TOPOLOGICAL REASONING USING A GENERATIVE REPRESENTATION AND A GENETIC ALGORITHM

A Thesis

Presented to

The School of Engineering

Cardiff University

In Partial Fulfilment

of the Requirements for the Degree of

Doctor of Philosophy

by

Yu Zhang

December 2009

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Acknowledgements

My PhD study at Cardiff University had been an enjoyable journey that I would cherish to remember for the rest of my life. This wouldn't have been possible without the people I'd like to mention here. Firstly and particularly, I'd like to thank Professor John Miles, who has been a mentor, a friend to me, for his continuous help and advisory. I'd like to thank Dr Alan Kwan for his guidance throughout my research. I enjoyed every meeting and discussion we had. I'd like to thank Dr Yacub Rafiq and Dr Yulia Hicks for all their comments and suggestions for my thesis. I'd also like to thank Chris Lee from the research office for all her help in arranging my viva exam and final submission of my thesis. Finally, I want to thank Ting, my wife, for being there for me, and my parents for all your love and support.

Abstract

This thesis studies the use of a generative representation with a genetic algorithm (GA) to solve topological reasoning problems.

Literature review indicates that generative representations outperform the non-generative ones for certain design optimisation and automation problems. However, it also indicates a lack of understanding of this relatively new class of representations. Many problems and questions about the implementation of generative representations are still to be addressed and answered.

The results and findings presented in this thesis contribute to the knowledge of generative representations by:

1. explaining why genotype formatting is important for the representation and how it influences the performance of both the representation and the algorithm;
2. providing different crossover and mutation methods, including both existing and newly developed ones, that are available to GA when used with the presentation and, more importantly, revealing their different properties in generating new individuals;
3. providing alternative ways to map turtle graphs into the design space to form the actual designs and showing the properties of these different mapping methods and how they influence the outcome of the search.

In general, this thesis examines the key issues in setting up and implementing generative representations with genetic algorithms. It improves the understanding of generative representations and contributes to the knowledge that is required to further develop them for real-world use. Based on the results and findings of this study, directions for future work are also provided.

Contents

Table of Figures

Chapter 1 Introduction

Since their introduction almost three decades ago, genetic algorithms have been very popular among researchers and engineers as a means to search for solutions due to their robustness and adaptability. A lot of work has been done to improve the algorithms' performance as well as to extend their capabilities to cope with new problems, among which the representation techniques have become a very active field of research in recent years. This thesis tackles the issues and challenges of a special representation method which is a particular implementation of generative representation using L-systems and turtle graphics. First in this chapter, a brief introduction for genetic algorithms and the impact of representation are given. Second, the definition of generative representations is provided and explained with examples. Issues of using generative representations that are noticed by the author and addressed in this thesis are presented the next as the contributions of the work. The outline of the thesis is given at the last.

1.1 Genetic Algorithms & the Impact of Representation

A Genetic algorithm (GA) is a search technique used in computing to find solutions for an optimization or search problem. Genetic algorithms (GAs) are a branch of evolutionary algorithms which are inspired by evolutionary biology. They simulate the nature's evolutionary circles by automating the processes including selection, reproduction (crossover) and mutation which are programmed into operators for the algorithms. Real-world domains that GAs are applied to range from everyday activities, such as timetable design, to the most advanced fields of science and engineering, such as finding new chemical molecules, designing composite materials and aerodynamic shapes for race cars, looking for optimized telecommunication routing and many more. The typical implementation circle of a GA is given in Chapter 2.

The fitness function, according to the generally accepted wisdom, is the only connection between the algorithm and the real problem. It is a result of the most commonly adopted practice of using a parameterized representation. However, this

is not true for many modern engineering problems where the reasoning about shape and topology is needed and especially where a parameterized representation does not serve as a satisfying solution. In such cases, an alternative form of representation is needed and plays an important role in connecting the real-world problem to the algorithm. It provides a way to encode the problem/design so that it can be understood and operated by the algorithm as well as to translate an encoded design to an actual design. The work by Zhang et al (Zhang, 2006) shows that, for shape and topological reasoning problems which are important in many areas of engineering, it is vital that appropriately formed representations are used. The choice of representation method and how it is applied can, to a great extent, determine the performance of the algorithm and the quality of the solutions it can achieve.

1.2 Generative Representations

A **generative representation** is defined by Hornby (2003a) as one in which an encoded design can reuse elements of its encoding in the translation to an actual design through either abstraction or iteration.

The meanings of abstraction and iteration are the same as they are in programming languages. Abstraction refers to both the ability to manipulate assemblies of components as units and to pass parameters to procedures. Iteration, also known as control-flow, refers to the control of execution which permits the conditional and repetitive use to structures. A generative representation's ability to reuse improves the algorithm's capability in navigating large design spaces. This is achieved through operating on units of compound elements rather than single elements. Also, the ability to reuse elements improves the representation's scalability to cope with problems of higher complexity. By making it possible to change several parts of a design simultaneously, it also allows the representation to capture design dependencies. The advantages of using generative representations are further discussed in Chapter 4.

Since generative representations are relatively new, examples of their applications are still rare. The work by Hornby (2003a) is one of the few good examples that can

be found in the literature. In his work, he used generative representations to solve a number of design automation problems including the design of tables, neural networks and neural-network controlled robots. Within the civil engineering discipline, the best example of using generative representations is probably the work by Kicinger et al (2005a) who use cellular automata to design bracing systems for tall buildings.

1.3 Contributions

In general, the study of representation techniques is an area of research that is worth the effort because it not only helps to improve the performance of the algorithms on existing problems but also enhances the algorithms by extending their capability to cope with new problem domains.

Generative representations have such potential; however, a lack of understanding of this particular form of representation still exists. Although Hornby in his work (Hornby, 2003a) applied generative representations to several design domains and yielded promising results, the results at their best only demonstrated that generative representations are applicable to certain design domains and proved their advantages over non-generative representations. Hornby's work does not provide discussions as to how generative representations should be used, for example, how they should be used in connection with GA operators such as crossover and mutation. In other words, there are still "black boxes" between generative representations and the results. This prevents people from understanding the representations and hence has limited value to those who want to apply generative representation to their own domains. Work by Kicinger et al (2005a) has the same problem.

Rather than dealing with multiple problems, this thesis focuses on a single design domain to which a GA with a particularly form of generative representation scheme is applied. The design domain is two-dimensional topological reasoning problem and the generative representation is structured using L-systems and turtle graphics. The thesis not only reassures that generative representations are indeed a competitive alternative representation technique for topological reasoning using GAs but also

looks further into the detailed implementation of the technique. It tackles the real challenges of generative representations, especially in connection with the native operators of GAs such as mutation and crossover. It also examines the adjustable attributes of the representation and suggests strategies for tuning them. In general, the thesis provides a more in-depth understanding of a particular form of generative representation than the existing works by investigating the fundamental properties of this form of generative representation. It contributes to the knowledge of generative representations by:

1. explaining why genotype formatting is important for the representation and how it influences the performance of both the representation and the algorithm;
2. providing different crossover and mutation methods, including both existing and newly developed ones, that are available to GA when used with the presentation and, more importantly, revealing their different properties in generating new individuals;
3. providing alternative ways to map turtle graphs into the design space to form the actual designs and showing the properties of these different mapping methods and how they influence the outcome of the search.

Although the study is based on a particular form of generative representation and a particular problem domain, it is nevertheless a useful reference for those who want to set up their own generative representation schemes for their own problem domains because it points out where to look at to structure a proper generative representation.

All experiments carried out in this thesis are done by a piece of software written by the author in C++. It uses genetic algorithm and generative representation to solve two-dimensional topological reasoning problems. For research purpose, it provides comprehensive access to algorithm and representation attributes.

1.4 Thesis Outline

The rest of the thesis is organized in the following way:

Chapter 2 reviews evolutionary computation in structural optimisation. As the technique used in this study, genetic algorithms and their applications are introduced and reviewed in more detail.

Chapter 3 reviews representation techniques that are commonly used with evolutionary computation on topological reasoning problems. Their advantages and limitations are discussed.

In Chapter 4, generative representation as the focus of the thesis is introduced. The generative representation used in the thesis, the one which used L-systems and turtle graphics, is also described.

Chapter 5 discusses the implementation of the particular generative representation described in Chapter 4, mainly focusing on the encoding and the decoding processes.

Chapter 6 also discusses the detailed implementation of the representation but it aims at issues that are connected with GA operators which are crossover and mutation.

Chapter 7 presents the experiment results. Experiments are carried out to test the influence of the different settings of the representation. The results are used to argue that generative representation is a competitive alternative to the original representation techniques for topological reasoning problems; however, the attributes of the representation need to be properly tuned to achieve a satisfying outcome.

Chapter 8 gives the conclusions of the thesis. Direction of future work is also suggested.

Chapter 2 Evolutionary Computation and Structural Design

Due to the development in information technology, seeking aid from computers to improve design activities is very common nowadays. In structural design, the role of computers is shifting from the exclusive tool for calculation based analysis to their holistic applications in design (Kicinger et al., 2005b). Evolutionary computation is one of the computational paradigms that power the modern use of computers in the structural design activities. In this chapter, the use of evolutionary computation in structural design is reviewed. Genetic algorithms, as one of the most popular techniques in evolutionary computation and the one used in this study, are introduced and reviewed in more detail.

2.1 Evolutionary Computation

Artificial intelligence (AI) is a significant component of computer science. Computational intelligence (CI) is a branch of AI. Its scope, as defined by the IEEE Computation Intelligence Society (http://www.ieee-cis.org/), involves the theory, design, application and development of computational paradigms that use techniques such as neural networks, fuzzy systems, evolutionary computation and other similar or related techniques, e.g. intelligent agents (Russell et al., 2003).

Evolutionary computation (EC) is frequently applied to combinatorial optimisation problems where the space of feasible solutions is, or can be reduced to, a discrete one. It often becomes the method of choice where deterministic techniques such as linear programming and gradient methods are found to be incompetent. Many of its applications (Beasley, 1997) and increasing interest are due to its advantages in solving complex problems. Evolutionary computation techniques require little knowledge about the problem being solved as long as one can provide evaluation for solutions and also bear the characteristics of easy implementation, robust and inherently parallel (Schwefel, 1997). One subcategory of evolutionary computation, swarm intelligence (SI), includes techniques such as ant colony optimisation (Dorigo and Stützle, 2004) and particle swarm optimisation (Eberhart and Kennedy, 1995).

Genetic algorithms, along with other techniques such as genetic programming (Koza, 1992), evolutionary programming (Fogel et al., 1966), evolutionary strategies (Beyer, 2001) and learning classifier system (Holland, 1986), fall into the other subcategory which is named as evolutionary algorithms (EA).

Techniques from both SI (swarm intelligence) and EA (evolutionary algorithms) implement population-based stochastic search of a solution space for the best possible solutions or optima. In SI, such search is based on the collective behaviour of decentralised, self-organized systems (agents). Take ant colony optimisation for example, the agents (simulated ants) interact locally with one another and with their environment. The synthesis of such interactions forms a more complex global behaviour in which, in its later phase, more agents are expected to locate better solutions. The process simulates real-world ant colonies and how they find the shortest path for food.

Evolutionary algorithms imitate nature's biological evolution. In an EA system, individuals of the population represent candidate solutions. The fitness value of an individual determines its influence in a survival-of-the-fittest environment. Instead of relying on a population's collective act, an EA system seeks better solutions through iterations of reproduction, mutation, selection and recombination on or among individuals. The process can be described as artificial evolution during which better solutions, who themselves are individuals of the population, are expected to be found in later generations. Genetic algorithms are examples of such implementation (Goldberg, 1989).

2.2 Genetic Algorithms

Genetic algorithms as an optimisation technique became well known through the work of Holland (Holland, 1975) in the early 1970s. Through their development, variations and additions have been made to improve the performance. The basic architecture of a canonical genetic algorithm (Goldberg, 1989) is given in Figure 2.1. The breakdown of an example GA system which is used in this study with slight modification from the canonical GA is described later in this section.

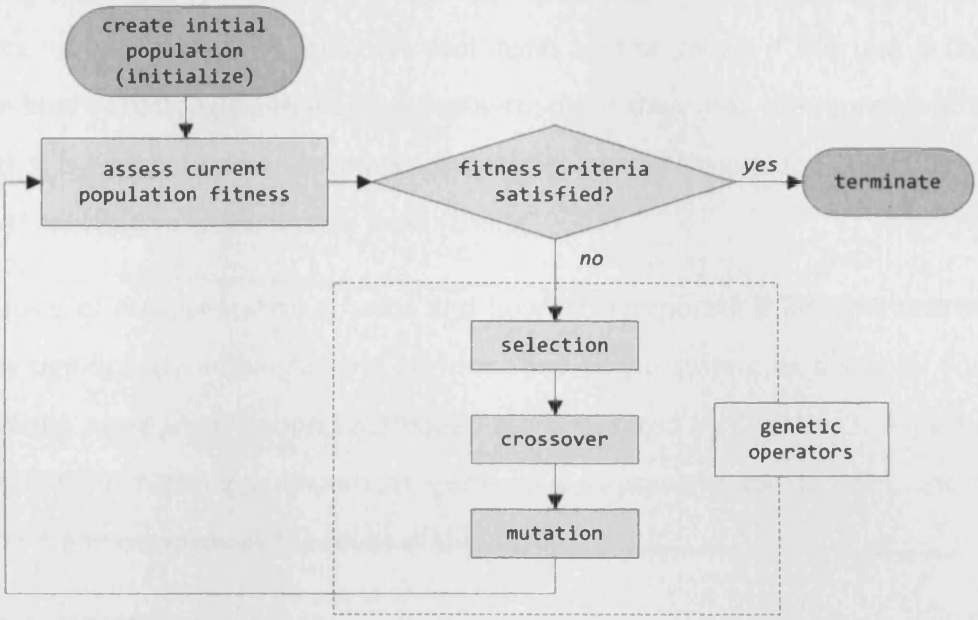


Figure 2.1 schematic representation of canonical GA

2.2.1 Representation

Every GA system requires a representation scheme. It defines how individuals are structured and described in their "genotypes" and how to convert them into their "phenotypes". The relation between genotype and phenotype is like that between DNA code and human being. A genotype is the genetic coding for an individual. In a GA system, it is digital information that can be replicated and passed from one individual to another. A phenotype is what its genotype decodes into. It can be considered as the physical instantiation, the analogue of the genotype.

The representation method decides how individuals are presented to the algorithm. Although some problem specific material may also be found in the fitness function which is described in a later section, what part of the problem the algorithm has direct access to is defined by the representation. For example, if the width w and the height h are to be used to describe a shape, which means a shape is presented by its width and height, the algorithm will only have access to these two variables and will depend on them to generate more shapes. No direct modification to the other properties of the shape can be done by the algorithm because it has no access to them. Furthermore, since only w and h are contained in each individual, the kind of shapes that are determinable by these two variables could only be rectangles

resulting from the genotype to phenotype conversion. The algorithm is able to produce various rectangles, but only rectangles and no more. If the task is to find certain kind of rectangles, there is no problem; other than that, the representation is limited. It is why parameter based representation should be avoided unless the form of the final solution is reasonably fixed (Zhang, 2006).

The choice of representation scheme and how to incorporate it into the rest of the system significantly influences the performance of the system as a whole. Further discussions on representation techniques are presented in Chapter 3. A particular representation technique known as generative representation is discussed from Chapter 4 and onwards as the focus of this study.

2.2.2 Initialization

A GA system manipulates a collection of individuals that is referred to as the population. The lifetime of a GA system run starts with initializing such a population. The size of the population $psize$ is predefined. Although there are no general guidelines for the population size, because it is highly dependent on the nature of the problem, it commonly ranges from several tens to thousands. Following the definition of the representation scheme, the initialization module of the system typically randomly generates abstract representations of feasible solutions one by one until the number reaches the population size. In some cases, the initial population is not generated completely at random; instead, by limiting the initialization, individuals are seeded in certain areas where optimal solutions are considered more likely to be found, in order to increase search efficiency. For the case of this study, however, as no previous knowledge about what the optimal solutions/topologies should be like is available, only a feasibility check is applied to the initialization. As is shown later, a feasibility check is necessary for generative representations because it is possible for individuals that cannot be assessed to be generated. The same restriction applies to mutation and crossover, as well, because they also generate new individuals that may not be feasible. Detailed discussion on this particular problem can be found in Chapter 5. Figure 2.2 illustrates the implementation of the initialization module used in this study.

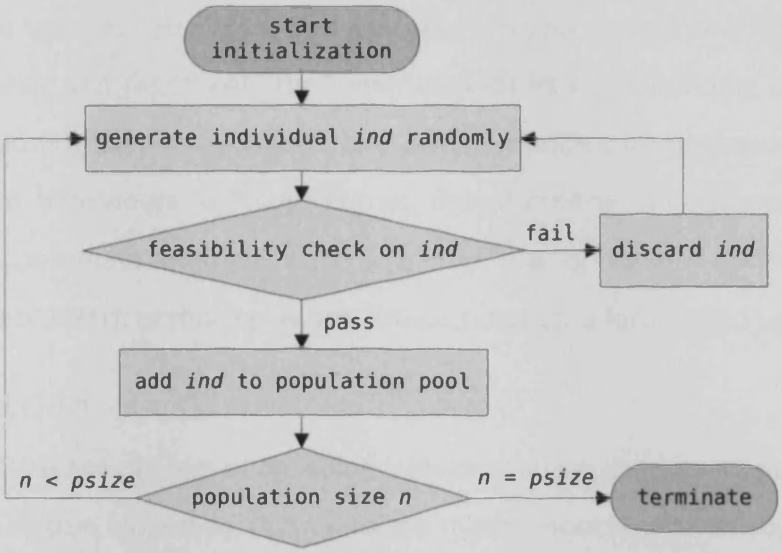


Figure 2.2 flowchart for population initialization

2.2.3 Evaluation

The fitness value expresses how good an individual is. It is achieved by evaluating an individual against predefined criteria which are formulated into a mathematical expression known as the fitness function. Composed from a quality measure in the phenotype space, a fitness function assigns a quality measure to genotypes, which forms the basis for selection (introduced in Section 2.2.4) (Eiben and Smith, 2003). A fitness function is a special type of but is not exactly the same as the term objective function defined in mathematical context. It is used to quantify an individual's optimality which often contains consideration of multiple aspects/design criteria which correlate closely yet need not fully describe the algorithm's goal. In other words a fitness function is not as well defined as an objective function. Besides, because GAs cannot directly handle constraints, this part of job is incorporated into the fitness function as an objective which is restricted by a penalty function/coefficient or some other mechanism.

A fitness function for multi-objective optimisation problems can take the form below:

$$f = \alpha \cdot (a_1 \cdot X_1 + a_2 \cdot X_2 + a_3 \cdot X_3 + \cdots + a_n \cdot X_n) \qquad (2.1)$$

, where f is the fitness value; α is a penalty coefficient generally used to discourage individuals with undesirable properties, e.g. exceeding constraints; $X_1, X_2, X_3, \cdots X_n$ corresponding to n different design criteria and are values that describe how much

the individual satisfies these criteria; $a_1, a_2, a_3, \cdots, a_n$ are weight coefficients each of which is a value that represents the importance of its corresponding criterion. The coefficients do not have to be static. They can be modified in runtime to encourage certain search behaviours to favour certain design criteria. An alternative to using weights to coordinate multiple criteria is the use of Pareto optimality/Pareto efficiency (Deb, 2001). In this thesis, the fitness function is formulated using weights.

2.2.4 Selection

Selection in GA is the process of choosing individuals from the current population for later recombination by putting them into the mating pool. The two most commonly used selection methods are fitness proportionate selection (also known as roulette-wheel selection) and tournament selection. Both of the two methods are fitness-based, which means that before the selection happens, all individuals of the population have to be evaluated and be given their fitness values. (Eiben and Smith, 2003)

In fitness proportionate selection, the probability for an individual to be selected is strictly associated with its fitness value comparing against those of the rest of the individuals, which can be calculated using the equation:-

$$p_i = \frac{f_i}{\sum_{j=1}^{psize} f_j} \tag{2.2}$$

where p is the probability; f is the fitness; i and j are the index values of the individuals. When using this selection method, users have no control over the selection pressure which varies across generations. The pressure of tournament selection, in contrast, can be easily set by the users. In a tournament among s individuals (competitors) which are randomly selected from the population, the winner is the individual with the highest fitness and is inserted into the mating pool. By changing the tournament size s, the selection pressure can be changed. Having more competitors means higher resulting pressure under which low-fitness individuals are more likely to be eliminated, and vice versa. The GA system used in the study features tournament selection and provides runtime selection pressure changing utility.

2.2.5 Mutation, Recombination and Iteration

New individuals are generated through recombining individuals from the mating pool and mutating existing individuals in emulation of nature's reproduction process.

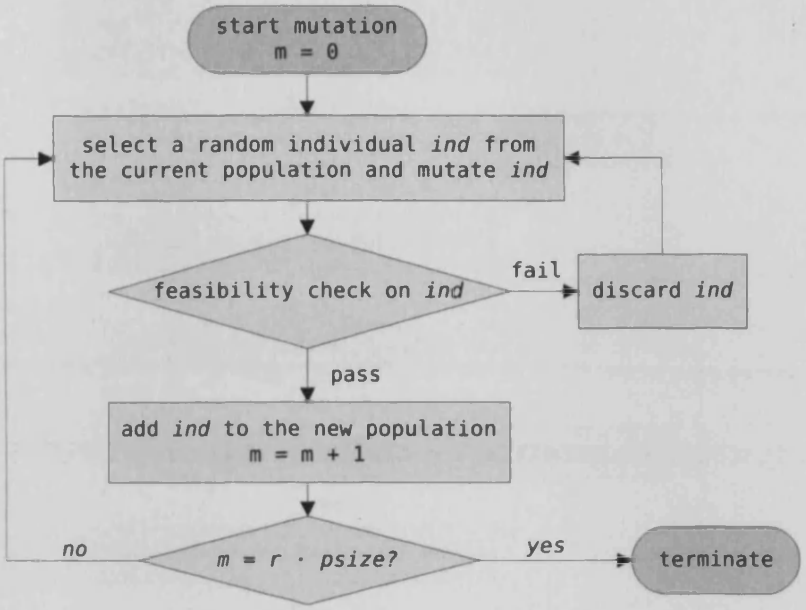


Figure 2.3 flowchart for mutation module

Mutation in a GA is done by a mutation operator generally used to maintain the diversity of the population from one generation to another. It is applied at a low probability, normally several percent, to avoid unnecessarily disturbing the search process. Given the mutation rate r and the population size $psize$, the number of new individuals to be generated by mutation and to be put into the new population is $r \cdot psize$. Figure 2.3 illustrates the general implementation of mutation module of the GA system.

The next step is recombination which generates new individuals by applying crossover operators on parent individuals selected from the mating pool. In some practice, more than two parents are selected for each crossover. The GA system used in the study for each time uses two parent individuals and produces two new individuals which, if feasible, are put into the new population. Figure 2.4 explains the process of using crossover, after mutation, to generate the rest of the new population.

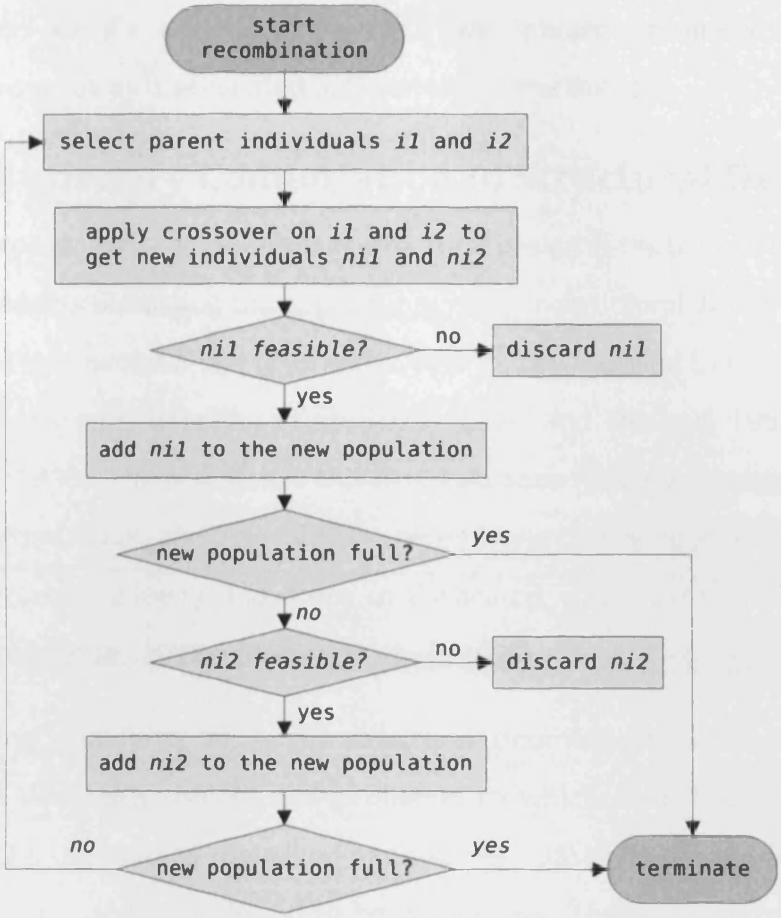


Figure 2.4 flowchart for recombination module

After a whole new population is generated, it replaces that of the previous generation and becomes the current population. Individuals of the population are re-evaluated and are assigned with their fitness values. The procedure then goes back to the selection stage for a new round of reproduction. Such iteration keeps going until it reaches a terminating condition in which

- an optimal solution is found, or
- the predefined maximum generation or execution time is reached, or
- the fitness of the best solution reaches a plateau where further improvement is considered unlike to happen, or
- a combination of the above conditions.

The GA system used in this study features the option to set the maximum number of generations as well as the ability to terminate executions manually based on user inspection. The detailed implementation of both mutation and crossover is highly

representation specific. How to apply these two operators properly is one of the focuses of this study as is presented in Chapter 6 of the thesis.

2.3 Evolutionary Computation in Structural Design

The use of evolutionary computation in structural design dates back to mid 1970's at which time studies discussing the applications of EC in structural design were mainly focused on simple evolutionary algorithms, such as GAs, applied to simple structural optimisation problems (Hoeffler et al., 1973) (Lawo and Thierauf, 1982) (Goldberg and Samtani, 1986). The research in this field had been focused on various aspect of structural optimisation and had only in recent years developed to the stage to address issues of creativity and more sophisticated ways to represent structural systems (Hamda et al., 2002a) (Bekiroglu et al., 2009).

The increasing popularity of EC in structural optimisation was a result of its capability to deal with complicated problems to which formal methods such as mathematical programming (Belegundu and Arora, 1985) and the optimality criteria method (Rozvany, 1992) are found to be inadequate. These formal methods work well on well-formed structural optimisation problems where the structural system's configuration is reasonably fixed; however, for more generalized problems which allow variations in the system's configuration, most formal methods are found to be deficient. Evolutionary computation, in contrast, is good at handling difficult optimisation problems with nonlinear, stochastic, or temporal components, and hence outperforms formal methods in dealing with structural optimisation problems with variable configuration of structural system.

