l$ . Given the Markov model, the range of values that  $k$  and  $l$  can take can be bounded. First, from the range of the double summation in equation (3.76), it can be seen that  $k = 1, 2, \dots, |\vec{r}|$ . Now,  $|\vec{r}|$  represents the size of the parameter vectors that, for the Markov model, will always be 2. That is because, for all the exponentially distributed random

variables that are added together to form the new distribution, there are only two distinct parameters, thus forming two erlang distributions. Therefore:

$$k = 1, 2$$

By observing equation (3.76) once more, the range of values that  $l$  takes are  $l = 1, 2, \dots, r_k$ , where  $r_1$  is subject to the individual's position in the queue and  $r_2 = 1$ . In essence, the hypoexponential distribution will be used with these bounds:

$$\begin{aligned} k = 1 & \Rightarrow l = 1, 2, \dots, r_1 \\ k = 2 & \Rightarrow l = 1 \end{aligned} \tag{3.80}$$

Thus the left hand side of equation (3.79) needs only to be defined for these bounds. The specific hypoexponential distribution investigated here is of the form:

$$Hypo((r_1, 1)(\lambda_1, \lambda_2))$$

Note the initial conditions  $\lambda_0 = 0, r_0 = 1$  defined in equation (3.76) also hold here. Thus the proof is split into two parts, for  $k = 1$  and  $k = 2$ .

- $k = 2, l = 1$

$$\begin{aligned} LHS &= -\frac{\partial^{1-1}}{\partial t^{1-1}} \left( \prod_{j=0, j \neq 2}^2 (\lambda_j + t)^{-r_j} \right) \\ &= -((\lambda_0 + t)^{-r_0} \times (\lambda_1 + t)^{-r_1}) \\ &= -(t^{-1} \times (\lambda_1 + t)^{-r_1}) \\ &= -\frac{1}{t(t + \lambda_1)^{r_1}} \end{aligned}$$

□

- $k = 1, l = 1, \dots, r_1$

$$\begin{aligned}
LHS &= -\frac{\partial^{l-1}}{\partial t^{l-1}} \left( \prod_{j=0, j \neq 1}^2 (\lambda_j + t)^{-r_j} \right) \\
&= -\frac{\partial^{l-1}}{\partial t^{l-1}} ((\lambda_0 + t)^{-r_0} \times (\lambda_2 + t)^{-r_2}) \\
&= -\frac{\partial^{l-1}}{\partial t^{l-1}} \left( \frac{1}{t(t + \lambda_2)} \right)
\end{aligned}$$

In essence, the final part of the proof is to show that:

$$-\frac{\partial^{l-1}}{\partial t^{l-1}} \left( \frac{1}{t(t + \lambda_2)} \right) = \frac{(-1)^l (l-1)!}{\lambda_2} \left[ \frac{1}{t^l} - \frac{1}{(t + \lambda_2)^l} \right]$$

**Proof by Induction:**

1. Base case ( $l = 1$ ):

$$\begin{aligned}
LHS &= -\frac{\partial^{1-1}}{\partial t^{1-1}} \left( \frac{1}{t(t + \lambda_2)} \right) = -\frac{1}{t(t + \lambda_2)} \\
RHS &= \frac{(-1)^1 (1-1)!}{\lambda_2} \left[ \frac{1}{t^1} - \frac{1}{(t + \lambda_2)^1} \right] \\
&= -\frac{t + \lambda_2 - t}{\lambda_2 t(t + \lambda_2)} \\
&= -\frac{1}{t(t + \lambda_2)} \\
LHS &= RHS
\end{aligned}$$

2. Assume true for  $l = x$ :

$$-\frac{\partial^{x-1}}{\partial t^{x-1}} \left( \frac{1}{t(t + \lambda_2)} \right) = \frac{(-1)^x (x-1)!}{\lambda_2} \left[ \frac{1}{t^x} - \frac{1}{(t + \lambda_2)^x} \right]$$

3. Prove true for  $l = x + 1$ :

$$\left( \text{Show that: } \frac{\partial^x}{\partial t^x} \left( \frac{1}{t(t + \lambda_2)} \right) = \frac{(-1)^{x+1} (x)!}{\lambda_2} \left[ \frac{1}{t^{x+1}} - \frac{1}{(t + \lambda_2)^{x+1}} \right] \right)$$

$$\begin{aligned}
LHS &= \frac{\partial}{\partial t} \left[ \frac{\partial^{x-1}}{\partial t^{x-1}} \left( \frac{-1}{t(t + \lambda_2)} \right) \right] \\
&= \frac{\partial}{\partial t} \left[ \frac{(-1)^x (x-1)!}{\lambda_2} \left( \frac{1}{t^x} - \frac{1}{(t + \lambda_2)^x} \right) \right] \\
&= \frac{(-1)^x (x-1)!}{\lambda_2} \left( \frac{(-x)}{t^{x+1}} - \frac{(-x)}{(t + \lambda_2)^x} \right) \\
&= \frac{(-1)^x (x-1)! (-x)}{\lambda_2} \left( \frac{1}{t^{x+1}} - \frac{1}{(t + \lambda_2)^x} \right) \\
&= \frac{(-1)^{x+1} (x)!}{\lambda_2} \left( \frac{1}{t^{x+1}} - \frac{1}{(t + \lambda_2)^x} \right) \\
&= RHS
\end{aligned}$$

□

### 3.4.3.5 Proportion within target for type 1 and type 2 individuals

Given the two CDFs of the Erlang and Hypoexponential distributions (equations (3.71) and (3.78)) a new function has to be defined to decide which one to use between the two. Based on the state of the model, there can be three scenarios when an individual arrives.

1. There is a free server and the individual does not have to wait

$$X_{(u,v)} \sim \text{Erlang}(1, \mu)$$

2. The individual arrives at the queue at the  $n^{\text{th}}$  position and the model has  $C > 1$  servers

$$X_{(u,v)} \sim \text{Hypo}((n, 1), (C\mu, \mu))$$

3. The individual arrives at a queue at the  $n^{\text{th}}$  position and the model has  $C = 1$  servers

$$X_{(u,v)} \sim \text{Erlang}(n + 1, \mu)$$

Note here that for the first case  $\text{Erlang}(1, \mu)$  is equivalent to  $\text{Exp}(\mu)$ . Define  $X_{(u,v)}^{(1)}$  as the distribution of type 1 individuals and  $X_{(u,v)}^{(2)}$  as the distribution of type 2 individuals, when arriving at state  $(u, v)$  of the model.

$$X_{(u,v)}^{(1)} \sim \begin{cases} \text{Erlang}(v, \mu), & \text{if } C = 1 \text{ and } v > 1 \\ \text{Hypo}(\vec{r} = (v - C, 1), \vec{\lambda} = (C\mu, \mu)), & \text{if } C > 1 \text{ and } v > C \\ \text{Erlang}(1, \mu), & \text{if } v \leq C \end{cases} \quad (3.81)$$

$$X_{(u,v)}^{(2)} \sim \begin{cases} \text{Erlang}(\min(v, T), \mu), & \text{if } C = 1 \text{ and } v, T > 1 \\ \text{Hypo}(\vec{r} = (\min(v, T) - C, 1), \vec{\lambda} = (C\mu, \mu)), & \text{if } C > 1 \text{ and } v, T > C \\ \text{Erlang}(1, \mu), & \text{if } v \leq C \text{ or } T \leq C \end{cases} \quad (3.82)$$

Equations (3.73) and (3.78) can now be used. Therefore, the probability that an individual arriving at a specific state is within a given time target  $t$  is given by the following formulas:

$$P(X_{(u,v)}^{(1)} < t) = \begin{cases} 1 - \sum_{i=0}^{v-1} \frac{1}{i!} e^{-\mu t} (\mu t)^i, & \text{if } C = 1 \\ & \text{and } v > 1 \\ 1 - (\mu C)^{v-C} \mu \sum_{k=1}^{|\vec{r}|} \sum_{l=1}^{r_k} \frac{\Psi_{k,l}(-\lambda_k) t^{r_k-l} e^{-\lambda_k t}}{(r_k-l)!(l-1)!}, & \text{if } C > 1 \\ \text{where } \vec{r} = (v - C, 1) \text{ and } \vec{\lambda} = (C\mu, \mu) & \text{and } v > C \\ 1 - e^{-\mu t}, & \text{if } v \leq C \end{cases} \quad (3.83)$$

$$P(X_{(u,v)}^{(2)} < t) = \begin{cases} 1 - \sum_{i=0}^{\min(v,T)-1} \frac{1}{i!} e^{-\mu t} (\mu t)^i, & \text{if } C = 1 \\ & \text{and } v, T > 1 \\ 1 - (C\mu)^{\min(v,T)-C} \mu \sum_{k=1}^{|\vec{r}|} \sum_{l=1}^{r_k} \frac{\Psi_{k,l}(-\lambda_k) t^{r_k-l} e^{-\lambda_k t}}{(r_k-l)!(l-1)!}, & \text{if } C > 1 \\ \text{where } \vec{r} = (\min(v, T) - C, 1) \text{ and } \vec{\lambda} = (C\mu, \mu) & \text{and } v, T > C \\ 1 - e^{-\mu t}, & \text{if } v \leq C \\ & \text{or } T \leq C \end{cases} \quad (3.84)$$

In addition the set of accepting states for type 1 ( $S_A^{(1)}$ ) and type 2 ( $S_A^{(2)}$ ) individuals defined in equations (3.21) and (3.22) are also needed here. Note here that,  $S$  denotes the set of all states of the Markov chain model.

Having defined everything, a formula similar to the ones of equations (3.23) and (3.58) can be generated. The following formula uses the state probability



vector  $\pi$  to get the weighted average of the probability below target of all states in the Markov model.

$$P(X^{(1)} < t) = \frac{\sum_{(u,v) \in S_A^{(1)}} P(X_{u,v}^{(1)} < t) \pi_{u,v}}{\sum_{(u,v) \in S_A^{(1)}} \pi_{u,v}} \quad (3.85)$$

$$P(X^{(2)} < t) = \frac{\sum_{(u,v) \in S_A^{(2)}} P(X_{u,v}^{(2)} < t) \pi_{u,v}}{\sum_{(u,v) \in S_A^{(2)}} \pi_{u,v}} \quad (3.86)$$

### 3.4.3.6 Overall proportion within target

The overall proportion of individuals for both type 1 and type 2 individuals is given by the equivalent formula of equations (3.24) and (3.26). The following formula uses the probability of lost individuals from both types to get the weighted sum of the two already existing probabilities.

$$P(L'_1) = \sum_{(u,v) \in S_A^{(1)}} \pi(u,v), \quad P(L'_2) = \sum_{(u,v) \in S_A^{(2)}} \pi(u,v)$$

$$P(X < t) = \frac{\lambda_1 P(L'_1)}{\lambda_2 P(L'_2) + \lambda_1 P(L'_1)} P(X^{(1)} < t) + \frac{\lambda_2 P(L'_2)}{\lambda_2 P(L'_2) + \lambda_1 P(L'_1)} P(X^{(2)} < t) \quad (3.87)$$

### 3.4.3.7 Implementation

This section focuses on the implementation of all necessary equations to calculate the proportion of individuals within target as described in Section 3.4.3. The first equation to be considered is the simplified version of  $\Psi_{k,\lambda}(t)$  described in equation (3.78). Code snippet 3.26 shows the implementation of this equation in python.

```
>>> def specific_psi_function(
...     arg, k, l, exp_rates, freq, a
... ):
...     """
...     The specific version of the Psi function that is used for the
...     purpose of this study. Due to the way the hypoexponential cdf
...     works the function is called only for values of k=1 and k=2.
...     For these values the following hold:
...         - k = 1 -> l = 1, ..., n
...         - k = 2 -> l = 1
...     """
...     if k == 1:
...         psi_val = (1 / (arg**l)) - (1 / (arg + exp_rates[2]) ** l)
```

```

...     psi_val *= (-1) ** l * math.factorial(l - 1) / exp_rates[2]
...     return psi_val
...     if k == 2:
...         psi_val = -1 / (arg * (arg + exp_rates[1]) ** freq[1])
...         return psi_val
...     return 0

```

Code snippet 3.26: Function for the simplified version of  $\Psi_{k,\lambda}(t)$ 

The piece of code shown in 3.27 returns the cumulative distribution function of the hypoexponential distribution. In essence this is the value of  $P(H < t)$  outlined in equation (3.78).

```

>>> def hypoexponential_cdf(
...     x, exp_rates, freq, psi_func=specific_psi_function
... ):
...     """
...     The function represents the cumulative distribution function of the
...     hypoexponential distribution. It calculates the probability that a
...     hypoexponentially distributed random variable has a value less than
...     x. In other words calculate  $P(S < x)$  where  $S \sim \text{Hypo}(\lambda, r)$ 
...     where: lambda is a vector with distinct exponential parameters and
...     r is a vector with the frequency of each distinct parameter
...     Note that: a Hypoexponentially distributed random variable can be
...     described as the sum of Erlang distributed random variables
...     Parameters
...     -----
...     x : float
...         The target we want to calculate the probability for
...     exp_rates : tuple
...         The distinct exponential parameters
...     freq : tuple
...         The frequency of the exponential parameters
...     psi_func : function, optional
...         The function to be used to get Psi, by default
...         specific_psi_function
...     Returns
...     -----
...     float
...          $P(S < x)$  where  $S \sim \text{Hypo}(\lambda, r)$ 
...     """
...     a = len(exp_rates)
...     exp_rates = (0,) + exp_rates
...     freq = (1,) + freq
...     summation = 0
...     for k in range(1, a + 1):
...         for l in range(1, (freq[k] + 1)):
...             psi = psi_func(
...                 arg=-exp_rates[k],
...                 k=k,
...                 l=l,
...                 exp_rates=exp_rates,
...                 freq=freq,
...                 a=a,
...             )
...             iteration = (
...                 psi * (x ** (freq[k] - 1)) * np.exp(-exp_rates[k] * x)
...             )

```

```

...         iteration /= (
...             np.math.factorial(freq[k] - 1)*np.math.factorial(1 - 1))
...         summation += float(iteration)
...     output = 1 - (
...         product_of_all_elements(
...             [exp_rates[j] ** freq[j] for j in range(1, a + 1)]
...         ) * summation
...     )
...     return output

```

Code snippet 3.27: Function for the cumulative distribution of the Hypoexponential distribution

Similarly the cumulative distribution function of the erlang distribution is also needed here. The code snippet in 3.28 shows the implementation of this function as described in equation (3.73).

```

>>> def erlang_cdf(mu, n, x):
...     """
...     Cumulative distribution function of the erlang distribution.
...      $P(X < x)$  where  $X \sim \text{Erlang}(\mu, n)$ 
...     Parameters
...     -----
...     mu : float
...         The parameter of the Erlang distribution
...     n : int
...         The number of Exponential distributions that are added together
...     x : float
...         The argument of the function
...     Returns
...     -----
...     float
...         The probability that the erlang distributed r.v. is less than x
...     """
...     return 1 - np.sum([
...         np.math.exp(-mu * x) * (mu * x) ** i
...         * (1 / np.math.factorial(i))
...         for i in range(n)
...     ])

```

Code snippet 3.28: Function for the cumulative distribution of the Erlang distribution

Having defined all functions necessary the code snippet in 3.29 chooses which of the two distributions to use to calculate the probability of an individual being within a given time target.

```

>>> def get_probability_of_waiting_time_in_system_less_than_target_for_state(
...     state,
...     class_type,
...     mu,
...     num_of_servers,
...     threshold,
...     target,
...     psi_func=specific_psi_function,

```

```

... ):
...     """
...     The function decides what probability distribution to use based on
...     the state we are currently on and the class type given. The two
...     distributions that are used are the Erlang and the Hypoexponential
...     distribution. The time it takes the system to exit a state and enter
...     the next one is known to be exponentially distributed. The sum of
...     exponentially distributed random variables is known to result in
...     either an Erlang distribution or a Hypoexponential distribution
...     The function works as follows:
...     - Checks whether the arriving individual will have to wait
...     - Finds the total number of states an individual will have to visit
...     - Depending on whether the parameters of the distributions to sum are
...     the same or not, call the appropriate cdf function.
...     Parameters
...     -----
...     state : tuple
...     class_type : int
...     mu : float
...     num_of_servers : int
...     threshold : int
...     target : int
...     psi_func : function, optional
...     Returns
...     -----
...     float
...     The probability of spending less time than the target in the
...     system when the individual has arrived at a given state
...     """
...     if class_type == 0:
...         arrive_on_waiting_space = state[1] > num_of_servers
...         rep = state[1] - num_of_servers
...     elif class_type == 1:
...         arrive_on_waiting_space = (
...             state[1] > num_of_servers and threshold > num_of_servers
...         )
...         rep = min(state[1], threshold) - num_of_servers
...     else:
...         raise ValueError("Class_type must be 0 or 1")
...
...     if arrive_on_waiting_space:
...         if num_of_servers == 1:
...             prob = erlang_cdf(mu=mu, n=rep + 1, x=target)
...         else:
...             param = num_of_servers * mu
...             prob = hypoexponential_cdf(
...                 x=target,
...                 exp_rates=(param, mu),
...                 freq=(rep, 1),
...                 psi_func=psi_func,
...             )
...     else:
...         prob = erlang_cdf(mu=mu, n=1, x=target)
...     return prob

```

Code snippet 3.29: Function for deciding which distribution to use to calculate the probability of an individual being within a given time target.

Finally, putting everything together, going through all the states of a given Markov chain model, the function defined in 3.30 calculates the probability of spending less time than the target in the system for a given individual type. This corresponds to both equations (3.85) and (3.86).

```
>>> def get_proportion_of_individuals_within_time_target(
...     all_states,
...     pi,
...     class_type,
...     mu,
...     num_of_servers,
...     threshold,
...     system_capacity,
...     buffer_capacity,
...     target,
...     psi_func=specific_psi_function,
...     **kwargs,
... ):
...     """
...     Gets the probability that a certain class of individuals is within a
...     given time target. This functions runs for every state the function
...     get_probability_of_waiting_time_in_system_less_than_target_for_state
...     and by using the state probabilities to get the average proportion of
...     individuals within target.
...     Parameters
...     -----
...     all_states : list
...     pi : numpy.array
...     class_type : int
...     mu : float
...     num_of_servers : int
...     threshold : int
...     system_capacity : int
...     buffer_capacity : int
...     target : float
...     psi_func : function, optional
...     Returns
...     -----
...     float
...     The probability of spending less time than the target in the
...     system
...     """
...     proportion_within_limit, probability_of_accepting = 0, 0
...     for (u, v) in all_states:
...         if abg.markov.utils.is_accepting_state(
...             state=(u, v),
...             class_type=class_type,
...             threshold=threshold,
...             system_capacity=system_capacity,
...             buffer_capacity=buffer_capacity,
...         ):
...             arriving_state = (u, v + 1)
...             if class_type == 1 and v >= threshold:
...                 arriving_state = (u + 1, v)
...
...             proportion_within_limit_at_state = (
...                 get_probability_of_waiting_time_in_system_less_than_target_for_state(
```

```

...             state=arriving_state,
...             class_type=class_type,
...             mu=mu,
...             num_of_servers=num_of_servers,
...             threshold=threshold,
...             target=target,
...             psi_func=psi_func,
...         )
...     )
...     proportion_within_limit += (
...         pi[u, v] * proportion_within_limit_at_state
...     )
...     probability_of_accepting += pi[u, v]
...     return proportion_within_limit / probability_of_accepting

```

Code snippet 3.30: Function for calculating the probability of spending less time than the target in the system for a given individual type.

Using all functions created so far, the proportion of individuals within target can be calculated for a given Markov chain model and a given individual type.

```

>>> import ambulance_game as abg
>>> import numpy as np
>>> all_states = abg.markov.build_states(
...     threshold=2,
...     system_capacity=4,
...     buffer_capacity=3,
... )
>>> Q = abg.markov.get_transition_matrix(
...     lambda_1=1,
...     lambda_2=1,
...     mu=4,
...     num_of_servers=1,
...     threshold=2,
...     system_capacity=4,
...     buffer_capacity=3
... )
>>> pi = abg.markov.get_markov_state_probabilities(
...     abg.markov.get_steady_state_algebraically(
...         Q, algebraic_function=np.linalg.solve
...     ), all_states
... )
>>> round(get_proportion_of_individuals_within_time_target(
...     all_states=all_states,
...     pi=pi,
...     class_type=0,
...     mu=4,
...     num_of_servers=1,
...     threshold=2,
...     system_capacity=3,
...     buffer_capacity=4,
...     target=1
... ), 10)
0.9190401179

```

Code snippet 3.31: Combining all functions to calculate the proportion of type 1 individuals within a time target of 1 time unit.

This shows that for the given set of parameters and for type 1 individuals the probability of spending less than 1 unit of time in the system is 91.9%.

## 3.5 Numeric results and timings

### 3.5.1 Markov chain waiting time approaches comparison

In Section 3.4.1 three different approaches for calculating the waiting time using the Markov chain model have been introduced. The three approaches are the recursive approach (Section 3.4.1.1), the direct approach (Section 3.4.1.2) and the closed form approach (Section 3.4.1.3). In this section the three approaches are compared in terms of accuracy and computation time.

In terms of accuracy the three approaches get close to identical results. Figures 3.11 and 3.12 show the differences of the three approaches for different values of  $N$  and  $M$ .

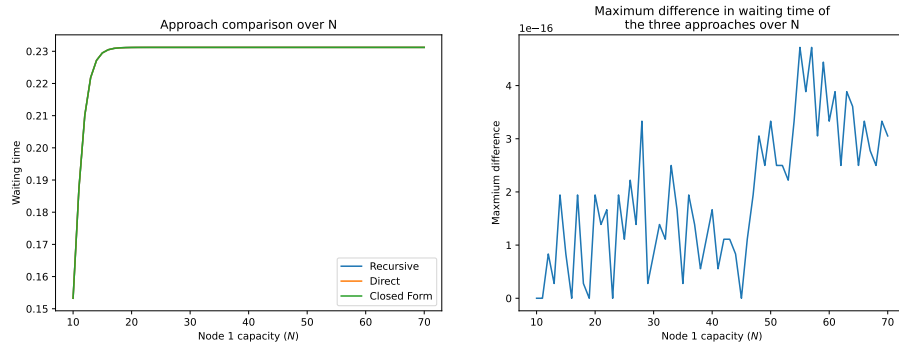


Figure 3.11: Waiting times of the three waiting time approaches for different values of  $N$  (left) and the maximum difference in waiting time among the three approaches over different values of  $N$  (right).

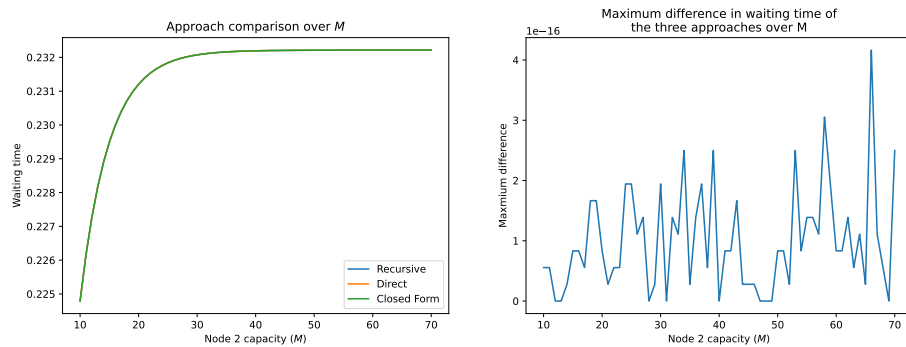


Figure 3.12: Waiting times of the three waiting time approaches for different values of  $M$  (left) and the maximum difference in waiting time among the three approaches over different values of  $M$  (right).

Since the results of the three approaches are almost identical, the computation time of the three approaches is the main factor that determines which approach will be used. Note that the right plots of Figures 3.11 and 3.12 have a y-axis scale of  $10^{-16}$ . Figures 3.13, 3.14 and 3.15 show the computation time needed to calculate the waiting time for different values of  $N$  and  $M$ . The numbers on these plots were generated by running each method 50 times on each value of  $N$  and  $M$  and taking the average computation time. These experiments were run on a computer with an Intel Core i7-1165G7 CPU running on four cores at 2.80 GHz and 16 GB of RAM.

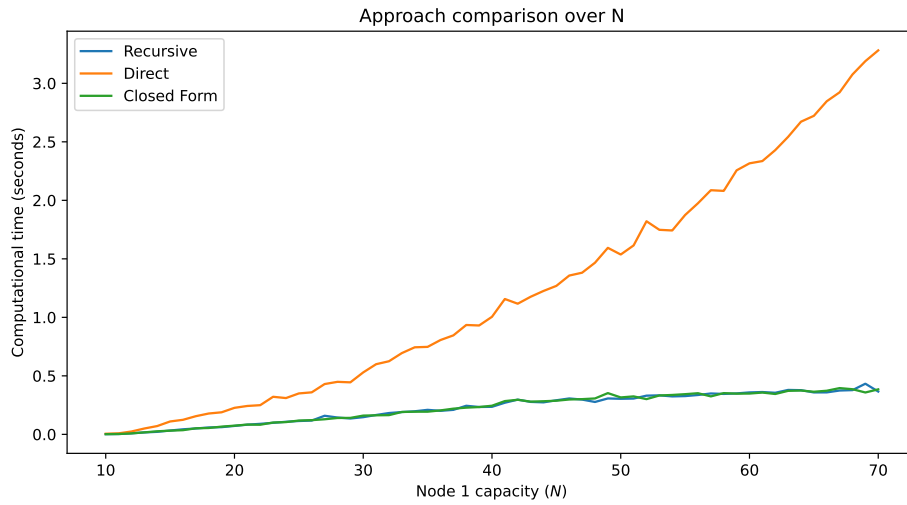


Figure 3.13: Computation time of the recursive, direct and closed-form waiting time approaches for different values of  $N$ .

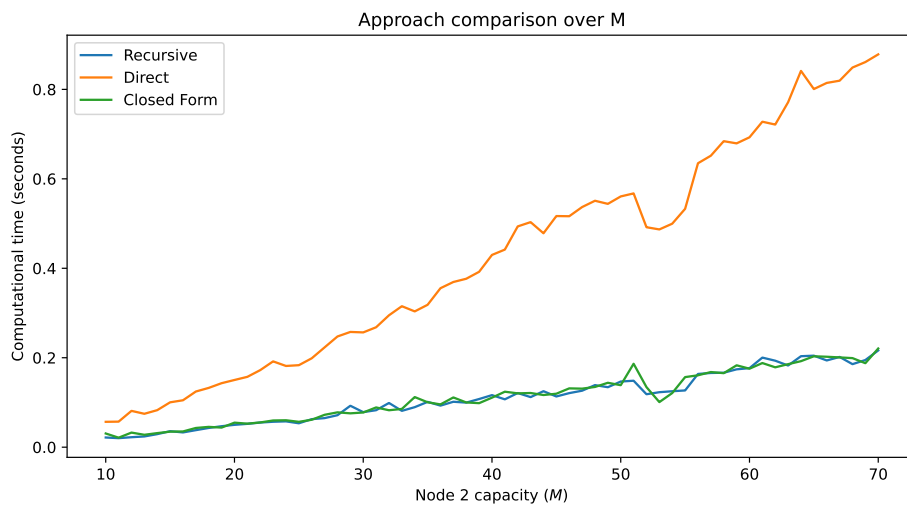


Figure 3.14: Computation time of the recursive, direct and closed-form waiting time approaches for different values of  $M$ .



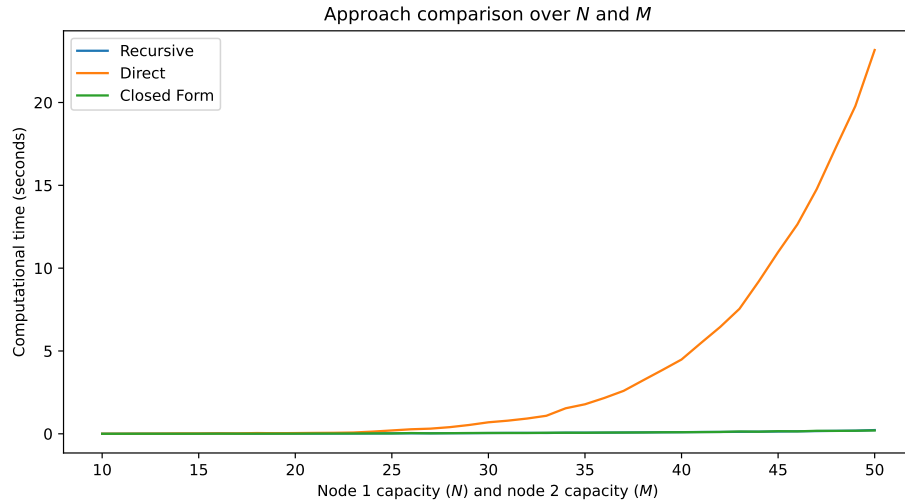


Figure 3.15: Computation time of the recursive, direct and closed-form waiting time approaches for different values of  $N$  and  $M$ .

It can be seen from Figures 3.13, 3.14 and 3.15 that while the recursive and closed form approaches appear to grow linearly with  $N$  and  $M$ , the direct approach appears to grow exponentially. Thus, the direct approach is the slowest approach, which makes sense since it requires solving a system of linear equations.

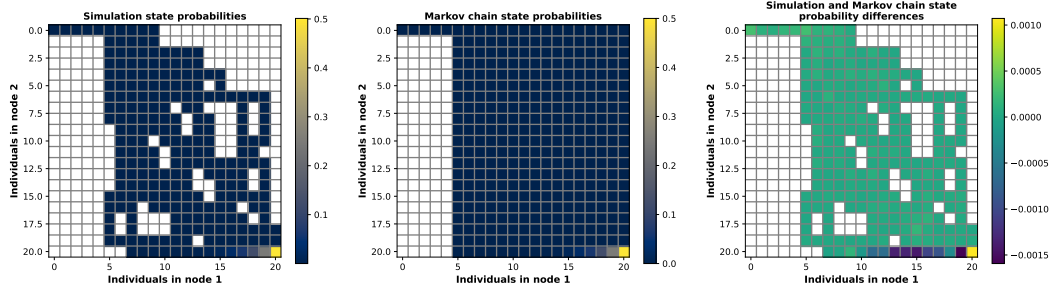
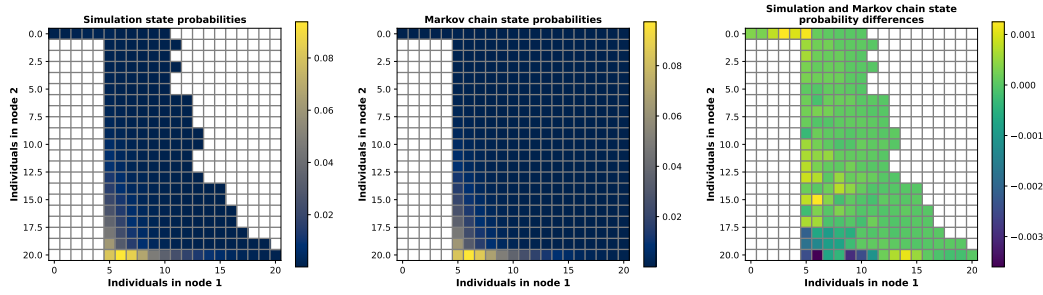
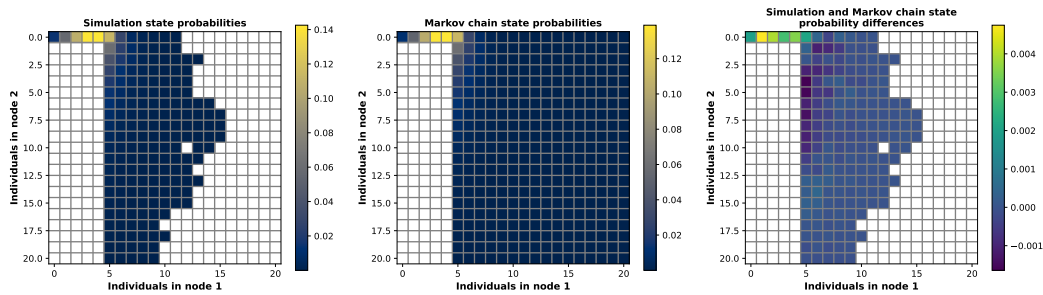
### 3.5.2 Accuracy of steady state probability calculations

Another comparison that can be made is the comparison between the steady state probabilities calculated using the Markov chain and the simulation. The steady state probabilities, defined in Section 3.3.1, are an essential measure since they are necessary in the calculation of all performance measures using the Markov chain model. Although, there is no need to calculate the steady state probabilities for the simulation, it is interesting to see how the two approaches compare. Note that for these comparisons the simulation was run 10 times, each with a runtime of 10,000 time units and a warm-up time of 500 time units.

Figures 3.16 to 3.20 show a comparison between the steady state probabilities between the Markov chain and the simulation for different values of  $\mu$ . The value of  $\mu$  gradually increases from  $\mu = 0.03$  to  $\mu = 0.27$  and for each one three plots are generated; the steady state probabilities generated by the Markov chain, the steady state probabilities generated by the simulation and the difference between the two. The values of the remaining parameters are shown in Table 3.1.

Table 3.1: Parameter values for steady state probabilities accuracy example 1

$\lambda_1$	$\lambda_2$	$\mu$	$C$	$T$	$N$	$M$
0.3	0.3	{0.03, 0.09, 0.15, 0.21, 0.27}	5	5	20	20

Figure 3.16: Heatmaps for  $\mu = 0.03$  of the state probabilities using the DES approach, the Markov chain approach and the differences between the two.Figure 3.17: Heatmaps for  $\mu = 0.09$  of the state probabilities using the DES approach, the Markov chain approach and the differences between the two.Figure 3.18: Heatmaps for  $\mu = 0.15$  of the state probabilities using the DES approach, the Markov chain approach and the differences between the two.

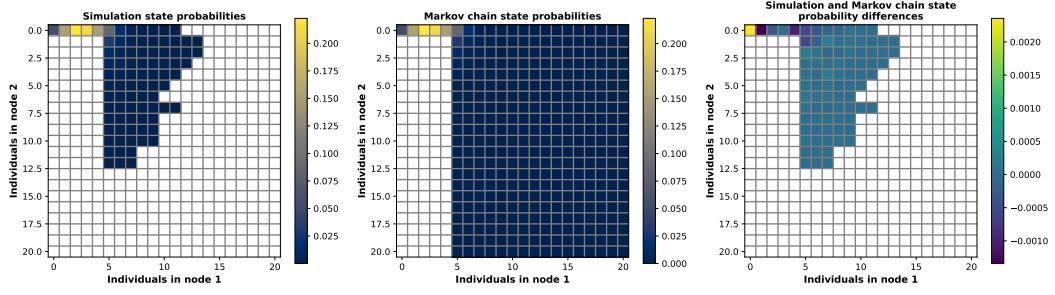


Figure 3.19: Heatmaps for  $\mu = 0.21$  of the state probabilities using the DES approach, the Markov chain approach and the differences between the two.

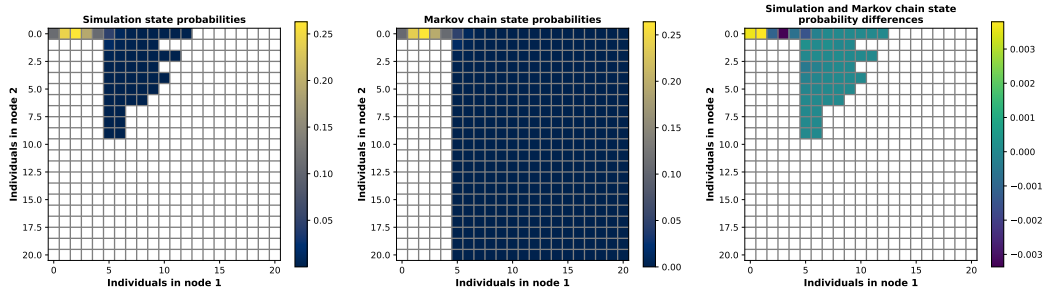


Figure 3.20: Heatmaps for  $\mu = 0.27$  of the state probabilities using the DES approach, the Markov chain approach and the differences between the two.

Figure 3.16 has the smallest value of  $\mu = 0.03$ . It can be seen that for both the Markov chain and the simulation the steady state probabilities are close to zero for most states apart from the states close to when the system is full. That is because the arrival rate of individuals is much lower than the service rate of individuals even with 5 servers. Note here that because the model is immediately flooded from the beginning, the simulation has no time to explore the state space, so the heatmap looks like it is missing some of its pieces. Additionally, it is interesting to note that as we increase the value of  $\mu$  in Figures 3.17 - 3.20 smaller states in the model have a higher value since individuals now exit the system faster. Also, note that for all values of  $\mu$  the difference between the Markov chain approach and the simulation is small.

Similarly Figures 3.21 - 3.25 show a comparison between the steady state probabilities between the Markov chain and the simulation for different values of the number of servers ( $C$ ). The value of  $C$  gradually increases from  $C = 1$  to  $C = 5$  and it can be seen that as the number of servers increases the steady state probabilities of the states tend to move towards the smaller states.

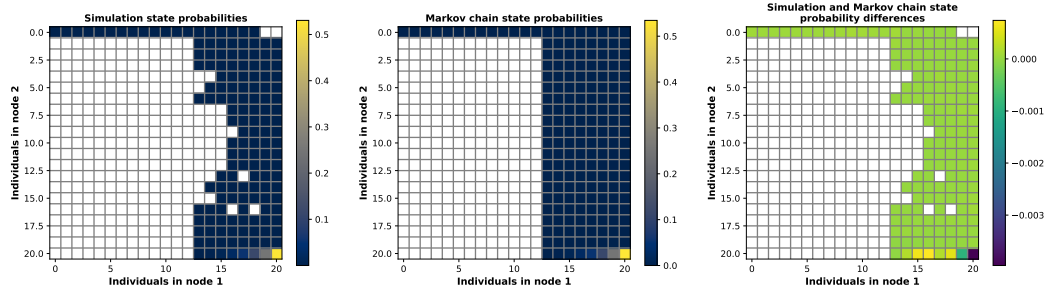


Figure 3.21: Heatmaps for  $C = 1$  of the state probabilities using the DES approach, the Markov chain approach and the differences between the two.

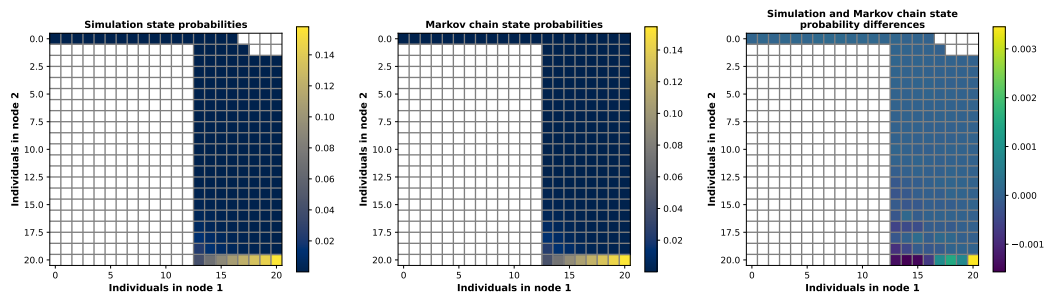


Figure 3.22: Heatmaps for  $C = 2$  of the state probabilities using the DES approach, the Markov chain approach and the differences between the two.

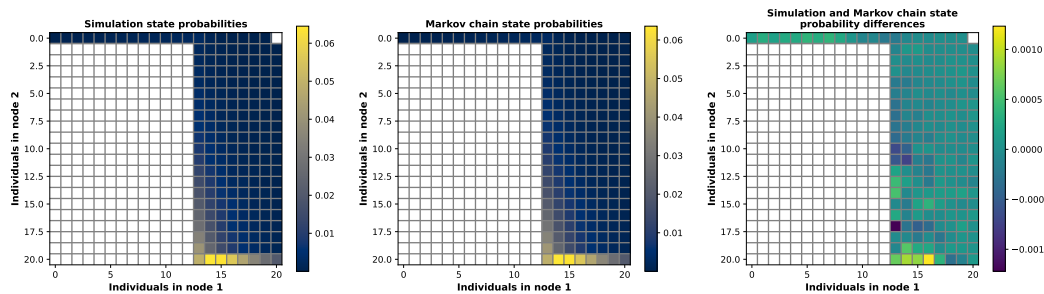


Figure 3.23: Heatmaps for  $C = 3$  of the state probabilities using the DES approach, the Markov chain approach and the differences between the two.

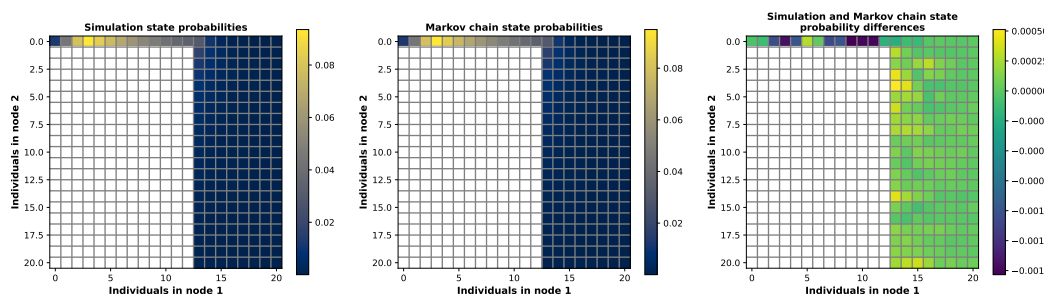


Figure 3.24: Heatmaps for  $C = 4$  of the state probabilities using the DES approach, the Markov chain approach and the differences between the two.

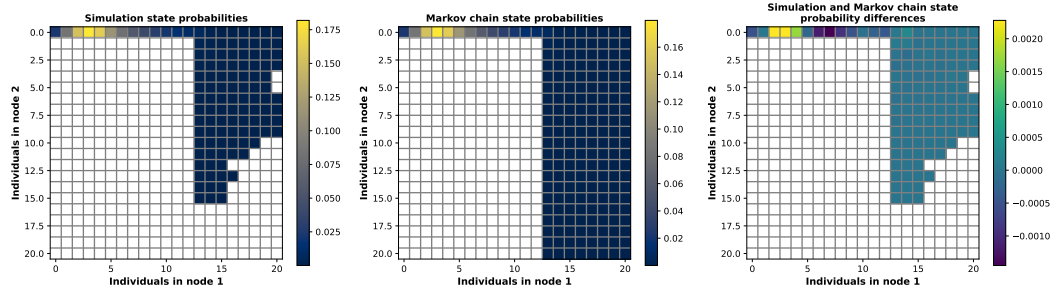


Figure 3.25: Heatmaps for  $C = 5$  of the state probabilities using the DES approach, the Markov chain approach and the differences between the two.

The parameter values used for Figures 3.21 - 3.25 are shown in Table 3.2.

Table 3.2: Parameter values for steady state probabilities accuracy example 2

$\lambda_1$	$\lambda_2$	$\mu$	$C$	$T$	$N$	$M$
1.5	1	0.7	$\{1, 2, 3, 4, 5\}$	13	20	20

### 3.5.3 Computation time of DES and Markov chain

The choice of the artificial truncation parameters  $N$  and  $M$  is an important decision of the model. The simulation can be used for both the truncated and untruncated models. This is not possible when obtaining the steady state probabilities of the finite state Markov chain. The value of  $N$  and  $M$  can be chosen to be arbitrarily large so as to approximate the untruncated model, but the computation time increases as the size of the state space increases. Table 3.3 shows the relative timings of the different approaches used to get the performance measures for different values of  $N$  and  $M$ . Note that  $N$  and  $M$  have the same value throughout the table. The simulation has a runtime of  $10^4$  time units and the displayed durations are for a single run of the simulation and similarly for 100 runs of the simulation. For getting the performance measures using the finite state Markov chain each timing recorded is for the computation of the steady state probabilities and then the corresponding performance measure.

Table 3.3: Relative timings for the computational time needed to get performance measures using the DES and Markov chain models. Note that these timings are all relative to the DES run with a single trial.

Value of $N$ and $M$	Simulation		Markov chain		
	Single trial	100 trials	Waiting formula	Blocking formula	Proportion formula
10	1	144.3	0.015	0.014	0.014
30	1	143.4	3.73	3.83	3.65
50	1	139.8	31.57	38.39	31.98
$\infty$	1	142.1	N/A	N/A	N/A

After some investigation it was found that a huge proportion of the duration of time needed to get the performance measures using the Markov chain approach is due to the creation of the generator matrix defined in equation (3.3). For example, for  $N = M = 50$  the state space of the Markov chain consists of approximately 2500 states (depending on the value of  $T$ ). Thus, the generator matrix, that consists of the rates from each state to every other state, has approximately  $2500^2 = 6,250,000$  entries. For larger values of  $N$  and  $M$ , the creation of this matrix is the most time consuming part of the Markov chain approach, even though most entries in the matrix are zero. By using equation (3.4), the set of states with a non-zero rate can be used to fill out the generator matrix. Thus, instead of iterating over the set states twice, the generator matrix can be filled out by iterating over the states only once and using  $\mathcal{M}(u, v)$  defined in equation (3.4). Table 3.4 shows how the relative timings of the different approaches change when using this smarter approach.

Table 3.4: Relative timings for the computational time needed to get performance measures using the DES model and the Markov chain model with the smarter approach.

Value of $N$ and $M$	Simulation		Markov chain		
	Single trial	100 trials	Waiting formula	Blocking formula	Proportion formula
10	1	119.2	0.000415	0.000146	0.000274
30	1	108.2	0.008040	0.035941	0.013451
50	1	109.3	0.147455	1.229303	0.179336
$\infty$	1	127.4	N/A	N/A	N/A

Overall, it can be seen that using the Markov chain approach is much faster than

simulating the system. Although, by choosing a larger value of  $N$  and  $M$  the computation time of the Markov chain model increases while the simulation time stays relatively similar.

### 3.5.4 Truncation effect on performance measures

This section is used to demonstrate the accuracy of the performance measure formulas of the constructed Markov model compared to the simulation as well as the effect of truncating the model. The simulation was run 100 times and the recorded mean waiting time at each iteration is used to populate the violin plots that are shown in Figures 3.26, 3.29 and 3.30.

Figures 3.26, 3.27 and 3.28 show a comparison between the calculated mean waiting time using Markov chains and the simulated waiting time using discrete event simulation over a range of values of  $\lambda_2$  (details of the discrete event simulation model can be found in Section 3.2). The parameter values are shown in Table 3.5.

Table 3.5: Parameter values for truncation effect example 1

$\lambda_1$	$\lambda_2$	$\mu$	$C$	$T$	$N$	$M$
2	[2, 6]	3	3	8	{10, 30, 50}	{10, 30, 50}

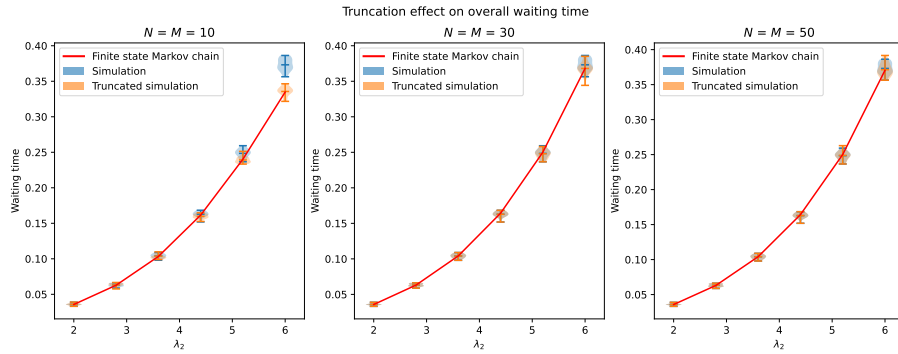


Figure 3.26: Example 1 - Comparison of overall mean waiting time between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation.

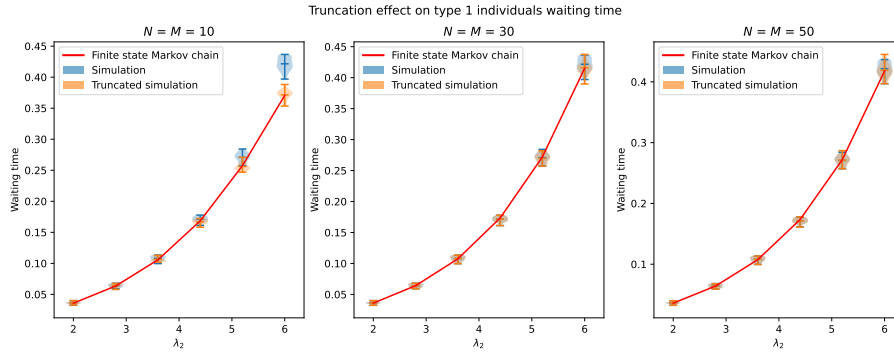


Figure 3.27: Example 1 - Comparison of type 1 individuals mean waiting time between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation.

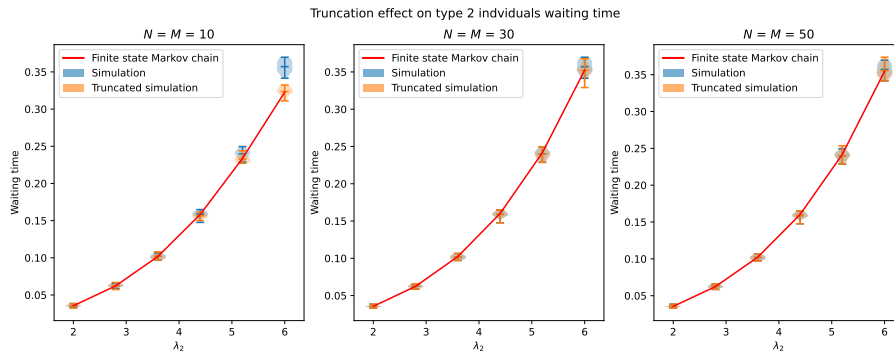


Figure 3.28: Example 1 - Comparison of type 2 individuals mean waiting time between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation.

In detail, Figure 3.26 shows the calculated mean waiting time using the Markov chain, using a truncated simulation and using a simulation with infinite capacity (without the artificial parameters  $N$  and  $M$ ). Each plot corresponds to different values of  $N$  and  $M$  and is run over different values of  $\lambda_2$ . The untruncated simulation values are the same at all three graphs since the effect of truncation does not apply to it. The waiting times generated by the truncated simulation match the ones generated by the Markov chains model. Note that this comparison includes both type 1 and type 2 individuals. Additionally Figures 3.27 and 3.28 show the mean waiting time for type 1 and type 2 individuals respectively. A similar observation to the overall mean waiting time can be made for the mean waiting time of type 1 and type 2 individuals.

Figure 3.29 shows the mean blocking time equivalent comparison between the three approaches used for the waiting time (Markov chain, truncated simulation and untruncated simulation). Similar to the waiting time, the blocking time



among the different approaches begin to get closer together as the value of  $N$  and  $M$  increases. Note that the blocking time can only be calculated for type 2 individuals.

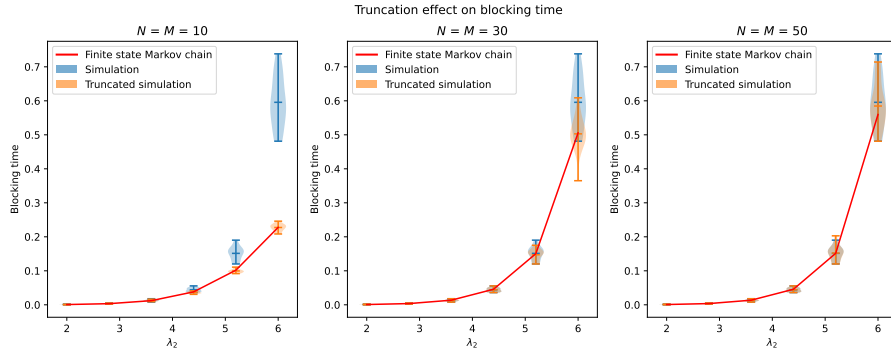


Figure 3.29: Example 1 - Comparison of mean blocking time between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation.

Finally, Figures 3.30, 3.31 and 3.32 show the overall proportion of individuals whose time in the system are within a time target for different values of  $N$  and  $M$ . Similar to the previous figures, as  $N$  and  $M$  increase the proportion of individuals between the simulation and the Markov chain get closer.

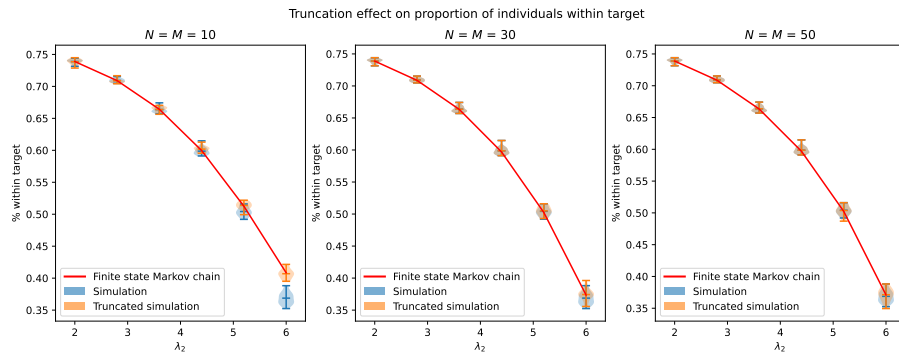


Figure 3.30: Example 1 - Comparison of overall proportion of individuals within target between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation.

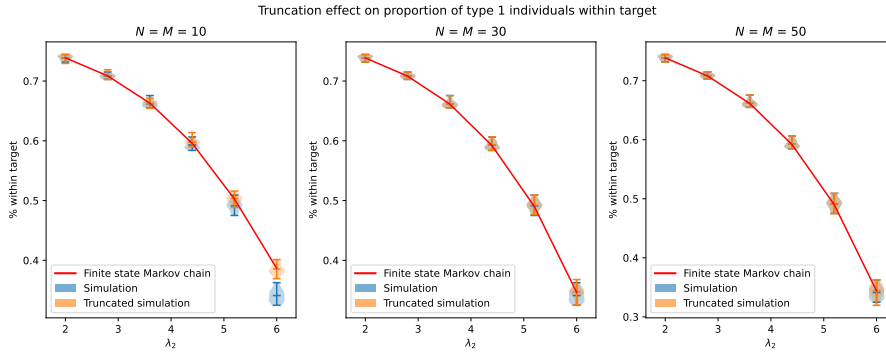


Figure 3.31: Example 1 - Comparison of proportion of type 1 individuals within target between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation.

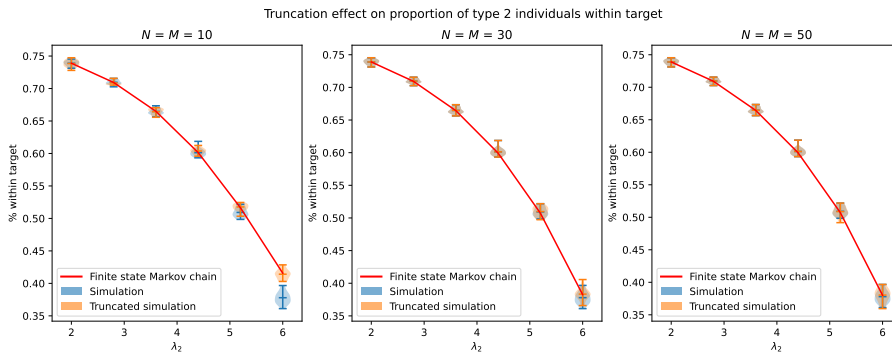


Figure 3.32: Example 1 - Comparison of proportion of type 2 individuals within target between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation.

For this particular set of parameters it can be seen that  $N = M = 50$  is a reasonable choice for the truncation parameters. The results obtained from the untruncated simulation are close to the ones obtained from the Markov chain model and the truncated simulation. In fact, for any set of parameters, increasing the values of  $N$  and  $M$  in the Markov chain model will result in a closer approximation to the untruncated simulation.

The same seven plots are also generated for a different set of parameters and higher values of  $\lambda_2$ . Using higher values of  $\lambda_2$  results in a more congested system where servers may not be able to serve as fast as individuals arrive. The parameters used for Figures 3.33 - 3.39 are shown in Table 3.6.

Table 3.6: Parameter values for truncation effect example 2

$\lambda_1$	$\lambda_2$	$\mu$	$C$	$T$	$N$	$M$
4	[2, 6]	2	5	12	{15, 30, 60}	{15, 30, 60}

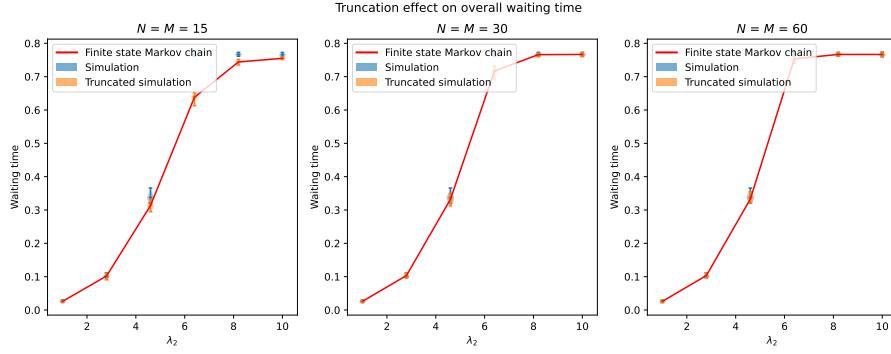


Figure 3.33: Example 2 - Comparison of overall mean waiting time between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation.

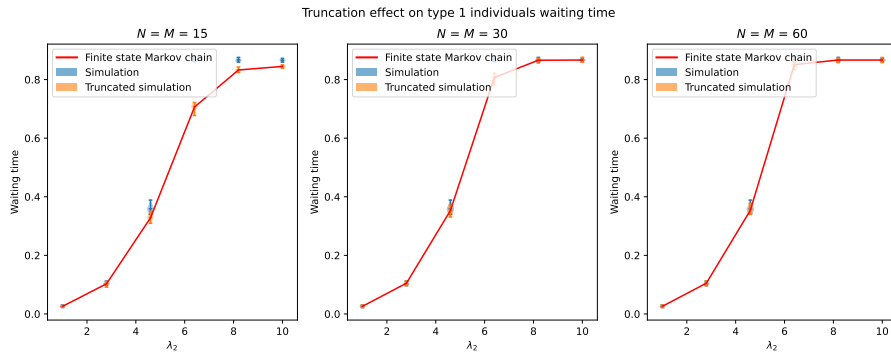


Figure 3.34: Example 2 - Comparison of type 1 individuals mean waiting time between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation.

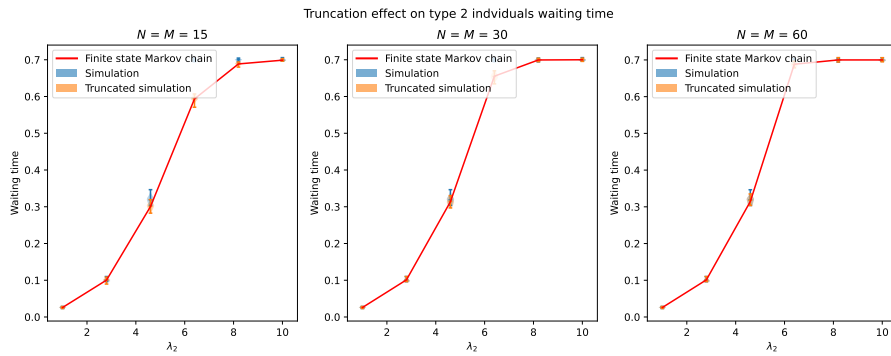


Figure 3.35: Example 2 - Comparison of type 2 individuals mean waiting time between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation.

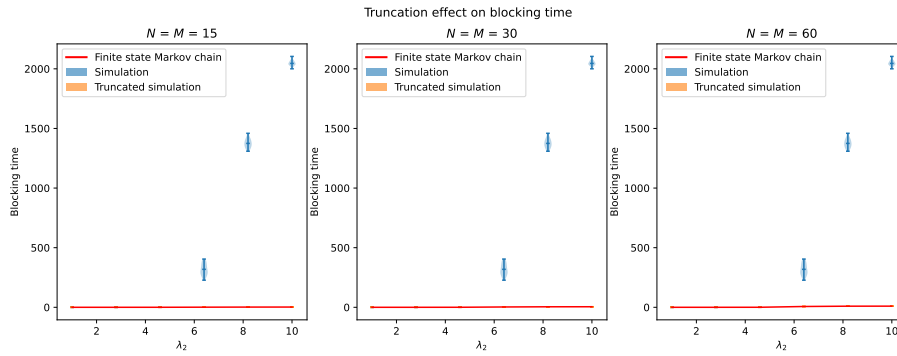


Figure 3.36: Example 2 - Comparison of mean blocking time between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation.

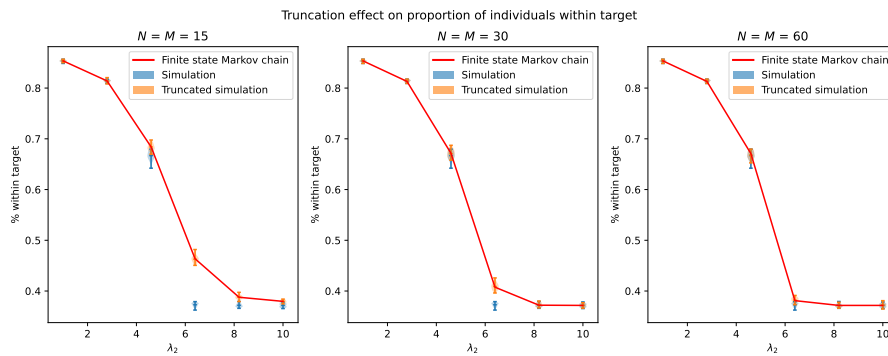


Figure 3.37: Example 2 - Comparison of overall proportion of individuals within target between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation.

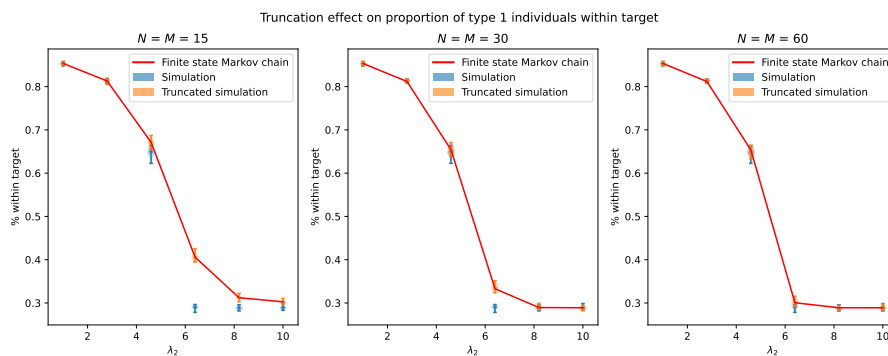


Figure 3.38: Example 2 - Comparison of proportion of type 1 individuals within target between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation.

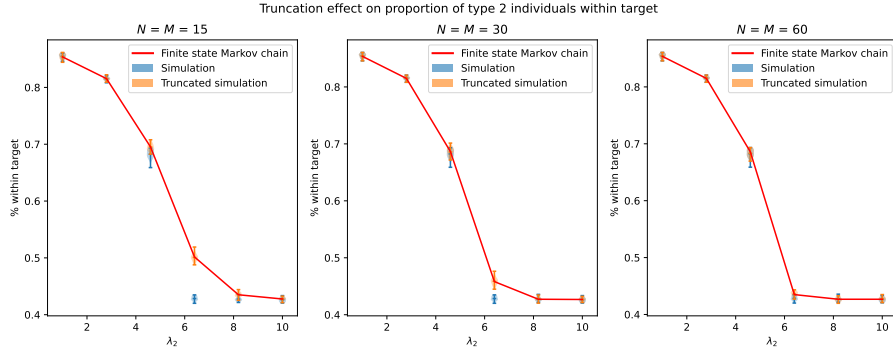


Figure 3.39: Example 2 - Comparison of proportion of type 2 individuals within target between values obtained from the Markov chain formula, values obtained from the truncated simulation and values obtained from the untruncated simulation.

Figure 3.36 shows that as  $\lambda_2$  increases the blocking time of the truncated and the untruncated simulation do not match. That is because as  $\lambda_2$  gets to a value that is beyond what the system can respond to, the truncated and untruncated system will never match. In essence, when the relative traffic intensity  $\rho = \frac{\lambda_1 + \lambda_2}{\mu \times C}$  is greater than 1 (i.e when  $\lambda_1 + \lambda_2 > \mu \times C$ ) the mean blocking time of the untruncated simulation will depend on the runtime of the simulation. The longer the simulation is run the more individuals will stay blocked in node 2, because there is no maximum capacity for node 2 and individuals will keep being added to it.

### 3.6 ED-EMS application

The queueing network described in Section 3 can be directly applied to a health-care setting. The healthcare scenario that is of interest here is at the interface between Emergency Department (ED) staff and Emergency Medical Services (EMS) staff. All parameters described in Section 3 can be mapped to some components of either the ED or the EMS.

- $\lambda_1 \rightarrow$  Type 1 individuals  $\rightarrow$  Individuals arriving without an ambulance (or via an ambulance that cannot be blocked)
- $\lambda_2 \rightarrow$  Type 2 individuals  $\rightarrow$  Individuals arriving with an ambulance (that can be blocked)
- $\mu \rightarrow$  Service rate  $\rightarrow$  The service rate of a patient
- $C \rightarrow$  Number of servers  $\rightarrow$  The number of staff available in the hospital

- $T \rightarrow$  Threshold  $\rightarrow$  The number of patients that need to be in the hospital to start blocking ambulances in the parking area.
- $N \rightarrow$  Node 1 capacity  $\rightarrow$  The overall hospital capacity (i.e. the number of beds in the hospital plus the queueing capacity)
- $M \rightarrow$  Node 2 capacity  $\rightarrow$  The parking capacity

Type 1 individuals are now all patients that arrive at the ED via some other means of transportation rather than an ambulance. Type 2 individuals are now all patients that arrive at the ED via an ambulance whose condition allows them to be delayed in the parking lot. The threshold parameter  $T$  is now the amount of patients that need to be in the hospital waiting area (and in service) to start blocking ambulances in the parking area. Figure 3.40 shows the applied version of Figure 3.1 that is the queueing network introduced in Section 3.

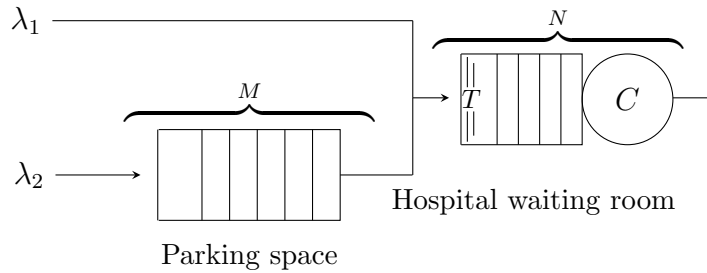


Figure 3.40: A diagrammatic representation of the Emergency Department. Similar to Figure 3.1, the threshold  $T$  only applies to type 2 individuals (i.e. patients arriving via an ambulance that can be blocked). If the number of individuals in the hospital's waiting room is greater than or equal to  $T$ , only type 1 patients are accepted while patients of type 2 are blocked in the parking space.

The performance measures of the queueing system have an additional meaning under the context of this new application. The average number of individuals in Node 1 and Node 2 are now the average number of patients in the hospital and the average number of ambulances in the parking space. The mean waiting time of individuals described in Section 3.4.1 is the average waiting time patients wait in Node 1 before they are seen by a doctor or nurse. Similarly, the mean blocking time from Section 3.4.2 is the mean time ambulances stay blocked in the parking space before the individuals they carry are allowed to enter the hospital. Finally, the proportion of individuals whose waiting time is longer than a predefined target time, described in Section 3.4.3 is now the percentage of patients that wait longer than a predefined time before they are seen by a doctor or nurse.

### 3.7 Chapter summary

This chapter introduces a novel queueing network model that is then used to model the emergent behaviour between EDs and the EMS. The model consists of two types of individuals and two queueing nodes. Type 1 individuals arrive instantly at node 1 and wait to receive their service while type 2 individuals arrive at node 2 and wait there until they are allowed to move to node 1. This decision is made based on the number of individuals in node 1 a pre-determined threshold  $T$ .

Two modelling approaches were used to model the queueing network. Section 3.2 gives an overview of the first approach that was used to model the queueing network with a discrete event simulation (DES) model. The DES model was also used to extract certain performance measures of the system. The implementation of this approach was written in `Python` and uses the `Ciw` library. The second approach that was used to model the queueing network was a Markov chain model. Section 3.3 introduces the Markov chain model that was used to model the queueing network and describes several approaches that were used to extract the steady state probabilities of the system (Section 3.3.1). The steady state probabilities of the system are calculated using numerical integration, linear algebra and a least squares approach. An attempt to derive a closed form solution for the steady state probabilities was also made using concepts from graph theory and combinatorics and is described in Appendix D. The Markov chain model was also implemented in `Python`.

Section 3.4 describes several algorithms that were used to extract performance measures from the Markov chain model. There are three key performance metrics that are of particular interest in this thesis. These are the average waiting time of individuals in the system, the average blocking time of individuals in node 2 and the proportion of individuals that are able to be served within a certain time target. Section 3.4.1 describes the different methods that were considered to extract the average waiting time of individuals in the system. Such methods were a recursive approach (Section 3.4.1.1.1), a direct approach (Section 3.4.1.2) and a closed form approach (Section 3.4.1.3). For the mean blocking time of individuals in node 2 only a direct approach was developed (Section 3.4.2). Because of the nature of arrivals in the system, a recursive approach was not formulated and neither was a closed form approach. The final performance measure that was considered was the proportion of individuals that are able to be served within a certain time target. For this performance measure the distribution of the waiting time of individuals had to be considered so that the cumulative distribution

function (CDF) could be extracted. The CDF could then be used to get the probability that the waiting time of individuals is less than a certain value. The CDF of the waiting time is partitioned into three cases, each one using a different distribution. The exponential distribution, the Erlang distribution and the Hypoexponential distribution were considered for the creation of this CDF.

Section 3.5 presents some numerical results and timings experiments for the queueing network, along with a comparison between the DES and Markov chain approach. Subsection 3.5.1 compares the three different approaches that were used to extract the average waiting time of individuals in the system and it is observed that the three approaches return the same results for a particular example of parameters. In addition, the computation time of the three approaches is compared for different values of the system's capacity. It is observed that, for small values of  $N$  and  $M$  all approaches take a similar amount of time to compute the average waiting time. As the values of  $N$  and  $M$  increase the direct approach takes a considerably longer time to compute the average waiting time. Subsection 3.5.2 compares the value of steady state probability vector that is obtained using the DES model and the Markov chain model. Subsection 3.5.3 compares the computation time of the DES model and the Markov chain model. Timing results are shown for a single trial and 100 trials of the DES model, and timings for each of the performance measure formula are shown for the Markov chain model. Consequently, the effect of truncating the Markov model is also investigated in Subsection 3.5.4. It is shown that as the arrival rate increases and the system is more busy, the truncation of the Markov model has a greater effect on the performance measures.