2.3.1 Structural Optimisation Problems

Structural optimisation problems can be divided into three major categories as topology optimisation, shape optimisation and sizing optimisation. Topology optimisation, also known as topological optimum design (TOD), is to look for an optimal distribution of material of a structural system and is mostly conducted in the conceptual design stage. Shape optimisation, is to seek optimal shape or contours for a structural system whose topology is determined. Sizing optimisation, which is related to the detailed design stage, is to search for optimal dimensions of

components of a structural system whose topology and shape are both determined. The structural optimisation problem used in this thesis falls into the category of topology optimisation for continuum structures.

2.3.2 Formal Methods

Formal methods were used in the early years of topology optimisation studies. These methods include the homogenization method introduced by Bendsøe and Kikuchi (1988) and evolutionary structural optimisation (ESO) proposed by Xie and Steven (1993).

Homogenization method is based on the assumption that the density and the orientation of each element contained in a grid of composite material are continuously variable. Development and applications of this method can be found in a series of work by Bendsøe and his fellow researchers (Bendsøe and Rodrigues, 1991) (Suzuki and Kikuchi, 1991) (Olhoff et al., 1991) (Bendsøe, 1995) (Bendsøe et al., 1996).

The ESO method, although has "evolutionary" in its name, is not an evolutionary computation method. The method follows the same concept as described in the work by Rozvany (1992) in which the optimization process starts from an initial design and then gradually removed material in areas of low stress. Later development in ESO includes the work by Hinton and Sienz (1995) who developed and integrated and interactive design approach based on ESO, Steven et al. (2000) who extended the applicability of ESO from its original continuum structural topology optimisation to combined topology and sizing optimisation of discrete structures, and Tanskanen (2002) who provided the mathematical foundation and outlined the theoretical basis of ESO.

Both homogenization method and ESO, as formal methods, suffer the same limitation, that is, they only work well on well-formed problems. Homogenization method ensures this by assuming the continuity of anisotropic materials of infinitely varying density, which is not always a feasible assumption. Consequently, there is a need for interpretation which results in a final structure which is someway different from that produced by the method. With ESO, starting the optimisation from an

initial structural system which is feasible and well-formed makes sure the optimisation problem is a well-formed one. This is a major limitation for ESO and renders the method inadequate at dealing with complex structural systems and providing creativity in designs.

2.3.3 Evolutionary Computation Methods

Deficiencies of formal methods and the increasing complexity of problems encountered contribute to the rise of EC methods in structural optimisation. Applications of EC in structural optimisation cover most algorithms and strategies in EC. To date, genetic algorithms remain the most commonly used method in this field. There are a few exceptions which use evolutionary strategies (Bohnenberger et al., 1995) (Murawski et al., 2000) and genetic programming (Yang and Kiong Soh, 2002) on discrete TOD problems. Recently, a few attempts were also made to use particle swarm method (Perez and Behdinan, 2007) and ant colony method (Kaveh et al., 2008) (Luh and Lin, 2009).

An approach based on GA for structural optimisation was introduced by Sandgren et al. (1990) to solve continuum TOD problems. For discrete TOD problems, a GA based approach was firstly proposed by Shankar and Hajela (1991). Development in the use of GAs in structural optimisation had been focused on improving the performance of the approaches through two major ways. The first direction is to combine GAs with other methods. For example, Sakamoto and Oda (1993) combined GA with optimality criteria method, Koumousis and Georgiou (1994) associate GA with logic programming, both to look for an optimal layout designs for truss structures, Soh and Yang (1996) developed a fuzzy logic controlled GA to search for optimal shape for truss structures, Ramasamy and Rajasekaran (1996) introduced the use of a GA and neural network based expert system for discrete TOD and sizing optimisation. The second direction is to make amendments or additions to canonical GA. For example, Cheng and Li (1997) applied Pareto GA to solve sizing optimisation of planar truss systems. In their method, a Pareto optimal subset is generated, from which a robust and compromise design can be selected. Another addition to canonical GA is the use of parallel GAs. Topping and Leite (1998), Sarma and Adeli (2001), Dimou and Koumousis (2003) all used parallel GAs to

improve the search. Another variation of canonical GA, known as non-dominated sorting genetic algorithm (NSGA) has also been used in structural optimisation, with examples being Deb and Goel (2001), Hamda et al. (2002b).

In recent years, creativity issues in structural design, which had not been addressed in the development of GAs in structural optimisation, started to draw attention. Studies are now focused on alternative ways to represent structural systems, i.e. representation techniques, to allow the representation of versatile structural systems in more sophisticate ways. Representation studies in structural design are still rare. Recent examples include topology description functions by de Ruiter and van Keulen (2000), morphological representation by Tai and Chee (2000) and Voronoi representation introduced by Hamda et al. (2002a). Detailed discussions on different representation methods that had been used in structural optimisation are given in Chapter 3. Generative representation initially proposed by Hornby (2003a) is the focus of this study. Issues of implementing generative representation in TOD are discussed and addressed in Chapter 4 and onwards.

Chapter 3 Representation

As stated in section 2.2.1, the choice of representation can significantly impact the performance of an algorithm. This chapter looks at some of the representation methods that have been used with evolutionary computation for topological reasoning and explores their advantages and limitations. Generative representations, as the focus of the study, are described in detail in the next chapter.

3.1 Parameter-based Representation

Parameter-based representation has been the main form of representation for most usage of EC in design. For most applications of such representation, solutions are explicitly described using a set of parameters representing dimensions, coordinates or a combination of both. Examples include Azid & Kwan (1999) who use real numbers X and Y to represent the locations of joints in order to find the optimum truss topology and Miles et al (2001) in whose work X and Y are used as coordinates to represent the locations of columns in building layout optimization.

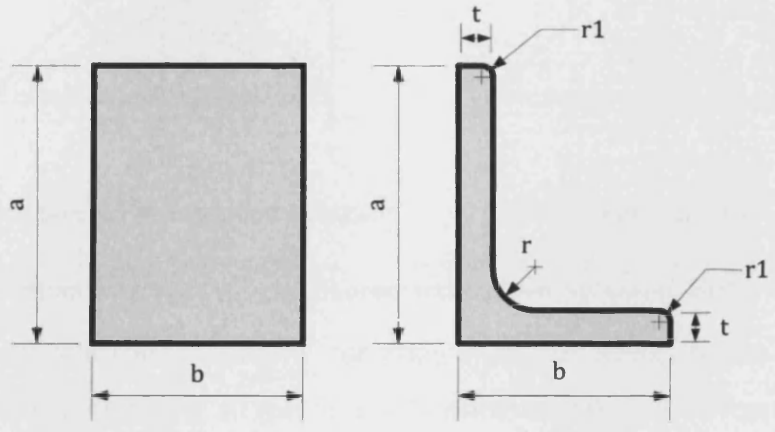


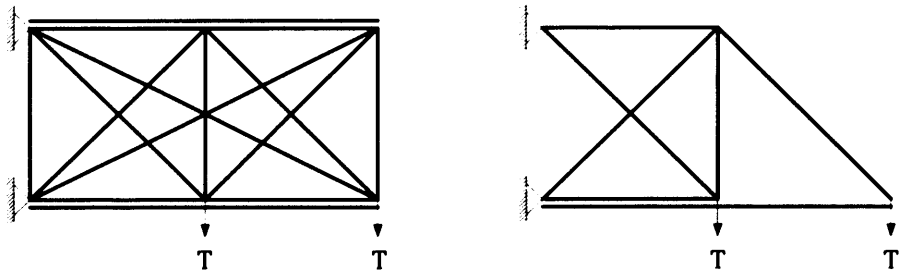
Figure 3.1 parameter-based representation

The advantage of parameter-based representation is its explicitness. Because parameters are directly used to describe solutions/topologies, the representation is usually straight forward and easy to employ. The explicitness also results in its limitation. As illustrated by Figure 3.1 (a), parameters a and b, with determined genotype to phenotype translation, are enough to represent a rectangle. However,

in order to allow the representation of more complicate shapes, such as the cross section of L-shaped steel (Figure 3.1 (b)), different and probably more parameters are required. In a domain where the topology of desired solution is unknown, to enable the representation of varied topologies through enumerating all necessary sets of parameters is practically impossible. Even if one can manage to reduce the parameter sets to a limited number, the algorithm would still struggle to recognize all of them and to cope with typical GA operators such as crossover. In a word, for domains where the form of solutions is reasonably fixed, parameter-base representation can be a possible and sometimes an efficient method of choice; other than that, it is quite limited.

3.2 Ground Structures

Representation using ground structures is mostly applied to optimization practice on truss-like structures. Examples include Hajela & Lee (1995), Azid and Kwan (1999) and Deb & Gulati (2001).



(a) 15-member, 6-node ground structure (b) optimized truss

Figure 3.2 ground structure representation (Deb and Gulati, 2001)

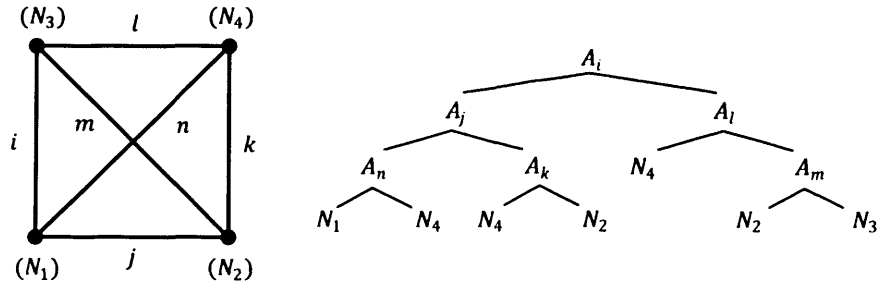
A ground structure is the "maximum" topology of a given design space. As shown in Figure 3.2 (a), by connecting all the six predetermined nodes a 15-member ground structure is formed. Each member of the ground structure m_i $(i \in [1,15])$ is given a starting cross-sectional area S_i $(i \in [1,15], S_i \geq 0)$ thus resulting in a fixed-length genome $S_1, S_2, \cdots, S_{15}$ that represents an initial individual. Using an evolutionary algorithm such as a GA, a population of such individuals is evolved to find the optimum solution. Mostly, the desired solution is a feasible solution with the minimum weight which is represented by the optimum set of cross-sectional areas. If a member's cross-sectional area is less than a given threshold S_{min} (a user-defined

small number), it is taken as "not existing". Hence, the optimized structure is often a "reduced" topology from the ground structure as shown in Figure 3.2 (b).

As indicated by its area of application, ground structure representation can be a method of choice to represent truss structures. It allows optimization to be carried out on both discrete (adding or removing members according to threshold S_{min}) and continuous (modifying cross-sectional areas in real number interval $[S_{min}, S_{max}]$, where S_{max} is a user-determined maximum acceptable cross-sectional area) levels. However, no matter how optimized a solution is, its topology still belongs to the topology of the ground structure. In other words, it prevents the possible optimized solutions from taking other forms than that defined by the ground structure. On the one hand, the use of ground structure simplifies the optimization problem by reducing its complexity to a much more manageable scale, which is largely based on one's knowledge of the problem domain. On the other hand, the restriction it imposes to the search could effectively prevent the algorithms from finding the real optimum solutions.

3.3 Graph-based Representation

Graph-based representation is another method that is mainly applied to the representation of trusses, e.g. (Yang and Soh, 2002) (Borkowski et al., 2003). An example of this representation by Yang and Soh (2002), who use GP (genetic programming) to search for optimum trusses, is given in Figure 3.3.



(a) a 6-member truss (b) the GP parse tree of the truss

Figure 3.3 graph-based representation (Yang and Soh, 2002)

As shown, a 4-node (N_1, N_2, N_3, N_4) 6-member (i, j, k, l, m, n) truss (Figure 3.3 (a)) is represented by a tree graph (Figure 3.3 (b)). The tree graph contains two kinds of

nodes connected by lines. The first kind is named as "inner nodes" and is denoted by $A_p(p = i, j, k, l, m)$. They are functions representing the cross-sectional areas of the members. The second kind is called the "leaves" denoted by N_q $(q = 1, 2, 3, 4)$ which represents the nodes of the truss. An inner node can connect two leaves. For instance, A_n connects N_2 and N_4, which means these two nodes in the truss are connected by member n whose cross-sectional area is represented by function A_n. An inner node can also connect one other inner node and a leaf, or connect two inner nodes. In these two cases, the upper inner node takes the "output" of each inner node it connects as one of its leaves. The output of an inner node is the node it connects to its lower-left corner if it has a connection line to its upper-right corner, or the node it connects to its lower-right corner if it has a connection line to its upper-left corner. For example, A_j connects two inner nodes A_n and A_k whose output is N_1 and N_2 respectively, which means that node N_1 and N_2 of the truss are connected via member j whose cross-sectional area is represented by function A_j.

Graph-based representation frees the representation of trusses from the use of ground structures, making it a more flexible and less problem-dependent approach (Yang and Soh, 2002). It does not rely on using a large number of nodal points to create ground structures that enable the production of complex truss structures. It allows the modulation of the connectivity between nodes while maintaining the possibility to vary the locations of nodes. Since no ground structure is used, graph-based representation requires little knowledge of the problem domain thus making it more capable to create efficient and innovative designs.

3.4 Voxel Representation

There is another type of topology that, unlike trusses, has a relatively large solid-void ratio. Instead of being composed of thin members that are connected through joints, they often take the forms of a solid piece of continuous area such as the cross section of a beam. Apparently, ground structure and graph-based methods are not suitable to represent such topologies while a representation method which is often referred to as voxel representation (the name based on its phenotype appearance) or bit-array representation (the name based on its genome format) is found to be

capable for such instance. Chapman et al. (1994), Baron et al. (1999), Griffiths & Miles (2003) and Wang & Tai (2005) all utilize this method. Figure 3.4 (Griffiths and Miles, 2003) illustrates how it works.

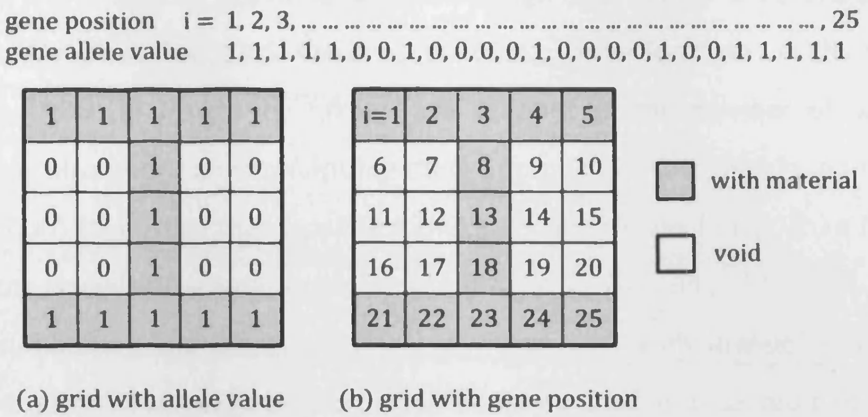


Figure 3.4 voxel representation (Griffiths and Miles, 2003)

As shown in Figure 3.4, with voxel representation, the design space is decomposed into a grid of identically sized squares called voxels. These voxels can be either filled with material or left void. To encode the designs, one can either use strings or arrays whose elements correspond to voxels according to positions. If a voxel is filled, its corresponding gene value is set to "1", otherwise to "0". The decoding process just works the opposite way.

It should be noted that, the resolution of the design space is determined by the voxels used. Higher definition requires finer grid, which means more voxels and consequently larger string or array size. It provides higher control over the design space but also increases the complexity of the problem being solved and the workload to handle encoded solutions. Since evolutionary algorithms such as GAs are highly population and iteration based, such increase can effectively compromise the efficiency of the algorithms. General practice is to look for a balance between the resolution and the resultant computational cost.

Voxel representation can represent any topology, with curved and non-horizontal/non-vertical edges being approximated by a series of steps. It can also be convenient for structural design and optimisation problems where finite element analysis (FEA) is necessary, because shapes/structures generated using this

representation already contain the required meshing which is defined by the voxel space. Unfortunately, besides topologies with jagged edges, the nature of the representation also allows the generation of isolated voxels and discontinuous topologies which are not desirable for most design optimizations. To overcome this drawback, Wang and Tai (2005) use an equality constraint function which emphasize the connectivity of designs by taking into account of the number of connected voxels. They also adopt a constraint handling approach further developed from that by Deb (2000) to ensure that feasible individuals are always better than infeasible ones in fitness value. Their method strengthens the survivability of feasible solutions, without eliminating the infeasible ones. It works well with low-definition design space where a relatively high proportion of the population is expected to be feasible designs with connected voxels. It becomes less competent when the problem requires a design space with more voxels.

3.5 Voronoi Representation

A representation method based on Voronoi diagrams is proposed by Hamda et al. (2002a). In computational geometry, a Voronoi diagram is a special kind of decomposition of a metric space determined by distances to a specified set of points called Voronoi sites in the space (Okabe, 2000) (Edelsbrunner, 2001). The decomposition is achieved through assigning to each of the Voronoi sites with a region of influence known as Voronoi region. Let $S \subseteq \mathbb{R}^2$ be the set of Voronoi sites, the Voronoi region of $p \in S$ is defined as:

$$V_p = \{x \in \mathbb{R}^2 \mid \| x - p \| \leq \| x - q \|, \forall q \in S\} \tag{3.1}$$

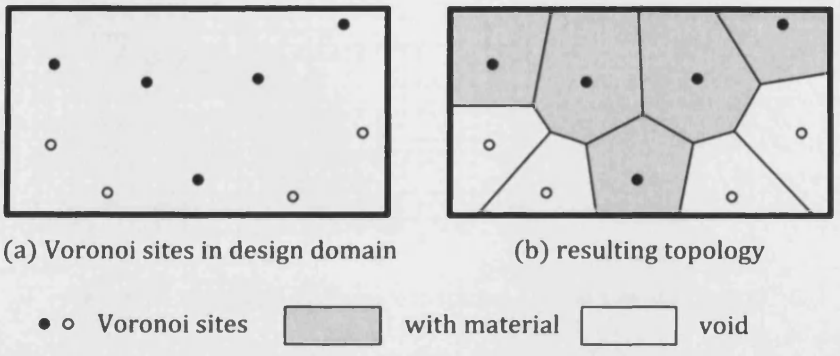


Figure 3.5 Voronoi representation (Hamda et al., 2002a)

To represent design topologies, $\mathbb{R}^2$ is replaced by a constrained design space which is a subset of $\mathbb{R}^2$. Similar to voxel representation, a characteristic value, either "0" or "1", is assigned to each of the Voronoi sites. If a Voronoi site is "1" (shown as • in Figure 3.5) its Voronoi region is filled with material; otherwise, is void. The resultant topology is then mapped into a predefined mesh for evaluation.

The most notable advantage of Voronoi representation over voxel representation is its self-adaptability, i.e. the complexity of the solutions can be autonomously adjusted by the algorithms. Unlike voxel representation that uses a fixed mesh of design space, Voronoi representation does not require a fixed number of Voronoi sites for each individual and is able to if it is necessary to increase the complexity of the representation to achieve desired solutions. The nature of the representation, however, still allows the generation of discontinuous topologies and infeasible solutions.

3.6 Morphological Representation

Work by Tai and his fellow researchers (Tai and Chee, 2000) (Tai et al., 2002) (Tai and Akhtar, 2005) uses a representation method referred to as morphological representation. It is named as such because it simulates the anatomical description of vertebrates. Figure 3.6 illustrates this representation.

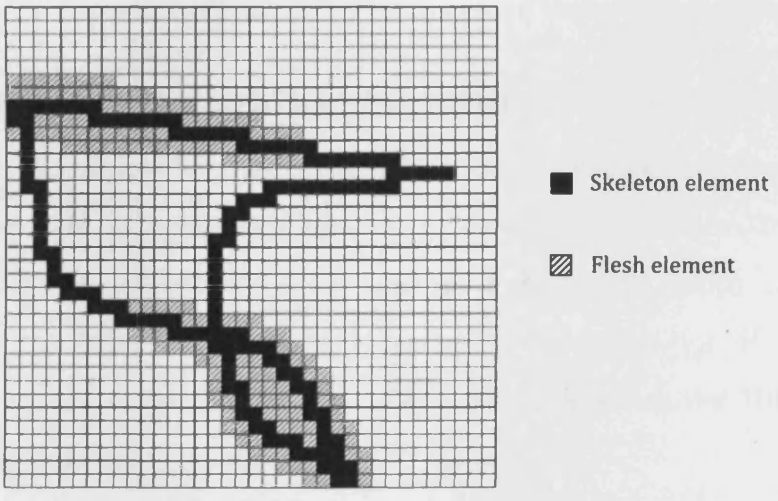


Figure 3.6 morphological representation (Tai and Akhtar, 2005)

As shown in Figure 3.6, the shape and topology of a structural continuum is represented by an arrangement of a "skeleton" and its surrounding "flesh" in a

decomposed design space just like that with voxel representation. The skeleton is generated using Bezier curves which are widely used in computer graphics to model smooth curves (Foley, 1997). A Bezier curve is defined by its start and end points with a series of control points in between. The skeleton is the set of elements through which each curve pass. Those elements are called "skeleton elements" around which all-round layers of "flesh elements" are added according to "thickness value". The start, end and control points of all curves plus the thickness values for all skeleton elements are then cast into a chromosome code to represent the entire shape and topology.

This representation method inherently ensures continuous topologies without isolated elements being generated. Tai and Akhtar (2005) also introduced a graph-theoretic chromosome scheme to be used with this representation to maintain the feasibility of the designs and to enhance the transmission of topological characteristics from parents to offspring during the use of typical GA operators such as crossover and mutation. The results presented in their work are all very simple topologies. Since the representation largely relies on the use of curves, one can imagined that to generate more complex topologies, a larger number of more complex curves should be used. However, it is questionable whether the complexity of the resultant topologies can scale-up the complexity of the representation and hence for the representation's capability to represent complex topologies.

3.7 Topology Description Function

In work by de Ruiter and van Keulen (2000) (2004), topologies are described using what is referred to as a topology description function (TDF). In a TDF approach, design variables are parameters that determine a function which can explicitly determines a topology. As shown in Figure 3.7, the superposition of a number of basis functions (a) forms a TDF (b). By using a cut-off level (c), the TDF is mapped into a topology (d).

TDF approach can literally describe any two-dimensional topologies; however, it does not guarantee continuous topologies. The capability of this representation is highly restricted when used with evolutionary algorithms such as GAs which rely on

population and iteration since it often requires on a large number of basis functions to describe complex topologies. However, traditional approaches such as gradient method are proved to be very suitable because the encoded topologies are in fact functions.

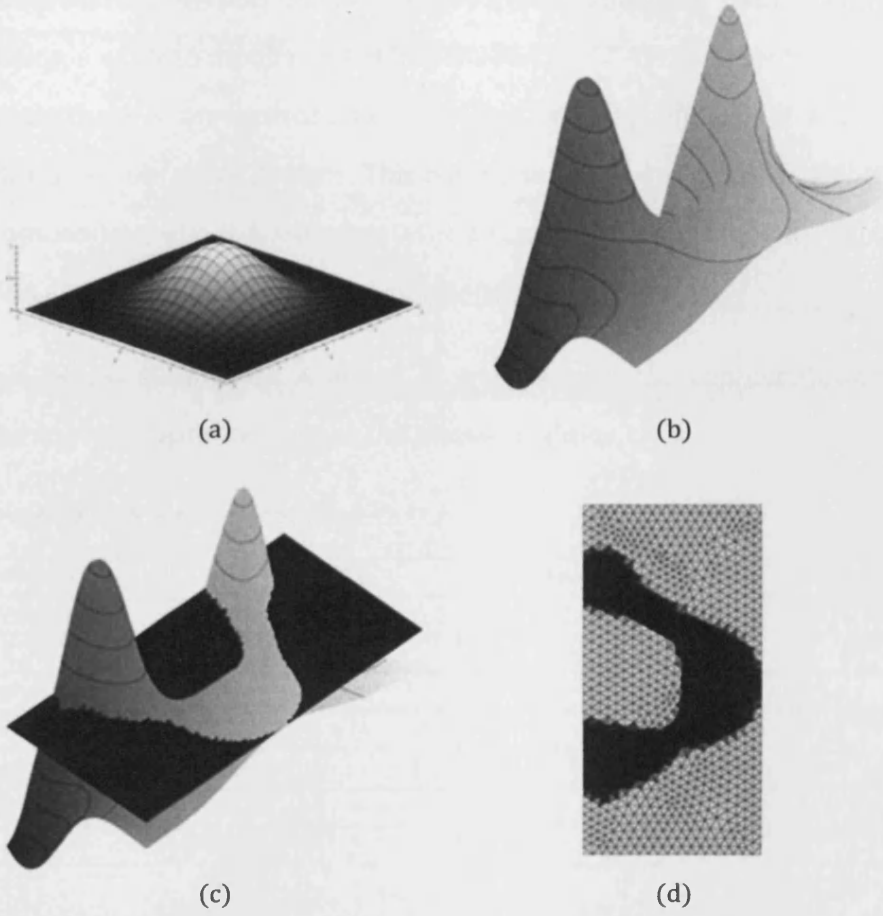


Figure 3.7 topology description functions (de Ruiter and van Keulen, 2004)

3.8 Summary

The review on representation method in this chapter is not intended to and certainly does not cover every approach that has been used. It does not point out the one best representation method. Supposedly, there is no such representation that is universally applicable and superior to the others. Through examining these selected methods, their limitations and advantages are revealed. More importantly, the review shows the most desired characteristics for a representation method which can be summarized as follows:

1. Flexibility – the representation should be flexible enough to represent a range of different solutions. This is crucial when it is to be used with evolutionary algorithms and especially for searching for innovative designs.
2. Feasibility – this refers to the resultant topologies. It is more favourable if a representation method does not easily create infeasible designs which require additional work to modify and refine them.
3. Compactness – a representation method is more efficient if it requires less material to represent designs. This can be in the forms of less variables, shorter chromosomes, etc. It is even better if a representation can adapt its complexity to the complexity of the desired solutions.

The table below summaries whether or not each of the representation methods discussed in this chapter possesses the above qualities.

Table 3.1 comparison of representations

Representation	Flexibility	Feasibility	Compactness
Parameter-based		•	
Ground Structure		•	
Graph-based	•	•	
Voxel	•		
Voronoi	•		•
Morphological		•	
TDF	•		

Unlike the above representation methods, the focus of this thesis, generative representation, has all of the three qualities and hence should be considered as a competitive alternative for representing topologies.

Chapter 4 Generative Representation

The mechanism and properties of generative representation is introduced in this chapter. Generative representation is a collective name for those representation schemes that have something in common, that is, being "generative". Before looking at the detailed implementation, this chapter starts with explaining the more general characteristics of generative representation. The specific implementation of a generative representation which is used in this thesis to represent two-dimensional topologies is then presented.