In addition Section 3.6 describes how the queueing network can be used in a healthcare scenario. The queueing structure is mapped to an ED that accepts two types of patients; non-urgent patients from the EMS and patients from other sources (walk-ins, urgent patients from the EMS, etc).

Overall, this Chapter has introduced a novel queueing network model that can be used to represent an ED that accepts patients from the EMS and other sources. This model will be used in the Chapter 4 to model the emergent behaviour between EDs and the EMS.



# Chapter 4

## Game theoretic model

### 4.1 Introduction

Apart from the queueing theoretic model the second main outcome of this research is the construction of a game theoretic model that uses the queueing model described in Section 3. This game theoretic framework consists of three players where these players (in Section 4.5) will represent the Emergency Medical Services and two Emergency Departments. The game theoretic model aims to look into behavioural patterns that emerge when the players are interacting with each other and act in such a way so that they maximise their utility. This chapter consists of four main sections:

- Section 4.2 gives a brief introduction to the game theoretic concepts
- Section 4.3 describes the formulation of the game theoretic model that is used in this research
- Section 4.4 describes the methodology that is used to solve the game theoretic model
- Section 4.5 describes the application of the game theoretic model to the Emergency Medical Services and two Emergency Departments

This chapter extends the concepts described in [114].

### 4.2 Game theory concepts

In game theory there are several different forms of games. This section outlines only the ones necessary for the formulation of the scenario studied in this research.

The first one is normal form games, the second is perfect information extensive form games, and the third is extensive form games with imperfect information. The documentation of the python library `nashpy` can be used to find more information about the different types of games and concepts that are discussed throughout this section.

### 4.2.1 Normal form games

Normal form games are strategic games that model strategic decision-makers. These decision makers are referred to as players where each player has a set of possible actions that they can take. The game captures the interaction between the players by taking into account the payoffs that each player receives for each possible combination of actions taken by all players. A strategic game consists of a set of players, a set of actions for each player, and a payoff function that maps each combination of actions to a payoff for each player [108].

Normal form games with 2 players are usually represented by 2 matrices that include the payoffs for each player for every possible combination of actions. The set of available actions of a player is denoted by  $|S|$ . A pure strategy is a strategy that is associated with a single action  $i$  and a mixed strategy  $\sigma$  is a strategy that is associated with a probability distribution over the pure strategies  $\sigma_i$ , where  $\sum_i^{|S^k|} \sigma_i^k = 1$  for each player  $k \in \{1, 2, \dots, n\}$ . A utility function  $u_k$  is a function that maps  $n$  strategy profiles (one for each player) to a payoff for player  $k$ . The payoff for player  $k$  is given by  $u_k(\sigma^1, \sigma^2, \dots, \sigma^n)$  where each  $\sigma^i$  represents a player.

For example, consider the Prisoner's Dilemma game shown in Table 4.1 [55]. In this game, player 1 can choose to either *Cooperate* (C) or *Defect* (D) and player 2 can also choose to either *Cooperate* (C) or *Defect* (D).

Table 4.1: A game theoretic matrix representation of the Prisoner's Dilemma game

Player 1 \ Player 2	Cooperate	Defect
Cooperate	3, 3	0, 5
Defect	5, 0	1, 1

An alternative way to represent the Prisoner's Dilemma game is by using the payoff matrices in (4.1). The payoff matrix  $A$  shows the payoffs for player 1 and

the payoff matrix  $B$  shows the payoffs for player 2.

$$A = \begin{pmatrix} 3 & 0 \\ 5 & 1 \end{pmatrix} \begin{matrix} C \\ D \end{matrix} \quad B = \begin{pmatrix} 3 & 5 \\ 0 & 1 \end{pmatrix} \begin{matrix} C \\ D \end{matrix} \quad (4.1)$$

The entry in the first row and first column of both matrix  $A$  and matrix  $B$  is 3. That indicates that if both players choose to *Cooperate* (i.e. player 1 chooses row 1 and player 2 chooses column 1) then they both receive a payoff of  $u_1 = 3$  and  $u_2 = 3$ . Similarly, the entry in the second row and first column of matrix  $A$  is 0 and the equivalent entry in matrix  $B$  is 5. That indicates that if player 1 chooses to *Defect* and player 2 chooses to *Cooperate* then player 1 receives a payoff of  $u_1 = 0$  and player 2 receives a payoff of  $u_2 = 5$ .

Equivalently, a 3-player normal form game is represented by three 3-dimensional matrices  $A$ ,  $B$  and  $C$ . The rows of each matrix correspond to the actions of player 1, the columns of each matrix correspond to the actions of player 2 and the third dimension of each matrix corresponds to the actions of player 3.

### 4.2.2 Nash Equilibrium

The Nash equilibrium is a concept that was developed by John Nash in the 1950s. It is a concept that is used to describe the behaviour of players in a game when they are playing against each other [83]. In essence, it is the state of the game where players are not able to improve their payoff by changing their strategy.

**Theorem 1** *In a 2-player game a player's strategy  $\hat{\sigma}^1$  is said to be a **best response** to the opposing's player strategy  $\sigma^2$  if the following holds:*

$$u_1(\hat{\sigma}^1, \sigma^2) \geq u_1(\sigma^1, \sigma^2) \quad \text{for all } \sigma^1 \in S^1 \quad (4.2)$$

The Nash equilibrium is a pair of strategies for the two players where neither player can improve their payoff by changing their strategy. Thus the following definition can be built upon the best response definition.

**Theorem 2** *A pair of strategies  $\hat{\sigma}^1$  and  $\hat{\sigma}^2$  is a Nash equilibrium if they are both best responses to each other. In other words, the following holds:*

$$u_1(\hat{\sigma}^1, \sigma^2) \geq u_1(\sigma^1, \sigma^2) \quad \text{and} \quad u_2(\sigma^1, \hat{\sigma}^2) \geq u_2(\sigma^1, \sigma^2) \quad \text{for all } \sigma^1 \in S^1, \sigma^2 \in S^2 \quad (4.3)$$

Consider the pig and piglet game where there are two players, the pig and the piglet and two strategies each, to *Push* (P) a lever or *Don't push* it (D). The payoff matrices  $A$  and  $B$  for this game are shown in (4.4) where matrix  $A$  corresponds to the pig's payoff and matrix  $B$  corresponds to the piglet's payoff.

$$A = \begin{matrix} & \begin{matrix} P & D \end{matrix} \\ \begin{matrix} P \\ D \end{matrix} & \begin{pmatrix} 4 & 2 \\ 5 & 0 \end{pmatrix} \end{matrix} \quad B = \begin{pmatrix} 2 & 3 \\ -1 & 0 \end{pmatrix} \quad (4.4)$$

Consider the case where the pig is playing the *Push* strategy. The piglet's best response to the pig is to play the *Don't push* strategy as this will result in a higher payoff of  $u_2 = 3$  instead of  $u_2 = 2$ . Similarly, if the pig is playing the *Don't push* strategy then the piglet's best response is still the *Don't push* strategy as this will result in a higher payoff of  $u_2 = 0$  instead of  $u_2 = -1$ . Thus, regardless of the pig's strategy the piglet's best response is to play the *Don't push* strategy. From the pig's perspective, if the piglet is playing the *Don't push* strategy then the pig's best response is to play the *Push* strategy as this will result in a higher payoff of 2 instead of 0. Therefore, one possible pair of strategies that is a Nash equilibrium is  $\sigma^1 = (1, 0)$  and  $\sigma^2 = (0, 1)$ . Note that  $\sigma = (p_1, p_2)$  where  $p_1$  is the probability of the player playing the first strategy and  $p_2$  is the probability of the player playing the second strategy.

This is only an example of the Nash equilibrium where it only consisted of pure strategies. In general, the Nash equilibrium can consist of mixed strategies. There are numerous algorithms that can be used to find the Nash equilibrium of a game. The algorithms that will be discussed in this section are the Lemke-Howson algorithm and the support enumeration algorithm.

#### 4.2.2.1 Lemke-Howson Algorithm

The Lemke-Howson algorithm is a method that can be used to find a Nash equilibrium of a 2-player normal form game [88]. Note that the algorithm is only applicable to 2-player normal form games and the algorithm outputs only one Nash equilibrium. The Lemke-Howson algorithm uses the concept of support enumeration described in [106] that is used to find all pairs of best responses for a given game. For a non-degenerate 2-player game [73] the Lemke-Howson algorithm performs the following steps:

1. Obtain the best response polytopes  $P$  and  $Q$ .
2. Choose a starting label to drop, this will correspond to a vertex of  $P$  or  $Q$ .

3. In that polytope, remove the label from the corresponding vertex and move to the vertex that shared that label. A new label will be picked up and duplicated in the other polytope.
4. In the other polytope drop the duplicate label and move to the vertex that shared that label.

Repeat steps 3 and 4 until there are no duplicate labels.

The Lemke-Howson algorithm is implemented using `nashpy` [143], which is a game theoretic python library. The code snippet in 4.1 shows how to use the Lemke-Howson algorithm to find the Nash equilibrium of the pig and piglet game described in equation (4.4).

```
>>> import nashpy as nash
>>> import numpy as np
>>> A = np.array([[4, 2], [6, 0]])
>>> B = np.array([[2, 3], [-1, 0]])
>>> game = nash.Game(A, B)
>>> sigma_1, sigma_2 = game.lemke_howson(initial_dropped_label=0)
>>> sigma_1
array([1., 0.])
>>> sigma_2
array([0., 1.])
```

Code snippet 4.1: Lemke-Howson python code for the pig and piglet game.

The outcome indicates that the Nash equilibrium of the pig and piglet game is  $\sigma^1 = (1, 0)$  and  $\sigma^2 = (0, 1)$ . That is, the pig should always push the lever and the piglet should never push the lever. Consider the game of Rock-Paper-Scissors now where the payoff matrices are shown in (4.5).

$$A = \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix} \begin{matrix} R \\ P \\ S \end{matrix} \quad B = \begin{pmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \\ 1 & -1 & 0 \end{pmatrix} \begin{matrix} R \\ P \\ S \end{matrix} \quad (4.5)$$

By implementing the Lemke-Howson algorithm on the Rock-Paper-Scissors game, one Nash equilibrium can be found.

```
>>> A = np.array([
...     [1, -1, 0],
...     [0, 1, -1],
...     [-1, 0, 1]
... ])
>>> B = np.array([
...     [-1, 1, 0],
...     [0, -1, 1],
...     [1, 0, -1]
```

```

... )
>>> game = nash.Game(A, B)
>>> sigma_1, sigma_2 = game.lemke_howson(initial_dropped_label=0)
>>> sigma_1
array([0.33333333, 0.33333333, 0.33333333])
>>> sigma_2
array([0.33333333, 0.33333333, 0.33333333])

```

Code snippet 4.2: Lemke-Howson python code for the Rock-Paper-Scissors game.

The outcome indicates that a Nash equilibrium of the Rock-Paper-Scissors game is  $\sigma^1 = (1/3, 1/3, 1/3)$  and  $\sigma^2 = (1/3, 1/3, 1/3)$ . That is, the players should play each strategy with equal probability.

#### 4.2.2.2 Support Enumeration

Another algorithm that can be used to find the Nash equilibrium of a 2-player normal form game is the support enumeration algorithm. The support enumeration algorithm can be used to find all Nash equilibria of a non-degenerate 2-player normal form game [73] by using all possible pairs of support of a game [104]. The following steps are performed by the support enumeration algorithm and return all pairs of best responses in a game with payoff matrices  $A$  and  $B$  [119]:

1. For all possible pairs of support  $(M_x, N_y)$  of the mixed strategies  $(x, y)$
2. Solve the following equations:

$$\sum_{i \in M_x} x_i B_{ij} = v, \quad \text{for all } j \in N_y \quad (4.6)$$

$$\sum_{i \in M_x} x_i = 1 \quad (4.7)$$

$$\sum_{j \in N_y} y_j A_{ij} = u, \quad \text{for all } i \in M_x \quad (4.8)$$

$$\sum_{j \in N_y} y_j = 1 \quad (4.9)$$

The support enumeration algorithm is implemented using the **nashpy** [143] python library. Consider the payoff matrices for the game of coordination shown in (4.10), where the two players would prefer to perform the same action if possible, but player 1 has a slight preference for action  $S_1$  and player 2 has a slight

preference for action  $S_2$ .

$$A = \begin{pmatrix} 3 & 1 \\ 0 & 2 \end{pmatrix} \begin{matrix} S_1 \\ S_2 \end{matrix} \quad B = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix} \begin{matrix} S_1 \\ S_2 \end{matrix} \quad (4.10)$$

The piece of code in 4.3 implements the support enumeration algorithm on the coordination game and returns all Nash equilibria of the game.

```
>>> A = np.array([
...     [3, 1],
...     [0, 2]
... ])
>>> B = np.array([
...     [2, 1],
...     [0, 3]
... ])
>>> game = nash.Game(A, B)
>>> nash1, nash2, nash3 = tuple(game.support_enumeration())
>>> nash1
(array([1., 0.]), array([1., 0.]))
>>> nash2
(array([0., 1.]), array([0., 1.]))
>>> nash3
(array([0.75, 0.25]), array([0.25, 0.75]))
```

Code snippet 4.3: Support enumeration python code for the coordination game.

The outcome indicates that there are three Nash equilibria of the coordination game. The first Nash equilibrium is  $\sigma^1 = (1, 0)$  and  $\sigma^2 = (1, 0)$ , which corresponds to both players playing action  $S_1$ . Similarly, the second Nash equilibrium is  $\sigma^1 = (0, 1)$  and  $\sigma^2 = (0, 1)$ , and corresponds to both players playing action  $S_2$ . The third Nash equilibrium is  $\sigma^1 = (\frac{3}{4}, \frac{1}{4})$  and  $\sigma^2 = (\frac{1}{4}, \frac{3}{4})$ , which means that player 1 should play action  $S_1$  with probability  $\frac{3}{4}$  and action  $S_2$  with probability  $\frac{1}{4}$  and player 2 should play action  $S_1$  with probability  $\frac{1}{4}$  and action  $S_2$  with probability  $\frac{3}{4}$ .

### 4.2.3 Learning Algorithms

Nash equilibria is a theoretical measure which can be inconsistent with intuitive notions about what should be the outcome of a game [103]. The concept of Nash equilibria is not always applicable to real-world situations. There are scenarios where a game has multiple Nash equilibria, but not all of them can be reached by allowing the players to repeatedly play the game. Evolutionary stable strategies (ESS) is a subsequent general concept that can be more applicable to real-world situations [107]. Consider a population that consists of all possible strategies of

a player and stronger strategies can invade weaker ones strategies and replace them in this population [128, 129]. A strategy is an ESS if no other strategy can replace it or invade it from the population of all strategies. Note that all strategies that are ESS are also Nash equilibrium. Learning algorithms can be used to reach certain ESS of a game. The benefit of using learning algorithms and ESS, instead of calculating the Nash equilibria, is that not only is it a more powerful concept of equilibrium, but also the decision journey of the players can be observed. Thus, players' decisions at each time step can be observed and the learning process can be visualised. There are numerous leaning algorithms that can be used to find ESS of a game. In this subsection an overview of some of them will be given.

#### 4.2.3.1 Fictitious play

One such learning algorithm is called Fictitious play [24, 52]. The fictitious play algorithm is a sequential learning algorithm that is based on the assumption that the players are rational and have perfect information about the game. At each time step, the players play a strategy that is based on the previous actions of the opposing player. In other words the players play a best response to their opponent's empirical frequency of actions.

Once again, using the `nashpy` python library [143], the fictitious play algorithm can be implemented on the game shown in 4.4.

```
>>> A = np.array([
...     [4, 1, 3],
...     [2, 0, 2],
...     [3, 4, 1]
... ])
>>> B = np.array([
...     [4, 5, 1],
...     [2, 3, 2],
...     [6, 4, 0]
... ])
>>> game = nash.Game(A, B)
>>> np.random.seed(0)
>>> play_counts = list(game.fictitious_play(iterations=1000))
>>> play_counts[-1]
[array([642.,    0., 358.]), array([757., 243.,    0.]])
```

Code snippet 4.4: Fictitious play python code for a 2-player game.

The output is the empirical frequency of actions played by each player. The outcome indicates that the fictitious play algorithm converges to the following Nash equilibrium:  $\sigma^1 = (\frac{2}{3}, 0, \frac{1}{3})$  and  $\sigma^2 = (\frac{3}{4}, \frac{1}{4}, 0)$ . That is the outcome of the last iteration of the fictitious play algorithm and normalised to sum to one. Figure 4.1 shows all iterations of the fictitious play algorithm and how the players



converge to a Nash equilibrium.

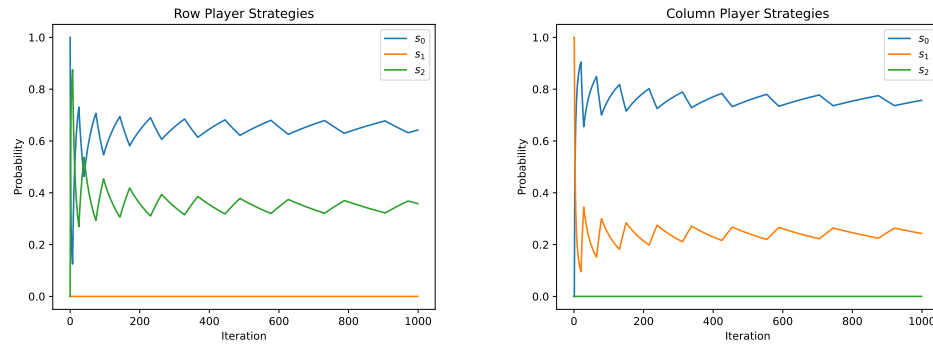


Figure 4.1: Example of fictitious play algorithm run for 1000 iterations that converges to a Nash equilibrium.

#### 4.2.3.2 Stochastic fictitious play

Another similar learning algorithm is called stochastic fictitious play [52, 67]. Stochastic fictitious play is a variation of the fictitious play algorithm where a stochastic perturbation  $\epsilon_i$  is added to each expected payoff where  $\epsilon_i \in [0, \bar{\epsilon}]$  where  $\bar{\epsilon}$  is a parameter needed for the algorithm.

```
>>> A = np.array([
...     [4, 1, 3],
...     [2, 0, 2],
...     [3, 4, 1]
... ])
>>> B = np.array([
...     [4, 5, 1],
...     [2, 3, 2],
...     [6, 4, 0]
... ])
>>> game = nash.Game(A, B)
>>> np.random.seed(0)
>>> play_counts_and_distributions = tuple(
...     game.stochastic_fictitious_play(iterations=1000)
... )
>>> end_play_count, end_distribution = play_counts_and_distributions[-1]
>>> end_play_count
[array([624.,  0., 376.]), array([767., 233.,  0.])]
```

Code snippet 4.5: Stochastic fictitious play python code for a 2-player game.

The output is the empirical probability of all actions played by player 1 and player 2. The outcome indicates that the stochastic fictitious play algorithm converges to the following strategy:  $\sigma^1 = (\frac{2}{3}, 0, \frac{1}{3})$  and  $\sigma^2 = (\frac{3}{4}, \frac{1}{4}, 0)$ . This is the outcome of the last iteration of the stochastic fictitious play which is also similar to the fictitious play outcome. Figure 4.2 shows all iterations of the stochastic fictitious play algorithm and how the players converge to a Nash equilibrium.

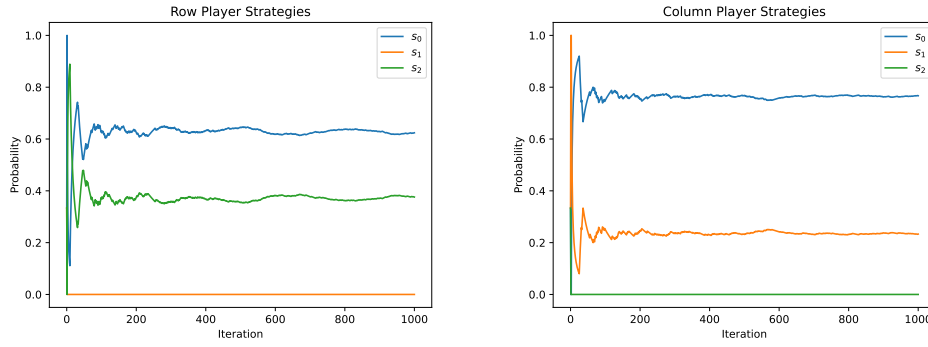


Figure 4.2: Example of stochastic fictitious play algorithm for 1000 iterations that converges to a Nash equilibrium.

#### 4.2.3.3 Asymmetric replicator dynamics

The learning algorithm that will be used most in this thesis is asymmetric replicator dynamics [1]. Replicator dynamics is a learning algorithm that is used to express the evolutionary dynamics of a population of players [81]. Consider a large population of some agents, also known as replicators. Different types of such replicators meet and interact. Each such interaction generates a certain payoff for each type of the replicators. This payoff is often referred to as the fitness of the replicator. In evolutionary game theory these replicators are the strategies of the two players.

Consider two types of individuals,  $A$  and  $B$ , each with their own set of strategies,  $S^A$  and  $S^B$ . Different strategies from  $S^A$  are assigned among the population of type  $A$  and different strategies from  $S^B$  are assigned among the population of type  $B$ . Individuals of type  $A$  are randomly paired with individuals of type  $B$  and perform their assigned strategies. As the game progresses the proportion of each strategy changes based on previous interactions. Given that the payoff matrices of the two players are  $A$  and  $B$ , the fitness of a strategy is given by:

$$f_x = Ay \quad f_y = x^T B \quad (4.11)$$

Note that  $x$  and  $y$  are the strategy vectors and correspond to the population proportion of each strategy. The average fitness of the two types of individuals is also given by:

$$\phi_x = f_x x^T \quad \phi_y = f_y y \quad (4.12)$$

Finally the rate of change of the strategies are captured by the following equations:

$$\frac{dx}{dt} = x_i((f_x)_i - \phi_x) \quad \text{for all } i \quad (4.13)$$

$$\frac{dy}{dt} = y_i((f_y)_i - \phi_y) \quad \text{for all } i \quad (4.14)$$

Asymmetric replicator dynamics is implemented by using the python `nashpy` library. Code snippet 4.6 shows how to use the asymmetric replicator dynamics algorithm to find the Nash equilibrium of a two-player game.

```
>>> A = np.array([
...     [4, 1, 3],
...     [2, 0, 2],
...     [3, 4, 1]
... ])
>>> B = np.array([
...     [4, 5, 1],
...     [2, 3, 2],
...     [6, 4, 0]
... ])
>>> game = nash.Game(A, B)
>>> xs, ys = game.asymmetric_replicator_dynamics(
...     timepoints=np.linspace(0, 100, 100),
... )
>>> np.round(xs[-1], 4)
array([0.9207, 0.      , 0.0793])
>>> np.round(ys[-1], 4)
array([0.7429, 0.2571, 0.      ])
```

Code snippet 4.6: Asymmetric replicator dynamics python code for a 2-player game.

The output of the asymmetric replicator dynamics algorithm is the latest population proportion of each strategy  $\sigma^1 = (0.92, 0, 0.08)$  and  $\sigma^2 = (0.74, 0.26, 0)$ . That doesn't mean that the strategies have reached a steady state. For this condition to be reached the rate of change of the strategies needs to be  $\frac{dx}{dt} = \frac{dy}{dt} = 0$ . In fact Figure 4.3 shows that the strategies played over time using the asymmetric replicator dynamics algorithm have in fact not reached a steady state.

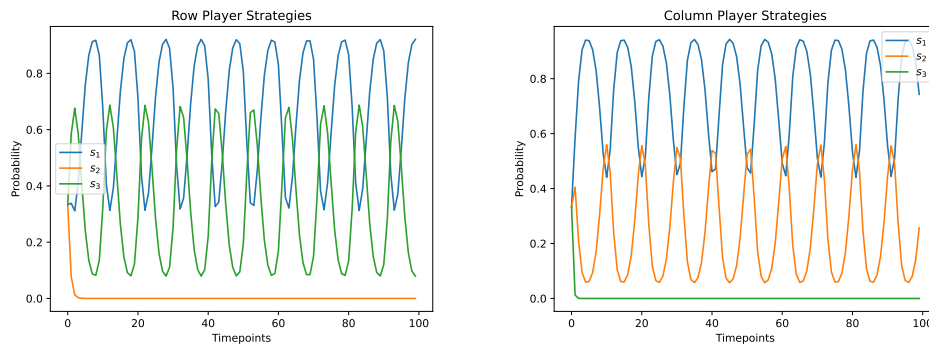


Figure 4.3: Example of asymmetric replicator dynamics algorithm that does not converge.

It can be seen that strategies player 1 alternates between strategy  $s_1$  and  $s_3$  while player 2 alternates between strategy  $s_1$  and  $s_2$ . This game cannot reach a steady state given the uniform initial distribution of the strategies. Consider a different game where the steady strategy choice can be reached.

```
>>> A = np.array([
...     [4, 1],
...     [2, 5],
... ])
>>> B = np.array([
...     [4, 5],
...     [2, 3],
... ])
>>> game = nash.Game(A, B)
>>> x0 = np.array([0.9, 0.1])
>>> y0 = np.array([0.9, 0.1])
>>> xs, ys = game.asymmetric_replicator_dynamics(
...     timepoints=np.linspace(0, 20, 100),
...     x0=x0,
...     y0=y0,
... )
>>> np.round(xs[-1], 4)
array([0., 1.])
>>> np.round(ys[-1], 4)
array([0., 1.])
```

Code snippet 4.7: Asymmetric replicator dynamics run on a game that is able to reach steady state

The output of the asymmetric replicator dynamics algorithm is the latest population proportion of each strategy  $\sigma^1 = (0, 1)$  and  $\sigma^2 = (0, 1)$ . This indicates that the strategies have reached a steady state. In replicator dynamics when a replicator is eliminated (in this case strategy  $s_1$ ), it cannot be recovered. In addition, note that for this game the asymmetric replicator dynamics algorithm was ran with a non-uniform initial distribution of the strategies. In fact, even though the initial distribution of the strategies was  $x_0 = (0.9, 0.1)$  and  $y_0 = (0.9, 0.1)$ ,

strategy  $s_2$  still managed to take over the population.

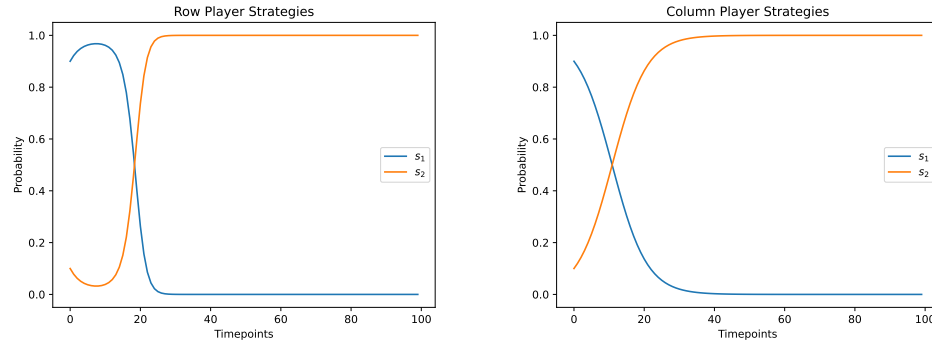


Figure 4.4: Example of asymmetric replicator dynamics algorithm that converges

#### 4.2.4 Perfect-information extensive form game

Unlike normal-form games extensive-form games work in a sequential way, where players do not make decisions at the same time. Instead, the first player chooses their strategy and then the opposing player, fully aware of the choice made by the first player, chooses their own strategy. There are numerous situations where decision makers can change their actions based on the actions of other decision makers. Such type of sequential games are also referred to as extensive form games. One of the most common types of extensive form games is the perfect-information extensive form game. In this type of game, the players are assumed to have perfect information about the previous actions of other players. There are four key components of a perfect-information extensive form game; the players, the terminal nodes, the player function and the preferences of the players [109]. Examples of such games are the game of chess and the game of Backgammon [66].

Perfect information extensive form games are represented by a tree where the nodes of the tree are the terminal nodes and represent the outcome of the game. Figure 4.5 shows an example of a perfect information extensive form game.

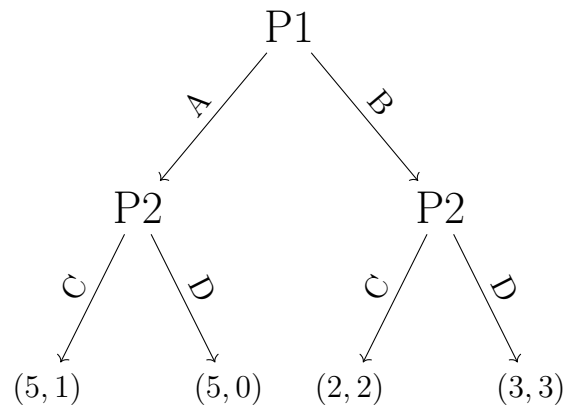


Figure 4.5: Example of a perfect information extensive form game with 2 players and 4 terminal nodes.

Figure 4.5 shows an example of a perfect information extensive form game. The game starts at the root node and the players take turns to make a move. Player 1 can choose to either perform action  $A$  or action  $B$ . Once player 1 has made a move, player 2 can choose to either action  $C$  or action  $D$ , while having complete awareness of the action taken by player 1 and hence their own position on the tree. The final nodes of the tree represent the outcome of the game.

#### 4.2.5 Imperfect-information extensive form game

An imperfect information game is defined as an extensive form game where some of the information about the game state is hidden for at least one of the players [17]. In other words, when making a decision, the players might not know their exact position on the tree. Similar to perfect information, imperfect information games are also represented by a tree. Figure 4.6 shows an example of an imperfect information extensive form game.

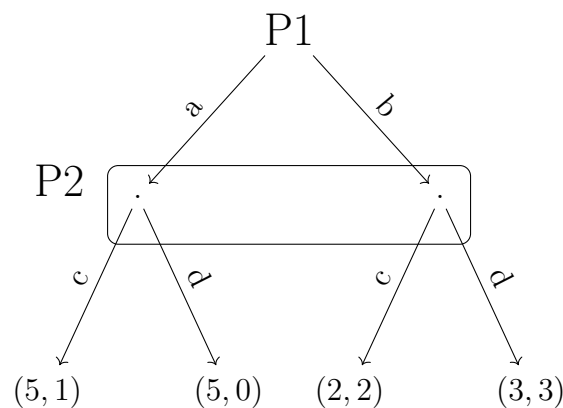


Figure 4.6: An example of an imperfect information extensive form game with 2 players and 4 terminal nodes.

Figure 4.6 shows an example of an imperfect information extensive form game where player 2, when making their decision, does not know whether they are in the left or right branch of the tree. The game starts at the root node and the players take turns to make a move. Player 1 can choose to either action  $a$  or action  $b$ . Once player 1 has made a move, player 2 can choose to either action  $c$  or action  $d$ , while having incomplete awareness of the action taken by player 1 and hence their own position on the tree. The final nodes of the tree represent the outcome of the game. This game can also be represented by a normal form game since both players end up being completely unaware of the actions taken by the other player. The payoff matrices in (4.15) show the normal form game representation of the imperfect information extensive form game shown in Figure 4.6.

$$A = \begin{matrix} & \begin{matrix} c & d \end{matrix} \\ \begin{matrix} a \\ b \end{matrix} & \begin{pmatrix} 5 & 5 \\ 2 & 3 \end{pmatrix} \end{matrix} \quad B = \begin{pmatrix} 1 & 0 \\ 2 & 3 \end{pmatrix} \quad (4.15)$$

### 4.3 Formulation

In order to formulate the game theoretic model one needs to define the players of the game, the strategies of each player and the payoffs of each pair of strategy being played.

#### 4.3.1 Players and parameters

The problem studied is a 3-player extensive form game that consists of three players. This will later be reduced to a 2-player standard normal form game [96]. The three players are:

- the decision makers of queueing system  $A$
- the decision makers of queueing system  $B$
- a distribution service that distributes individuals to the two queueing systems

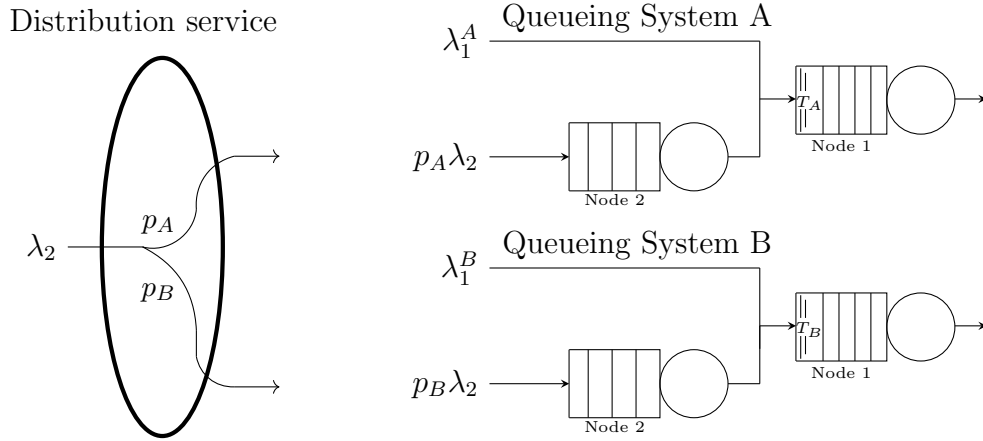


Figure 4.7: A diagrammatic representation of the game theoretic model. Individuals arrive at the distribution service at a rate of  $\lambda_2$  and then a proportion of them are distributed to Queueing system A ( $p_A$ ) and the remaining proportion to Queueing system B ( $p_B$ ) so that  $p_A + p_B = 1$ . The corresponding arrival rates of type 2 individuals to Queueing systems A and B are thus given by:  $p_A \lambda_2$  and  $p_B \lambda_2$ .

Each player has their own objective which they aim to optimise. More specifically, the queueing systems' objective is captured by an upper bound of the time that a fixed proportion of individuals spend in the system, while the distribution service aims to minimise the time that its individuals stay blocked at each queueing system's node 2. These objectives are more formally defined in Section 4.3.2. The parameters of the game theoretic model are:

- $\lambda_2$ : The arrival rate of type 2 individuals arriving at the distribution service that need to be distributed to the queueing systems
- $\lambda_{1_i}$ : The arrival rate of type 1 individuals to queueing system  $i \in \{A, B\}$
- $\mu_i$ : The service rate of individuals at queueing system  $i \in \{A, B\}$
- $C_i$ : The number of servers in queueing system  $i \in \{A, B\}$
- $T_i$ : The strategy that queueing system  $i \in \{A, B\}$  chooses to play which corresponds to the threshold at which they start blocking type 2 individuals at node 2.
- $N_i$ : The total capacity of node 1 in queueing system  $i \in \{A, B\}$
- $M_i$ : The total capacity of node 2 in queueing system  $i \in \{A, B\}$
- $t$ : The time target for both queueing systems
- $\alpha \in [0, 1]$ : Weight of blocking time and lost individuals (defined in equa-



tion (4.22))

- $\hat{P}$ : is the percentage target of individuals that need to be within that target (this is set to 95% unless otherwise stated)

### 4.3.2 Strategies

Each player is given a predetermined set of strategies from which to choose. The strategies of the two queueing systems are the range of thresholds that they can choose from. In essence the strategy space of queueing system  $A$  is the set of integers from 1 to the capacity of node 1 in queueing system  $A$  ( $N_A$ ), while the strategy space of queueing system  $B$  is the set of integers from 1 to the capacity of node 1 in queueing system  $B$  ( $N_B$ ).

$$T_A \in \{1, 2, \dots, N_A\} \quad \text{and} \quad T_B \in \{1, 2, \dots, N_B\} \quad (4.16)$$

In essence this means that for either queueing system ( $A$  or  $B$ ), every strategy choice generates a different queueing network of the form that is described in Section 3. In other words, different strategies are equivalent to different thresholds which is one of the core parameters of the queueing network. Consider queueing system  $A$  as one of the three players of the game theoretic model with node 1 capacity of  $N_A = 6$  and node 2 capacity of  $M_A = 3$ . The strategy space of queueing system  $A$  is then  $T_A \in \{1, 2, 3, 4, 5, 6\}$ . Every possible value of  $T_A$  corresponds to a different queueing network. Figures 4.8 - 4.13 show all possible Markov chain models that arise from such a strategy space.

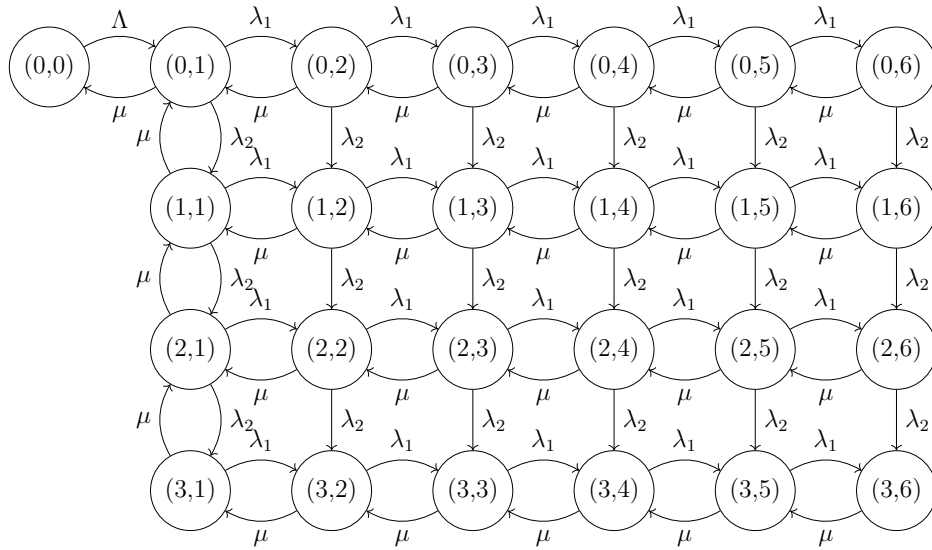


Figure 4.8: The Markov chain model that will be generated when queueing system  $A$  chooses to play a strategy of  $T_A = 1$ .

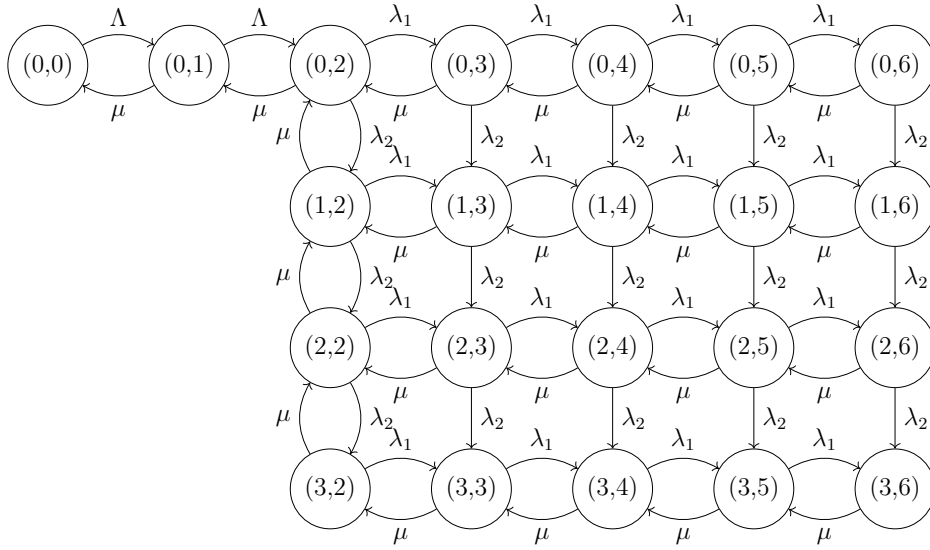


Figure 4.9: The Markov chain model that will be generated when queueing system  $A$  chooses to play a strategy of  $T_A = 2$ .

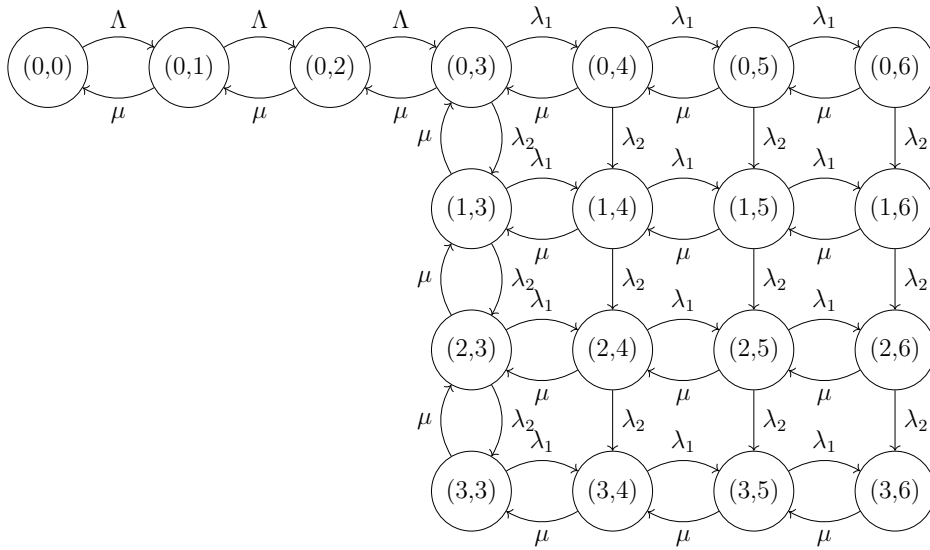


Figure 4.10: The Markov chain model that will be generated when queueing system  $A$  chooses to play a strategy of  $T_A = 3$ .

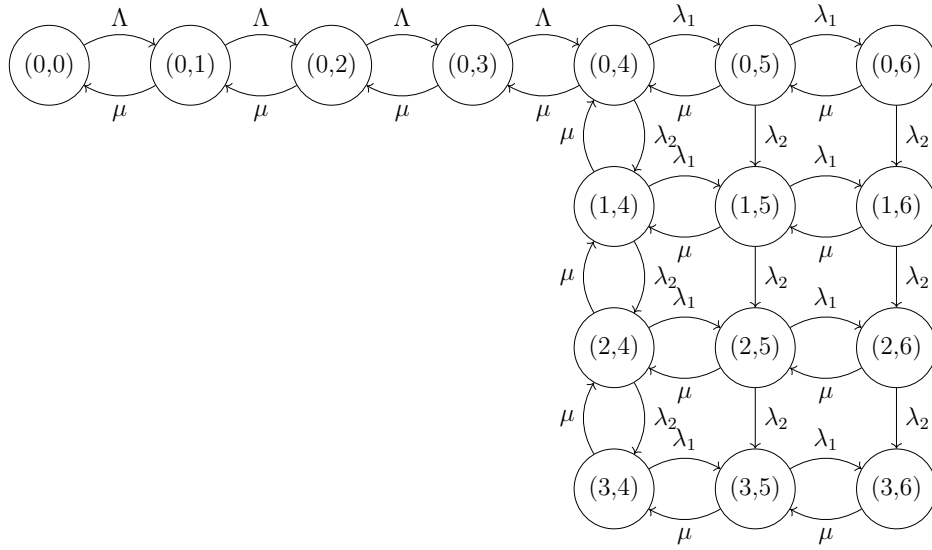


Figure 4.11: The Markov chain model that will be generated when queueing system  $A$  chooses to play a strategy of  $T_A = 4$ .

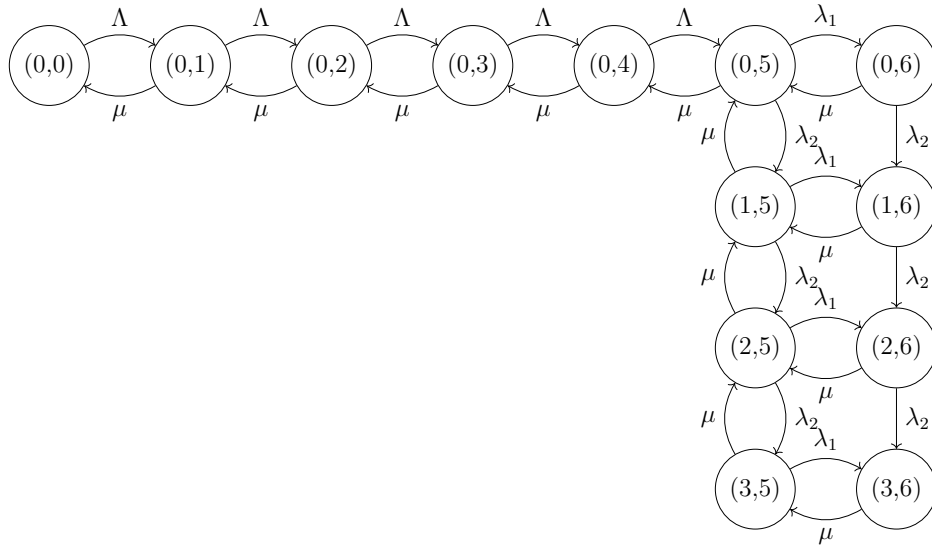


Figure 4.12: The Markov chain model that will be generated when queueing system  $A$  chooses to play a strategy of  $T_A = 5$ .

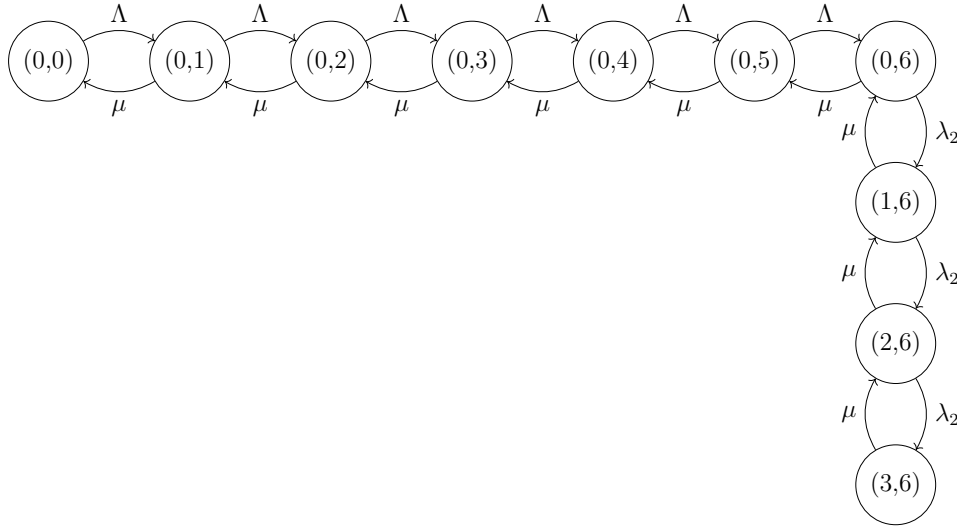


Figure 4.13: The Markov chain model that will be generated when queueing system  $A$  chooses to play a strategy of  $T_A = 6$ .

Each of these Markov chains generate unique performance measures, state probabilities and overall queueing network behaviour. By varying the threshold  $T_A$  from 1 to 6, the queueing network of queueing system  $A$  has a significant change in the mean waiting time and in the mean blocking time. As a matter of fact, as the threshold is increased the mean waiting time is non-increasing while the mean blocking time is non-decreasing. Having a low threshold means that queueing system  $A$  will block type 2 individuals more often and thus the mean blocking time will be higher and hence the mean waiting time will be lower. The same logic applies to queueing system  $B$ , with  $T_B \in \{1, 2, \dots, N_B\}$ .

The strategy space of the distribution service, which is the third player of the game is the range of all possible ways to distribute individuals to the two queueing systems. That is the proportions of individuals to send to queueing system  $A$  and the proportion of individuals to send to queueing system  $B$  ( $p_A, p_B$ ).

$$\begin{aligned} p_A &\in [0, 1] \quad \text{and} \quad p_B \in [0, 1] \\ \text{such that} \quad p_A + p_B &= 1 \end{aligned} \tag{4.17}$$

Since  $p_B$  is dependent on  $p_A$  and vice versa, equation (4.17) can be further simplified so that the strategy space is defined solely by  $p_A$ .

$$p_A \in [0, 1] \quad \text{and} \quad p_B = 1 - p_A \tag{4.18}$$

Similar to the strategy space of the queueing networks, the strategy space of the distribution service is also translated back into the queueing network. Different values of  $p_A$  and  $p_B$ , directly affect the arrival rates of type 2 individuals to the two queueing systems. In fact the arrival rate of type 2 individuals to queueing system  $A$  is  $p_A\lambda_2$  and to queueing system  $B$  is  $p_B\lambda_2$ . Thus by increasing  $p_A$  both the mean waiting time and the mean blocking time of queueing system  $A$  will increase but equivalently will decrease for queueing system  $B$ .

### 4.3.3 Payoffs

Consider the three players of the game: queueing system  $A$ , queueing system  $B$  and the distribution service. Apart from a strategy space each player also has some objective that they would want to either minimise or maximise.

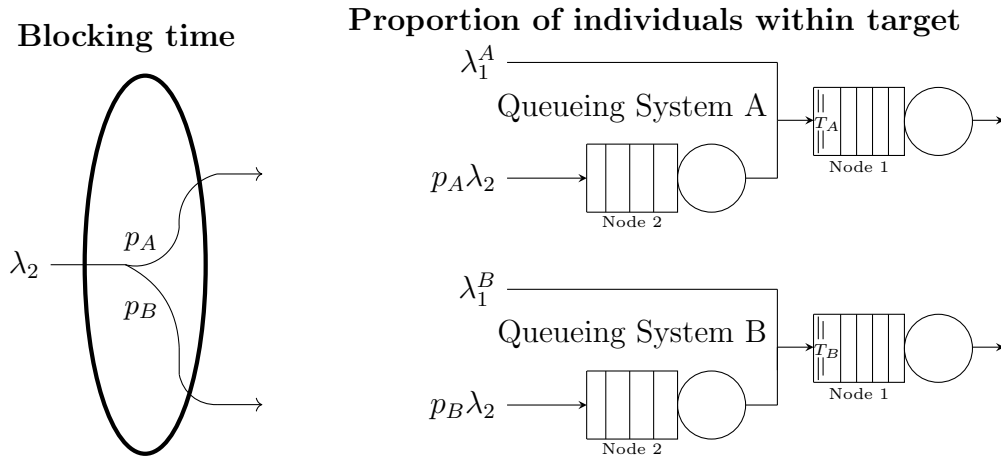


Figure 4.14: A diagrammatic representation of the game theoretic model listing the performance measures that correspond to each player's utilities.

The payoffs of the players are directly related to the performance measures of the queueing networks. More specifically the utilities of the queueing systems focus on the proportion of individuals whose waiting time and service time falls within a predefined time target, where this is defined in Section 3.4.3. Similarly the utilities of the distribution service focus on the blocking time of the queueing systems defined in Section 3.4.2.

Consider the queueing system players first. The objective of either queueing system should be to pick a threshold so that their own mean waiting time is minimised.

$$\arg \min_{T_i} \mathbb{E}[W_i] \quad i \in A, B \quad (4.19)$$

The mean waiting time is defined in Section 3.4.1 as the average time that an individual spends in node 1. Although this might seem like a sensible objective at first glance, it is not necessarily a realistic one. By using the utility function of equation (4.19) the queueing systems best response will always be to pick strategy of  $T = 1$ . A threshold of  $T = 1$  will always result in the lowest mean waiting time since it will prevent type 2 individuals from entering node 1 as long as there is at least one individual in node 1. Depending on the number of available servers this might be true for other values of  $T$  as well. For example if there is one available server in node 1, then a threshold of  $T = 1$  will always result in a mean waiting time of zero for type 2 individuals. That is because type 2 individuals can only enter node 1 if there is nobody else in node 1, which means that there will always be a free server for type 2 individuals. Similarly if there are two available servers in node 1, then a threshold of either  $T = 1$  or  $T = 2$  will also result in a mean waiting time of zero for type 2 individuals.

Therefore a more sophisticated objective that doesn't force the queueing systems to pick the lowest threshold is also considered. The new objective function is defined as:

$$\arg \max_{T_i} 1 - \left( \hat{P} - P(W_i < t) \right)^2 \quad i \in A, B \quad (4.20)$$

where  $W$  is the waiting time of a potential individual,  $t$  is the time target and  $\hat{P}$  is the percentage target of individuals that need to be within that target. In other words, their aim is to find the threshold that minimises the difference between the probability  $P(W_i < t)$  and the percentage goal, or maximise its negation.

The third player, the distribution service, has its own choices to make and its own goals to satisfy. The strategy set of the third player is a proportion  $0 \leq p_A \leq 1$  that corresponds to the proportion of individuals to send to queueing system A (defined in equation (4.18)). The choice of  $p_A$  and  $p_B$ , are based on minimising any potential blockages that may occur, given the pair of thresholds chosen by the two queueing systems. Thus, its objective is to minimise the blocked time of the individuals ( $B_A$  and  $B_B$ ) that they send to queueing systems  $A$  and  $B$ .

Apart from the time being blocked, an additional aspect that may affect the decision of the distribution service is the probability that an individual becomes lost to a queueing system. A type 2 individual can become lost to a queueing system if they arrive in a queueing system and node 2 is at its maximum capacity  $M$ . Therefore, the probability that an individual is lost to queueing system  $i$  is

given by  $L_i$  where  $i \in \{A, B\}$ :

$$L_i = \sum_{\substack{(u,v) \in S \\ u=M}} \pi_{(u,v)} \quad (4.21)$$

where  $S$  is the set of all states in queueing system  $i$ ,  $M$  is the maximum capacity of node 2 and  $\pi_{(u,v)}$  is the probability of being in state  $(u, v)$  (as defined in Section 3).

For each queueing system, there is a penalty of sending a proportion of  $p_i$  individuals to that queueing system. That penalty is given by:

$$\alpha L_i(p_i) + (1 - \alpha) B_i(p_i), \quad i \in A, B \quad (4.22)$$

Equation (4.22) can be used to capture a mixture between the two objectives  $L_i$  and  $B_i$  where  $i \in \{A, B\}$ . Here,  $\alpha$  represents the “weight” of each objective [58], where a high  $\alpha$  indicates a higher weight on the proportion of lost individuals and a smaller  $\alpha$  a higher weight on the time blocked. In fact, the best response of the distribution service can be found by equating the penalty of sending  $p_A$  individuals to queueing system  $A$  and the penalty of sending  $p_B$  individuals to queueing system  $B$  (where  $p_B = 1 - p_A$ ).

$$\alpha L_A(p_A) + (1 - \alpha) B_A(p_A) = \alpha L_B(p_B) + (1 - \alpha) B_B(p_B) \quad (4.23)$$

There are some cases where the best response of the distribution service is to distribute all individuals to one of the two queueing systems. For example, if queueing system  $A$  has a low threshold and queueing system  $B$  has a high threshold, then the distribution service’s best response may be to send all individuals to queueing system  $B$ . Thus equation (4.23) can be modified to be:

$$\begin{aligned} O(p_A; T_A, T_B) &= \alpha L_A(p_A) + (1 - \alpha) B_A(p_A) - \alpha L_B(1 - p_A) - (1 - \alpha) B_B(1 - p_A) \\ \arg \min_{p_A} |O(p_A; T_A, T_B)| & \end{aligned} \quad (4.24)$$

The choice of  $p_A$  and  $p_B$  rely solely on equation (4.24). Note that the current system is modelled using unobservable queues which is not necessarily an unrealistic approach [126]. Another approach would be to allow the distribution service

to observe the state of each queueing system before sending each individual to either of them and making a decision based on that state. There are several other factors that can affect the routing of the individuals that are outside the scope of this thesis. For example the distance from the distribution service to each queueing system or even the priority level of each individual may be a significant factor that affects the distribution service's decision.

#### 4.3.4 Imperfect information extensive form game

The game can be modelled as an imperfect information extensive form game as described in Section 4.2.5. The strategies and payoffs of all players can be put together to form the following game tree:

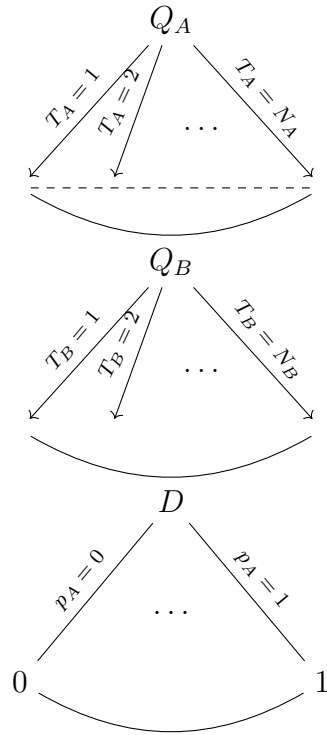


Figure 4.15: Imperfect information extensive form game between the distribution service and the 2 queueing systems

The game tree is a representation of all possible sequences of decisions that can be made by the players. Initially, queueing system  $A$  chooses which threshold  $T_A$  to play out of  $N_A$  possible choices. Then, queueing system  $B$  chooses its own threshold  $Q_B$  without knowing the threshold chosen by queueing system  $A$ . Note here that it doesn't matter which queueing system chooses its threshold first since the other queueing system will always choose its threshold independently of the first queueing system's choice. Afterwards, the distribution service chooses the proportion of individuals to send to queueing system  $A$ , from the continuous



strategy space of  $0 \leq p_A \leq 1$ . The distribution service has all the information about the state that the game is at when it's choosing the value of  $p_A$ . Thus, the distribution service's choice of  $p_A$  is based on the thresholds chosen by the queueing systems. The payoffs for all players are then calculated.

## 4.4 Methodology

A big part of the methodology that is used to solve the game requires the use of backwards induction. Backwards induction is a method that is used to solve a game by starting at the terminal nodes and working backwards to the root node [155]. The terminal nodes from Figure 4.15 are the nodes that are connected to the choice of the distribution service. In essence, by working backwards from the choice of the distribution service and then to the choices of the queueing systems, which happen simultaneously, the game can be solved. Furthermore, by finding the distribution's service response for all possible pairs of strategies that the two queueing systems can choose from, the game can be reduced to a two-player Normal form game.

### 4.4.1 Distribution service and Brent's algorithm

Form the distribution service's perspective all information is known and can be used to find the best possible strategy to maximise their payoff. In fact, having the two strategy choices of the two queueing systems the distribution service can find the optimal strategy that satisfies equation (4.24). Consider the pair of strategies  $(T_A^*, T_B^*)$  that correspond to a possible strategy choice of queueing system  $A$  and queueing system  $B$ . The distribution service can then find the best strategy by solving equation (4.24) for  $T_A = T_A^*$  and  $T_B = T_B^*$ . The particular numerical algorithm used for this is Brent's algorithm [22].

Brent's algorithm is a root-finding algorithm which combines the bisection method [34], the secant method [115] and inverse quadratic interpolation [43]. The algorithm is used to find the root  $x^*$  of a function  $f(x)$  from within the interval  $[a, b]$  such that  $f(a) < f(x^*) < f(b)$ . One of the requirements for Brent's algorithm is  $f(a)f(b) < 0$ . In other words the function must change sign within the interval  $[a, b]$ .

Consider equation (4.24). Under the assumption that  $O(p_A; T_A, T_B)$  is either non-increasing or non-decreasing in  $p_A$ , the root can be found by using Brent's algorithm for  $p_A \in [0, 1]$ .

$$O(p_A; T_A, T_B) = \alpha L_A(p_A) + (1 - \alpha)B_A(p_A) - \alpha L_B(1 - p_A) - (1 - \alpha)B_B(1 - p_A)$$

For this particular scenario, the function  $f(p_A)$  needs to have a different sign for  $f(p_A = 0)$  and  $f(p_A = 1)$  so that  $f(a)f(b) < 0$  is satisfied. In the case  $f(a)f(b) \geq 0$ , Brent's algorithm cannot be used. Instead, the value of  $p_A$  becomes:

$$p_A = \begin{cases} 1 & \text{if } f(0) < 0 \text{ and } f(1) < 0 \\ 0 & \text{if } f(0) > 0 \text{ and } f(1) > 0 \end{cases} \quad (4.25)$$

The first case of equation (4.25) corresponds to the event where both  $f(0)$  and  $f(1)$  are negative. Therefore, for all values of  $p_A \in [0, 1]$  the objective function is negative, which means that:

$$\alpha L_A(p_A) + (1 - \alpha)B_A(p_A) < \alpha L_B(p_B) - (1 - \alpha)B_B(p_B), \quad \text{for all } p_A \in [0, 1]$$

Thus, the distribution service's best response would be to send all individuals to queueing system  $A$  ( $p_A = 1, p_B = 0$ ). Similarly, the second case of equation (4.25) corresponds to the event where for all values of  $p_A \in [0, 1]$  the objective function is positive, which means that:

$$\alpha L_A(p_A) + (1 - \alpha)B_A(p_A) > \alpha L_B(p_B) - (1 - \alpha)B_B(p_B), \quad \text{for all } p_A \in [0, 1]$$

Equivalently, this indicates that the distribution service's best response would be to send all individuals to queueing system  $B$  ( $p_A = 0, p_B = 1$ ). Therefore, the methodology that is used to find the best  $p_A$  that satisfies equation (4.24) can be calculated in the following way:

$$p_A = \begin{cases} 1, & \text{if } O(0) < 0 \text{ and } O(1) < 0 \\ 0, & \text{if } O(0) > 0 \text{ and } O(1) > 0 \\ \text{Use Brent's algorithm,} & \text{if } O(0)O(1) < 0 \end{cases} \quad (4.26)$$

where  $O(p_A)$  is the objective function of the distribution service described in

equation (4.24).

#### 4.4.1.1 Examples

Consider a distribution service whose arrival rate of type 2 individuals is  $\lambda_2 = 4$  and the ‘weight’ is  $\alpha = 0.2$ . Additionally, let queueing system  $A$  and queueing system  $B$  have the parameters shown in Table 4.2.

Table 4.2: Parameter values for Brent’s method examples.

$D$		Queueing system $A$						Queueing system $B$					
$\lambda_2$	$\alpha$	$\lambda_1^A$	$\mu^A$	$C^A$	$T^A$	$N^A$	$M^A$	$\lambda_1^B$	$\mu^B$	$C^B$	$T^B$	$N^B$	$M^B$
4	0.2	2	3	3	8	15	10	1	1	3	10	10	5

The distribution service’s best response for this particular example can be found at the intersection of the two decision values of the two queueing systems over  $p_A$ . Figure 4.16 illustrates the distribution service’s best response for this particular example.

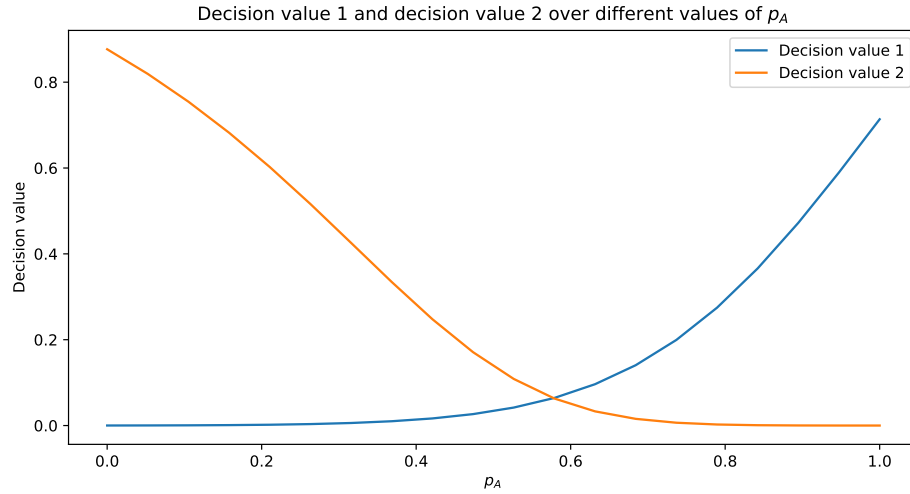


Figure 4.16: Decision values for queueing system  $A$  and queueing system  $B$  where (*decision value 1*) =  $\alpha L_A(p_A) + (1 - \alpha)B_A(p_A)$  and (*decision value 2*) =  $\alpha L_B(p_B) - (1 - \alpha)B_B(p_B)$

In order to apply Brent’s algorithm to the current example the differences between the two decision values need to be calculated. Figure 4.17 shows that the value of  $p_A$  that the distribution service should pick is where the function crosses the  $x$ -axis.

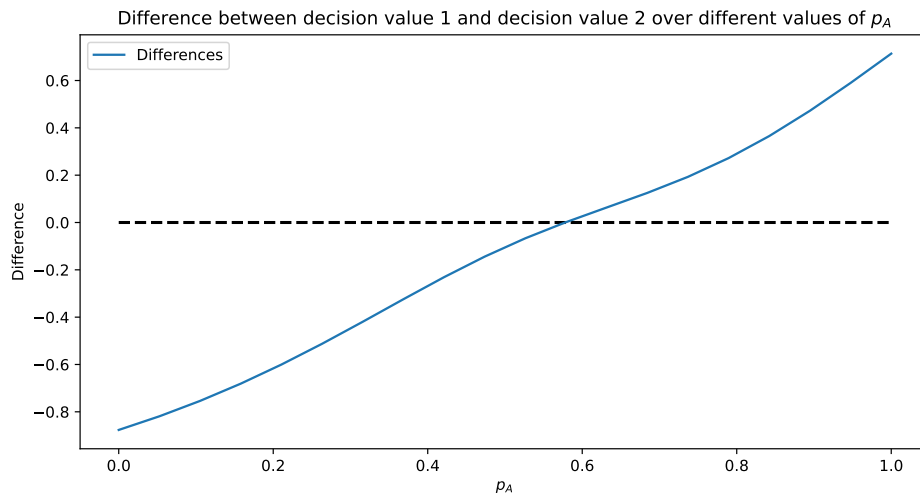


Figure 4.17: Visualisation of Brent's algorithm showing the differences between the two decision values and the point at which the function crosses the  $x$ -axis

In fact, the value of  $p_A$  that the distribution service should pick, for this particular example, is  $p_A = 0.58$ . That is the point at which the line of the difference between the two decision values crosses the  $x$ -axis.

Consider now the same parameters as in the previous example, for different values of the service rate of queueing system  $A$ ,  $\mu^A = \{1, 1.5, 2, 2.5, 3\}$ .

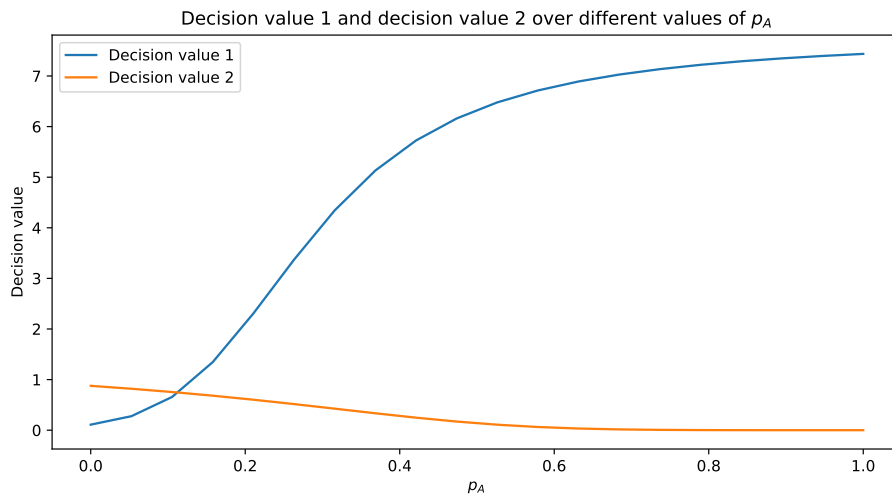


Figure 4.18: Brent's algorithm example where the service parameter of queueing system  $A$  is  $\mu^A = 1$

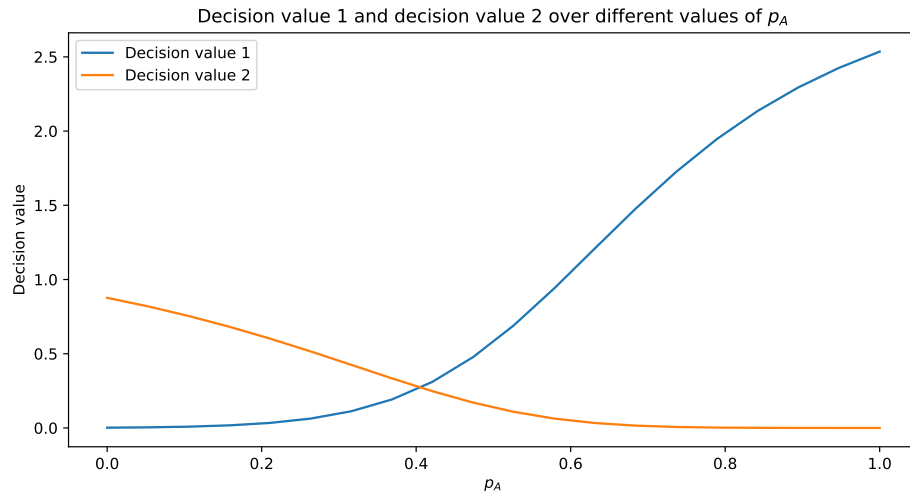


Figure 4.19: Brent's algorithm example where the service parameter of queueing system A is  $\mu^A = 1.5$

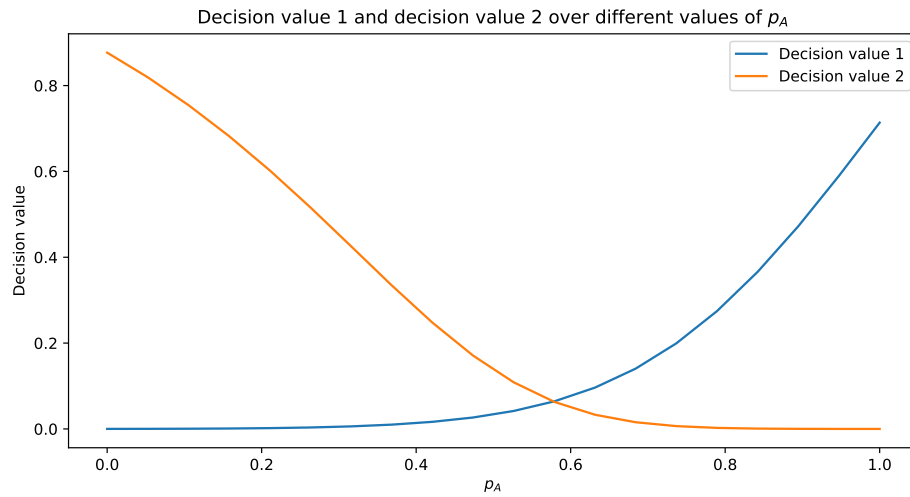


Figure 4.20: Brent's algorithm example where the service parameter of queueing system A is  $\mu^A = 2$

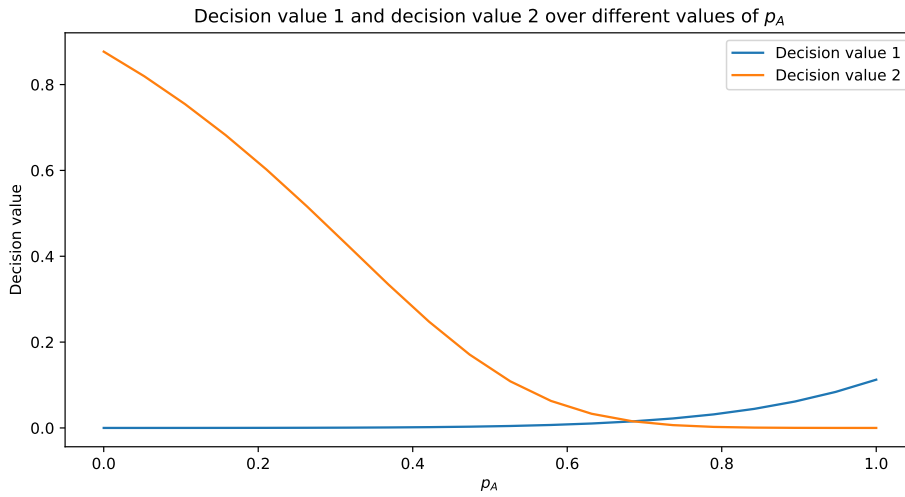


Figure 4.21: Brent's algorithm example where the service parameter of queueing system A is  $\mu^A = 2.5$



Figure 4.22: Brent's algorithm example where the service parameter of queueing system A is  $\mu^A = 3$

It can be seen from Figures 4.18 - 4.22 that as the service rate of queueing system A increases, the intersection point of the two decision values moves from  $p_A = 0$  towards  $p_A = 1$ .