4.1 Non-generative V.S. Generative

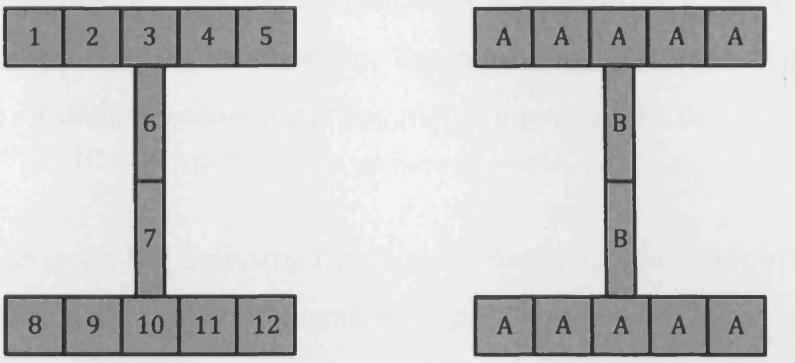
The one characteristic that distinguishes generative representation from non-generative representation is being generative which means the elements in the encoded designs can be reused in the process of translating to the actual designs. The actual form of the generative coding differs according to the specific representation schemes being employed, hence the elements being reused can also differ. Although a non-generative representations can also use elements to construct designs, such as the bits/voxels in a voxel representation, the bar members (encoded in cross-sectional areas) in a ground structure representation, and the Voronoi sites (encoded in coordinates) in a Voronoi-based representation, they do not intend to and cannot reuse their elements, which a generative representation, in contrast, can and always intends to do.

Figure 4.1 illustrates the difference between non-generative and generative presentations. As shown in Figure 4.1 (a), a non-generative representation relies on a total of 12 components to represent an I-shaped structure. The encoded design can take the following format where each number represents a component.

$[1,2,3,4,5]$	$[6,7]$	$[8,9,10,11,12]$
Top-flange	**Web**	**Bottom-flange**

In this non-generative representation, each of components exclusively represents one part of the actual design and is used only once in the translation. Although some

of them are identical in shape and size, they all refer to different components in the representation.



(a) non-generative representation (b) generative representation

Figure 4.1 non-generative representation and generative representation

To represent the same structure, a generative presentation as shown in Figure 4.1 (b) works in a different way. The representation uses two kinds of components ("A" and "B") and a sort of procedure to build the design by repeating these components. The encoded design looks more like a program shown in Table 4.1:

Table 4.1 generative representation as a program code

```
m = 0, n = 0;                          // initialize parameters

repeat if m < 5                        // build the flanges
{
     use component A at top;
     use component A at bottom;
     increase m by 1;
}
repeat if n < 2                        // build the web
{
     use component B in the middle;
     increase n by 1;
}
```

The reuse of components and the program-like data structure of encoded designs give generative representations some handy features. First of all, unlike non-generative representations which treat the components in the encoded data structures individually, generative representations are able to manipulate them as assemblies. For example, in order to change the thickness of the flanges of the structure shown in Figure 4.1, the non-generative representation needs to go through all the 10 flange components and to make the exact same changes to each

of them which can be very difficult to achieve with search algorithms such as GAs due to randomness. The generative representation only needs to apply proper change to component "A" and the same change is automatically made to all the other flange components as an assembly. This can be an advantage of generative representations if simultaneously modifying multiple parts of the design is a desired feature.

Secondly, the program-like data structure has the ability to use control-flow which introduces abstraction and compactness into the representation. Like most programming languages, the two forms of control-flow in generative representation are conditionals and iterations both of which are able to take parameters. As shown in Figure 4.1 (b) and Table 4.1, the generative presentation actually expresses the procedure that generates the design. The conditionals and iterations, in fact, describe this procedure in the abstract. It grants the representation a different way to navigate the search space through abstraction of procedures. In other words, whether or not a certain procedure is to be executed and, for how many times it is to be executed, are both encoded. The use of control-flow also grants the representation compactness as the representation is able to adapt to the complexity of the problem through modifying the parameters of its conditional and iterative expressions. It should be noted that, the full benefits of compactness are not apparent in the above example because of its simplicity. It is only for more complex problems that the true benefits of compactness become apparent.

4.2 Explicit V.S. Implicit

Generative representation can be explicit or implicit. For explicit generative representation, a design is represented using "meaningful" components served as "building blocks" such as the flange component and the web component used in the example shown in Figure 4.1 (b). These components directly become a part of the design thus the assembly of them explicitly represents the design. Implicit generative presentation, in comparison, requires a sort of transformation in translating encoded designs to actual designs. Although it may also include the use of design components, it relies on a set of design rules that interact to construct a design. To choose between explicit representation and implicit representation, one

needs to consider the available design-specific knowledge and the objective of the problem.

4.3 Previous Work

As a relatively new approach, applications of generative representation are rarely found within structural engineering the exceptions being Rosenman (1996, 1997, 1999) who proposed a hierarchical grammar for building floor plans which can be considered as an attempt to use explicit generative representations and the more recent work by Kicinger et al. (2005a) who use cellular automata to generate design concepts of steel structures in tall buildings which is an example of using a implicit generative representation.

The works by Hornby (2001, 2002, 2003b, 2004) including his PhD thesis (2003a) are probably the most representative works on generative presentation within the discipline of design automation. In his PhD thesis, applications of generative representations on the design of voxel structures (tables), neural networks and robots are presented with considerably good results. Figure 4.2 shows one of the best table designs evolved using a genetic algorithm and a generative representation by Hornby.

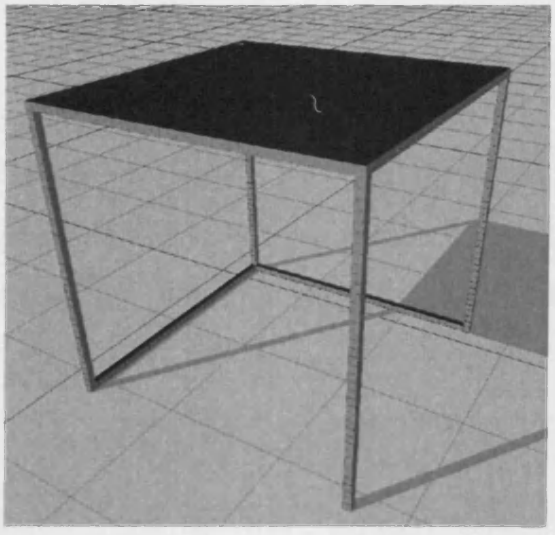


Figure 4.2 a table evolved using generative representation (Hornby, 2003a)

Although the works by Hornby generate confidence in generative representation by providing good experimental results, there is very inadequate description and

discussion about the detailed implementation of the representation in connection with the search algorithms. To others, the method and software Hornby describes in his thesis work like a black box. Generative representation is a complex subject. It comes with new challenges which Hornby's works fail to address or discuss. For the representation to be better understood and used, these challenges have to be examined and dealt with in this work. Discussions about these representation-specific challenges are to be found throughout the rest of this thesis.

4.4 A Generative Representation

In this section, an implicit generative representation scheme is presented. It is used as the representation of choice in this thesis. The representation is based on Lindenmayer systems (Lindenmayer, 1968) and turtle graphics (Abelson and DiSessa, 1981), and it is capable to represent any 2D voxel structures. Although the scheme can be extended to represent 3D voxel structures (Figure 4.2), the study presented in this thesis decided to use a less complicated 2D setup. This is because the object of the study is to reveal the properties of this generative representation and it can be better achieved without being obstructed by the complexity of the problem being solved.

4.4.1 L-systems and Turtle Graphics

A Lindenmayer system or L-system is a parallel rewriting system introduced and developed by Lindenmayer. A rewriting system consists of a set of symbols and a set of rules according to which the symbols are replaced. Beginning from rewriting a starting symbol, a complex string is created by iteratively applying the rules to existing symbols. For example, a simple L-system is shown in Table 4.2.

Table 4.2 a simple L-system

$Symbols : A, B$ $Start : A$ $Rules : (A \rightarrow AB), (B \rightarrow A)$

According to the L-system given by Table 4.2, during rewriting, a symbol '*A*' of an existing string will be replaced by symbols '*AB*', whereas a symbol '*B*' will be replaced by a symbol '*A*'. It should be noted that, as defined by the L-system, the

result of rewriting a symbol '*A*' has to be '*AB*' which has to be in that particular order. In other words, the order of symbols is meaningful for an L-system, which ensures that for a given times of rewriting, the result is unique. For example, the L-system given by Table 4.2 will always produce the following strings given by (4.1) for each time (n) of rewriting.

$$\begin{aligned} n &= 0 : A \\ n &= 1 : AB \\ n &= 2 : ABA \\ n &= 3 : ABAAB \\ n &= 4 : ABAABABA \\ n &= 5 : ABAABABAABAAB. \end{aligned} \tag{4.1}$$

A more complicate L-system, known as parametric L-system (Prusinkiewicz and Lindenmayer, 1990), is defined as an ordered quadruplet $G = \langle V, \Sigma, \omega, P \rangle$, where

- V (variables) is the alphabet (set of symbols) of the system that can be replace in writing;
- Σ (constants) is the set of formal parameters which are symbols that remain fixed;
- ω is a nonempty parametric word called the *axiom* which is a string of symbols from V defining the initial state of the system;
- P is a finite set of productions or production rules defining the way variables can be replaced with combinations of constants and other variables.

In a parametric L-system, a production consists of three components – the *predecessor*, the *condition* and the *successor* which are separated using symbols : and $\rightarrow$. For example, a production with predecessor $A(x, y)$, condition $y \leq 3$ and successor $A(x * 2, x + y)B(x)C$ is written as

$$A(x, y) : y \leq 3 \rightarrow A(x * 2, x + y)B(x)C, \tag{4.2}$$

and an example of a parametric L-system is given in Table 4.3.

Table 4.3 a parametric L-system

$\omega : B(2)A(4,4)$ $P_1 : A(x,y) : y \leq 3 \rightarrow A(x*2, x+y)B(x)C$ $P_2 : A(x,y) : y > 3 \rightarrow B(x)$ $P_3 : B(x) : x < 1 \rightarrow C$ $P_4 : B(x) : x \geq 1 \rightarrow B(x-1)$

L-systems can be used with turtle graphics to create many interesting images including fractal plants. Turtle graphics are a computer graphics term for a method of programming vector graphics using a relative cursor (the "turtle") upon a Cartesian plane (Abelson and DiSessa, 1981). Graphs are drawn by controlling the movement of the turtle using commands that are relative to the position of the turtle, such as "move forward 2 steps" or "turn right 90 degrees". For example, assuming the head of the turtle points to the right at the start, the following ordered commands:-

- repeat 3 times of:
    - move forward 2 steps
    - turn right 90 degrees
- move forward 4 steps

generate the following command string:-

$$FFRFFRFFRFFFF, \tag{4.3}$$

where F directs the turtle to move one step forward and R makes it to turn 90 degrees to its right. The command string controls the turtle to generate the figure shown in Figure 4.3.

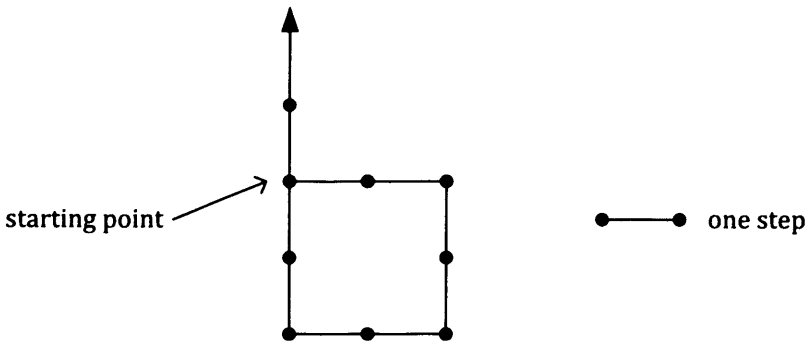


Figure 4.3 turtle graphics

The commands in turtle graphics can be coded into an L-system as its symbols and the string of commands to control the movement of the turtle can be generate through rewriting. Table 4.4 lists some example commands that can be used as L-system symbols to generate turtle graphs.

Table 4.4 example L-system symbols for turtle graph generation

Symbol/Command	Description
$F(n)$	move n steps forward
$R(\theta)$	turn θ degree(s) to the right
$L(\theta)$	turn θ degree(s) to the left

With the commands listed in Table 4.4, an L-system given in Table 4.5 generates turtle graphs shown in Figure 4.4 for each time (n) of rewriting.

Table 4.5 an L-system for turtle graph generation

$Symbols: F(1), R(90), L(90)$
$Start: F(1)$
$Rule: F(1) \rightarrow F(1)L(90)F(1)R(90)F(1)R(90)F(1)L(90)F(1)$

The L-system given in Table 4.5 is a very simple one. The parametric L-system described earlier in this section can be used to generate more complex turtle graphs. The generative representation studied in this thesis uses parametric L-systems which will be treated in more details in Chapter 5.

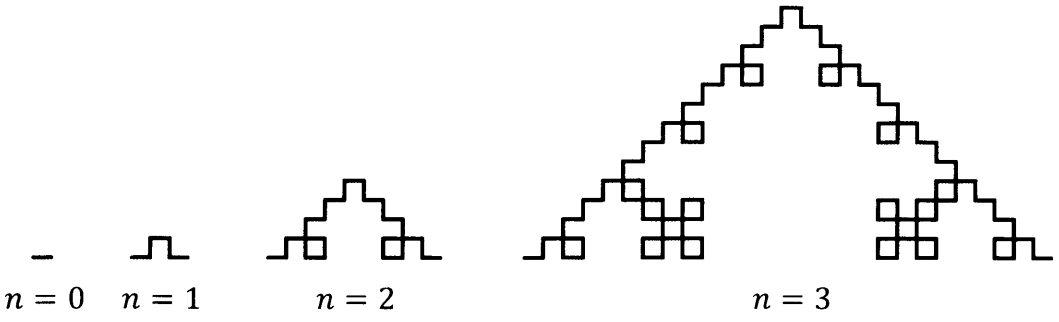
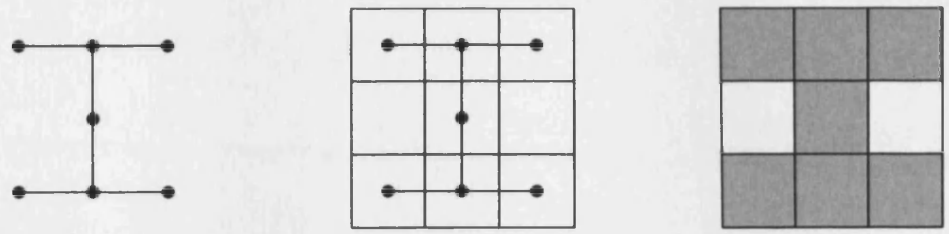


Figure 4.4 turtle graphs generated by an L-system

4.4.2 Representing Voxel Topologies

This thesis uses generative representations based on parametric L-systems and turtle graphics to represent voxel topologies and studies the properties of such representations. As described in Section 4.4.1, the resultant turtle graphs by rewriting an L-system are curves that are composed of connected line segments.

These curves can be used to define voxel topologies through a translation process shown in Figure 4.5.



(a) turtle graph (b) mapping turtle graph into decomposed design space (c) resultant voxel topology

Figure 4.5 from turtle graph to voxel topology

As shown, a voxel topology is generated by mapping a turtle graph into a decomposed design space and filling the voxels that have line segments pass through them with "solid" material. Such voxel topologies are continuous by nature because the turtle graphs are assemblies of connected line segments. This eliminates the generation of isolated voxels which is considered to be a drawback of voxel (bit-array) representation (Zhang, 2004) .

The detailed translation process can vary. The same turtle graph can be translated into different voxel topologies if different strategies are used. For example, whether or not multiple line segments (not overlapped with each other) are allowed to pass though a same voxel, how to determine the anchor of a turtle graph (a base point with respect to the turtle graph and used as a handle of the graph) and the insertion point (a point in the design space where the anchor of the turtle graph is placed), etc. can all influence the resultant voxel topologies. This part of the implementation of generative representation is not looked into and discussed in the literature. This thesis addresses it in details in Chapter 5.

4.5 Summary

As described in this chapter, generative representation is different from conventional non-generative representation with the ability to reuse components in translating the encoded designs to actual artefacts. It also has the characteristics of being compact and abstract, which is induced by control-flow. To convert these

differences into advantages, the properties of generative representation need to be understood by examining its detailed implementation and the challenges wherein.

Chapter 5 Encoding & Decoding

This chapter looks into the encoding and decoding process of using generative representation to represent two-dimensional shapes. In brief, individuals are encoded as L-systems which can be decoded into shapes/designs by rewriting these L-systems and then mapping the resultant turtle graphs into a decomposed design space. The detailed implementation can vary and effectively influence the generated designs.

5.1 Individuals as L-systems

5.1.1 Symbols

Besides the common turtle graphic commands listed in Table 4.4, the L-systems used in this thesis also include other symbols that provide additional utilities. Symbol "[" is "push" command to save the current status (position and orientation) of the turtle. Symbol "]" is "pop" command that retrieves the most recently saved status and restores the turtle to that status. The use of push and pop symbols enables the easy generation of turtle graphs with branches. Symbols "{" and "}" are used to enclose a block of symbols to be replicated according to the parameter they take. For example, assuming the turtle starts from $(0, 0)$ and points to the top of the page, $F(n)$ means moving forward n step(s), $R(n)$ or $L(n)$ means turning right or left $n \times 90$ degrees, the command string $\{F(1)[R(1)F(1)][L(1)F(1)]\}(2)$ directs the turtle to take the following actions listed in Table 5.1 and produces the graph shown in Figure 5.1.

Table 5.1 actions by command string $\{F(1)[R(1)F(1)][L(1)F(1)]\}(2)$

Step	Symbol	Action	Position	Orientation
1	$\{$	position to the start and prepare to replicate the following steps	$(0,0)$	↑
2	$F(1)$	move one step forward	$(0,1)$	↑
3	[	save current status	$(0,1)$	↑
4	$R(1)$	turn 90 degrees to the right	$(0,1)$	→
5	$F(1)$	move one step forward	$(1,1)$	→
6	]	restore to the most recently saved status	$(0,1)$	↑
7	[	save current status	$(0,1)$	↑
8	$L(1)$	turn 90 degrees to the left	$(0,1)$	←
9	$F(1)$	move one step forward	$(-1,1)$	←
10	]	restore to the most recently saved status	$(0,1)$	↑
11	$\}(2)$	repeat step 2: forward	$(0,2)$	↑

12		repeat step 3: save status	(0,2)	↑
13		repeat step 4: turn right	(0,2)	→
14		repeat step 5: forward	(1,2)	→
15		repeat step 6: restore to saved status	(0,2)	↑
16		repeat step 7: save status	(0,2)	↑
17		repeat step 8: turn left	(0,2)	←
18		repeat step 9: forward	(−1,2)	←
19		repeat step 10: restore to saved status	(0,2)	↑

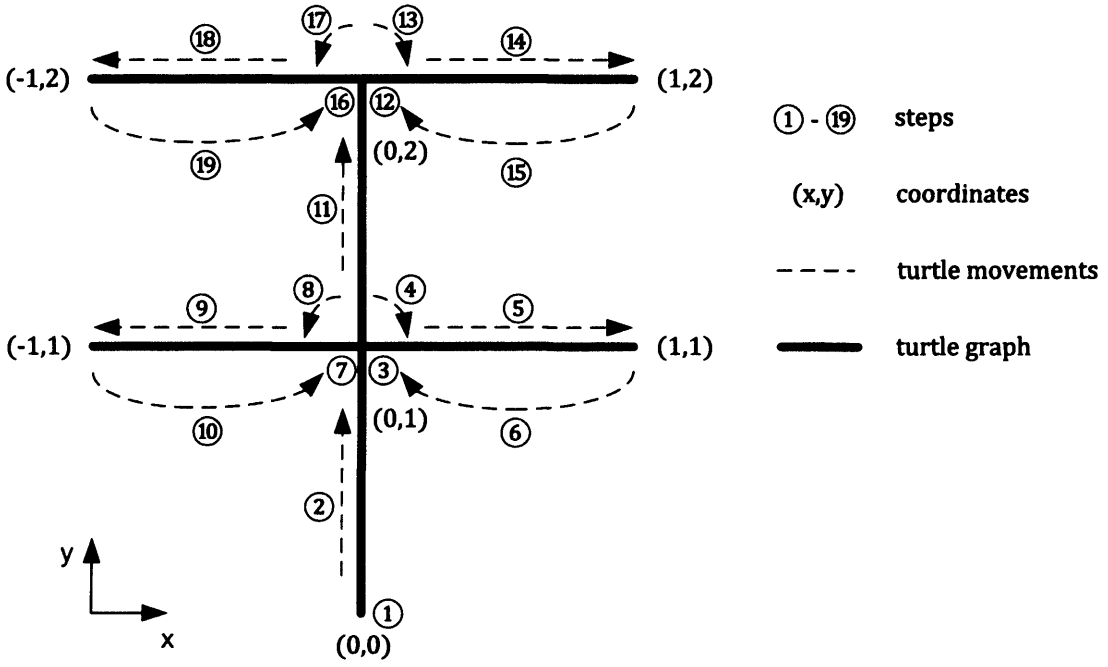


Figure 5.1 turtle graph with branches

The symbols described above are all constant symbols which remain fixed. Production symbols $P_1, P_2, \cdots, P_n$ denote production rules according to which production symbols are replaced during rewrite. Unlike symbols F, L, R and block symbols who take single parameter, production symbols can take one or multiple parameters. A complete list of symbols for the L-systems used in this thesis is given in Table 5.2.

Table 5.2 list of L-system symbols

Symbol	Description	Number of Parameter	Constant
$F(n)$	move n step(s) forward	1	Yes
$L(n)$	turn left $n \times 90$ degree(s)	1	Yes
$R(n)$	turn right $n \times 90$ degree(s)	1	Yes
[	push to saved the current status	0	Yes
]	pop to restore the most recently saved status	0	Yes

$\{$	start of a block	1	Yes
$\}(n)$	end of a block which is to be replicated n time(s)		Yes
$P_i(N_i)$	production rule P_i with its parameter set being N_i	≥ 1	No

5.1.2 Parameters

It should be noted that, the term "parameter" used here refers to that defined in computer science and used in programming context. From an engineering point of view, these "parameters" are actually variables because their values can and are changed during processing. In this thesis, they are still called parameters to follow the convention in related work. Parameters taken by symbols can take the following three forms:

- Real numbers, e.g. $F(1)$, $L(2)$,
- Variables passed by productions, e.g. $P_i(N_i)$: $F(n_0)\cdots, (n_0 \in N_i)$, or
- Algebraic expressions of the above two kinds of parameters, e.g. $P_i(N_i)$: $F(2 \times n_0)L(n_0 + n_1)\cdots, (n_0, n_1 \in N_i)$.

The use of parameters is an important feature of parametric L-systems which introduces abstraction and control-flow into the representation. While parametric symbols like $F(n)$, $L(n)$ and $R(n)$ can enable simple abstraction, parametric block symbols $\{\cdots\}(n)$ make it possible to abstract complex clusters. Theoretically, the value of these parameters can be any real number with an exception being the parameter taken by block symbols for which only non-negative integers make sense. In practice, a reduced value space is often applied since allowing parameters to take any real-number value is not only unnecessary but also undesirable in certain circumstances. Restriction for parameter values is discussed in detail in Section 5.4.1.

5.1.3 Production Rules

As previously stated, a production rule for a parametric L-system has three components, namely *predecessor*, *condition* and *successor*. A predecessor indicates the symbol to be replaced. If the parameter(s) taken by the symbol satisfies the condition, the symbol will be replaced by the corresponding successor during rewrite. The same predecessor can have different condition-successor pairs. An example is given below.

$$P_0(n_0, n_1) : \begin{matrix} n_0 \leq 5 \rightarrow F(n_0)P_1(n_1 + 1, n_0)R(1) \\ n_1 \geq 10 \rightarrow P_3(n_0, 1) \end{matrix} \quad (5.1)$$

This feature facilitates control-flow as a same symbol can be rewritten using different rules according to the parameter(s) it takes. It is also possible to control whether or not a symbol is to be rewritten because the parameter(s) may or may not satisfy any of the conditions.

5.1.4 Format

In this thesis, all encoded individuals in the form of parametric L-systems follow the same format as defined by the C++ structure data type **Individual** as shown in Table 5.3.

Table 5.3 structure Individual

```
struct Individual
{
      // index of starting production:
      int p;
      // array of initial production parameters:
      int n[NUM_PARA];
      // proposed number of rewrites:
      int rw;
      // array of conditions:
      Condition Cond[NUM_PROD][NUM_PAIR];
      // array of successors:
      Symbol Succ[NUM_PROD][NUM_PAIR][NUM_SYMB];
}
```

Both **Condition** and **Symbol** are structure data types whose definitions are given in Table 5.4 and Table 5.5. Explanations of constants are given in Table 5.6.

Table 5.4 structure Condition

```
struct Condition
{
      // indicator of production parameter type:
      char para;
      // relation (0 indicates '≥', 1 indicates '≤'):
      char r;
      // value that para is compared to:
      int v;
}
```

Table 5.5 structure Symbol

```
struct Symbol
{
      // symbol type
      char t;
      // production index (only useful if symbol is a production)
      int p;
      // array of parameter types
      char para[NUM_PARA][2]
      // array of parameter values
      int value[NUM_PARA][2]
      // array of operators (only useful when the parameters are
      // algebraic expressions)
      char operator[NUM_PARA]
}
```

Table 5.6 constants

Constant	Explanation
NUM_PARA	the number of parameters a production is designed to take
NUM_PROD	the number of different production symbols for an individual
NUM_PAIR	the number of condition-successor pairs for each production symbol
NUM_SYMB	the maximum number of symbols for each successor

Using the format described above, an individual in the form of an L-system can be initialized though the following steps:

1. A production symbol is randomly selected as the starting symbol.
2. Initialize the parameter(s) of the starting symbol by generating random numbers with certain restrictions detailed in Section 5.4.1.
3. Randomly decide the number of proposed rewrites. This is the maximum number of rewrite that will happen before it automatically stops. Note that the rewriting may be forced to stop before reaching this number due to other restrictions detailed in Section 5.4.2.
4. Generate condition array `Cond`. This is a `NUM_PROD` × `NUM_PAIR` array with its variable type being structure **`Condition`**. For example, `Cond[1][2]` means this is condition No. 2 for production symbol No. 1. Noting that the index starts from 0, it is actually the third condition for the second production symbol in index order. The generation of an array element has three steps to follow. First, choose a parameter of the production symbol to be considered by randomly selecting an integer within the range of [0, `NUM_PARA`) as the parameter index. Second, randomly decide the relation type (either $\geq$ or $\leq$) for this condition.

Third, generate a random integer and use it as the value that the parameter value is compared to. For example, structure $\{0,0,4\}$ yields condition $n_0 \geq 4$. Iteration continues until array `Cond` is fully initialized.

5. Generate successor array `Succ` which is a `NUM_PROD` × `NUM_PAIR` × `NUM_SYMB` array of **`Symbol`** type variables. For example, `Succ [0][1][2]` is the third symbol of the second successor of the first production. Structure **`Symbol`** is designed as such (Table 5.5) so that it is able to cope with any of the types of symbols of the proposed L-system. For different symbol and parameter types, different members of the structure become active accordingly. Hence, the first step to generate a symbol is to choose a type from those listed in Table 5.2. Note that the selection of symbol types is not completely random because special care need to be taken to avoid creating illegal successors due to inappropriate use of push/pop and block symbols. Different strategies can be adopted to ensure legal generation of successors, which are detailed and discussed in Section 5.4.3. The next step is to generate the parameter(s) for the symbol if it is eligible to take any. A parameter can take any of the three forms stated in Section 5.1.2 and should be generated according to the form taken. Again, this is not a completely random process. Restrictions discussed in Section 5.4.1 should be applied. Iterative symbol generation continues until the array `Succ` is fully initialized. Note that `NUM_SYMB` is the maximum number of symbols per successor and it is not always necessary to reach this limit. Dumb symbols are randomly inserted into the array just to take up places so that successors with various length can be generated.

Despite of certain restrictions, the generation of encoded individuals can be considered as a random process while maintaining the same format among all generated individuals. As an example, the format is presented by Figure 5.2 assuming that:

1. the L-system consists of 15 production symbols each of which has 3 condition-successor pairs and takes 2 parameters,
2. each of the successors consists no more than 15 symbols, and
3. rewrite starts from $P_3(7,1)$ and repeats 5 times maximum.

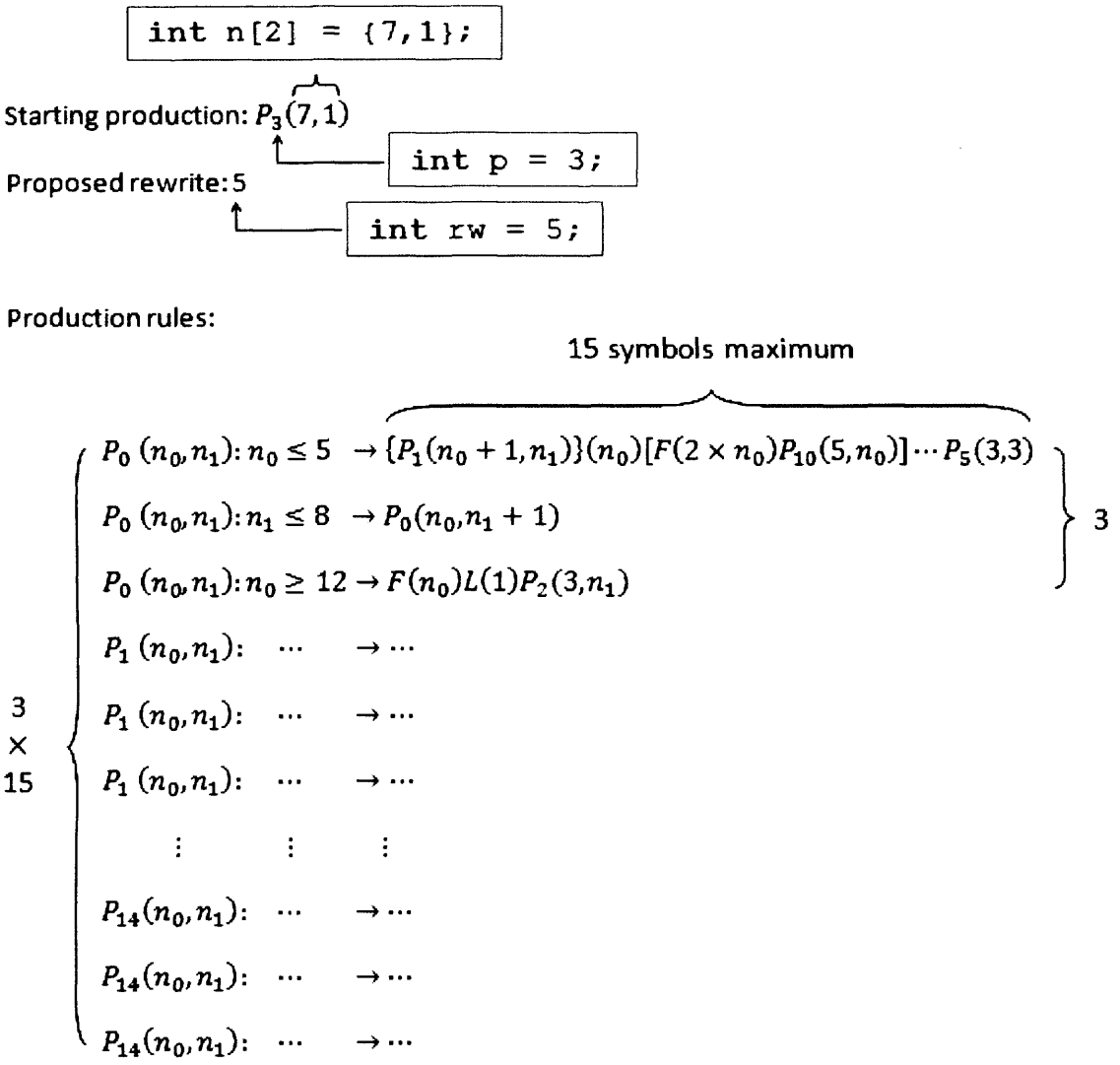


Figure 5.2 individual format

5.2 Rewrite

A turtle graphic command string is generated by replacing the production symbols with their corresponding successors. This is also the process often referred to as 'rewrite'. The rewrite starts from replacing the starting production according to its initial parameters. The following rewrites act in a scan-and-replace manner. For each rewrite, it firstly scans for production symbols by going through the current string from the first symbol to the last. Whenever a production symbol is found, it checks the production's parameters to see if any condition is satisfied. If a condition is satisfied, it then replaces the production symbol with its corresponding successor into which parameter values are substituted. Note that replacing production symbols with their successors may introduce new production symbols. Those new

productions are ignored for the current rewrite and will be checked in the next possible rewrite. Those old productions that do not satisfy any of the conditions remain unchanged.

During the rewrite, it is possible for a production to satisfy more than one condition. In such case, the first satisfied condition and its corresponding successor is always chosen to ensure constant rewriting results.

When rewrite is terminated, either by reaching the proposed times or by meeting any of the other conditions that are to be discussed in Section 5.4.2, all production symbols are then removed from the string. The resultant string which only contains meaningful construction commands ('$F(n)$', '$L(n)$', '$R(n)$', '[', ']', '{' and '}(n)') is the final command string for turtle graph generation.

5.3 Mapping Mechanisms

Turtle graphs are drawn according to the command strings generated by rewriting individual L-systems. Figure 5.3 shows one of those turtle graphs created from a randomly generated individual that follows the exact format shown in Figure 5.2. As shown in Figure 5.3, the turtle starts from point A and draws a graph (red lines) that ends at point B. The example turtle graph has a span of 72 steps in both the horizontal and the vertical directions. So if the coordinates of its lower-left corner C1 are $(0,0)$, those of its upper-right corner C2 are $(72,72)$. This turtle graph is produced by a randomly generated, stand-along individual. It is presented here only to illustrate and compare the different mapping methods.

The turtle graph, consisting of line segments, has to be mapped into a decomposed design space to generate the actual design. Figure 4.5 only illustrates the basic idea of this process. In fact, there are different strategies for the detailed implementation to follow, which can effectively cause different designs to be generated from the same individual.

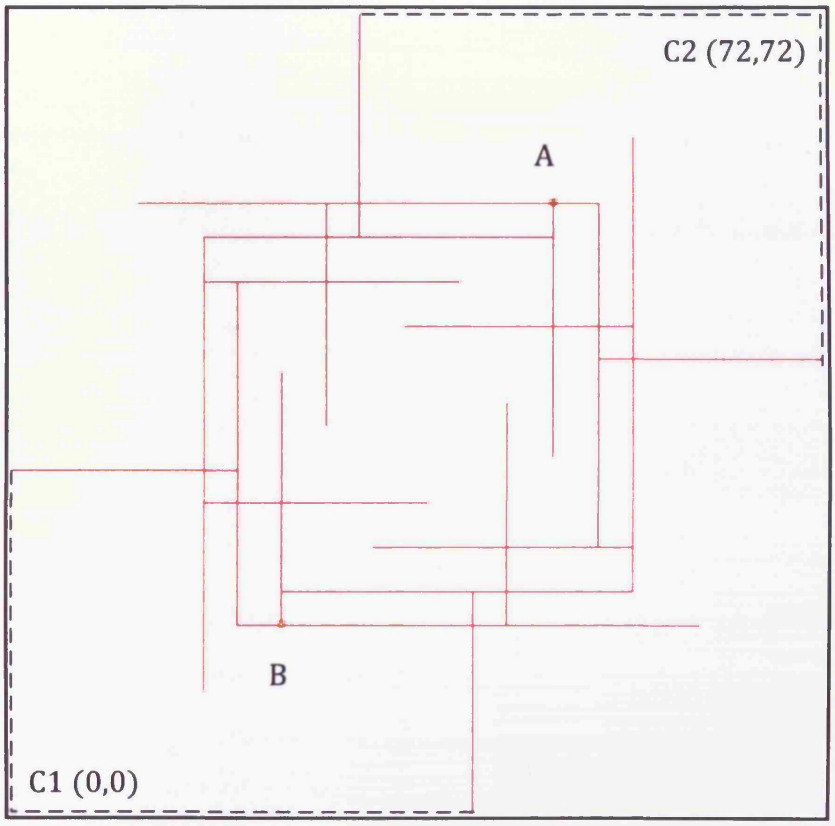


Figure 5.3 turtle graph of a randomly generated individual

5.3.1 Static Mapping

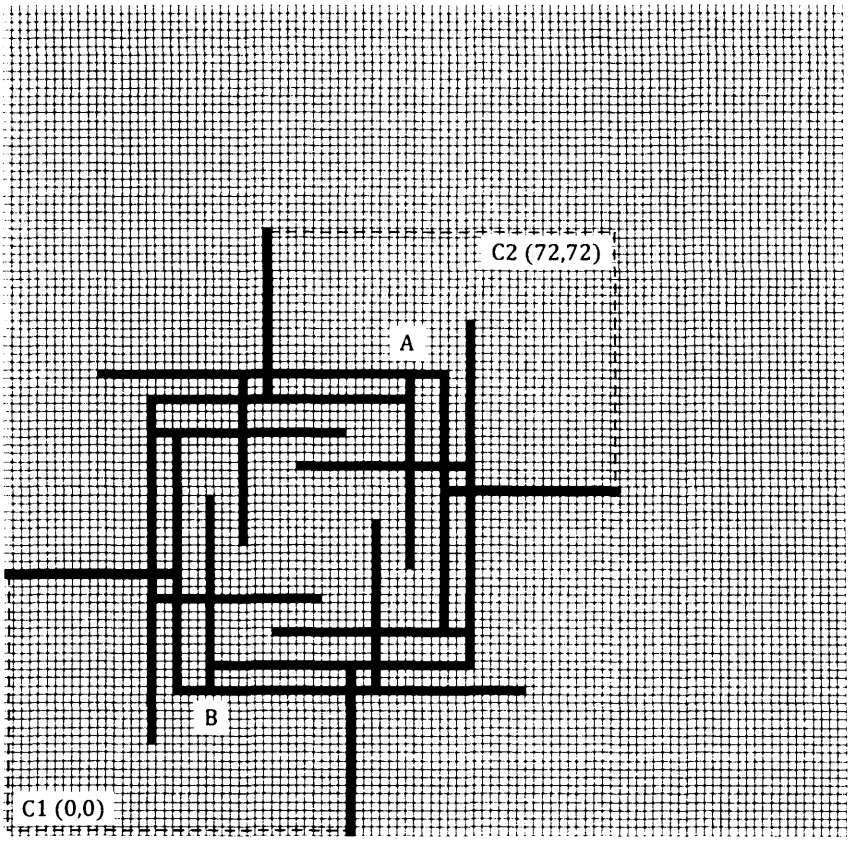


Figure 5.4 static mapping

The first possible strategy is the one used by Hornby (2003a). In his method, he uses a fixed design space with the size of its voxels matching the step of the turtle graphs. This means if the turtle move one step forward ($F(1)$) from the center of a voxel, it will end up at the center of an adjacent voxel. The turtle always starts from the center of a voxel and all voxels with the line segments passing though are filled with material. Using this method, the turtle graph shown in Figure 5.3 generates the design shown in Figure 5.4.

Due to randomness, there are chances that turtle graphs generated from individuals exceed the pre-defined design space. In Hornby's work, there is no mention of how to cope with this situation. However, since the size of turtle graphs generally increases with the number of rewrites, one can always force the rewriting to stop if such situation is observed, place penalties on the fitness values of such individuals or consider them as illegal and delete them.

In this thesis, the above method is referred to as 'static mapping'. It is static in the sense that both the voxel size and the turtle graph step, scale statically to the real-world dimension. The scale does not change throughout the entire search progress. The reset of this section will introduce two alternative methods that are referred to as 'semi-static mapping' and 'dynamic mapping'.

5.3.2 Semi-static Mapping

Semi-static mapping involves dynamic scaling which is applied to turtle graphs. Although the design space is still decomposed into static voxel world, the turtle graphs generated are freed from static scale and can be scaled up or down to fit into the grid. The example shown in Figure 5.5 uses a 20×20 grid which has a static scale to the real-world dimension. It means if the grid represents a 100 mm × 100 mm design space, each of the voxels represents a square of 5 mm × 5 mm in size. This grid and its scale to the real-world design space remains the same for turtle graphs of any size to be mapped into it. The size of a turtle graph can be measured in turtle graph steps. For example, the one shown in Figure 5.3 has a span of seventy-two steps in both horizontal and vertical directions. To map it into the grid, the actual dimension of one step needs to be known. In the static mapping described above, one step represents the same dimension as the side length of a voxel and it remains constant. For semi-static mapping, this scale changes according to the size of the turtle graphs in step. For example, to map the same turtle graph tightly into the grid shown in Figure 5.5, each step equals to $\frac{20}{72}$ of the side length of a voxel and represents $5\times\frac{20}{72}$ mm in real-world. To generalize it, supposing the span of a turtle graph is s_h steps in horizontal and s_v steps in vertical, the side lengths of the design space are S_h and S_v respectively, the real-world dimension d that one step of the turtle graph represents can be described by (5.2).

$$d = \min\left(\frac{S_h}{s_h}, \frac{S_v}{s_v}\right) \tag{5.2}$$

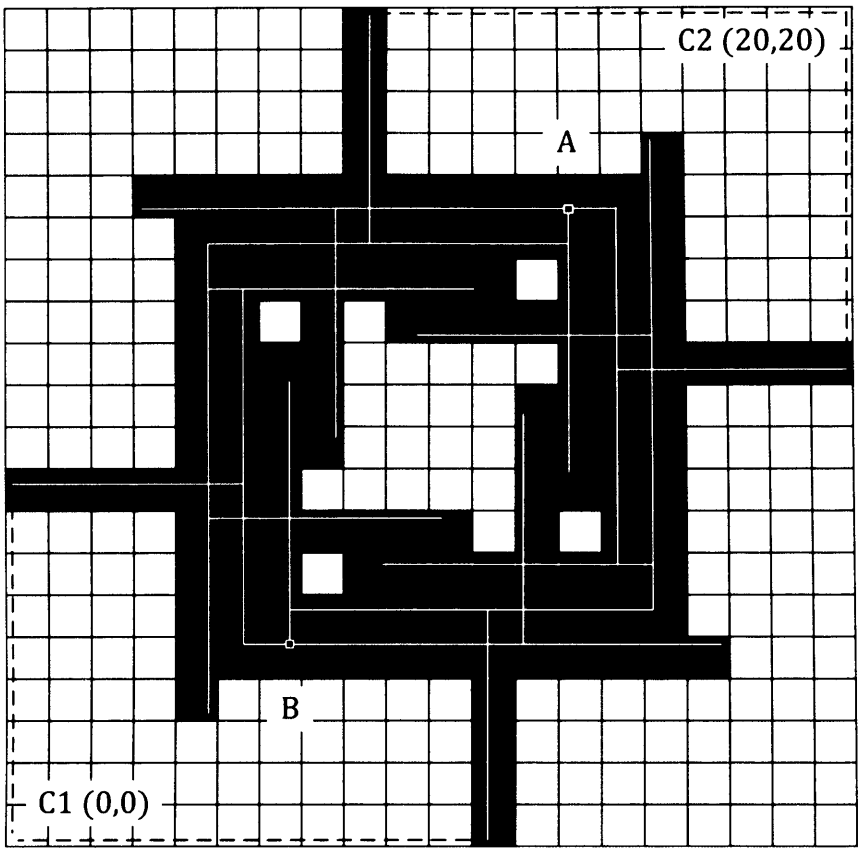


Figure 5.5 semi-static mapping

5.3.3 Dynamic Mapping

In contrast with static and semi-static mapping, dynamic mapping use neither static grid nor static step scale. However, the scale of the step against the size of the voxel remains as a constant which, in the case of Figure 5.6, is 1. The grid shown in Figure 5.6 covers the entire design space. The size of the turtle graph (in step) and the scale between the step and the size of the voxel determine how fine the grid is. As the scale remains constant, the larger the turtle graph is, the finer the grid has to be. As shown in Figure 5.6, a grid of seventy-three voxels by seventy-three voxels is dynamically generated for the turtle graph to fit into.

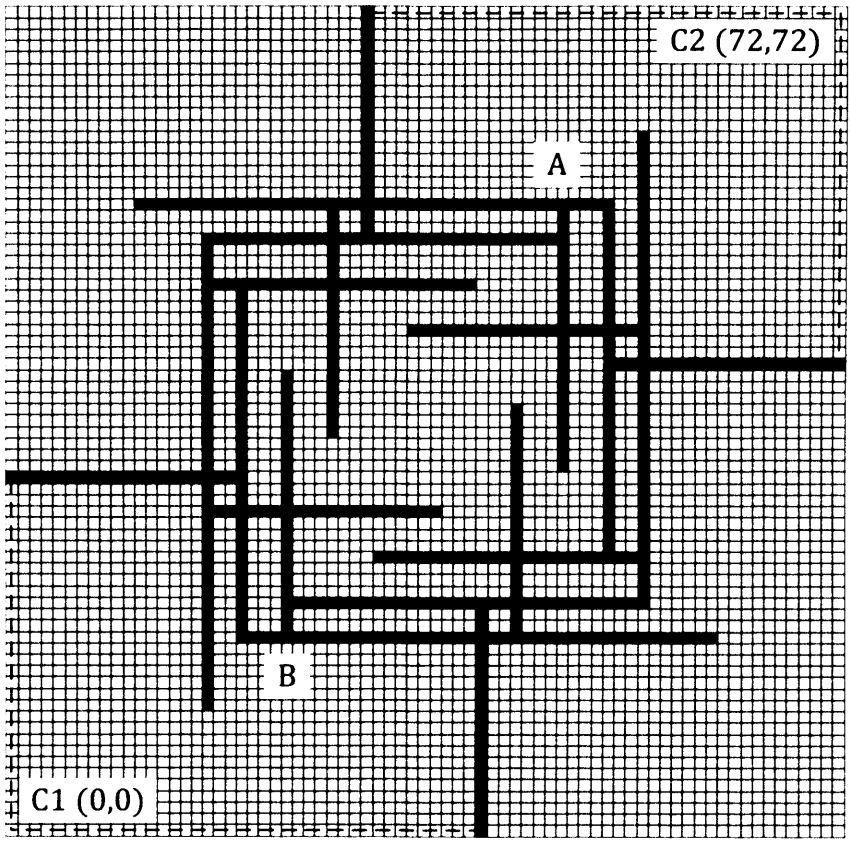


Figure 5.6 dynamic mapping

As shown by Figure 5.4, Figure 5.5 and Figure 5.6, by using different mapping methods, different designs are generated from the same turtle graph. If there are criteria for these designs to be evaluated against, it is very likely that different evaluation results are yield. Comparison results and properties of the three presented mapping methods are given and discussed in more details in Chapter 7.

5.4 Discussions

5.4.1 Restriction for Parameters

As given in Section 5.1.2, there are three forms that parameters can take. Parameters can be divided into two kinds – command parameters and production parameters. Parameters represented by command symbols which include 'F', 'L', 'R' and '$\}$' are command parameters. These parameters control the recursive application of construction commands. The second kind, production parameters, can be considered as the status of the productions which are evaluated against the conditions for further rewrite. These parameters may also be substituted in the

corresponding successors and contribute to the parameters for the new command and production symbols generated by rewriting.

Eventually, production symbols and their parameters will be removed from the command string, leaving only command symbols and command parameters. At this final stage, some circumstances may not be very favourable. For example, a forward command F that takes a parameter of a very large value comparing to those of the other commands can effectively result in a very 'thin' design being generated, which is not always desirable and is very likely to happen if there is no restriction on the parameters.

For variable turtle graphs to be generated, the command strings need to be able to direct the turtle to do both smaller and bigger movements. Allowing the parameters to take just any value will generally result in a very small chance for smaller movements. To balance the chance for smaller and bigger movements, a limitation of $[1,10]$ is applied to random integer generation for parameters. The reason for restricting the parameters' values to integers is because one step ($F(1)$) is defined as the smallest movement unit for the turtle and commands like $F(2.8)$ will only complicate the problem without additional benefit. A parameter's value may contain a fractional part if the parameter is in the form of an algebraic expression which contains division operation. In such case, the value is rounded up to the nearest integer that is bigger than it to avoid 0 value which may result in illegal expression when substituted into successors. The limitation of $[1,10]$ is achieved based on trials of the problem discussed in this thesis. It may vary for different problems.

5.4.2 Termination of Rewrite

Each individual has a proposed number of rewrites before it automatically stops. In certain circumstances, the rewrite has to be terminated before it reaches this number. There are two such circumstances in which, if the command string (compiled generative representation) is rewritten again, it exceeds the maximum allowable length or, the turtle graph it generates exceeds the maximum allowable size.

During rewriting, if a production symbol is replaced by its corresponding successor, the length of the string will increase. Replacing a production symbol that is outside of a block can increase the string by a maximum number of 29 symbols. This effect can be magnified by several times if the production symbol is inside of a block. The length of the command string can increase dramatically in several rewrites and become very computationally expensive, which can significantly influence the performance of an algorithm like a GA which is population and iteration based. Hence, it is necessary to place a restriction on the string length which, in this thesis, is set to 10,000 symbols. In this case, if the length of the string exceeds this limitation at any point during a rewrite, all replacements done in the current rewrite will be revoked and the string from the previous rewrite will be reserved as the final result.

For all the mapping methods described in this chapter, the restriction on command string length is applied with no difference. However, the way to apply restrictions on turtle graph size varies. For static mapping, a turtle graph has to have a span that is no larger than $n - 1$ steps for it to fit into an n by n voxel world. So if it is intended that all turtle graphs generated have to be able to fit into the grid, such restriction needs to be applied. For semi-static mapping, since a turtle graph of any size can be scaled up or down to fit into the grid, no restriction needs to be applied. For dynamic mapping, a restriction on turtle graph size is applied only to prevent the grid from becoming too fine as the turtle graph step and the voxel size of the grid scaling down together. Because the size of a turtle graph generally increases with rewriting, if it exceeds the limitation after a rewrite, that rewrite will be revoked and the string from the previous rewrite will be reserved as the final result.

The maximum allowable length for a compiled generative representation and the maximum allowable size for a turtle graph are both problem-specific. The basic idea is to find the numbers that are large enough to cope with the complexity of the problem while maintaining an acceptable computational cost. The most straight forward way to find these numbers is through tests and trials.

5.4.3 Legality

If the push/pop and block symbols are not used properly, illegal individual will be generated. First of all, these brackets or parentheses have to appear in pairs. If there is a '{', there has to be a corresponding '}' to enclose a block. It is the same for '[' and ']' because there is no point pushing the status of the turtle without ever popping it out. Secondly, the left hand bracket or parenthesis should always appear before the right hand since replicating a block or popping a status that does not exist is meaningless and can often cause a runtime error. Third, pairs of brackets or parentheses can be nested with but cannot be partially overlapped. For example, $\{\cdots[\cdots]\cdots\}(n)$ is legal but $\{\cdots[\cdots\}(n)\cdots]$ or $\{\cdots\{\cdots\}(m)\cdots\}(n)$ is not.

Two different strategies have been developed to ensure all the above three requirements are met during initialization and recombination. The first strategy divides the space which can hold a predefined maximum number of symbols for a successor into several virtual blocks of a same size. During the generation of a successor, it goes through these blocks one by one and randomly decides whether or not a block is enclosed. If yes, it then decides what kind of enclosure it is, that is, either by push/pop symbols or block symbols. Spaces that have not been taken until then will be filled with other symbols including dumb symbols. By using this method, legal individuals are easily ensured during initialization. By limiting crossover to happen between these virtual blocks, new individuals generated are also ensured to be legal. As described, this strategy is easy to apply and facilitates both initialization and recombination for the legal generation of individuals. However, the use of virtual blocks also makes the strategy a bit too rigid as it trades variety for convenience.

The second strategy abandons the use of virtual blocks and favours randomness. The general guideline is to make sure there is a legal spot available for the right bracket/parenthesis before inserting a left one. This spot should be to the right of where the left one is going to be inserted. It should also be within the brackets/parentheses that enclose the left one if the left one is to be enclosed. The inserted brackets/parentheses should be able to enclose at least one symbol. If any of the above conditions is not met, insertion will not be applied. For example, given

the circumstance shown in Figure 5.7, no left one can be inserted into spot 3 because no legal spot is available to its right one. If a left one is to be inserted into spot 6, its right one has to be in spot 8, 9 or 10. This strategy ensures legal initialization of successors without being assisted and restricted by virtual blocks. To implement this strategy, additional work is needed to track the positions of all the brackets and parentheses within an individual in both initialization and recombination. An increase in computational cost is inevitable.

Figure 5.7 insertion of brackets/parentheses

5.5 Summary

In this chapter, the encoding and decoding process of the generative representation is elaborated and discussed. Three different mapping methods are introduced. Two of the mapping methods, namely semi-static mapping and dynamic mapping, are new. The chapter also introduces and discusses the two different strategies to initialize an individual, which had not been done by existing work. These strategies and methods have different impact on the performance of the algorithms. For example, using different mapping methods will make the algorithm to find solutions of different qualities. Comparison studies and experiment results are given in Chapter 7.