In addition consider a different example with the same parameters as before but by increasing the threshold of queueing system A from  $T^A = 8$  to  $T^A = 10$  and decreasing the threshold of queueing system B from  $T^B = 10$  to  $T^B = 2$ . Figure 4.23 shows the decision values that correspond to the two queueing systems of the new example.

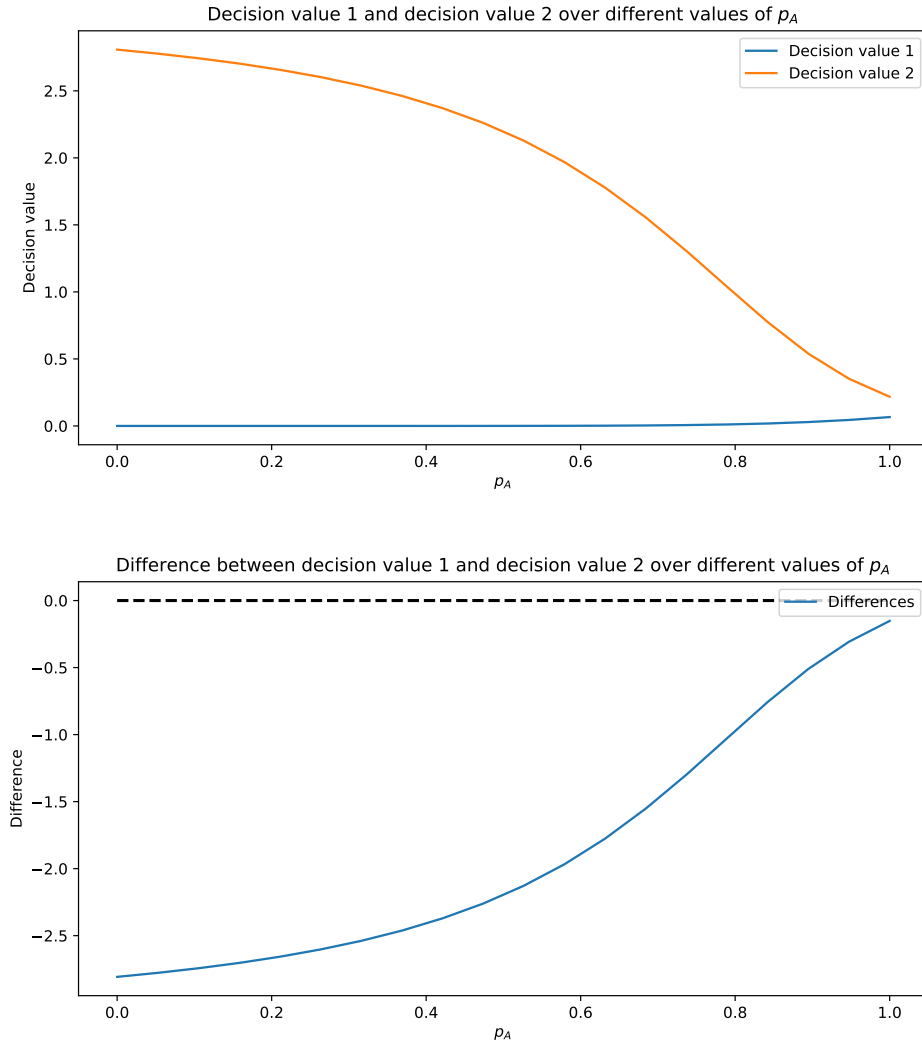


Figure 4.23: Decision values for queueing system  $A$  and queueing system  $B$  (top) and the differences between them (bottom).

It can be observed that for all values of  $p_A$  the decision value of queueing system  $A$  is less than the decision value of queueing system  $B$ . Since the differences of them don't pass through the x-axis within the interval  $[0, 1]$ , Brent's algorithm cannot be used since  $f(0)f(1) < 0$ . Therefore, using equation (4.26) the distribution service's best response should be  $p_A = 1$ .

#### 4.4.1.2 Implementation

The first part of the implementation of the distribution service's best response is to calculate the difference between the decision values of the two queueing systems. Function `get_mean_blocking_difference_using_markov` defined in 4.8 is the python implementation of the first part of equation (4.24).

```

>>> import ambulance_game as abg
>>> def get_mean_blocking_difference_using_markov(
...     prop_1,
...     lambda_2,
...     lambda_1_1,
...     lambda_1_2,
...     mu_1,
...     mu_2,
...     num_of_servers_1,
...     num_of_servers_2,
...     threshold_1,
...     threshold_2,
...     system_capacity_1,
...     system_capacity_2,
...     buffer_capacity_1,
...     buffer_capacity_2,
...     alpha=0,
... ):
...     """
...     Get a weighted mean blocking difference between two systems. This
...     function is to be used as a routing function to find the point at
...     which it is set to 0. This function calculates:
...         -  $a*(1 - P(A_1)) + (1 - a)*B_1$ 
...         -  $a*(1 - P(A_2)) + (1 - a)*B_2$ 
...     and returns their difference.
...     Parameters
...     -----
...     prop_1 : float
...         The proportion of class 2 individuals to distribute to the first
...         system
...     lambda_2 : float
...         The overall arrival rate of class 2 individuals for both systems
...     lambda_1_1 : float
...         The arrival rate of class 1 individuals in the first system
...     lambda_1_2 : float
...         The arrival rate of class 1 individuals in the second system
...     mu_1 : float
...     mu_2 : float
...     num_of_servers_1 : int
...     num_of_servers_2 : int
...     threshold_1 : int
...     threshold_2 : int
...     system_capacity_1 : int
...     system_capacity_2 : int
...     buffer_capacity_1 : int
...     buffer_capacity_2 : int
...     Returns
...     -----
...     float
...         The weighted mean difference between the decision values of the
...         two systems
...     """
...     lambda_2_1 = prop_1 * lambda_2
...     lambda_2_2 = (1 - prop_1) * lambda_2
...
...     mean_blocking_time_1 = abg.markov.
get_mean_blocking_time_using_markov_state_probabilities(
...         lambda_2=lambda_2_1,

```



```

...     lambda_1=lambda_1_1,
...     mu=mu_1,
...     num_of_servers=num_of_servers_1,
...     threshold=threshold_1,
...     system_capacity=system_capacity_1,
...     buffer_capacity=buffer_capacity_1,
... )
...     mean_blocking_time_2 = abg.markov.
get_mean_blocking_time_using_markov_state_probabilities(
...         lambda_2=lambda_2_2,
...         lambda_1=lambda_1_2,
...         mu=mu_2,
...         num_of_servers=num_of_servers_2,
...         threshold=threshold_2,
...         system_capacity=system_capacity_2,
...         buffer_capacity=buffer_capacity_2,
...     )
...     prob_accept_1 = abg.markov.
get_accepting_proportion_of_class_2_individuals(
...         lambda_1=lambda_1_1,
...         lambda_2=lambda_2_1,
...         mu=mu_1,
...         num_of_servers=num_of_servers_1,
...         threshold=threshold_1,
...         system_capacity=system_capacity_1,
...         buffer_capacity=buffer_capacity_1,
...     )
...     prob_accept_2 = abg.markov.
get_accepting_proportion_of_class_2_individuals(
...         lambda_1=lambda_1_2,
...         lambda_2=lambda_2_2,
...         mu=mu_2,
...         num_of_servers=num_of_servers_2,
...         threshold=threshold_2,
...         system_capacity=system_capacity_2,
...         buffer_capacity=buffer_capacity_2,
...     )
...     decision_value_1 = (
...         alpha * (1 - prob_accept_1) + (1 - alpha) * mean_blocking_time_1
...     )
...     decision_value_2 = (
...         alpha * (1 - prob_accept_2) + (1 - alpha) * mean_blocking_time_2
...     )
...     return decision_value_1 - decision_value_2

```

Code snippet 4.8: Function that gets the mean blocking difference using the Markov chain model

Using the same example as in section 4.4.1.1 the differences between the two decision values can be calculated as shown in 4.9. Note, that the outcome of the function corresponds to the point of the line in Figure 4.17 where  $p_A = 0.5$ .

```

>>> import numpy as np
>>>
>>> lambda_1_A = 2
>>> mu_A = 2

```

```

>>> num_of_servers_A = 3
>>> threshold_A = 8
>>> system_capacity_A = 15
>>> buffer_capacity_A = 10
>>> lambda_1_B = 1
>>> mu_B = 1
>>> num_of_servers_B = 3
>>> threshold_B = 10
>>> system_capacity_B = 10
>>> buffer_capacity_B = 5
>>>
>>> lambda_2 = 4
>>> alpha = 0.2
>>> p_A = 0.5
>>>
>>> np.round(get_mean_blocking_difference_using_markov(
...     prop_1=p_A,
...     lambda_2=lambda_2,
...     lambda_1_1=lambda_1_A,
...     lambda_1_2=lambda_1_B,
...     mu_1=mu_A,
...     mu_2=mu_B,
...     num_of_servers_1=num_of_servers_A,
...     num_of_servers_2=num_of_servers_B,
...     threshold_1=threshold_A,
...     threshold_2=threshold_B,
...     system_capacity_1=system_capacity_A,
...     system_capacity_2=system_capacity_B,
...     buffer_capacity_1=buffer_capacity_A,
...     buffer_capacity_2=buffer_capacity_B,
...     alpha=alpha,
... ), 8)
-0.10424302

```

Code snippet 4.9: Using the function defined in 4.8 to calculate the difference between the decision values of the two systems at  $p_A = 0.5$

In addition function `calculate_class_2_individuals_best_response` uses an implementation of Brent's algorithm, implemented by the `scipy` library, to find the point at which the difference between the two decision values is set to 0.

```

>>> import scipy.optimize
>>> def calculate_class_2_individuals_best_response(
...     lambda_2,
...     lambda_1_1,
...     lambda_1_2,
...     mu_1,
...     mu_2,
...     num_of_servers_1,
...     num_of_servers_2,
...     threshold_1,
...     threshold_2,
...     system_capacity_1,
...     system_capacity_2,
...     buffer_capacity_1,
...     buffer_capacity_2,
...     lower_bound=0.01,

```

```

...     upper_bound=0.99,
...     alpha=0,
...     xtol=1e-04,
...     rtol=8.9e-16,
... ):
...     """
...     Obtains the optimal distribution of class 2 individuals such that the
...     blocking times in the two systems are identical and thus minimised.
...     The brentq function is used which is an algorithm created to find the
...     root of a function that combines root bracketing, bisection, and
...     inverse quadratic interpolation. In this specific example the root to
...     be found is the difference between the blocking times of two systems.
...     In essence the brentq algorithm attempts to find the value of prop_1
...     where the difference is zero.
...
...     Parameters
...     -----
...     lower_bound : float, optional
...         The lower bound of p_1, by default 0.01
...     upper_bound : float, optional
...         The upper bound of p_1, by default 0.99
...     routing_function : function, optional
...         The function to find the root of
...     Returns
...     -----
...     float
...         The value of p_1 such that routing_function = 0
...     """
...
...     routing_function = get_mean_blocking_difference_using_markov
...     check_1 = routing_function(
...         prop_1=lower_bound,
...         lambda_2=lambda_2,
...         lambda_1_1=lambda_1_1,
...         lambda_1_2=lambda_1_2,
...         mu_1=mu_1,
...         mu_2=mu_2,
...         num_of_servers_1=num_of_servers_1,
...         num_of_servers_2=num_of_servers_2,
...         threshold_1=threshold_1,
...         threshold_2=threshold_2,
...         system_capacity_1=system_capacity_1,
...         system_capacity_2=system_capacity_2,
...         buffer_capacity_1=buffer_capacity_1,
...         buffer_capacity_2=buffer_capacity_2,
...         alpha=alpha,
...     )
...     check_2 = routing_function(
...         prop_1=upper_bound,
...         lambda_2=lambda_2,
...         lambda_1_1=lambda_1_1,
...         lambda_1_2=lambda_1_2,
...         mu_1=mu_1,
...         mu_2=mu_2,
...         num_of_servers_1=num_of_servers_1,
...         num_of_servers_2=num_of_servers_2,
...         threshold_1=threshold_1,
...         threshold_2=threshold_2,
...         system_capacity_1=system_capacity_1,

```

```

...     system_capacity_2=system_capacity_2,
...     buffer_capacity_1=buffer_capacity_1,
...     buffer_capacity_2=buffer_capacity_2,
...     alpha=alpha,
... )
...
... if check_1 >= 0 and check_2 >= 0:
...     return 0
... if check_1 <= 0 and check_2 <= 0:
...     return 1
...
... brentq_arguments = (
...     lambda_2,
...     lambda_1_1,
...     lambda_1_2,
...     mu_1,
...     mu_2,
...     num_of_servers_1,
...     num_of_servers_2,
...     threshold_1,
...     threshold_2,
...     system_capacity_1,
...     system_capacity_2,
...     buffer_capacity_1,
...     buffer_capacity_2,
...     alpha,
... )
...
... optimal_split = scipy.optimize.brentq(
...     routing_function,
...     a=lower_bound,
...     b=upper_bound,
...     args=brentq_arguments,
...     xtol=xtol,
...     rtol=rtol,
... )
... return optimal_split

```

Code snippet 4.10: Using Brent’s algorithm to find the point where the differences of the two decision values are zero.

The code in 4.11 uses function `calculate_class_2_individuals_best_response` to find the optimal split of class 2 individuals between the two queueing systems. The same set of parameters are used as in the example in Section 4.4.1.1. Note that this is the value of  $p_A$  for which the line of Figure 4.17 crosses the  $x$ -axis.

```

>>> np.round(calculate_class_2_individuals_best_response(
...     lambda_2=lambda_2,
...     lambda_1_1=lambda_1_A,
...     lambda_1_2=lambda_1_B,
...     mu_1=mu_A,
...     mu_2=mu_B,
...     num_of_servers_1=num_of_servers_A,
...     num_of_servers_2=num_of_servers_B,
...     threshold_1=threshold_A,
...     threshold_2=threshold_B,
...     system_capacity_1=system_capacity_A,

```

```

...     system_capacity_2=system_capacity_B,
...     buffer_capacity_1=buffer_capacity_A,
...     buffer_capacity_2=buffer_capacity_B,
...     alpha=alpha,
... ), 8)
0.5778495

```

Code snippet 4.11: Calculating the optimal split between type 1 and type 2 individuals. Note that the outcome of the function is the value of  $p_A$  while the value of  $p_B$  is given by  $1 - p_A$ .

#### 4.4.1.3 Brent's algorithm - tolerance sensitivity analysis

The implementation of Brent's algorithm uses the `brentq` function from SciPy. The function receives two essential arguments; another function for which to find the root of, and the interval in which the root is located. In addition to these two arguments, the function also receives two optional arguments; `xtol` and `rtol` [152]. These two parameters are the ones that define the tolerance of the algorithm. In other words, the smaller the tolerance parameters are, the more accurate the result will be. However, the smaller the tolerance parameters are, the more iterations will be needed to find the root. Therefore, the tolerance parameters are a trade-off between accuracy and computation time.

The documentation of the `brentq` function states that the default values of the tolerance parameters are `xtol=2e-12` and `rtol=8.881784197001252e-16`. Within the `brentq` function these two parameters are used to ensure that `allclose(x, x0, atol=xtol, rtol=rtol) = True`. Function `allclose` is implemented by the `numpy` library [65]. and checks if two arrays are element-wise similar given a certain tolerance. The way the internal mechanisms of the `allclose` function work is that given two values  $a$  and  $b$ , with some absolute tolerance (`atol`) and relative tolerance (`rtol`) parameters, the function returns `True` if:

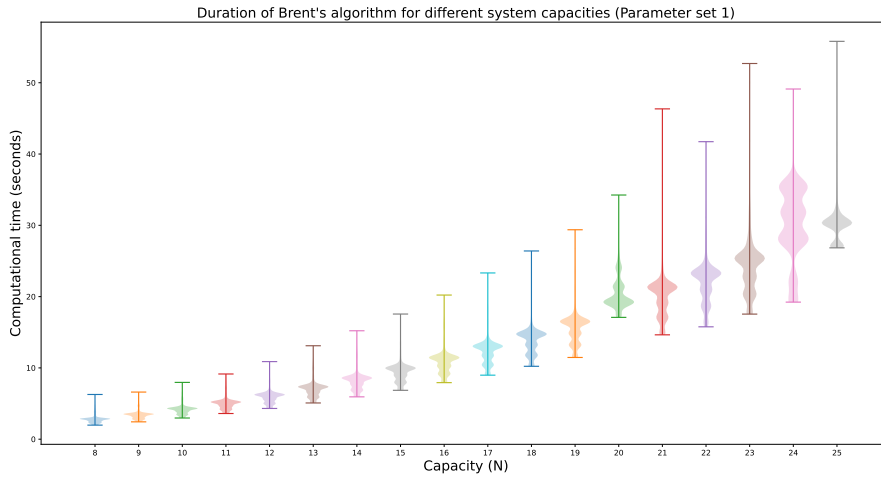
$$|a - b| \leq (\text{atol} + \text{rtol} \times |b|) \quad (4.27)$$

These tolerance parameters are the ones used by Brent's algorithm to determine if the root has been found. In the remainder of this subsection the effect of the absolute tolerance parameter `xtol` will be analysed. To determine the effect of the absolute tolerance parameters on the accuracy and computation time of Brent's algorithm, the algorithm was ran for different set of parameters and different values of the absolute tolerance parameter. The two parameter sets that were used for these experiments are shown in Table 4.3.

Table 4.3: Parameter sets that were used for the tolerance sensitivity analysis.

Distributor				Queueing system $A$					Queueing system $B$				
$\lambda_2$	$\alpha$	$\lambda_1^A$	$\mu^A$	$C^A$	$T^A$	$N^A$	$M^A$	$\lambda_1^B$	$\mu^B$	$C^B$	$T^B$	$N^B$	$M^B$
4	0.2	3	4	2	4	[8, 25]	8	3	3	3	5	8	8
5	0.2	2	3	3	7	[7, 24]	10	2	2	4	10	15	10

Note the system capacity of queueing system  $A$  varies for both parameter sets. For every value of  $N_A$  Brent's algorithm was run for different values of the absolute tolerance parameter `xtol`. The values of the absolute tolerance parameter that were used are: `xtol` = [1e-10, 1e-9, 1e-8, 1e-7, 1e-6, 1e-5, 0.0001, 0.001, 0.01, 0.1]. For each value of the absolute tolerance parameter, the algorithm was run 200 times and the computation time in seconds was recorded for each run. These experiments were run on a computer with an Intel Core i7-1165G7 CPU running on four cores at 2.80 GHz and 16 GB of RAM.

Figure 4.24: Violinplots of the duration of Brent's algorithm for different values of  $N_A$  for the first parameter set.

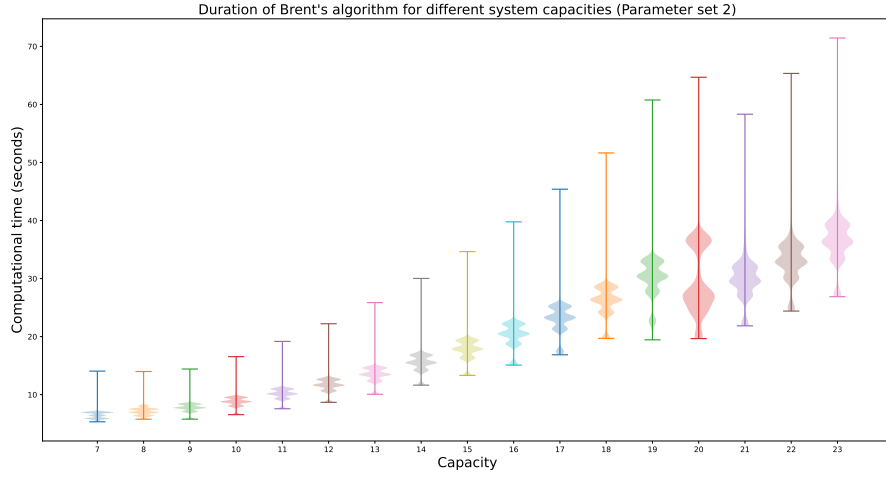


Figure 4.25: Violinplots of the duration of Brent's algorithm for different values of  $N_A$  for the second parameter set.

It can be seen that for both parameter sets in Figures 4.24 and 4.25 the duration of the algorithm is increasing as  $N^A$  increases. Note that the violinplots include all values of the tolerance parameters `xtol` that were used.

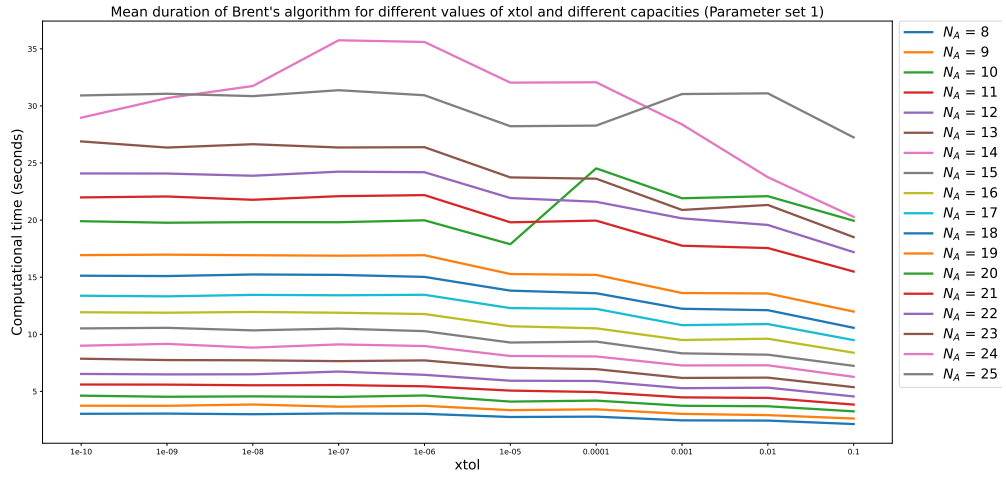


Figure 4.26: Line plots of the duration of Brent's algorithm for different values of  $N_A$  over different values of the absolute tolerance parameter `xtol` for the first parameter set.

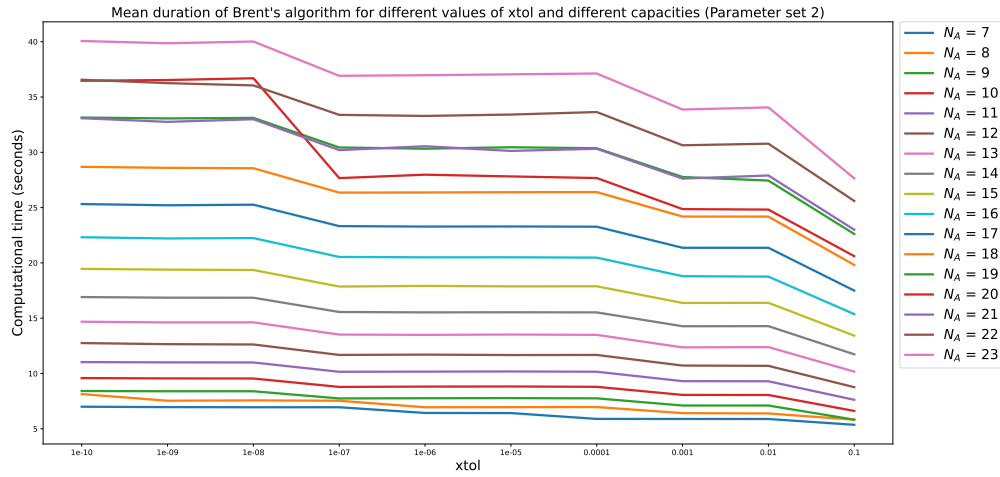


Figure 4.27: Line plots of the duration of Brent's algorithm for different values of  $N_A$  over different values of the absolute tolerance parameter  $\text{xtol}$  for the second parameter set.

The plots of Figure 4.26 and 4.27 show for each value of  $N^A$  how the duration of the algorithm changes as the absolute tolerance parameter  $\text{xtol}$  increases. It can be seen that for both parameter sets the duration of the algorithm on most cases the computational time decreases as the tolerance parameter increases.

#### 4.4.2 Routing Matrix

Section 4.4.1.1 showed how Brent's algorithm can be used to find the best response of the distribution service given the pair of strategies played by the queueing systems  $(T_A, T_B)$ . In order to properly solve the game, best response of the distribution service needs to be calculated for every possible pair of strategies. In essence, one needs to find the values of  $p_A$  and  $p_B$  that correspond to every pair of  $(T_A, T_B)$ , and then use these values to construct the routing matrix. The routing matrix holds the values  $(p_A, p_B)$  which are the proportion of type 2 individuals to send to queueing systems  $A$  and  $B$ . Each pair  $(p_A, p_B)$  can be calculated using equation (4.22), as shown in Section 4.4.1.1, for all possible pairs of thresholds. Thus, the routing matrix is a  $N_A \times N_B$  matrix, where  $N_A$  and  $N_B$  are the capacities of Node 1 for queueing systems  $A$  and  $B$ , respectively.

$$R = \begin{pmatrix} (p_{1,1}^A, p_{1,1}^B) & (p_{1,2}^A, p_{1,2}^B) & \cdots & (p_{1,N_B}^A, p_{1,N_B}^B) \\ (p_{2,1}^A, p_{2,1}^B) & (p_{2,2}^A, p_{2,2}^B) & \cdots & (p_{2,N_B}^A, p_{2,N_B}^B) \\ \vdots & \vdots & \ddots & \vdots \\ (p_{N_A,1}^A, p_{N_A,1}^B) & (p_{N_A,2}^A, p_{N_A,2}^B) & \cdots & (p_{N_A,N_B}^A, p_{N_A,N_B}^B) \end{pmatrix} \quad (4.28)$$



Note that since  $p_{i,j}^A + p_{i,j}^B = 1$  the routing matrix needs only to store one of the two values; either  $p_{i,j}^A$  or  $p_{i,j}^B$ . Thus, the routing matrix  $R$  can be simplified to:

$$R = \begin{pmatrix} p_{1,1}^A & p_{1,2}^A & \cdots & p_{1,N_B}^A \\ p_{2,1}^A & p_{2,2}^A & \cdots & p_{2,N_B}^A \\ \vdots & \vdots & \ddots & \vdots \\ p_{N_A,1}^A & p_{N_A,2}^A & \cdots & p_{N_A,N_B}^A \end{pmatrix} \quad (4.29)$$

#### 4.4.2.1 Example

Using the same example as in Section 4.4.1.1, the routing matrix can be calculated by finding the values of  $p_A$  and  $p_B$  for every possible pair of thresholds  $(T_A, T_B)$ . The arrival rate of type 2 individuals is arrival rate of type 2 individuals is  $\lambda_2 = 4$  and the ‘weight’ is  $\alpha = 0.2$ . The remaining parameters that relate to the two queueing systems are shown in Table 4.4.

Table 4.4: Parameter values for routing matrix example.

Distributor		Queueing system A					Queueing system B				
$\lambda_2$	$\alpha$	$\lambda_1^A$	$\mu^A$	$C^A$	$N^A$	$M^A$	$\lambda_1^B$	$\mu^B$	$C^B$	$N^B$	$M^B$
4	0.2	2	3	3	15	10	1	1	3	10	5

Note that the thresholds are not defined for the routing matrix since they are not constants. In fact the thresholds can take values from 1 to  $N_i$  for each queueing system. Thus,  $T^A \in \{1, 2, \dots, 15\}$  and  $T^B \in \{1, 2, \dots, 10\}$ . The routing matrix is then going to be a  $15 \times 10$  matrix where each entry  $i, j$  consists of the best response of the distribution service when  $Q_A$  plays a strategy of  $T_A = i$  and  $Q_B$  plays a strategy of  $T_B = j$ .

$$R = \begin{pmatrix} 0.59 & 0.22 & 0.16 & 0.15 & 0.15 & 0.15 & 0.15 & 0.14 & 0.13 & 0.06 \\ 0.94 & 0.67 & 0.51 & 0.49 & 0.47 & 0.46 & 0.45 & 0.44 & 0.41 & 0.31 \\ 1.00 & 0.85 & 0.71 & 0.67 & 0.64 & 0.62 & 0.60 & 0.57 & 0.54 & 0.45 \\ 1.00 & 0.86 & 0.74 & 0.70 & 0.67 & 0.64 & 0.62 & 0.60 & 0.57 & 0.48 \\ 1.00 & 0.88 & 0.76 & 0.72 & 0.69 & 0.67 & 0.64 & 0.62 & 0.59 & 0.51 \\ 1.00 & 0.89 & 0.78 & 0.74 & 0.71 & 0.68 & 0.66 & 0.64 & 0.61 & 0.54 \\ 1.00 & 0.90 & 0.79 & 0.75 & 0.72 & 0.70 & 0.68 & 0.66 & 0.63 & 0.56 \\ 1.00 & 0.91 & 0.81 & 0.77 & 0.74 & 0.71 & 0.69 & 0.67 & 0.64 & 0.58 \\ 1.00 & 0.91 & 0.82 & 0.78 & 0.75 & 0.73 & 0.71 & 0.68 & 0.66 & 0.59 \\ 1.00 & 0.92 & 0.83 & 0.80 & 0.76 & 0.74 & 0.72 & 0.70 & 0.67 & 0.61 \\ 1.00 & 0.93 & 0.84 & 0.80 & 0.77 & 0.75 & 0.73 & 0.71 & 0.68 & 0.62 \\ 1.00 & 0.93 & 0.85 & 0.81 & 0.78 & 0.76 & 0.74 & 0.72 & 0.69 & 0.64 \\ 1.00 & 0.95 & 0.87 & 0.83 & 0.80 & 0.78 & 0.75 & 0.73 & 0.71 & 0.65 \\ 1.00 & 0.98 & 0.90 & 0.86 & 0.83 & 0.80 & 0.78 & 0.76 & 0.73 & 0.68 \\ 1.00 & 1.00 & 0.99 & 0.94 & 0.90 & 0.87 & 0.84 & 0.82 & 0.79 & 0.74 \end{pmatrix}$$

Note that the entries of the routing matrix correspond to different pairs of thresholds  $(T_A, T_B)$ . In other words, the entry  $R_{i,j}$  corresponds to the pair  $(T_A = i, T_B = j)$ . For example, consider the example discussed in Section 4.4.1.1 that had the same set of parameters with thresholds  $T_A = 8, T_B = 10$ . The best response of the distribution service is calculated to be  $p_A = 0.58$  and it can also be found in the routing matrix at the 8<sup>th</sup> row and 10<sup>th</sup> column (i.e.  $R_{8,10} = 0.58$ ).

#### 4.4.2.2 Implementation

The function defined in 4.12 shows how the routing matrix can be calculated by using function `calculate_class_2_individuals_best_response` for every possible pair of thresholds.

```
>>> import itertools
>>> import numpy as np
>>> def get_routing_matrix(
...     lambda_2,
...     lambda_1_1,
...     lambda_1_2,
...     mu_1,
...     mu_2,
...     num_of_servers_1,
...     num_of_servers_2,
...     system_capacity_1,
...     system_capacity_2,
...     buffer_capacity_1,
...     buffer_capacity_2,
...     alpha=0,
... ):
...     """
```

```

...     Get the optimal distribution matrix that consists of the proportion of
...     individuals to be distributed to each hospital for all possible
...     combinations of thresholds of the two hospitals (T_1, T_2). For every
...     set of thresholds, the function fills the entries of the matrix using
...     the proportion of individuals to distribute to hospital 1.
...
...     Parameters
...     -----
...     lambda_2 : float
...     lambda_1_1 : float
...     lambda_1_2 : float
...     mu_1 : float
...     mu_2 : float
...     num_of_servers_1 : int
...     num_of_servers_2 : int
...     system_capacity_1 : int
...     system_capacity_2 : int
...     buffer_capacity_1 : int
...     buffer_capacity_2 : int
...     routing_function : function, optional
...         The function to use to get the optimal distribution of patients
...     Returns
...     -----
...     numpy array
...         The matrix with proportions of all possible combinations of
...         threshold
...     """
...     routing_matrix = np.zeros((system_capacity_1, system_capacity_2))
...     for threshold_1, threshold_2 in itertools.product(
...         range(1, system_capacity_1 + 1), range(1, system_capacity_2 + 1)
...     ):
...         opt = calculate_class_2_individuals_best_response(
...             lambda_2=lambda_2,
...             lambda_1_1=lambda_1_1,
...             lambda_1_2=lambda_1_2,
...             mu_1=mu_1,
...             mu_2=mu_2,
...             num_of_servers_1=num_of_servers_1,
...             num_of_servers_2=num_of_servers_2,
...             system_capacity_1=system_capacity_1,
...             system_capacity_2=system_capacity_2,
...             buffer_capacity_1=buffer_capacity_1,
...             buffer_capacity_2=buffer_capacity_2,
...             threshold_1=threshold_1,
...             threshold_2=threshold_2,
...             alpha=alpha,
...         )
...         routing_matrix[threshold_1 - 1, threshold_2 - 1] = opt
...     return routing_matrix

```

Code snippet 4.12: Function that returns the routing matrix for a given set of game parameters.

Using the same set of parameters as in the example discussed in Section 4.4.2.1, the routing matrix can be calculated.

```

>>> get_routing_matrix(
...     lambda_2=lambda_2,
...     lambda_1_1=lambda_1_A,
...     lambda_1_2=lambda_1_B,
...     mu_1=mu_A,
...     mu_2=mu_B,
...     num_of_servers_1=num_of_servers_A,
...     num_of_servers_2=num_of_servers_B,
...     system_capacity_1=system_capacity_A,
...     system_capacity_2=system_capacity_B,
...     buffer_capacity_1=buffer_capacity_A,
...     buffer_capacity_2=buffer_capacity_B,
...     alpha=alpha,
... )
array([[0.58864276, 0.22335975, 0.15533059, 0.15265663, 0.15078926,
        0.14922469, 0.14719237, 0.14271416, 0.12874758, 0.06396441],
       [0.93574605, 0.67443129, 0.51051637, 0.48986689, 0.474517,
        0.46234589, 0.45136222, 0.43786699, 0.41004722, 0.31135186],
       [1., 0.84928478, 0.70944125, 0.66945048, 0.63960878,
        0.61601954, 0.59573161, 0.5746382, 0.54157719, 0.44625113],
       [1., 0.86430904, 0.73661752, 0.69683427, 0.66686909,
        0.64296716, 0.62233703, 0.6011736, 0.56942852, 0.48372486],
       [1., 0.8769602, 0.75856893, 0.71911699, 0.68913141,
        0.66505568, 0.64416384, 0.62297243, 0.59220666, 0.51352681],
       [1., 0.88782106, 0.77691989, 0.73780511, 0.7078792,
        0.68366777, 0.66270079, 0.64146588, 0.61149112, 0.5382421],
       [1., 0.89728669, 0.79261746, 0.75382586, 0.72401867,
        0.69978202, 0.6787237, 0.65751624, 0.6282157, 0.55939581],
       [1., 0.90564353, 0.80614796, 0.7677999, 0.73815484,
        0.71398819, 0.69284969, 0.67167208, 0.64301254, 0.5778495],
       [1., 0.91312427, 0.81813225, 0.78016576, 0.75073204,
        0.72662301, 0.70549771, 0.68440782, 0.65624748, 0.59425681],
       [1., 0.91997634, 0.82894698, 0.79141059, 0.76218245,
        0.73812947, 0.71705972, 0.69606716, 0.66837977, 0.60914055],
       [1., 0.92662384, 0.83917256, 0.80200443, 0.77297551,
        0.74903033, 0.7279936, 0.70708275, 0.67987751, 0.62310053],
       [1., 0.93417364, 0.85009707, 0.81320876, 0.7843084,
        0.7604156, 0.73937585, 0.7185154, 0.69171265, 0.63719748],
       [1., 0.94615952, 0.8654791, 0.82845358, 0.79936661,
        0.77525057, 0.75399993, 0.73296783, 0.70639403, 0.65395543],
       [1., 0.97522952, 0.89772397, 0.85877431, 0.82812693,
        0.80267354, 0.78024478, 0.75829045, 0.73124439, 0.68050787],
       [1., 1., 0.98966703, 0.94066708, 0.90265479,
        0.87136701, 0.84412235, 0.81814844, 0.78822172, 0.73774973]])

```

Code snippet 4.13: Using the function defined in the code snippet 4.12 to get the routing matrix

### 4.4.3 Queueing systems and normal form games

In subsection 4.4.1 it is shown that given the strategies played by queueing systems  $A$  and  $B$  the best response of the distribution service can be found. Consider the routing matrix  $R$  defined in equation (4.28).

$$R = \begin{pmatrix} p_{1,1}^A & p_{1,2}^A & \cdots & p_{1,N_B}^A \\ p_{2,1}^A & p_{2,2}^A & \cdots & p_{2,N_B}^A \\ \vdots & \vdots & \ddots & \vdots \\ p_{N_A,1}^A & p_{N_A,2}^A & \cdots & p_{N_A,N_B}^A \end{pmatrix}$$

Every entry  $R_{i,j}$  of the routing matrix represents best response of the distribution service when  $T^A = i$  and  $T^B = j$ . In other words, given  $T^A = i$  and  $T^B = j$ , the distribution service's strategy that maximises its utility should be to route a proportion of  $p_{i,j}^A$  of the individuals to queueing system  $A$  and a proportion of  $1 - p_{i,j}^A$  individuals to queueing system  $B$ . Assuming that for every pair of strategies  $T^A$  and  $T^B$  the best response of the distribution service is known to the queueing systems, then the formulation of the game can be simplified even more. In fact the imperfect information extensive form game defined in Section 4.2.5 can be now transformed into a 2-player normal form game between the two queueing systems. From equation (4.20) the utility of queueing system  $i$  when the pair of strategies  $(T^A, T^B)$  is played, is defined as:

$$U_{T^A, T^B}^i = 1 - \left( \hat{P} - P(W_i < t) \right)^2 \quad i \in A, B \quad (4.30)$$

$U_{T^A, T^B}^A$  and  $U_{T^A, T^B}^B$  are essentially artificial metrics that queueing systems  $A$  and  $B$  aim to maximise. Essentially, by maximising equation (4.30) for both queueing systems, the queueing systems are trying to minimise the difference between the proportion of individuals that are served within the target time  $t$  and the proportion target  $\hat{P}$ . For example, given a proportion target  $\hat{P} = 0.9$  which means that 90% of the individuals should be served within the target time  $t$ , and given that the proportion of individuals that are served within the target time is  $P(W_i < t) = 0.8$ , then the utility of the queueing system  $i$  is given by  $U_{T^A, T^B}^i = 1 - (0.9 - 0.8)^2 = 0.99$ . The payoff matrices of the game can be populated by these utilities for all possible pairs of strategies  $(T^A, T^B)$ .

$$A = \begin{pmatrix} U_{1,1}^A & U_{1,2}^A & \cdots & U_{1,N_B}^A \\ U_{2,1}^A & U_{2,2}^A & \cdots & U_{2,N_B}^A \\ \vdots & \vdots & \ddots & \vdots \\ U_{N_A,1}^A & U_{N_A,2}^A & \cdots & U_{N_A,N_B}^A \end{pmatrix}, B = \begin{pmatrix} U_{1,1}^B & U_{1,2}^B & \cdots & U_{1,N_B}^B \\ U_{2,1}^B & U_{2,2}^B & \cdots & U_{2,N_B}^B \\ \vdots & \vdots & \ddots & \vdots \\ U_{N_A,1}^B & U_{N_A,2}^B & \cdots & U_{N_A,N_B}^B \end{pmatrix} \quad (4.31)$$

Matrix  $A$  consists of all possible utilities of queueing system  $A$ , and matrix  $B$

consists of all possible utilities of queueing system  $B$ . The game is now a 2-player normal form game with payoff matrices  $A$  and  $B$ .

#### 4.4.3.1 Implementation

Consider an example of the described game with the following parameters:

Table 4.5: Parameter values for game formulation example

Distributor				Queueing system $A$					Queueing system $B$				
$\lambda_2$	$t$	$\hat{P}$	$\alpha$	$\lambda_1^A$	$\mu^A$	$C^A$	$N^A$	$M^A$	$\lambda_1^B$	$\mu^B$	$C^B$	$N^B$	$M^B$
4	1	0.95	0.2	2	3	2	7	6	1	1	3	4	3

```

>>> lambda_2 = 4
>>> target = 1
>>> p_hat = 0.95
>>> alpha = 0.2
>>>
>>> lambda_1_A = 2
>>> mu_A = 3
>>> num_of_servers_A = 2
>>> system_capacity_A = 7
>>> buffer_capacity_A = 6
>>>
>>> lambda_1_B = 1
>>> mu_B = 1
>>> num_of_servers_B = 3
>>> system_capacity_B = 4
>>> buffer_capacity_B = 3

```

Code snippet 4.14: Variables that correspond to the parameter set from Table 4.5

The first function to create is one that takes a pair of thresholds  $T^A = i, T^B = j$  and gets the best response of the distribution service  $p_A$ . Then, using  $p_A$ , the function finds the value of  $U_{i,j}^A$  and  $U_{i,j}^B$  and returns the tuple  $(i, j, p_A, U_{i,j}^A, U_{i,j}^B)$ .

```

>>> import numpy as np
>>> def get_individual_entries_of_matrices(
...     lambda_2,
...     lambda_1_1,
...     lambda_1_2,
...     mu_1,
...     mu_2,
...     num_of_servers_1,
...     num_of_servers_2,
...     threshold_1,
...     threshold_2,
...     system_capacity_1,
...     system_capacity_2,
...     buffer_capacity_1,
...     buffer_capacity_2,
...     alpha,
...     target,

```

```

...     p_hat,
... ):
...     """
...     Gets the (i,j)th entry of the payoff matrices and the routing matrix
...     where i=threshold_1 and j=threshold_2.
...
...     Parameters
...     -----
...     lambda_2 : float
...     lambda_1_1 : float
...     lambda_1_2 : float
...     mu_1 : float
...     mu_2 : float
...     num_of_servers_1 : int
...     num_of_servers_2 : int
...     threshold_1 : int
...     threshold_2 : int
...     system_capacity_1 : int
...     system_capacity_2 : int
...     buffer_capacity_1 : int
...     buffer_capacity_2 : int
...     alpha : float
...     target : float
...
...     Returns
...     -----
...     tuple
...     A tuple of the form (i, j, R[i,j], A[i,j], B[i,j])
...     """
...     prop_to_hospital_1 = calculate_class_2_individuals_best_response(
...         lambda_2=lambda_2,
...         lambda_1_1=lambda_1_1,
...         lambda_1_2=lambda_1_2,
...         mu_1=mu_1,
...         mu_2=mu_2,
...         num_of_servers_1=num_of_servers_1,
...         num_of_servers_2=num_of_servers_2,
...         system_capacity_1=system_capacity_1,
...         system_capacity_2=system_capacity_2,
...         buffer_capacity_1=buffer_capacity_1,
...         buffer_capacity_2=buffer_capacity_2,
...         threshold_1=threshold_1,
...         threshold_2=threshold_2,
...         alpha=alpha,
...     )
...     prop_to_hospital_2 = 1 - prop_to_hospital_1
...
...     proportion_within_target_1 = (
...         abg.markov.proportion_within_target_using_markov_state_probabilities
...     (
...         lambda_2=lambda_2 * prop_to_hospital_1,
...         lambda_1=lambda_1_1,
...         mu=mu_1,
...         num_of_servers=num_of_servers_1,
...         threshold=threshold_1,
...         system_capacity=system_capacity_1,
...         buffer_capacity=buffer_capacity_1,
...         class_type=None,
...         target=target,

```

```

...     )
... )
... proportion_within_target_2 = (
...     abg.markov.proportion_within_target_using_markov_state_probabilities
... (
...     lambda_2=lambda_2 * prop_to_hospital_2,
...     lambda_1=lambda_1_2,
...     mu=mu_2,
...     num_of_servers=num_of_servers_2,
...     threshold=threshold_2,
...     system_capacity=system_capacity_2,
...     buffer_capacity=buffer_capacity_2,
...     class_type=None,
...     target=target,
... )
... )
... utility_1 = 1 - (
...     (np.nanmean(proportion_within_target_1) - p_hat) ** 2
... )
... utility_2 = 1 - (
...     (np.nanmean(proportion_within_target_2) - p_hat) ** 2
... )
...
... return (
...     threshold_1,
...     threshold_2,
...     prop_to_hospital_1,
...     utility_1,
...     utility_2
... )

```

Code snippet 4.15: Function that takes as inputs the given strategies (thresholds) of the players and calculates the corresponding utilities of the players.

The function `get_individual_entries_of_matrices` can now be used to get the  $(i, j)^{\text{th}}$  entry of the routing matrix  $R$  payoff matrix  $A$  and payoff matrix  $B$ . For example consider the case when  $T_A = 7$  and  $T_B = 4$ . The equivalent values of  $R, A$  and  $B$  on the  $(7, 4)^{\text{th}}$  position can be calculated using the code shown in 4.16.

```

>>> _, _, p_A, U_A, U_B = np.round(get_individual_entries_of_matrices(
...     lambda_2=lambda_2,
...     lambda_1_1=lambda_1_A,
...     lambda_1_2=lambda_1_B,
...     mu_1=mu_A,
...     mu_2=mu_B,
...     num_of_servers_1=num_of_servers_A,
...     num_of_servers_2=num_of_servers_B,
...     threshold_1=7,
...     threshold_2=4,
...     system_capacity_1=system_capacity_A,
...     system_capacity_2=system_capacity_B,
...     buffer_capacity_1=buffer_capacity_A,
...     buffer_capacity_2=buffer_capacity_B,
...     alpha=alpha,
...     target=target,

```



```

...     p_hat=p_hat
... ), 8)
>>> p_A
0.80583153
>>> U_A
0.96146225
>>> U_B
0.87505776

```

Code snippet 4.16: Example of using the function defined in 4.15

The second step is to use the `get_individual_entries_of_matrices` function to calculate the entries of the routing matrix  $R$ , payoff matrix  $A$  and payoff matrix  $B$ . Thus, by iterating over all possible values of  $T_A$  and  $T_B$ , the routing matrix  $R$ , payoff matrix  $A$  and payoff matrix  $B$  can be calculated. Note that in Section 4.4.2.2 the `get_routing_matrix` function was defined that returns the routing matrix  $R$ . The function defined in 4.17 gets all matrices in a much more computationally efficient way.

```

>>> import itertools
>>> def get_payoff_matrices(
...     lambda_2,
...     lambda_1_1,
...     lambda_1_2,
...     mu_1,
...     mu_2,
...     num_of_servers_1,
...     num_of_servers_2,
...     system_capacity_1,
...     system_capacity_2,
...     buffer_capacity_1,
...     buffer_capacity_2,
...     target,
...     alpha,
...     p_hat,
...     alternative_utility=False,
... ):
...     """
...     The function uses the distribution array (that is the array that holds
...     the optimal proportion of individuals to send to each hospital), to
...     calculate the proportion of patients within time for every possible set
...     of thresholds chosen by each system.
...     Parameters
...     -----
...     lambda_2 : float
...     lambda_1_1 : float
...     lambda_1_2 : float
...     mu_1 : float
...     mu_2 : float
...     num_of_servers_1 : int
...     num_of_servers_2 : int
...     system_capacity_1 : int
...     system_capacity_2 : int
...     buffer_capacity_1 : int
...     buffer_capacity_2 : int
...     target : float

```

```

...         The target time that individuals should be within
...
...     Returns
...     -----
...     numpy.array, numpy.array
...     The payoff matrices of the game
...     """
...     utility_matrix_1 = np.zeros((system_capacity_1, system_capacity_2))
...     utility_matrix_2 = np.zeros((system_capacity_1, system_capacity_2))
...     routing_matrix = np.zeros((system_capacity_1, system_capacity_2))
...     for threshold_1, threshold_2 in itertools.product(
...         range(1, system_capacity_1 + 1), range(1, system_capacity_2 + 1)
...     ):
...         T_A, T_B, p_A, U_A, U_B = get_individual_entries_of_matrices(
...             lambda_2=lambda_2,
...             lambda_1_1=lambda_1_1,
...             lambda_1_2=lambda_1_2,
...             mu_1=mu_1,
...             mu_2=mu_2,
...             num_of_servers_1=num_of_servers_1,
...             num_of_servers_2=num_of_servers_2,
...             threshold_1=threshold_1,
...             threshold_2=threshold_2,
...             system_capacity_1=system_capacity_1,
...             system_capacity_2=system_capacity_2,
...             buffer_capacity_1=buffer_capacity_1,
...             buffer_capacity_2=buffer_capacity_2,
...             alpha=alpha,
...             target=target,
...             p_hat=p_hat,
...         )
...         utility_matrix_1[T_A - 1, T_B - 1] = U_A
...         utility_matrix_2[T_A - 1, T_B - 1] = U_B
...         routing_matrix[T_A - 1, T_B - 1] = p_A
...
...     return utility_matrix_1, utility_matrix_2, routing_matrix

```

Code snippet 4.17: Function that returns the payoff matrices and the routing matrix

The code shown in 4.18 returns matrices  $A$ ,  $B$  and  $R$  for the parameters of the example given above.

```

>>> A, B, R = get_payoff_matrices(
...     lambda_2=lambda_2,
...     lambda_1_1=lambda_1_A,
...     lambda_1_2=lambda_1_B,
...     mu_1=mu_A,
...     mu_2=mu_B,
...     num_of_servers_1=num_of_servers_A,
...     num_of_servers_2=num_of_servers_B,
...     system_capacity_1=system_capacity_A,
...     system_capacity_2=system_capacity_B,
...     buffer_capacity_1=buffer_capacity_A,
...     buffer_capacity_2=buffer_capacity_B,
...     target=target,
...     alpha=alpha,
...     p_hat=p_hat,

```

```

... )
>>> A
array([[0.9997736 , 0.9997736 , 0.9997736 , 0.9997736 ],
       [0.99918651, 0.99923556, 0.99934362, 0.99945211],
       [0.99406363, 0.99485147, 0.99635215, 0.99753246],
       [0.9797136 , 0.9828782 , 0.98884481, 0.99308839],
       [0.95263545, 0.9599928 , 0.97510779, 0.98528894],
       [0.92089495, 0.92932621, 0.95614727, 0.97424141],
       [0.92414307, 0.92414307, 0.94021899, 0.96146225]])
>>> B
array([[0.89253416, 0.88888203, 0.87438695, 0.79835823],
       [0.89253416, 0.89054246, 0.8796009 , 0.82819449],
       [0.89253416, 0.89081835, 0.8809942 , 0.837463  ],
       [0.89253416, 0.89107223, 0.88222127, 0.84490465],
       [0.89253416, 0.89136098, 0.88349414, 0.85178752],
       [0.89253416, 0.89186314, 0.8853596 , 0.86018873],
       [0.89253416, 0.89253416, 0.88964749, 0.87505776]])
>>> R
array([[0.92344478, 0.4632574 , 0.25667497, 0.14225553],
       [1.          , 0.85220313, 0.62624741, 0.45231728],
       [1.          , 0.87813784, 0.67951058, 0.52328851],
       [1.          , 0.89972766, 0.72107423, 0.57760157],
       [1.          , 0.92227658, 0.76050588, 0.62710433],
       [1.          , 0.95776622, 0.81369017, 0.68824209],
       [1.          , 1.          , 0.92566552, 0.80583153]])

```

Code snippet 4.18: Example of code that gets the values of the payoff matrices and the routing matrix for example 1

The final task is to use the `nashpy` library to build the game using the payoff matrices.

```

>>> import nashpy as nash
>>> game = nash.Game(A, B)
>>> game
Bi matrix game with payoff matrices:
<BLANKLINE>
Row player:
[[0.9997736  0.9997736  0.9997736  0.9997736 ]
 [0.99918651 0.99923556 0.99934362 0.99945211]
 [0.99406363 0.99485147 0.99635215 0.99753246]
 [0.9797136  0.9828782  0.98884481 0.99308839]
 [0.95263545 0.9599928  0.97510779 0.98528894]
 [0.92089495 0.92932621 0.95614727 0.97424141]
 [0.92414307 0.92414307 0.94021899 0.96146225]]
<BLANKLINE>
Column player:
[[0.89253416 0.88888203 0.87438695 0.79835823]
 [0.89253416 0.89054246 0.8796009  0.82819449]
 [0.89253416 0.89081835 0.8809942  0.837463  ]
 [0.89253416 0.89107223 0.88222127 0.84490465]
 [0.89253416 0.89136098 0.88349414 0.85178752]
 [0.89253416 0.89186314 0.8853596  0.86018873]
 [0.89253416 0.89253416 0.88964749 0.87505776]]

```

Code snippet 4.19: Build the `nashpy` object that consists of the game

#### 4.4.4 Solving the game

This section describes how everything introduced in this chapter so far can be put together to obtain numerical results of the 3-player game between two queueing systems and the individual distribution service. Having formulated the game (Section 4.3) and obtained the payoff matrices for the two players (Section 4.4.3), the strategies of the players can now be investigated. For a 2-player game once the two payoff matrices have been calculated, the game can be solved using the methods described in Section 4.2. More specifically, the concept of Nash equilibrium described in Section 4.2.2 and the concept of Evolutionary Stable Strategies described in Section 4.2.3 will be used to analyse the behaviour of the players in the game.

Consider a game between queueing systems  $A$  and  $B$  and a distribution service  $D$  with the following parameters.

Table 4.6: Parameter values for example on solving the game

Distributor				Queueing system $A$					Queueing system $B$				
$\lambda_2$	$t$	$\hat{P}$	$\alpha$	$\lambda_1^A$	$\mu^A$	$C^A$	$N^A$	$M^A$	$\lambda_1^B$	$\mu^B$	$C^B$	$N^B$	$M^B$
1	1	0.8	0.5	1	2	1	2	2	1	3	1	2	2

```

>>> lambda_2 = 1
>>> target = 1
>>> p_hat = 0.8
>>> alpha = 0.5

>>> lambda_1_A = 1
>>> mu_A = 2
>>> num_of_servers_A = 1
>>> system_capacity_A = 2
>>> buffer_capacity_A = 2

>>> lambda_1_B = 1
>>> mu_B = 3
>>> num_of_servers_B = 1
>>> system_capacity_B = 2
>>> buffer_capacity_B = 2

```

Code snippet 4.20: Variables that correspond to the parameter set from Table 4.6

Note that these are two small queueing systems with a maximum capacity of 2. In other words, since both systems have a capacity of 2, their possible strategies are to either choose a threshold of  $T_i = 1$  or  $T_i = 2$  where  $i \in \{A, B\}$ . In other words the queueing systems could start blocking type 2 individuals either when there is 1 individual in node 1 or when there are 2 individuals. See Figure 4.7 for a visual representation of the game. In this example, the only difference between

the two queueing systems is that queueing system  $A$  has a lower service rate than queueing system  $B$ .

As shown in Section 4.4.3 the payoff matrices for the two players can be obtained using the function `get_payoff_matrices`. Every function described in this section is also implemented in the `ambulance_game` python library that was created for this project. A detailed description of the library can be found in Appendix A. The code snippet shown in 4.21 shows how the payoff matrices can be obtained using the `ambulance_game` library.

```
>>> import ambulance_game as abg
>>> A, B, R = abg.game.get_payoff_matrices(
...     lambda_2=lambda_2,
...     lambda_1_1=lambda_1_A,
...     lambda_1_2=lambda_1_B,
...     mu_1=mu_A,
...     mu_2=mu_B,
...     num_of_servers_1=num_of_servers_A,
...     num_of_servers_2=num_of_servers_B,
...     system_capacity_1=system_capacity_A,
...     system_capacity_2=system_capacity_B,
...     buffer_capacity_1=buffer_capacity_A,
...     buffer_capacity_2=buffer_capacity_B,
...     target=target,
...     alpha=alpha,
...     p_hat=p_hat,
... )
>>> game = nash.Game(A, B)
>>> game
Bi matrix game with payoff matrices:
<BLANKLINE>
Row player:
[[0.99934675 0.99934675]
 [0.99282972 0.99828249]]
<BLANKLINE>
Column player:
[[0.98725977 0.99408002]
 [0.98725977 0.99312791]]
```

Code snippet 4.21: Using the `ambulance_game` library to build the game

Having the payoff matrices, the Nash Equilibrium can be obtained using the support enumeration algorithm or the Lemke-Howson algorithm implemented in the `nashpy` library. See Section 4.2.2 for more details on these algorithms.

```
>>> tuple(game.support_enumeration())
((array([1., 0.]), array([0., 1.])),)
>>> tuple(game.lemke_howson(initial_dropped_label=0))
(array([1., 0.]), array([0., 1.]))
```

Code snippet 4.22: Support enumeration and Lemke-Howson computation on the payoff matrices of the two players

Both algorithms return the same set of equilibria, which is for queueing system  $A$  to always choose a threshold of  $T_A = 1$  and for queueing system  $B$  to always choose a threshold of  $T_B = 2$ . This means, that queueing system  $A$  will block type 2 individuals when there is 1 individual in node 1 and queueing system  $B$  will block type 2 individuals only when node 1 is at maximum capacity.

As the sizes of the queueing systems considered grow the strategy spaces will as well. This can increase the complexity of finding equilibria. In addition, equilibria might not necessarily emerge, thus the learning algorithms described in Section 4.2.3 can be used to obtain Evolutionary Stable Strategies (ESS) for the game. The following code runs the fictitious play algorithm for 100 iterations and returns the number of times each strategy was played.

```
>>> np.random.seed(5)
>>> play_counts = tuple(game.fictitious_play(iterations=100))
>>> play_counts[-1]
[array([99.,  1.]), array([ 1., 99.])]
```

Code snippet 4.23: Fictitious play algorithm on the payoff matrices of the two players

It can be seen that the Nash equilibrium set of strategies found by the support enumeration algorithm and the Lemke-Howson algorithm is also reached here. Using learning algorithms, not only can one find a Nash equilibrium, but also visualise how the players of the game reach it. Figure 4.28 shows how the players evolve their strategies over time.

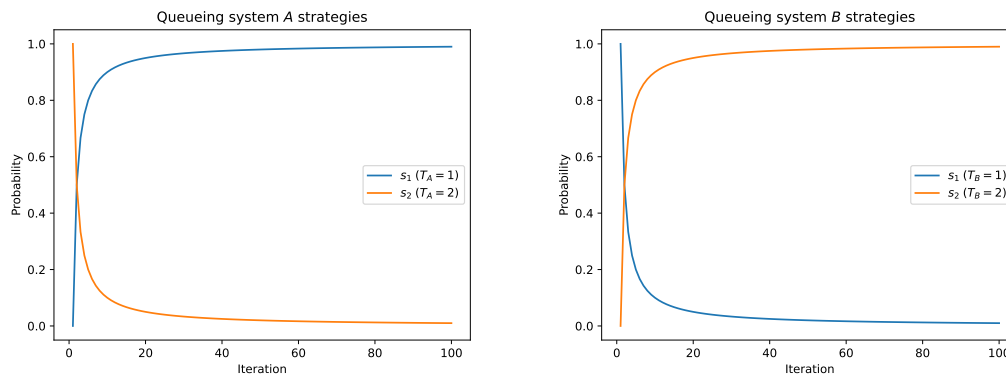


Figure 4.28: Fictitious play algorithm run on the strategies of the players.

Similarly, the stochastic fictitious play algorithm is ran for 1000 iterations. The code shown in 4.24 shows how the number of times each strategy was played and the distribution of strategies over time.

```

>>> np.random.seed(0)
>>> play_counts_and_distributions = tuple(
...     game.stochastic_fictitious_play(iterations=1000)
... )
>>> plays, dist = play_counts_and_distributions[-1]
>>> plays
[array([509., 491.]), array([512., 488.])]
>>> dist
[array([0.51785058, 0.48214942]), array([0.47022964, 0.52977036])]

```

Code snippet 4.24: Stochastic fictitious play algorithm on the payoff matrices of the two players

The results are not similar to the case of the fictitious play algorithm. In fact the stochastic fictitious play algorithm does not converge to a Nash equilibrium. Instead, the algorithm converges to a mixed strategy equilibrium where both players have a probability of playing each strategy. This might be due to the choice of parameters of the algorithm  $\eta$  and  $\bar{\epsilon}$  or the fact that the values of the payoff matrices are smaller than the algorithm can handle.

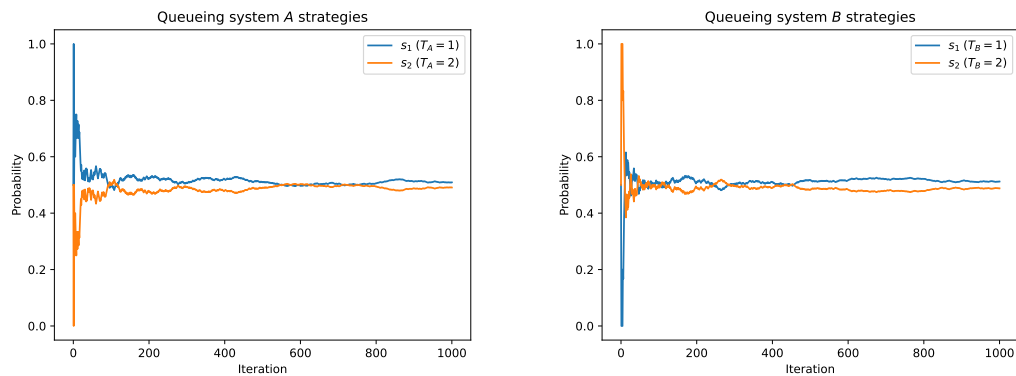


Figure 4.29: Stochastic fictitious play algorithm on the strategies of the players.

Finally, the last algorithm to be ran is the asymmetric replicator dynamics algorithm. This algorithm is ran for 10,000 timepoints and the final strategies of both players are calculated.

```

>>> xs, ys = game.asymmetric_replicator_dynamics(
...     timepoints=np.linspace(0, 10000, 100)
... )
>>> np.round(xs[-1], 4)
array([1., 0.])
>>> np.round(ys[-1], 4)
array([0., 1.])

```

Code snippet 4.25: Asymmetric replicator dynamics algorithm on the payoff matrices of the two players

The resulted set of strategies are a Nash equilibrium. Similar to the fictitious play algorithm, the asymmetric replicator dynamics algorithm can be used to visualise how the players of the game reach a Nash equilibrium. Figure 4.30 shows how the players evolve their strategies over time.

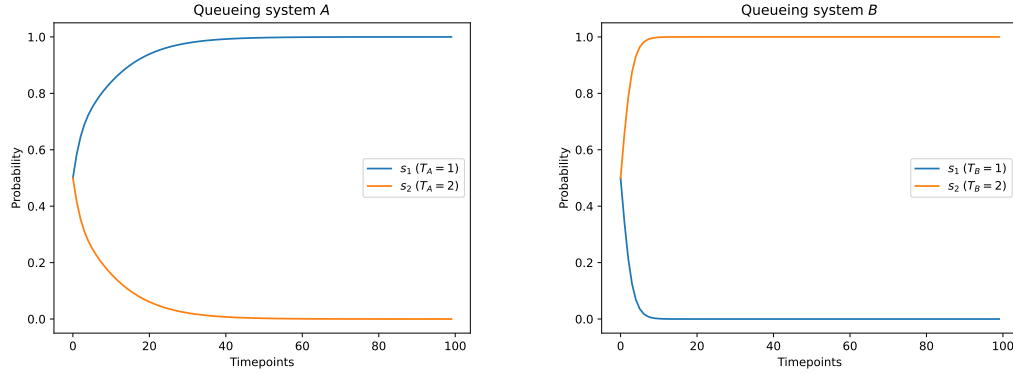


Figure 4.30: Asymmetric replicator dynamics on the strategies of the players.

The asymmetric replicator dynamics algorithm is the learning algorithm that will be used in the following sections to study the effect of the parameters of the game on the strategies of the players.

## 4.5 ED-EMS application

Similar to Section 3.6, the game theoretic model can be applied to the same healthcare setting. All concepts described in this Section can be mapped to some components of either the ED or the EMS.

The EMS has to decide how to distribute its patients among the two EDs so that the weighted combination of the ambulance blocking time and the percentage of lost ambulances is minimised. This can be illustrated by Figure 4.7. The interaction between the two EDs is a normal form game that is then used to inform the decision of the EMS. Note that the formulated game here assumes that prior to making a choice the EMS knows the strategies that each ED is playing (Figure 4.15). This corresponds to reacting to experienced delays.

The queueing systems of the hospitals are designed in such a way where they can accept two types of individuals (Chapter 3). Each hospital may then choose to block type 2 individuals when the hospital reaches a certain capacity. The strategy sets for each hospital is the set  $\{T \in \mathbb{N} \mid 1 \leq T \leq N\}$  where  $N \in \{N_A, N_B\}$  are the total capacities of hospitals A and B. The chosen actions from the strategy set are denoted as  $T_A, T_B$  and are the *thresholds*.



Both hospitals follow a queueing model with two waiting spaces for individuals. The first waiting space (i.e. the waiting space of the hospital) is where the patients queue right before receiving their service and has a queue capacity of  $N - C$ , where  $N$  is the total capacity of the hospital and  $C$  is the number of healthcare professionals able to see them. The second waiting space (i.e. the parking space for ambulances) is where ambulances, that are sent from the EMS distributor, stay until their patients are allowed to enter the hospital. The parking space has a capacity of  $M$  and no servers. This is shown diagrammatically in Figure 3.1.

Note here that both types of individuals can become lost to the system. An individual allocated from the ambulance service becomes lost to the system whenever an arrival occurs and the parking space is at full capacity ( $M$  ambulances already parked). Similarly, type 1 individuals get lost whenever they arrive at the waiting space of the hospital and it is at full capacity ( $N - C$  individuals already waiting). Numerical results on the ED-EMS game theoretic model are presented and discussed in Chapter 5.

There are certain assumptions that are made in this application. Firstly, it is assumed that the distance from any patient's location to any hospital is not a factor that will affect the EMS's decision. That means that under the scope of this application, the EMS does not have to consider the closest hospital to the patient's location. Secondly, it is assumed that a patient's timer (from the perspective of the ED) does not start counting until the patient enters the hospital. For instance, consider the case where a patient is sent from the EMS to hospital  $A$  and is blocked in the parking space of hospital  $A$  for 6 hours. The patient then proceeds to wait in the hospital for an additional 2 hours and is then receiving their treatment for 1 hour. The patient's total time in the hospital is assumed to be 3 hours ( $2 + 1$ ). Finally, the last assumption that is made is that arrival and service times are exponentially distributed.

## 4.6 Chapter summary

This chapter introduces a 3-player game theoretic model between the decision makers of two queueing systems and a distribution service that distributes individuals to them. The game theoretic model is used to investigate the behaviour of the players when they interact and try to maximise their own utilities.

Section 4.2 gives a brief introduction to the game theoretic concepts that are used in this chapter. A brief introduction on Normal Form games and the concept of Nash equilibrium are given, along with some examples of games and their equiv-

alent Nash equilibria. Additionally, some learning algorithms are introduced. The learning algorithm that is mostly used in this chapter is the asymmetric replicator dynamics algorithm. Finally, a description of perfect-information and imperfect-information normal form games is given.

Section 4.3 describes the formulation of the game theoretic model that is used in this thesis. An overview of the three players and the parameters that are used in the game is given. The parameter that is of most interest here is the threshold parameter that is used to determine when individuals are being blocked from entering node 1 of the queueing system. The set of strategies of the two players is essentially all possible values that the threshold parameter can take. Furthermore, the payoffs for the two players are described. The distribution service's payoff is determined by the number of individuals that are blocked in node 2 of the queueing system. The queueing system's payoff is determined by the proportion of individuals whose waiting time is less than a predefined target time. Thus, the imperfect-information extensive form game is introduced.

Section 4.4 describes the methodology that is used to solve the game theoretic model. The methodology uses Brent's algorithm to find the optimal split of individuals that the distribution service can distribute to the two queueing systems to minimise its own blocking time. The routing matrix is then described that contains all possible values of the proportion of individuals that can be distributed to the two queueing systems. The game theoretic model is then reduced to a 2-player normal form game where the utilities of the game are decided the distribution service. Following this, a game with some example set of parameters is given and is solved using the support enumeration algorithm and the Lemke-Howson algorithm. Additionally, some learning algorithms are applied to the game to observe the behaviour of the players when they interact together. The learning algorithms that are used are fictitious play, stochastic fictitious play, and asymmetric replicator dynamics.

Finally, Section 4.5 maps the game theoretic model to a healthcare setting to observe the behavioural gaming that takes place at the EMS - ED interface. The three players now become the Emergency Medical Services (EMS) and two Emergency Departments (EDs). The game essentially consists of the EMS deciding what proportion of ambulances to send to each hospital so that the blocking time is minimised and the EDs choosing what threshold parameter to use so that a certain policy objective is met. In line with the current NHS policy [145], the objective is set to be that 95% of the patients that are admitted to the EDs should be admitted within 4 hours.

# Chapter 5

## Numerical Results

### 5.1 Introduction

This chapter presents the results of the numerical experiments conducted to study the behavioural patterns that can emerge from the interaction between the two EDs and the EMS. Additional numerical experiments can also be found in appendix B. This chapter consists of the following sections:

- Section 5.2 describes the data collection process.
- Section 5.3 gives an overview and some descriptive statistics of the data.
- Section 5.4 presents the results of the numerical experiments and discusses the implications of the results.

Note that the data presented in this chapter are archived and can be found in [112]. This chapter extends the results presented in [114].

### 5.2 Data Collection

The data presented in this chapter were collected by solving the game theoretic model from the interaction of the two EDs and the EMS. The collected data are the matrices  $A$ ,  $B$  and  $R$  described in Section 4.4.3. Each triplet of matrices  $(A, B, R)$  was generated by solving the corresponding game for a different set of parameters.

Different values of the parameters were used to generate this dataset. The parameters that were changed throughout these experiments are listed in Table 5.1.

Table 5.1: Data collection: Parameters

Parameter	Description
$\lambda_2$	Arrival rate of patients in EMS
$\alpha$	Weight of lost individuals over time blocked
$t$	Time target for the EDs
$\lambda_1^A$	Arrival rate of patients in ED $A$
$\lambda_1^B$	Arrival rate of patients in ED $B$
$\mu^A$	Service rate of patients in ED $A$
$\mu^B$	Service rate of patients in ED $B$
$C^A$	Number of servers in ED $A$
$C^B$	Number of servers in ED $B$
$N^A$	Maximum capacity in ED $A$
$N^B$	Maximum capacity in ED $B$
$M^A$	Maximum capacity of parking space in ED $A$
$M^B$	Maximum capacity of parking space in ED $B$

Overall, 5,160,404 different sets of parameters were used. For each parameter set, the script solves the game and stores the resulting entries of the matrices  $(A, B, R)$  as well as the parameters used to generate them in a sub-directory of the `data` directory. All parameter values of each parameter set, along with the corresponding values of  $A, B$  and  $R$ , are mapped to a unique fixed-size hash value which is used as the name of the sub-directory for that parameter set. The hash function that is used for this operation is the MD5 message-digest algorithm [121].

Having the name of the sub-directory for the current parameter set, the resulting matrices are stored in a compressed `.npz` file in that sub-directory. The `.npz` file format is a zipped archive of files named after the variables they contain, which are stored using numpy's `savez_compressed` function [65]. The `.npz` files can be loaded using numpy's `load` function. In addition, to the `.npz` file, a `README.md` file containing general instructions and a `.csv` file containing the value of each parameter is also stored in each sub-directory. The code snippet in 5.1 shows an example of how the data are stored and loaded using numpy.

```

>>> import numpy as np
>>> array_1, array_2 = np.arange(10), np.arange(10, 20)
>>> np.savez_compressed(
...     "demo", array_1=array_1, array_2=array_2
... )
>>> loaded_file = np.load("demo.npz")
>>> loaded_file["array_1"]
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> loaded_file["array_2"]
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])

```

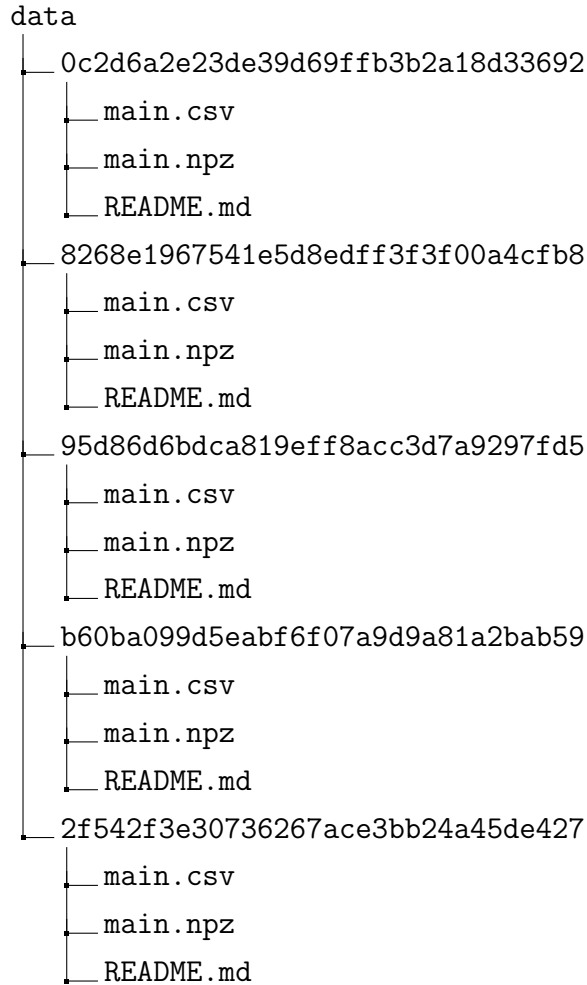
Code snippet 5.1: Example of saving and loading compressed data

Apart from every sub-directory containing the data for a specific parameter set, there is also a sub-directory named `_parameters`. This sub-directory contains a `.csv` file containing the values of all parameters used in the experiments along with their corresponding hash values. This file can be used to map the hash values of the sub-directories to the corresponding parameter values. Consider the first 5 entries of the `.csv` file from the `_parameters` sub-directory and the first 5 directories in the `data` directory.

Table 5.2: Contents of `_parameters/main.csv` file

0	5	4	0	0	0.1	2	3	3	2	6	7	0	0c2d6a2e23de39d69ffb3b2a18d33692
0	5	4	0	0	0.1	2	3	3	2	6	7	1	8268e1967541e5d8edff3f3f00a4cfb8
0	5	4	0	0	0.1	2	3	3	2	6	7	2	95d86d6bdca819eff8acc3d7a9297fd5
0	5	4	0	0	0.1	2	3	3	2	6	7	3	b60ba099d5eabf6f07a9d9a81a2bab59
0	5	4	0	0	0.1	2	3	3	2	6	7	4	2f542f3e30736267ace3bb24a45de427

The corresponding sub-directories of the `data` directory are shown in the tree structure in Figure 5.1:

Figure 5.1: Structure of the `data` directory

The `main.csv` file in each sub-directory contains the values of the parameters used to generate the data in that sub-directory. The `main.npz` file contains the compressed entries of the matrices  $A$ ,  $B$  and  $R$ . The complete dataset along with the scripts used to generate it have been archived using [www.zenodo.org](http://www.zenodo.org) and can be found with the following DOI: 10.5281/zenodo.7501988 [112].

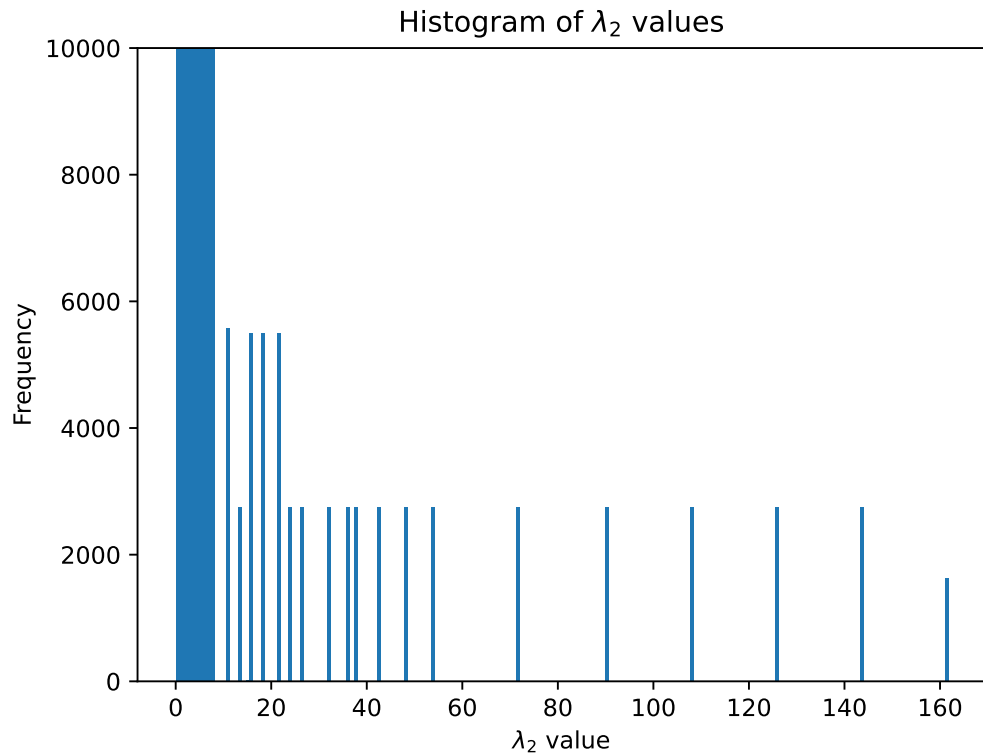
### 5.3 Dataset description

As mentioned in Section 5.2, the dataset contains matrices  $A$ ,  $B$  and  $R$  for a total of 5,160,404 parameter sets. A descriptive analysis of the values of the parameters used in the experiments is shown in Table 5.3.

Table 5.3: Descriptive statistics of the dataset

Parameter	Mean	Standard Deviation	Minimum	Maximum
$\lambda_2$	4.545320	7.045888	0.100000	161.821844
$\alpha$	0.504179	0.305070	0.000000	1.000000
$t$	4.999915	3.036935	0.000000	10.000000
$\lambda_1^B$	1.217896	2.050153	0.000000	34.019111
$\mu^A$	2.024239	0.400677	0.420571	6.773554
$C^A$	1.081755	0.680075	1.000000	9.000000
$N^A$	2.107661	0.821679	2.000000	24.000000
$M^A$	2.045482	0.456713	1.000000	20.000000
$\lambda_1^A$	1.156440	2.628927	0.000000	60.961985
$\mu^B$	2.052536	0.421094	2.000000	6.602015
$C^B$	1.102124	0.784007	1.000000	9.000000
$N^B$	2.121707	0.804273	2.000000	28.000000
$M^B$	2.071752	0.560980	2.000000	16.000000

Not all values listed in Table 5.3 are used equally often. In fact consider the values of the parameter  $\lambda_2$  that range from 0.1 to 162. Figure 5.2 shows the number of times each value of  $\lambda_2$  is used in the dataset.

Figure 5.2: Number of times each value of  $\lambda_2$  is used in the dataset

It can be seen that the values of  $\lambda_2$  that are used the most are the ones from 0.1 to 10. In fact, the y-axis of Figure 5.2 has been cut at 10,000 to better show the values of  $\lambda_2$  that are greater than 10. Figure 5.3 shows the zoomed-in version of Figure 5.2 where only values of  $\lambda_2$  from 0 to 10 are shown. From Figures 5.2 and 5.3 it can be seen that the values of  $\lambda_2$  that are used the most are the ones from 0 to 10.

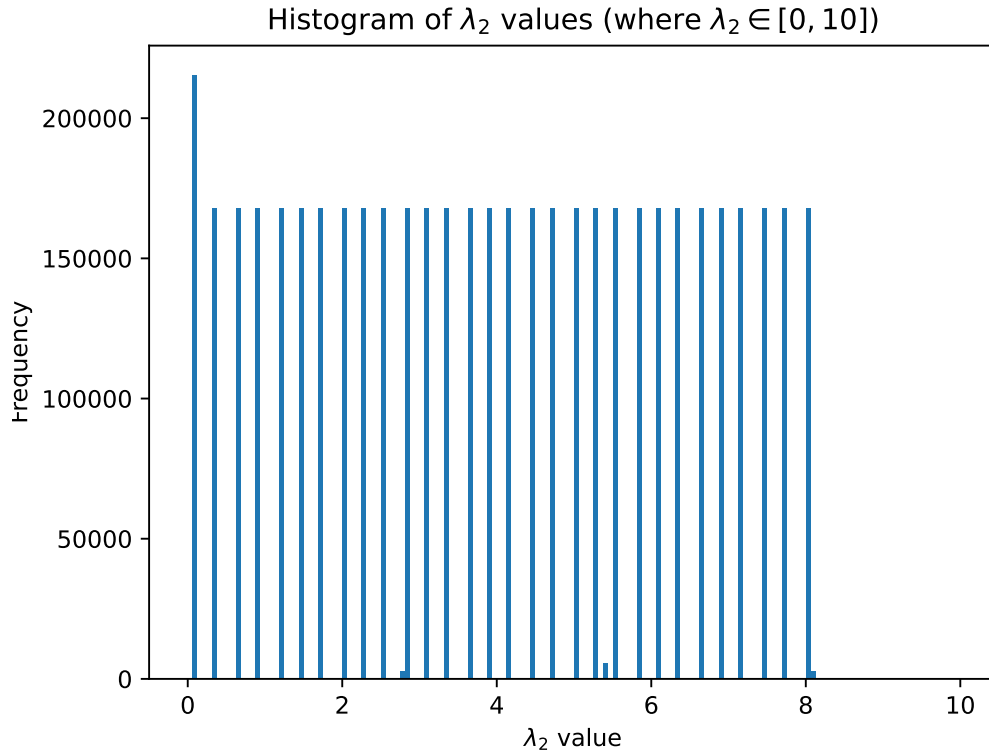
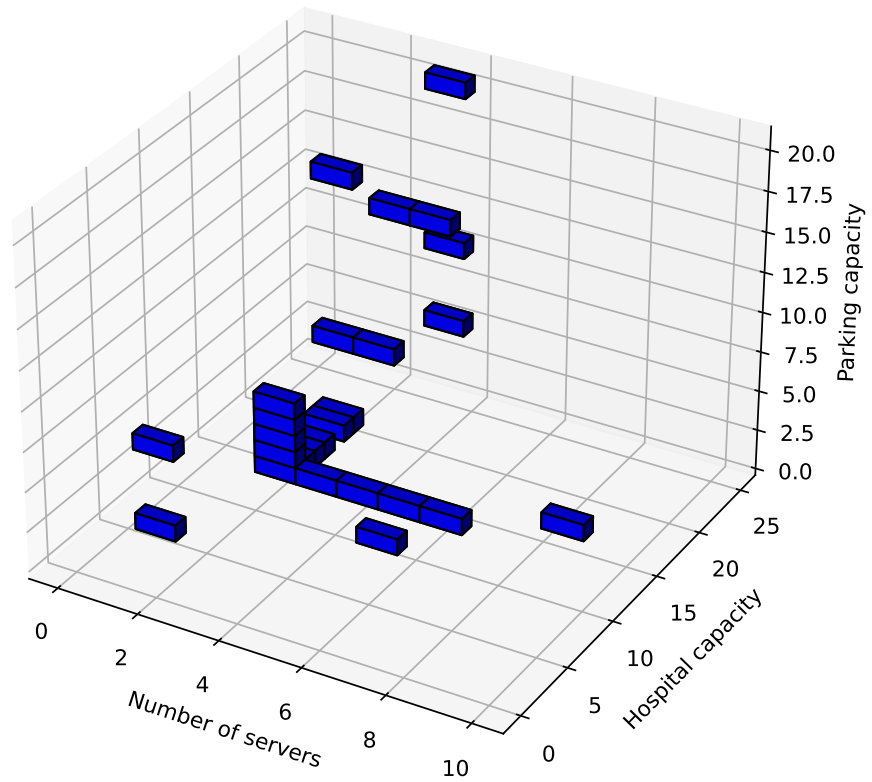


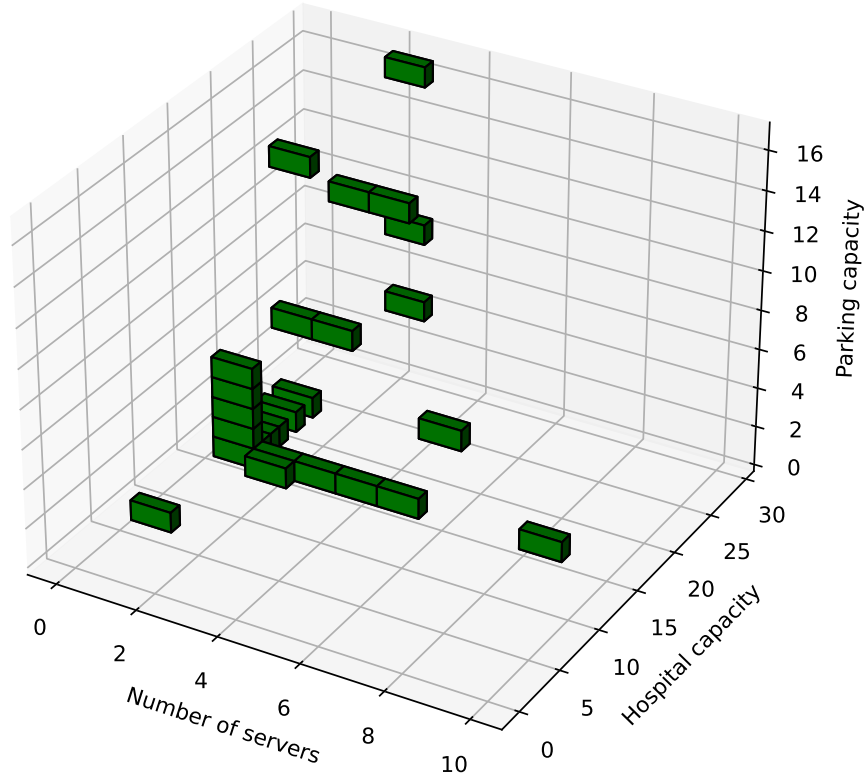
Figure 5.3: Number of times each value of  $\lambda_2$  is used in the dataset for  $\lambda_2 \in [0, 10]$

In addition, in terms of the values of  $C, N$  and  $M$  of the two players  $A$  and  $B$ , only some combinations of these values were explored. Figures 5.4 and 5.5 show the explored combinations of the values of  $C, N$  and  $M$  for player  $A$  and player  $B$  respectively.



Explored parameter space for player A

Figure 5.4: Explored combinations of the values of  $C$ ,  $N$  and  $M$  for player  $A$

Explored parameter space for player  $B$ Figure 5.5: Explored combinations of the values of  $C$ ,  $N$  and  $M$  for player  $B$ 

From Figures 5.4 and 5.5 it can be seen that only a small subset of the possible combinations of the values of  $C$ ,  $N$  and  $M$  were explored. Note here that for each of this explored combinations additional combinations of the values of  $\lambda_2$ ,  $\lambda_1^A$ ,  $\lambda_1^B$ ,  $\mu^A$ ,  $\mu^B$ ,  $\alpha$  and  $t$  were also used.

## 5.4 What if scenarios

This section aims to analyse the generated dataset and investigate how gaming can affect the performance measures of the two hospitals. In addition, under this gaming framework, this section aims to research how players (i.e. hospitals) can be incentivised in such a way so that they can be motivated to play a more cooperative game with the EMS provider.

### 5.4.1 Example 1

Consider the game defined by the parameters in Table 5.4.

Table 5.4: Parameter values for the first example of the what if scenarios.

Distributor				Queueing system A					Queueing system B				
$\lambda_2$	t	$\hat{P}$	$\alpha$	$\lambda_1^A$	$\mu^A$	$C^A$	$N^A$	$M^A$	$\lambda_1^B$	$\mu^B$	$C^B$	$N^B$	$M^B$
2	2	0.95	0.5	1	2	2	10	6	2	2.5	2	10	6

The set of possible actions to choose from for player 1 and player 2 is the set of thresholds that the EDs can choose from:

$$T^A \in [1, 10], \quad T^B \in [1, 10] \quad (5.1)$$

The resulting payoff matrices of the two hospitals, A and B, along with the corresponding routing matrix R, are shown in (5.2):

$$\begin{aligned}
 A = & \begin{bmatrix} 0.99919 & 0.99919 & 0.99919 & 0.99919 & 0.99919 & 0.99919 & 0.99919 & 0.99919 & 0.99919 & 0.99919 \\ 0.99937 & 0.99932 & 0.9993 & 0.99928 & 0.99927 & 0.99925 & 0.99924 & 0.99922 & 0.9992 & 0.99919 \\ 0.99985 & 0.99971 & 0.99964 & 0.99957 & 0.99952 & 0.99947 & 0.99942 & 0.99937 & 0.99931 & 0.9992 \\ 0.99998 & 0.99999 & 0.99992 & 0.99984 & 0.99977 & 0.9997 & 0.99963 & 0.99955 & 0.99946 & 0.99931 \\ 0.99943 & 0.9998 & 0.99998 & 1. & 0.99995 & 0.99989 & 0.99981 & 0.99973 & 0.99962 & 0.99944 \\ 0.99802 & 0.99867 & 0.99957 & 0.99989 & 0.99999 & 0.99999 & 0.99995 & 0.99987 & 0.99976 & 0.99957 \\ 0.99595 & 0.99595 & 0.99845 & 0.9994 & 0.9998 & 0.99996 & 1. & 0.99997 & 0.99988 & 0.99969 \\ 0.99379 & 0.99379 & 0.99643 & 0.99843 & 0.99934 & 0.99976 & 0.99994 & 1. & 0.99996 & 0.99981 \\ 0.99253 & 0.99253 & 0.99253 & 0.99671 & 0.99843 & 0.99928 & 0.99972 & 0.99993 & 1. & 0.99991 \\ 0.99347 & 0.99347 & 0.99347 & 0.99347 & 0.99594 & 0.99786 & 0.99893 & 0.99954 & 0.99988 & 1. \end{bmatrix} \\
B = & \begin{bmatrix} 0.99852 & 0.99868 & 0.99909 & 0.99954 & 0.99992 & 0.99997 & 0.99961 & 0.99901 & 0.9987 & 0.99908 \\ 0.99852 & 0.99861 & 0.99885 & 0.99917 & 0.99952 & 0.99984 & 1. & 0.99987 & 0.99922 & 0.99908 \\ 0.99852 & 0.99858 & 0.99877 & 0.99903 & 0.99932 & 0.99962 & 0.99986 & 0.99999 & 0.99992 & 0.99928 \\ 0.99852 & 0.99856 & 0.99871 & 0.99892 & 0.99917 & 0.99943 & 0.99967 & 0.99987 & 0.99999 & 0.99987 \\ 0.99852 & 0.99855 & 0.99867 & 0.99884 & 0.99905 & 0.99927 & 0.9995 & 0.99971 & 0.9999 & 1. \\ 0.99852 & 0.99853 & 0.99863 & 0.99877 & 0.99895 & 0.99915 & 0.99935 & 0.99955 & 0.99976 & 0.99996 \\ 0.99852 & 0.99852 & 0.99859 & 0.99872 & 0.99887 & 0.99904 & 0.99922 & 0.99941 & 0.99961 & 0.99985 \\ 0.99852 & 0.99852 & 0.99856 & 0.99867 & 0.9988 & 0.99895 & 0.99911 & 0.99928 & 0.99947 & 0.99972 \\ 0.99852 & 0.99852 & 0.99852 & 0.99861 & 0.99872 & 0.99885 & 0.99899 & 0.99914 & 0.99932 & 0.99957 \\ 0.99852 & 0.99852 & 0.99852 & 0.99852 & 0.99858 & 0.99869 & 0.9988 & 0.99893 & 0.99909 & 0.99934 \end{bmatrix}
 \end{aligned} \quad (5.2)$$

$$R = \begin{bmatrix} 0.5348 & 0.1965 & 0.1282 & 0.0711 & 0.0221 & 0 & 0 & 0 & 0 & 0 \\ 0.9552 & 0.6177 & 0.4967 & 0.4041 & 0.3288 & 0.264 & 0.2034 & 0.1379 & 0.0471 & 0 \\ 1 & 0.7347 & 0.6156 & 0.5241 & 0.4492 & 0.3843 & 0.3239 & 0.2599 & 0.1752 & 0.0232 \\ 1 & 0.8217 & 0.7043 & 0.6138 & 0.5394 & 0.475 & 0.415 & 0.3523 & 0.2722 & 0.1348 \\ 1 & 0.8903 & 0.7746 & 0.6852 & 0.6116 & 0.5476 & 0.4882 & 0.4268 & 0.3504 & 0.224 \\ 1 & 0.9466 & 0.8327 & 0.7445 & 0.6717 & 0.6084 & 0.5496 & 0.4894 & 0.4161 & 0.2984 \\ 1 & 1 & 0.8829 & 0.796 & 0.724 & 0.6613 & 0.6033 & 0.5442 & 0.4734 & 0.3627 \\ 1 & 1 & 0.9308 & 0.8447 & 0.7734 & 0.7111 & 0.6535 & 0.5953 & 0.5267 & 0.4215 \\ 1 & 1 & 1 & 0.9033 & 0.8311 & 0.7681 & 0.7101 & 0.6518 & 0.5843 & 0.4831 \\ 1 & 1 & 1 & 1 & 0.9405 & 0.8706 & 0.807 & 0.7446 & 0.6743 & 0.573 \end{bmatrix}$$

Using the Lemke-Howson algorithm on this example the following pure strategy Nash equilibrium is found:

$$\sigma^A = (0, 0, 0, 0, 0, 0, 0, 0, 0, 1), \quad \sigma^B = (0, 0, 0, 0, 0, 0, 0, 0, 0, 1) \quad (5.3)$$

For this example, there exists a Nash equilibrium of the game where both players choose a threshold of  $T^A = 10, T^B = 10$  at all times. This means that the two players' best response to each other is to only block ambulances when hospitals reach their maximum capacity.

The same conclusion can be obtained using a learning algorithm as well. Asymmetric replicator dynamics is also used here, not only to confirm the same previous result but also to show that the particular strategy is an ESS (see Section 4.2.3) and to observe how the strategies of these two players evolve to reach the particular equilibrium. Figure 5.6 shows the results of the asymmetric replicator dynamics algorithm for this example.

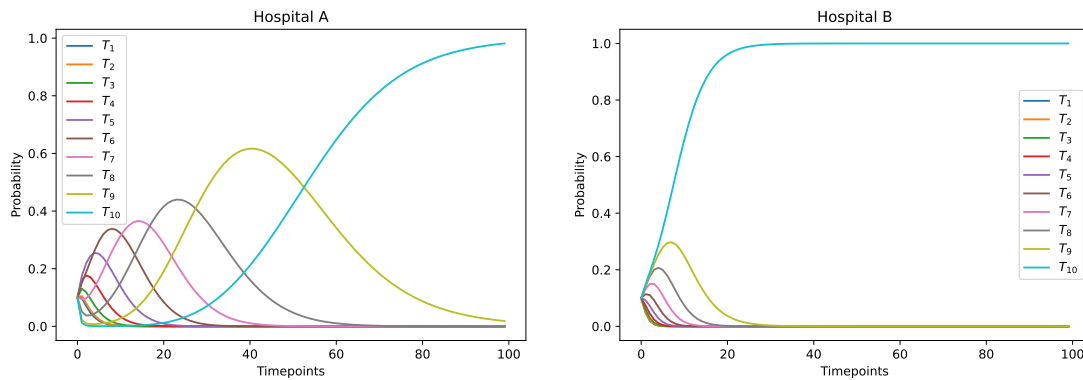


Figure 5.6: Example 1: Asymmetric replicator dynamics

What is more important in this example is how the two hospitals reached these decisions which also highlights the importance of using a learning algorithm. Hospital B is able to reach the final decision in a shorter amount of time while

hospital  $A$  takes longer and goes through numerous strategies to get there based on the strategy choices of hospital  $B$ . By observing the strategy choices of hospital  $A$  more closely, it can be seen that it starts out by blocking all ambulances and then slowly starts to unblock them.

Consider the utility function of the two players from equation (4.30):

$$U_{T^A, T^B}^i = 1 - \left( \hat{P} - P(W_i < t) \right)^2 \quad i \in A, B$$

where the hospitals' aim is to have a proportion of patients  $\hat{P}$  within the target time  $t$ . The utility function attempts to get the difference between the actual proportion of patients within the target time  $t$  and the target proportion  $\hat{P}$  as close to zero as possible.

Thus, for the current example, the fact that two hospitals are motivated to play a strategy of  $T^A = 10, T^B = 10$  means that the target of  $\hat{P} = 0.95$  and  $t = 2$  is effortlessly achieved given the current parameters. Hospitals  $A$  and  $B$  are able to achieve a target of  $t = 2$  and do not need to block ambulances unless they reach their maximum capacity.

What if the target time  $t$  was decreased from  $t = 2$  to  $t = 1.7$ ? Now, in order for the hospitals to maximise their utility, 95% of patients would need to receive treatment within 1.7 hours from their time of arrival instead of 2 hours. Using the Lemke-Howson algorithm, the following pure strategy Nash equilibrium arises:

$$\sigma^A = (0, 0, 1, 0, 0, 0, 0, 0, 0, 0), \quad \sigma^B = (0, 0, 0, 0, 1, 0, 0, 0, 0, 0) \quad (5.4)$$

This corresponds to hospital  $A$  playing a strategy of  $T^A = 3$  and hospital  $B$  playing a strategy of  $T^B = 5$ . This means that hospital  $A$  will only block ambulances when the number of patients in hospital  $A$  reaches 3 and hospital  $B$  will only block ambulances when the number of patients in hospital  $B$  reaches 5. Figure 5.7 shows the results of the asymmetric replicator dynamics algorithm with the decreased target time  $t = 1.7$ .

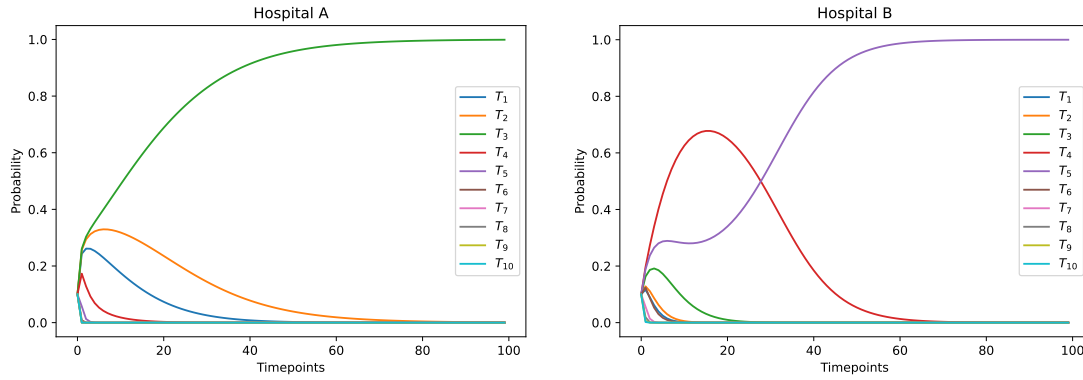


Figure 5.7: Example 1: Asymmetric replicator dynamics (what if  $t = 1.7$ )

It can be seen from Figure 5.7 that the two hospitals have reached the same output as the Lemke-Howson algorithm. Hospital  $A$  chooses a strategy of  $T^A = 3$  from a relatively early stage while hospital  $B$  first starts playing a strategy of  $T^B = 4$  and then changes to  $T^B = 5$  after a few iterations. Therefore, given the results of the Lemke-Howson algorithm and the asymmetric replicator dynamics algorithm, it can be seen that the more strict the time target  $t$  is, the more hospitals would want to play a strategy where the threshold is lower, and consequently, more ambulances will be blocked.

Additionally, the time target  $t$  can be decreased further to  $t = 1.5$ . Using the Lemke-Howson algorithm, the following pure strategy Nash equilibrium arises:

$$\sigma^A = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0), \quad \sigma^B = (0, 1, 0, 0, 0, 0, 0, 0, 0, 0) \quad (5.5)$$

This corresponds to hospital  $A$  playing a strategy of  $T^A = 1$  and hospital  $B$  playing a strategy of  $T^B = 2$ . Now consider the equivalent asymmetric replicator dynamics algorithm run on the modified example.

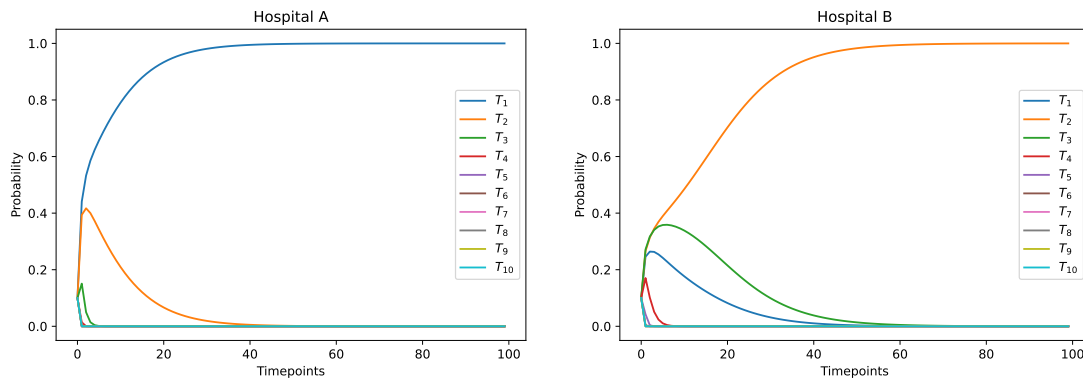


Figure 5.8: Example 1: Asymmetric replicator dynamics (what if  $t = 1.5$ )

It can be observed from Figure 5.8 that the two hospitals choose to play a strategy of  $T^A = 1$  and  $T^B = 2$ . This is the same as the pure strategy Nash equilibrium found using the Lemke-Howson algorithm. Therefore, it can be seen that by decreasing the time target  $t$  further, hospitals tend to play a lower threshold strategy and consequently, more ambulances will be blocked.

Table 5.5 shows the strategies played and the performance measures that arise from these strategy choices for the three different time targets  $t = 2, 1.7, 1.5$ .

Table 5.5: Example 1: Strategies played and performance measures

Time target	Hospital A			Hospital B		
	Strategy	Waiting	Blocking	Strategy	Waiting	Blocking
$t = 2$	$T = 10$	0.198	0.0006	$T = 10$	0.186	0.0006
$t = 1.7$	$T = 3$	0.102	0.0904	$T = 5$	0.213	0.0894
$t = 1.5$	$T = 1$	0.033	0.5852	$T = 2$	0.11	0.5592

It can be seen from Table 5.5 that the mean blocking time of ambulances increases as the time target  $t$  decreases. Similarly, the mean waiting time of patients decreases as the time target  $t$  decreases. Note that for hospital  $B$  there is a slight increase of the mean waiting time of patients when the time target  $t$  is decreased from  $t = 2$  to  $t = 1.7$ . That is because of the strategy played by the third player; the EMS. Observe the entries of the routing matrix  $R$  from equation (5.2). The best response of the EMS when the hospital play  $(T^A = 10, T^B = 10)$  is to send a proportion of  $R_{10,10} = 0.57$  patients to hospital  $A$  while the best response of the EMS when the hospital play  $(T^A = 3, T^B = 5)$  is to send a proportion of  $R_{3,5} = 0.45$  patients to hospital  $A$ . For that reason since hospital  $B$  receives more patients from the EMS when the time target  $t$  is decreased from  $t = 2$  to  $t = 1.7$ , the mean waiting time of patients increases slightly.

### 5.4.2 Example 2

Consider another example now where the parameters  $\lambda_2, \lambda_1^A \lambda_1^B$  are set to a relatively high value.

Table 5.6: Parameter values for the second example of the what if scenarios.

Distributor				Queueing system A					Queueing system B				
$\lambda_2$	t	$\hat{P}$	$\alpha$	$\lambda_1^A$	$\mu^A$	$C^A$	$N^A$	$M^A$	$\lambda_1^B$	$\mu^B$	$C^B$	$N^B$	$M^B$
10.7	2	0.95	0.9	4.5	2	3	6	5	6	3	2	7	4

Recall that the relative traffic intensity of an  $M|M|c$  queue is given by  $\rho = \frac{\lambda}{c\mu}$  where  $\lambda$  is the arrival rate,  $c$  is the number of servers and  $\mu$  is the service rate. The relative traffic intensity is a metric that measures how congested a queue is, based on the inflow and outflow of individuals in the queue. When  $\rho < 1$  the rate at which individuals leave the queue is larger than the rate at which individuals enter the queue and when  $\rho > 1$  the rate at which individuals enter the queue is larger than that of those leaving the queue [7].

In this example, without solving the game, the relative traffic intensity of each hospital cannot be calculated since the arrival rate of type 2 patients among the two hospitals may vary based on the strategy played by the EMS. Each hospital's relative traffic intensity is given by:

$$\rho^A = \frac{\lambda_1^A + \lambda_2 p^A}{C^A \mu^A}, \quad \rho^B = \frac{\lambda_1^B + \lambda_2(1 - p^A)}{C^B \mu^B}, \quad p^A \in [0, 1] \quad (5.6)$$

where  $p^A$  is the proportion of type 2 patients that are sent to hospital  $A$  by the EMS. By substituting in all the values for the parameters, the relative traffic intensity of each hospital is given by:

$$\rho^A = \frac{4.5 + 10.7p^A}{6}, \quad \rho^B = \frac{16.7 - 10.7p^A}{6}, \quad p^A \in [0, 1] \quad (5.7)$$

Thus, the traffic intensity of hospital  $A$  can take values  $\rho^A \in [0.75, 2.53]$  while the traffic intensity of hospital  $B$  can take values  $\rho^B \in [1, 2.78]$ . In fact the combined relative traffic intensity of the two hospitals is given by:

$$\rho^{A,B} = \rho^A + \rho^B = \frac{4.5 + 10.7p^A}{6} + \frac{16.7 - 10.7p^A}{6} = \frac{21.2}{6} = 3.53 \quad (5.8)$$

Any value of  $\rho^{A,B} > 2$  indicates that the combined inflow of the two hospitals is higher than the combined outflow and in this case it is well above 2. Given these two highly congested hospitals, consider the game played by the EMS and the two hospitals. Note that for the presentation of these data an affine transformation has been applied to the values of the payoff matrices to make it easier for the reader ( $A_{ij} = 10,000(a_{ij} - 0.999)$  and  $B_{ij} = 10,000(b_{ij} - 0.999)$ ).



$$\begin{aligned}
A &= \begin{bmatrix} 5.0518 & 5.0518 & 5.0518 & 5.0518 & 5.0518 & 5.0518 & 5.0518 \\ 5.4989 & 5.4977 & 5.4960 & 5.4924 & 5.4844 & 5.4654 & 5.3875 \\ 6.8232 & 6.8192 & 6.8150 & 6.8065 & 6.7871 & 6.7334 & 6.4906 \\ 9.0298 & 9.0244 & 9.0187 & 9.0078 & 8.9827 & 8.9082 & 8.5145 \\ 9.9996 & 9.9994 & 9.9992 & 9.9987 & 9.9972 & 9.9893 & 9.8571 \\ 8.7740 & 8.8006 & 8.8249 & 8.8660 & 8.9438 & 9.1295 & 9.7157 \end{bmatrix} \\
B &= \begin{bmatrix} 1.7127 & 2.5822 & 4.6186 & 6.8497 & 8.9418 & 9.9999 & 8.2148 \\ 1.7127 & 2.5477 & 4.5634 & 6.8047 & 8.9150 & 9.9996 & 8.3358 \\ 1.7127 & 2.4528 & 4.3784 & 6.6441 & 8.8278 & 9.9965 & 8.5306 \\ 1.7127 & 2.4141 & 4.2867 & 6.5470 & 8.7656 & 9.9919 & 8.6745 \\ 1.7127 & 2.3415 & 4.0998 & 6.3265 & 8.6058 & 9.9716 & 8.9634 \\ 1.7127 & 2.1269 & 3.4930 & 5.4885 & 7.8353 & 9.7075 & 9.7322 \end{bmatrix} \\
R &= \begin{bmatrix} 0.22 & 0.06 & 0.05 & 0.05 & 0.04 & 0.03 & 0.01 \\ 0.95 & 0.6 & 0.47 & 0.37 & 0.28 & 0.2 & 0.11 \\ 0.97 & 0.81 & 0.72 & 0.62 & 0.51 & 0.37 & 0.21 \\ 0.97 & 0.85 & 0.77 & 0.68 & 0.57 & 0.44 & 0.26 \\ 0.98 & 0.89 & 0.83 & 0.76 & 0.66 & 0.53 & 0.35 \\ 1 & 0.95 & 0.91 & 0.87 & 0.8 & 0.7 & 0.52 \end{bmatrix} \tag{5.9}
\end{aligned}$$

Matrices  $A$  and  $B$  are the payoff matrices of the two hospitals and can be used to get the Nash equilibrium of the game. Using the Lemke-Howson algorithm, the following pure strategy Nash equilibrium is found:

$$\sigma^A = (0, 0, 0, 0, 1, 0), \quad \sigma^B = (0, 0, 0, 0, 0, 1, 0) \tag{5.10}$$

The output of the Lemke-Howson algorithm indicates that the game is at a Nash equilibrium when hospital  $A$  plays a strategy of  $T^A = 5$  and hospital  $B$  plays a strategy of  $T^B = 6$ . In fact, this is the only Nash equilibrium of the game. A similar outcome is obtained when using a learning algorithm. Figure 5.9 shows the output of the run of the asymmetric replicator dynamics algorithm.

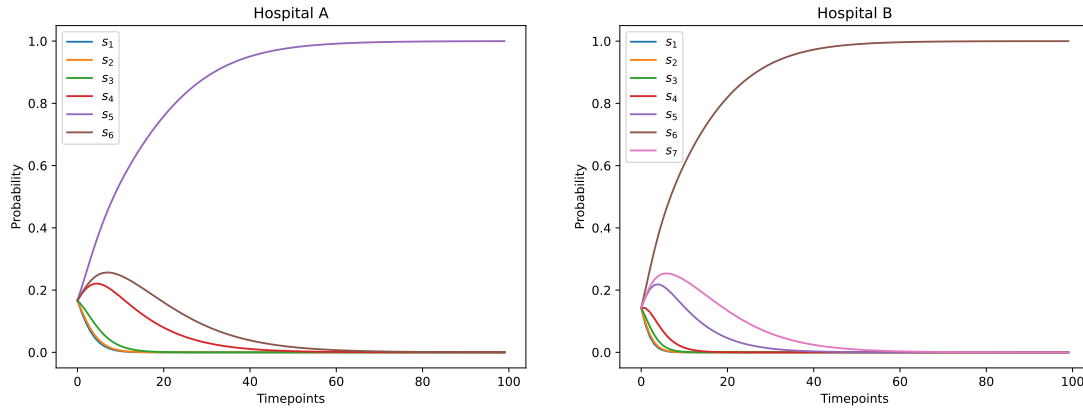


Figure 5.9: Example 2: Asymmetric replicator dynamics

Asymmetric replicator dynamics converges to the same strategy as the Lemke-Howson algorithm, which is the pair of strategies  $(T^A = 5, T^B = 6)$ .

Consider now the concept of the Price of Anarchy ( $PoA$ ) [124].  $PoA$  is a measure in game theory that is used to quantify the efficiency of the outcome of a game when players behave in a selfish way. Examples of using this measure in a healthcare setting include [76, 79, 105]. More specifically, the  $PoA$  measures the ratio between the worst possible equilibria outcome of a game (so that no players have an incentive to deviate) and the best possible centrally controlled outcome of the game (the best possible collective situation). The  $PoA$  of a game is defined as:

$$PoA = \frac{\max_{s \in E} F(s)}{\min_{s \in S} F(s)} \quad (5.11)$$

where  $S$  is the set of all possible strategies of the players,  $E$  is the set of all possible equilibria of the game and  $F(s)$  is a cost function to measure the efficiency of when the players play strategy  $s$ . The  $PoA$  is a measure that is used to describe the overall efficiency of the game, rather the independent efficiency of each player.

For the purpose of this study a measure is introduced that considers the ratio between each hospital's best achievable blocking time and the one that is being played. This is defined as the compartmentalised price of anarchy of the players of the game and is defined as  $PoA_i(s)$  where  $i \in \{A, B\}$  and  $s \in S$  is the strategy played by player  $i$ . The compartmentalised price of anarchy is defined as:

$$PoA_i(s) = \frac{B_i(s)}{\min_{s' \in S} B_i(s')} \quad (5.12)$$

That is the ratio between the blocking time of player  $i$  when playing the chosen strategy  $s$  and the minimum blocking time player  $i$  could achieve from any strategy  $s' \in S$ . In other words, this is the range of values that the compartmentalised  $PoA$  can take are  $PoA_i(s) \in [1, +\infty)$ , where 1 is the best possible outcome. Consider, once again the asymmetric replicator dynamics run from Figure 5.9. One may plot the compartmentalised  $PoA$  of each player alongside the outcome of the learning algorithm. Figure 5.10 shows the asymmetric replicator dynamics run of the game along with the compartmentalised  $PoA$  of each player.

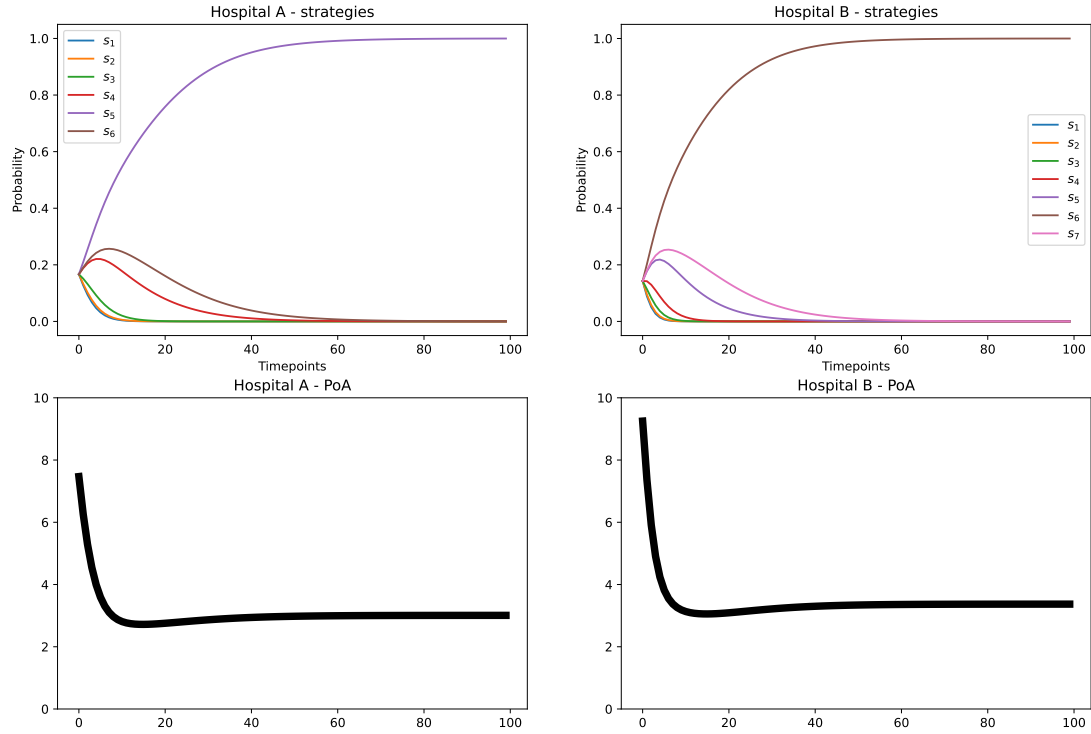


Figure 5.10: Example 2: Compartmentalised  $PoA$

In Figure 5.10 it can be seen that the  $PoA$  of both players is always greater than 1. This means that the chosen strategies are not the best possible strategies in terms of minimising the blocking time of ambulances. So, the question to be asked here is: what can be changed in the game to escape these learned inefficiencies?

For the rest of this section, asymmetric replicator dynamics will be used in a slightly different manner. One could run asymmetric replicator dynamics on a modified version of the game where certain parameters are changed and observe the outcome of the learning algorithm. Although this is a sensible approach, doing that means that the learned strategies from Figure 5.10 are not considered. As mentioned earlier, the aim is to investigate how to escape these learned inefficiencies, not what would happen if they never existed. Therefore, a different approach is considered. Learning algorithms require only matrices  $A$  and  $B$  to

run. Therefore, asymmetric replicator dynamics is ran on the original game and stopped at a certain point. After changing the parameters of the game, the new matrices arise,  $\tilde{A}$  and  $\tilde{B}$ . Asymmetric replicator dynamics is then ran again on the new matrices while using the final strategies from the previous run as the initial strategies. This approach is used to investigate how the strategies and the  $PoA$  of the players change when the values of the parameters change.

One sensible idea would be to increase the artificial values of the hospital capacities  $N^i$ . The same learning algorithm is ran again with the same parameters as before, but at some point the artificial values are increased from  $N^A = 6$  and  $N^B = 7$  to  $N^A = 7$  and  $N^B = 8$ . Figure 5.11 shows the output of the asymmetric replicator dynamics algorithm and the compartmentalised  $PoA$  of each player. Note that, when the artificial values are increased, hospital  $A$  gets a new strategy of  $T^A = 7$  and hospital  $B$  gets a new strategy of  $T^B = 8$ .

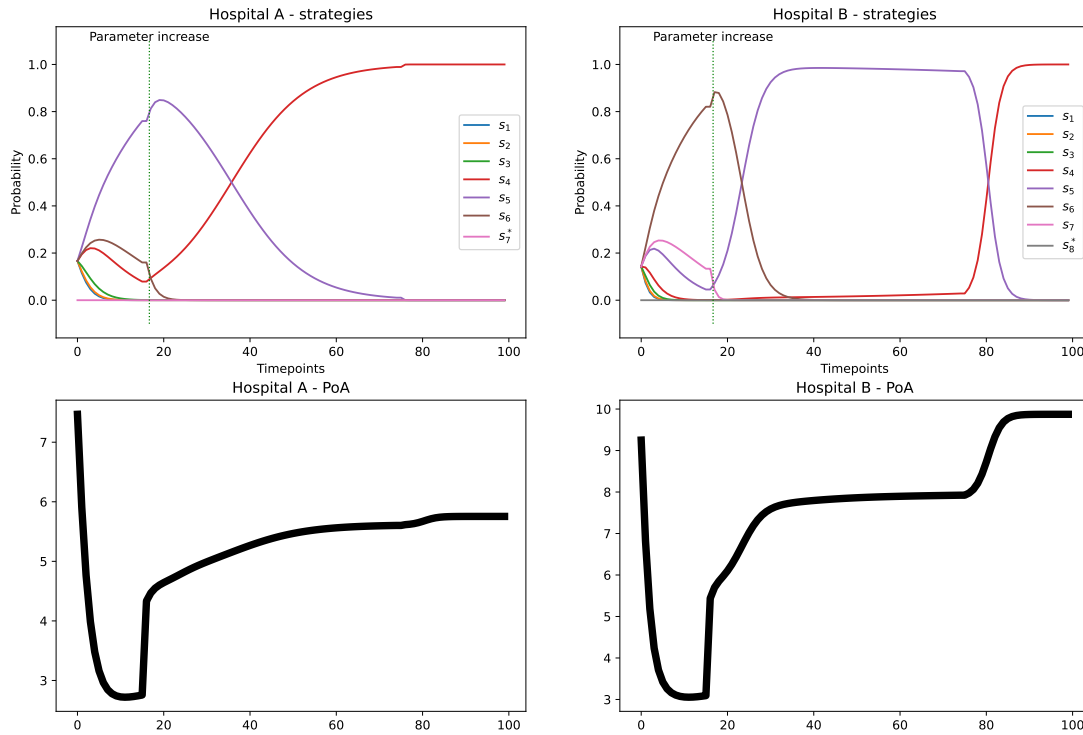


Figure 5.11: Example 2: Asymmetric replicator dynamics and compartmentalised  $PoA$  with increased hospital capacity  $N$

In fact, by increasing the capacity of the hospitals, both hospitals become even more inefficient. The moment the capacities are increased, both hospitals change their strategies to close their doors for ambulance patients even earlier. That causes the blocking time of ambulances to increase and thus the  $PoA$  of both players increase. This happens because there are so many individuals coming in that the hospitals cannot cope with the demand.

Similarly, the same learning algorithm is ran again with the same parameters as before, but at some point the parking capacity of the hospitals is increased from  $M^A = 5$  and  $M^B = 4$  to  $M^A = 20$  and  $M^B = 16$ . Figure 5.12 shows the output of the asymmetric replicator dynamics algorithm and the compartmentalised *PoA* of each player.

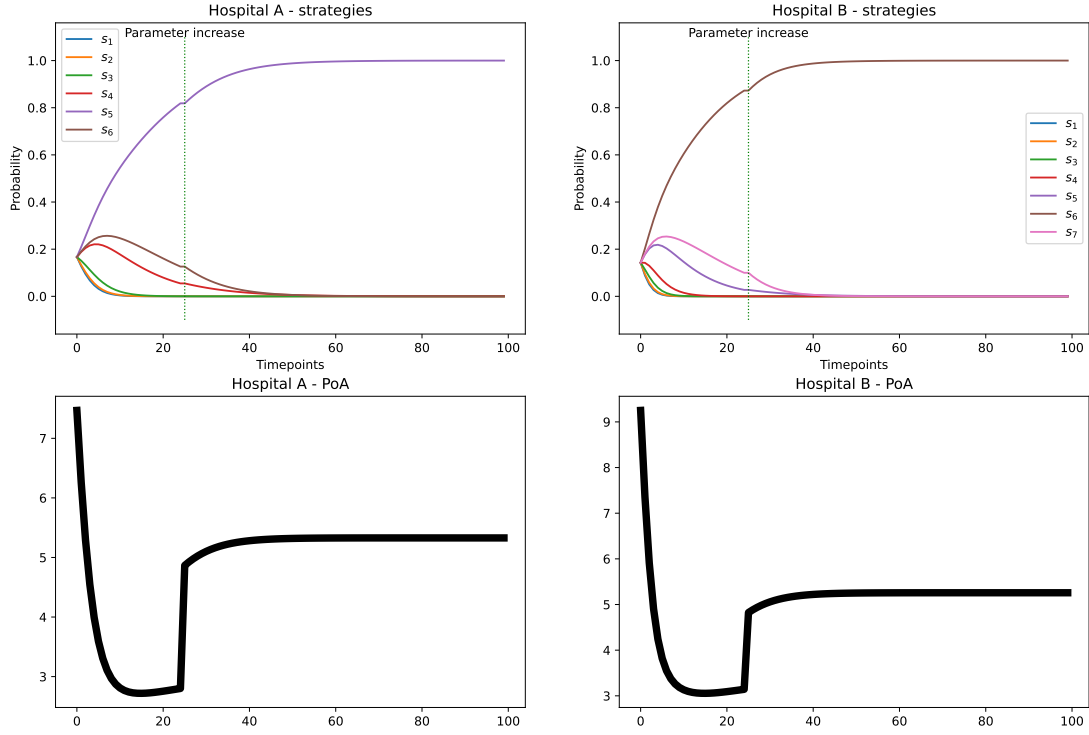


Figure 5.12: Example 2: Asymmetric replicator dynamics and compartmentalised *PoA* with increased parking capacity  $M$

Although this time the strategies do not change, the *PoA* of both players increases, which makes the entire game more inefficient in terms of the blocking time of ambulances. This is because the parking capacity of the hospitals is increased, which means that patients can now wait longer in the parking space. Therefore, the *PoA* gets worse because the blocking time of ambulances when playing strategies  $T^A = 5$  and  $T^B = 6$  is increased.

Figures 5.12 and 5.11 demonstrate that increasing the hospital capacities or the parking capacities make the outcome of the game even more inefficient. Consider now the case that the arrival rate of type 2 individuals  $\lambda_2$  is increased at some point during the learning algorithm. Similar to before, the learning algorithm is ran with the initial parameters, and then  $\lambda_2$  is increased. Figure 5.13 shows the output of the asymmetric replicator dynamics algorithm and the compartmentalised *PoA* of each player.

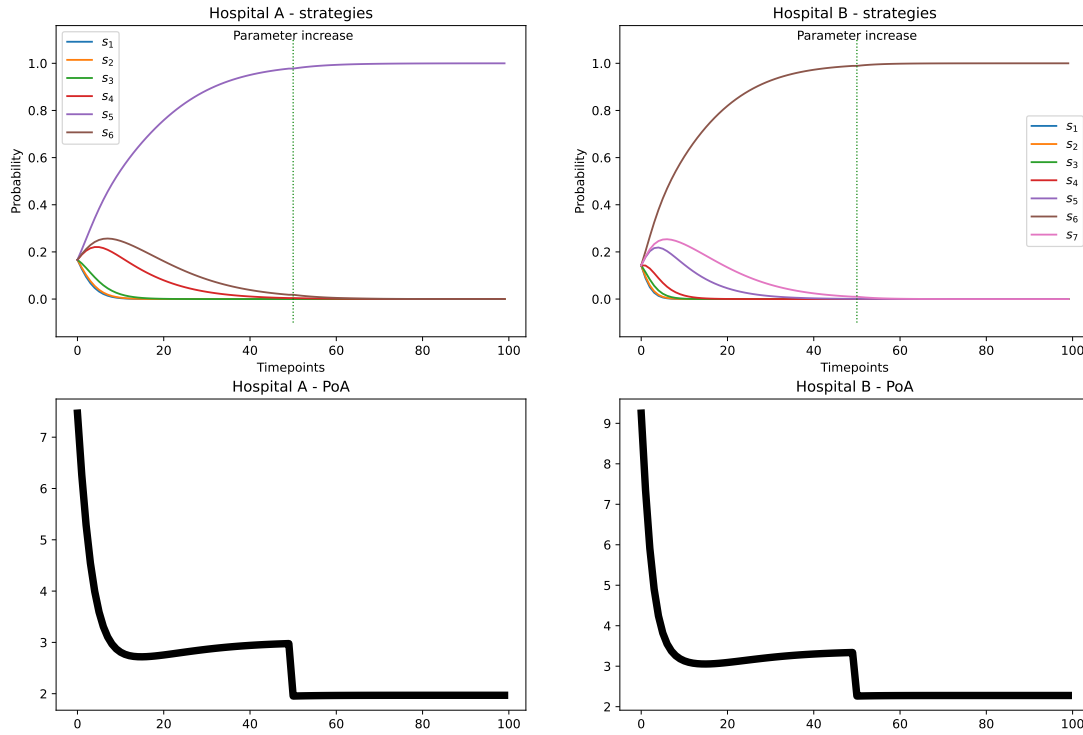


Figure 5.13: Example 2: Asymmetric replicator dynamics and compartmentalised  $PoA$  with increased arrival of type 2 patients  $\lambda_2$

While the strategies of the players don't change, the  $PoA$  of both players actually decreases. Increasing the arrival rate of type 2 patients (ambulance patients) shouldn't make the game more efficient, but it does. That is not because the blocking time is decreased. As a matter of fact the blocking time of ambulances increases for both players. The reason why the  $PoA$  decreases can be traced back to the definition of the  $PoA$  from equation (5.12).  $PoA_i(s)$  is defined as ratio between the blocking time when strategy  $s$  is played and the best achievable blocking time from all strategies. Therefore, when the best achievable blocking time becomes worse, the  $PoA$  of the players decreases, which is exactly what happens in this case. Although this is an interesting outcome, it is not a sensible one. Increasing the arrival rate of an already flooded system does not make the system more efficient. This is similar to the findings of [77].

A more appropriate way to increase the efficiency of the system is to increase the number of staff  $C$ . Figure 5.14 shows the output of the asymmetric replicator dynamics algorithm and the compartmentalised  $PoA$  of each player when the number of staff available is increased from  $C^A = 3$  and  $C^B = 2$  to  $C^A = 4$  and  $C^B = 3$ .

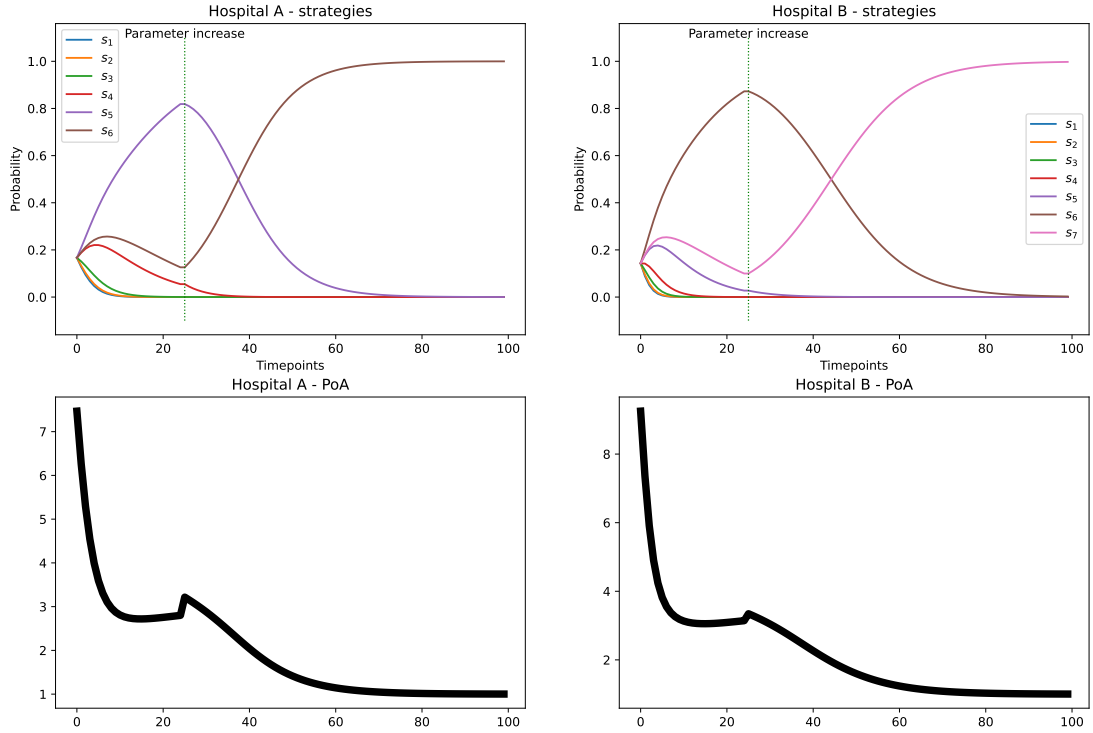


Figure 5.14: Example 2: Asymmetric replicator dynamics and compartmentalised *PoA* with increased number of staff  $C$

Not only do the strategies of both players change, but the *PoA* of both players also decreases. Strategies  $T^A = 5$  and  $T^B = 6$  are no longer the strategies that are played. Instead, players play strategies  $T^A = 6$  and  $T^B = 7$ , which makes the hospitals accept more patient from ambulances, which in turn decreases the mean blocking time of ambulances. As a result the *PoA* decrease for both players indicating that the game has reached a more efficient outcome in terms of the overall blocking time.

Although, Figure 5.14 shows a valid way to increase the efficiency of the system, it might not be the most cost-effective method, since more staff means more costs. Another attempt to increase the efficiency of the system is to apply certain incentives to the players to change their strategies. The outcomes of the asymmetric replicator dynamics algorithm are derived directly from matrices  $A$  and  $B$ . Therefore by carefully altering the values of these matrices, the strategies of the players can be changed. In other words, by penalising the chosen strategies the players could be forced to play different ones. For instance, consider the following example of matrices  $A$  and  $B$ :

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \end{pmatrix} \quad (5.13)$$

These matrices now form a game that can be solved using the asymmetric replicator dynamics algorithm. In addition, these matrices can also be altered so that certain strategies are penalised. Strategies for player  $A$  are represented by rows of matrix  $A$ , and strategies for player  $B$  are represented by columns of matrix  $B$ . Therefore, an entire row of matrix  $A$  can be multiplied by a certain constant  $p \in [0, 1]$  to penalise a strategy of player  $A$  and an entire column of matrix  $B$  can be multiplied by the same factor to penalise a strategy of player  $B$ . The resulting two matrices are denoted by  $\tilde{A}$  and  $\tilde{B}$  and have certain strategies

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ pa_{21} & pa_{22} & pa_{23} & pa_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & pb_{12} & b_{13} & b_{14} \\ b_{21} & pb_{22} & b_{23} & b_{24} \\ b_{31} & pb_{32} & b_{33} & b_{34} \end{pmatrix} \quad (5.14)$$

Therefore consider the payoff matrices of the two hospitals  $A$  and  $B$  with the played strategies highlighted.

$$A = \begin{bmatrix} 5.0518 & 5.0518 & 5.0518 & 5.0518 & 5.0518 & 5.0518 & 5.0518 \\ 5.4989 & 5.4977 & 5.4960 & 5.4924 & 5.4844 & 5.4654 & 5.3875 \\ 6.8232 & 6.8192 & 6.8150 & 6.8065 & 6.7871 & 6.7334 & 6.4906 \\ 9.0298 & 9.0244 & 9.0187 & 9.0078 & 8.9827 & 8.9082 & 8.5145 \\ \mathbf{9.9996} & \mathbf{9.9994} & \mathbf{9.9992} & \mathbf{9.9987} & \mathbf{9.9972} & \mathbf{9.9893} & \mathbf{9.8571} \\ 8.7740 & 8.8006 & 8.8249 & 8.8660 & 8.9438 & 9.1295 & 9.7157 \end{bmatrix}$$

$$B = \begin{bmatrix} 1.7127 & 2.5822 & 4.6186 & 6.8497 & 8.9418 & \mathbf{9.9999} & 8.2148 \\ 1.7127 & 2.5477 & 4.5634 & 6.8047 & 8.9150 & \mathbf{9.9996} & 8.3358 \\ 1.7127 & 2.4528 & 4.3784 & 6.6441 & 8.8278 & \mathbf{9.9965} & 8.5306 \\ 1.7127 & 2.4141 & 4.2867 & 6.5470 & 8.7656 & \mathbf{9.9919} & 8.6745 \\ 1.7127 & 2.3415 & 4.0998 & 6.3265 & 8.6058 & \mathbf{9.9716} & 8.9634 \\ 1.7127 & 2.1269 & 3.4930 & 5.4885 & 7.8353 & \mathbf{9.7075} & 9.7322 \end{bmatrix}$$

Hospital  $A$  plays a strategy of  $T^A = 5$  and hospital  $B$  plays a strategy of  $T^B = 6$ . Consider a set of penalties that are applied to the entries of matrices  $A$  and  $B$  that could be used to encourage the hospitals to play different strategies. This can be done by multiplying the entire row of matrix  $A$  that corresponds to the strategy



$T^A = 5$  by a penalty factor  $p \in [0, 1]$  and multiplying the entire column of matrix  $B$  that corresponds to the strategy  $T^B = 6$  by the same penalty factor. Note here that in a real-life scenario, applying such a penalty factor would correspond to carefully incentivising the hospitals. The precision that is required to choose the penalty factor is equivalent on the precision required to apply such incentives to hospital staff.

By applying a penalty to the strategies of  $T^A = 5$  and  $T^B = 6$ , the resulting payoff matrices  $\tilde{A}$  and  $\tilde{B}$  are given. Note that the chosen penalty factor of 0.9997 is applied to the payoff matrices and then the affine transformation that was applied to the matrices in (5.9) is also applied here.

$$\tilde{A} = \begin{bmatrix} 5.0518 & 5.0518 & 5.0518 & 5.0518 & 5.0518 & 5.0518 & 5.0518 \\ 5.4989 & 5.4977 & 5.4960 & 5.4924 & 5.4844 & 5.4654 & 5.3875 \\ 6.8232 & 6.8192 & 6.8150 & 6.8065 & 6.7871 & 6.7334 & 6.4906 \\ 9.0298 & 9.0244 & 9.0187 & 9.0078 & 8.9827 & 8.9082 & 8.5145 \\ \textcolor{blue}{6.9996} & \textcolor{blue}{6.9994} & \textcolor{blue}{6.9992} & \textcolor{blue}{6.9987} & \textcolor{blue}{6.9972} & \textcolor{blue}{6.9893} & \textcolor{blue}{6.8571} \\ 8.7740 & 8.8006 & 8.8249 & 8.8660 & 8.9438 & 9.1295 & 9.7157 \end{bmatrix}$$

$$\tilde{B} = \begin{bmatrix} 1.7127 & 2.5822 & 4.6186 & 6.8497 & 8.9418 & \textcolor{red}{6.9999} & 8.2148 \\ 1.7127 & 2.5477 & 4.5634 & 6.8047 & 8.9150 & \textcolor{red}{6.9996} & 8.3358 \\ 1.7127 & 2.4528 & 4.3784 & 6.6441 & 8.8278 & \textcolor{red}{6.9965} & 8.5306 \\ 1.7127 & 2.4141 & 4.2867 & 6.5470 & 8.7656 & \textcolor{red}{6.9919} & 8.6745 \\ 1.7127 & 2.3415 & 4.0998 & 6.3265 & 8.6058 & \textcolor{red}{6.9716} & 8.9634 \\ 1.7127 & 2.1269 & 3.4930 & 5.4885 & 7.8353 & \textcolor{red}{6.7076} & 9.7322 \end{bmatrix}$$

Having penalised the strategies of  $T^A = 5$  and  $T^B = 6$ , Figure 5.15 shows the outcome of asymmetric replicator dynamics when initialising the algorithm with the initial matrices  $A$  and  $B$  and at some point replacing them with the penalised ones  $\tilde{A}$  and  $\tilde{B}$ .

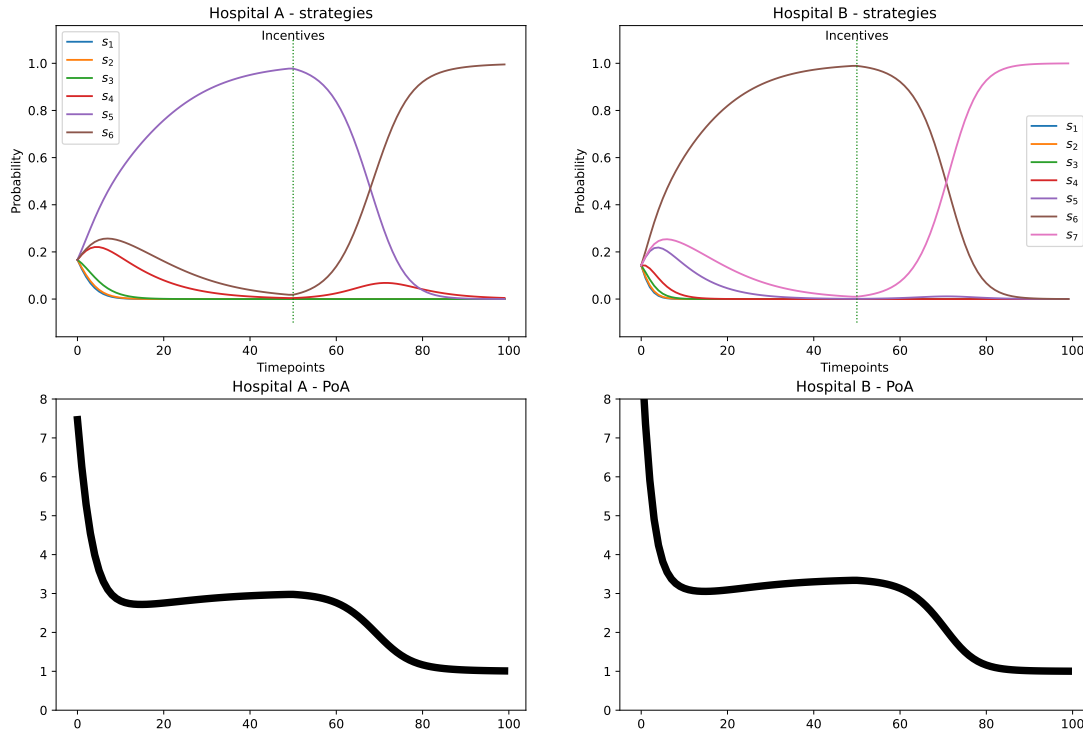


Figure 5.15: Example 2: Asymmetric replicator dynamics and compartmentalised *PoA* with incentivisation.