Chapter 6 Crossover & Mutation

In the context of genetic algorithms, crossover and mutation are used as operators to generate new individuals from the old ones. Crossover produces new individuals (offspring) by exchanging "parts" among multiple selected individuals (parents). Mutation operates on single individuals and generates variant ones by applying "modifications" to them. These exchanges of parts and modifications all happen on genotypes – encoded individuals. What the "parts" and the "modifications" actually are and how the procedures are implemented largely depend on the representation method. Not only does this chapter presents the possible crossover and mutation methods for this work, but it also discusses the differences among them by examining what they do in the process of generating new individuals and how they influence the search.

6.1 Crossover

The software used to produce experimental results (Chapter 7) for this thesis contains four different crossover methods, including the three given by Hornby (2003a) and a newly developed ones. The three methods given by Hornby (2003a) are referred to as block-based crossover, successor-based crossover, pair-based crossover. The new method is named as mass crossover. There is no discussion about the effectiveness and impact of different crossover methods in Hornby's work. However, it is important to understand how effective the different methods are in terms of facilitating the search and improving the capability of the algorithm. As mentioned earlier, the implementation of crossover is highly representation-specific. The problems with crossover are also found to come from the particular features of the generative representation studied in this thesis. In the following sections, the four crossover methods are examined in connection with the representation.

In order to illustrate the behaviours of the crossover methods in a clearer manner, a colour-and-shape coding is adopted, in which a coloured shape, e.g.  or  , represents a block of commands which does not contain a production symbol. A production rule is formed by combining production symbols with one kind of these

shapes. In this way, it is possible to reveal the behaviours of each of the crossover methods by tracing how materials from parent individuals are replicated and rearranged to generate child individuals.

6.1.1 Block-based Crossover

In this method, a subsequence (block) of symbols in a successor of $child$ is replaced by a subsequence of symbols in a successor of $parent\ 2$. To illustrate how the method works and its properties, the following two individuals are used as parent individuals.

$parent\ 1$:

$P0$: $P1$
$P1$: $P2$ (6.1)
$P2$: $P0$

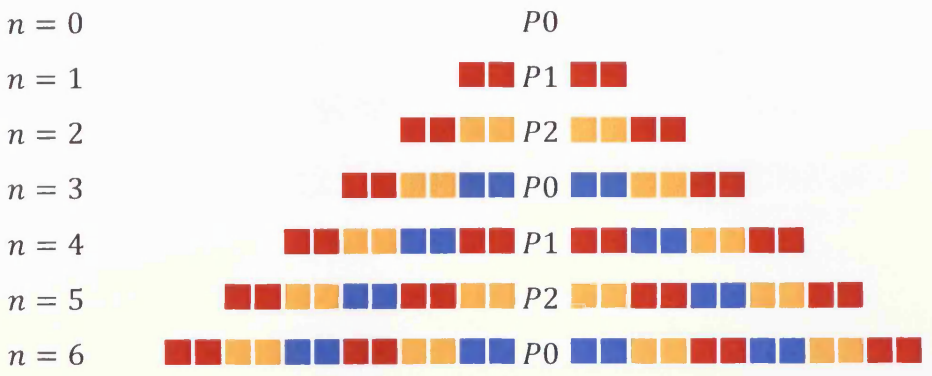
$parent\ 2$:

$P0$: $P2$
$P1$: $P0$ (6.2)
$P2$: $P1$

Similar individuals are used to demonstrate the other methods. These individuals are designed to reveal the properties of different crossover methods in a more observable way by minimizing or hiding the noise of the other factors such as the exact command symbols and parameter values.

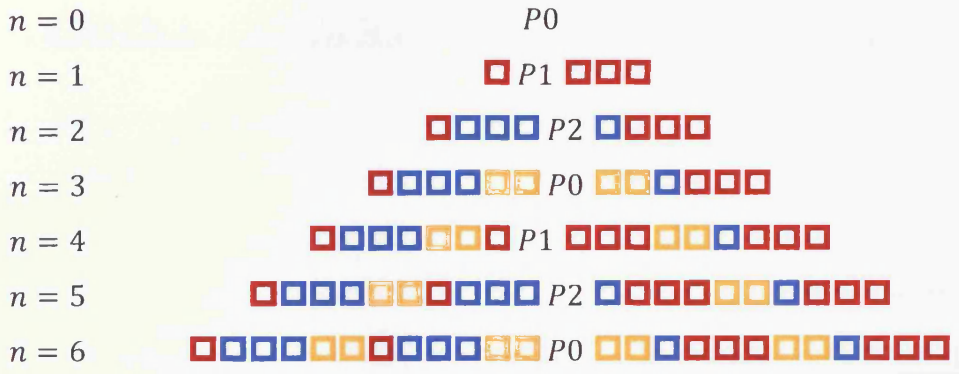
Supposing both $parent\ 1$ and $parent\ 2$ start from symbol $P0$, their rewritings follow what are given by (6.3) and (6.4) (n = times of rewriting). After six times of rewriting for each, they produce the following two sequences given by (6.5) and (6.6) (production symbols are removed). Because of the different arrangement of production symbols, the resultant sequences have different patterns.

Rewriting *parent* 1:



(6.3)

Rewriting *parent* 2:



(6.4)

Resultant sequence of *parent* 1:

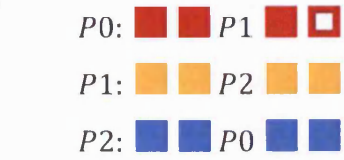
(6.5)

Resultant sequence of *parent* 2:

(6.6)

Depending on the block that is replaced, the crossover method has different effects on the properties of the child individual. In the first instance, the replacement does not involve production symbols, which means no production symbol is swapped out of or swapped into *parent* 1. For example, (6.7) and (6.9) show the child individuals generated by such crossover between $P0$ of *parent* 1 and $P0$ of *parent* 2. The sequences that the child individuals produce by six times of rewriting are given by (6.8) and (6.10) respectively.

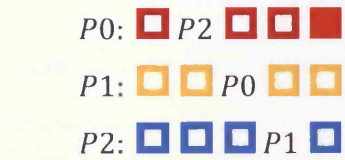
child 1:



(6.7)

(6.8)

child 2:



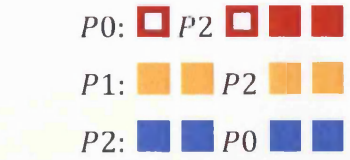
(6.9)

(6.10)

Since *parent* 1 provides most of the materials for *child* 1, it is the dominant parent for *child* 1. Likewise, the dominant parent for *child* 2 is *parent* 2. By comparing (6.8) with (6.5) and (6.10) with (6.6), it can be seen that the resultant sequence produced by *child* 1 inherits the pattern of the sequence generated by *parent* 1, and the same applies to *child* 2 and *parent* 2. The replacement of block does not destroy the patterns of the dominant parents which are passed onto the corresponding children. It should be noted that, the pattern here refers to the arrangement of command blocks in the final command sequence and cannot be considered equivalent to the topological pattern of the resultant design. This is because change in the command sequence is very likely to cause changes in the resultant topology. However, child individuals generated by such crossover are expected to inherit certain topological characteristics from the dominant parents because there is a good chance that the changes in topology are relatively small due to small changes in command sequence. It is also possible that some characteristics of the non-dominant parents are also carried by the child individuals supposing that the part that a child takes from its non-dominant parent represents certain recognisable properties of that parent which are able to stand out among those of the dominant parent.

The situation becomes more complicated when production symbols get involved in the replacement. Examples of child individuals produced by such crossover are given by (6.11) and (6.13). The sequences they produce are given by (6.12) and (6.14).

child 1:



(6.11)

(6.12)

child 2:

(6.13)

(6.14)

Since both of the child individuals suggest the same characteristics of the crossover method, starting from here, the thesis uses *child* 1 as the example for discussions. For this crossover, since *child* 1 still gets most of its materials from *parent* 1, *parent* 1 can still be considered as the dominant parent of *child* 1. As it can be seen by comparing (6.12) with (6.5), the final sequence produced by *child* 1 does not have the same pattern as that produced by *parent* 1. In other words, the crossover modifies the pattern of *parent* 1 which is passed onto the child. However, it should be noted that the modification has limited area of effect. According to (6.11), what the crossover does by replacing P1 of *parent* 1 with P2 of *parent* 2, considering only the production symbols, is to switch $P1$ to $P2$ and reposition it in the successor, which results in a different rewrite of $P0$ and an absence of $P1$ and is the reason why is absent from the resultant sequence of *child* 1. However, the replacement does not influence the rewrite of $P2$, which makes the child still possess a fragment of the pattern of parent 1, which is shown in (6.15).

(6.15)

The crossover also brings in something that belongs to $parent\ 2$. The positioning of $P2$ in the successor of the first production rule of $parent\ 2$ creates the following fragment of pattern extracted from (6.6).

(6.16)

In the sequence (6.12) produced by $child\ 1$ (6.11), the pattern given by (6.16) is partially preserved with some elements of $parent\ 1$ replacing those of $parent\ 2$, which results in a modified pattern shown in (7.8). This also makes it possible that certain topological properties of $parent\ 2$, although partially and probably modified, can be passed to $child\ 1$.

(6.17)

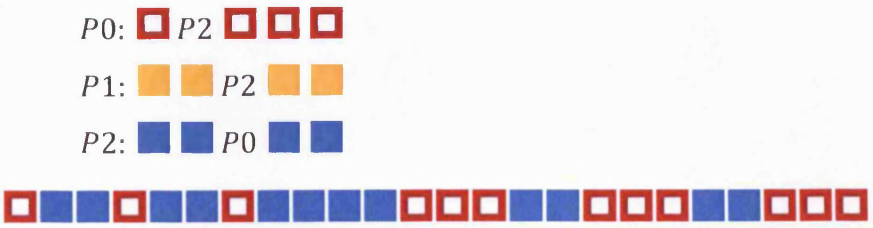
In general, block-based crossover that involves production symbols can introduce bigger changes to the pattern of the command sequence. The recombination of successor blocks appears to result in a recombination of partial patterns of the command sequence. Again, $child$ produced in this way can be expected to inherit topological characteristics from both of its parents.

6.1.2 Successor-based Crossover

This method replaces an entire successor of one parent with an entire successor of the other parent to generate a child individual. An example $child\ 1$ and the command sequence it produces are given in (6.18) and (6.19).

Unlike sequence (6.12), the fragmental pattern from $parent\ 2$ given in (6.16) is completely preserved in sequence (6.19) in addition to that from $parent\ 1$ given in (6.15). Comparing to block-based crossover, successor-base crossover is more likely to generate child individuals who bear recognisable topological properties from both of the parents.

child 1:



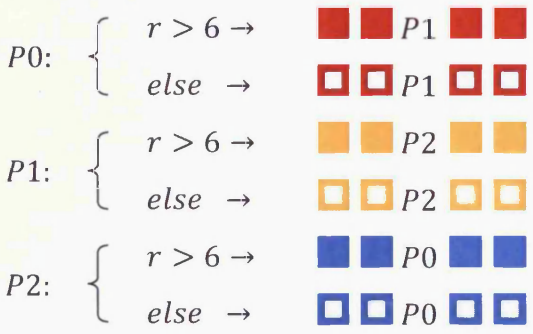
(6.18)

(6.19)

6.1.3 Pair-based Crossover

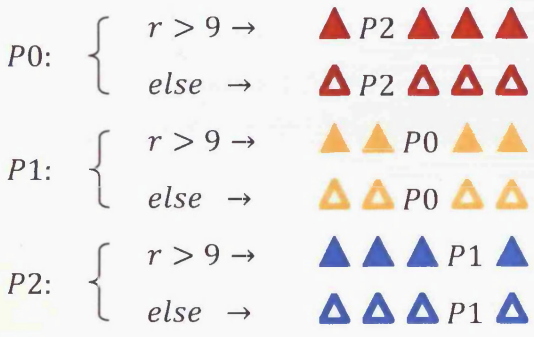
A condition-successor pair of *child* is replaced by a condition-successor pair of *parent* 2 in this method. To reveal the properties of pair-based crossover, parent individuals used in previous example need to be modified to include conditions. New parent individuals are given by (6.20) and (6.21) in which r is the time of rewrite.

parent 1:



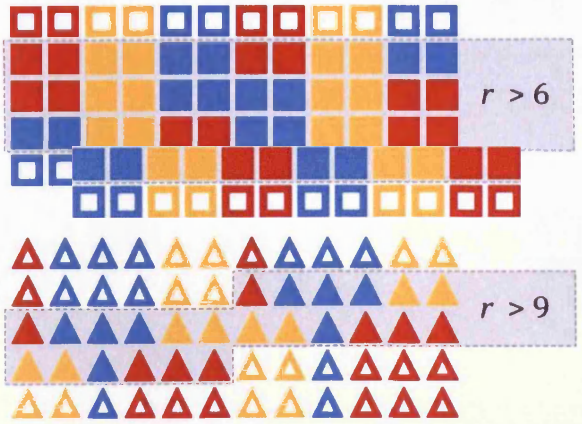
(6.20)

parent 2:



(6.21)

If both parent individuals are given fifteen times of rewrite, they generate the following two sequences as shown in (6.22) and (6.23) respectively.



(6.22)

(6.23)

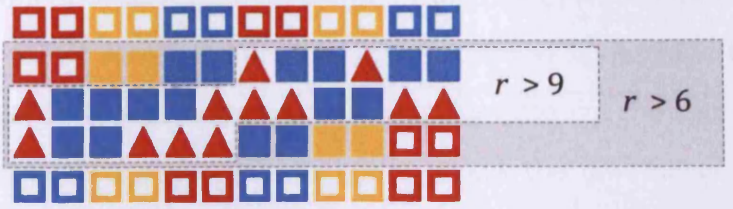
The use of conditions introduces changes of rewriting on changes of circumstances which, in the example above, is the time of rewriting. When replacing a production symbol, different successors are used depending on whether or not the time of rewrite exceeds the threshold defined by the condition. Take $parent\ 1$ for example, after six times of rewriting, a different set of successors that consist of solid symbols (e.g. ■ & ▲) takes the place of the successors that consist of hollow symbols (e.g. □ & △) and are used to replace production symbols for further rewrites. The part of the sequence that is generated after the shift is shaded in (6.22) and (6.23).

To illustrate the influence of crossover that involves conditions, the following $child\ 1$ (6.24) is used. It is generated by replacing the first condition-successor pair of $parent\ 1$ with the first condition-successor pair of $parent\ 2$. As is shown by (6.24), for this $child\ 1$, $parent\ 1$ is still the dominant parent and $parent\ 2$ is still the non-dominant one.

$child\ 1$:

$$
\begin{aligned}
P0:&\ \begin{cases} r > 9 \rightarrow & \blacktriangle\, P2\, \blacktriangle\, \blacktriangle\, \blacktriangle \\ else \rightarrow & \square\, \square\, P1\, \square\, \square \end{cases} \\
P1:&\ \begin{cases} r > 6 \rightarrow & \blacksquare\, \blacksquare\, P2\, \blacksquare\, \blacksquare \\ else \rightarrow & \square\, \square\, P2\, \square\, \square \end{cases} \\
P2:&\ \begin{cases} r > 6 \rightarrow & \blacksquare\, \blacksquare\, P0\, \blacksquare\, \blacksquare \\ else \rightarrow & \square\, \square\, P0\, \square\, \square \end{cases}
\end{aligned}
\qquad (6.24)
$$

Like its parents, if given fifteen times of rewriting starting from $P0$, it generates the sequence given by (6.25). Partial sequences that are generated after condition shifts are shaded or highlighted.



(6.25)

Similar to successor-based crossover, completely replacing a condition-successor pair with another one will most likely change the outcome of the rewriting. The difference is, with the successor being swapped along with its condition, rewriting that uses this successor should occur under the same condition as it does in the individual the successor comes from. For example, the solid triangle ▲ only appears under the condition of $r > 9$ in the sequence produced by $parent\ 2$ (6.23), which is preserved in the sequence produced by $child\ 1$ (6.25). Using the successor under the same condition can maximize the recurrence of its functionality in the new individual. Considering that, in the real representation, conditions are used to judge the values of parameters that are passed to the successors, pair-base crossover makes the successor that replaces the previous one still take parameters of the same range of value in the new individual ($child\ 1$) as it does in its original individual ($parent\ 2$). This can result in that the command sequence produced by $child\ 1$ contains fragments that can also be found in $parent\ 2$. There are chances that the inheritance of topological properties from $parent\ 2$ to $child\ 1$ can happen through these fragments. When parameters are taken into account, pair-based crossover is more effective than block-based and successor-based methods to introduce topological properties of the non-dominant parent into the corresponding child individual.

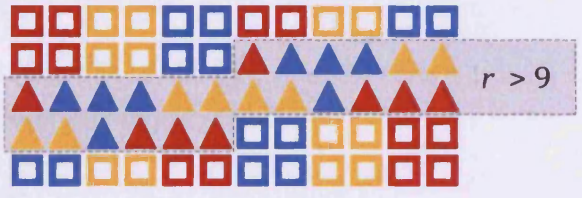
6.1.4 Mass Crossover

Considering an individual which has fifteen production rules, each of which contains three condition-successor pairs (proposed representation scheme, Section 5.1.4), despite the differences between block-based, successor-based and pair-based

crossover methods, they all have something in common, that is, for each child individual generated, one of its parents appears to be dominant and the other one appears to be non-dominant. In this way, the genetic information (building blocks) of a child individual is mostly inherited from one of its two parents and only a little from the other. As a new method, mass crossover is designed to naturally reduce this inequality. It does not seek to completely even out the contribution of building blocks between the two parents but it makes the differences much smaller. For example, two child individuals generated by a mass crossover and the sequences they produce are given below.

child 1:

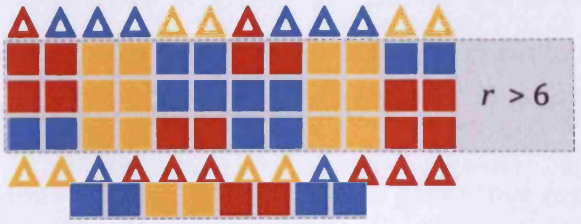
$$\begin{aligned} P0{:}&\begin{cases} r>9 \rightarrow & \blacktriangle P2 \blacktriangle\blacktriangle\blacktriangle \\ else \rightarrow & \square\square P1 \square\square \end{cases} \\ P1{:}&\begin{cases} r>9 \rightarrow & \blacktriangle\blacktriangle P0 \blacktriangle\blacktriangle \\ else \rightarrow & \square\square P2 \square\square \end{cases} \\ P2{:}&\begin{cases} r>9 \rightarrow & \blacktriangle\blacktriangle\blacktriangle P1 \blacktriangle \\ else \rightarrow & \square\square P0 \square\square \end{cases} \end{aligned} \tag{6.26}$$



(6.27)

child 2

$$\begin{aligned} P0{:}&\begin{cases} r>6 \rightarrow & \blacksquare\blacksquare P1 \blacksquare\blacksquare \\ else \rightarrow & \triangle P2 \triangle\triangle\triangle \end{cases} \\ P1{:}&\begin{cases} r>6 \rightarrow & \blacksquare\blacksquare P2 \blacksquare\blacksquare \\ else \rightarrow & \triangle\triangle P0 \triangle\triangle \end{cases} \\ P2{:}&\begin{cases} r>6 \rightarrow & \blacksquare\blacksquare P0 \blacksquare\blacksquare \\ else \rightarrow & \triangle\triangle\triangle P1 \triangle \end{cases} \end{aligned} \tag{6.28}$$



(6.29)

A mass crossover exchanges a number of condition-successor pairs between two parents to generate child individuals. For the example given above, half of the condition-successor pairs are exchanged. Mass crossover is not just another way for recombination. The example above shows the method's tendency to preserve the patterns of both parents in either of the two children. Take sequence (6.27), the characteristics of both $parent\ 1$ and $parent\ 2$ are more recognizable than that in any of the previous sequences generated by the other crossover methods. Mass crossover introduces the possibility of generating child individuals that are more in-between to their parents by evening out the contribution between the two parents. This does not mean child individuals that are significantly closer to one parent than the other are not desirable. What mass crossover provides is a different possibility in the outcome of a crossover implementation. By combining the properties of the different crossover methods, more extensive search can be achieved, which is why using multiple crossover methods in the single GA system is often found to be superior than using just one method.

6.2 Mutation

Unlike crossover, mutation works on single individuals. In common practice, a small number of new individuals that are generated by crossover are selected and mutated for each generation. In this thesis, mutation is used as an operator to generate new individuals from the previous generation, thus a new generation consists of two kinds of individuals, those generated directly by crossover and those generated directly by mutation. This setup is for research consideration. If a new individual is generated only by mutating an individual from the previous generation, the influence of crossover can be completely ruled out from the process of generating this new individual. Whatever changes that are introduced to this new individual are introduced by that particular implementation of mutation only.

Knowing which individuals in a new population are generated by mutation only helps to track and monitor the influence of mutation without being interfered by the influence of crossover.

In this section, the influence of mutation on generative representation is illustrated using a randomly generated individual which matches the format given in Section 5.1.4. The starting symbol of the individual is P_{11} which takes two parameters $n_0 = 10$ and $n_1 = 6$. After ten times of rewriting, the individual generates the topology given by Figure 6.1. A new production table is also included in Figure 6.1 and in the rest of the examples of this section to provide additional information that is not visible through the turtle graphs and the resultant shapes. Because of the involvement of conditions, not all successors are guaranteed to be used. The table is used to display which successors of which productions are involved in the rewriting process. If a successor is used, its corresponding square in the table is filled; otherwise, blank. For example, according to the production table in Figure 6.1, the successor of pair 0 (condition-successor pair 0) of production 1 is used in rewriting; whereas that of pair 1 of production 5 is not. The mapping mechanism used here is dynamic mapping (Section 5.3.3).

Mutation modifies the genetic information of an individual. In other words, it works to the genotype of an individual and applies changes to an encoded design. Applying mutation to different parts of an individual leads to different result. The genetic information of an individual that can be modified by mutation includes:

1. the starting symbol which includes the production symbol and the parameters it takes;
2. the number of rewriting cycles/times;
3. the condition which includes the condition parameters, n_0 or n_1, and the value that the parameter is compared to; and
4. the successor symbol which also includes the parameter(s) it takes.

In this thesis, each application of mutation can only use one of the above possibilities. In addition, the modification is limited to one charge. For example, if one parameter of a successor symbol is changed, the other parameter and the

symbol itself will not be changed. There is an exemption when changing a NULL symbol into one other symbol because the whole symbol needs to be initialized including the symbol type and the parameter(s) it takes. The reason for this limitation is to increase the specialty of a single mutation. It also helps to understand mutation operators with generative representations. Knowing that different mutations have different functionalities and using them based on their specialities can better facilitate the search. The rest of this section examines mutations of the four different types listed above.

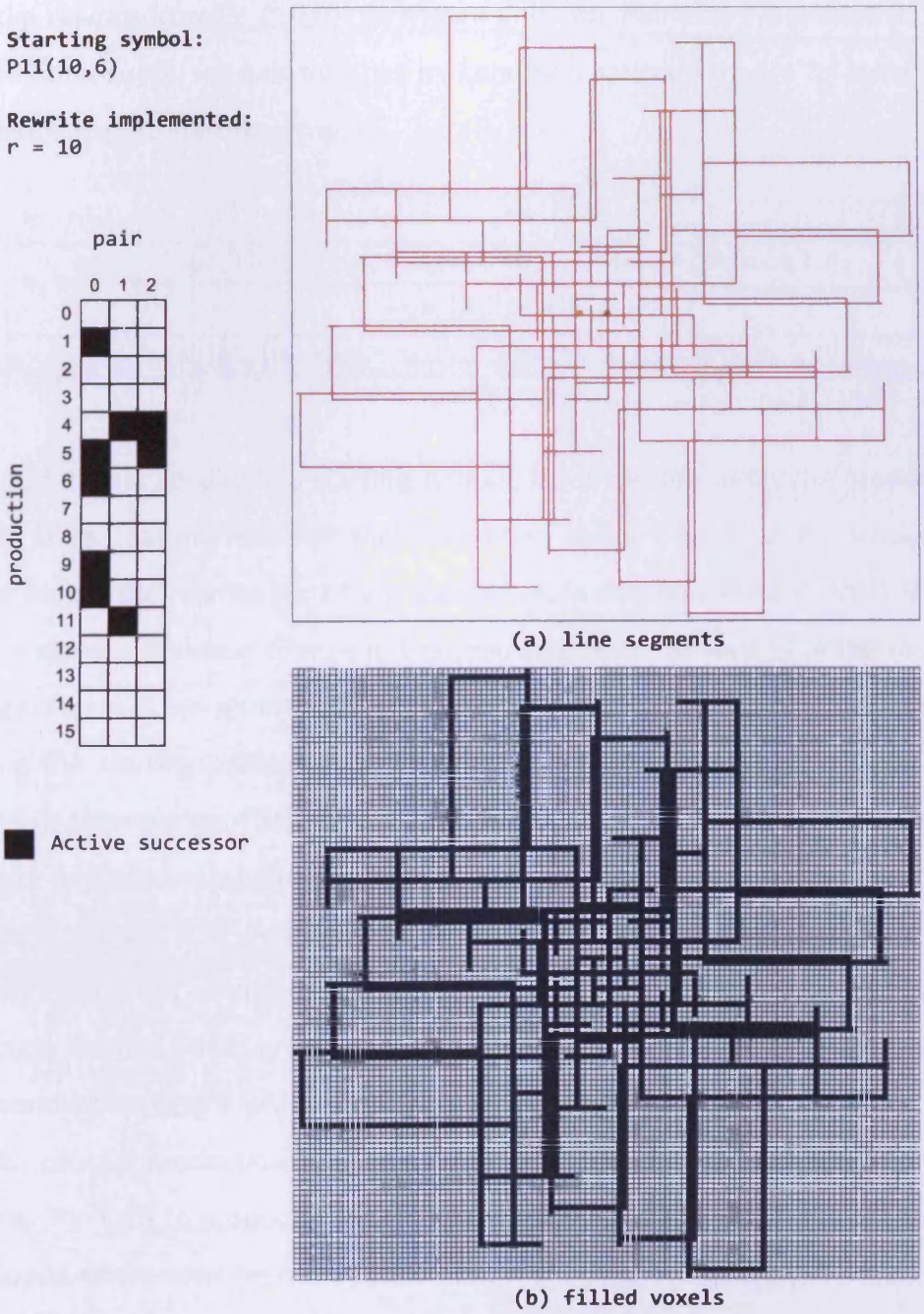


Figure 6.1 original individual before mutation

6.2.1 Starting Symbol

A starting symbol consists of a production symbol and two parameter values that the production takes. They together define from which symbol the rewrite starts and how the rewrite is done for the first time. The example individual (Figure 6.1) starts the rewrite from $P_{11}(10,6)$. As shown in (6.30), Pair 1 of Production 11 is the first condition-successor pair that has its condition satisfied, hence its successor is used to replace the starting symbol.