It can be seen that the hospitals start out by playing their usual strategies of  $T^A = 5$  and  $T^B = 6$ . After the penalised matrices  $\tilde{A}$  and  $\tilde{B}$  are introduced, the hospitals start to play strategies of  $T^A = 6$  and  $T^B = 7$ . That has a similar effect to the ones of Figure 5.14 but no additional members of staff have been added. Additionally, the *PoA* for both hospitals is decreased, indicating that the overall blocking time is reduced.

The results of this example show that using careful incentivisation on a managerial level can result in an improvement of the overall blocking time for the hospitals. Although, arguably by adding more staff, the hospitals could have achieved a greater reduction in the blocking time, these results indicate that when that is not possible or not desirable, incentivisation can be used to achieve a similar effect.

## 5.5 Chapter summary

This chapter presents the results of the numerical experiments conducted to study the behavioural patterns that can emerge from the interaction between the two EDs and the EMS. Matrices  $A$ ,  $B$  (the payoff matrices of the two EDs) and  $R$  (the routing matrix of the EMS) were generated and analysed for different values

of the parameters to study the behaviour that emerges from playing the game. Additional numerical experiments can also be found in Appendix B.

Section 5.2 describes the data collection process. The parameter sets that were used to generate the dataset are presented in along with the structure of the repository that contains all data. Section 5.3 gives an overview and some descriptive statistics of the data. Some plots are also presented to give a visual representation of the data and to highlight some interesting patterns.

Section 5.4 presents the results of the numerical experiments and discusses the implications of the results. More specifically, matrices  $A$ ,  $B$  and  $R$  are used to run the asymmetric replicator dynamics algorithm. Several what-if scenarios are presented and discussed where the system is flooded with individuals. Such scenarios include the effect of increasing arrival rates, the effect of increasing the number of available servers, the effect of increasing the different capacities and a certain incentivisation approach. The incentivisation approach is based on the idea carefully penalising some of the EDs strategies could lead them to choose strategies that are more cooperative and can help reduce the overall blocking time.

# Chapter 6

## Extending to agent-based model

### 6.1 Introduction

This chapter aims to explore an extension to the game described earlier. The extension is based on the idea of an agent-based model [39, 47, 80]. What if the servers of the queueing system described in Section 3 could be treated as entities that could make their own decisions?

The idea here is that servers could choose their own service rate so as to maximise their own utility. This would mean that the servers would be able to choose their own speed at which they serve customers while the system is running. Such decisions could be based on a number of factors, such as minimising the number of customers in the system, minimising the proportion of patients lost to the system, maximising their own idle time and so on.

The previous chapters are based on the assumption that hospitals and ambulances play a non-cooperative game and act in their own self-interest. Although this might be true in some cases, insights from ethnographic studies [6] have shown that some cooperation and empathy can be found among staff in such gaming environment. The motivation for this chapter was initiated by these insights.

This chapter consists of the following sections:

- Section 6.2 describes a variant of the queueing system described in Chapter 3 where the service rate of the servers is dependent on the state of the system.
- Section 6.3 describes another variant of the queueing system of Chapter 3 where the service rate of each server can be different.

- Section 6.4 combines the concepts described in Sections 6.2 and 6.3 to create a different variation of the queueing system where the rates of the servers are both dependent on the state of the system and the servers itself.
- Section 6.5 uses the state and server-dependent model to create an agent-based model where the servers can choose their own service rate in order to maximise their own utility.
- Section 6.6 uses the agent-based model to create a reinforcement learning model where the servers can learn to choose their own service rate in order to maximise their own utility.

## 6.2 State-dependent variation

This section focuses on creating a state-dependent variation of the queueing system described in Section 3. The state-dependent version is based on the idea that the speed of the servers could be based on the number of individuals present in the system.

In the previous chapters of this thesis, the value of the service rate  $\mu$  was set to a constant value. For this variation of the model  $\mu$  will have a different value depending on the number of individuals present in the two nodes of the queue. In other words  $\mu$  will take a different value depending on the state  $(u, v)$  of which the system is in. A parametric service rate  $\mu = \mu_{(u,v)}$  for a given  $(u, v) \in S$  is defined as follows:

$$\mu = \begin{cases} \mu_{(0,0)}, & \text{if } (u, v) = (0, 0) \\ \mu_{(0,1)}, & \text{if } (u, v) = (0, 1) \\ \vdots & \vdots \\ \mu_{(M,N)}, & \text{if } (u, v) = (M, N) \end{cases} \quad (6.1)$$

Consider an example of the queueing system described in Section 3 where the number of servers is set to  $C = 1$ , the threshold is set to  $T = 1$ , node 1 capacity is  $N = 2$  and node 2 capacity is  $M = 1$ . Figure 6.1 shows the state-dependent Markov chain model of the queueing system.

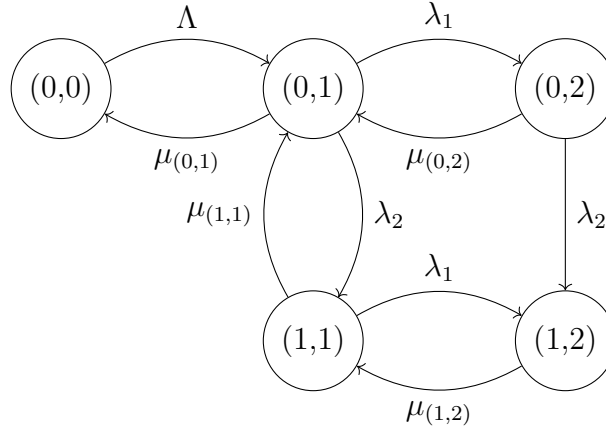


Figure 6.1: Markov chain example with  $C = 1, T = 1, N = 2, M = 1$

Consider the following value of  $\mu$  for the Markov chain model shown in Figure 6.1.

$$\mu = \begin{cases} \mu_{(0,0)}, & \text{if } (u, v) = (0, 0) \\ \mu_{(0,1)}, & \text{if } (u, v) = (0, 1) \\ \mu_{(0,2)}, & \text{if } (u, v) = (0, 2) \\ \mu_{(1,1)}, & \text{if } (u, v) = (1, 1) \\ \mu_{(1,2)}, & \text{if } (u, v) = (1, 2) \end{cases}$$

### 6.2.1 Implementation

The state-dependent variation of the queueing system was implemented using the python library `ciw` [142] only on the Discrete Event Simulation (DES) version of the queueing model described in Section 3.2. All the code used to implement the state-dependent variation of the queueing system is archived at [111] and developed openly on GitHub. More details on the code can be found in Appendix A. The `ciw` library is structured in a way that allows the user to create their own class for the service time distribution. This class must inherit from the `ciw.dists.Distribution` class and implement the `__init__` and `sample` methods. The code snippet in 6.1 shows the implementation of the state-dependent service time distribution class.

```
>>> import ciw
>>> class StateDependentExponential(
...     ciw.dists.Distribution
... ):
...     def __init__(self, rates):
...         self.rates = rates
...
...     def sample(self, t=None, ind=None):
...         """
```

```

...         This method is used to sample the service time for an
...         individual based on the current state
...         """
...         state = (
...             len(ind.simulation.nodes[1].individuals[0]),
...             len(ind.simulation.nodes[2].individuals[0]),
...         )
...         is_invalid_state = (
...             state[0] > 0 and state[1] < ind.simulation.threshold
...         )
...         if is_invalid_state:
...             state = (state[0] - 1, state[1] + 1)
...         rate = self.rates[state]
...         return random.expovariate(rate)

```

Code snippet 6.1: State-dependent service time distribution class.

The function `simulate_model` described in Section 3.2 takes `mu` as one of its arguments. If `mu` is set to a dictionary, with keys the states of the system and values the service rate for each state (as shown in code snippet 6.2), the service distribution that will be used in `ciw` will be a `StateDependentExponential` object. The following code shows the implementation of the `simulate_model` function for the state-dependent variation of the queueing system.

```

>>> import ambulance_game as abg
>>> import ciw
>>> import numpy as np
>>>
>>> lambda_1 = 1
>>> lambda_2 = 1
>>> num_of_servers = 1
>>> threshold = 1
>>> system_capacity = 2
>>> buffer_capacity = 1
>>> runtime = 1000
>>> seed_num = 0
>>>
>>> Q = abg.simulation.simulate_model(
...     lambda_1=lambda_1,
...     lambda_2=lambda_2,
...     mu={(0, 0): 2, (0, 1): 3, (0, 2): 4, (1, 1): 4, (1, 2): 5},
...     num_of_servers=num_of_servers,
...     threshold=threshold,
...     seed_num=seed_num,
...     system_capacity=system_capacity,
...     buffer_capacity=buffer_capacity,
...     runtime=runtime,
... )
>>> mean_waiting_time = np.mean([w.waiting_time for w in Q.get_all_records()])
>>> np.round(mean_waiting_time, 8)
0.06028274

```

Code snippet 6.2: Example of the state-dependent variation of the queueing system.

## 6.3 Server-dependent variation

This section describes the creation of the server-dependent variation of the queueing system described in Section 3. The server-dependent model has a similar structure as the state-dependent model described in Section 6.2. In this variation each server has its own service rate.

Similar to Section 6.2 the server-dependent variation of the parametric service rate  $\mu = \mu_i$  for a given  $i \in 1, 2, \dots, C$  (where  $C$  is the number of servers) is defined as follows:

$$\mu = \begin{cases} \mu_1, & \text{for server 1} \\ \mu_2, & \text{for server 2} \\ \vdots & \vdots \\ \mu_C, & \text{for server } C \end{cases} \quad (6.2)$$

Consider an example of the queueing system described in Section 3 where the number of servers is set to  $C = 4$ , the threshold is set to  $T = 1$ , node 1 capacity is  $N = 2$  and node 2 capacity is  $M = 1$ . The value of the service rate  $\mu$  for the DES model can take the following values:

$$\mu = \begin{cases} \mu_1, & \text{for server 1} \\ \mu_2, & \text{for server 2} \\ \mu_3, & \text{for server 3} \\ \mu_4, & \text{for server 4} \end{cases}$$

### 6.3.1 Implementation

The server-dependent variation of the queueing system is implemented in a similar way as the state-dependent implementation. Using the python library `ciw` the server-dependent service time distribution is defined as a class that inherits from the `Distribution` class shown in 6.3.

```
>>> import random
>>> import ciw
>>> class ServerDependentExponential(
...     ciw.dists.Distribution
... ):
...     def __init__(self, rates):
...         self.rates = rates
...
...     def sample(self, t=None, ind=None):
```



```

...         """
...         This method is used to sample the service time for an individual
...         based
...         on the server that the individual is assigned to
...         """
...         server = ind.server.id_number
...         rate = self.rates[server]
...         return random.expovariate(rate)

```

Code snippet 6.3: Server-dependent service time distribution class.

Similar to the state-dependent implementation, the function `simulate_model` from Section 3.2 is used to simulate the server-dependent variation of the queueing system. The main difference from the state-dependent case is that the service rate  $\mu$  is now a dictionary with the server's id as the key and the service rate of that server as the value.

```

>>> import ambulance_game as abg
>>> import ciw
>>> import numpy as np
>>>
>>> lambda_1 = 1
>>> lambda_2 = 1
>>> num_of_servers = 4
>>> threshold = 1
>>> system_capacity = 8
>>> buffer_capacity = 2
>>> runtime = 1000
>>> seed_num = 0
>>>
>>> Q = abg.simulation.simulate_model(
...     lambda_1=lambda_1,
...     lambda_2=lambda_2,
...     mu={
...         1: 0.5,
...         2: 0.5,
...         3: 1.0,
...         4: 1.5,
...     },
...     num_of_servers=num_of_servers,
...     threshold=threshold,
...     seed_num=seed_num,
...     system_capacity=system_capacity,
...     buffer_capacity=buffer_capacity,
...     runtime=runtime,
... )
>>> mean_waiting_time = np.mean([w.waiting_time for w in Q.get_all_records()])
>>> np.round(mean_waiting_time, 8)
0.02668017

```

Code snippet 6.4: Example of the server-dependent variation of the queueing system.

## 6.4 State and server-dependent model

In addition, the final variation of the queueing model is one that is both state and server-dependent. That is, for each server and for each state there is a different service rate. In other words, each server can have a different service rate for every possible scenario that the system can be in.

The new service rate  $\mu$  that will be used in this scenario is defined as a combination of equations (6.1) and (6.2) where:

$$\mu = \begin{cases} \mu_{1,(0,0)}, & \text{for server 1 if } (u, v) = (0, 0) \\ \mu_{1,(0,1)}, & \text{for server 1 if } (u, v) = (0, 1) \\ \vdots & \vdots \\ \mu_{1,(M,N)}, & \text{for server 1 if } (u, v) = (M, N) \\ \mu_{2,(0,0)}, & \text{for server 2 if } (u, v) = (0, 0) \\ \mu_{2,(0,1)}, & \text{for server 1 if } (u, v) = (0, 1) \\ \vdots & \vdots \\ \mu_{2,(M,N)}, & \text{for server 2 if } (u, v) = (M, N) \\ \vdots & \vdots \\ \mu_{C,(0,0)}, & \text{for server } C \text{ if } (u, v) = (0, 0) \\ \mu_{C,(0,1)}, & \text{for server } C \text{ if } (u, v) = (0, 1) \\ \vdots & \vdots \\ \mu_{C,(M,N)}, & \text{for server } C \text{ if } (u, v) = (M, N) \end{cases} \quad (6.3)$$

Consider an example where the number of servers is set to  $C = 2$ , the threshold is set to  $T = 1$ , node 1 capacity is  $N = 3$  and node 2 capacity is  $M = 1$ . For this particular example the possible values that  $\mu$  can take shown by equation (6.4).

$$\mu = \left\{ \begin{array}{ll} \mu_{1,(0,0)}, & \text{for server 1 if } (u, v) = (0, 0) \\ \mu_{1,(0,1)}, & \text{for server 1 if } (u, v) = (0, 1) \\ \mu_{1,(0,2)}, & \text{for server 1 if } (u, v) = (0, 2) \\ \mu_{1,(0,3)}, & \text{for server 1 if } (u, v) = (0, 3) \\ \mu_{1,(1,0)}, & \text{for server 1 if } (u, v) = (1, 0) \\ \mu_{1,(1,1)}, & \text{for server 1 if } (u, v) = (1, 1) \\ \mu_{1,(1,2)}, & \text{for server 1 if } (u, v) = (1, 2) \\ \mu_{1,(1,3)}, & \text{for server 1 if } (u, v) = (1, 3) \\ \mu_{2,(0,0)}, & \text{for server 2 if } (u, v) = (0, 0) \\ \mu_{2,(0,1)}, & \text{for server 2 if } (u, v) = (0, 1) \\ \mu_{2,(0,2)}, & \text{for server 2 if } (u, v) = (0, 2) \\ \mu_{2,(0,3)}, & \text{for server 2 if } (u, v) = (0, 3) \\ \mu_{2,(1,0)}, & \text{for server 2 if } (u, v) = (1, 0) \\ \mu_{2,(1,1)}, & \text{for server 2 if } (u, v) = (1, 1) \\ \mu_{2,(1,2)}, & \text{for server 2 if } (u, v) = (1, 2) \\ \mu_{2,(1,3)}, & \text{for server 2 if } (u, v) = (1, 3) \end{array} \right. \quad (6.4)$$

### 6.4.1 Implementation

The implementation of the state and server-dependent model is the combination of the state-dependent and server-dependent models' implementation. Once again the implementation is done using the python library `ciw` library. The distribution for the state and server-dependent model is defined as a class that inherits from `ciw`'s class `Distribution` from the `dists` module. Note that as opposed to the classes defined earlier an additional method `id` defined in this class. Method `update_server_attributes` adds additional attributes to the server objects to allow for further analysis of each server's performance.

```
>>> import random
>>> import ciw
>>> class StateServerDependentExponential(
...     ciw.dists.Distribution
... ):
...     def __init__(self, rates):
...         self.simulation = None
...         self.rates = rates
...
...     def sample(self, t=None, ind=None):
...         """
...         This method is used to sample the service time for an individual
...         based on the current state and the server that the individual is
```

```

...         assigned to. The following steps are being taken:
...             1. Find the server
...             2. Find the state
...             3. Check if the state is valid. Note that there are some cases
...                 where the visited state is not valid. These are the cases
...                 where the state '(u, T-1)' is visited where 'u > 0'. This
...                 is meant to be an unreachable state. In such case remap
...                 the state to '(u+1, T)'
...             4. Get the service rate for that server and state
...             5. Sample the service time
...             6. Update any possible attributes for the server
...         """
...         server = ind.server.id_number
...         state = (
...             len(ind.simulation.nodes[1].individuals[0]),
...             len(ind.simulation.nodes[2].individuals[0]),
...         )
...         is_invalid = state[0] > 0 and state[1] < ind.simulation.threshold
...         if is_invalid:
...             state = (state[0] - 1, state[1] + 1)
...         rate = self.rates[server][state]
...         service_time = random.expovariate(rate)
...         self.update_server_attributes(ind, service_time)
...         return service_time
...
...     def update_server_attributes(self, ind, service_time):
...         """
...         Updates the server's attributes
...         """
...         if hasattr(ind.server, "served_inds"):
...             ind.server.served_inds.append(self.simulation.current_time)
...         else:
...             ind.server.served_inds = [self.simulation.current_time]
...
...         if hasattr(ind.server, "service_times"):
...             ind.server.service_times.append(service_time)
...         else:
...             ind.server.service_times = [service_time]

```

Code snippet 6.5: Example of the state and server-dependent variation of the queueing system

Now consider an example where the number of servers is set to  $C = 2$ , the threshold is set to  $T = 1$ , node 1 capacity is  $N = 3$  and node 2 capacity is  $M = 1$ . Let the service rates be defined in such a way where:

1. Server 1's service rate is 0.5 whenever there are less than two individuals in the entire system, otherwise the service rate is 1.
2. Server 2's service rate is 0.7 whenever there are less than three individuals in the entire system, otherwise the service rate is 1.5.

The following code snippet shows how to define the model and run the simulation for 1000 time units.

```

>>> import ambulance_game as abg
>>> import numpy as np
>>>
>>> lambda_1 = 1
>>> lambda_2 = 1
>>> num_of_servers = 2
>>> threshold = 1
>>> system_capacity = 3
>>> buffer_capacity = 1
>>> runtime = 1000
>>> seed_num = 0
>>>
>>> all_states = abg.markov.build_states(
...     threshold=threshold,
...     system_capacity=system_capacity,
...     buffer_capacity=buffer_capacity
... )
>>> mu = {k: {} for k in range(1, num_of_servers + 1)}
>>> mu[1] = {(u, v): 0.5 if u + v < 2 else 1 for u, v in all_states}
>>> mu[2] = {(u, v): 0.7 if u + v < 3 else 1.5 for u, v in all_states}
>>>
>>> Q = abg.simulation.simulate_model(
...     lambda_1=lambda_1,
...     lambda_2=lambda_2,
...     mu=mu,
...     num_of_servers=num_of_servers,
...     threshold=threshold,
...     seed_num=seed_num,
...     system_capacity=system_capacity,
...     buffer_capacity=buffer_capacity,
...     runtime=runtime,
... )
>>> for srv, mean_service_time in enumerate([
...     np.round(np.mean(s.service_times), 8)
...     for s in Q.nodes[2].servers
... ]):
...     print(f"Server_{srv+1}:", mean_service_time)
Server 1: 1.79718861
Server 2: 0.87253758

```

Code snippet 6.6: Example code of the state and server-dependent variation of the queueing system

Note that, in the implementation shown in code snippet 6.6 the individuals are paired with a server in an “unfair” way since the default behaviour of `ciw` does not focus on the fairness of server allocation. Server 1 is always assigned an individual if they are free, while Server 2 is only assigned an individual if Server 1 is busy. For the purposes of this project though, it is important to have a more “fair” allocation of individuals to servers. This can be done by using the `server_priority_function` argument of the `simulate_model` function.

```

>>> Q = abg.simulation.simulate_model(
...     lambda_1=lambda_1,
...     lambda_2=lambda_2,

```

```

...     mu=mu,
...     num_of_servers=num_of_servers,
...     threshold=threshold,
...     seed_num=seed_num,
...     system_capacity=system_capacity,
...     buffer_capacity=buffer_capacity,
...     runtime=runtime,
...     server_priority_function=lambda srv, ind: random.random()
... )
>>> for srv, mean_service_time in enumerate([
...     np.round(np.mean(s.service_times), 8)
...     for s in Q.nodes[2].servers
... ]):
...     print(f"Server_{srv+1}:", mean_service_time)
Server 1: 1.31071177
Server 2: 1.02142722

```

Code snippet 6.7: Example of using fair allocation of individuals to servers

### 6.4.2 Game theoretic model

The state and server dependent model can be used in the game theoretic model described from Section 4. However, since the state and server dependent model variant of the queueing theoretic model is only implemented using Discrete Event Simulation (DES), it is not possible to use the Markov chain model here. Thus, the implementation of the game was modified so that it can be used with the DES model as well. More details on the code can be found in Appendix A. At its core, the only metric that is needed from the Markov model are the performance measures of the queue (blocking time and proportion of individuals within target) which can also be obtained using the DES model. Consequently, the payoff matrices can be calculated using these performance measures.

Something that needs to be taken into consideration here is whether the results of the game are the same when using the DES model and the Markov chain model. In fact using the DES approach, several other parameters are introduced that can affect the results of the game. Such parameters are the runtime, the warm up time and the cooldown time of the simulation. The runtime of the simulation is the total time that the simulation is run for, the warm up time is the time that the simulation is run for before the data collection starts and the cooldown time is the time that the simulation is run for after the data collection has finished. The runtime of the simulation is the parameter that is most likely to affect the performance measures of the queue, since the longer the simulation is run for, the better the estimates of the performance measures will be.

Consider a game with the parameters shown in Table 6.1.

Table 6.1: Parameter values for game theoretic model example to observe the differences of running the game with DES and Markov chains.

Distributor				Queueing system A					Queueing system B				
$\lambda_2$	t	$\hat{P}$	$\alpha$	$\lambda_1^A$	$\mu^A$	$C^A$	$N^A$	$M^A$	$\lambda_1^B$	$\mu^B$	$C^B$	$N^B$	$M^B$
2	2	0.95	0.2	2	2	2	6	2	2	2	2	6	2

For a complete description of parameter notations see Section 4.3.1. Figure 6.2 shows the asymmetric replicator dynamics run of the game when the payoff matrices are calculated using the Markov chain model.

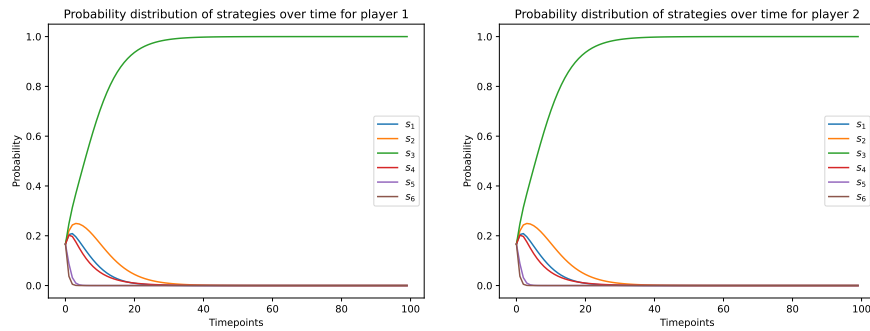


Figure 6.2: Asymmetric replicator dynamics algorithm run on the game obtained from the Markov chain model.

It can be seen that both player end up playing strategy  $s_3$  which corresponds to choosing a threshold of  $T^{(A)} = 3$  and  $T^{(B)} = 3$ . Figures 6.3, 6.4 and 6.5 show the asymmetric replicator dynamics run of the game when the payoff matrices are calculated using the DES model with different runtimes. More specifically, the runtime of the DES model is set to 300, 500 and 1000 time units respectively.

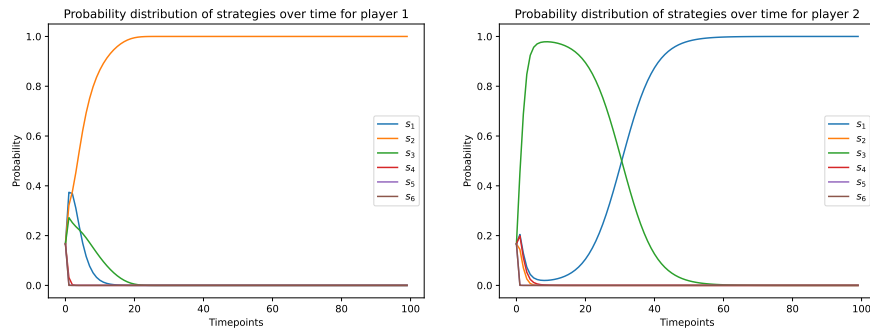


Figure 6.3: Asymmetric replicator dynamics algorithm run on the game obtained from the DES model using a runtime of 300 time units.

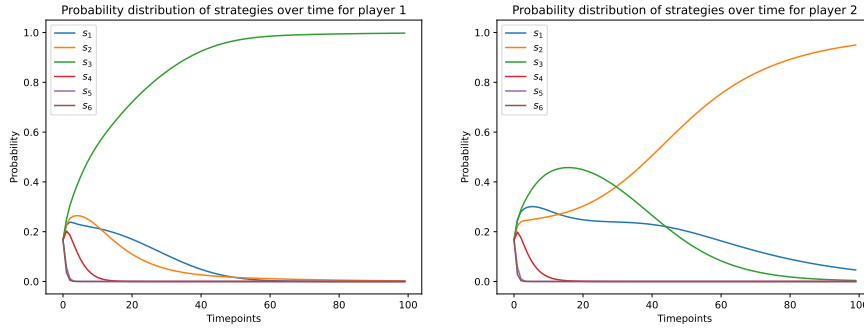


Figure 6.4: Asymmetric replicator dynamics algorithm run on the game obtained from the DES model using a runtime of 500 time units.

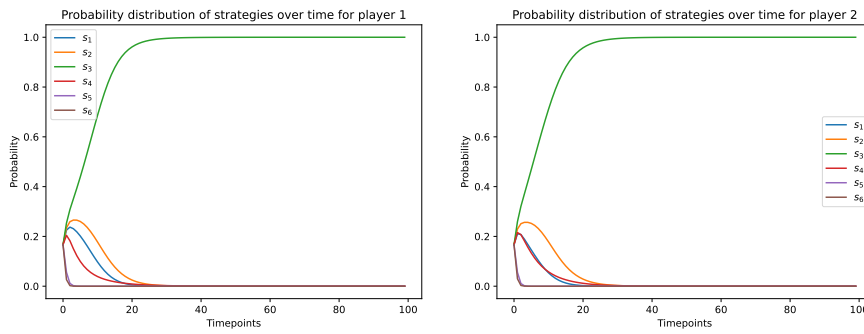


Figure 6.5: Asymmetric replicator dynamics algorithm run on the game obtained from the DES model using a runtime of 1000 time units.

By observing the asymmetric replicator dynamics run of the game with the DES model when using different runtimes, it can be noticed that the results from using a runtime of 300 time units and a runtime of 500 time units are different from the ones obtained using the Markov chain model. However, the resultant strategies from using a runtime of 1000 time units are the same as the ones obtained using the Markov chain model. This is due to the fact that the runtime of the DES model needs to be long enough for the simulation to reach a steady state. For this particular example, it seems that a runtime of 1000 time units is sufficient.

Having found a reasonable runtime for the DES model, the state and server dependent distribution of the service rate can be used in the game theoretic model. Consider a small change to the parameter values defined earlier, where the service rate is now state and server dependent. Let the service rate for queueing system 1 be  $\mu^{(1)} = 6$  only for server 1 if there are 4 or more individuals in the system and  $\mu^{(1)} = 2$  otherwise. Additionally, let the service rate for queueing system 2 be  $\mu^{(2)} = 8$  only for server 1 if there are 3 or more individuals in node 1 and  $\mu^{(2)} = 2$  otherwise. In other words the service rate is now defined as:



$$\mu^{(1)} = \begin{cases} 6, & \text{if server id} = 1 \text{ and } u_1 + v_1 \geq 4 \\ 2, & \text{otherwise} \end{cases} \quad (6.5)$$

$$\mu^{(2)} = \begin{cases} 8, & \text{if server id} = 1 \text{ and } u_1 \geq 3 \\ 2, & \text{otherwise} \end{cases} \quad (6.6)$$

The game theoretic model is then run using the DES model with a runtime of 3000 time units. Figure 6.6 shows the asymmetric replicator dynamics run of the game when the payoff matrices are calculated using the DES model with this new service rate distribution.

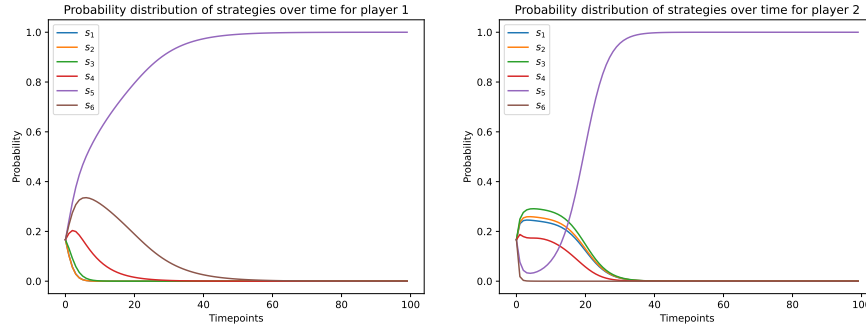


Figure 6.6: Asymmetric replicator dynamics algorithm run on the game obtained from the DES model using a runtime of 3000 time units and a state and server dependent service rate.

It can be seen that the outcome of the game is different than before. Player 1 chooses to play strategy  $s_5$  and player 2 also ends up playing strategy  $s_5$ . That corresponds to choosing a threshold of  $T^{(1)} = 5$  and a threshold of  $T^{(2)} = 5$ . The change in the service distribution made server 1 of queueing system 1 and server 1 of queueing system 2 work faster when their corresponding system was getting busier. This change in the service rate distribution made the queueing systems increase their thresholds in the game, and thus block less type 2 individuals at node 1. Refer to Figure 3.1 for a visual representation of the types of individuals and the nodes of the queueing systems.

## 6.5 The agent-based model

Section 6.4 describes an extension to the queueing system described in Section 3 that allows each server to have their own service rate for every possible state of the system. In this section, an agent-based model is proposed where servers

are considered as agents that can make their own decisions. The idea is that every agent in the system can choose their own service rate based on some utility function that they aim to maximise. This would mean that the servers would be able to choose and update their own speed at which they serve individuals while the system is running. Such decisions could be based on a number of factors, such as minimising the number of patients in the system, minimising the proportion of patients lost to the system, maximising their own idle time and so on.

### 6.5.1 Utility functions

Utility functions are a way of quantifying how “happy” an agent is with the current condition of the system [48, 49]. Each agent in the system can have their own utility function that they aim to maximise. In a realistic scenario, these utility functions could be based in a number of factors, where each agent can have a different weight for each factor.

At the end of the simulation run there are some key performance measures that can be extracted to quantify the performance of the overall system and each server individually. Table 6.2 shows these measures that will then be used to formulate the utility functions.

Table 6.2: Performance measures that could affect each agent’s utility

Notation	Description
$I$	Total number of individuals (both served and lost)
$I_s$	Number of served individuals
$I_L$	Number of individuals that are lost due to the system being full
$I_s^{(k)}$	All individuals served by server $k$
$R$	Overall runtime of the simulation
$B^{(k)}$	Busy time of server $k$
$R - B^{(k)}$	Idle time of server $k$
$\bar{\mu}^{(k)}$	Mean service rate of server $k$
$\bar{m}^{(k)}$	Mean service time of server $k$

Note that the difference between the mean service rate and the mean service time is that the mean service rate is the average out of all the service rates for every state  $(u, v)$  with no particular weight given to any of them. In contrast, the mean service time is the average of all the service times that each server has experienced. That means that if a particular state has not been visited by a server, then the mean service time will not be affected by that state.

The above measures could be combined together in a number of ways to formulate utility functions. Some examples of utility functions that could be used are the

following:

$$U_k^{(1)} = e I_s^{(k)} + (1 - e) (R - B^{(k)}) \quad (6.7)$$

$$U_k^{(2)} = e \frac{I_s^{(k)}}{I_s} + (1 - e) \frac{R - B^{(k)}}{R} \quad (6.8)$$

$$U_k^{(3)} = e \bar{m}^{(k)} + (1 - e) (R - B^{(k)}) \quad (6.9)$$

$$U_k^{(4)} = e \bar{\mu}^{(k)} + (1 - e) (R - B^{(k)}) \quad (6.10)$$

$$U_k^{(5)} = e \frac{I_s^{(k)}}{I_s} + (1 - e) \bar{m}^{(k)} \quad (6.11)$$

$$U_k^{(6)} = e \frac{I_s^{(k)}}{I} + (1 - e) \frac{1}{\bar{m}^{(k)}} \quad (6.12)$$

$$U_k^{(7)} = e \frac{I_s}{I} + (1 - e) \frac{R - B^{(k)}}{R} \quad (6.13)$$

$$(6.14)$$

where  $U_k^{(i)}$  is utility  $i$  of server  $k$  and  $e$  is a parameter that can be used to weight the importance of each measure. For example,  $e = 0.5$  would mean that the two measures are equally important for the agent. Table 6.3 gives a brief description of each utility function.

Table 6.3: Utility functions that can be used to measure each server's happiness

Utility function	Description
$U_k^{(1)}$	Weighted average between the number of served individuals by server $k$ and idle time of server $k$
$U_k^{(2)}$	Weighted average between overall proportion of served individuals by server $k$ and proportion of idle time of server $k$
$U_k^{(3)}$	Weighted average between mean service time of server $k$ and idle time of server $k$
$U_k^{(4)}$	Weighted average between mean service rate of server $k$ and idle time of server $k$
$U_k^{(5)}$	Weighted average between proportion of served individuals by server $k$ and mean service time of server $k$
$U_k^{(6)}$	Weighted average between proportion of served individuals by server $k$ and inverse of mean service time of server $k$
$U_k^{(7)}$	Weighted average between overall proportion of served individuals and proportion of idle time of server $k$

### 6.5.2 Case study

In this subsection, an empirical study is presented to show how the above utility functions can be used to measure each server's "happiness". The study builds upon the empirical study on the queueing system described in [64]. This subsection will use the same data set as in [64], but applied to the queueing system described in Section 3 and the agent-based model described in Section 6.5.

In that study a set of data was collected from a large emergency department in Wales, UK. The data was collected over 6 months and contained information on each patient that arrived at the emergency department. Such information included the time of arrival, the time of service, the triage category and the time of discharge. The dataset consisted of 4,832 patients that were considered as "urgent". In [64] it was shown that as the workload of the system changed, the service times of the serves also changed. In fact, low to moderate workload levels resulted in service times that were lower than the service times experienced at high workload levels.

The system was modelled as a single-server queueing system with two service speeds. Arrivals follow a Poisson distribution with mean inter-arrival time of 92 minutes, thus the arrival rate can be set to  $\lambda = \frac{1}{92}$ . The service speed was partitioned into two distributions; one for low to moderate workload levels and one for high workload levels. The service speed for the low to moderate workload levels was found to follow a lognormal distribution with a mean of 86 minutes and the service speed for the high workload levels was found to follow a lognormal distribution with a mean of 62 minutes. Therefore, the service rates can be set to  $\mu_1 = \frac{1}{86}$  and  $\mu_2 = \frac{1}{62}$ . It was observed that the slow service speed was used for when 6 or less individuals were in the system and the fast service speed was used for when 7 or more individuals were present.

The above parameters were slightly modified so that they are applied to the queueing system described in Section 3. In essence, the queueing system is now a 4 server system and the arrival rate is set to  $\lambda = 4 \times \frac{1}{92}$  and the two service speeds stay the same. In addition, the new arrival rate is now split into two distributions, one for type 1 individuals and one for type 2 individuals. Type 1 arrivals follow a Poisson distribution with mean inter-arrival time of 57.5 minutes and type 2 arrivals follow a Poisson distribution with mean inter-arrival time of 38.3 minutes. Therefore, the arrival rates for type 1 and type 2 individuals are  $\lambda_1 = \frac{1}{57.5}$  and  $\lambda_2 = \frac{1}{38.3}$  respectively.

Additionally, for this modified example the 4 servers fall into one of three groups;

experienced, moderate and intern. In particular server 1 is an experienced server, server 2 and server 3 are moderate servers and server 4 is an intern. This means that server 1 has a slightly higher service rate than the other servers and if they are available they will always be assigned the incoming individual. Servers 2 and 3 have the same service rate and if they are available they may be assigned the incoming individual with an equal probability. Finally, server 4 has the lowest service rate and they will only be assigned the incoming individual if the other servers are unavailable. More specifically, the service rates for the “experienced” server is multiplied by a factor of 1.2 and the service rates for the “intern” server is multiplied by a factor of 0.8, while the “moderate” servers stay unchanged. Whilst this might not be a realistic example, it is used here to demonstrate the utilities in this agent-based model.

Because of the way the queueing system described in Section 3 is modelled, there are some additional parameters that need to be considered. These are the capacity of Node 1, the capacity of Node 2, and the threshold. The capacity of Node 1 is set to  $N = 35$ , the capacity of Node 2 is set to  $M = 20$  and the threshold is set to  $T = 10$ . Therefore the complete set of parameters for the queueing system are given by equation (6.15) and Table 6.4.

$$\begin{aligned}\mu^{(1)} &= \begin{cases} 1.2 \times \frac{1}{86}, & \text{if } u + v < 7 \\ 1.2 \times \frac{1}{62}, & \text{if } u + v \geq 7 \end{cases} \\ \mu^{(i)} &= \begin{cases} \frac{1}{86}, & \text{if } u + v < 7 \\ \frac{1}{62}, & \text{if } u + v \geq 7 \end{cases} \quad \text{for } i \in \{2, 3\} \\ \mu^{(4)} &= \begin{cases} 0.8 \times \frac{1}{86}, & \text{if } u + v < 7 \\ 0.8 \times \frac{1}{62}, & \text{if } u + v \geq 7 \end{cases}\end{aligned} \quad (6.15)$$

Table 6.4: Parameter values used in the case study.

$\lambda_1$	$\lambda_2$	$C$	$T$	$N$	$M$
$\frac{1}{57.5}$	$\frac{1}{38.3}$	4	10	35	20

Running the simulation for 100,000 time units each server’s utilisation (i.e. what percentage of time they were busy), the proportion of individuals that they served and their mean service time were recorded.

Table 6.5: Each server's performance measure for a run of the simulation.

Server	Server utilisation	Proportion of individuals served	Mean service time
1	87.24	33.07	62.44
2	82.51	24.54	79.56
3	83.55	24.26	81.51
4	78.45	18.13	102.3

These performance measures can now be used to populate the different utility functions described in equations (6.7) - (6.13). All utility functions are a weighted average between two performance measures that are chosen to be the important factors for a server. The weigh parameter  $e$  can take on any value between 0 and 1. Tables 6.6 - 6.12 show the utility functions for each server for different values of  $e$ .

Table 6.6: Utility function 1 ( $U_k^{(1)}$ ) for the 4 servers and different values of  $e$ 

Server	$e = 0$	$e = 0.1$	$e = 0.2$	$e = 0.3$	$e = 0.4$	$e = 0.5$	$e = 0.6$	$e = 0.7$	$e = 0.8$	$e = 0.9$	$e = 1$
1	17703	16101	14500	12899	11298	9697	8096	6494	4893	3292	1691
2	20729	18754	16780	14805	12830	10855	8880	6905	4931	2956	981
3	21270	19247	17224	15200	13177	11154	9131	7108	5084	3061	1038
4	25599	23111	20624	18136	15648	13161	10673	8186	5698	3211	723

Table 6.7: Utility function 2 ( $U_k^{(2)}$ ) for the 4 servers and different values of  $e$ 

Server	$e = 0$	$e = 0.1$	$e = 0.2$	$e = 0.3$	$e = 0.4$	$e = 0.5$	$e = 0.6$	$e = 0.7$	$e = 0.8$	$e = 0.9$	$e = 1$
1	0.128	0.148	0.168	0.189	0.209	0.229	0.25	0.27	0.29	0.311	0.331
2	0.175	0.182	0.189	0.196	0.203	0.21	0.217	0.224	0.232	0.239	0.246
3	0.165	0.172	0.18	0.188	0.196	0.204	0.212	0.219	0.227	0.235	0.243
4	0.216	0.212	0.209	0.205	0.202	0.199	0.195	0.192	0.188	0.185	0.182

Table 6.8: Utility function 3 ( $U_k^{(3)}$ ) for the 4 servers and different values of  $e$ 

Server	$e = 0$	$e = 0.1$	$e = 0.2$	$e = 0.3$	$e = 0.4$	$e = 0.5$	$e = 0.6$	$e = 0.7$	$e = 0.8$	$e = 0.9$	$e = 1$
1	0.1	6.4	12.6	18.8	25.1	31.3	37.5	43.7	50.0	56.2	62.4
2	0.2	8.1	16.1	24.0	31.9	39.9	47.8	55.7	63.7	71.6	79.6
3	0.2	8.3	16.4	24.6	32.7	40.8	49.0	57.1	65.2	73.4	81.5
4	0.2	10.4	20.6	30.8	41.0	51.3	61.5	71.7	81.9	92.1	102.3

Table 6.9: Utility function 4 ( $U_k^{(4)}$ ) for the 4 servers and different values of  $e$ 

Server	$e = 0$	$e = 0.1$	$e = 0.2$	$e = 0.3$	$e = 0.4$	$e = 0.5$	$e = 0.6$	$e = 0.7$	$e = 0.8$	$e = 0.9$	$e = 1$
1	0.128	0.117	0.105	0.094	0.083	0.072	0.061	0.05	0.038	0.027	0.016
2	0.175	0.159	0.142	0.126	0.11	0.094	0.078	0.061	0.045	0.029	0.013
3	0.165	0.149	0.134	0.119	0.104	0.088	0.073	0.058	0.043	0.027	0.012
4	0.216	0.195	0.174	0.154	0.133	0.113	0.092	0.072	0.051	0.03	0.01

Table 6.10: Utility function 5 ( $U_k^{(5)}$ ) for the 4 servers and different values of  $e$ 

Server	$e = 0$	$e = 0.1$	$e = 0.2$	$e = 0.3$	$e = 0.4$	$e = 0.5$	$e = 0.6$	$e = 0.7$	$e = 0.8$	$e = 0.9$	$e = 1$
1	62.44	56.23	50.02	43.81	37.6	31.39	25.18	18.96	12.75	6.54	0.33
2	79.56	71.62	63.69	55.76	47.83	39.9	31.97	24.04	16.11	8.18	0.25
3	81.51	73.38	65.25	57.13	49.0	40.87	32.75	24.62	16.5	8.37	0.24
4	102.3	92.08	81.87	71.66	61.45	51.24	41.03	30.82	20.6	10.39	0.18

Table 6.11: Utility function 6 ( $U_k^{(6)}$ ) for the 4 servers and different values of  $e$ 

Server	$e = 0$	$e = 0.1$	$e = 0.2$	$e = 0.3$	$e = 0.4$	$e = 0.5$	$e = 0.6$	$e = 0.7$	$e = 0.8$	$e = 0.9$	$e = 1$
1	0.02	0.05	0.08	0.11	0.14	0.17	0.2	0.24	0.27	0.3	0.33
2	0.01	0.04	0.06	0.08	0.11	0.13	0.15	0.18	0.2	0.22	0.25
3	0.01	0.04	0.06	0.08	0.1	0.13	0.15	0.17	0.2	0.22	0.24
4	0.01	0.03	0.04	0.06	0.08	0.1	0.11	0.13	0.15	0.16	0.18

Table 6.12: Utility function 7 ( $U_k^{(7)}$ ) for the 4 servers and different values of  $e$ 

Server	$e = 0$	$e = 0.1$	$e = 0.2$	$e = 0.3$	$e = 0.4$	$e = 0.5$	$e = 0.6$	$e = 0.7$	$e = 0.8$	$e = 0.9$	$e = 1$
1	0.128	0.215	0.302	0.389	0.476	0.563	0.65	0.737	0.824	0.911	0.999
2	0.175	0.257	0.34	0.422	0.504	0.587	0.669	0.751	0.834	0.916	0.999
3	0.165	0.248	0.331	0.415	0.498	0.582	0.665	0.748	0.832	0.915	0.999
4	0.216	0.294	0.372	0.45	0.529	0.607	0.685	0.764	0.842	0.92	0.999

Tables 6.5 - 6.12 show the range of values that the utility functions can take for the different servers and different values of  $e$ . Throughout the rest of this section, these utility functions will be used as the key performance indicators that the servers will use to make decisions about their own service speed.

## 6.6 Reinforcement learning algorithm

The agent-based mode described in Section 6.5 describes how the agents can interact with the environment. The next step is to describe how the agents can learn from their interactions. A reinforcement learning algorithm is described in this section where the agents update their service rates based on the utilities they receive from the queueing system [10, 71, 92]. The concepts described in Section 3 and Section 6.5 are incorporated in the reinforcement learning algorithm so that the agents decide how fast they should serve the customers in order to maximise their own utility.

The reinforcement learning algorithm is a policy iteration algorithm [93] where the agents update their service rates based on the utilities they receive from the queueing system. The service rates will be referred to as the policy in this section.

A policy is a set of service rates for every server for each possible state they can be in. The general form of a policy is given by equation (6.3).

The following pseudo-code describes the reinforcement learning algorithm. At each iteration, the agent receives a utility  $U$  from the queueing system and updates its service rate  $s$  based on the utility it received. A policy update is considered a change in the service rate of a server  $k$  at a state  $(u, v)$  that was visited in the last simulation run (i.e. increasing or decreasing the value of  $\mu_{k,(u,v)}$  for some  $k$  and  $(u, v)$ ). Recall from Section 6.5 that each agent (server) has a different service rate for each possible state they can be in.

```

Choose an initial policy
Run simulation with current policy
Calculate initial utility U_current
For each iteration
    1. Choose a server k
    2. Choose a state (u, v)
    3. Update policy for server k and state (u, v)
    4. Rerun simulation
    5. Calculate utility U_new
    6. If utility U is higher than previous utility
        - Accept policy s
        - Update U_current to U_new

```

The algorithm has been implemented in python and the scripts are archived and can be found in [113].

The total number of iterations is a parameter that needs to be set to a high enough value to ensure that the agents have enough time to learn from their interactions with the queueing system. Additionally, in order to eliminate any stochasticity from the simulation at each iteration, the simulation is run for a fixed runtime and a fixed number of trials. Furthermore, when choosing a new policy for a server and a state, the algorithm will choose a new policy that is within a certain range of the current policy and that cannot be below zero.

An additional rule is that there is a maximum service rate that a server can have for any given state. This is to ensure that unrealistic service rates are not chosen. Some numeric results on why this rule is necessary are presented in Section 6.6.1.4.

Consider an example of a queueing system with 2 servers and the following set of states:



$$\mathcal{S} = \{(0, 0), (0, 1), (0, 2), (1, 2), (0, 3), (1, 3)\} \quad (6.16)$$

The 2 servers have a different service rate for each state  $(u, v) \in \mathcal{S}$ . An initial policy is chosen where the service rates for the 2 servers is set to:

$$\mu^{(1)} = \begin{cases} 1 & \text{if } v < 2 \\ 1.5 & \text{otherwise} \end{cases} \quad \mu^{(2)} = \begin{cases} 0.8 & \text{if } v < 3 \\ 2 & \text{otherwise} \end{cases}$$

Figures 6.7 - 6.9 show the policy for the 2 servers for the first 3 iterations of the reinforcement learning algorithm.

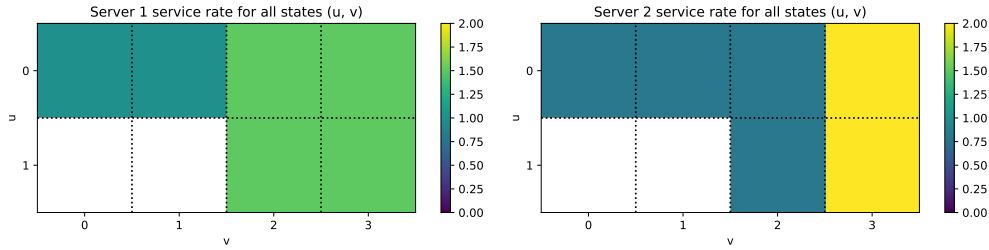


Figure 6.7: Example policy for server 1 and server 2 at iteration 1

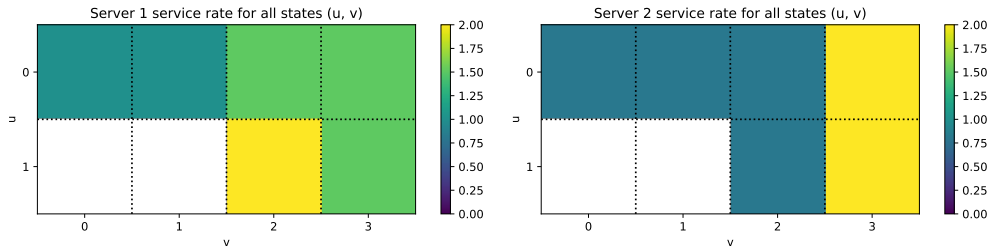


Figure 6.8: Example policy for server 1 and server 2 at iteration 2

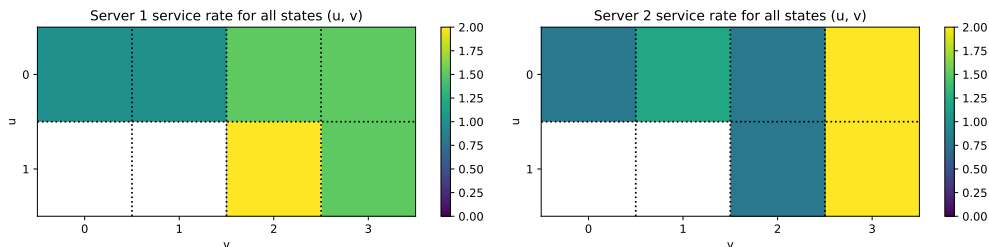


Figure 6.9: Example policy for server 1 and server 2 at iteration 3

### 6.6.1 Numeric results

The reinforcement learning algorithm is implemented in python and the scripts along with the generated data presented in this section have been archived and can be found in [113]. Additional numerical experiments can also be found in Appendix C.

Consider a queueing system with the parameters presented in Table 6.13.

Table 6.13: Parameter values for the reinforcement learning algorithm experiments.

$\lambda_1$	$\lambda_2$	$\mu$	$C$	$T$	$N$	$M$
0.5	1	0.7	4	7	10	7

Note that for the initial policy, the service rates of the 4 servers are set to  $\mu = 0.7$  for all servers and all states. In addition, the 4 servers are set to be of the same expertise level as described in Section 6.5.2. That is, server 1 is an experienced server, server 2 and server 3 are moderate servers and server 4 is an intern. That means that when server 1 is not busy they will always receive the incoming individual, otherwise either server 2 or server 3 will receive that individual with an equal probability. Finally, server 4 may serve an individual only when every other server is busy.

The remaining of this section focuses on the results of the reinforcement learning algorithm from using utility functions  $U_k^{(3)}$  and  $U_k^{(7)}$  with different values of  $e$ . The two utility functions are chosen because they were considered to be the most realistic ones. In addition utility function  $U_k^{(7)}$  is used to express the effect of having to balance the server's best interest with the system's best interest. It is assumed that a server would like the proportion of individuals served to be as high as possible, while also maximising their own idle time. This is what utility function  $U_k^{(7)}$  aims to capture.

#### 6.6.1.1 Utility function 3 ( $U_k^{(3)}$ ) with $e = 0.1$

Consider utility function 3 defined in (6.9) where  $U_k^{(3)} = e \bar{m}^{(k)} + (1-e)(R - B^{(k)})$ . Thus a server's utility corresponds to the weighted average of the server's service time and their idle time. This means that in order for servers to increase their utility they either need to work faster or increase the amount of time they are idle. Figure 6.10 shows the utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(3)}$  with  $e = 0.1$  and 100,000 iterations.

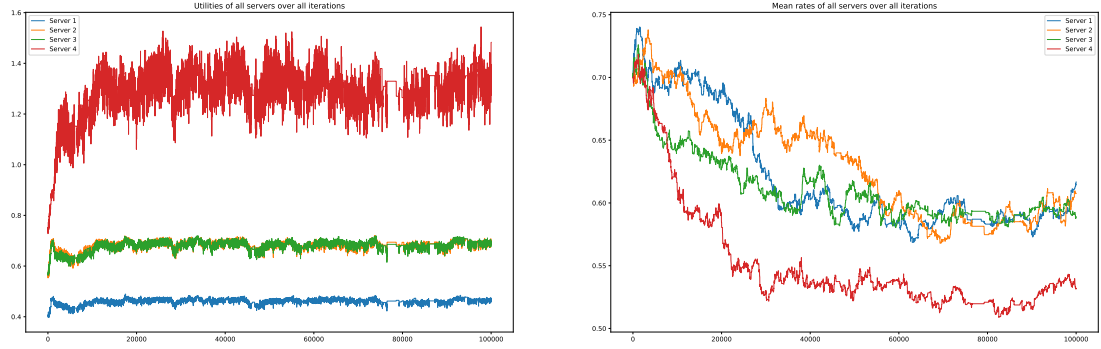


Figure 6.10: Utilities (left) and mean service rate (right) of servers from the reinforcement learning run using utility function  $U_k^{(3)}$  with  $e = 0.1$  and 100,000 iterations

It can be seen from Figure 6.10 that the utility of server 1 (that has the highest priority on incoming individuals) is the lowest, while server 4 (that has the lowest priority on incoming) individuals has the highest utility. In addition, the mean service rate of server 4 is the lowest while servers 1, 2 and 3 have similar mean service rates. All servers managed to increase their utility by reducing their mean service rate (i.e. working faster). Figure 6.11 shows the same example as above but with 500,000 iterations.

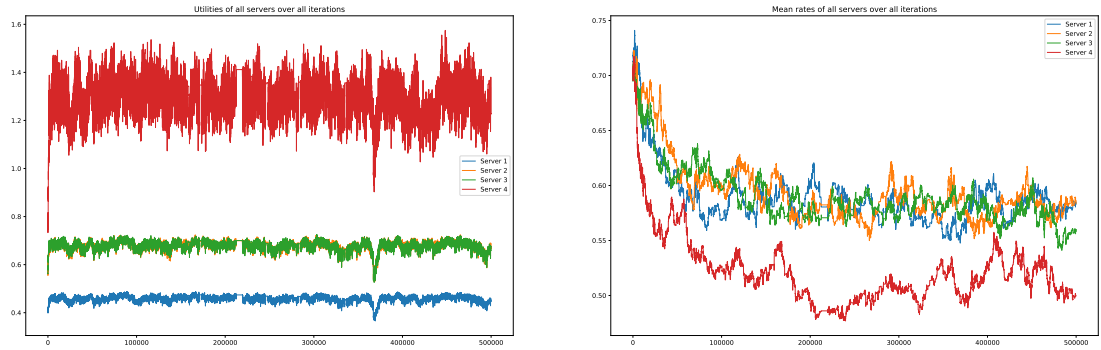


Figure 6.11: Utilities (left) and mean service rate (right) of servers from the reinforcement learning run using utility function  $U_k^{(3)}$  with  $e = 0.1$  and 500,000 iterations

Running the algorithm again for a longer runtime of 500,000 iterations does not change the results significantly. The same observations can be made for both figures.

#### 6.6.1.2 Utility function 7 ( $U_k^{(7)}$ ) with $e = 0.1$

Consider the plot of the mean rates of Figures 6.10 and 6.11 (rightmost graph). The service rate of each server consists of a different service rate for each state

$(u, v)$  of the system. The way the reinforcement learning algorithm is constructed, the service rate of each server is updated at each iteration based on the visited states of the system. This means that it doesn't make sense to calculate the mean service rate from all states of the system. In this subsection, the weighted mean service rate is used instead, where the weight of each state is the probability of visiting that state. The weighted mean service rate of server  $k$  is defined as:

$$\hat{\mu}^{(k)} = \sum_{(u,v) \in \mathcal{S}} \pi(u, v) \mu_{k,(u,v)} \quad (6.17)$$

Consider the same example as in the previous subsection but with utility function 7 defined in (6.13) where  $U_k^{(7)} = e \frac{I_s}{I} + (1 - e) \frac{R - B^{(k)}}{R}$  and  $e = 0.1$ . Note that given the nature of this utility function the utilities of the agents can now only be within the range  $[0, 1]$ .

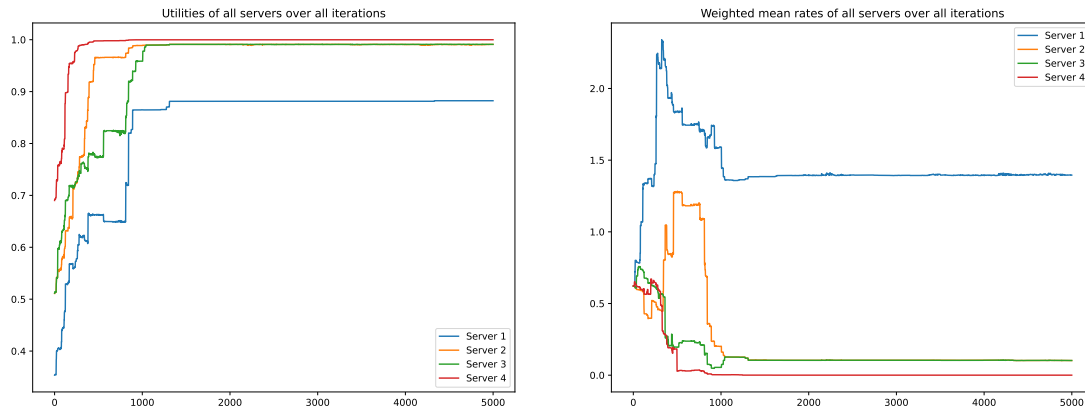


Figure 6.12: Utilities (left) and weighted mean service rate (right) of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.1$ .

Figure 6.12 shows the utilities and weighted mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$ . It can be seen from Figure 6.12 that the utilities follow a similar pattern to the ones from Section 6.6.1.1. That is, the utility of server 1 is the lowest while the utility of server 4 is the highest, leaving servers 2 and 3 with similar utilities in between. In terms of the weighted mean service rate, servers 2, 3 and 4 managed to reduce their service rate while server 1 had to increase its service rate in order to maximise its utility.

What happens if the initial service rate of all servers is changed? Would the servers manage to arrive to the same policies and same utilities? Figure 6.13 shows the utilities and weighted mean service rates of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.1$  and an initial service rate of

0.2 for all servers.

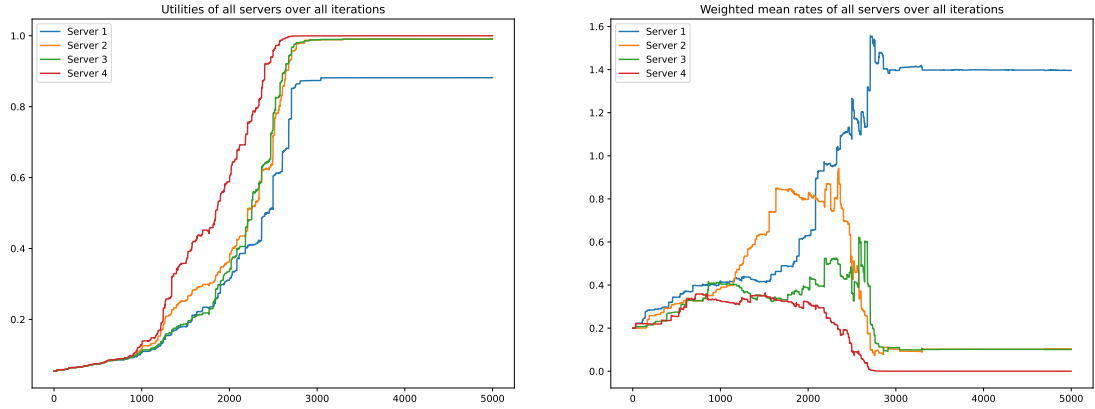


Figure 6.13: Utilities (left) and weighted mean service rate (right) of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.1$  and an initial service rate of 0.2 for all servers.

Figure 6.13 shows that, even though it takes longer to stabilise, the reinforcement learning algorithm is able to find the same utilities as before. In addition, it can be seen from the weighted mean service rates that servers have also managed to find the same policies as before. Note that more exploration is needed from the agents in order to reach the same policies as before.

Now, consider the scenario where the initial service rate of all servers is increased. Figure 6.14 shows the utilities and weighted mean service rates of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.1$  and an initial service rate of 1.5 for all servers.

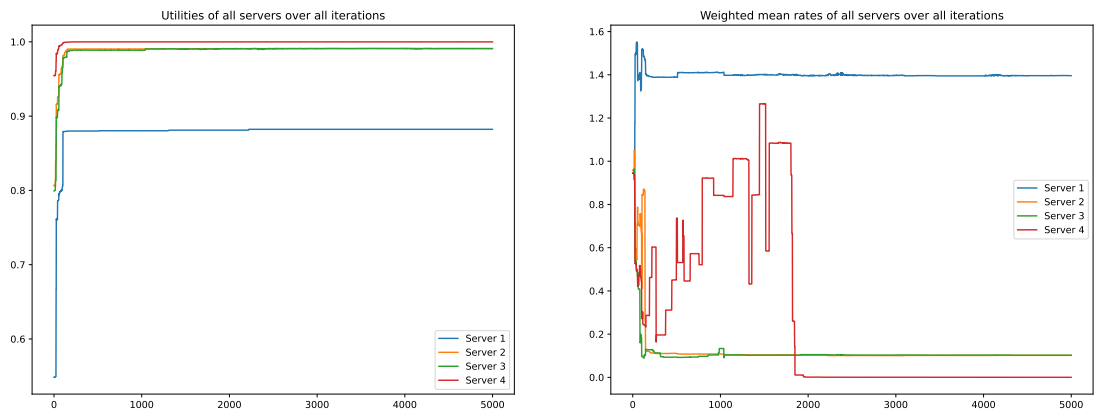


Figure 6.14: Utilities (left) and weighted mean service rate (right) of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.1$  and an initial service rate of 1.5 for all servers.

### 6.6.1.3 Utility function 7 ( $U_k^{(7)}$ ) with $e = 0.5$

This subsection considers the same parameters and utility function as in Section 6.6.1.2 but with a different value for parameter  $e$ . Figure 6.15 shows the utilities and weighted mean service rates of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.5$ .

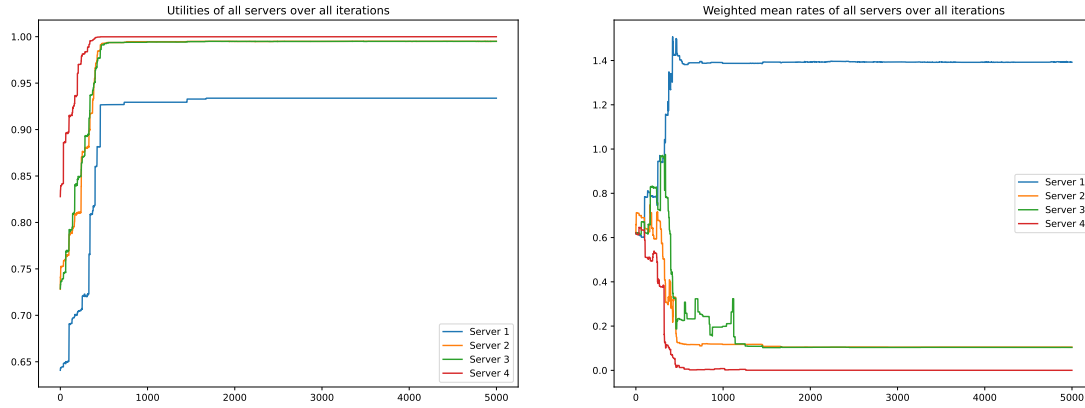


Figure 6.15: Utilities (left) and weighted mean service rate (right) of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.5$ .

A similar pattern to the one in Figure 6.12 can be seen from Figure 6.15. The policies found by the agents at the end of the run are similar to the ones found when using  $e = 0.1$ . The only difference is that the utilities are slightly higher when using  $e = 0.5$ .

Consider the scenario where the arrival rates of the two servers are increased. Figure 6.16 shows the utilities and weighted mean service rates of servers when increasing arrival rates of  $\lambda_1 = 2$  and  $\lambda_2 = 2.5$ .

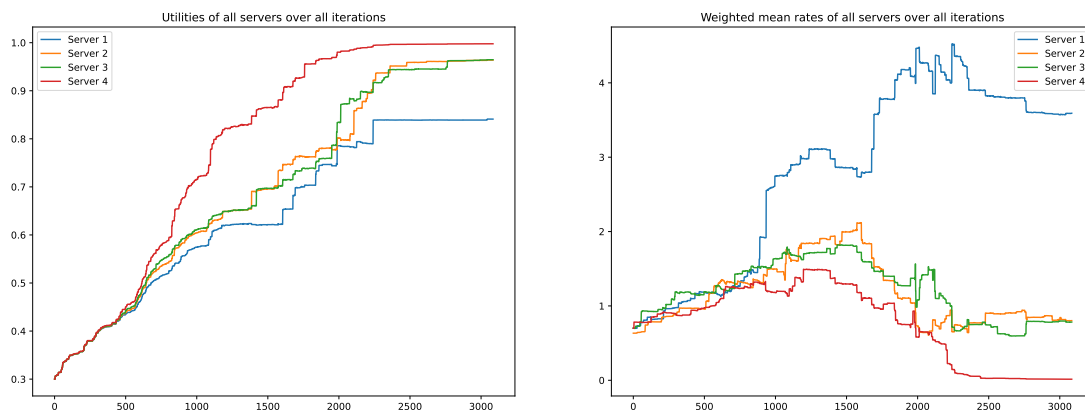


Figure 6.16: Utilities (left) and weighted mean service rate (right) of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.5$  and increased arrival rates of  $\lambda_1 = 2$  and  $\lambda_2 = 2.5$ .



$$\begin{bmatrix}
0 & 4 \times 10^{15} & 9.4 & 0.5 & 2.1 & 1.8 & 0.8 & 0.4 & 0.7 & 0.7 & 0.7 \\
\text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & 0.7 & 1.9 & 0.08 & 0.7 \\
\text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & 1.6 & 0.7 & 0.1 & 0.7 \\
\text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & 1.3 & 0.7 & 0.7 & 0.7 \\
\text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & 0.7 & 0.2 & 0.7 & 0.7 \\
\text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & 0.7 & 0.7 & 0.7 & 0.7 \\
\text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & 0.7 & 0.7 & 0.7 & 0.7 \\
\text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & 0.7 & 0.7 & 0.7 & 0.7
\end{bmatrix} \quad (6.19)$$

Equation (6.18) shows the state probabilities of the system and equation (6.19) shows the service rates of server 1. The service rates of the remaining servers are not shown since they are not relevant. Note that the missing values in equation (6.18) indicate that not only the state probabilities are 0 for these states, but can't even be visited by the system. With a service rate of  $4 \times 10^{15}$  for server 1, state probability  $2.44 \times 10^{-19}$  for state  $(0, 1)$  and state probability 1.0 for state  $(0, 0)$ , the weighted mean service rate of server 1 is:

$$(0 \times 1.0) + [(4 \times 10^{15}) \times (2.44 \times 10^{-19})] \approx 0.001 \quad (6.20)$$

That is the reason why an upper bound on the service rates is needed. Without an upper bound, servers could choose an extremely high service rate and thus making the system reach unreachable scenarios.

#### 6.6.1.5 Changing arrival rates during the run

Consider once again the same parameters and utility function as in Section 6.6.1.3. That is a using a utility function  $U_k^{(7)}$  with  $e = 0.5$  and arrival rates of  $\lambda_1 = 0.5$  and  $\lambda_2 = 1$ . In this subsection the reinforcement learning algorithm is run with the same parameters, but the arrival rates are changed during the run. The total runtime of the reinforcement learning algorithm is 5000 iterations. The arrival rates are set to  $\lambda_1 = 0.5$  and  $\lambda_2 = 1$  for the first 2000 iterations. Then the arrival rates are increased to  $\lambda_1 = 3$  and  $\lambda_2 = 3.5$  for iterations 2000 to 4000 and then the arrival rates are decreased back to  $\lambda_1 = 0.5$  and  $\lambda_2 = 1$  for the last 1000 iterations.



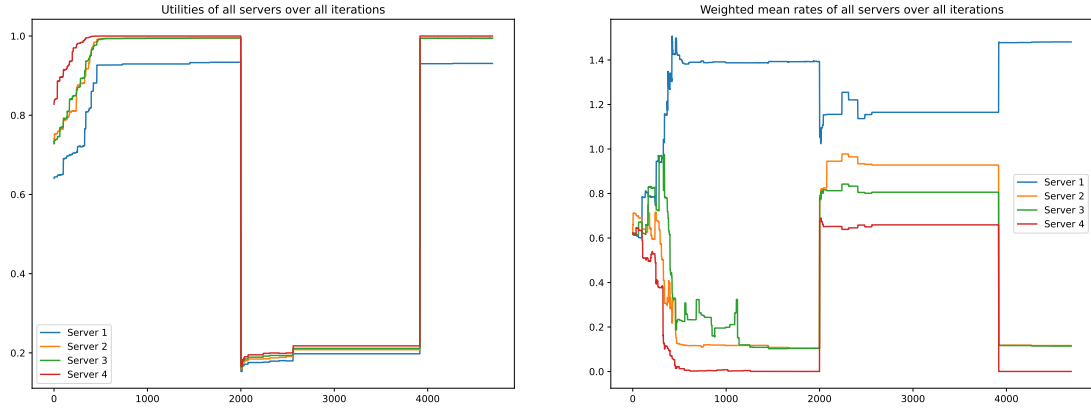


Figure 6.18: Utilities (left) and weighted mean service rate (right) of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.5$  and changing arrival rates throughout the run.

Figure 6.18 shows the utilities and the weighted mean service rates of the servers. The first 2000 iterations are identical with the first 2000 iterations of the reinforcement learning run of Figure 6.15. At iteration 2000, when the arrival rates are increased to  $\lambda_1 = 3$  and  $\lambda_2 = 3.5$ , the utilities of the agents drop significantly. At the same time the weighted mean service rates of servers 2, 3 and 4 increase significantly while server 1's rates are decreased. In other words the arrival rates are increased so much that the priority of the servers does not matter that much anymore. All servers are constantly busy and they end up having more similar service rates with each other.