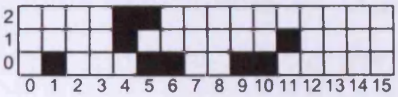
$$P_{11}\begin{array}{|c|l|c|l|}\hline 0 & n_1 > 10 & \rightarrow & \{\{F(n_0)R(1)\}(n_0)R(1)L(1)\{L(1)R(1)\}(n_1)\}(n_1)L(2)R(2)R(2) \\ \hline 1 & n_0 > 5 & \rightarrow & L(1)L(3)\{P_4(n_1-n_0, n_0-n_1)L(1)L(3)\}(n_1)P_9(n_0+1, n_0 \\ & & & \quad -n_1)R(2)F(4)F(n_1) \\ \hline 2 & else & \rightarrow & [\{P_{11}(n_1+5, 8-n_0)P_2(8+n_0, 9+n_0)\}(n_0)]P_{14}(n_0-10, n_0 \\ & & & \quad -n_1)F(8)R(2)P_6(n_1-n_0, n_1-n_0)L(1)R(2) \\ \hline\end{array} \quad (6.30)$$

When mutation is applied to a starting symbol, it can either modify the production's number or the parameters that the production takes. Changing the production's number makes the rewrite start from a completely different symbol. This will most certainly cause a dramatic change in the rewriting result as well as in the resultant topology. Figure 6.2 is an example of such mutation. As is shown in Figure 6.2, by changing the starting production from P_{11} to P_6, a different set of successors get involved in the rewrite. These successors are referred to as the active successors which are indicated using the production table in Figure 6.2. Changing the starting production symbol is a very aggressive way of mutation. The change in active successors is a result of the change in "call map". If the current sequence contains a production symbol, that production will be called by the next rewrite and its corresponding successor will be returned to take its place. The returned successor may also contain productions symbols which are to be called for the next circle of rewriting. The calls to productions continue throughout the rewrite. If a call changes at one point of the rewrite, it may cause the rest of the rewrite to call a completely different sequence of productions. Mutating the starting production symbol is likely to make this change at the very beginning of the rewrite, which is more than enough to generate a completely different design from the original one.

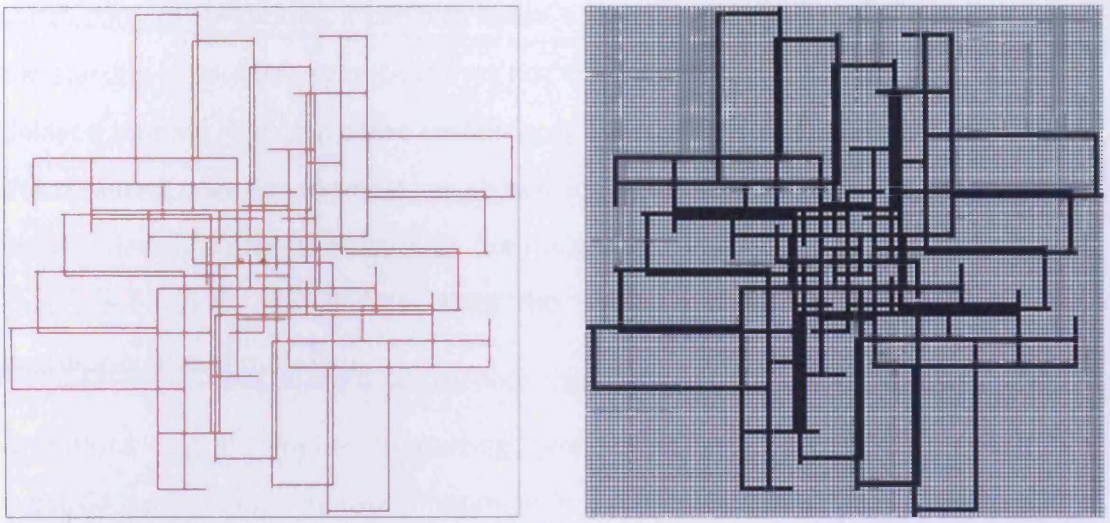
Starting symbol:
P11(10,6):n0>6 -> L(1)L(3){P4(n1-n0,n0-n1)L(1)L(3)}(n1)P9(n0+1,n0-n1)R(2)F(4)F(n1)

Active successors:

pair



Times of rewrite: r = 10

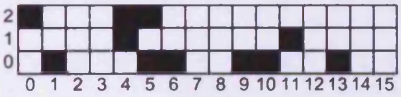


(a) Original

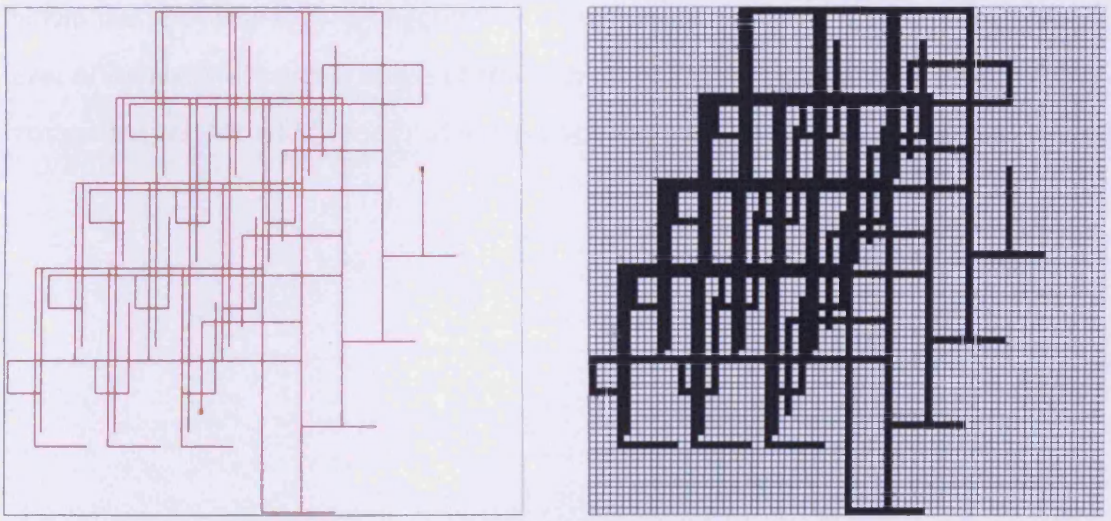
Starting symbol:
P6(10,6):n0>1 -> [L(1)R(1)L(2)P4(10-n1,n1+n2)R(1)[]R(3)]

Active successors:

pair



Times of rewrite: r = 7



(b) Mutated

Figure 6.2 mutation: starting production symbol

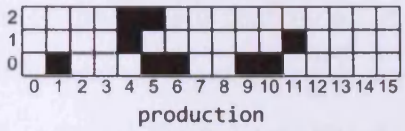
Mutation can also be applied to the parameter values that the starting production symbol takes. Figure 6.3 presents an example of such mutation. In this example, one of the parameter values of the starting symbol, n_1, is changed from $n_1 = 6$ to $n_1 = 7$. Since the parameter value determines the choice of successors based on the satisfaction of conditions, it can also cause a change in the call map. Unlike changing the starting production symbol, it may not change the call at the start but in a more delayed manner. This can cause certain properties of the original design to persist in the mutated one. For example, as shown in Figure 6.3, the axial symmetry of the original design is preserved through the mutation. However, as long as the influence shows eventually, though later than the previous mutation, it is also a rather aggressive kind of mutation.

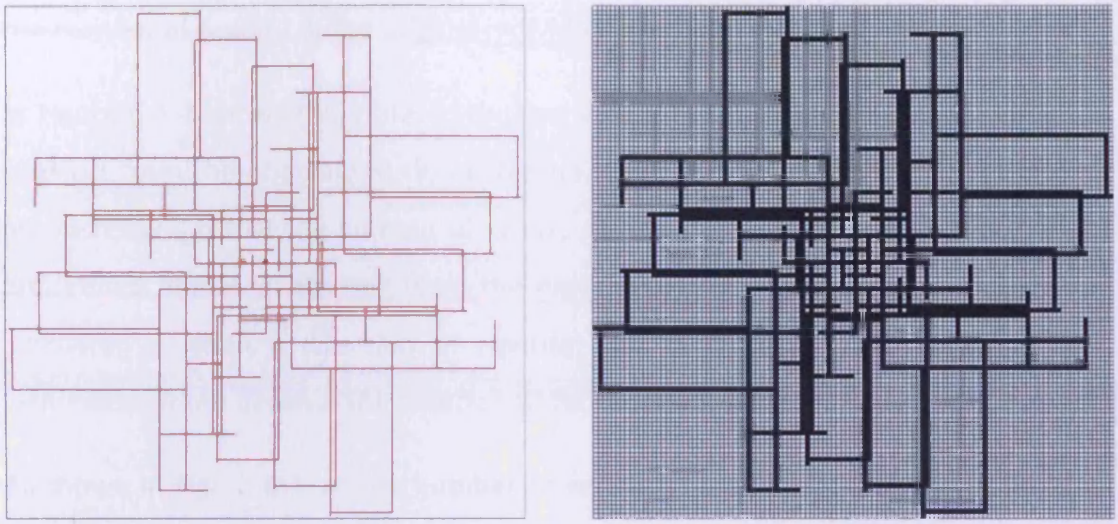
Mutations that are applied to starting symbols are very aggressive. In nature and most GA applications, mutation happens in a much moderate way. A GA mutation operator is generally used to introduce subtle change to an individual by slightly modifying its chromosome. For most representation schemes, a slight change in an individual's genotype means a slight change in its phenotype. However, for generative representations, this is not always the case. Changing the starting symbol is one of the occasions when mutation introduces significant changes to an individual. Allowing such an aggressive mutation can help to maintain a required level of diversity at certain stage of the search. It is also a complementation to the crossover operator which does not involve starting symbols.

Starting symbol:
P11(10,6):n0>6 -> L(1)L(3){P4(n1-n0,n0-n1)L(1)L(3)}(n1)P9(n0+1,n0-n1)R(2)F(4)F(n1)

Active successors:

Times of rewrite: r = 10



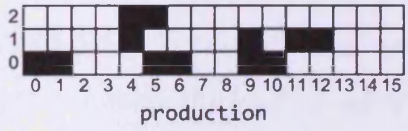


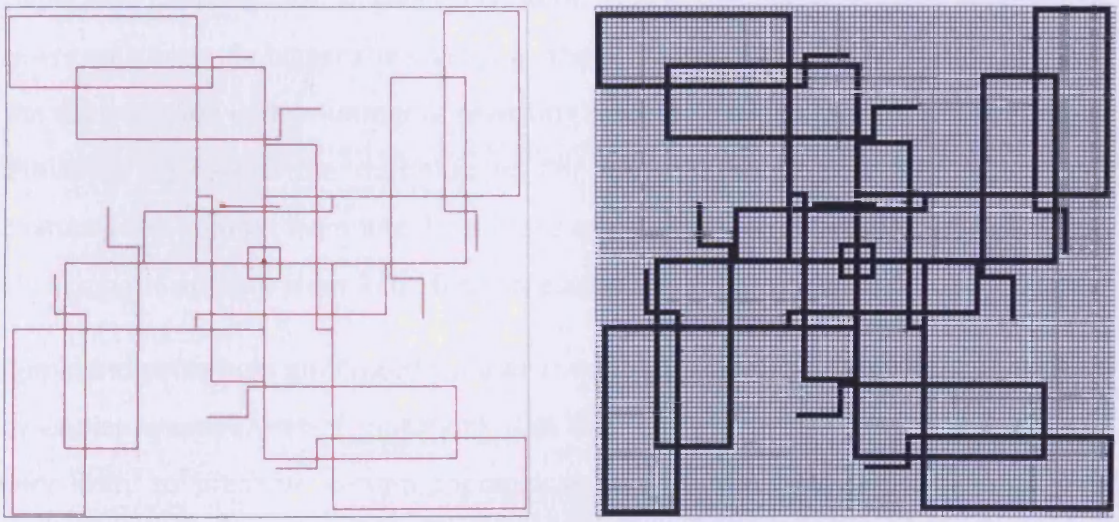
(a) Original

Starting symbol:
P11(10,7):n0>6 -> L(1)L(3){P4(n1-n0,n0-n1)L(1)L(3)}(n1)P9(n0+1,n0-n1)R(2)F(4)F(n1)

Active successors:

Times of rewrite: r = 10





(b) Mutated

Figure 6.3 mutation: starting symbol parameter

6.2.2 Number of Rewriting Times

For each time of rewriting, productions of the current sequence are called and their corresponding successors are returned to take their place. In this way, the sequence is expanded throughout the rewrite. A series of examples of mutations that change the number of rewrite of the original individual are given in Figure 6.4.

In Figure 6.4 it shows the mutated designs generated by decreasing the number of rewrites from the original ten times to nine, eight and seven times. The reason for not increasing is that the number of symbols of the resultant sequence exceeds the predefined limitation starting from the eleventh rewrite. Hence, even though the individual is given a direction of rewrite that is more than ten times, it still terminates at ten because the criterion of termination is met.

As shown in Figure 6.4, as the number of rewrite increases from seven to ten, the complexity of the resultant designs also increases, which corresponds to the increase in the number of symbols in the final command sequence and the number of production calls that are involved in the rewrite. Mutations that change the number of rewrites should also be considered as very aggressive. Depending on the original number of rewriting times and the change, the level of aggressiveness varies. Generally speaking, for a given L-system, given the same original number of rewriting times, the bigger the change is, the more aggressive the mutation is; given the same change in the number of rewriting times, the bigger the original number is, the more aggressive the mutation is. For example, as is shown by Figure 6.4, changing the number from 8 to 10 is more aggressive than changing it from 9 to 10; changing the number from 9 to 10 is more aggressive than changing it from 7 to 8.

Command sequences generated by later rewrites are always built on that generated by earlier rewrites. Hence mutations that only change the number of rewrites are very likely to preserve certain topological properties of the original design. For example, the axial symmetry of the original design can still be recognized in all its mutated designs shown in Figure 6.4.

Despite the fact that mutating the number of rewrites can preserve certain topological properties of the original design, it is still a very aggressive way of

mutation. Just as that with the starting symbol, allowing mutation to change the number of rewriting times maintains an effective way to introduce diversity into the population. As is shown by Figure 6.4, given different numbers of rewriting times, an L-system produces different designs. Since the ideal number of rewriting times for an L-system, that is, the number which makes the L-system produce the best possible design is not known beforehand, each individual is given a random integer as a proposed number of rewriting. Hence, having the option of modifying the number of rewriting times improves the chance to locate better potential solutions. Unlike starting symbol mutation, the aggressiveness of mutating the number of rewriting times can be controlled by restricting the change to a small integer such as 1, although it does not stop it from being a highly aggressive way of mutation.

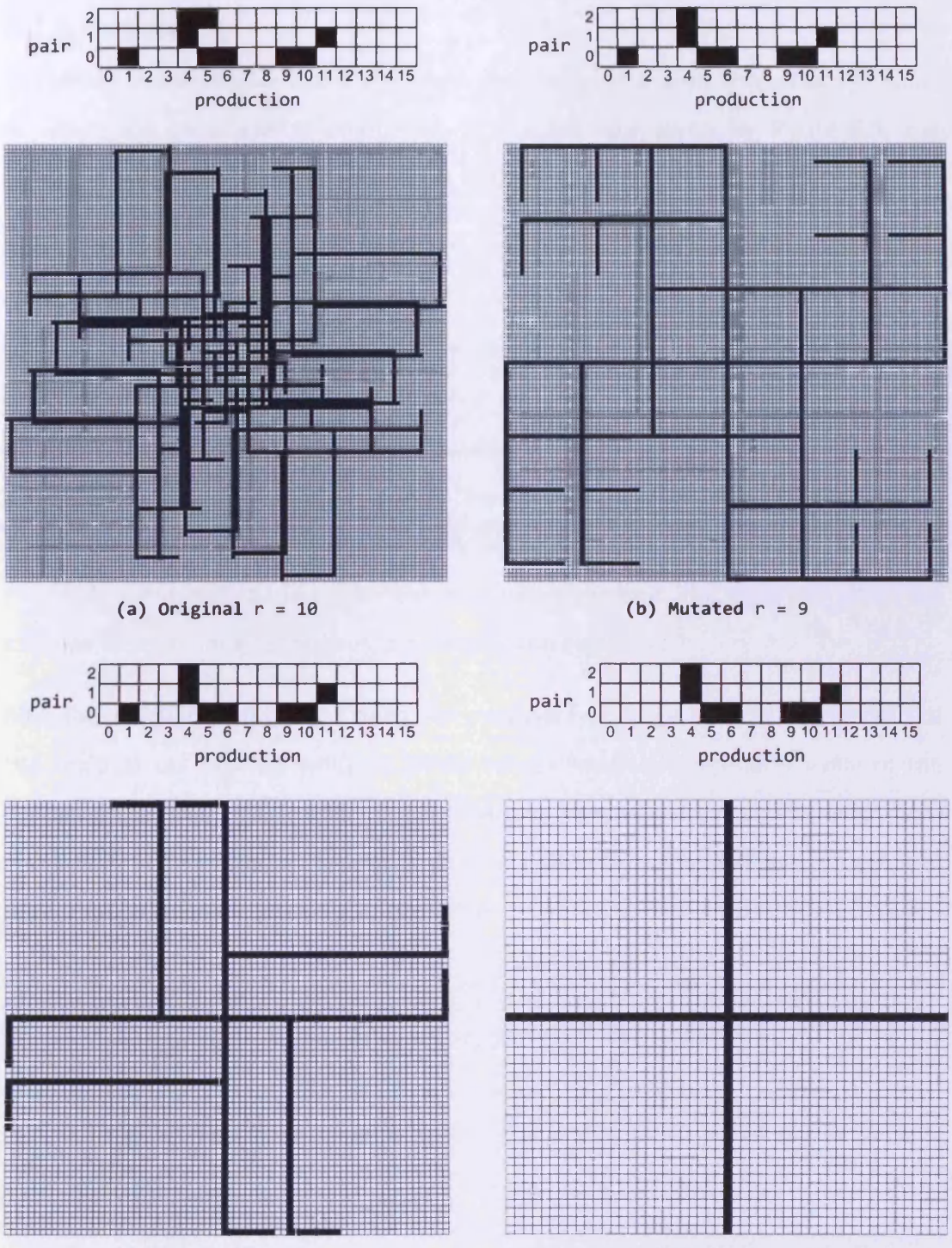


Figure 6.4 mutation: rewrite times

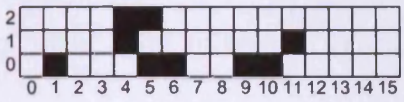
6.2.3 Condition

Mutations that are applied to a condition either change the parameter or the value to which the parameter is compared. In the example given by Figure 6.5, the condition of Pair 0 of Production 4 (P_4) is changed from $n_0 > 9$ to $n_0 > 0$.

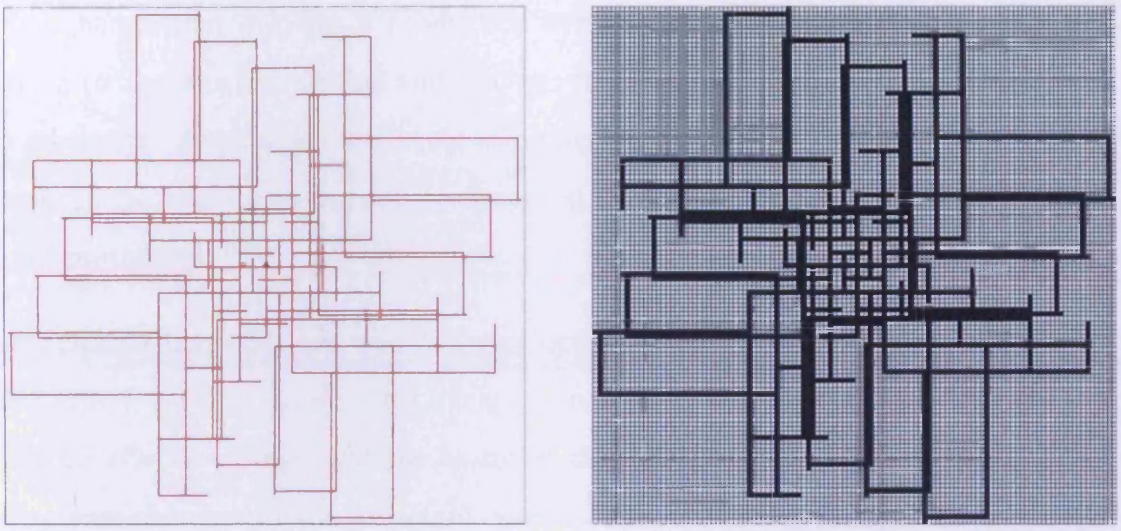
Conditions control which successor is returned when a production is called during rewrite. Mutating a condition can change the return value (in the form of a successor) of a production call and hence change the call map thereafter. For example, in the original individual shown in Figure 6.5, when P_4 is called, either the successor of its Pair 1 or Pair 2 will be returned based on the parameter value it takes. In the mutated individual, due to the change in the condition of Pair 0 of P_4, the successor of Pair 0 becomes the only active successor of the production. Since the three successors contain different production symbols, the mutation alters the call map from the original after P_4 is called for the first time.

Because this kind of mutation does not necessarily change the call map from the beginning of the rewrite, which is similar to mutating the parameter value of the staring production, it is able to keep certain topological properties (e.g. axial symmetry) of the original design. Since the call map is changed nevertheless, it should still be considered as very aggressive.

```
P4: n0>9  -> {R(2){L(2)R(2)}(n0)L(2)F(10)R(2)F(n1)}(n0)P5(n1-n0,n0+7)
    n1>10 -> {P11(n0-n1,3+n0)[[R(3)F(8)F(9)P5(9,n1+n0)[F(8)]]]}(n0)
    else  -> P6(2+n0,n1-2)[{R(3)}(n0)]R(1)R(2){R(2)}(8)
```



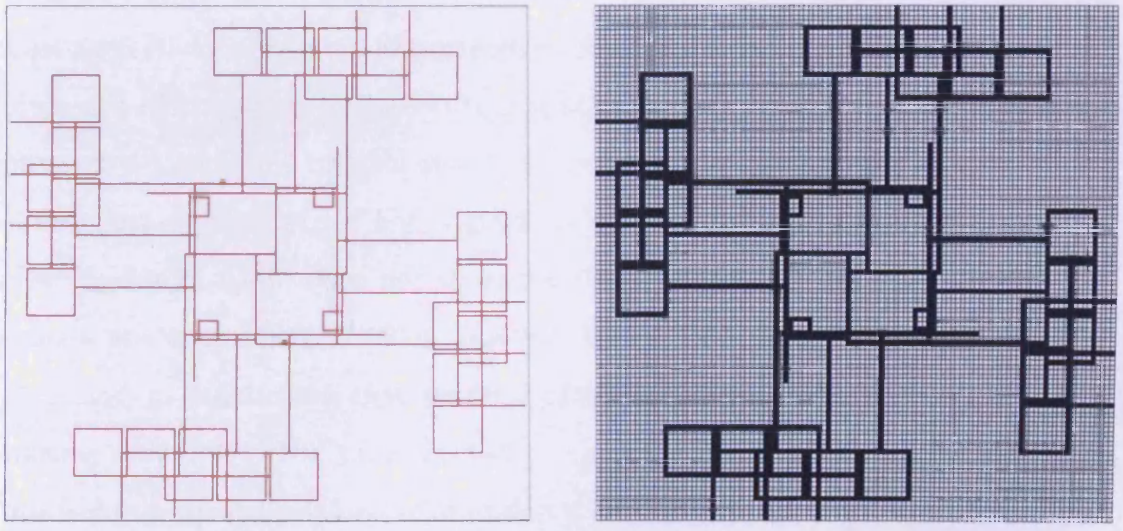
Times of rewrite: r = 10



(a) Original

```
P4: n0>0  -> {R(2){L(2)R(2)}(n0)L(2)F(10)R(2)F(n1)}(n0)P5(n1-n0,n0+7)
    n1>10 -> {P11(n0-n1,3+n0)[[R(3)F(8)F(9)P5(9,n1+n0)[F(8)]]]}(n0)
    else  -> P6(2+n0,n1-2)[{R(3)}(n0)]R(1)R(2){R(2)}(8)
```

Times of rewrite: r = 7



(b) Mutated

Figure 6.5 mutation: condition

6.2.4 Successor Symbol

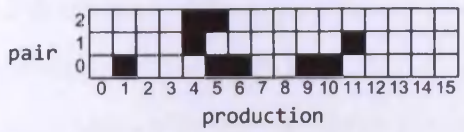
To mutate a successor symbol, one can either change the type of the symbol or a parameter that the symbol takes. The influence of this type of mutation varies according to circumstances.

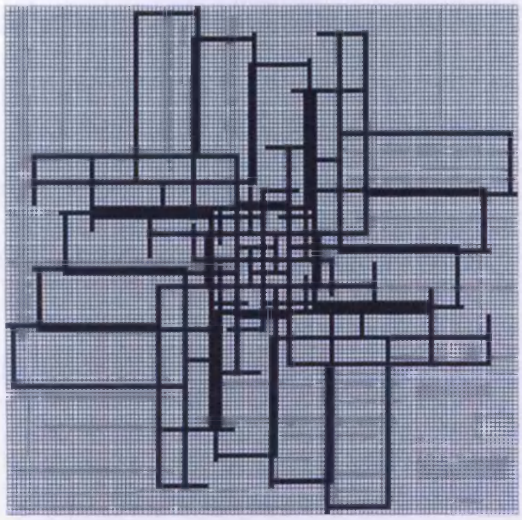
Any change that involves a production symbol, which includes changes that are made to a production symbol and changes that make a non-production symbol into a production symbol, is most likely to be aggressive because it will change the call map of the rewrite as previously discussed. Figure 6.6(b) presents an example of such mutation.

Any change that does not involve a production symbol but non-production symbols like L, R, {, }, [or], is also most likely to be aggressive. Although the call map will not be affected, these symbols as turtle commands are strong enough to cause dramatic changes in the resultant graphs. For example, changes made to the commands L and R can effectively cause a turtle graph to develop in different directions. An example of such mutation is given by Figure 6.6(c).

The above two categories exclude two last circumstances which are mutations that change the parameter of symbol F and mutations that change a NULL symbol into a symbol F. Compared to all the other mutations, these two kinds of mutation are the least aggressive ones. In the example given by Figure 6.6(d), a symbol $F(2)$ takes the place of a NULL symbol between the first $L(1)$ and the first $L(3)$. By comparing the mutated design to the original one it can be seen that the changes introduced are almost just translations of line segments. For illustration purposes, the example given by Figure 6.6(d) does not show the finest change the mutation can do. The change are even finer if $F(1)$ is inserted instead of $F(2)$ or make the change in successors of productions that are later called in the rewrite instead of that of the starting symbol P_{11}. The same applies to changing the parameter of an F symbol. Due to this property, this kind of mutation is ideal for fine turnings.