At iteration 4000 the arrival rates are decreased back to  $\lambda_1 = 0.5$  and  $\lambda_2 = 1$ . The utilities of the agents increase again and the weighted mean service rates of servers 2, 3 and 4 decrease while server 1's rates are increased. What is more important here is how the utilities and the weighted mean service rates change. Having learned their optimal service rates during the first 2000 iterations the servers are able to quickly retrieve their chosen service rates when the arrival rates are decreased back to  $\lambda_1 = 0.5$  and  $\lambda_2 = 1$ .

## 6.7 Chapter summary

This chapter aims to explore an extension to the queueing network described in Section 3. The proposed model is an agent-based model where the servers can choose their own service rate. In addition, a reinforcement learning algorithm is implemented to allow the servers to learn their own service rate so that they maximise their own utility function.

Section 6.2 describes a variant of the queueing system described in Chapter 3

where the service rate of the servers is dependent on the state of the system. In this model there is a service rate for each particular state of the system. This modification attempts to capture the idea that the servers might be more likely to serve patients faster when the system is under pressure. In addition, Section 6.3 describes another variant of the queueing system where the service rate can be dependent on the server. This can be used to capture the individual behaviour of servers, where some servers might be more likely to serve patients faster than others. Finally, Section 6.4 combines the two concepts into a model where the service rate of the servers is dependent on both the servers and the state of the system. An example of combining the state and server-dependent model with the game theoretic model described in Chapter 4 is also given.

Section 6.5 then proceeds to use the state and server-dependent model to create an agent-based model where the servers can choose their own service rate to maximise a utility function. Some possible utility functions are given and a case study is described where one of the utility functions is used to model the behaviour of the servers. Finally, Section 6.6 uses the agent-based model to create a reinforcement learning model where the servers can learn to choose their own service rate in order to maximise their own utility function. Some numerical results are also given to show how the change in the servers' behaviour can affect the overall performance of the system. A particular scenario is investigated where the reinforcement learning algorithm is used to train the servers and when the servers have reached a stable service rate, the system is flooded with individuals. The results show how the servers are able to adapt to the new situation and when the system is no longer flooded, the servers are able to return to their pre-learned service rates.

# Chapter 7

## Conclusions

This Chapter aims to summarise the research presented in this thesis and to provide a reflection on the research process and the contributions that have been made. Finally, it will provide some recommendations for future work. Each chapter of this thesis included a *Chapter Summary* section, and so the summary here will be brief.

### 7.1 Research Overview

The motivation behind this thesis has been that emergency departments are under a lot of pressure to treat patients. This is, in practice, often centrally controlled through a mechanism of some sort of performance measure target. The research presented in this thesis shows how this can negatively impact the pathway of both the ambulance patients and the ambulance service itself. Due to some managerial decision making that takes place at the Emergency Department (ED), ambulances may stay blocked outside of the ED in the hospital's parking zone in an attempt to satisfy these regulations.

This thesis presents a queueing network model that is used to describe an ED that accepts patients arriving by ambulance and patients that arrive by other means. The model is then used to construct a game theoretic model that is informed by the queueing network model. The game theoretic model is presented as a 3-player game between the Emergency Medical Services (EMS) and two EDs. The game theoretic model is then used to explore the impact of different strategies on the performance of the EDs.

Chapter 1 provides an overview of Operational Research (OR) and the problem of congestion in healthcare which served as the motivation for this research.

Chapter 2 then provides a literature review of the relevant research that has been done. Namely, a review of the literature on OR models applied to healthcare systems, a review of the conjunction of queueing theory and game theory and a review of the literature on game theoretic models applied to healthcare has been presented. Moreover, a brief review on behavioural OR is also presented to provide some context for the agent-based model that is presented in this thesis. Chapter 3 then introduces a queueing network model with two waiting spaces that accepts two types of individuals. The types of individuals are then used to describe an ED that accepts patients arriving by ambulance and patients that arrive by other means. The modelling approaches along with the calculations for the model's performance measures are also presented. In addition, some numerical comparisons of the different approaches are given as a form of validation of the different approaches. Chapter 4 introduces a game theoretic model using the queueing network model as a basis. Essentially, the queueing network model is used to construct a 3-player game between the decision makers of two queueing networks and a provider that distributes individuals between the two queueing networks. This is later mapped to a 3-player game between the EMS and two EDs. In the methodology Brent's algorithm was used to find the best response of the third player (the EMS), for all possible combinations of strategies that the other two players (the two EDs) might choose. The resultant game is then reduced to a 2-player game between the two EDs where already existing Nash Equilibrium algorithms and evolutionary learning algorithms could be applied. Some results of the numerical experiments are then presented and discussed in Chapter 5. The particular scenario that was explored was one between two EDs that were heavily congested. Several what-if scenarios were investigated to determine ways to reduce the ambulance congestion at the EDs.

In addition to the game theoretic model, an agent-based model was also developed in Chapter 6. Instead of the previous constant service times, the queueing model was expanded to use state and server-dependent service times. As a result, an agent-based model is built with different service times for every server and system state. The learning that occurs when servers determine the speed at which they serve customers in order to maximise some utility is then observed using a reinforcement learning algorithm. Some numerical results are then presented and discussed in Section 6.6.1.

The motivation for the agent-based extension came from certain ethnographic insights where it was observed that ambulance staff and ED staff were in fact playing a more cooperative game. In fact, the players were not only trying to maximise their own utility but also the utility of the system. Thus, some of

the utility functions used for the agent-based model were structured in a way that aimed to increase both each staff member's happiness and the overall social welfare of the system.

## 7.2 Contributions

The research presented in this thesis has made some novel contributions to the literature on OR models and healthcare applications. The findings of this thesis that relate to the queueing network and the game theoretic model have also been published and are presented in [114]. The contributions are as follows:

- A novel queueing network model with two waiting spaces where one serves as a buffer for the other. The model is used to describe an ED that accepts patients arriving by ambulance and patients that arrive by other means.
- Performance measure calculations for the queueing network model. Such performance metrics include the average number of individuals in the system, mean waiting time, mean blocking time and proportion of individuals that are served within a certain time.
- A 3-player game theoretic model between two queueing networks and a provider that distributes individuals between the two queueing networks. The game is then mapped to a 3-player game between the EMS and two EDs.
- Numerical experiments showing emergent behaviour of gaming between EDs and the EMS. A scenario where two EDs are heavily congested is explored and several what-if scenarios are investigated to determine ways to reduce the ambulance congestion at the EDs.
- An agent-based model with reinforcement learning that is used to explore the learning that occurs when servers determine the speed at which they serve customers in order to maximise some utility. The model is built using the queueing network model as a basis.
- Numerical experiments using the agent-based model with reinforcement learning to explore the learning that occurs.

Although this research is motivated by the particular EMS-ED example, the developed modelling framework and behavioural insights has applications to similar systems across a range of sectors and settings. The queueing model can be applied to any setting where individuals may be blocked on a separate queue. An

example of such setting can be any type of delivery service where customers can purchase goods either online or in-person. At busier times, the person delivering the product may be blocked outside the store in an attempt to improve the waiting times for walk-in customers.

The key findings from this paper that were observed when playing the game between two EDs and the EMS are:

- Inefficiencies can be learned and emerge naturally;
- Targeted incentivisation of behaviours can help escape inefficiencies.

The former relates to the results of asymmetric replicator dynamics shown in Chapter 5. The results showed that inefficient scenarios can arise when letting the players play the game by prioritising their best interests, while the latter implies that these learned inefficiencies can be escaped by carefully applying certain incentives to the players. In theory, this careful incentivisation of behaviours is done by applying some form of penalty to the payoff matrices of the players to force them change their strategy. In practice, getting that penalty is more difficult.

The motivation for this thesis has been the problem of ambulance congestion at EDs. The findings of the thesis suggest that by letting the EDs and the EMS play a non-cooperative game, some behaviour that might not be optimal for the EMS, is likely to emerge. Such behaviour could be escaped with careful incentivisation of the EDs. In practice, applying this incentive mechanism to an ED would be difficult because the EDs are not a single entity but rather a collection of individuals that could be incentivised in different ways. Although it would require further research to determine the best way to apply such incentive mechanisms in practice, the findings of this thesis could be used to inform the design of such incentive mechanisms.

Apart from the theoretical contributions, this thesis also made some contributions to open-source software. The following software contributions were made as part of this thesis:

- **ambulance\_game**: A Python package that implements the queueing network model and the game theoretic model presented in this thesis. A detailed description of the package can be found in Appendix A.
- **nashpy** contribution: Implemented the asymmetric replicator dynamics algorithm in the **nashpy** Python library.

- **ciw** contribution: Implemented custom server priorities in the **ciw** Python library.
- **ciw** contribution: Implemented server dependent distributions in the **ciw** Python library.

### 7.3 Future Work

The model that is being discussed here presupposes the presence of only two players that can receive individuals. However, in a realistic healthcare scenario an ambulance may have to decide among multiple EDs. An immediate extension of this work would be to consider a multiplayer system that could represent a group of hospitals in a concentrated area. Moreover, the developed game theoretic model employs a discrete strategy space for the EDs (something that is also present in various related literature [40, 77]). The single threshold parameter that is used for the ED's decision may not be a good representation of the way EDs actually operate. In reality ED managers might adopt far more complex parameters for their decision making process. Moreover, the game theoretic model of this work assumes that the EMS and EDs act in a selfish and rational way by only aiming to satisfy their own objectives. In some settings, cooperation may be observed and would therefore require an adapted modelling approach. The creation of the agent-based model that was introduced in Chapter 6.5 was motivated by the potential cooperation between the EMS and EDs. Further research could be done to explore the potential for cooperation in this setting and how it could be further investigated. Finally, future work could touch upon the completion of the work presented in Appendix D, where an attempt to develop a closed form formula for the steady state probabilities of the queueing model was made. The formula was not completed and therefore further research could be done to derive the formula and investigate its properties.

# Bibliography

- [1] Elvio Accinelli, Edgar J Sánchez Carrera, et al. Evolutionarily stable strategies and replicator dynamics in asymmetric two-population games. *Dynamics, games and science i*, 1:25–35, 2011.
- [2] Philipp Afeche. Decentralized service supply chains with multiple time-sensitive customer segments: Pricing capacity decisions and coordination. Technical report, Working paper, University of Toronto, 2007.
- [3] K. Erlang Agner. The theory of probabilities and telephone conversations. *Nyt Tidsskrift for Matematik B*, 20, 1909.
- [4] Vivienne Aitken. The red cross drafted into scotland’s biggest hospital as ambulances queue up. *Daily Record*, 2021.
- [5] Mohamed Akkouchi. On the convolution of exponential distributions. *Journal Chungcheong Mathematical Society*, 21(4):501–510, 2008.
- [6] Davina Allen, Lesley Griffiths, and Patricia Lyne. Understanding complex trajectories in health and social care provision. *Sociology of health & illness*, 26(7):1008–1030, 2004.
- [7] Márcio AC Almeida and Frederico RB Cruz. A note on bayesian estimation of traffic intensity in single-server markovian queues. *Communications in Statistics-Simulation and Computation*, 47(9):2577–2586, 2018.
- [8] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users’ Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.
- [9] Dave Astels. *Test driven development: A practical guide*. Prentice Hall Professional Technical Reference, 2003.
- [10] Urtzi Ayesta. Reinforcement learning in queues. *Queueing Systems*, 100(3-



- 4):497–499, 2022.
- [11] Rangaswami Balakrishnan and Kanna Ranganathan. *A textbook of graph theory*, chapter 2, pages 37–47. Springer Science & Business Media, 2012.
- [12] Dragan Banjevic. Recursive relations for the distribution of waiting times in a markov chain. *Journal of applied probability*, 33(1):48–56, 1996.
- [13] T Başar and R Srikant. A stackelberg network game with a large number of followers. *Journal of optimization theory and applications*, 115(3):479–490, 2002.
- [14] Gely P. Basharin, Amy N. Langville, and Valeriy A. Naumov. The life and work of a.a. markov. *Linear Algebra and its Applications*, 386:3–26, 2004.
- [15] Francis Bashforth and John Couch Adams. *An attempt to test the theories of capillary action by comparing the theoretical and measured forms of drops of fluid*. University Press, 1883.
- [16] Edward A Bender and S Gill Williamson. *Lists, decisions and graphs*. S. Gill Williamson, 2010.
- [17] Dietmar Berwanger and Laurent Doyen. On the power of imperfect information. *Leibniz International Proceedings in Informatics, LIPIcs*, 2:73–82, 2008.
- [18] Ugur Bilge and Osman Saka. Agent based simulations in healthcare. *Studies in Health Technology and Informatics*, 124:699–704, 2006.
- [19] Patrick Maynard Stuart Blackett. *Studies of war, nuclear and conventional*. Hill and Wang, 1962.
- [20] John D Blischak, Emily R Davenport, and Greg Wilson. A quick introduction to version control with git and github. *PLoS computational biology*, 12(1):e1004668, 2016.
- [21] Béla Bollobás. *Modern graph theory*, volume 184, chapter 1, pages 1–22. Springer Science & Business Media, 1998.
- [22] Richard P Brent. *Algorithms for Minimization Without Derivatives*. Englewood Cliffs, 1973.
- [23] Andrei Z Broder. Generating random spanning trees. In *FOCS*, volume 89, pages 442–447. Citeseer, 1989.

- [24] George W Brown. Iterative solution of games by fictitious play. *Act. Anal. Prod Allocation*, 13(1):374, 1951.
- [25] Apostolos N Burnetas. Customer equilibrium and optimal strategies in markovian queues in series. *Annals of Operations Research*, 208(1):515–529, 2013.
- [26] Gérard P Cachon and Fuqiang Zhang. Obtaining fast service in a queueing system via performance-based allocation of demand. *Management Science*, 53(3):408–420, 2007.
- [27] Seth Chaiken and Daniel J Kleitman. Matrix tree theorems. *Journal of combinatorial theory, Series A*, 24(3):377–381, 1978.
- [28] Hong Chen and Yat-Wah Wan. Price competition of make-to-order firms. *IIE Transactions*, 35(9):817–832, 2003.
- [29] Hong Chen and Yat-wah Wan. Capacity competition of make-to-order firms. *Operations Research Letters*, 33(2):187–194, 2005.
- [30] Wuhua Chen, Zhe George Zhang, and Xiaohong Chen. On two-tier health-care system under capacity constraint. *International Journal of Production Research*, 58(12):3744–3764, 2020.
- [31] Hsing Kenneth Cheng, Haluk Demirkan, and Gary J Koehler. Price and capacity competition of application services duopoly. *Information Systems and e-Business Management*, 1(3):305–329, 2003.
- [32] A Clarey, M Allen, S Brace-McDonnell, and MW Cooke. Ambulance handovers: can a dedicated ed nurse solve the delay in ambulance turnaround times? *Emergency Medicine Journal*, 31(5):419–420, 2014.
- [33] Owain Clarke. A&e queues mean wales ambulances can’t take 999 calls. *BBC Wales*, 2021.
- [34] George Corliss. Which root does the bisection algorithm find? *Siam Review*, 19(2):325–327, 1977.
- [35] Julie Crouch. Thousands of hours lost as ambulance idle outside hospitals in derbyshire. *Staffordshire Live*, 2021.
- [36] Charles Francis Curtiss and Joseph O Hirschfelder. Integration of stiff equations. *Proceedings of the National Academy of Sciences*, 38(3):235–243, 1952.

- 
- [37] Bernardo D’Auria and Spyridoula Kanta. Pure threshold strategies for a two-node tandem network under partial information. *Operations Research Letters*, 43(5):467–470, 2015.
- [38] Suzanne Day. Woman, 85, has ‘appalling’ three hour wait for ambulance. *East Anglia*, 2021.
- [39] Scott De Marchi and Scott E Page. Agent-based models. *Annual Review of political science*, 17:1–20, 2014.
- [40] Sarang Deo and Itai Gurvich. Centralized vs. decentralized ambulance diversion: A network perspective. *Management Science*, 57(7):1300–1319, 2011.
- [41] John R Dormand and Peter J Prince. A family of embedded runge-kutta formulae. *Journal of computational and applied mathematics*, 6(1):19–26, 1980.
- [42] Christine Duguay and Fatah Chetouane. Modeling and improving emergency department systems using discrete event simulation. *Simulation*, 83(4):311–320, 2007.
- [43] James F Epperson. *An introduction to numerical methods and analysis*. John Wiley & Sons, 2021.
- [44] Paula Escudero-Marin and Michael Pidd. Using abms to simulate emergency departments, 2011.
- [45] Ming Fan, Subodha Kumar, and Andrew B Whinston. Short-term and long-term competition between providers of shrink-wrap software and software as a service. *European Journal of Operational Research*, 196(2):661–671, 2009.
- [46] Stefano Favaro and Stephen G Walker. On the distribution of sums of independent exponential random variables via wilks’ integral representation. *Acta applicandae mathematicae*, 109(3):1035–1042, 2010.
- [47] Angelico Giovanni Fetta, Paul Robert Harper, Vincent Anthony Knight, Israel Teixeira Vieira, and Janet Elizabeth Williams. On the peter principle: An agent based investigation into the consequential effects of social networks and behavioural factors. *Physica A: Statistical Mechanics and its Applications*, 391(9):2898–2910, 2012.
- [48] Peter C Fishburn. Utility theory. *Management science*, 14(5):335–378,

- 1968.
- [49] Peter C Fishburn. *Utility theory for decision making*, volume 1. Research analysis corp McLean VA, 1970.
  - [50] Python Software Foundation. Python package index - pypi.
  - [51] L Alberto Franco and Raimo P Hämäläinen. Engaging with behavioral operational research: On methods, actors and praxis. *Behavioral operational research: Theory, methodology and practice*, pages 3–25, 2016.
  - [52] Drew Fudenberg, Fudenberg Drew, David K Levine, and David K Levine. *The theory of learning in games*, volume 2. MIT press, 1998.
  - [53] Denis Getsios, Steve Blume, K. Jack Ishak, and Grant D.H. MacLaine. Cost effectiveness of donepezil in the treatment of mild to moderate alzheimer’s disease: A uk evaluation using discrete-event simulation. *PharmacoEconomics*, 28(5):411–427, 2010.
  - [54] Erik Giesen, Wolfgang Ketter, and Rob Zuidwijk. An agent-based approach to improving resource allocation in the dutch youth health care sector, 2009.
  - [55] Nikoleta E Glynatsi and Vincent A Knight. A bibliometric study of research topics, collaboration, and centrality in the iterated prisoner’s dilemma. *Humanities and Social Sciences Communications*, 8(1):1–12, 2021.
  - [56] A. Gosavi, S.L. Murray, and N. Karagiannis. A markov chain approach for forecasting progression of opioid addiction, 2020.
  - [57] Andrew Greasley and Chris Owen. Behavior in models: A framework for representing human behavior. *Behavioral operational research: Theory, methodology and practice*, pages 47–63, 2016.
  - [58] Nyoman Gunantara. A review of multi-objective optimization: Methods and its applications. *Cogent Engineering*, 5(1):1502242, 2018.
  - [59] Ian Hacking et al. *Representing and intervening: Introductory topics in the philosophy of natural science*. Cambridge university press, 1983.
  - [60] Reidar Hagtvedt, Mark Ferguson, Paul Griffin, Gregory Todd Jones, and Pinar Keskinocak. Cooperative strategies to reduce ambulance diversion. In *Proceedings of the 2009 Winter Simulation Conference (WSC)*, pages 1861–1874. IEEE, 2009.
  - [61] Amanda Halliwell. Cqc covid insight: Winter pressures on emergency care.

- Practice Management*, 31(4):28–30, 2021.
- [62] Raimo P Hämäläinen, Jukka Luoma, and Esa Saarinen. On the importance of behavioral operational research: The case of understanding and communicating about dynamic systems. *European Journal of Operational Research*, 228(3):623–634, 2013.
- [63] F.Z. Hamdani, L. Jdaini, M. Masmoudi, and J. Roche. Quantitative modelling of elderly people flow within french healthcare system, 2015.
- [64] Paul R Harper. Server behaviours in healthcare queueing systems. *Journal of the Operational Research Society*, 71(7):1124–1136, 2020.
- [65] Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585:357–362, 2020.
- [66] Sergiu Hart. Games in extensive and strategic forms. *Handbook of game theory with economic applications*, 1:29–32, 1992.
- [67] Josef Hofbauer and William H Sandholm. On the global convergence of stochastic fictitious play. *Econometrica*, 70(6):2265–2294, 2002.
- [68] Hiroyuki Iizuka, Keiji Suzuki, Masahito Yamamoto, and Azuma Ohuchi. Learning of virtual words utilized in negotiation process between agents. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E83-A(6):1075–1082, 2000.
- [69] Joshua Conrad Jackson, David Rand, Kevin Lewis, Michael I Norton, and Kurt Gray. Agent-based modeling: A guide for social psychologists. *Social Psychological and Personality Science*, 8(4):387–395, 2017.
- [70] Rafael C Jiménez, Mateusz Kuzak, Monther Alhamdoosh, Michelle Barker, Bérénice Batut, Mikael Borg, Salvador Capella-Gutierrez, Neil Chue Hong, Martin Cook, Manuel Corpas, et al. Four simple recommendations to encourage best practices in research software. *F1000Research*, 6, 2017.
- [71] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Rein-

- forcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.
- [72] Ehud Kalai, Morton I Kamien, and Michael Rubinovitch. Optimal service speeds in a competitive environment. *Management Science*, 38(8):1154–1163, 1992.
- [73] Hans Keiding. On the structure of the set of nash equilibria of weakly nondegenerate bimatrix games. *Annals of Operations Research*, 84:231–238, 1998.
- [74] John G Kemeny and J Laurie Snell. *Markov chains*, volume 6. Springer-Verlag, New York, 1976.
- [75] Matthew Knapper, Dave & Dresch. Ambulances queue 10 in a row outside hospital buckling under ‘significant’ pressure. *Mirror*, 2021.
- [76] Vincent Knight, Paul Harper, Jeff Griffiths, Izabela Komenda, and Rob Shone. An incomplete overview of some applications of game theory to patient flow, 2014.
- [77] Vincent Knight, Izabela Komenda, and Jeff Griffiths. Measuring the price of anarchy in critical care unit interactions. *Journal of the Operational Research Society*, 68(6):630–642, 2017.
- [78] Vincent Knight and Geraint Palmer. *Applied Mathematics with Open-Source Software: Operational Research Problems with Python and R*, chapter 3, pages 29–44. CRC Press, 2022.
- [79] Vincent A Knight and Paul R Harper. Selfish routing in public services. *European Journal of Operational Research*, 230(1):122–132, 2013.
- [80] Vincent A Knight, Janet E Williams, and I Reynolds. Modelling patient choice in healthcare systems: development and application of a discrete event simulation with agent-based decision making. *Journal of Simulation*, 6(2):92–102, 2012.
- [81] Natalia L Komarova. Replicator–mutator equation, universality property and population dynamics of learning. *Journal of theoretical biology*, 230(2):227–239, 2004.
- [82] Elias Koutsoupias and Christos Papadimitriou. Worst-case equilibria. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 404–413. Springer, 1999.

- 
- [83] David M Kreps. Nash equilibrium. In *Game Theory*, pages 167–177. Springer, 1989.
- [84] Mateusz Krukowski and Filip Turoboś. Approximate solutions to the travelling salesperson problem on semimetric graphs. *arXiv preprint arXiv:2105.07275*, 2021.
- [85] Martin Kunc, Paul Harper, and Konstantinos Katsikopoulos. A review of implementation of behavioural aspects in the application of or in healthcare. *Journal of the Operational Research Society*, 71(7):1055–1072, 2020.
- [86] F Shampine Lawrence. Some practical runge-kutta formulas. *Mathematics of Computation*, 46:135–150, 1986.
- [87] Benjamin Legros and Oualid Jouini. A linear algebraic approach for the computation of sums of erlang random variables. *Applied Mathematical Modelling*, 39(16):4971–4977, 2015.
- [88] C. E. Lemke and J. T. Howson, Jr. Equilibrium points of bimatrix games. *Journal of the Society for Industrial and Applied Mathematics*, 12(2):413–423, 1964.
- [89] Richard Lemmer. Portsmouth’s queen alexandra hospital patients see more than 250 hour-long ambulance handover delays. *The News*, 2021.
- [90] Lionel Levine. Sandpile groups and spanning trees of directed line graphs. *Journal of Combinatorial Theory, Series A*, 118(2):350–364, 2011.
- [91] Hao-yu Liao, Willie Cade, and Sara Behdad. Markov chain optimization of repair and replacement decisions of medical equipment. *Resources, Conservation and Recycling*, 171, 2021.
- [92] Bai Liu, Qiaomin Xie, and Eytan Modiano. Reinforcement learning for optimal control of queueing systems. In *2019 57th annual allerton conference on communication, control, and computing (allerton)*, pages 663–670. IEEE, 2019.
- [93] Sridhar Mahadevan. Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Recent advances in reinforcement Learning*, pages 159–195, 1996.
- [94] Elizabeth Mahase. Covid-19: Hospitals in crisis as ambulances queue and staff are asked to cancel leave. *The BMJ*, 2020.

- [95] Rajesh Mansharamani. An overview of discrete event simulation methodologies and implementation. *Sadhana*, 22(5):611–627, 1997.
- [96] Michael Maschler, Eilon Solan, and Eilon Zamir. *Game Theory*. Cambridge University Press, 2013.
- [97] Ben McAdams. Caldicot girl waited nine hours for ambulance after fall. *South Wales Argus*, 2021.
- [98] Sally McClean and Peter Millard. Using markov models to manage high occupancy hospital care. *IEEE Intelligent Systems*, pages 256–260, 2006.
- [99] Erin C McKiernan, Philip E Bourne, C Titus Brown, Stuart Buck, Amye Kenall, Jennifer Lin, Damon McDougall, Brian A Nosek, Karthik Ram, Courtney K Soderberg, et al. How open science helps researchers succeed. *elife*, 5:e16800, 2016.
- [100] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.
- [101] Geoff Mulgan, Tom Steinberg, and Omar Salem. *Wide Open: Open source methods and their future potential*. Demos London, 2005.
- [102] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.
- [103] Roger B Myerson. Refinements of the nash equilibrium concept. *International journal of game theory*, 7(2):73–80, 1978.
- [104] Roger B Myerson. *Game theory: analysis of conflict*. Harvard university press, 1997.
- [105] Christos Nicolaides, Luis Cueto-Felgueroso, and Ruben Juanes. The price of anarchy in mobility-driven contagion dynamics. *Journal of The Royal Society Interface*, 10(87):20130495, 2013.
- [106] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, 2007.



- 
- [107] Martin Nowak. Evolutionary and spatial dynamics of the prisoner's dilemma. In *Pre-proceedings Second Euro. Conf. Art. Life*, volume 2, pages 863–870, 1993.
- [108] Martin J Osborne et al. *An introduction to game theory*, volume 3, chapter 2, pages 11–19. Oxford university press New York, 2004.
- [109] Martin J Osborne et al. *An introduction to game theory*, volume 3, chapter 5, pages 151–159. Oxford university press New York, 2004.
- [110] Geraint I. Palmer, Vincent A. Knight, Paul R. Harper, and Asyl L. Hawa. Ciw: An open-source discrete event simulation library. *Journal of Simulation*, 13(1):68–82, 2019.
- [111] Michalis Panayides. Michalispanayides/ambulancedecisiongame, December 2021.
- [112] Michalis Panayides. Michalis panayides thesis: Game dataset, January 2023.
- [113] Michalis Panayides. Michalis Panayides thesis: Reinforcement learning dataset, January 2023.
- [114] Michalis Panayides, Vince Knight, and Paul Harper. A game theoretic model of the behavioural gaming that takes place at the ems - ed interface. *European Journal of Operational Research*, 305(3):1236–1258, 2023.
- [115] Joanna M Papakonstantinou and Richard A Tapia. Origin and evolution of the secant method in one dimension. *The American Mathematical Monthly*, 120(6):500–517, 2013.
- [116] Linda Petzold. Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. *SIAM Journal on Scientific and Statistical Computing*, 4(1):136–148, 1983.
- [117] Daniele Procida. Diátaxis documentation framework.
- [118] Alexander Pushkin. *Eugene Onegin*. Penguin UK, 2003.
- [119] Safraz Rampersaud, Lena Mashayekhy, and Daniel Grosu. Computing nash equilibria in bimatrix games: Gpu-based parallel support enumeration. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3111–3123, 2014.
- [120] A Ravi Ravindran. *Operations research applications*. CRC Press, 2008.

- 
- [121] Ronald Rivest. The md5 message-digest algorithm, 1992.
- [122] Stewart Robinson. Discrete-event simulation: From the pioneers to the present, what next? *Journal of The Operational Research Society - J OPER RES SOC*, 56:619–629, 06 2005.
- [123] Damien Rolon-Mérette, Matt Ross, Thaddé Rolon-Mérette, and Kinsey Church. Introduction to anaconda and python: Installation and setup. *Quant. Methods Psychol*, 16(5):S3–S11, 2016.
- [124] Tim Roughgarden. *Selfish routing and the price of anarchy*. MIT press, 2005.
- [125] Somayeh Sadat, Hossein Abouee-Mehrizi, and Michael W Carter. Can hospitals compete on quality? *Health care management science*, 18(3):376–388, 2015.
- [126] Rob Shone, Vincent A Knight, and Janet E Williams. Comparisons between observable and unobservable m/m/1 queues with respect to optimal customer behavior. *European Journal of Operational Research*, 227(1):133–141, 2013.
- [127] Khaled Smaili, Therrar Kadri, and Seifedine Kadry. Hypoexponential distribution with different parameters. *Applied Mathematics*, 2013.
- [128] J Maynard Smith. Game theory and the evolution of fighting. *On evolution*, pages 8–28, 1972.
- [129] JMPGR Smith and George R Price. The logic of animal conflict. *Nature*, 246(5427):15–18, 1973.
- [130] Diomidis Spinellis. Git. *IEEE software*, 29(3):100–101, 2012.
- [131] Hayden Stainsby, Manel Taboada, and Emilio Luque. Towards an agent-based simulation of hospital emergency departments, 2009.
- [132] Lachlan Standfield, Tracy Comans, and Paul Scuffham. Markov modeling and discrete event simulation in health care: A systematic comparison. *International Journal of Technology Assessment in Health Care*, 30(2):165–172, 2014.
- [133] Gilbert Strang. *Linear algebra and its applications*. Belmont, CA: Thomson, Brooks/Cole, 2006.
- [134] J. Stringer. Operational research for “multi-organizations”. *Journal of the*

- Operational Research Society*, 18(2):105–120, 1967.
- [135] Wei Sun, Shi-yong Li, Nai-shuo Tian, and Hong-ke Zhang. Equilibrium analysis in batch-arrival queues with complementary services. *Applied Mathematical Modelling*, 33(1):224–241, 2009.
- [136] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [137] János Sztrik. Queueing theory and its applications, a personal view. In *Proceedings of the 8th international conference on applied informatics*, volume 1, pages 9–30, 2010.
- [138] Keiki Takadama, Tetsuro Kawai, and Yuhsuke Koyama. Can agents acquire human-like behaviors in a sequential bargaining game? - comparison of roth’s and q-learning agents. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4442 LNAI:156–171, 2007.
- [139] Keiki Takadama, Tetsuro Kawai, and Yuhsuke Koyama. Micro- and macro-level validation in agent-based simulation: Reproduction of human-like behaviors and thinking in a sequential bargaining game. *JASSS*, 11(2), 2008.
- [140] Keiki Takadama, Yutaka L. Suematsu, Norikazu Sugimoto, Norberto E. Nawa, and Katsunori Shimohara. Towards verification and validation in multiagent-based systems and simulations: Analyzing different learning bargaining agents, 2003.
- [141] Athina Tellidou and Anastasios Bakirtzis. A q-learning agent-based model for the analysis of the power market dynamics, 2006.
- [142] The Ciw library developers. Ciw: v2.3.1, 2022.
- [143] The Nashpy project developers. Nashpy: v0.0.25, August 2021.
- [144] James Thomas. Ambulances queue at hereford hospital as nhs pressure mounts. *Hereford Times*, 2021.
- [145] Nuffield Trust. A&E waiting times. Available online at “<https://www.nuffieldtrust.org.uk/resource/a-e-waiting-times>” and accessed on 2023-03-10, 2023.
- [146] Sung Sik U. Enumeration for spanning trees and forests of join graphs based on the combinatorial decomposition. *Electron. J. Graph Theory*

- Appl.*, 4(2):171–177, 2016.
- [147] Takeaki Uno. An algorithm for enumerating all directed spanning trees in a directed graph. In *Algorithms and Computation: 7th International Symposium, ISAAC'96 Osaka, Japan, December 16–18, 1996 Proceedings* 7, pages 166–173. Springer, 1996.
- [148] Guido Van Rossum and Fred L Drake Jr. *Python tutorial*, volume 620. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.
- [149] Guido van Rossum, Barry Warsaw, and Nick Coghlan. Style guide for Python code, 2001.
- [150] Ari Veltman and Refael Hassin. Equilibrium in queueing systems with complementary products. *Queueing Systems*, 50(2):325–342, 2005.
- [151] Julie Leanne Vile, Jonathan W Gillard, Paul R Harper, and Vincent A Knight. A queueing theoretic approach to set staffing levels in time-dependent dual-class service systems. *Decision Sciences*, 48(4):766–794, 2017.
- [152] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.
- [153] John Von Neumann and Oskar Morgenstern. Theory of games and economic behavior, 1947.
- [154] Jinting Wang, Zhongbin Wang, Zhe George Zhang, and Fang Wang. Efficiency-quality trade-off in allocating resource to public healthcare systems. *International Journal of Production Research*, pages 1–22, 2021.
- [155] Joel Watson. *Strategy: an introduction to game theory*, volume 139. WW Norton New York, 2002.
- [156] E. Williams, T. Szakmany, I. Spernaes, B. Muthuswamy, and P. Holborn.

- Discrete-event simulation modeling of critical care flow: New hospital, old challenges. *Critical care explorations*, 2(9), 2020.
- [157] Lauren K Williams. The combinatorics of hopping particles and positivity in markov chains. *arXiv preprint arXiv:2202.00214*, 2022.
- [158] Greg Wilson, Dhavide A Aruliah, C Titus Brown, Neil P Chue Hong, Matt Davis, Richard T Guy, Steven HD Haddock, Kathryn D Huff, Ian M Mitchell, Mark D Plumbley, et al. Best practices for scientific computing. *PLoS biology*, 12(1):e1001745, 2014.
- [159] Weigen Yan and Fuji Zhang. Enumeration of spanning trees of graphs with rotational symmetry. *Journal of Combinatorial Theory, Series A*, 118(4):1270–1290, 2011.
- [160] Nazatul Nurlisa Zolkifli, Amir Ngah, and Aziz Deraman. Version control system: A review. *Procedia Computer Science*, 135:408–415, 2018.
- [161] Bin Zou, Maosong Yan, and Guangqian Xie. Comparisons among pricing methods in pool-based electricity market by agent-based simulation part one model and algorithm. *Dianli Xitong Zidonghua/Automation of Electric Power Systems*, 28(15):7–14, 2004.

# Appendix A

## The `ambulance_game` Python library

This chapter of the appendix provides an overview of the `ambulance_game` Python library. This is a library that accompanies all chapters of this thesis and provides the functionality to run the mathematical models and simulations described in the thesis. The library uses the best available tools to ensure the code is correct, readable, well-documented and properly tested. These tools are outlined in Section 1.4.4. The library is also publicly available on GitHub and has been archived on Zenodo [111].

This appendix chapter is structured based on the Diataxis framework [117]. The structure of the chapter is as follows:

- Section A.1 provides instructions on how to install the library.
- Section A.2 a learning-oriented lesson on performing a specific task using the library.
- Section A.3 provides a goal-oriented how-to guide on how to use the library.
- Section A.4 provides some technical descriptions of the library and how to use it.
- Section A.5 provides a discussion of some of the details of the library.

## A.1 Installation

The `ambulance_game` library is published on the Python Package Index (PyPI) and can be installed using the `pip` package manager [50]:

```
$ python -m pip install ambulance_game
```

Alternatively, a development version of the library can be installed from GitHub. The following commands clone the repository, activate the `conda` environment, install the `flit` package manager and install the library in development mode:

```
$ git clone https://github.com/MichalisPanayides/AmbulanceDecisionGame.git
$ cd AmbulanceDecisionGame
$ conda env create --file environment.yml
$ conda activate ambulance_game
$ python -m pip install flit
$ python -m flit install --symlink
```

To make sure that the library is installed correctly, and to check that the tests pass, the following command install the `tox` package manager and runs the tests:

```
$ python -m pip install tox
$ python -m tox
```

## A.2 Tutorial

The following tutorial provides the steps to get the Nash equilibrium of an instance of the game between two hospitals and an ambulance service provider using the `ambulance_game` library. Table A.1 provides the parameters of the game instance.

$\lambda_2$	t	$\hat{P}$	$\alpha$	
8	2	0.7	0.5	

$\lambda_1^A$	$\mu^A$	$C^A$	$N^A$	$M^A$
1	3	2	10	5

$\lambda_1^B$	$\mu^B$	$C^B$	$N^B$	$M^B$
2	1	3	10	5

Table A.1: Parameters of the game

A full description of the parameters of the game can also be found in Section 4.3.1. The code snippet in Listing A.1 defines the parameters of the game instance using Python.

```

>>> lambda_2 = 8
>>> target = 2
>>> alpha = 0.5
>>> p_hat = 0.7
>>>
>>> lambda_1_1 = 1
>>> mu_1 = 3
>>> num_of_servers_1 = 2
>>> system_capacity_1 = 10
>>> buffer_capacity_1 = 5
>>>
>>> lambda_1_2 = 2
>>> mu_2 = 1
>>> num_of_servers_2 = 3
>>> system_capacity_2 = 10
>>> buffer_capacity_2 = 5

```

Code snippet A.1: Python code that defines the parameters.

The arrival rate of type 2 individuals (ambulance patients) is set to `lambda_2 = 8`. The parameters that correspond to the policy imposed to the hospitals are `target = 2` and `p_hat = 0.7`. This means that the hospitals should aim to serve 70% of the patients that arrive at the hospital within 2 time units.

The python code shown in A.2 uses the parameters of the current game to create matrices  $A$ ,  $B$  and  $R$  that represent the payoff matrices of the game and the routing matrix respectively. For more information on the matrices of the game, refer to Section 4.4.3. The code snippet also uses the `nashpy` library [143] to define the game object.

```

>>> import ambulance_game as abg
>>> import numpy as np
>>> import nashpy as nash
>>>
>>> A, B, R = abg.game.get_payoff_matrices(
...     lambda_2=lambda_2,
...     target=target,
...     alpha=alpha,
...     p_hat=p_hat,
...     lambda_1_1=lambda_1_1,
...     lambda_1_2=lambda_1_2,
...     mu_1=mu_1,
...     mu_2=mu_2,
...     num_of_servers_1=num_of_servers_1,
...     num_of_servers_2=num_of_servers_2,
...     system_capacity_1=system_capacity_1,
...     system_capacity_2=system_capacity_2,
...     buffer_capacity_1=buffer_capacity_1,
...     buffer_capacity_2=buffer_capacity_2,
... )
>>> game = nash.Game(A, B)

```

Code snippet A.2: Python code that defines the game instance.



Thus, a Nash equilibrium of the game can be found using the `lemke_howson` function of the `nashpy` library. The Python code shown in Listing A.3 gets a Nash equilibrium of the game instance.

```
>>> strat_1, strat_2 = game.lemke_howson(initial_dropped_label=0)
>>> strat_1
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 1.])
>>> strat_2
array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.])
```

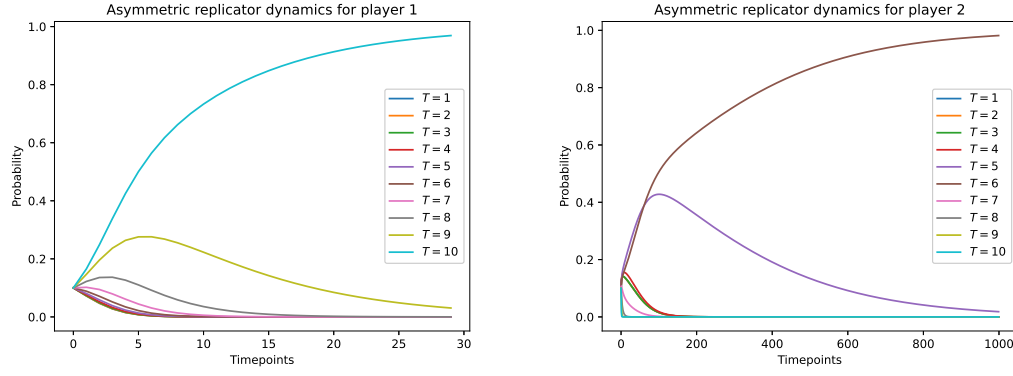
Code snippet A.3: Python code that finds a Nash equilibrium of the game instance.

This corresponds to player 1 playing a strategy of  $\sigma^A = (0, 0, 0, 0, 0, 0, 0, 0, 0, 1)$  and player 2 playing a strategy of  $\sigma^B = (0, 0, 0, 0, 0, 1, 0, 0, 0, 0)$ . This in turn corresponds to player 1 choosing a threshold of  $T^A = 10$  and player 2 choosing a threshold of  $T^B = 6$ .

Apart from the Nash equilibrium, a learning algorithm can also be used to get an evolutionary stable strategy (ESS) of the game. In the code snippet shown in Listing A.4, the results of the asymmetric replicator dynamics algorithm run are shown. The code snippet also uses the `matplotlib` library to plot the results of the algorithm.

```
>>> import matplotlib.pyplot as plt
>>> xs,ys = game.asymmetric_replicator_dynamics(
...     timepoints=np.linspace(0, 10000, 1000)
... )
>>>
>>> plt.plot(xs[0:30])
>>> plt.title("Asymmetric replicator dynamics for player 1")
>>> plt.xlabel("Timepoints")
>>> plt.ylabel("Probability")
>>> plt.show()
>>>
>>> plt.plot(ys)
>>> plt.title("Asymmetric replicator dynamics for player 2")
>>> plt.xlabel("Timepoints")
>>> plt.ylabel("Probability")
>>> plt.show()
```

Code snippet A.4: Python code that runs the asymmetric replicator dynamics algorithm.



From the results of both the Lemke-Howson algorithm and the asymmetric replicator dynamics algorithm, the same pair of strategies is found. That is player 1 choosing a threshold of  $T^A = 10$  and player 2 choosing a threshold of  $T^B = 6$ . The equivalent strategy of the third player; the ambulance service, can be found from the routing matrix  $R$  in position  $R_{10,6}$ .

```
>>> threshold_1, threshold_2 = 10, 6
>>> prop_A = R[threshold_1 - 1, threshold_2 - 1]
>>> prop_B = 1 - prop_A
>>> np.round(prop_A, 2), np.round(prop_B, 2)
(0.95, 0.05)
```

Code snippet A.5: Python code for the strategy of the third player.

In response to the thresholds chosen by the first two players, the ambulance service chooses to send 95% of the ambulances to the first hospital and 5% of the ambulances to the second hospital. Having this percentage of ambulances sent to each hospital, the overall waiting time of patients at each hospital can be calculated. The Python code shown in Listing A.6 calculates the waiting time of patients at each hospital.

```
>>> mean_wait_1 = abg.markov.
    get_mean_waiting_time_using_markov_state_probabilities(
...     lambda_2=lambda_2 * prop_A,
...     lambda_1=lambda_1_1,
...     mu=mu_1,
...     num_of_servers=num_of_servers_1,
...     threshold=threshold_1,
...     system_capacity=system_capacity_1,
...     buffer_capacity=buffer_capacity_1,
... )
>>> mean_wait_2 = abg.markov.
    get_mean_waiting_time_using_markov_state_probabilities(
...     lambda_2=lambda_2 * (1 - prop_A),
...     lambda_1=lambda_1_2,
...     mu=mu_2,
...     num_of_servers=num_of_servers_2,
...     threshold=threshold_2,
...     system_capacity=system_capacity_2,
```

```
...     buffer_capacity=buffer_capacity_2,
... )
>>> np.round(mean_wait_1, 3)
1.257
>>> np.round(mean_wait_2, 3)
0.632
```

Code snippet A.6: Python code for the waiting time of patients at each hospital.

The results of the calculations show that the average waiting time of patients at the first hospital is 1.257 time units and the average waiting time of patients at the second hospital is 0.632 time units. Going back to the parameters that relate to the policy imposed on the hospitals, `target = 2` and `p_hat = 0.7`. As stated earlier, that corresponds to both hospitals serving 70% of the patients within 2 time units. The mean total time of patients is calculated by adding the average waiting time and the average service time of patients.

```
>>> np.round(mean_wait_1 + (1 / mu_1), 3)
1.591
>>> np.round(mean_wait_2 + (1 / mu_2), 3)
1.632
```

Code snippet A.7: Python code for the mean total time of patients at each hospital.

The first mean time in hospital 1 for patients is 1.591 time units and the mean time in hospital 2 for patients is 1.632 time units.

## A.3 How-to guides

This section contains a series of how-to guides that explain how to use the library. This section contains four subsections that explain how to use the library for the following purposes:

- Simulating the hospital using discrete event simulation.
- Getting the Markov chain representation of the hospital.
- Creating an object oriented implementation of the functionality of both the discrete event simulation and the Markov chain representation.
- Creating a game theoretic model of the hospital.

### A.3.1 Discrete Event Simulation

For the purposes of this study, a discrete event simulation (DES) model was constructed that is also described in Section 3.2. The queueing model was built in python using the Ciw library [142].

The same performance measures described in Section 3.4.1, Section 3.4.2 and Section 3.4.3 can also be calculated using the DES model. The simulation can be ran a number of times to eliminate stochasticity and the outcomes of the two methods can be directly comparable.

The DES representation of the hospital network is a discrete event simulation that is implemented using the `ciw` library [142]. The required arguments that need to be passed to the `simulate_model` function are the following:

- `lambda_1` ( $\lambda_1$ ): The arrival rate of class 1 individuals.
- `lambda_2` ( $\lambda_2$ ): The arrival rate of class 2 individuals.
- `mu` ( $\mu$ ): The service rate of the servers.
- `num_of_servers` ( $C$ ): The number of servers in the system.
- `threshold` ( $T$ ): The threshold that indicates when to start blocking type 2 individuals.

```
>>> lambda_1 = 3
>>> lambda_2 = 2
>>> mu = 1
>>> num_of_servers = 6
>>> threshold = 10
```

To get the simulation object with all the data records, the following code can be used:

```
>>> import ambulance_game as abg
>>> import numpy as np
>>> simulation = abg.simulation.simulate_model(
...     lambda_1=lambda_1,
...     lambda_2=lambda_2,
...     mu=mu,
...     num_of_servers=num_of_servers,
...     threshold=threshold,
...     seed_num=0,
... )
>>> simulation.get_all_records()[4]
Record(id_number=2, customer_class=0, node=2, arrival_date=0.5727571550618586,
       waiting_time=0.0, service_start_date=0.5727571550618586, service_time
       =0.7159547497671506, service_end_date=1.2887119048290092, time_blocked=0.0,
       exit_date=1.2887119048290092, destination=-1, queue_size_at_arrival=1,
       queue_size_at_departure=3)
```

Additional arguments that can be passed to the function are:

- `system_capacity` ( $N$ ): The maximum number of individuals in waiting zone 1.
- `buffer_capacity`  $M$ : The maximum number of individuals in waiting zone 2.
- `seed_num`: The seed number for the random number generator.
- `runtime`: How long to run the simulation for.

From a single run of the simulation the data records can be used to get the average for certain performance measures. The following code can be used to get the mean waiting time, blocking time, service time and the proportion of individuals within target.

```
>>> records = simulation.get_all_records()
>>> mean_wait = np.mean(
...     [w.waiting_time for w in records]
... )
>>> mean_wait
0.23845862661827116

>>> mean_block = np.mean(
...     [b.time_blocked for b in records]
... )
>>> mean_block
0.08501727452006658

>>> mean_service = np.mean(
...     [s.service_time for s in records]
... )
>>> mean_service
0.7102610863960119

>>> target = 1
>>> proportion_within_target = np.mean(
...     [r.waiting_time + r.service_time <= target for r in records]
... )
>>> proportion_within_target
0.6200119712689545
```

To reduce the effects of stochasticity in the simulation, the simulation can be run numerous times and get the average performance measures out of all the runs.

```
>>> all_simulations = abg.simulation.get_multiple_runs_results(
...     lambda_1=lambda_1,
...     lambda_2=lambda_2,
...     mu=mu,
...     num_of_servers=num_of_servers,
...     threshold=threshold,
...     system_capacity=20,
...     buffer_capacity=10,
```

```

...     seed_num=0,
...     runtime=2000,
...     num_of_trials=10,
...     target=1,
... )
>>> mean_wait = np.mean([
...     np.mean(w.waiting_times) for w in all_simulations
... ])
>>> mean_wait
0.35585979549204577

>>> mean_service = np.mean([
...     np.mean(s.service_times) for s in all_simulations
... ])
>>> mean_service
1.002184850213415

>>> mean_block = np.mean([
...     np.mean(b.blocking_times) for b in all_simulations
... ])
>>> mean_block
0.3976966024549059

>>> mean_prop = np.mean([
...     p.proportion_within_target for p in all_simulations
... ])
>>> mean_prop
0.45785790578122043

```

To get the steady state probabilities of the model based on the simulation the following code can be used:

```

>>> import numpy as np
>>> import ambulance_game as abg
>>> simulation_object = abg.simulation.simulate_model(
...     lambda_1=1,
...     lambda_2=2,
...     mu=2,
...     num_of_servers=2,
...     threshold=3,
...     system_capacity=4,
...     buffer_capacity=2,
...     seed_num=0,
...     runtime=2000,
... )
>>> probs = abg.simulation.get_simulated_state_probabilities(
...     simulation_object=simulation_object,
... )
>>> np.round(probs, decimals=3)
array([[0.166, 0.266, 0.192, 0.147, 0.025],
       [ nan,   nan,   nan, 0.094, 0.024],
       [ nan,   nan,   nan, 0.058, 0.027]])

>>> total = np.nansum(probs)
>>> np.round(total, decimals=5)
1.0

```

Similarly to get the average steady state probabilities over multiple runs, one can use the code in Listing A.8 to get the average steady state probabilities out of multiple runs.

```
>>> import numpy as np
>>> import ambulance_game as abg
>>> probs = abg.simulation.get_average_simulated_state_probabilities(
...     lambda_1=1,
...     lambda_2=2,
...     mu=2,
...     num_of_servers=2,
...     threshold=3,
...     system_capacity=4,
...     buffer_capacity=2,
...     seed_num=0,
...     runtime=2000,
...     num_of_trials=10,
... )
>>> np.round(probs, decimals=3)
array([[0.18 , 0.267, 0.197, 0.144, 0.024],
       [ nan,   nan,   nan, 0.085, 0.022],
       [ nan,   nan,   nan, 0.054, 0.026]])

>>> total = np.nansum(probs)
>>> np.round(total, decimals=5)
1.0
```

Code snippet A.8: The average steady state probabilities of the model based on the simulation.

As an additional feature, the simulation can use a service rate that is state dependent, server dependent or both. This feature was implemented to accompany Chapter 6 of this thesis. The state-dependent service rate is defined as a dictionary with the keys being the states (see Section 6.2). The server-dependent service rate is defined as a dictionary with the keys being the servers (see Section 6.3). The state and server dependent service rate is defined as a dictionary with the keys being the servers and the values being dictionaries with the keys being the states (see Section 6.4). The code snippet in Listing A.9 shows examples of how to define these service rates.

```
>>> state_dependent_service_rate = {
...     (0, 0): np.nan,
...     (0, 1): 0.5,
...     (0, 2): 0.3,
...     (0, 3): 0.2,
...     (1, 3): 0.2,
...     (0, 4): 0.2,
...     (1, 4): 0.4,
... }
>>> server_dependent_service_rate = {
...     1: 0.5,
...     2: 0.3,
... }
>>> state_server_dependent_service_rate = {
```

```

...     1: {
...         (0, 1): 0.5,
...         (0, 2): 0.3,
...         (0, 3): 0.2,
...         (1, 3): 0.2,
...         (0, 4): 0.2,
...         (1, 4): 0.4,
...     },
...     2: {
...         (0, 1): 1.5,
...         (0, 2): 1.3,
...         (0, 3): 1.2,
...         (1, 3): 1.2,
...         (0, 4): 1.2,
...         (1, 4): 1.4,
...     },
... }

```

Code snippet A.9: Examples of how to define the state-dependent, server-dependent and state and server dependent service rates.

Note that for this particular example server 1 is much slower than server 2 for all states. The code shown in A.10 shows how to use the state and server dependent service rate in the simulation. At the end the busy time of the two servers can be compared.

```

>>> simulation_object = abg.simulation.simulate_model(
...     lambda_1=0.2,
...     lambda_2=0.15,
...     mu=state_server_dependent_service_rate,
...     num_of_servers=2,
...     threshold=4,
...     seed_num=0,
...     runtime=100,
... )
>>> servers = simulation_object.nodes[2].servers
>>> [server.busy_time for server in servers]
[32.88159812544271, 8.576662185652019]

```

Code snippet A.10: The simulation using the state and server dependent service rate.

It can be seen that the busy time of server 1 is much higher than the busy time of server 2.

### A.3.2 Markov Chains

This subsection presents a guide on how to use the `ambulance_game` library to create and solve the Markov chain (MC) model of the hospital network discussed in this thesis (see Section 3.3). The parameters used are defined in Listing A.11. For more information on the parameters, refer to Section 3.



```
>>> lambda_1 = 1
>>> lambda_2 = 2
>>> mu = 5
>>>
>>> num_of_servers = 1
>>> threshold = 2
>>> system_capacity = 3
>>> buffer_capacity = 1
```

Code snippet A.11: Python code for the parameters used in the Markov chain model.

This section focuses on the `markov` module of the `ambulance_game` library which focuses on the Markov model. Function `build_states` takes in the `threshold`, the `system_capacity` and the `buffer_capacity` variables and returns a list of all the states of the MC model.

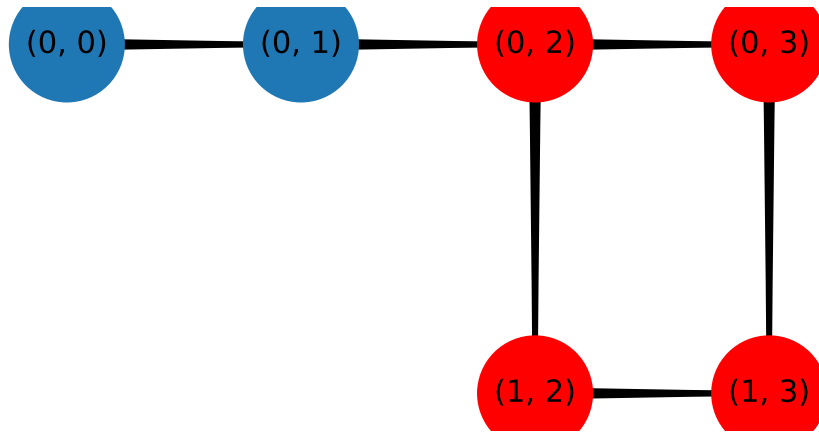
```
>>> import ambulance_game as abg
>>> all_states = abg.markov.build_states(
...     threshold=threshold,
...     system_capacity=system_capacity,
...     buffer_capacity=buffer_capacity,
... )
>>> all_states
[(0, 0), (0, 1), (0, 2), (1, 2), (0, 3), (1, 3)]
```

Code snippet A.12: Python code for building the states of the Markov chain model.

To visualise the list of states that are returned by the `build_states` function, the `visualise_markov_chain` function can be used.

```
>>> abg.markov.visualise_markov_chain(
...     num_of_servers=num_of_servers,
...     threshold=threshold,
...     system_capacity=system_capacity,
...     buffer_capacity=buffer_capacity,
... )
```

Code snippet A.13: Python code for visualising the Markov chain model.



The function `get_transition_matrix` builds the transition matrix of the MC model. The transition matrix is defined in Equation 3.3.

```
>>> Q = abg.markov.get_transition_matrix(
...     lambda_1=lambda_1,
...     lambda_2=lambda_2,
...     mu=mu,
...     num_of_servers=num_of_servers,
...     threshold=threshold,
...     system_capacity=system_capacity,
...     buffer_capacity=buffer_capacity,
... )
>>> Q
array([[ -3.,   3.,   0.,   0.,   0.,   0.],
       [  5.,  -8.,   3.,   0.,   0.,   0.],
       [  0.,   5.,  -8.,   2.,   1.,   0.],
       [  0.,   0.,   5.,  -6.,   0.,   1.],
       [  0.,   0.,   5.,   0.,  -7.,   2.],
       [  0.,   0.,   0.,   5.,   0.,  -5.]])
```

Code snippet A.14: Python code for building the transition matrix of the Markov chain model.

The functions shown in Listing A.15 can be used to calculate the steady state probabilities of the MC model. The steady state probabilities can be calculated using a numerical method or an algebraic method (see Section 3.3.1). For more information on the arguments of these methods, refer to Section A.5.

```

>>> pi = abg.markov.get_steady_state_numerically(Q)
>>> pi
array([0.44853393, 0.26912036, 0.16147222, 0.07381587, 0.02306746,
       0.02399016])

>>> pi = abg.markov.get_steady_state_algebraically(Q)
>>> pi
array([0.44853393, 0.26912036, 0.16147222, 0.07381587, 0.02306746,
       0.02399016])

```

Code snippet A.15: Python code for building the steady state probabilities of the Markov chain model.

The functions shown in Listing A.16 can be used to calculate the expected number of patients in the system, service area and buffer centre. The mathematical formulas for these calculations can be found in Section 3.4.

```

>>> import numpy as np
>>> np.round(
...     abg.markov.get_mean_number_of_individuals_in_system(
...         pi=pi, states=all_states
...     ), 3
... )
0.979

>>> np.round(
...     abg.markov.get_mean_number_of_individuals_in_service_area(
...         pi=pi, states=all_states
...     ), 3
... )
0.881

>>> np.round(
...     abg.markov.get_mean_number_of_individuals_in_buffer_center(
...         pi=pi, states=all_states
...     ), 3
... )
0.098

```

Code snippet A.16: Python code for getting the expected number of patients in the Markov chain model.

To get the mean waiting time of patients in the system, the code snippet shown in Listing A.17 can be used. The waiting time formula can be found in Section 3.4.1.

```

>>> np.round(
...     abg.markov.get_mean_waiting_time_using_markov_state_probabilities(
...         lambda_1=lambda_1,
...         lambda_2=lambda_2,
...         mu=mu,
...         num_of_servers=num_of_servers,
...         threshold=threshold,
...         system_capacity=system_capacity,
...         buffer_capacity=buffer_capacity,
...     ), 4

```

```
... )
0.1195
```

Code snippet A.17: Python code for getting the expected waiting time of individuals in the Markov chain model.

Note that an additional argument `class_type` can be used to get the mean waiting time of type 1 or type 2 individuals that takes values 0 and 1, respectively. The default value of `class_type` is set to `None` which returns the mean waiting time of all individuals.

To get the mean blocking time of type 2 patients in the system (ambulance patients), the code snippet shown in Listing A.18 can be used. The mathematical formula for this calculation can be found in Section 3.4.2.

```
>>> np.round(
...     abg.markov.get_mean_blocking_time_using_markov_state_probabilities(
...         lambda_1=lambda_1,
...         lambda_2=lambda_2,
...         mu=mu,
...         num_of_servers=num_of_servers,
...         threshold=threshold,
...         system_capacity=system_capacity,
...         buffer_capacity=buffer_capacity,
...     ), 4
... )
0.0542
```

Code snippet A.18: Python code for getting the expected blocking time of type 2 individuals in the Markov chain model.

To get the proportion of individuals that are seen within a time target, the code snippet shown in Listing A.19 can be used. The formulas and distributions used for this calculation can be found in Section 3.4.3.

```
>>> np.round(
...     abg.markov.proportion_within_target_using_markov_state_probabilities(
...         lambda_1=lambda_1,
...         lambda_2=lambda_2,
...         mu=mu,
...         num_of_servers=num_of_servers,
...         threshold=threshold,
...         system_capacity=system_capacity,
...         buffer_capacity=buffer_capacity,
...         class_type=None,
...         target=0.5,
...     ), 3
... )
0.791
```

Code snippet A.19: Python code for getting the proportion of individuals within target using the Markov chain model.

### A.3.3 Game Theoretic Model

This section explains how to use the `ambulance_game` library to create a game theoretic model between two hospitals and an ambulance service as described in Chapter 4. The parameters of the model are defined in Listing A.20.

```
>>> lambda_2 = 20
>>> p_hat = 0.99
>>> alpha = 0.7
>>> target = 1
>>>
>>> lambda_1_1 = 2
>>> mu_1 = 5
>>> num_of_servers_1 = 3
>>> threshold_1 = 4
>>> system_capacity_1 = 6
>>> buffer_capacity_1 = 4
>>>
>>> lambda_1_2 = 1
>>> mu_2 = 7
>>> num_of_servers_2 = 2
>>> threshold_2 = 3
>>> system_capacity_2 = 5
>>> buffer_capacity_2 = 4
```

Code snippet A.20: Parameters for the game theoretic model.

The parameters `threshold_1` and `threshold_2` will only be used in code snippet A.21 to determine the best response of the ambulance service, given the strategies of the hospitals. This relates to the concepts described in Section 4.4.

```
>>> import ambulance_game as abg
>>> import numpy as np
>>> best_response = abg.game.calculate_class_2_individuals_best_response(
...     lambda_2=lambda_2,
...     lambda_1_1=lambda_1_1,
...     lambda_1_2=lambda_1_2,
...     mu_1=mu_1,
...     mu_2=mu_2,
...     num_of_servers_1=num_of_servers_1,
...     num_of_servers_2=num_of_servers_2,
...     threshold_1=threshold_1,
...     threshold_2=threshold_2,
...     system_capacity_1=system_capacity_1,
...     system_capacity_2=system_capacity_2,
...     buffer_capacity_1=buffer_capacity_1,
...     buffer_capacity_2=buffer_capacity_2,
... )
>>> np.round(best_response, 3)
0.507
```

Code snippet A.21: Calculating the best response of the ambulance service given the strategies of the hospitals ( $T_1, T_2$ ).

For the particular example the best response of the ambulance service is to send

0.507 individuals to hospital 1 and  $1 - 0.507 = 0.493$  to hospital 2. This can also be done for all possible strategies of the hospitals. The code shown in Listing A.22 calculates the best response of the ambulance service for all possible strategies of the hospitals and stores the results in a `numpy` array. For more information on the routing matrix see Section 4.4.2.

```
>>> R = abg.game.get_routing_matrix(
...     lambda_2=lambda_2,
...     lambda_1_1=lambda_1_1,
...     lambda_1_2=lambda_1_2,
...     mu_1=mu_1,
...     mu_2=mu_2,
...     num_of_servers_1=num_of_servers_1,
...     num_of_servers_2=num_of_servers_2,
...     system_capacity_1=system_capacity_1,
...     system_capacity_2=system_capacity_2,
...     buffer_capacity_1=buffer_capacity_1,
...     buffer_capacity_2=buffer_capacity_2,
...     alpha=alpha
... )
>>> np.round(R, 2)
array([[0.33, 0.18, 0.17, 0.17, 0.15],
       [0.59, 0.38, 0.36, 0.35, 0.32],
       [0.71, 0.5 , 0.48, 0.46, 0.42],
       [0.72, 0.53, 0.5 , 0.48, 0.45],
       [0.73, 0.55, 0.52, 0.5 , 0.47],
       [0.77, 0.6 , 0.57, 0.55, 0.51]])
```

Code snippet A.22: Routing matrix for the ambulance service.

Thus, the ambulance service will send 33% of the individuals to hospital 1 if both hospitals choose a threshold of  $T_i = 1$ , 59% if hospital 1 chooses a threshold of  $T_1 = 2$  and hospital 2 chooses a threshold of  $T_2 = 1$ , and so on. Finally, the code shown in Listing A.23 creates the game using `nashpy` and calculates a Nash equilibrium of the game using the Lemke-Howson algorithm that is implemented in `nashpy`. For more information on the 2-player normal form game that is created see Section 4.4.

```
>>> game = abg.game.build_game_using_payoff_matrices(
...     lambda_2=lambda_2,
...     lambda_1_1=lambda_1_1,
...     lambda_1_2=lambda_1_2,
...     mu_1=mu_1,
...     mu_2=mu_2,
...     num_of_servers_1=num_of_servers_1,
...     num_of_servers_2=num_of_servers_2,
...     system_capacity_1=system_capacity_1,
...     system_capacity_2=system_capacity_2,
...     buffer_capacity_1=buffer_capacity_1,
...     buffer_capacity_2=buffer_capacity_2,
...     target=target,
...     alpha=alpha,
...     p_hat=p_hat,
... )
```

```
>>> game.lemke_howson(initial_dropped_label=0)
(array([0., 0., 0., 0., 1., 0.]), array([0., 0., 0., 0., 1.]))
```

Code snippet A.23: Building the game and getting the Nash equilibrium of the game.

## A.4 Reference

The `ambulance_game` library is structured as shown in Figure A.1.

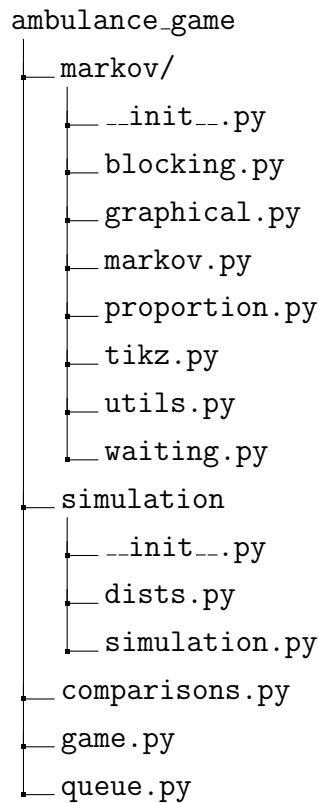


Figure A.1: Structure of the modules in the `ambulance_game` library

Below is a list of all the functions in the `ambulance_game` library sorted by the modules they are in. The entries that start with a capital letter are classes and the rest are functions.

- `blocking.py`
  - `get_coefficients_row_of_array_associated_with_state`
  - `get_blocking_time_linear_system`
  - `convert_solution_to_correct_array_format`
  - `get_blocking_times_of_all_states_using_direct_approach`
  - `mean_blocking_time_formula_using_direct_approach`

- `mean_blocking_time_formula_using_closed_form_approach`
- `get_mean_blocking_time_using_markov_state_probabilities`
- `get_mean_blocking_difference_using_markov`
- `graphical.py`
  - `reset_L_and_R_in_array`
  - `find_next_permutation_over`
  - `find_next_permutation_over_L_and_R`
  - `generate_next_permutation_of_edges`
  - `check_permutation_is_valid`
  - `get_rate_of_state_00_graphically`
  - `get_all_permutations`
  - `get_permutations_ending_in_R`
  - `get_permutations_ending_in_D_where_any_RL_exists`
  - `get_permutations_ending_in_L_where_any_RL_exists`
  - `get_permutations_ending_in_RL_where_RL_exists_only_at_the_end`
  - `get_coefficient`
- `markov.py`
  - `build_states`
  - `visualise_markov_chain`
  - `get_transition_matrix_entry`
  - `get_symbolic_transition_matrix`
  - `get_transition_matrix_by_iterating_through_all_entries`
  - `get_all_pairs_of_states_with_non_zero_entries`
  - `state_after_threshold`
  - `state_before_threshold`
  - `state_at_threshold`
  - `get_transition_matrix`
  - `convert_symbolic_transition_matrix`
  - `is_steady_state`
  - `get_steady_state_numerically`
  - `get_steady_state_algebraically`
  - `get_markov_state_probabilities`



- `get_mean_number_of_individuals_in_system`
- `get_mean_number_of_individuals_in_service_area`
- `get_mean_number_of_individuals_in_buffer_center`
- `proportion.py`
  - `product_of_all_elements`
  - `general_psi_function`
  - `specific_psi_function`
  - `hypoexponential_cdf`
  - `erlang_cdf`
  - `get_probability_of_waiting_time_in_system_less_than_target_for_state`
  - `get_proportion_of_individuals_within_time_target`
  - `overall_proportion_of_individuals_within_time_target`
  - `proportion_within_target_using_markov_state_probabilities`
- `tikz.py`
  - `generate_code_for_tikz_figure`
  - `build_body_of_tikz_spanning_tree`
  - `get_tikz_code_for_permutation`
  - `generate_code_for_tikz_spanning_trees_rooted_at_00`
- `utils.py`
  - `is_waiting_state`
  - `is_blocking_state`
  - `is_accepting_state`
  - `expected_time_in_markov_state_ignoring_arrivals`
  - `expected_time_in_markov_state_ignoring_class_2_arrivals`
  - `prob_service`
  - `prob_class_1_arrival`
  - `get_probability_of_accepting`
  - `get_proportion_of_individuals_not_lost`
  - `get_accepting_proportion_of_class_2_individuals`
  - `get_accepting_proportion_of_individuals`
- `waiting.py`
  - `get_waiting_time_for_each_state_recursively`

- `mean_waiting_time_formula_using_recursive_approach`
  - `get_coefficients_row_of_array_for_state`
  - `get_waiting_time_linear_system`
  - `convert_solution_to_correct_array_format`
  - `get_waiting_times_of_all_states_using_direct_approach`
  - `mean_waiting_time_formula_using_direct_approach`
  - `mean_waiting_time_formula_using_closed_form_approach`
  - `overall_waiting_time_formula`
  - `get_mean_waiting_time_using_markov_state_probabilities`
- `simulation.py`
  - `build_model`
  - `build_custom_node`
  - `simulate_model`
  - `extract_times_from_records`
  - `extract_times_from_individuals`
  - `get_list_of_results`
  - `get_multiple_runs_results`
  - `extract_total_individuals_and_the_ones_within_target_for_both_classes`
  - `get_mean_proportion_of_individuals_within_target_for_multiple_runs`
  - `get_simulated_state_probabilities`
  - `get_average_simulated_state_probabilities`
  - `get_mean_blocking_difference_between_two_systems`
- `dists.py`
  - `StateDependentExponential`
  - `ServerDependentExponential`
  - `StateServerDependentExponential`
  - `is_state_dependent`
  - `is_server_dependent`
  - `is_state_server_dependent`
  - `get_service_distribution`
  - `get_arrival_distribution`
- `comparisons.py`

- `get_heatmaps`
- `get_mean_waiting_time_from_simulation_state_probabilities`
- `get_mean_blocking_time_from_simulation_state_probabilities`
- `get_proportion_within_target_from_simulation_state_probabilities`
- `get_waiting_time_comparisons`
- `get_blocking_time_comparisons`
- `get_proportion_comparison`
- `get_simulation_and_markov_outputs`
- `plot_output_comparisons`
- `game.py`
  - `calculate_class_2_individuals_best_response`
  - `get_routing_matrix`
  - `get_individual_entries_of_matrices`
  - `compute_tasks`
  - `build_matrices_from_computed_tasks`
  - `get_payoff_matrices`
  - `build_game_using_payoff_matrices`
- `queue.py`
  - `Queue`

## A.5 Explanation

This section provides some additional information about the `ambulance_game` library. The information provided in this section is not necessary to use the library, but it may be useful to understand how the library works.