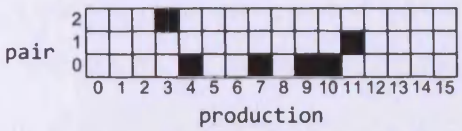
P11(10,6):n0>6 -> L(1)L(3){P4(n1-n0,n0-n1)
L(1)L(3)}(n1)P9(n0+1,n0-n1)R(2)F(4)F(n1)

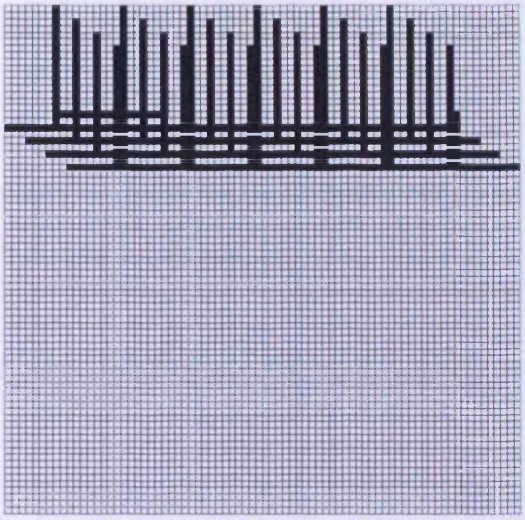




(a) Original

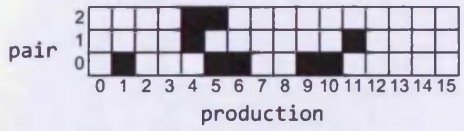
P11(10,6):n0>6 -> L(1)L(3){P7(n1-n0,n0-n1)
L(1)L(3)}(n1)P9(n0+1,n0-n1)R(2)F(4)F(n1)

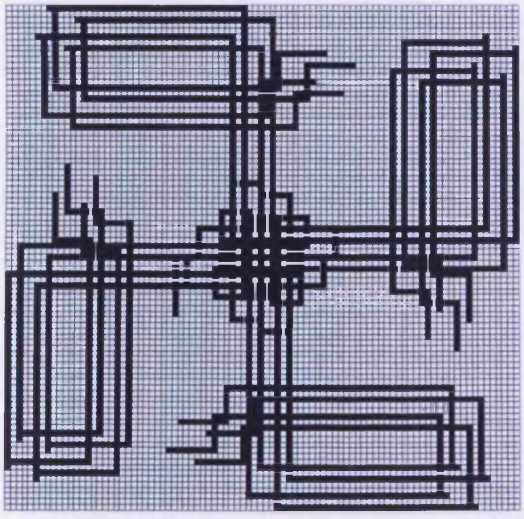




(b) Mutated P4 -> P7

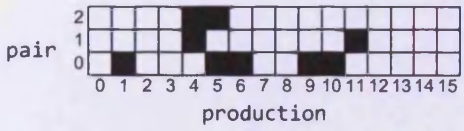
P11(10,6):n0>6 -> L(1)L(3){P4(n1-n0,n0-n1)
L(1)L(3)}(n1)P9(n0+1,n0-n1)R(1)F(4)F(n1)

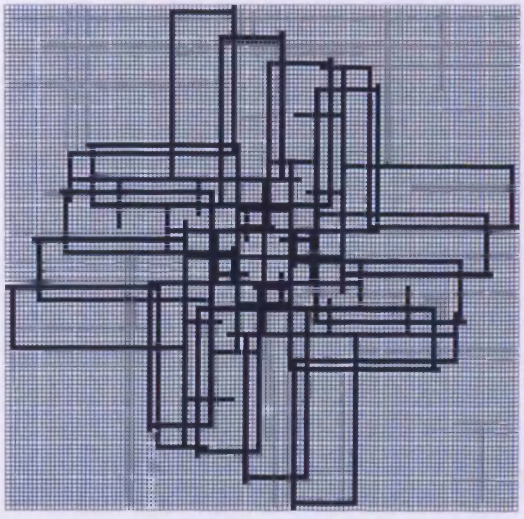




(c) Mutated R(2) -> R(1)

P11(10,6):n0>6 -> L(1)F(2)L(3){P4(n1-n0,n0-
n1)L(1)L(3)}(n1)P9(n0+1,n0-n1)R(2)F(4)F(n1)





(d) Mutated insert F(2)

Figure 6.6 mutation: successor symbol

6.3 Summary

In this chapter, different crossover and mutation methods that can be used with the generative representation studied in this thesis are introduced and discussed. From what is shown by the examples presented in this chapter, it can be concluded that these different methods facilitate the search in different ways. A proper arrangement of these methods can help to form a more efficient search because some methods are more suitable than the others at certain stage of the search. Hence, it should be considered a good strategy to maintain the control over the probability for each of the methods to happen during the search.

Chapter 7 Tests & Results

In this chapter, experiments carried out on topological reasoning problems and their results are presented and discussed. Firstly, results from previous experiments on "shape-matching" problems (Zhang, 2008) are presented. The chapter then focuses on a more real-world testing problem, that is, to find a shape that best matches certain topological properties. The test problem is described in detail and the theory for calculation is introduced. Experiments are then carried out to investigate the adjustable genotype attributes. Different genotype-phenotype mapping methods are also tested. Experimental results are presented and discussed.

7.1 Previous Results of Shape-matching Problems

A series of shape-matching problems have been used to study generative representations (Zhang, 2008). Unlike real-world problems where the solutions are unknown, a shape-matching problem has a predefined target shape which can be considered as a known solution. However, no knowledge of the target shape is given to the algorithm. The algorithm only knows how well an individual matches the target shape, which is measured by the percentage of voxels that the individual gets correct. Hence, a shape-matching problem can be considered as a shape optimisation problem where it is known that a single optimum solution exists.

Target shapes used include a diagonal shape (Figure 7.1(a)), a cross shape (Figure 7.2 (a)), a circular shape (Figure 7.3(a)) and an I shape (Figure 7.4(a)) which has been presented in Zhang et al (2008). Experiments have been carried out using semi-static mappings (see Section 5.3.2) from line segments (e.g. Figure 7.1(c)) to voxel shapes (e.g. Figure 7.1(b)) except for that of the circular target shape (Figure 7.3) which uses a static mapping (see Section 5.3.1). Apart from this, they all used the same settings including a population size of 200 and an identical 30 by 30 voxel space.

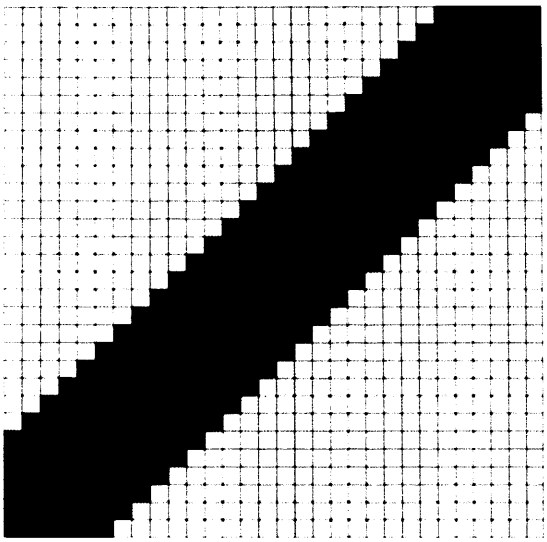
The observations from these previous results are:

1. Genetic algorithms with generative representations are able to locate high quality solutions.

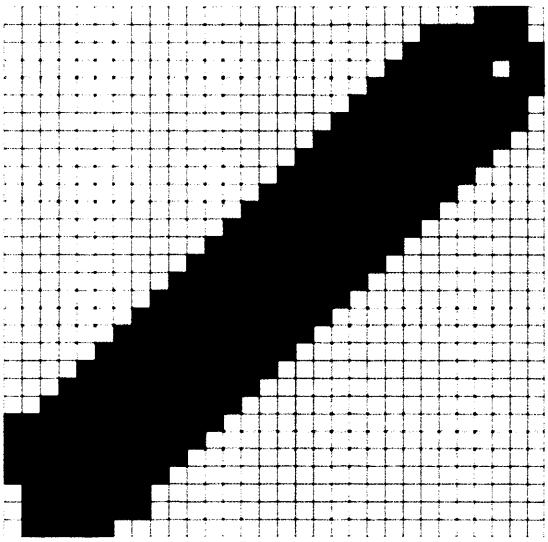
2. These solutions are essentially continuums, i.e. they do not contain isolated voxels or blocks.
3. The reuse of elements plays an importation role in generating these solutions.

Although the results from the previous tests suggest that genetic algorithms and generative representations can be a good combination for topological search problems, they are produced by test problems that are still quite different from the real-world ones. Moreover, the adjustable attributes of the representation and its implementation have not been fully examined by these tests. To address the above issues, a new test problem is formulated to simulate a real-world design problem.

(a) target shape



(b) result in voxel shape



(c) result in line segments

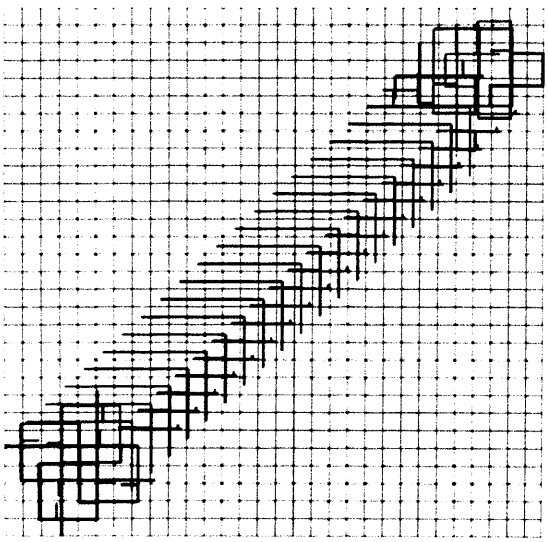
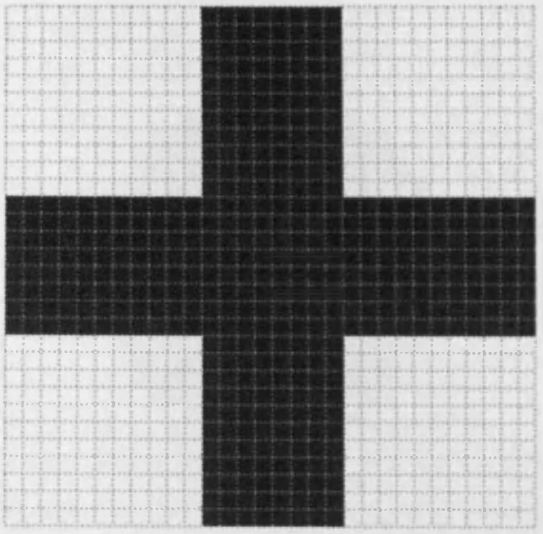
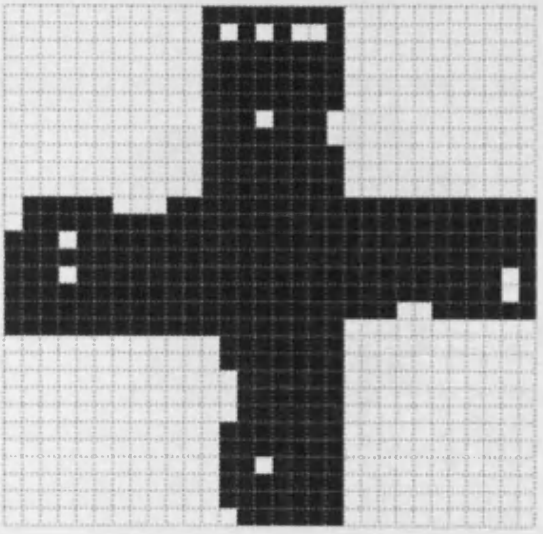


Figure 7.1 shape-matching problem: diagonal shape

(a) target shape



(b) result in 2D continuum



(c) result in line segments

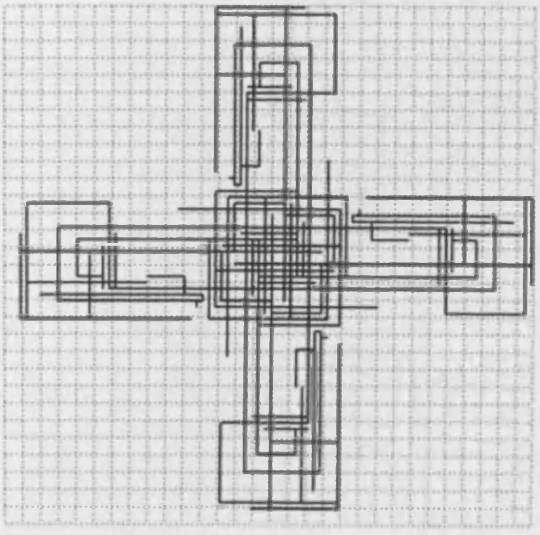
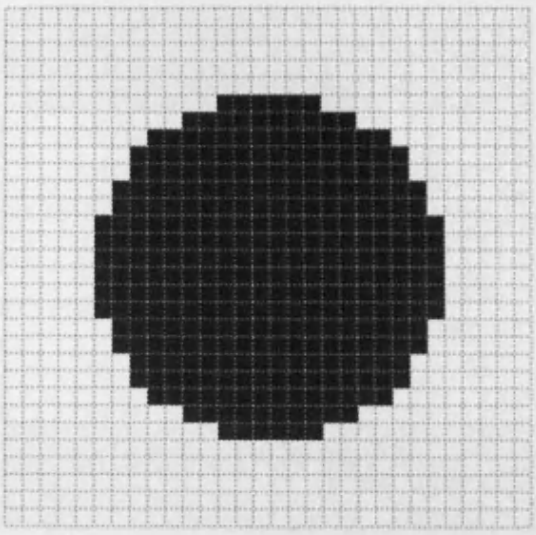
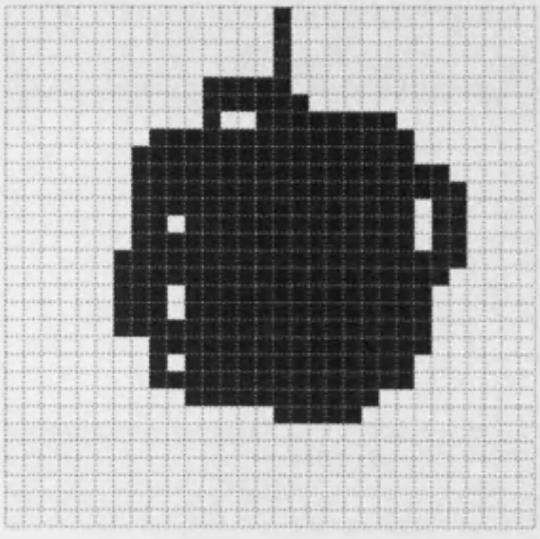


Figure 7.2 shape-matching problem: cross shape

(a) target shape



(b) result in 2D continuum



(c) result in line segments

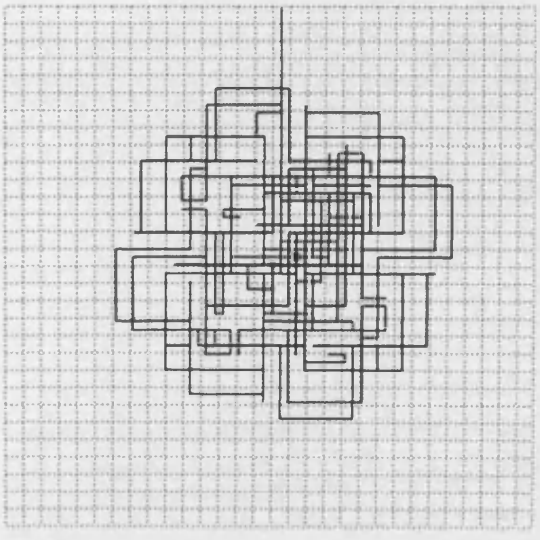
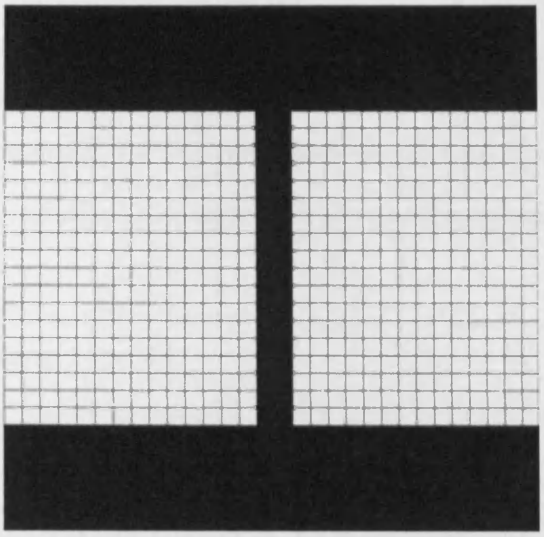
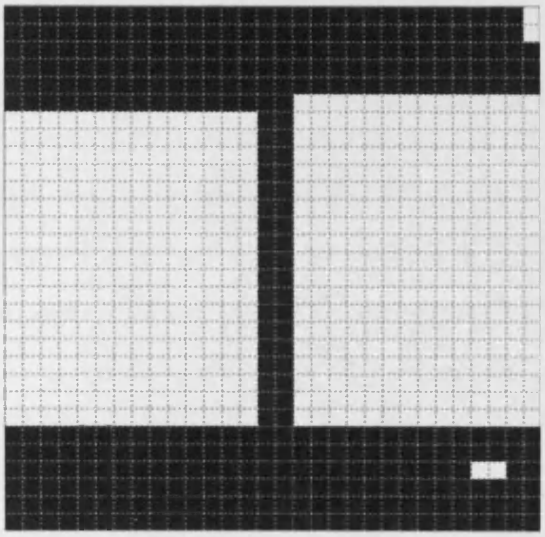


Figure 7.3 shape-matching problem: circular shape

(a) target shape



(b) result in 2D continuum



(c) result in line segments

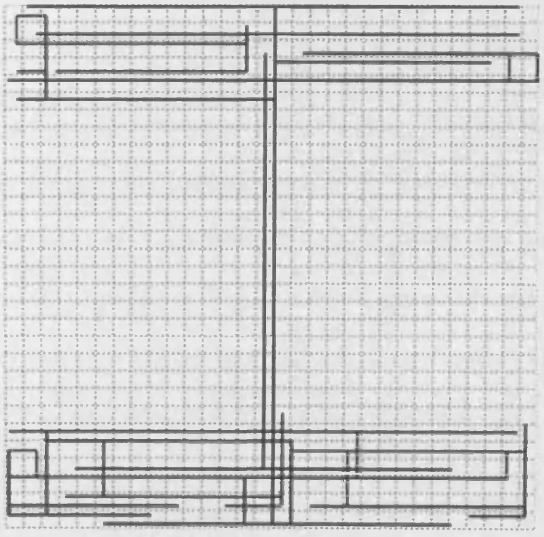


Figure 7.4 shape-matching problem: I shape

7.2 Problem Description and Calculation Basis

In real-world design activities, engineers always design against certain criteria. The criteria form a set of requirements and limitations which become the foundations of design activities. For example, a simple structural design scenario can be described as follows:

A structure component needs to be designed and it

1. must be able to carry certain amount of load;
2. must be able to fit into a space of a given size;
3. is better if lighter.

The first two points are the hard constraints of the design. Satisfying the design criteria makes a design feasible. The third point can be described as a desired feature or a soft constraint for the design. Assuming both criteria are met, a lighter design means a better design. The process of looking for better designs or the best possible designs is the process of design optimisation. With genetic algorithms, the search for feasible and optimal designs is combined into a single process using fitness functions. A fitness function is designed in such a way that an infeasible design is awarded a less competitive fitness value, even though it may have a better feature, for example, being lighter.. It is generally accepted that the population do not have to be all feasible as long as the best individual found is because it will become the solution when generation iteration terminates.

In this chapter, a simulated structural design problem is used to test generative representations. The cross section of a homogeneous beam that is under the action of pure bending is to be designed. Under pure bending, a beam only has normal stress distributed along the length and perpendicular to its cross sectional plane. Normally for a feasible beam design, a size constraint is often applied and the bending stresses must remain below material's elastic stress limit.

Assuming that the position of the neutral axis is known, the maximum bending stress can be calculated using:-

$$\sigma_{max} = \frac{M \cdot d_{max}}{I_{NA}} \tag{7.1}$$

where σ_{max} is the maximum bending stress within the cross section; M is the bending moment; d_{max} is the maximum perpendicular distance from any point within the cross section to the neutral axis; I is the second moment of area of the cross section about its neutral axis.

According to Equation (7.1), given a bending moment M, for the maximum bending stress σ_{max} not to exceed the material's elastic stress limit σ_a, that is, $\sigma_{max} \leq \sigma_a$, the following expression must be true.

$$\frac{M \cdot d_{max}}{I_{NA}} \leq \sigma_a \tag{7.2}$$

Moving all known values to the right side of (7.2), the mechanical criterion of the design can be described as:-

$$\frac{d_{max}}{I_{NA}} \leq \frac{\sigma_a}{M} \tag{7.3}$$

which, in fact, together with the size constraint, defines the geometrical property of the cross section. Hence, the problem, in the sense of design optimisation, is to find a distribution of material within the constrained design space to met the criterion described by (7.3) with the minimum amount of material.

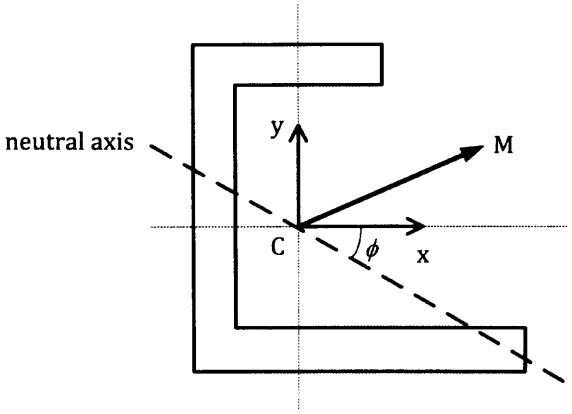


Figure 7.5 neutral axis of asymmetric cross section

Due to the randomness of the cross sections generated by the algorithm, asymmetry needs to be considered when determining the location and orientation of the neutral axis. When a beam is subject to pure bending, the neutral axis has to pass through the centroid of its cross section. However, the orientation of the neutral axis depends on the orientation of the moment vector and the cross sectional shape of the beam. Assuming homogeneity and elasticity, the orientation of the neutral axis as show in Figure 7.5 is given by:-

$$\tan\phi = -\frac{M_y I_x - M_x I_{xy}}{M_x I_y - M_y I_{xy}} \tag{7.4}$$

where M_x and M_y are the bending moments with regard to the x and y centroidal axes; I_x and I_y are the second moment of area about the x and y axes; I_{xy} is the product moment of area. For the test problem, only M_x is applied, that is, $M_y = 0$. Hence, Equation (7.4) can be simplified to Equation (7.5).

$$\tan\phi = \frac{I_{xy}}{I_y} \tag{7.5}$$

Normally, the second moment of area of a cross section is calculated by:-

$$I_\lambda = \int_A y^2 dA \tag{7.6}$$

where I_λ is the second moment of area about axis λ; dA is an elemental area; y is the perpendicular distance from element dA to axis λ. For a cross section described in voxels, first, the second moment of area of each active voxel about the neutral axis of the cross section is calculated according to parallel axis theorem using:-

$$I_{NA}^{(i)} = I_C^{(i)} + Ad^2 \tag{7.7}$$

where $I_{NA}^{(i)}$ is the second moment of area of voxel i with respect to the neutral axis of the cross section; $I_C^{(i)}$ is the second moment of area of voxel i about an axis which is parallel to the neutral axis of the cross section and passing through the centroid of the voxel; A is the area of the voxel; d is the perpendicular distance between the

neutral axis and the centroid of the voxel. Then the second moment of area of the entire cross section I_{NA} is calculated by combining the second moment of are of each active voxel with respect to the neutral axis using:-

$$I_{NA} = I_{NA}^{(1)} + I_{NA}^{(2)} + \cdots + I_{NA}^{(n)} = \sum_{i=1}^{n} I_{NA}^{(i)} \tag{7.8}$$

where n is the total number of active voxels that the cross section is composed of.

The product moment of area (also known as product of inertia) I_{xy} is given by:-

$$I_{xy} = \int_A xy\, dA \tag{7.9}$$

where x and y are the perpendicular distances from element dA to x and y axes, respectively. To calculate the product moment of area of a cross section described in voxels, the parallel axis theorem and the combination method still apply. The parallel axis theorem for product moment of area is given by:-

$$I_{xy}^{(i)} = I_{xCyC}^{(i)} + Abd \tag{7.10}$$

where $I_{xy}^{(i)}$ is the product moment of area of voxel i about x and y axes, respectively; $I_{xCyC}^{(i)}$ is the voxel's product moment of area about its own centroidal axes xC and yC, respectively; b and d are the perpendicular distances from xC to x and from yC to y, respectively; A is the area of the voxel. Using the combination method, the product moment of area of the entire cross section I_{xy} is given by:-

$$I_{xy} = I_{xy}^{(1)} + I_{xy}^{(1)} + \cdots + I_{xy}^{(n)} = \sum_{i=1}^{n} I_{xy}^{(i)} \tag{7.11}$$

where n is the total number of active voxels that the cross section is composed of.

The second moment of area of the cross section with respect to the y axis can be calculated using (7.7) and (7.8) by replacing neutral axis with y axis. Knowing I_{xy} and I_y, the orientation of the neutral axis can be calculated using (7.5). Finally, knowing

the orientation ϕ, d_{max} and I_{NA} can be properly calculated for the determination of the feasibility of the individual using (7.3).

The fitness function used in this study is formulated as is given by (7.12). The fitness value f of an individual is calculated in different ways depending on its feasibility.

$$f = \begin{cases} m_{max} - m, & \sigma_{max} \leq \sigma_a \text{ OR } \dfrac{d_{max}}{I_{NA}} \leq \dfrac{\sigma_a}{M} \text{ (feasible)} \\ f_{avg} \cdot \alpha, & \sigma_{max} > \sigma_a \text{ OR } \dfrac{d_{max}}{I_{NA}} > \dfrac{\sigma_a}{M} \text{ (infeasible)} \end{cases} \tag{7.12}$$

In (7.12), m_{max} is the maximum possible mass of design for the given design space, that is, the mass of the design that fills the entire design space; m is the mass of the design represented by the individual; f_{avg} is the average fitness value of all feasible designs of the current generation; α is a coefficient that reflects how far the individual goes into infeasibility which is calculated by (7.13).

$$\alpha = 2 - \frac{\sigma_{max}}{\sigma_a} = 2 - \frac{d_{max} \cdot M}{I_{NA} \cdot \sigma_a} \tag{7.13}$$

According to (7.12), if a design is evaluated as feasible, its fitness value is calculated by $m_{max} - m$, which ensures that a lighter design is considered as a better design and thus is awarded a higher fitness value. The fitness value of a feasible design is always positive.