### A.5.1 Additional information

Some of the functions and general functionality of the library has not been explained in the previous sections and are not necessary to use the library.

One of the functions that has not been explained is one that relates to the transition matrix and is the `get_symbolic_transition_matrix` function. This function is part of the `markov.py` module and is used to calculate a symbolic version of the transition matrix of the Markov chain. The function makes use of the `sympy` library [100] to get a symbolic version of the transition matrix where the entries

are symbols. In essence, the function returns a matrix in terms of  $\lambda_1, \lambda_2$  and  $\mu$ . An additional function is provided to convert the symbolic version of the transition matrix to a numerical version of the transition matrix. This function is called `convert_symbolic_transition_matrix`.

Another functionality that has not been explained is the way the transition matrix itself is calculated. Initially, the transition matrix was calculated by iterating through all possible states and calculating the entry of the transition matrix for each state. This method was not computationally efficient and was replaced by a more efficient method. The new method creates a matrix with zeros and visits only the entries that will have a non-zero value. The new method makes use of the function `get_all_pairs_of_states_with_non_zero_entries`. This function corresponds to the function introduced in equation (3.4) of Section 3.3.

The module `tikz.py` has been created for faster creation of Markov chain tikz figures. There are two main functionalities of this module. The first functionality is the ability to generate a tikz figure of the specific Markov chain that is described in this thesis with any set of parameters. This is done by using the function `generate_code_for_tikz_figure`. The second functionality is the ability to generate a tikz figure of all possible spanning trees rooted at state  $(0,0)$  of a Markov chain [90, 146]. This is done by using the function `generate_code_for_tikz_spanning_trees_rooted_at_00`. More information about the investigation between spanning trees and Markov chains that was done in this thesis can be found in appendix D.

### A.5.2 Other libraries

Numerous libraries were used in the construction of this library. Some of the key libraries that were used are:

- `numpy` [65]
- `scipy` [152]
- `ciw` [142]
- `nashpy` [143]

In particular the `numpy` library was used to get the steady state probabilities of the Markov chain algebraically and using the least squares method. Similarly, the `scipy` library was used to get the steady state probabilities of the Markov chain numerically using the `odeint` and `solve_ivp` functions. Apart from that,

the `scipy` library was also used to get the find the best response of the ambulance service using the `brentq` function. The `brentq` function is part of the `optimize` module of the `scipy` library and implements Brent’s algorithm which is a root-finding algorithm.

The primary tool that was used in the construction of the discrete event simulation model was the python library `ciw`. See Ciw’s documentation for a more detailed explanation of how it works and what are its capabilities [142]. The way the library is structured, allowed for the creation of a custom node class that inherits from the `Node` class of `ciw` and was used to create a waiting area that individuals could be blocked in if there were more individuals in the next node.

Finally, the `nashpy` library [143] was used for all the calculations related to the Nash equilibrium and the learning algorithms that were applied to the game. The `nashpy` library is a game theoretic Python library that provides tools for the solution of two-player normal form games. See the documentation of `nashpy` for a more detailed list of the functionality that is provided by the library [143].

## Appendix B

# Game theoretic model - Numerical results

Appendix B contains additional numerical results for the game theoretic model presented in Chapter 4. These results build on the results presented in Chapter 5 and were omitted from the main text for brevity.

This appendix presents additional results of the asymmetric replicator dynamics run and PoA of the game theoretic model with different time targets and multiple values of the “weight” parameter. Refer to Chapter 5 for a description of the asymmetric replicator dynamics run and PoA metrics. Section B.1 shows some runs of the asymmetric replicator dynamics while changing the value of the time target parameter  $t$ . Section B.2 presents a different set of runs of the asymmetric replicator dynamics while changing the value of the “weight” parameter  $\alpha$ . Finally, Section B.3 shows the results of the game theoretic model for some other custom values of the parameters.

## B.1 Changing time targets

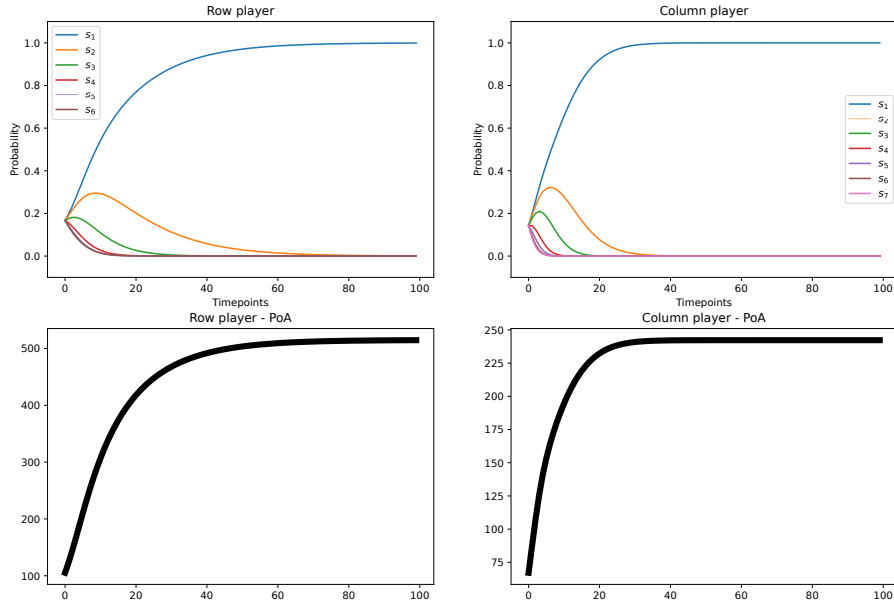


Figure B.1: Asymmetric replicator dynamics run and PoA of the game theoretic model with time target 1.0 and parameters:  $\alpha = 0.97$ ,  $\lambda_2 = 0.1$ ,  $\lambda_1^{(1)} = 3.0$ ,  $\lambda_1^{(2)} = 4.5$ ,  $\mu^{(1)} = 2.0$ ,  $\mu^{(2)} = 3.0$ ,  $C^{(1)} = 3$ ,  $C^{(2)} = 2$ ,  $N^{(1)} = 6$ ,  $N^{(2)} = 7$ ,  $M^{(1)} = 5$ ,  $M^{(2)} = 4$ .

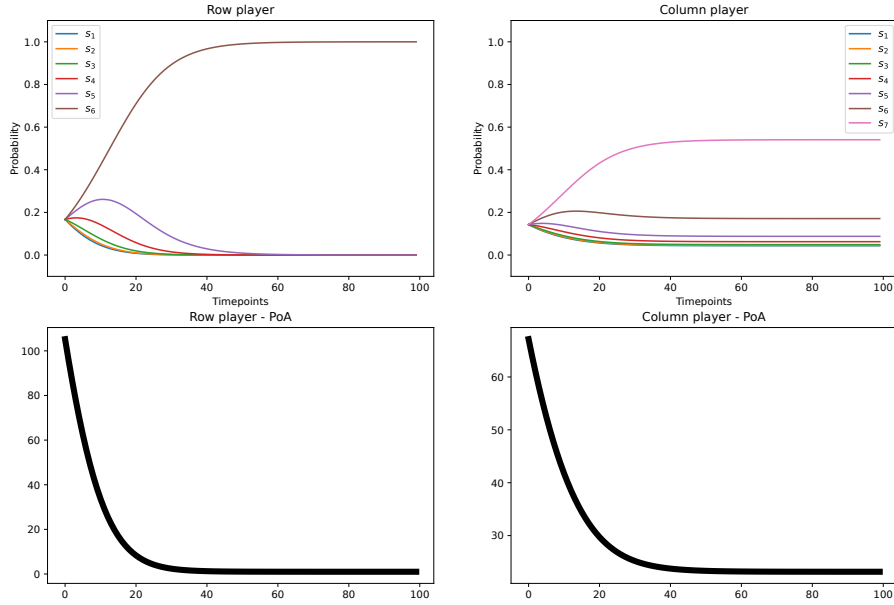


Figure B.2: Asymmetric replicator dynamics run and PoA of the game theoretic model with time target 3.0 and parameters:  $\alpha = 0.97$ ,  $\lambda_2 = 0.1$ ,  $\lambda_1^{(1)} = 3.0$ ,  $\lambda_1^{(2)} = 4.5$ ,  $\mu^{(1)} = 2.0$ ,  $\mu^{(2)} = 3.0$ ,  $C^{(1)} = 3$ ,  $C^{(2)} = 2$ ,  $N^{(1)} = 6$ ,  $N^{(2)} = 7$ ,  $M^{(1)} = 5$ ,  $M^{(2)} = 4$ .

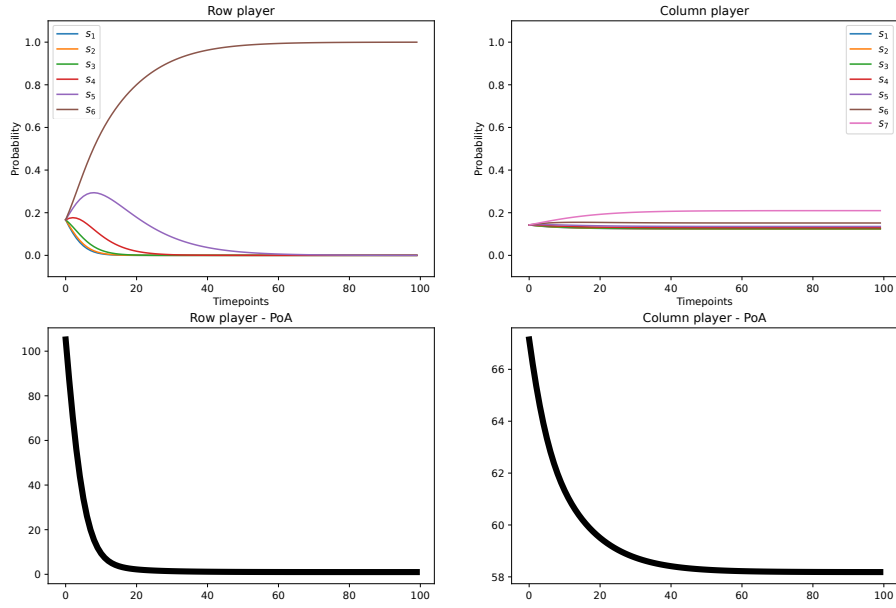


Figure B.3: Asymmetric replicator dynamics run and PoA of the game theoretic model with time target 5.0 and parameters:  $\alpha = 0.97$ ,  $\lambda_2 = 0.1$ ,  $\lambda_1^{(1)} = 3.0$ ,  $\lambda_1^{(2)} = 4.5$ ,  $\mu^{(1)} = 2.0$ ,  $\mu^{(2)} = 3.0$ ,  $C^{(1)} = 3$ ,  $C^{(2)} = 2$ ,  $N^{(1)} = 6$ ,  $N^{(2)} = 7$ ,  $M^{(1)} = 5$ ,  $M^{(2)} = 4$ .

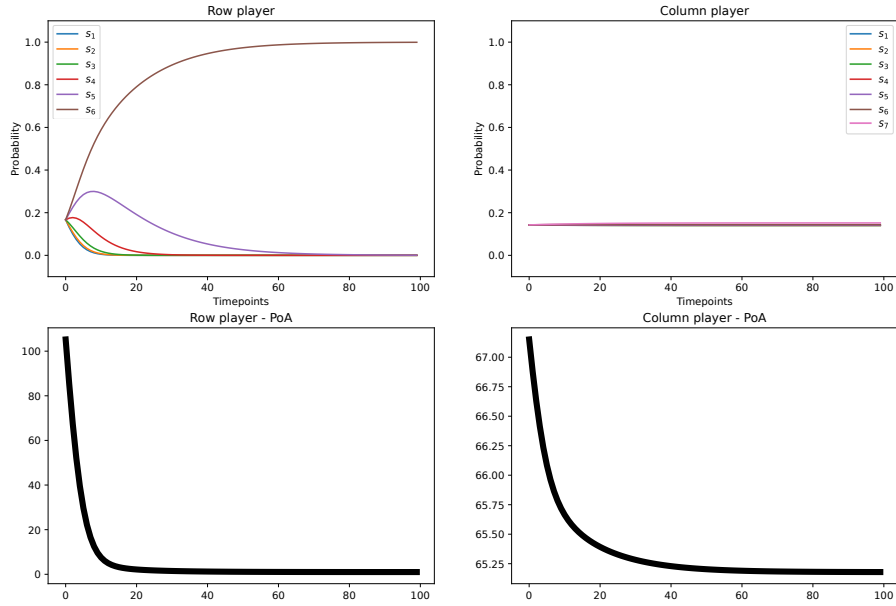


Figure B.4: Asymmetric replicator dynamics run and PoA of the game theoretic model with time target 7.0 and parameters:  $\alpha = 0.97$ ,  $\lambda_2 = 0.1$ ,  $\lambda_1^{(1)} = 3.0$ ,  $\lambda_1^{(2)} = 4.5$ ,  $\mu^{(1)} = 2.0$ ,  $\mu^{(2)} = 3.0$ ,  $C^{(1)} = 3$ ,  $C^{(2)} = 2$ ,  $N^{(1)} = 6$ ,  $N^{(2)} = 7$ ,  $M^{(1)} = 5$ ,  $M^{(2)} = 4$ .



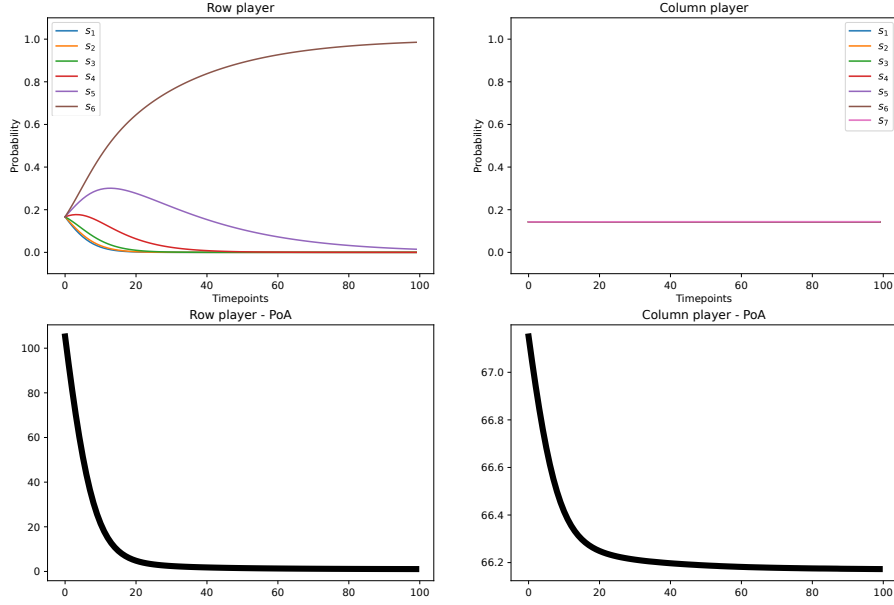


Figure B.5: Asymmetric replicator dynamics run and PoA of the game theoretic model with time target 9.0 and parameters:  $\alpha = 0.97$ ,  $\lambda_2 = 0.1$ ,  $\lambda_1^{(1)} = 3.0$ ,  $\lambda_1^{(2)} = 4.5$ ,  $\mu^{(1)} = 2.0$ ,  $\mu^{(2)} = 3.0$ ,  $C^{(1)} = 3$ ,  $C^{(2)} = 2$ ,  $N^{(1)} = 6$ ,  $N^{(2)} = 7$ ,  $M^{(1)} = 5$ ,  $M^{(2)} = 4$ .

## B.2 Multiple values of “weight” parameter

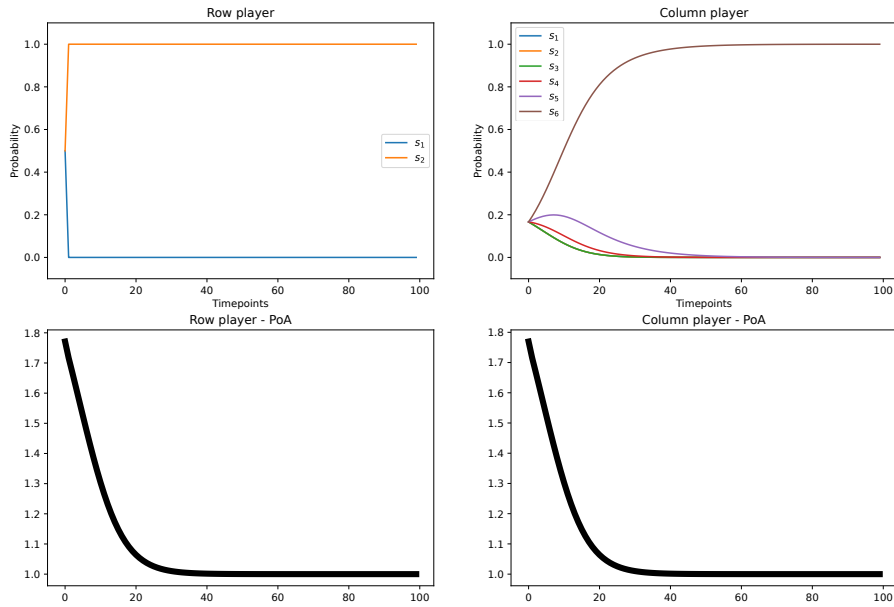


Figure B.6: Asymmetric replicator dynamics run and PoA of the game theoretic model with  $\alpha = 0.0$  and parameters:  $\lambda_2 = 32.05$ ,  $\lambda_1^{(1)} = 0.0$ ,  $\lambda_1^{(2)} = 0.0$ ,  $\mu^{(1)} = 4.2$ ,  $\mu^{(2)} = 6.6$ ,  $C^{(1)} = 1$ ,  $C^{(2)} = 3$ ,  $N^{(1)} = 2$ ,  $N^{(2)} = 6$ ,  $M^{(1)} = 7$ ,  $M^{(2)} = 4$ ,  $t = 2.0$ .

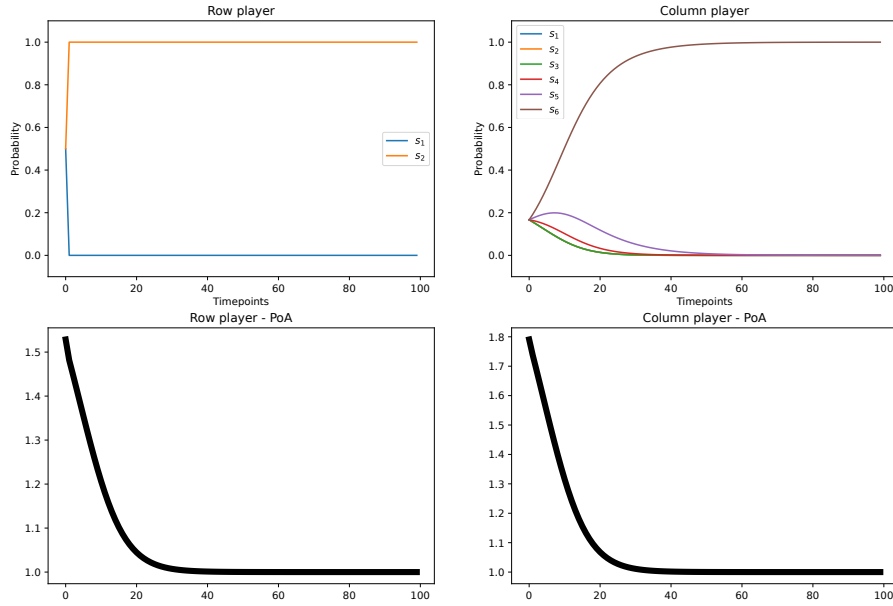


Figure B.7: Asymmetric replicator dynamics run and PoA of the game theoretic model with  $\alpha = 0.3$  and parameters:  $\lambda_2 = 32.05, \lambda_1^{(1)} = 0.0, \lambda_1^{(2)} = 0.0, \mu^{(1)} = 4.2, \mu^{(2)} = 6.6, C^{(1)} = 1, C^{(2)} = 3, N^{(1)} = 2, N^{(2)} = 6, M^{(1)} = 7, M^{(2)} = 4, t = 2.0$ .

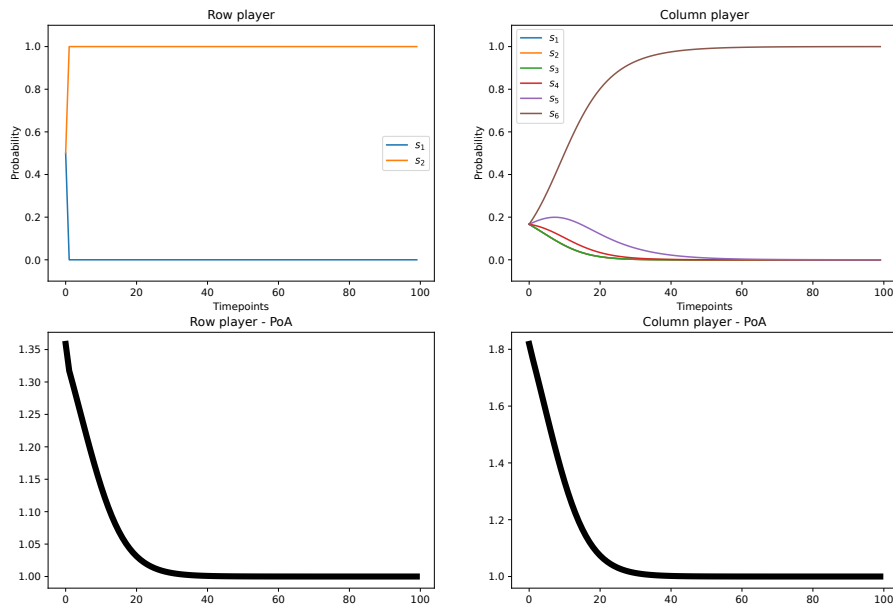


Figure B.8: Asymmetric replicator dynamics run and PoA of the game theoretic model with  $\alpha = 0.6$  and parameters:  $\lambda_2 = 32.05, \lambda_1^{(1)} = 0.0, \lambda_1^{(2)} = 0.0, \mu^{(1)} = 4.2, \mu^{(2)} = 6.6, C^{(1)} = 1, C^{(2)} = 3, N^{(1)} = 2, N^{(2)} = 6, M^{(1)} = 7, M^{(2)} = 4, t = 2.0$ .

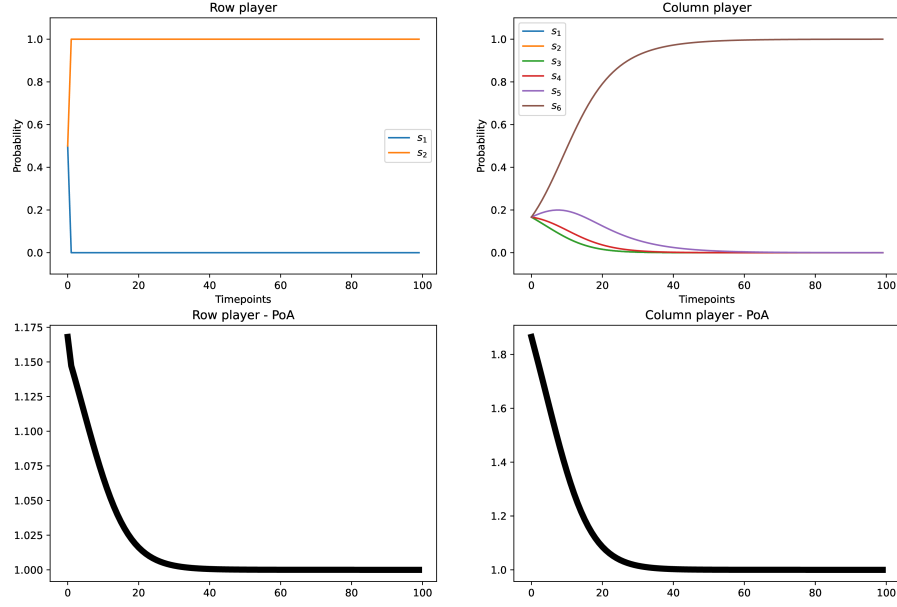


Figure B.9: Asymmetric replicator dynamics run and PoA of the game theoretic model with  $\alpha = 0.9$  and parameters:  $\lambda_2 = 32.05, \lambda_1^{(1)} = 0.0, \lambda_1^{(2)} = 0.0, \mu^{(1)} = 4.2, \mu^{(2)} = 6.6, C^{(1)} = 1, C^{(2)} = 3, N^{(1)} = 2, N^{(2)} = 6, M^{(1)} = 7, M^{(2)} = 4, t = 2.0$ .

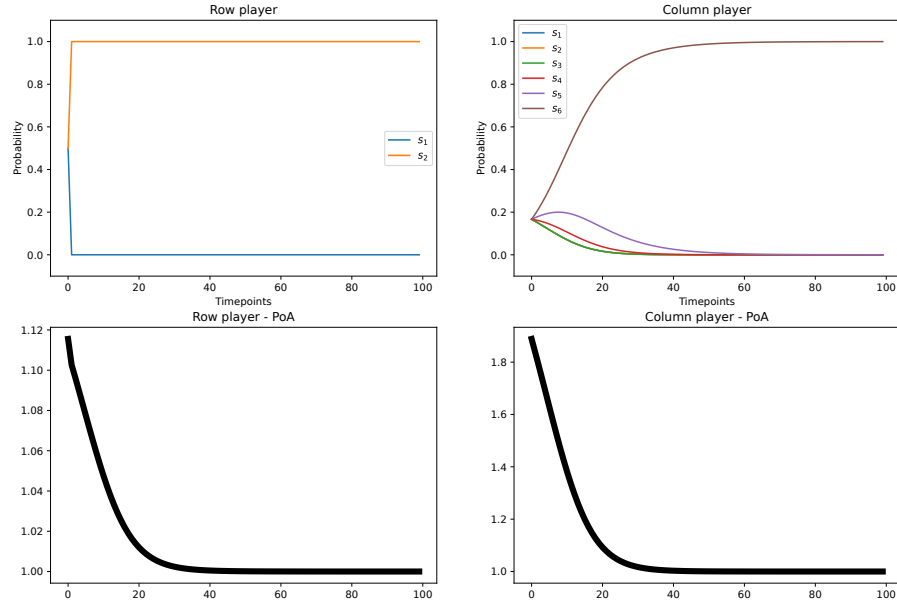


Figure B.10: Asymmetric replicator dynamics run and PoA of the game theoretic model with  $\alpha = 1$  and parameters:  $\lambda_2 = 32.05, \lambda_1^{(1)} = 0.0, \lambda_1^{(2)} = 0.0, \mu^{(1)} = 4.2, \mu^{(2)} = 6.6, C^{(1)} = 1, C^{(2)} = 3, N^{(1)} = 2, N^{(2)} = 6, M^{(1)} = 7, M^{(2)} = 4, t = 2.0$ .

### B.3 Other numerical results

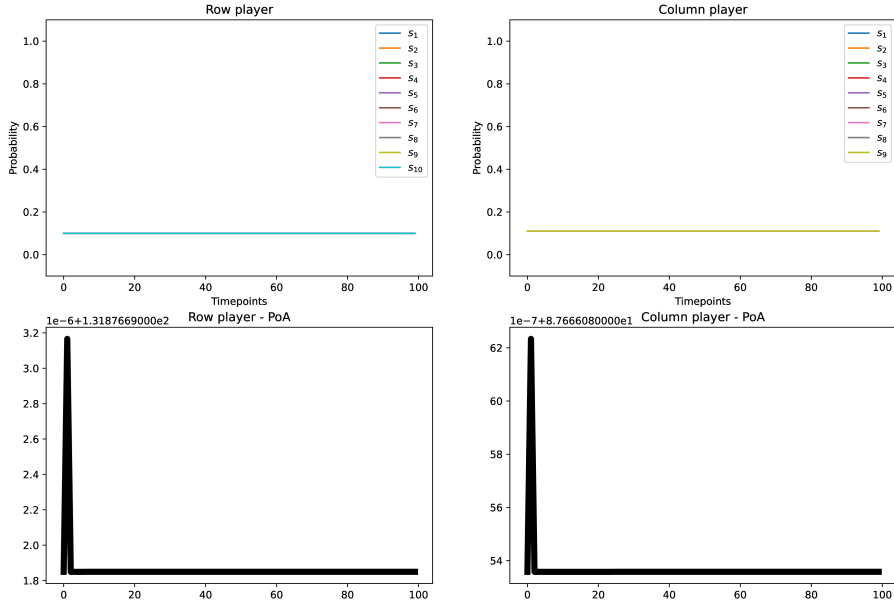


Figure B.11: Asymmetric replicator dynamics run and PoA of the game theoretic model with parameters:  $\alpha = 0.95$ ,  $\lambda_2 = 36.04$ ,  $t = 6.0$ ,  $\lambda_1^{(1)} = 15.24$ ,  $\lambda_1^{(2)} = 0.0$ ,  $\mu^{(1)} = 6.77$ ,  $\mu^{(2)} = 2.22$ ,  $C^{(1)} = 9$ ,  $C^{(2)} = 9$ ,  $N^{(1)} = 10$ ,  $N^{(2)} = 9$ ,  $M^{(1)} = 4$ ,  $M^{(2)} = 3$ .

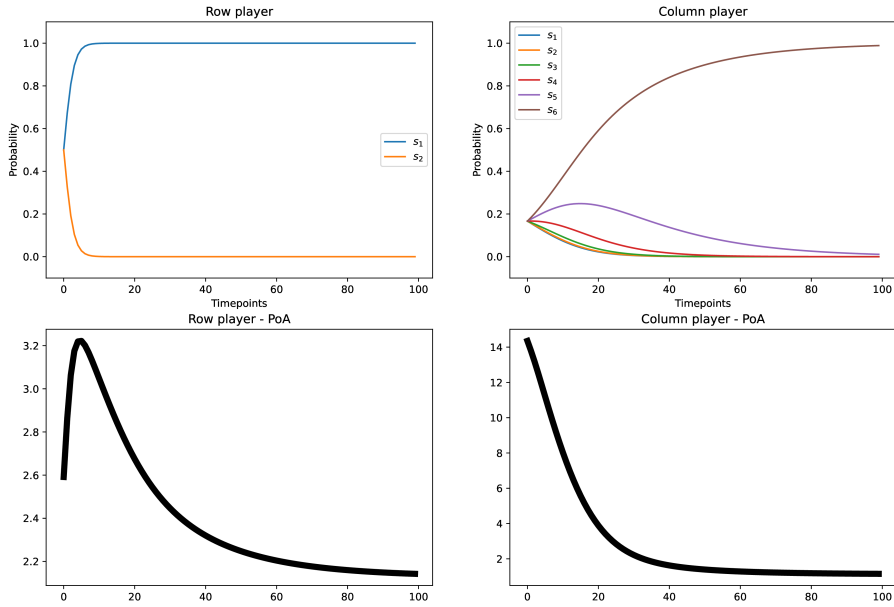


Figure B.12: Asymmetric replicator dynamics run and PoA of the game theoretic model with parameters:  $\alpha = 0.96$ ,  $\lambda_2 = 21.4$ ,  $t = 1.0$ ,  $\lambda_1^{(1)} = 4.2$ ,  $\lambda_1^{(2)} = 19.8$ ,  $\mu^{(1)} = 4.2$ ,  $\mu^{(2)} = 6.6$ ,  $C^{(1)} = 1$ ,  $C^{(2)} = 3$ ,  $N^{(1)} = 2$ ,  $N^{(2)} = 6$ ,  $M^{(1)} = 7$ ,  $M^{(2)} = 4$ .

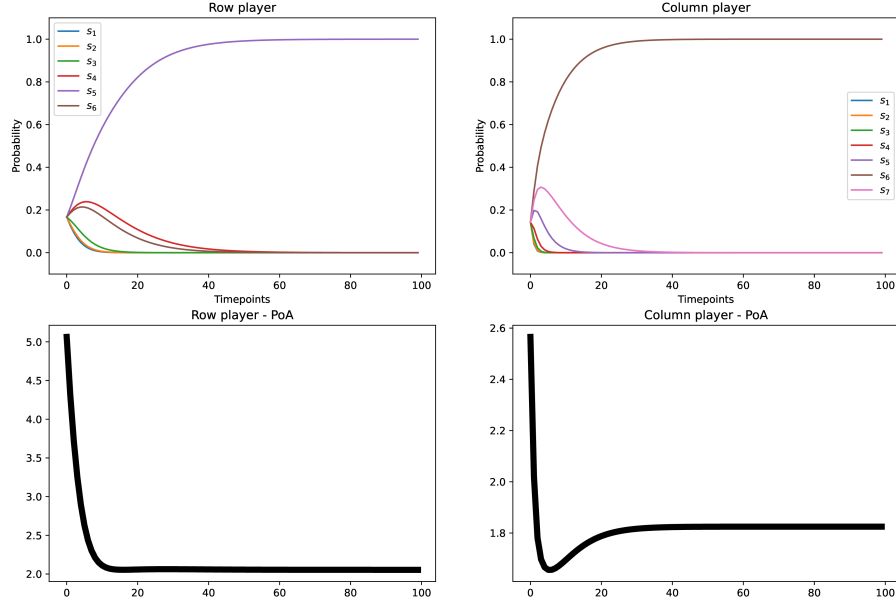


Figure B.13: Asymmetric replicator dynamics run and PoA of the game theoretic model with parameters:  $\alpha = 0.97$ ,  $\lambda_2 = 18.7$ ,  $t = 2.0$ ,  $\lambda_1^{(1)} = 4.5$ ,  $\lambda_1^{(2)} = 3.0$ ,  $\mu^{(1)} = 2.0$ ,  $\mu^{(2)} = 3.0$ ,  $C^{(1)} = 3$ ,  $C^{(2)} = 2$ ,  $N^{(1)} = 6$ ,  $N^{(2)} = 7$ ,  $M^{(1)} = 5$ ,  $M^{(2)} = 4$ .

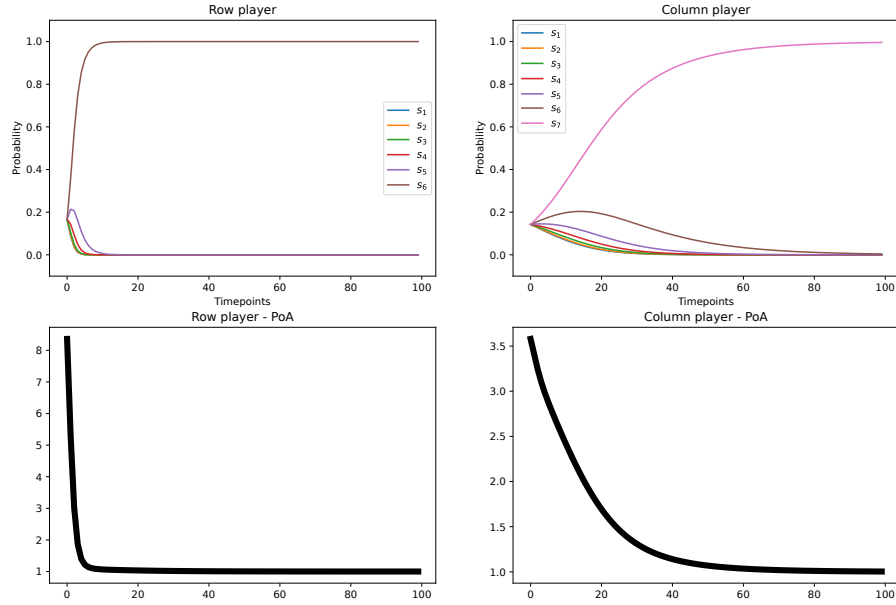


Figure B.14: Asymmetric replicator dynamics run and PoA of the game theoretic model with parameters:  $\alpha = 1.0$ ,  $\lambda_2 = 24.0$ ,  $t = 5.0$ ,  $\lambda_1^{(1)} = 6.0$ ,  $\lambda_1^{(2)} = 4.5$ ,  $\mu^{(1)} = 2.0$ ,  $\mu^{(2)} = 3.0$ ,  $C^{(1)} = 3$ ,  $C^{(2)} = 2$ ,  $N^{(1)} = 6$ ,  $N^{(2)} = 7$ ,  $M^{(1)} = 5$ ,  $M^{(2)} = 4$ .

## Appendix C

# Reinforcement Learning - Numerical results

Appendix C contains additional numerical results for the reinforcement learning algorithm presented in Chapter 6. These results build on the results presented in Section 6.6 and were omitted from the main text because they are not essential for the understanding of the reinforcement learning algorithm.

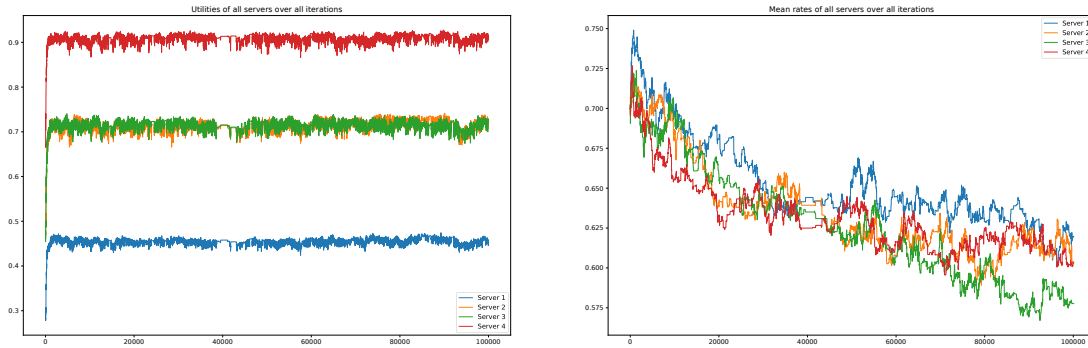


Figure C.1: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(3)}$  with  $e = 0$  and 100,000 time steps

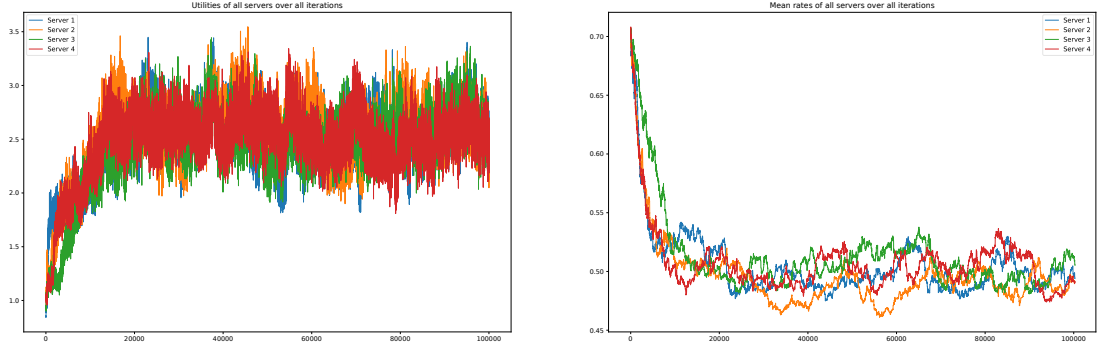


Figure C.2: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(3)}$  with  $e = 0.5$  and 100,000 time steps

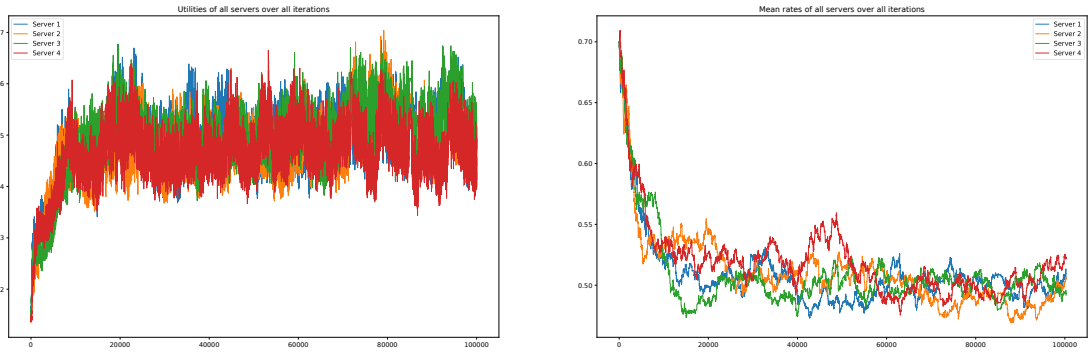


Figure C.3: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(3)}$  with  $e = 1$  and 100,000 time steps

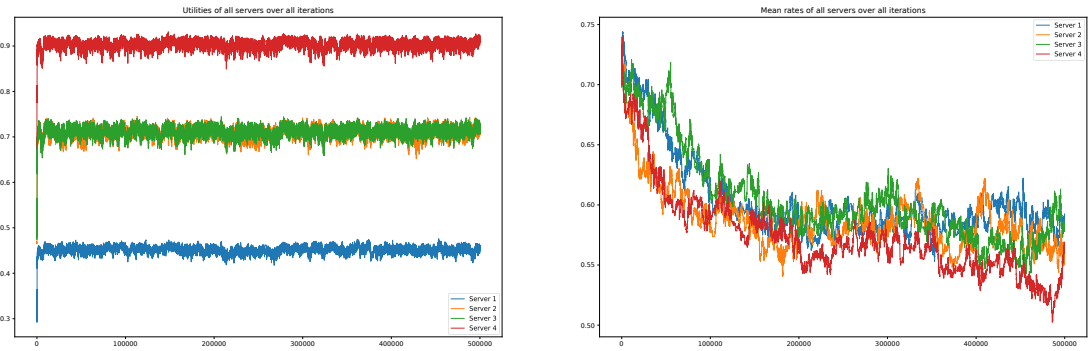


Figure C.4: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(3)}$  with  $e = 0$  and 500,000 time steps

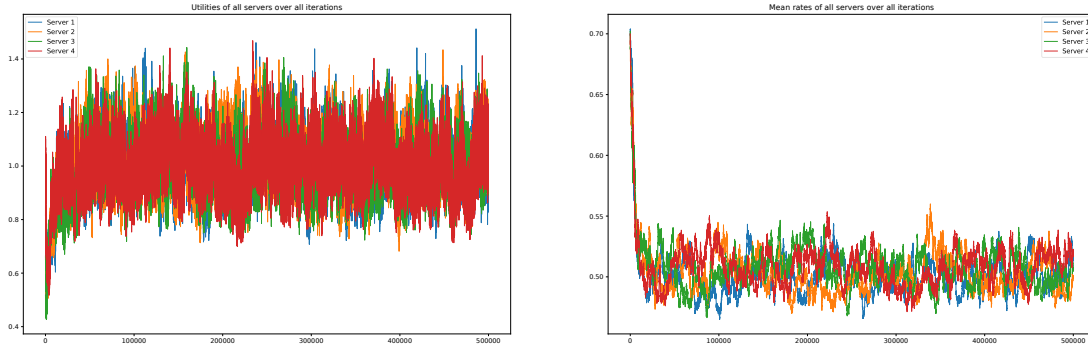


Figure C.5: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(3)}$  with  $e = 0.2$  and 500,000 time steps

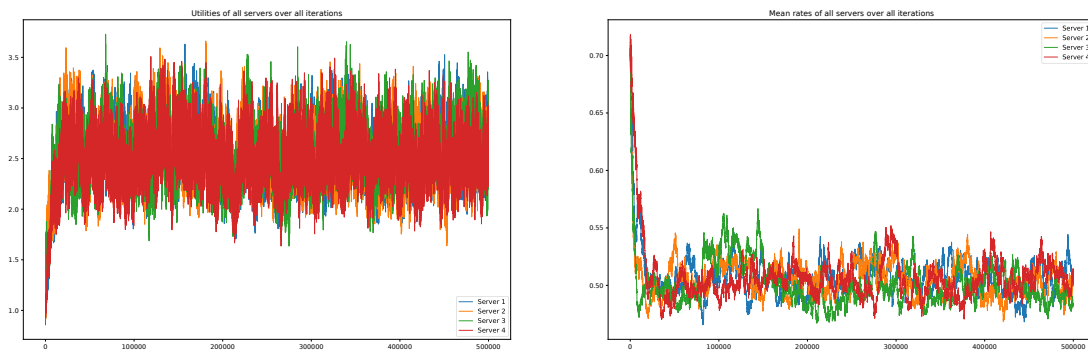


Figure C.6: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(3)}$  with  $e = 0.5$  and 500,000 time steps

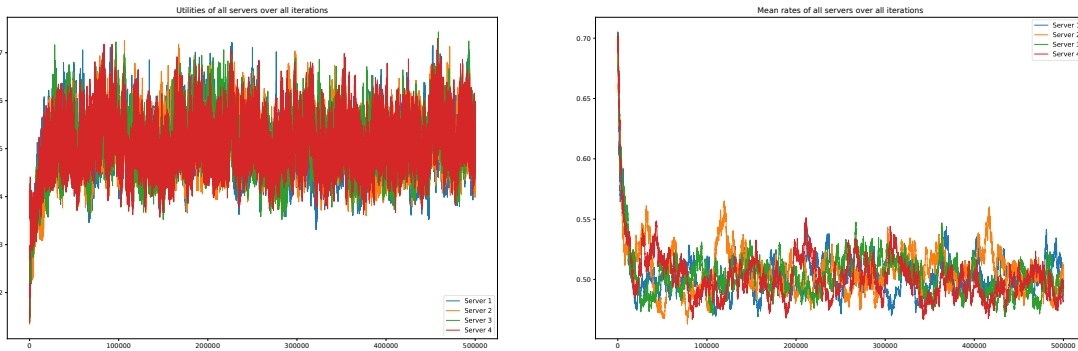


Figure C.7: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(3)}$  with  $e = 1$  and 500,000 time steps



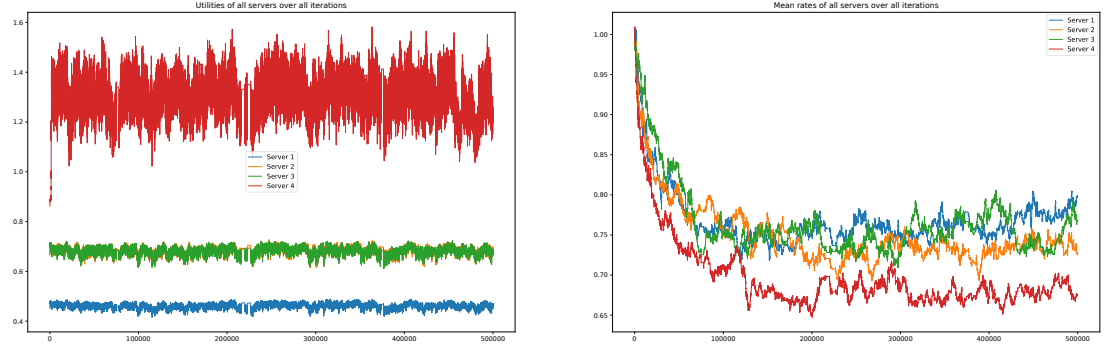


Figure C.8: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(3)}$  with  $e = 0.1$ , 500,000 time steps and an initial service rate of  $\mu = 1$  for all servers

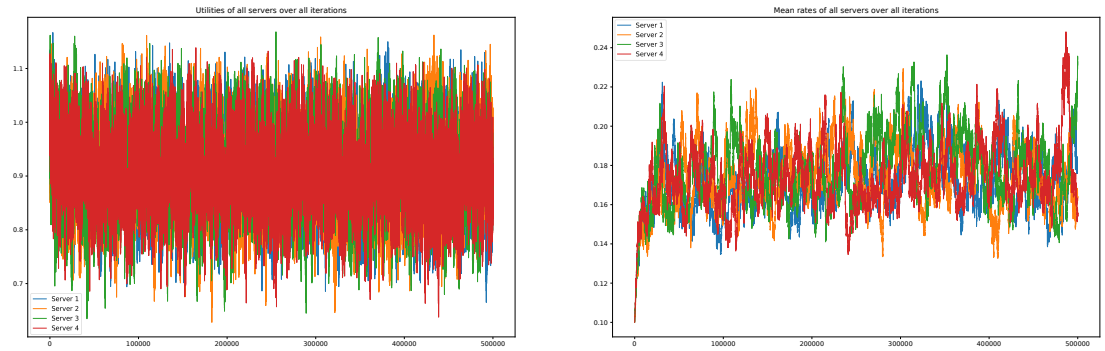


Figure C.9: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(3)}$  with  $e = 0.1$ , 500,000 time steps and an initial service rate of  $\mu = 0.1$  for all servers

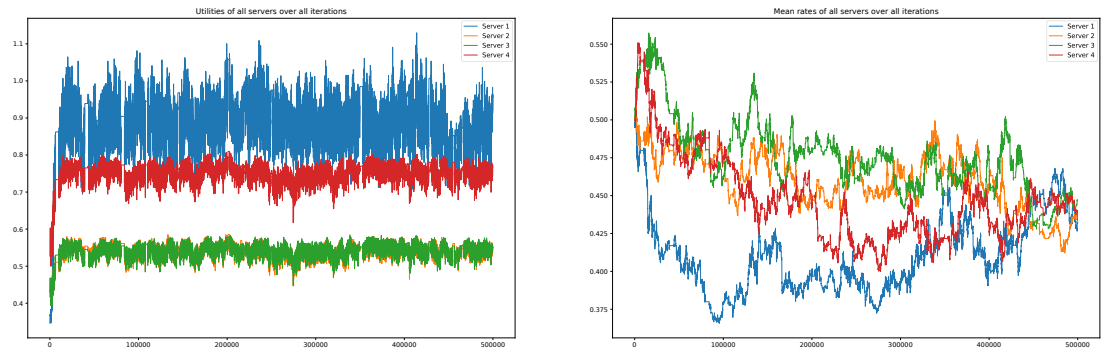


Figure C.10: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(3)}$  with  $e = 0.1$ , 500,000 time steps and an initial service rate of  $\mu = 0.5$  for all servers

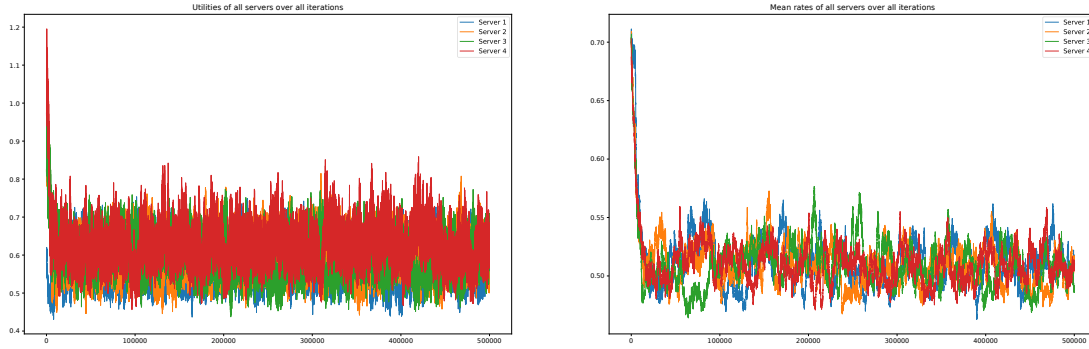


Figure C.11: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(3)}$  with  $e = 1$  and 500,000 time steps decreasing  $\lambda_2$  to 0.5

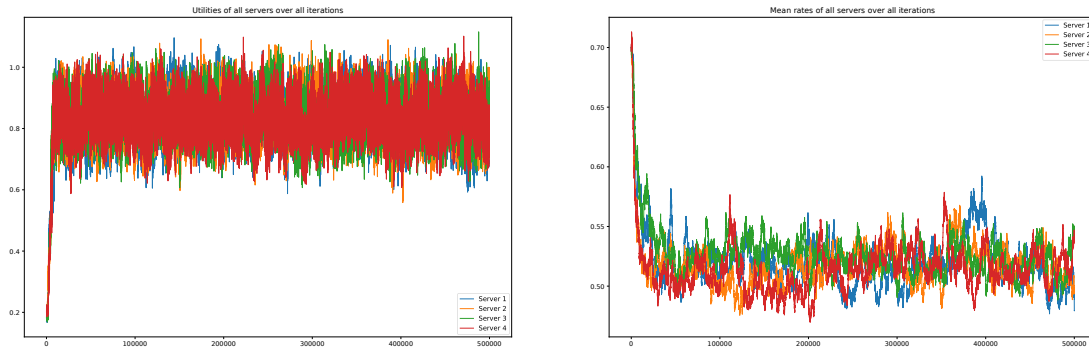


Figure C.12: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(3)}$  with  $e = 1$  and 500,000 time steps increasing  $\lambda_2$  to 1.5

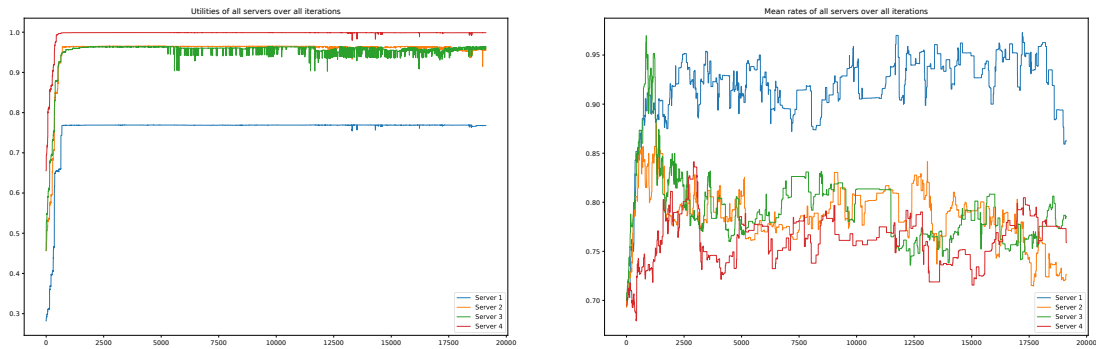


Figure C.13: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0$ .

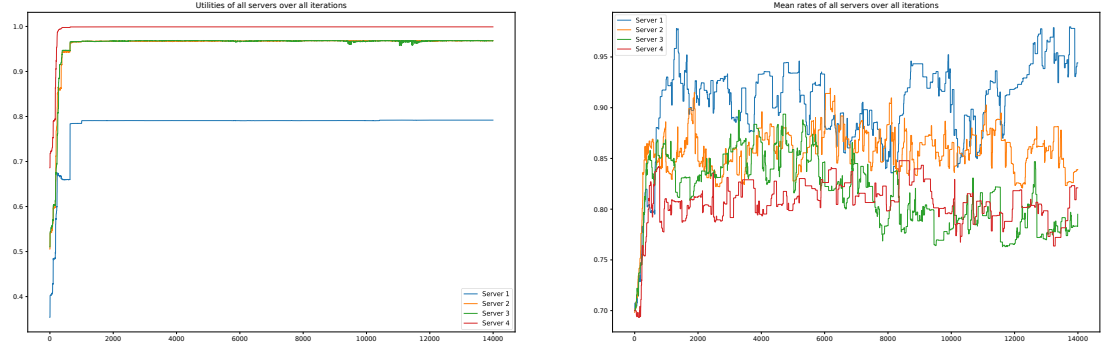


Figure C.14: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.1$ .

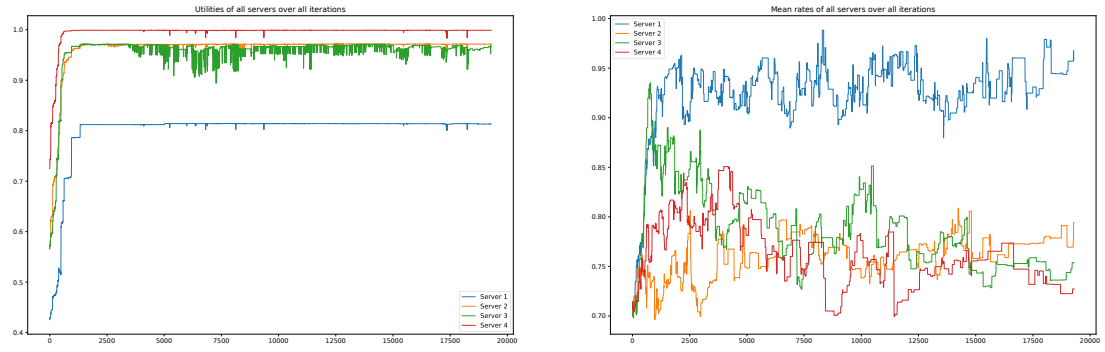


Figure C.15: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.2$ .

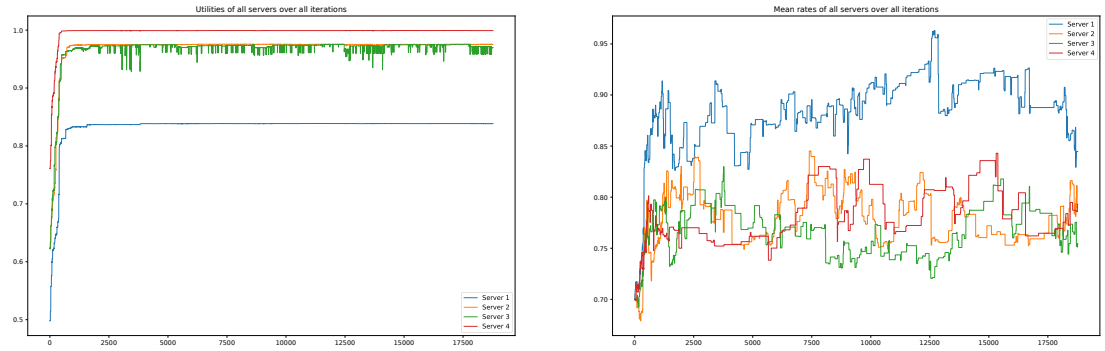


Figure C.16: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.3$ .

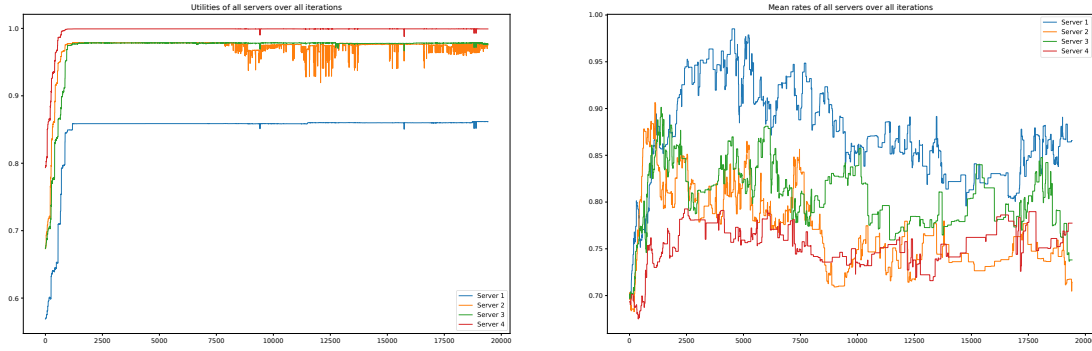


Figure C.17: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.4$ .

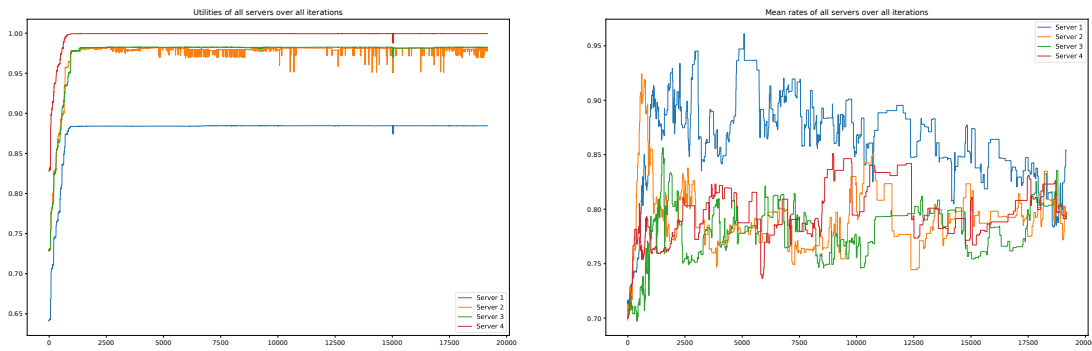


Figure C.18: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.5$ .

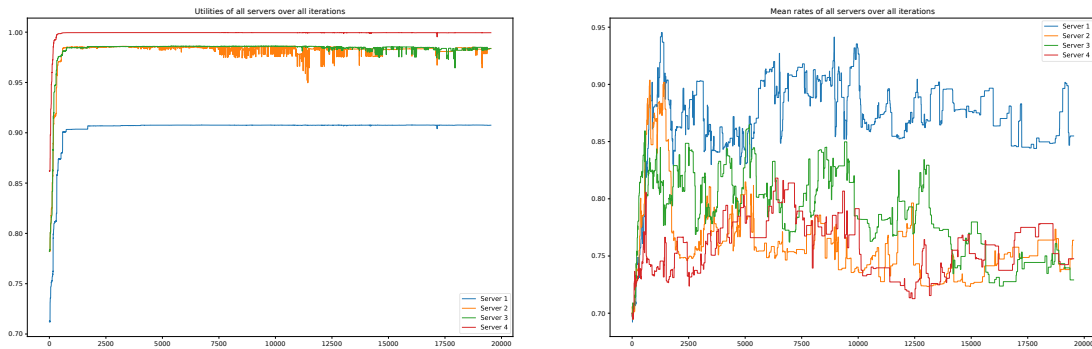


Figure C.19: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.6$ .

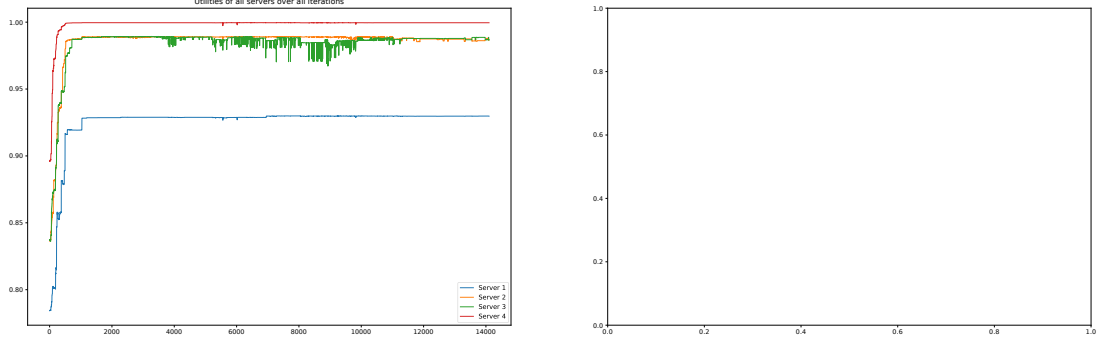


Figure C.20: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.7$ .

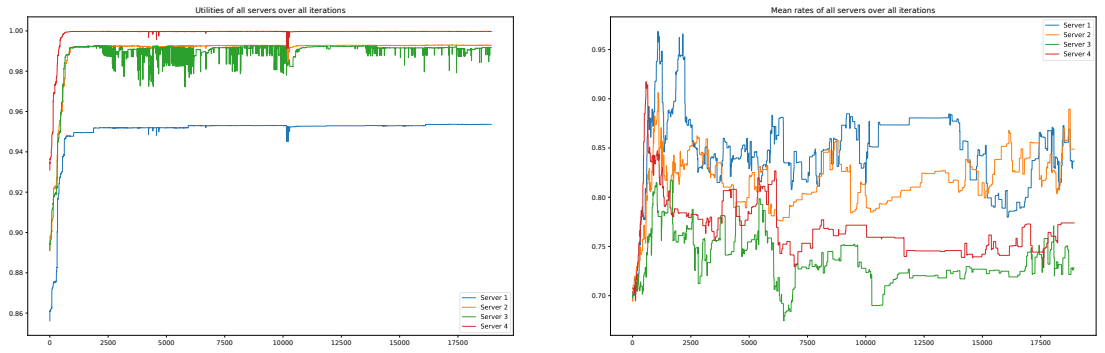


Figure C.21: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.8$ .

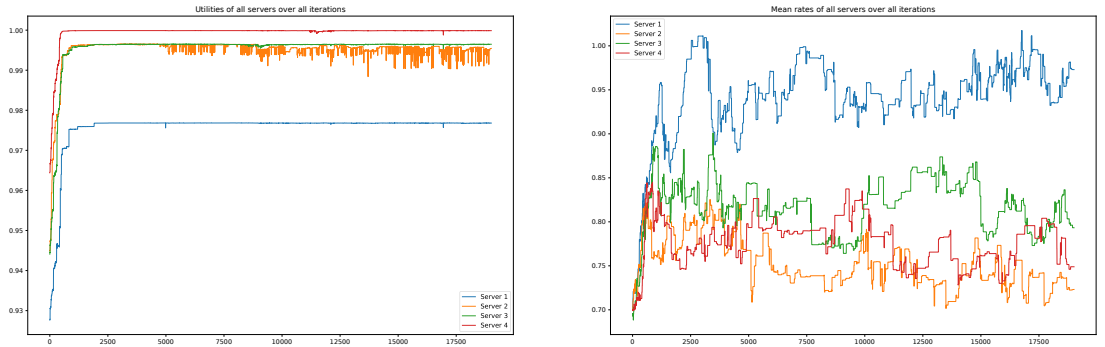


Figure C.22: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.9$ .

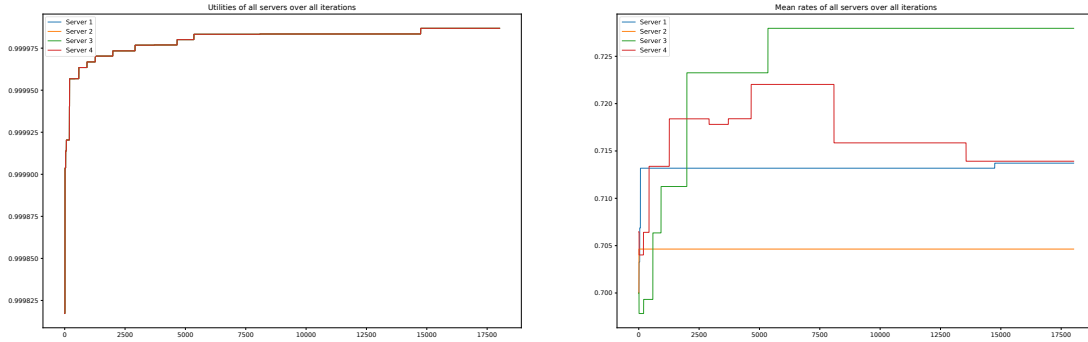


Figure C.23: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 1$ .

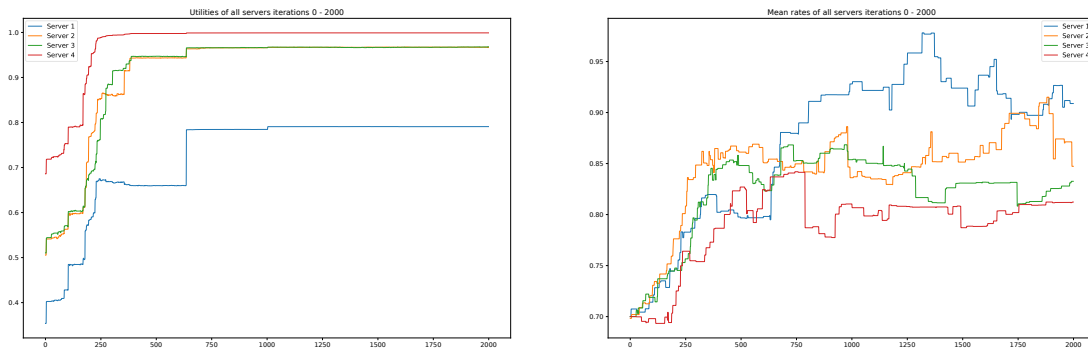


Figure C.24: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.1$  (only the early iterations)

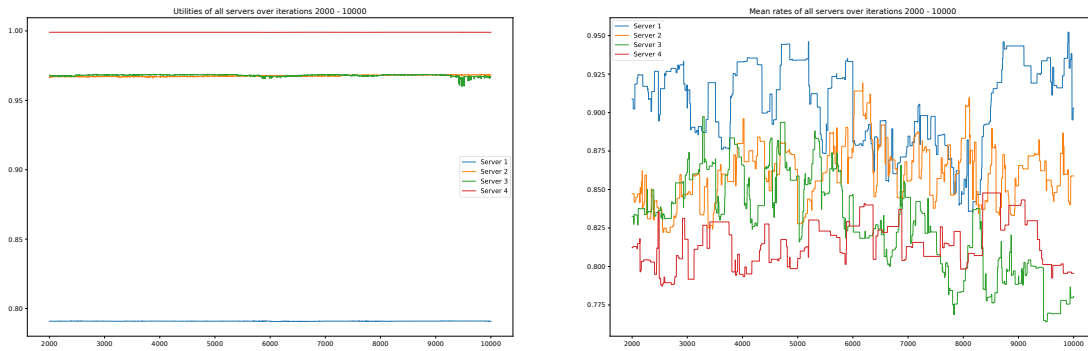


Figure C.25: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.1$  (only the late iterations)

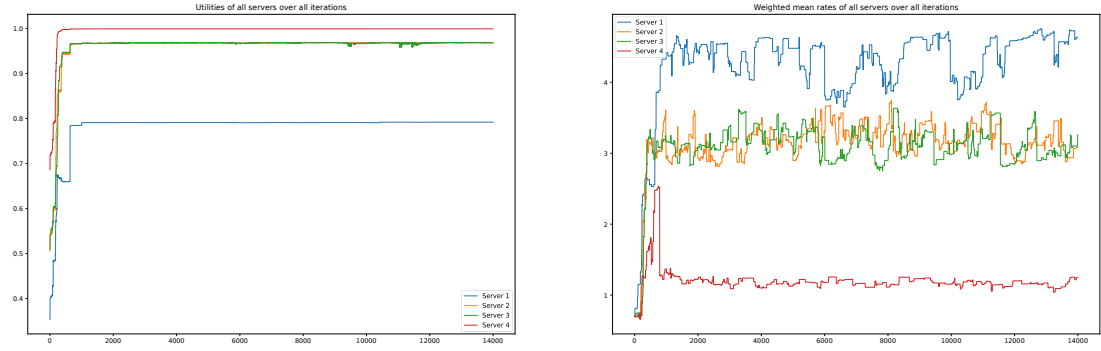


Figure C.26: Utilities and approximation of the weighted mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.1$ . Note that the approximation uses the Markov chain model to get the state probabilities instead of the DES state probabilities.

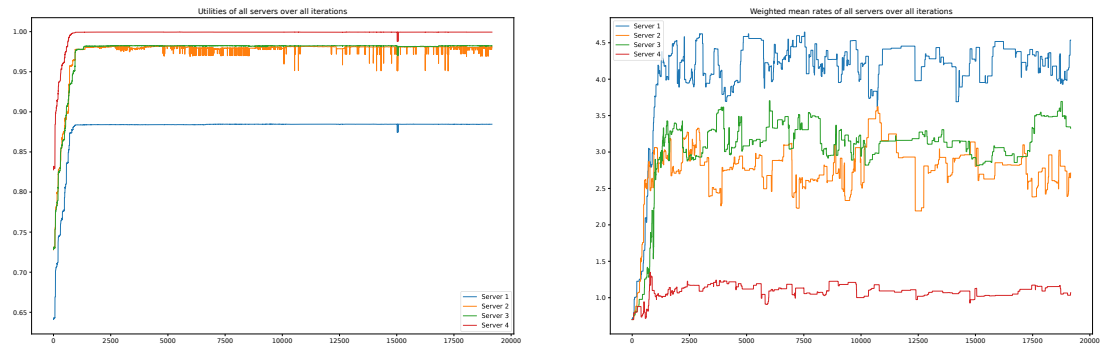


Figure C.27: Utilities and approximation of the weighted mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.5$ . Note that the approximation uses the Markov chain model to get the state probabilities instead of the DES state probabilities.

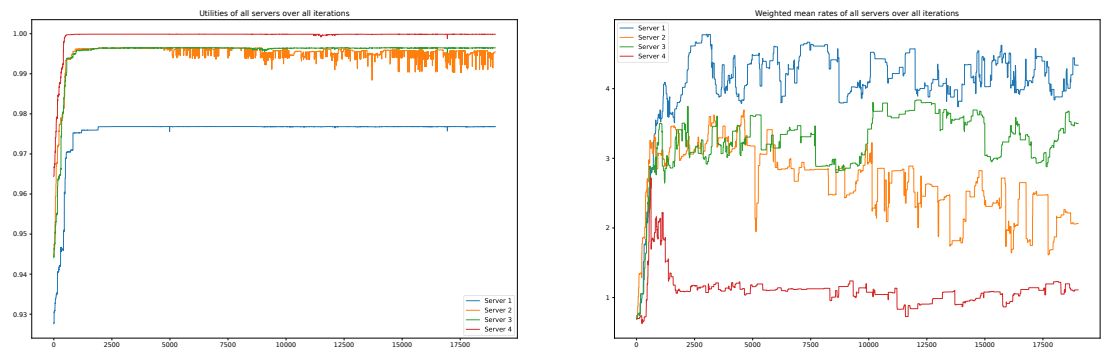


Figure C.28: Utilities and approximation of the weighted mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.9$ . Note that the approximation uses the Markov chain model to get the state probabilities instead of the DES state probabilities.

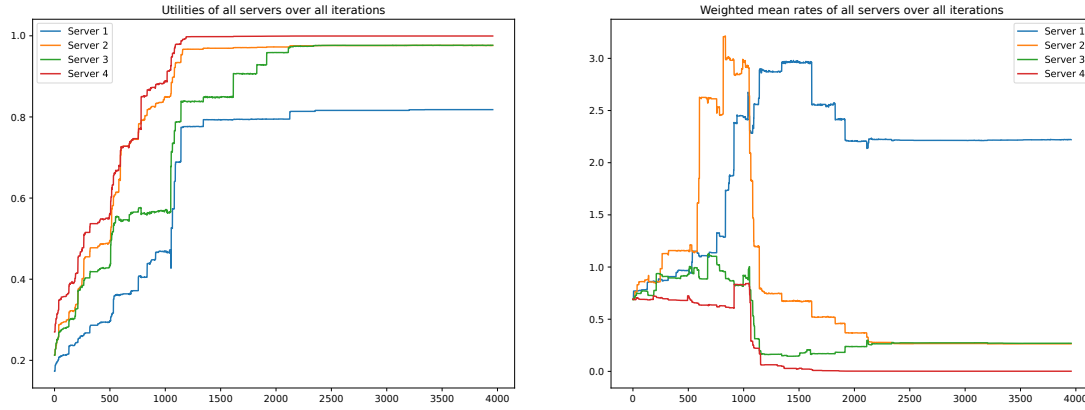


Figure C.29: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.1$  and increased arrival rates of  $\lambda_1 = 1.0$  and  $\lambda_2 = 1.5$ .

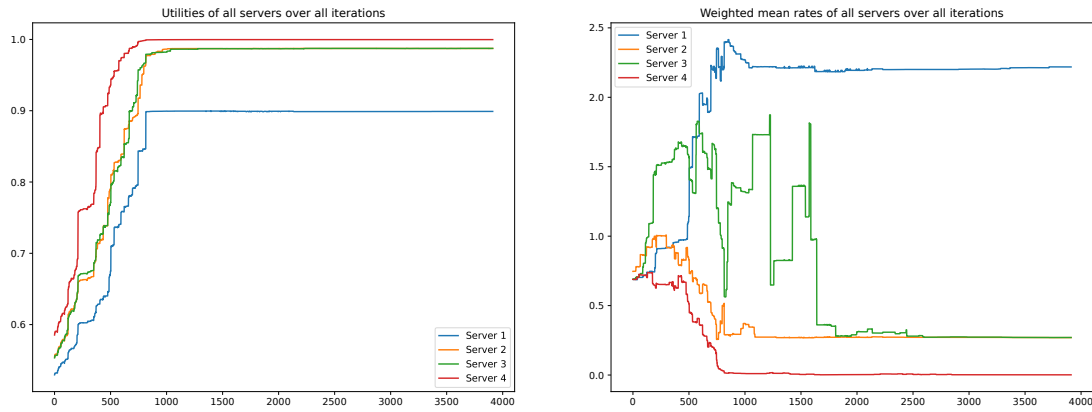


Figure C.30: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.5$  and increased arrival rates of  $\lambda_1 = 1.0$  and  $\lambda_2 = 1.5$ .

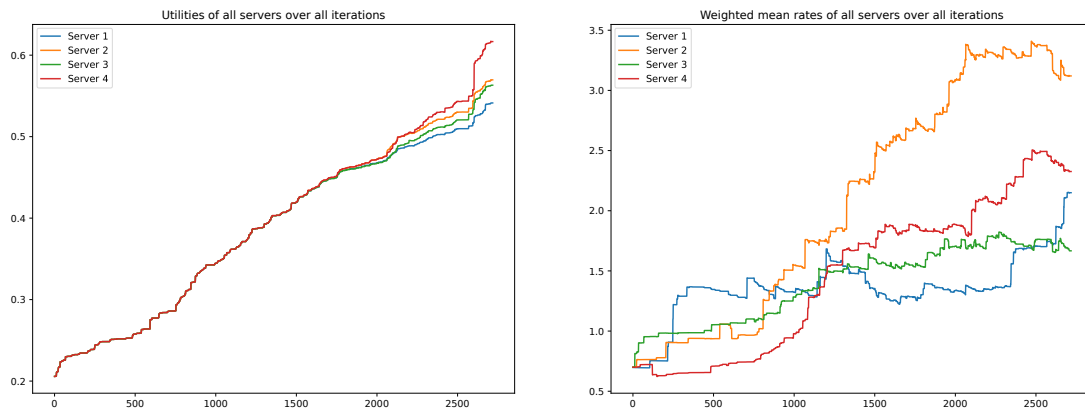


Figure C.31: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.5$  and increased arrival rates of  $\lambda_1 = 3.0$  and  $\lambda_2 = 3.5$ .



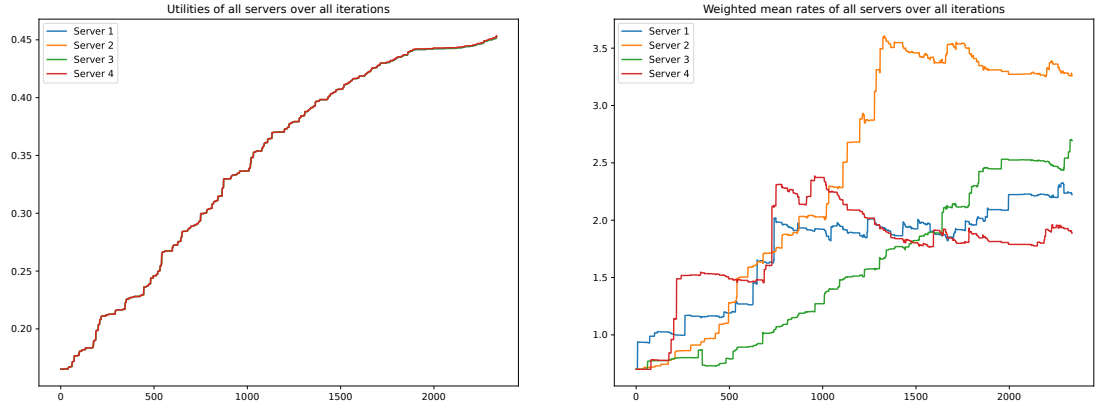


Figure C.32: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.5$  and increased arrival rates of  $\lambda_1 = 4.0$  and  $\lambda_2 = 4.5$ .

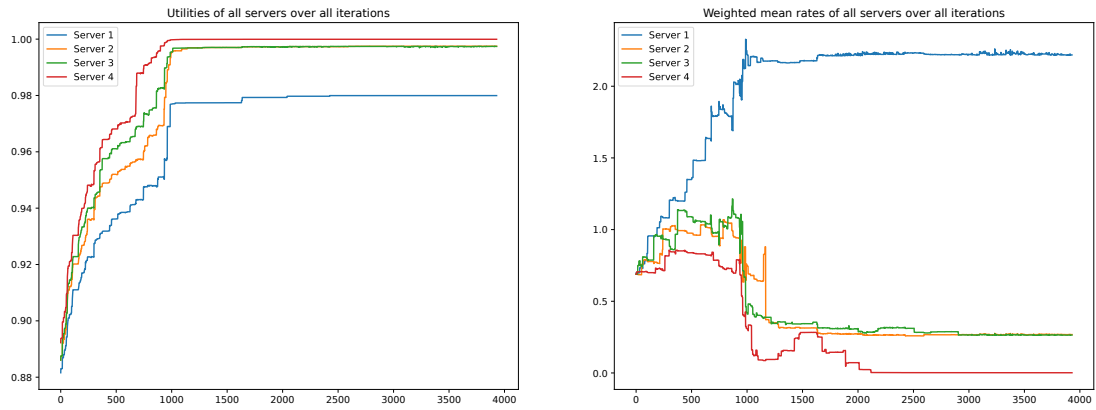


Figure C.33: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.9$  and increased arrival rates of  $\lambda_1 = 1.0$  and  $\lambda_2 = 1.5$ .

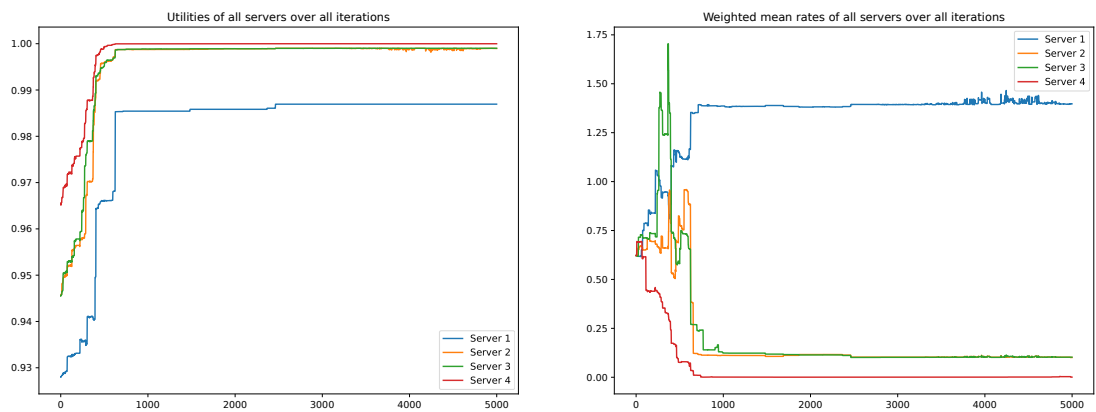


Figure C.34: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.9$ .

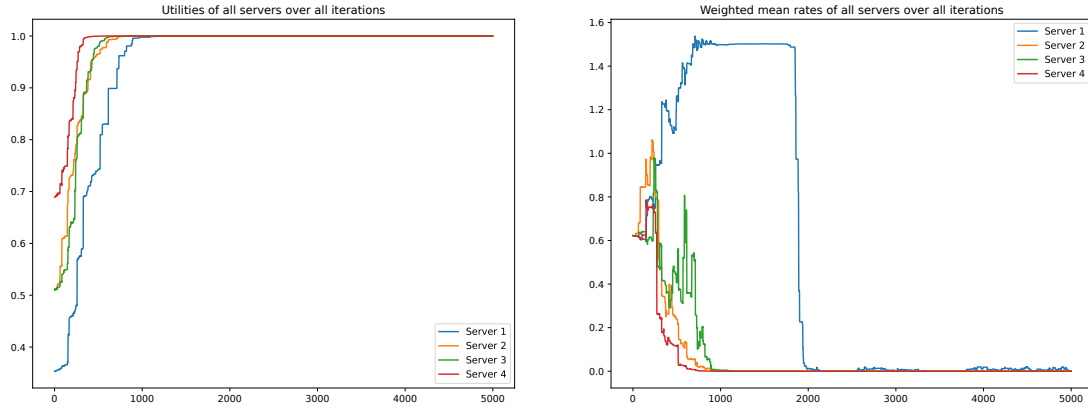


Figure C.35: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.1$  and no upper bound on the service rate.

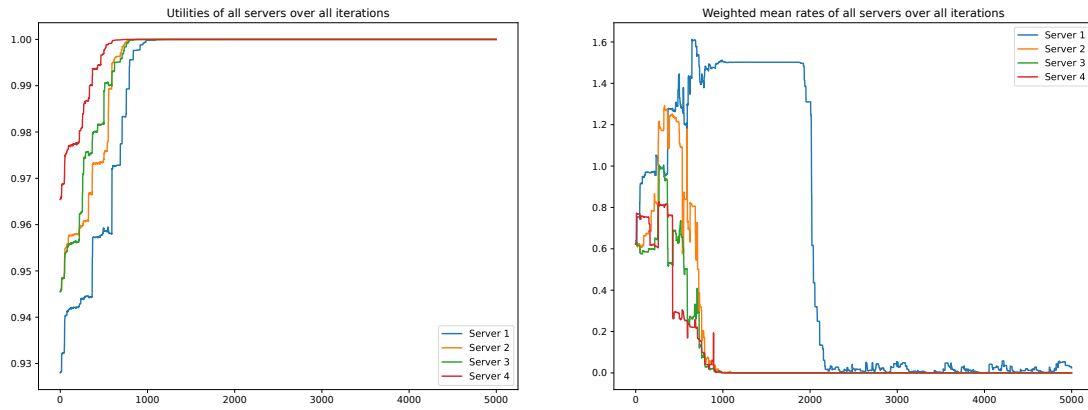


Figure C.36: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.9$  and no upper bound on the service rate.

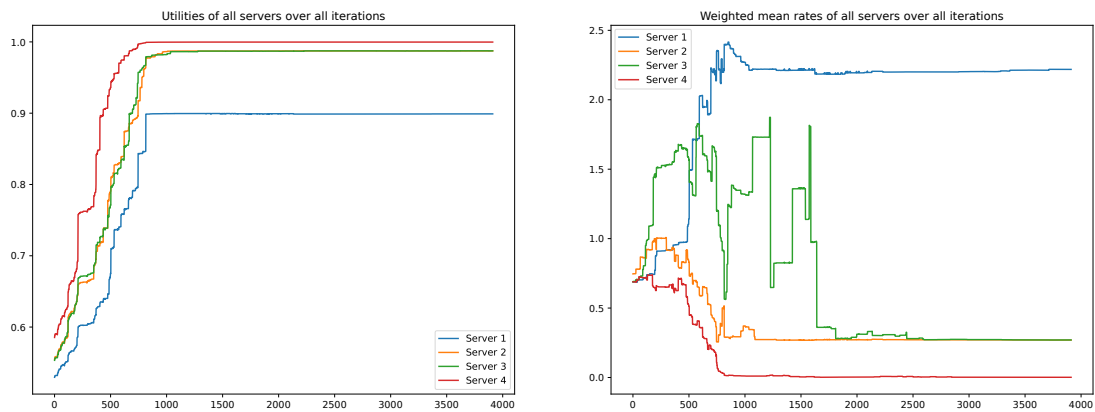


Figure C.37: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.5$  and increased arrival rates of  $\lambda_1 = 1.0$  and  $\lambda_2 = 1.5$ .

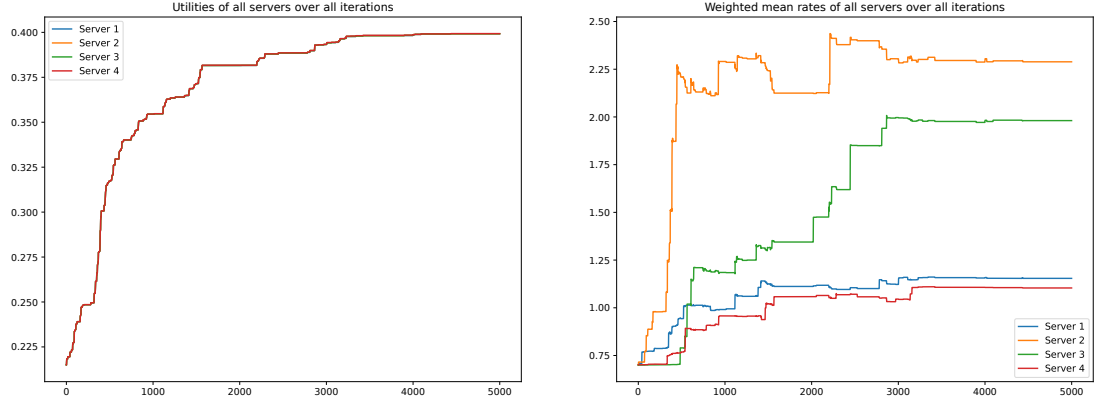


Figure C.38: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.5$  and increased arrival rates of  $\lambda_1 = 3.0$  and  $\lambda_2 = 3.5$ .

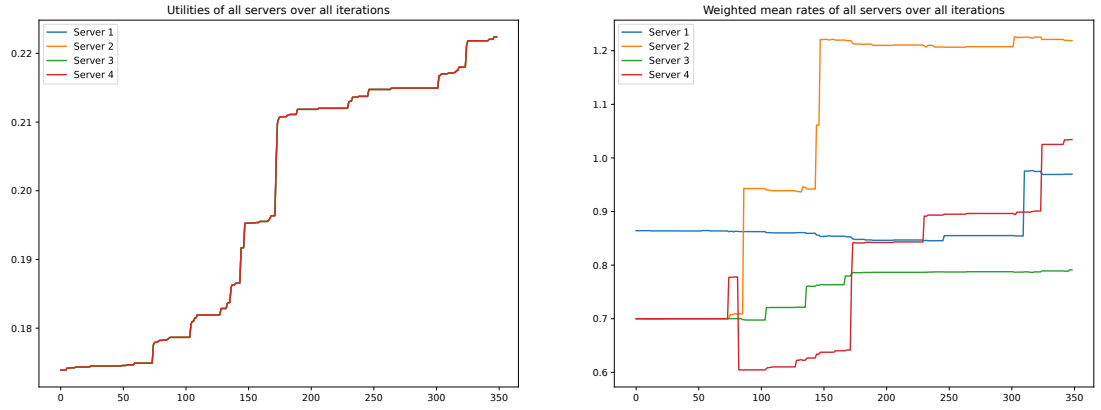


Figure C.39: Utilities and mean service rate of servers from the reinforcement learning run using utility function  $U_k^{(7)}$  with  $e = 0.5$  and increased arrival rates of  $\lambda_1 = 4.0$  and  $\lambda_2 = 4.5$ .

# Appendix D

## Steady state probabilities (closed-form)

This section aims to explore alternative ways of calculating the steady state probabilities using the connection between Markov chains and graph theory.

### D.1 Graph Theory

In mathematics a graph  $G = (V, E)$  is a structure that consists of a set of vertices  $V = \{v_1, v_2, \dots, v_n\}$  and a set of edges  $E = \{e_1, e_2, \dots, e_m\}$  that connect the vertices together [16]. Every edge is expressed as  $e = (v_i, v_j)$  where  $v_i, v_j \in V$ .

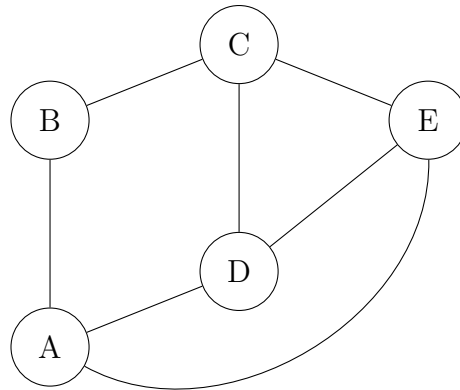


Figure D.1: An example of a simple undirected graph with 5 vertices and 7 edges.

An additional type of graph is a directed graph, where the edges are directed from one vertex to another. In this case, the edges are expressed as  $e = (v_i, v_j)$  where  $v_i, v_j \in V$  and  $(v_i, v_j) \neq (v_j, v_i)$  [11].

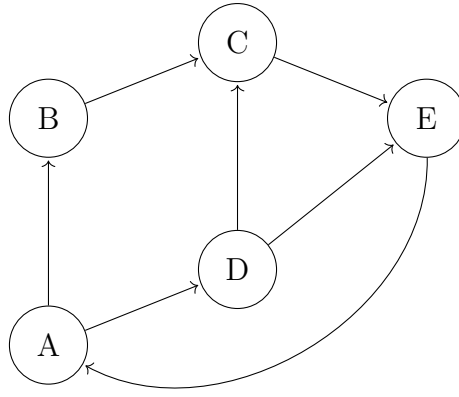


Figure D.2: An example of a directed graph with 5 vertices and 7 edges.

Furthermore, a weighted graph is a graph where each edge has a weight attached to it [84]. In this case, the edges are expressed as  $e = (v_i, v_j, w)$  where  $v_i, v_j \in V$  and  $w \in \mathbb{R}$ .

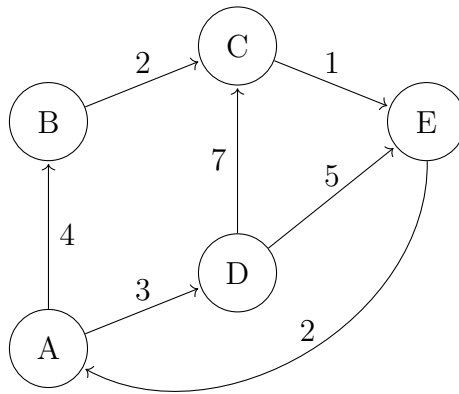


Figure D.3: An example of a weighted directed graph with 5 vertices and 7 edges.

## D.2 A graph theoretic model underlying the Markov chain

It can be assumed that a Markov chain model  $M$  can be translated as a weighted directed graph  $G_M = (V, E)$  where  $V = S$  from equation (3.2) and  $(v_i, v_j) \in E$  if and only if  $q_{v_i, v_j} > 0$ , where  $v_i, v_j \in V$ . Furthermore, the weight of each edge is given by:

$$w(v_i, v_j) = q_{v_i, v_j}$$

As described in Section 3 the parameters considered as inputs are:

- the number of servers  $C$ ,
- the threshold  $T$ ,
- the capacity of node 1  $N$ ,
- the capacity of node 2  $M$ .

These are the parameters that directly affect the structure of the Markov chain as a graph. Additional parameters of the model are the type 1 individuals arrival rate, the type 2 individuals arrival rate and the service rate  $(\lambda_1, \lambda_2, \mu)$ . More specifically, the way these parameters are translated into the model are:

- **Number of servers ( $C$ ):** Affects the weight of all edges  $(v_i, v_j) \in E$  in the Markov chain that correspond to a service rate. These edges have a weight of:

$$w_{(v_i, v_j)} = q_{v_i, v_j}$$

where  $q_{i,j}$  is defined in equation (3.3). Thus, the coefficients of the service rate have a lower bound of 1 and an upper bound of  $C$ .

- **Threshold ( $T$ ):** Determines the length of the left *arm* of the model. In essence the threshold acts as a breakpoint between states where  $u = 0$  and states where  $0 \leq u \leq M$ . Increasing  $T$  results in having more set of states where  $u$  can only be 0.
- **Node 1 capacity ( $N$ ):** Is the upper bound of  $v$  for all states  $(u, v)$ .
- **Node 2 capacity ( $M$ ):** Is the upper bound of  $u$  for all states  $(u, v)$  such that  $v \geq T$ .

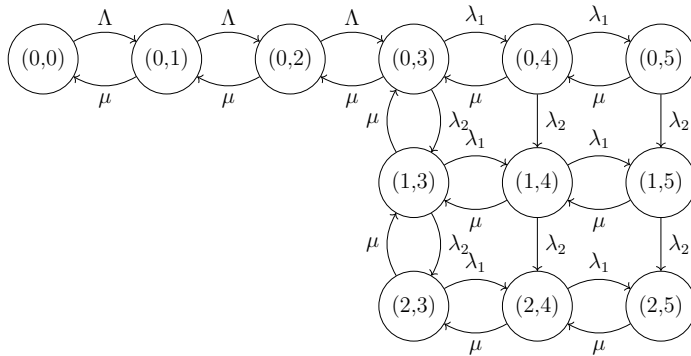


Figure D.4: Example of a Markov model with  $C = 1, T = 3, N = 5, M = 2$

In Figure D.4 an example of such a Markov model is shown where  $C = 1, T = 3$

which means that the *left arm* of the model has a length of 3,  $N = 5$  that indicates that the right-most states  $(u, v)$  are of the form  $(u, 5)$  and  $M = 2$  that equivalently shows that the bottom states are of the form  $(2, v)$ .

### D.3 Spanning trees

A *spanning tree* is defined as a subset of the graph that visits all the vertices of the graph and does not include any cycles [21]. Unlike undirected spanning trees, directed ones also have a root which means that a *directed spanning tree* that is rooted at a vertex  $v$  has to have a path from any other vertex to vertex  $v$  [90]. For example, consider the graph shown in Figure D.5. The graph points out a spanning tree that is rooted at vertex 3.

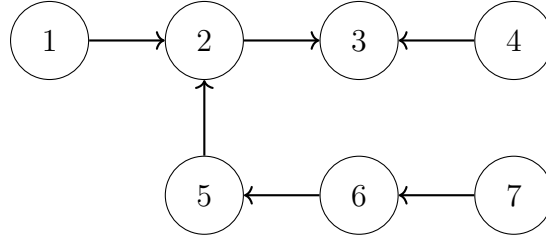


Figure D.5: Example of one of the spanning trees rooted at vertex 3 of the directed graph. Note that the graph corresponds to a Markov model with parameters  $T = 1, N = 3, M = 1$ .

Let us denote the set of all spanning trees of a graph  $G$  as  $T(G)$  and the subset of  $T(G)$  that includes only the spanning trees that are rooted at vertex  $v$  as  $T_v(G)$ . The weight of a spanning tree  $t$  can be defined as the product of the weights of the edges it contains [157]:

$$w(t) = \prod_{e \in t} w(e)$$

**Theorem 3** *Markov chain tree theorem [23]: Let  $M$  be an irreducible Markov chain on  $n$  states with stationary distribution  $\pi_1, \pi_2, \dots, \pi_n$ . Let  $G_M$  be the directed graph associated with  $M$ . Then the probability of being at state  $u$  is given by:*

$$\pi_i = \frac{\sum_{t \in T_i(G_M)} w(t)}{\sum_{t \in T(G_M)} w(t)} \quad (\text{D.1})$$

Equation D.1 states that the probability of being at state  $u$  can be found by dividing the sum of the weights of all trees in  $T_u(G)$  by the sum of the weights of

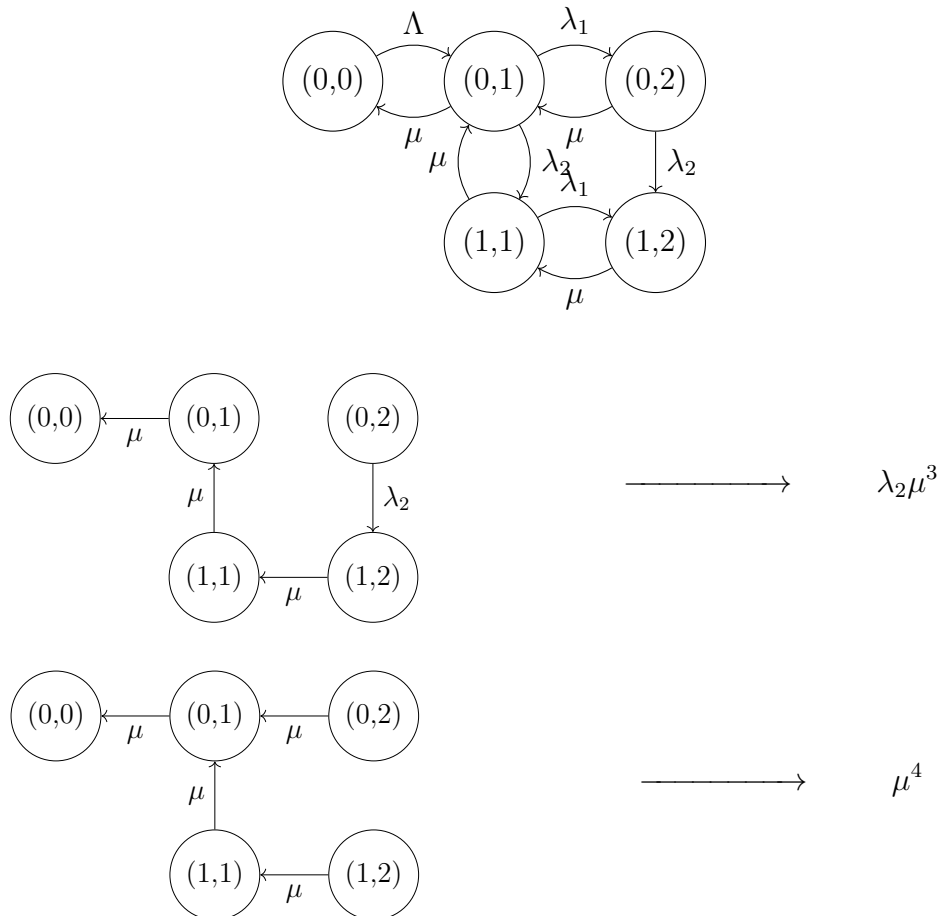
all trees in  $T(G)$ . Let us ignore the denominator of that fraction for now and focus only on the numerator denoted as  $\tilde{\pi}_i = \sum_{t \in T_i(G_M)} w(t)$ . Another useful theorem that can be utilised is Kirchhoff's theorem that gives some useful insights on the number of spanning trees of a graph.

**Theorem 4** *Kirchhoff's theorem [27]: The number of directed spanning trees rooted at a state  $i$  can be found by calculating the determinant of the Laplacian matrix  $Q$  of the directed graph and removing row  $i$  and column  $i$ .*

Some additional research papers that on directed spanning trees enumeration are [146, 147, 159].

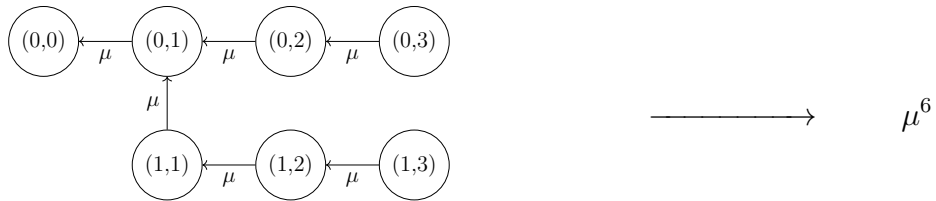
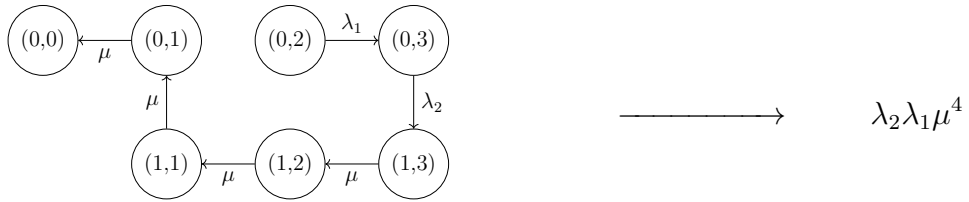
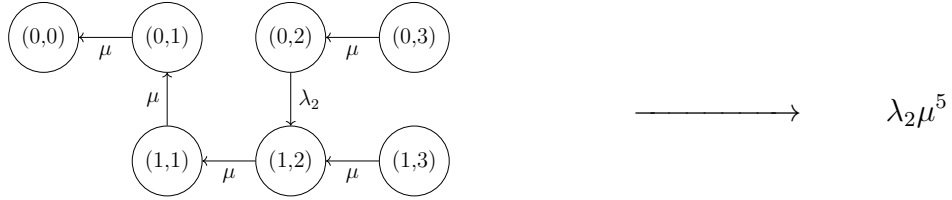
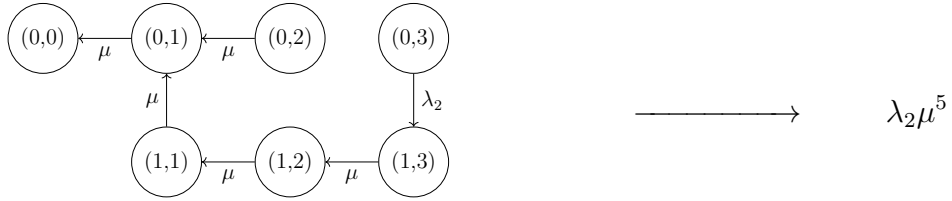
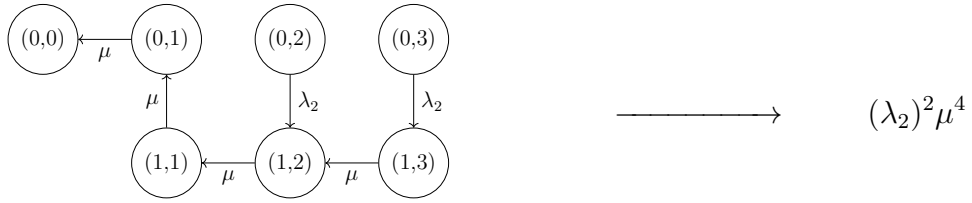
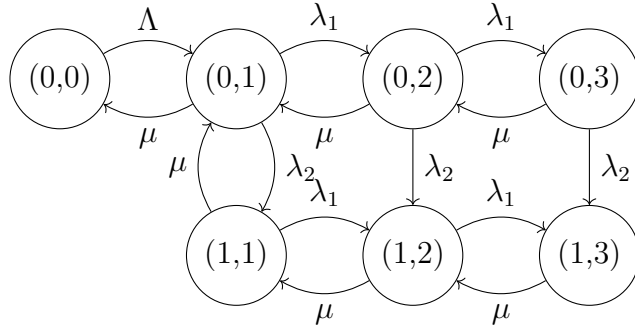
## D.4 Spanning Trees rooted at (0,0)

Let us now consider some examples of spanning trees that are rooted at (0,0). For each of the following examples the complete graph  $G$  is shown, then all possible trees of  $T_{(0,0)}(G)$  along with the weight associated with each spanning tree. As well as this, the sum of all the weights of the spanning trees denoted by  $\tilde{\pi}_{(0,0)}$  is also included.

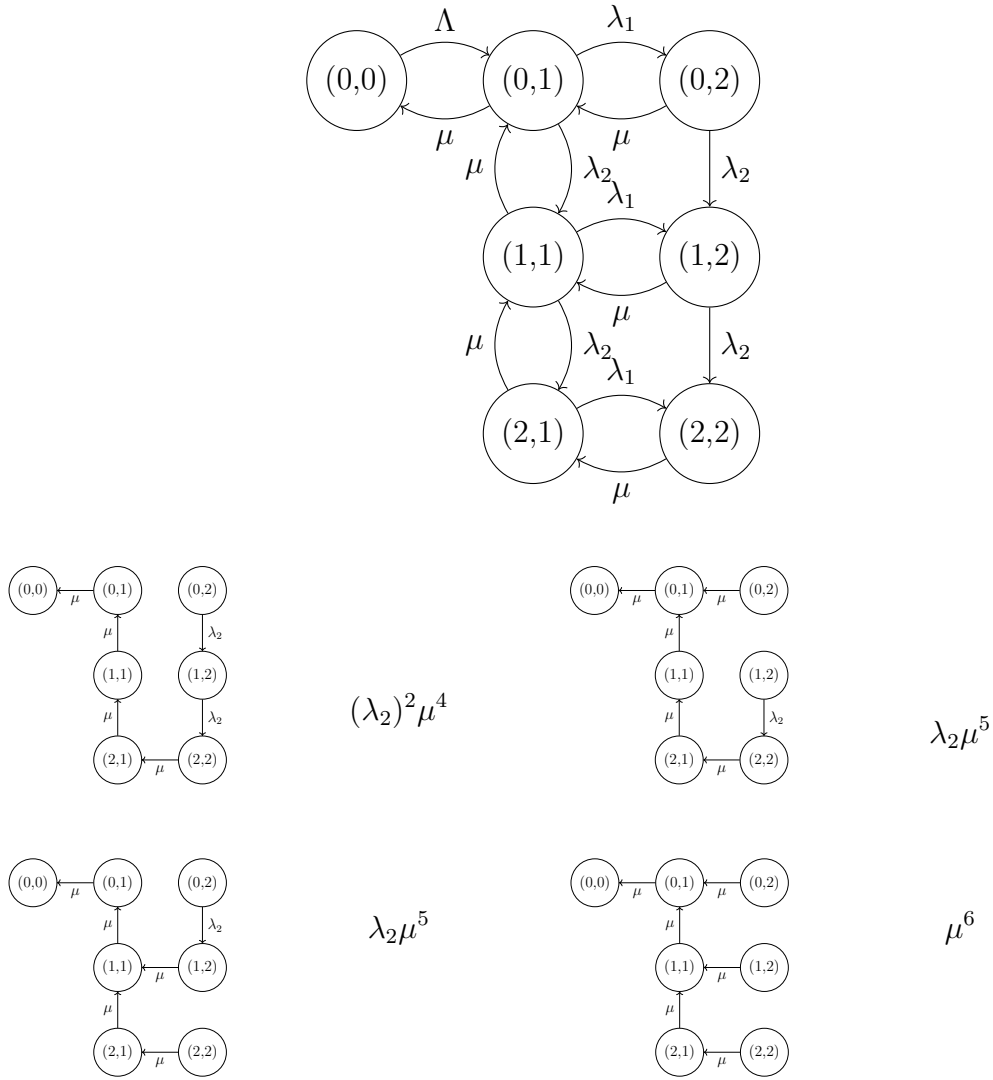




$$\tilde{\pi}_{(0,0)} = \mu^4 + \lambda_2 \mu^3$$



$$\tilde{\pi}_{(0,0)} = (\lambda_2)^2 \mu^4 + 2\lambda_2 \mu^5 + \lambda_2 \lambda_1 \mu^4 + \mu^6$$



$$\tilde{\pi}_{(0,0)} = (\lambda_2)^2 \mu^4 + 2\lambda_2 \mu^5 + \mu^6$$

## D.5 Conjecture of adding rows

Let us consider three Markov models with the same number of servers  $C = 1$ , the same threshold  $T = 1$ , the same node 1 capacity  $N = 2$  but  $M \in \{1, 2, 3\}$ .

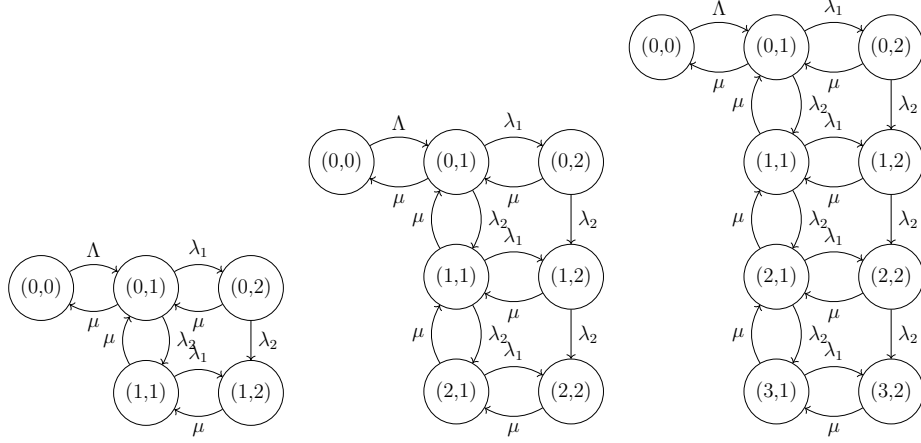


Figure D.6: Examples of Markov chain models with different values of  $M$ , where  $M = 1$  (left),  $M = 2$  (middle) and  $M = 3$  (right).

By increasing the node 2 capacity of the system it can be observed that  $|T_{(0,0)}(G)|$  increases as well since more combinations of paths can be generated using the new edges and vertices. The corresponding values of  $\tilde{\pi}_{(0,0)}$  of the three systems are:

$$M = 1 : \tilde{\pi}_{(0,0)} = \mu^4 + \mu^3 \lambda_2 = \mu^3(\mu + \lambda_2) \quad (\text{D.2})$$

$$M = 2 : \tilde{\pi}_{(0,0)} = \mu^6 + 2\mu^5 \lambda_2 + \mu^4 (\lambda_2)^2 = \mu^4 (\mu^2 + 2\mu \lambda_2 + (\lambda_2)^2) = \mu^4 (\mu + \lambda_2)^2 \quad (\text{D.3})$$

$$\begin{aligned} M = 3 : \tilde{\pi}_{(0,0)} &= \mu^8 + 3\mu^7 \lambda_2 + 3\mu^6 (\lambda_2)^2 + \mu^5 (\lambda_2)^3 \\ &= \mu^5 (\mu^3 + 3\mu^2 \lambda_2 + 3\mu (\lambda_2)^2 + (\lambda_2)^3) \\ &= \mu^5 (\mu + \lambda_2)^3 \end{aligned} \quad (\text{D.4})$$

Note that in equations (D.2), (D.3) and (D.4), the following equation holds:

$$\tilde{\pi}_{(0,0)} = \mu^{(N+M)} (\mu + \lambda_2)^M \quad (\text{D.5})$$

A generalisation of equation (D.5), where  $N \geq 1$ , is given in terms of an unknown function  $k(C, T, N)$  as:

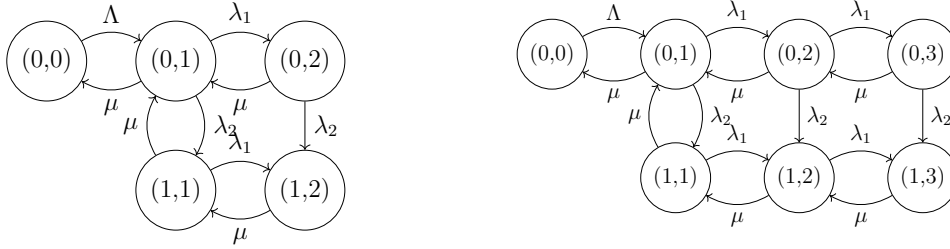
$$\tilde{\pi}_{(0,0)} = \mu^{(N+M)} (k(C, T, N))^M \quad (\text{D.6})$$

Thus, having investigated the effect of adding rows (increasing  $M$ ) the next thing to investigate is the effect of adding columns (increasing  $N$ ) and finding an ex-

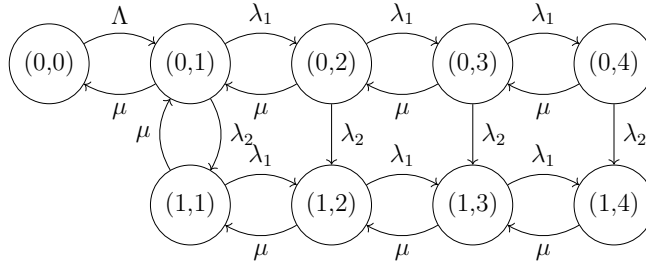
pression for  $k(C, T, N)$ .

## D.6 The effect of increasing N

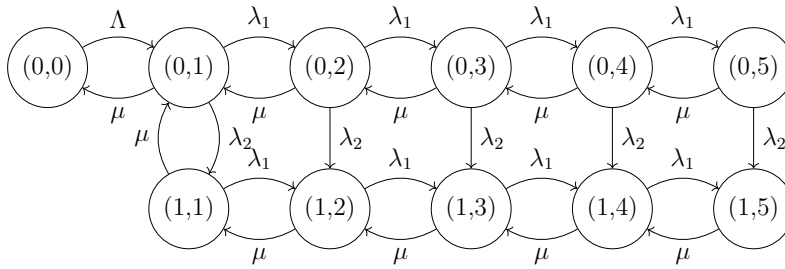
In this section we will consider a node 2 capacity of  $M = 1$  and see the effect of modifying other parameters on  $k(C, T, N)$ .



$$\begin{aligned} \tilde{\pi}_{(0,0)} &= \mu^3[\lambda_2 + \mu] & \tilde{\pi}_{(0,0)} &= \mu^4[(\lambda_2)^2 + \lambda_2\lambda_1 + 2\lambda_2\mu + \mu^2] \end{aligned} \quad (D.7)$$



$$\tilde{\pi}_{(0,0)} = \mu^5[(\lambda_2)^3 + 2(\lambda_2)^2\lambda_1 + 3(\lambda_2)^2\mu + \lambda_2(\lambda_1)^2 + 2\lambda_2\lambda_1\mu + 3\lambda_2\mu^2 + \mu^3] \quad (D.8)$$



$$\begin{aligned} \tilde{\pi}_{(0,0)} = & \mu^6 [(\lambda_2)^4 + 3(\lambda_2)^3 \lambda_1 + 4(\lambda_2)^3 \mu + 3(\lambda_2)^2 (\lambda_1)^2 + 6(\lambda_2)^2 \lambda_1 \mu \\ & + 6(\lambda_2)^2 \mu^2 + \lambda_2 (\lambda_1)^3 + 2\lambda_2 (\lambda_1)^2 \mu + 3\lambda_2 \lambda_1 \mu^2 + 4\lambda_2 \mu^3 + \mu^4] \end{aligned} \quad (\text{D.9})$$

As explained in equation (D.5) the expressions defined above can be reduced to a general form equation of the form  $\tilde{\pi}_{(0,0)} = \mu^{(N+M)} (k(C, T, N))^M$ . The only thing missing is an expression for  $k(C, T, N)$ . An initial attempt to get such an expression can be seen below:

$$\begin{aligned} k(C, T, N) &= \sum_{p_1=0}^{C-1} \sum_{p_2=0}^{C-p_1-1} \sum_{p_3=C-p_1-p_2-1}^{C-p_1-p_2-1} R(p_1, p_2, p_3) (\lambda_2)^{p_1} (\lambda_1)^{p_2} \mu^{p_3} \\ &= \sum_{p_1=0}^{C-1} \sum_{p_2=0}^{C-p_1-1} R(p_1, p_2, C-p_1-p_2-1) (\lambda_2)^{p_1} (\lambda_1)^{p_2} \mu^{C-p_1-p_2-1} \end{aligned} \quad (\text{D.10})$$

In equation (D.10) the coefficient function  $R(p_1, p_2, p_3)$  is introduced where takes as arguments the powers of  $\lambda_2, \lambda_1$  and  $\mu$ . Note here that  $p_3$ , the power of  $\mu$ , is defined as  $p_3 = C - p_1 - p_2 - 1$  since for all base models they need to satisfy  $p_1 + p_2 + p_3 = C - 1$ . For the starting coefficients of the model the function  $R(p_1, p_2, p_3)$  gives the values of the coefficients and is defined as:

$$R(p_1, p_2, p_3) = \begin{cases} 0 & \text{if } p_1 = 0 \text{ and } p_2 > 0 \\ 1 & \text{if } p_1, p_2 = 0 \text{ and } p_3 > 0 \\ \binom{p_1+p_3}{p_3} & \text{if } p_2 = 0 \text{ and } p_1 > 0 \\ \binom{p_1+p_2-1}{p_2} & \text{if } p_3 = 0 \text{ and } p_1, p_2 > 0 \\ p_3 + 1 & \text{if } p_1 = 1 \\ \binom{p_1+p_3+1}{p_1} + p_3 \binom{p_1+p_3}{p_3+1} - \binom{p_1+p_3}{p_3} & \text{if } p_2 = 1 \text{ and } p_1 > 1 \\ \binom{p_1+p_2+1}{p_1} - \binom{p_1+p_2-1}{p_2-1} \\ \quad + \sum_{i=p_2}^{p_1+p_2-2} i \binom{i-1}{p_2-1} & \text{if } p_3 = 1 \text{ and } p_1, p_2 > 1 \\ U_{p_1, p_2, p_3} & \text{otherwise} \end{cases} \quad (\text{D.11})$$

Note here that the final value  $U_{p_1, p_2, p_3}$  corresponds to coefficients that are unknown and are currently investigated. The function  $R$  takes as arguments a possible combination of numbers of  $\lambda_2, \lambda_1$  and  $\mu$  for a given system and outputs the coefficient of that term which in turn represents how many spanning trees exist in the graph with that specific combination. For instance consider the coefficients  $(p_1, p_2, p_3)$  of some of the terms from the equations above:

- (D.7)  $\Rightarrow (\lambda_2)^2: R(2, 0, 0) = \binom{2+0}{0} = 1$
- (D.7)  $\Rightarrow \lambda_2 \lambda_1: R(1, 1, 0) = \binom{1+1-1}{1} = 1$
- (D.7)  $\Rightarrow 2\lambda_2 \mu: R(1, 0, 1) = \binom{1+1}{1} = 2$
- (D.7)  $\Rightarrow \mu^2: R(0, 0, 2) = 1$
- (D.8)  $\Rightarrow 2(\lambda_2)^2 \lambda_1: R(2, 1, 0) = \binom{2+1-1}{1} = 2$
- (D.9)  $\Rightarrow 6(\lambda_2)^2 \lambda_1 \mu: R(2, 1, 1) = \binom{2+1+1}{2} + 1 \binom{2+1}{1+1} - \binom{2+1}{1} = 6 + 3 - 3 = 6$
- (D.8)  $\Rightarrow 3(\lambda_2)^2 \mu: R(2, 0, 1) = \binom{2+1}{1} = 3$
- (D.8)  $\Rightarrow 3\lambda_2 \mu^2: R(1, 0, 2) = \binom{1+2}{2} = 3$
- (D.9)  $\Rightarrow 3(\lambda_2)^3 \lambda_1: R(3, 1, 0) = \binom{3+1-1}{1} = 3$
- (D.9)  $\Rightarrow 3(\lambda_2)^2 (\lambda_1)^2: R(2, 2, 0) = \binom{3}{2} = 3$
- (D.9)  $\Rightarrow 6(\lambda_2)^2 \mu^2: R(2, 0, 2) = \binom{2+2}{2} = 6$

$$\begin{aligned}
 \text{(e.g.) } \Rightarrow (\lambda_2)^2 (\lambda_1)^2 \mu: R(2, 2, 1) &= \binom{2+2+1}{2} - \binom{2+2-1}{2-1} + \sum_{i=2}^{2+2-2} i \binom{i-1}{2-1} \\
 &= 10 - 3 + (2 \times 1) = 9
 \end{aligned}$$

## D.7 Unknown terms

The terms that are still unknown are the terms where  $p_1, p_2, p_3 \geq 2$ . Here are some of these values with the corresponding values of the  $R(p_1, p_2, p_3)$  function.

- $R(2, 2, 2) = 18$
- $R(3, 3, 2) = 100$
- $R(3, 2, 2) = 60$
- $R(3, 2, 3) = 120$
- $R(2, 3, 2) = 24$
- $R(2, 4, 2) = 30$
- $R(2, 2, 3) = 30$
- $R(2, 3, 3) = 40$
- $R(4, 2, 2) = 150$
- $R(2, 2, 4) = 45$

## D.8 DRL arrays

In this section a new combinatorial object is defined: DRL arrays. It will be shown that there is a bijection between DRL arrays and the spanning trees in  $G_M$ . DRL arrays will then be enumerated which in turn enumerates the trees of  $T_{(0,0)}(G_M)$ . Consider the following Markov model and the spanning trees rooted at state  $(0, 0)$  that are associated with it.

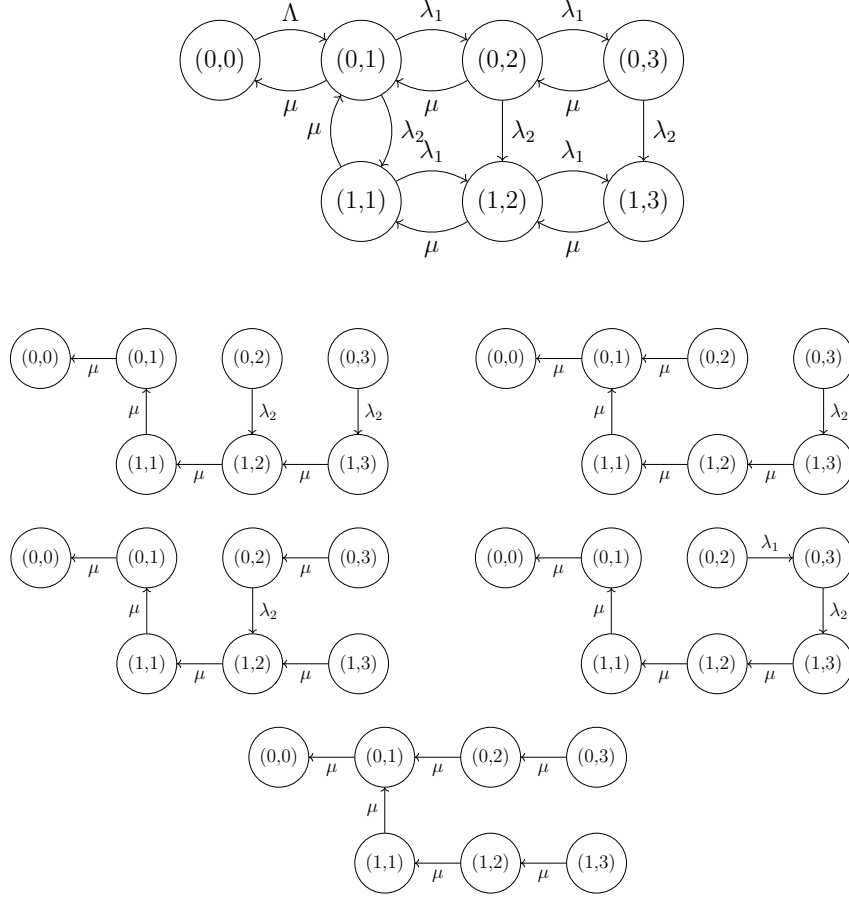


Figure D.7: Markov chain model with  $C = 1, T = 1, N = 3, M = 1$  with all of its equivalent spanning trees rooted at  $(0, 0)$

Looking at these spanning trees from a different perspective it can be observed that all spanning trees of the specific model have some edges in common.

- $(0, 1) \rightarrow (0, 0)$
- $(1, 1) \rightarrow (0, 1)$
- $(1, 2) \rightarrow (1, 1)$
- $(1, 3) \rightarrow (1, 2)$

These edges are the ones on the bottom row of the model, on the *threshold column* and on the *arm* of the model. In general the set of edges that are present on all spanning trees can be denoted by:

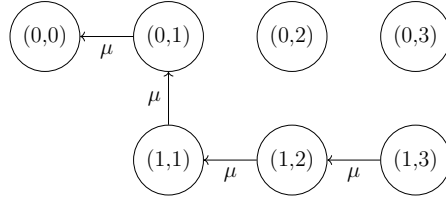
$$\begin{aligned}
 S &= S_1 \cup S_2 \cup S_3 \\
 S_2 &= \{(M, v) \rightarrow (M, v - 1) \mid T < v \leq N\} \\
 S_1 &= \{(u, T) \rightarrow (u - 1, T) \mid 0 < u \leq M\} \\
 S_3 &= \{(0, v) \rightarrow (0, v - 1) \mid 0 < v \leq T\}
 \end{aligned} \tag{D.12}$$

In addition, these edges that are common to every spanning tree (for a threshold

of  $T = 1$ ) have the same weight of  $\mu$ . In this specified model there are four of these edges, each with a weight of  $\mu$ . Thus, since these edges exist on all spanning trees, the weight of every spanning tree must have include a term  $\mu^4$ . Consider the expression of  $\tilde{\pi}_{(0,0)}$  associated with this Markov model:

$$\tilde{\pi}_{(0,0)} = \mu^4 [(\lambda_2)^2 + \lambda_2 \lambda_1 + 2\lambda_2 \mu + \mu^2] \quad (\text{D.13})$$

It can be seen that there is a  $\mu^4$  term that is a common factor of all the terms. This term can be more generally calculated as  $\mu^{M+N}$  and by not considering all these edges that belong in  $S$  the problem can be slightly simplified.



The specific problem has now been reduced to finding all possible combinations of two edges where one starts from  $(0, 2)$  and the other from  $(0, 3)$ . The possible edges that can be utilised here may have a direction of either left, right, or down. Thus, the objective of the problem can be transformed into finding all possible permutations of an array of size 2 where elements can be  $L, R$  or  $D$  and obey certain rules so that the permutation corresponds to a valid spanning tree. These rules are:

1. Permutations ending with an  $R$  are not valid.
2. Permutations that have an  $R$  followed by an  $L$  are not valid.

If any of these two rules do not hold, then the permutation should not be considered. Rule 1 points to the cases where the final state has an edge pointing to the right of it, which cannot occur since that state is the right-most state of the first row. Rule 2 makes sure that there are no neighbour states that point to each other since that would create a cycle and would not generate a valid spanning tree.



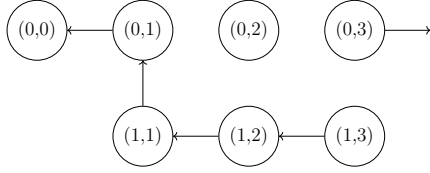


Figure D.8: Example of a permutation that does not produce a valid spanning tree based on Rule 1

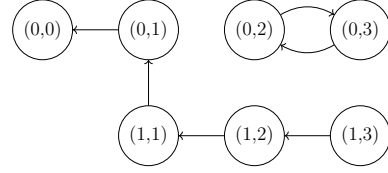
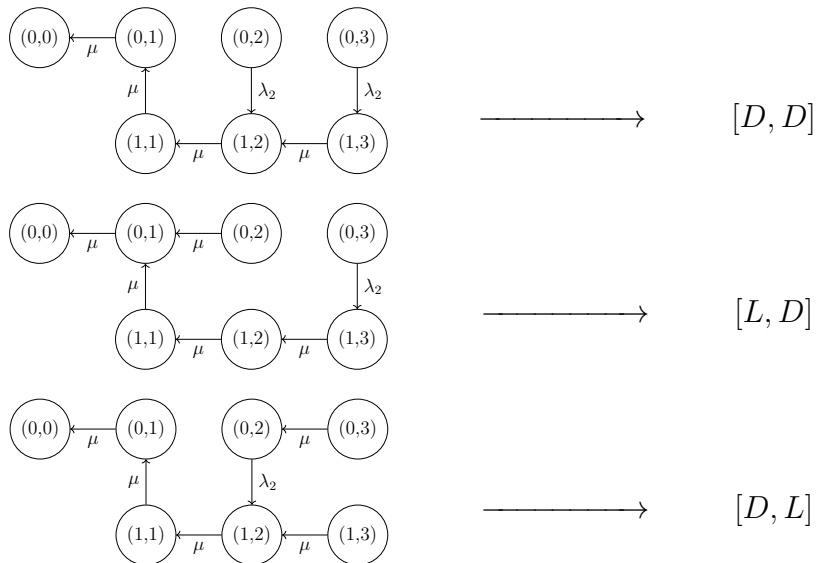


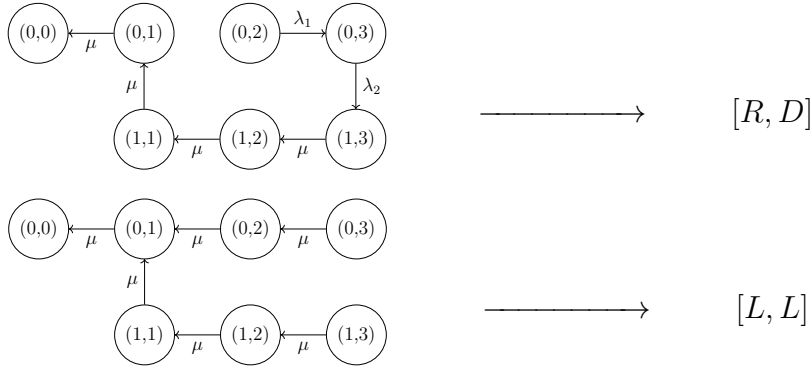
Figure D.9: Example of a permutation that does not produce a valid spanning tree based on Rule 2

Shown below are all possible permutations of the DRL array along with the excluded cases. The valid permutations (on the left) are shown in the same order with their corresponding spanning trees from Figure D.5 and the excluded permutations (on the right) are followed by the rule that determines them not valid.

- |            |  |
|------------|--|
| • $[D, D]$ | • $[\cancel{R}, \cancel{R}] \rightarrow \text{Rule 1}$ |
| • $[L, D]$ | • $[\cancel{L}, \cancel{R}] \rightarrow \text{Rule 1}$ |
| • $[D, L]$ | • $[\cancel{R}, \cancel{L}] \rightarrow \text{Rule 2}$ |
| • $[R, D]$ | • $[\cancel{D}, \cancel{R}] \rightarrow \text{Rule 1}$ |
| • $[L, L]$ |  |

## D.9 Examples of mappings of directed spanning trees to permutation arrays





## D.10 Closed-form approach for state probabilities

A general formula for finding all such permutations can be found, where the inputs are  $p_1$ ,  $p_2$  and  $p_3$  that correspond to the number of  $D$ ,  $R$  and  $L$  respectively and the output would be the coefficient of the term  $(\lambda_2)^{p_1}(\lambda_1)^{p_2}\mu^{p_3}$ . For instance, by applying such a formula to the example in equation (D.13), the desired output should be:

- $(\lambda_2)^2 \rightarrow p_1 = 2, p_2 = 0, p_3 = 0 \rightarrow \text{coefficient} = 1$
- $\lambda_2\lambda_1 \rightarrow p_1 = 1, p_2 = 1, p_3 = 0 \rightarrow \text{coefficient} = 1$
- $2\lambda_2\mu \rightarrow p_1 = 1, p_2 = 0, p_3 = 1 \rightarrow \text{coefficient} = 2$
- $\mu^2 \rightarrow p_1 = 0, p_2 = 0, p_3 = 2 \rightarrow \text{coefficient} = 1$

Thus, given all possible and valid combinations of powers among  $\lambda_2$ ,  $\lambda_1$  and  $\mu$  (i.e.  $p_1, p_2, p_3$ ) generated by equation (D.10), an alternative and improved form of the value of  $R(p_1, p_2, p_3)$  described in equation (D.11) is given by:

$$R(p_1, p_2, p_3) = T(p_1, p_2, p_3) - E_R(p_1, p_2, p_3) - E_D(p_1, p_2, p_3) - E_L(p_1, p_2, p_3) - E_{RL}(p_1, p_2, p_3) \quad (\text{D.14})$$

The term  $T(p_1, p_2, p_3)$  denotes the number of all permutations where neither rule is applied, i.e. all possible ways one can arrange the elements of the array. Removing the number of terms that correspond to rule 1 and rule 2 from the total number of permutations, the desired coefficient is obtained.

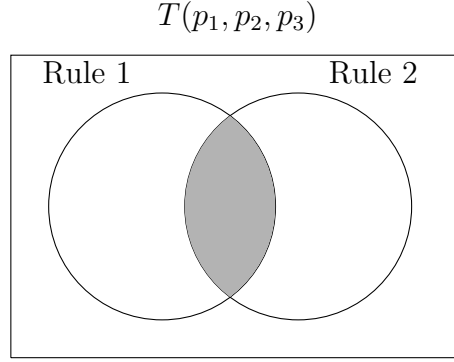


Figure D.10: The two sets of permutations that should not be considered in the calculation of  $R(p_1, p_2, p_3)$ . The left-most set consists of all the permutations that are excluded by Rule 1 and the right-most are excluded by Rule 2

The difficulty with this formulation is that the highlighted intersection from Figure D.10 will be counted twice and is hard to identify how many such terms there are. An alternative approach is to break down the Rule 2 subset into three parts as show in Figure D.11.

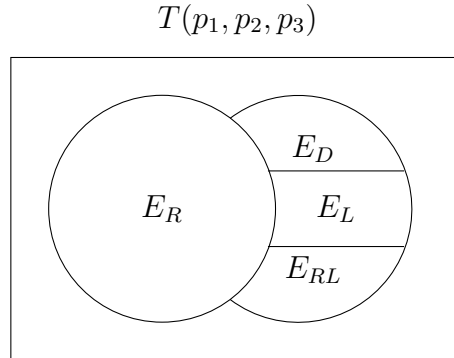


Figure D.11: All permutations that are excluded by Rule 1 and Rule 2 partitioned into four disjoint subsets.

The terms  $E_R, E_L, E_D, E_{RL}$  consist of all cases that should be excluded based on rule 1 and rule 2. The term  $E_R(p_1, p_2, p_3)$  denotes the number of permutations that end in  $R$ , which needs to be removed from the total of all permutations so that rule 1 is satisfied. Having excluded all permutations that end in  $R$ , the permutations that have an  $R$  followed by an  $L$  (rule 2) need to be excluded as well. Although, removing all permutations ending in  $R$  was not too complicated, removing all permutations that follow rule 2 is slightly more complex. This is because equation  $E_R(p_1, p_2, p_3)$  already considers some cases where there is an  $R$  followed by an  $L$ . Therefore, in order to consider only new cases, permutations of rule 2 are split into three new terms;  $E_D, E_L$  and  $E_{RL}$ . These terms denote the permutations that have an  $R$  followed by an  $L$  AND do not end in  $R$ . The

term  $E_D$  considers all permutations that end in  $D$  while  $E_L$  the ones that end in  $L$ . Finally, the last term ( $E_{RL}$ ) denotes all permutations that end in  $R, L$  where there is no other  $R$  followed by an  $L$  in any other position apart from the last two. This term is used because in the  $E_L$  term, such cases (where  $R$  and  $L$  are in the last two positions) are only considered when there is another  $R$  followed by an  $L$  somewhere. Thus, the term  $E_{RL}$  is a particular set of permutations that the formula of  $E_L$  fails to include by itself.

- $T$ : All permutations
- $E_R$ : Permutations ending with  $R$
- $E_D$ : Permutations ending with  $D$  that have an  $R$  followed by an  $L$  somewhere
- $E_L$ : Permutations ending with  $L$  that have an  $R$  followed by an  $L$  somewhere apart from the end of the array
- $E_{RL}$ : Permutations ending in  $R, L$  that do not have an  $R$  followed by an  $L$  anywhere else

Here's the expression for each of these terms:

$$T(p_1, p_2, p_3) = \frac{(p_1 + p_2 + p_3)!}{p_1! \times p_2! \times p_3!} \quad (\text{D.15})$$

$$E_R(p_1, p_2, p_3) = \frac{(p_1 + p_2 + p_3 - 1)!}{p_1! \times (p_2 - 1)! \times p_3!} \quad (\text{D.16})$$

$$E_D(p_1, p_2, p_3) = \sum_{i=1}^{\min(R,L)} (-1)^{i+1} \frac{(p_1 + p_2 + p_3 - i - 1)!}{(p_1 - 1)! \times (p_2 - i)! \times (p_3 - i)! \times (i)!} \quad (\text{D.17})$$

$$E_L(p_1, p_2, p_3) = \sum_{i=1}^{\min(R,L-1)} (-1)^{i+1} \frac{(p_1 + p_2 + p_3 - i - 1)!}{p_1! \times (p_2 - i)! \times (p_3 - i - 1)! \times (i)!} \quad (\text{D.18})$$

$$E_{RL}(p_1, p_2, p_3) = \sum_{i=1}^{\min(R,L)} (-1)^{i+1} \frac{(p_1 + p_2 + p_3 - i - 1)!}{p_1! \times (p_2 - i)! \times (p_3 - i)! \times (i - 1)!} \quad (\text{D.19})$$

$$\begin{aligned} R(p_1, p_2, p_3) = & T(p_1, p_2, p_3) - E_R(p_1, p_2, p_3) - E_D(p_1, p_2, p_3) \\ & - E_L(p_1, p_2, p_3) - E_{RL}(p_1, p_2, p_3) \end{aligned}$$

## D.11 Example of the permutation algorithm

Consider the term  $(\lambda_2)(\lambda_1)\mu^2$  and the above expressions. In order to get the coefficient of that term the permutation algorithm needs to be applied with an input of  $p_1 = 1, p_2 = 1, p_3 = 2$ , i.e. 1  $D$ , 1  $R$  and 2  $L$ s in the array. The permutations that correspond to each expression can be seen below:

$$T(p_1, p_2, p_3) = \frac{(1 + 1 + 2)!}{1! 1! 2!} = 12$$

$$\begin{array}{llllll} [D, R, L, L] & [R, D, L, L] & [D, L, R, L] & [R, L, D, L] & [D, L, L, R] & [R, L, L, D] \\ [L, D, R, L] & [L, R, D, L] & [L, D, L, R] & [L, R, L, D] & [L, L, D, R] & [L, L, R, D] \end{array}$$

$$E_R(p_1, p_2, p_3) = \frac{(1 + 1 + 2 - 1)!}{1! (1 - 1)! 2!} = 3$$

$$[D, L, L, |R] \quad [L, D, L, |R] \quad [L, L, D, |R]$$

$$E_D(p_1, p_2, p_3) = \sum_{i=1}^1 (-1)^{i+1} \frac{(1 + 1 + 2 - i - 1)!}{0! (1 - i)! (2 - i)! (i)!} = 1 \times \frac{2}{0! 0! 1! 1!} = 2$$

$$[R, L, L, |D] \quad [L, R, L, |D]$$

$$E_L(p_1, p_2, p_3) = \sum_{i=1}^1 (-1)^{i+1} \frac{(1 + 1 + 2 - i - 1)!}{1! (1 - i)! (2 - i - 1)! (i)!} = 1 \times \frac{2}{1! 0! 0! 1!} = 2$$

$$[D, R, L, |L] \quad [R, L, D, |L]$$

$$E_{RL}(p_1, p_2, p_3) = \sum_{i=1}^1 (-1)^{i+1} \frac{(1+1+2-i-1)!}{1! (1-i)! (2-i)! (i_1)!} = 1 \times \frac{2}{1! 0! 1! 0!} = 2$$

$$[D, L, |R, L] \quad [L, D, |R, L]$$

Although this method is useful, currently this can be used to find only spanning trees that are rooted at state  $(0, 0)$  and for the case of  $C = 1$ . In order to get the steady state probabilities, a more general approach that is generic to all states  $(u, v)$  is needed. Further work on this specific topic will not be part of this research project.