If a design is evaluated as infeasible, it is still awarded a fitness value; however, its fitness value is calculated by $f_{avg} \cdot \alpha$. The coefficient α given by (7.13) is always smaller than 1, which means if f is the fitness value of an infeasible design, $f < f_{avg}$ is always true. Meanwhile, according to (7.13), the more σ_{max} exceeds the material's elastic stress limit σ_a or the more d_{max}/I_{NA} exceeds σ_a/M, the lower the fitness value is. The fitness value of an infeasible design can be negative. Formulating the fitness function for infeasible designs in this way ensures: first, the fitness values for infeasible designs are always lower than the average fitness value of all feasible designs of the same generation, which makes the infeasible designs

generally less competitive; second, even among infeasible designs, there are better and worse.

Using $f_{avg} \cdot \alpha$ to calculate the fitness value for an infeasible design is not a very harsh strategy. It makes it possible that certain infeasible designs may be awarded with fitness values that are even higher than some of the feasible ones. This strategy encourages feasible designs; at the same time, it also makes sure that infeasible designs, especially near-feasible designs, are not eliminated too easily. Despite being infeasible, some near-feasible designs may be very close to the optimum. In fact, removing a slightest piece of material from the optimal design will result in such a near-feasible but infeasible design. Keeping such infeasible designs in the population and giving them the opportunity to take part in generating the new population can help to improve the efficiency of the search.

7.3 Genotype Format

The genotype format adopted by Hornby (2003a) is described in Chapter 5 (Section 5.1.4). The L-system used had fifteen production rules. Each production rule had two parameters and three condition-successor pairs. The maximum length of a successor was set to fifteen commands and the maximum length of a compiled generative representation (resultant command string from rewrite) was set to 10,000 commands. Although it appeared that the above genotype setting was good for the problem used in Hornby's work, the choice had not been justified. In this section, experiments are carried out to investigate the genotype format in order to learn the influence of these adjustable attributes on the behaviour of the representation.

The adjustable attributes first include the total number of production rules, the number of parameters for each production and the number of condition-successor pairs for each production. Why do these numbers matter? A simple L-system can be formed by a single production rule with no parameters, no conditions and no other production rules. However, it would lack the ability of abstraction and control-flow which is explained by the example given by Figure 7.6. The graphs shown in Figure 7.6 are generated by an L-system of a single, non-parameterized and non-conditional production rule given by (7.14).

$$P_0: F(1)P_0L(3)P_0P_0F(3) \tag{7.14}$$

As shown in Figure 7.6, by increasing n (the number of rewrites), the graphs generated simply repeat a uniform pattern which can be clearly recognized in Figure 7.6 (a). It can be seen by comparing a graph generated by a lower number of rewrites (e.g. Figure 7.6 (a)) with that generated by a higher number of rewrites (e.g. Figure 7.6 (f)) that, if an L-system of a single, non-parameterized and non-conditional production rule is used, increasing the number of rewrites can increase the size of the resultant graph; however, the complexity of the graph, with respect to the elements it is composed of, remains unchanged.

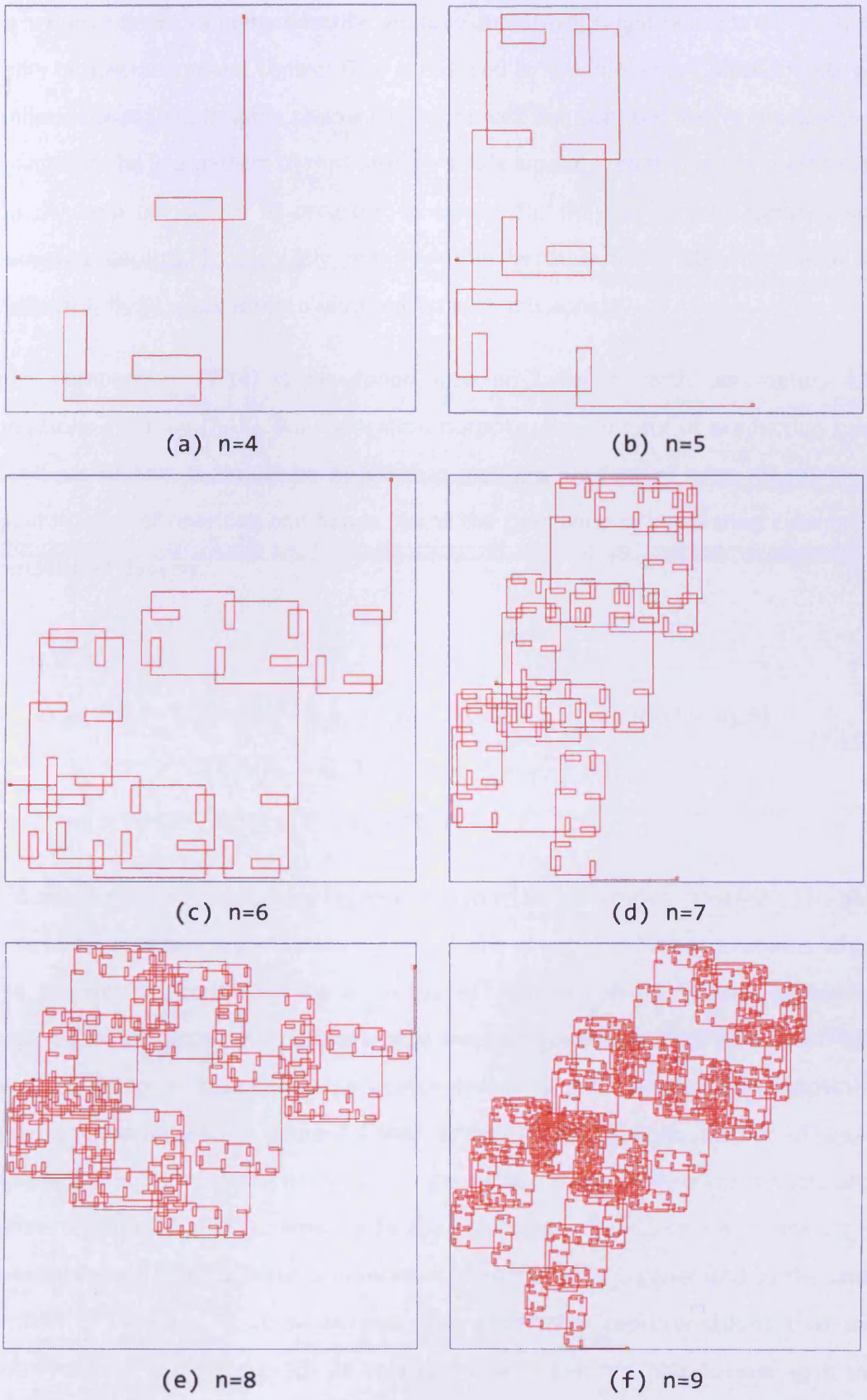


Figure 7.6 an example L-system of a single, non-parameterized and non-conditional production rule

If a representation can only describe an individual by reusing one single element, its ability of abstraction and control-flow is reduced to the minimum, which effectively confines the search. In some special circumstances, for example, where the design is expected to be in a pattern of replicating a single element, confining the search may actually help the search to progress; however, for the searches of complex and innovated designs, it is usually not desirable because being able to search an adequately large space is often required for such missions.

As a comparison, (7.14) is developed into an L-system with parameters and conditions given by (7.15). For illustration purpose, the number of production rules is still set to one. It should be noted that multiple production rules means more possible ways of rewriting and hence opens the possibility of generating even more complicated designs.

$$
\begin{aligned}
& n_0 = 4;\ n_1 = 8 \\
& P_0\colon\ n_0 > 6 \ \rightarrow F(2)P_0(n_0 - 2, n_1 + 1)L(1)P_0(n_1 + 1, n_1 + 2)P_0(1 - n_0, 9) \\
& \quad n_1 > 7 \ \rightarrow F(2)[P_0(n_1 + n_0, 8)]\{P_0(n_0 - 1, n_1 - 1)\}(3) \\
& \quad n_1 > 0 \ \rightarrow \{F(2)[P_0(n_1 + 1, n_0 + 1)]\}(2)
\end{aligned}
\tag{7.15}
$$

As discussed in Chapter 4, for a representation to be generative, it needs to be able to describe a design in the abstract through the reuse of different elements which also respond to conditions as a means of control-flow. It is how generative representations distinguish themselves to the non-generative ones and is also why they are better in handling large search space to find complex and innovative designs. It can be seen in Figure 7.7 that, as rewriting progresses, instead of simply repeating a uniform element, the graph generated starts to develop clusters with different characteristics. Comparing to the graph given by Figure 7.6 (f), the graph given by Figure 7.7 (f) is more complicated, even though it is generated by the same number of rewrites. It demonstrates that generative representations that use parameterized L-systems (such as that given by (7.15)) are able to deal with the representation of more complicated designs.

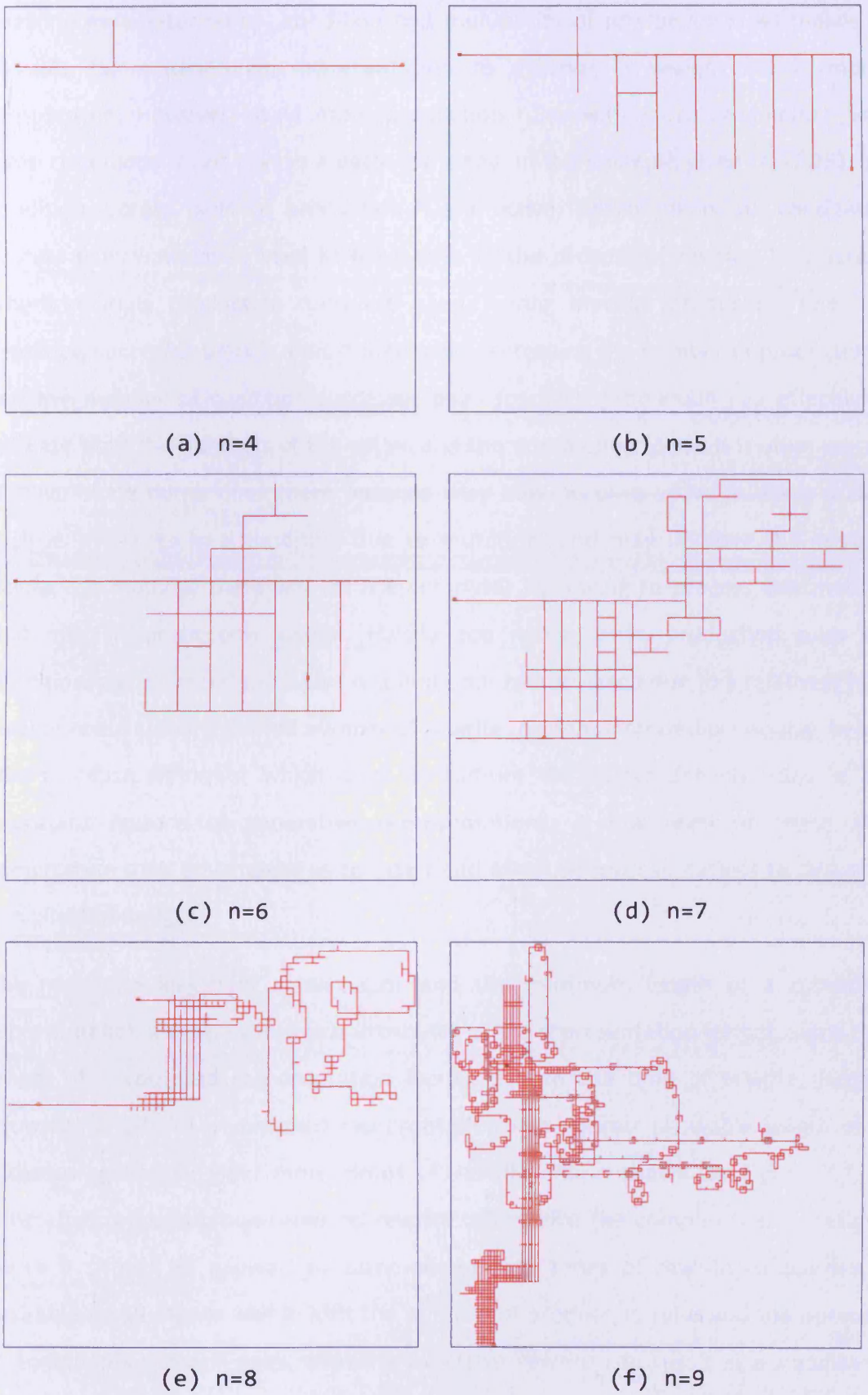


Figure 7.7 an example L-system with parameters and conditions

Enabling parameterization, condition and multiplicity of production rules makes it possible for a generative representation to describe a design that is more complicated. However, using more production rules with more parameters and more conditions is not always a better strategy. In the example given by (7.15), all condition-success pairs of production P_0 are active, which means all condition-success pairs have been used at least once in the process of rewrite. In practice where multiple production rules are used, having inactive production rules or condition-successor pairs is almost inevitable. Increasing the number of productions and the number of condition-successor pairs for each production can effectively increase both the numbers of the active and the dumb ones. While it is often worth to have some dumb ones there because they may become active at some point, such as a change in a condition due to mutation, and may improve the design, having too many of them will tax the computer by having to process information that may never become useful. Having too many active production rules or condition-successor pairs can also result in poor performance due to a relatively low level of reuse within a limited number of rewrite. As demonstrated previously, being able to reuse elements, which helps to capture the design dependencies, is an important feature of generative representations. A low level of reuse will compromise their advantages as compact and effective representations to describe complicated designs.

The maximum length of a successor and the maximum length of a compiled representation are also adjustable attributes of the representation format. Since the length of a compiled representation increases with the time of rewrite, longer allowable length of a compiled representation and shorter allowable length of a successor generally yield more times of rewrite. As is shown by Figure 7.7, a generative representation relies on rewrite to describe the complexity of a design; hence it should be allowed to carry out enough times of rewrite to achieve a desirable result. Again, like it with the number of production rules and the number of condition-successor pairs, allowing excessive rewrite can result in a significant increase in computational cost. Experiments have been carried out to investigate the influence of these two adjustable attributes. Table 7.1 presents the result of the

experiment that examines the influence of the maximum length of a compiled representation. For this experiment, the maximum length of a successor is set to a constant 10. Data shown in Table 7.1 are drawn from averaging results of ten GA runs of one hundred generations with a population pool of two hundred individuals. Data plotted in Figure 7.8 are standardized to display increments for the purpose of comparison.

Table 7.1 influence of the maximum length of a compiled representation

A	T	R	L
0.5	218.63	11.08	1544.41
1.0	486.15	11.53	3073.48
1.5	937.25	11.71	5115.65
2.0	1319.51	11.72	6452.01
2.5	2024.75	11.81	7823.91
3.0	2720.29	11.90	9215.55

A – maximum length of a compiled representation (in $\times$ 1000 commands)
T – time taken by processing one hundred generations (in second)
R – average times of rewrite of all individuals processed
L – average length of the compiled representation of all individual processed (in command)

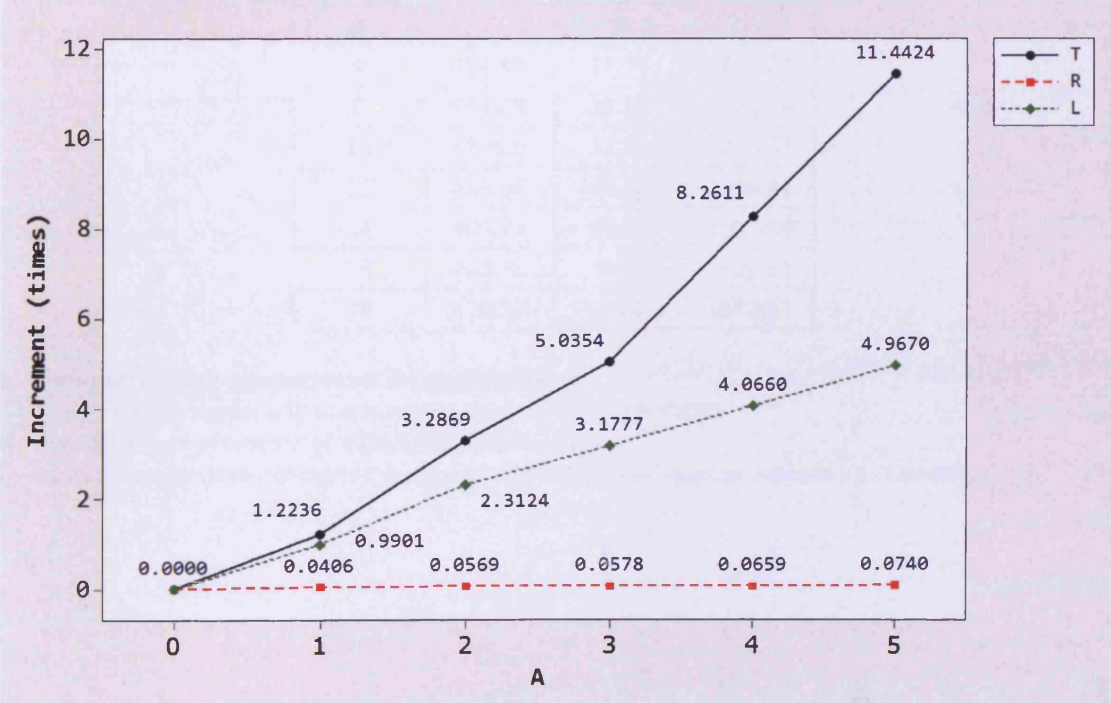


Figure 7.8 increment in T, R and L against increment in A

As is shown by the result above, an increment in the maximum length for a compiled representation (A) can cause significant increase in computational cost (T), although

it causes proportional ($\approx 1:1$) increments in the average de facto length of all compiled representations (L) of its own increments and very slight increments in the average times of rewrite. Since a generative representation relies on more times of rewrite to describe a design of a higher complexity, increasing the maximum length for a compiled representation has very limited effect on improving the overall complexity of a population comparing to it has on increasing the computational cost.

With the experiment result presented below, the influence of the maximum length of a successor is explained. For this experiment, the maximum length of a compiled representation is set to a constant 10,000. As same as that of the previous experiment, the data presented are drawn from averaging a total number of ten GA runs of one hundred generations with a population pool of two hundred individuals. The raw data from the experiment are listed in Table 7.2 influence of the maximum length of a successorData plotted in Figure 7.9 are standardized to display increments with negative values indicating decrements.

Table 7.2 influence of the maximum length of a successor

B	T	R	L
6	456.89	19.76	2894.26
8	534.03	13.34	3371.79
10	483.69	11.41	3149.05
12	463.69	10.54	2936.91
14	503.72	10.03	3182.34
16	429.80	9.08	2927.56
18	470.58	4.88	3040.17

B – maximum length of a successor (in command)
T – time taken by processing one hundred generations (in second)
R – average times of rewrite of all individuals processed
L – average length of the compiled representation of all individual processed (in command)

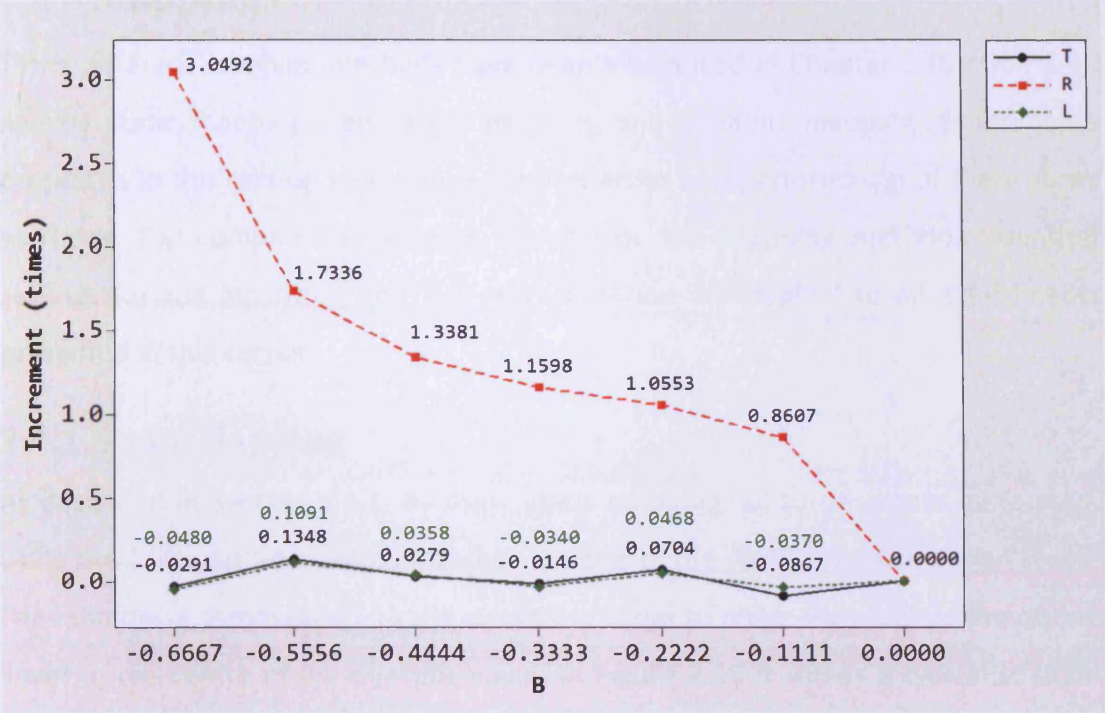


Figure 7.9 increment in T, R and L against increment in B

As is shown in Figure 7.9, decreasing the maximum length for a successor can effectively result in an increase in the population's average times of rewrite while having little influence on the average length of compiled representations and computational cost. Hence, the maximum length for a successor should be the attribute of choice when the overall complexity of the population needs to be adjusted.

Because the problems to be dealt with vary from one to another, there is not a genotype format that is universally good for any problems. As long as the properties of these attributes are understood, a good genotype format is often just a few tests away. It is well worth the effort to find a good format that adapts to the problem because it can not only help the algorithm to work efficiently by reducing unnecessary computational cost but also helps to improve the chance to find good solutions by making sure that the representations are actually capable of describing designs at the desired level of complexity.

7.4 Mappings

Three different mapping methods have been introduced in Chapter 5 (Section 5.3), namely static mapping, semi-static mapping and dynamic mapping. Experiments presented in this section investigate the properties and performance of these three methods. For comparison purpose, except for the mapping methods, identical settings for the algorithm and the representation are applied to all experiments presented in this section.

7.4.1 Static Mapping

As described in Section 5.3.1, by using static mapping, all turtle graphs generated, use a fixed step distance which matches the size of the voxel of a fixed design space. For example, a command $F(1)$ will direct the turtle to move from the centre of one voxel to the centre of an adjacent voxel. In Figure 7.10 it shows a typical solution found by using static mapping using a fixed design space of 30 voxels by 30 voxels.

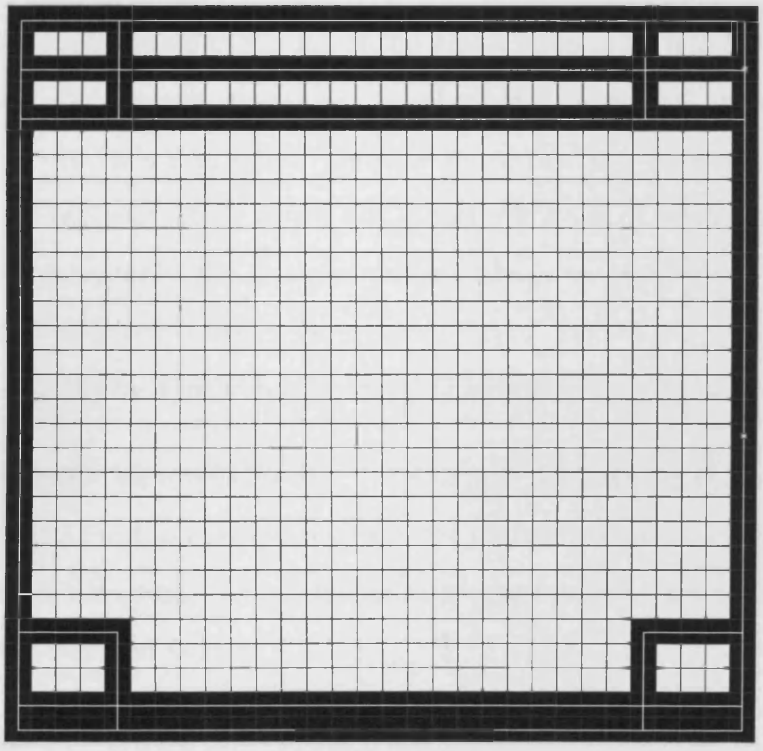


Figure 7.10 an example solution found by using static mapping

It can be seen from Figure 7.10 that, although, to a certain degree, the algorithm managed to capture the intention of design – to place most material to the top and bottom part of the design space to achieve a required I value with the minimum amount of material – the solution found is some way from being satisfactory.

Table 7.3 statistical data for static mapping

R	L	I
5.84	196.14	8.21%

R – average times of rewrite of all individuals processed
L – average length of the compiled representation of all individual processed (in command)
I – average best fitness improvement

Table 7.3 presents the statistical data from five independent runs of 1,000 generations using static mapping. The data indicate that, static mapping places a very strong constraint on the representation. Because the turtle graphs generated need to be able to fit into the design space, the algorithm can only accepts an individual that represents a graph within a predefined size as a legal individual. For a 30 by 30 design space, the dimensions of the graph have to be less than 29 steps in both horizontal and vertical directions. It effectively limits the average times of rewrite of all individuals processed, because more rewrites generally results in a longer compiled representation which has a better chance of being oversize. Using static mapping highly limits the representation's ability to explore the search space as the search is restricted to zones where only 'fitted' individuals reside. It explains why the fitness of the best individual has very little improvement over generations.

Even by adjusting the genotype format, for example, to increase the average times of rewrite by reducing the maximum length for successors, static mapping has not been found to be a method that is capable to produce satisfactory solutions for the test problem described in Section 7.2.

7.4.2 Dynamic Mapping

By using dynamic mapping, the restriction on the maximum size of a turtle is removed. As described in Section 5.3.3, the mapping uses neither a static grid for a voxel shape nor a static step size for a turtle graph. However, the size of a voxel is still consistent with the step distance of a turtle graph, which means one step

forward ($F(1)$) still directs the turtle to move from the centre of one voxel to the centre of an adjacent voxel. The difference is that both quantities are able to adjust their scales against the design space in accordance with the size of the turtle graph. In this way, any turtle graph is guaranteed to fit in the design space, no matter the size.

Figure 7.11 shows one of the two kinds of typical solutions found by using dynamic mapping method. In most cases, although the restriction on the size of a turtle graph is lifted, the algorithm still favours turtle graphs of smaller size. As is shown in Figure 7.11, the turtle graph (white lines) is in the size of fourteen steps by fourteen steps which can be considered to be very small considering a compiled representation is allowed to take up to 10,000 commands; however, mapping the turtle graph into the design space results in a moderately good solution. Although there is still redundant material in the middle part, the solution contains clearly formed top and bottom flanges to provide enough support for bending moment. Compared to the solution given by Figure 7.11, it appears to be a more reasonable design.

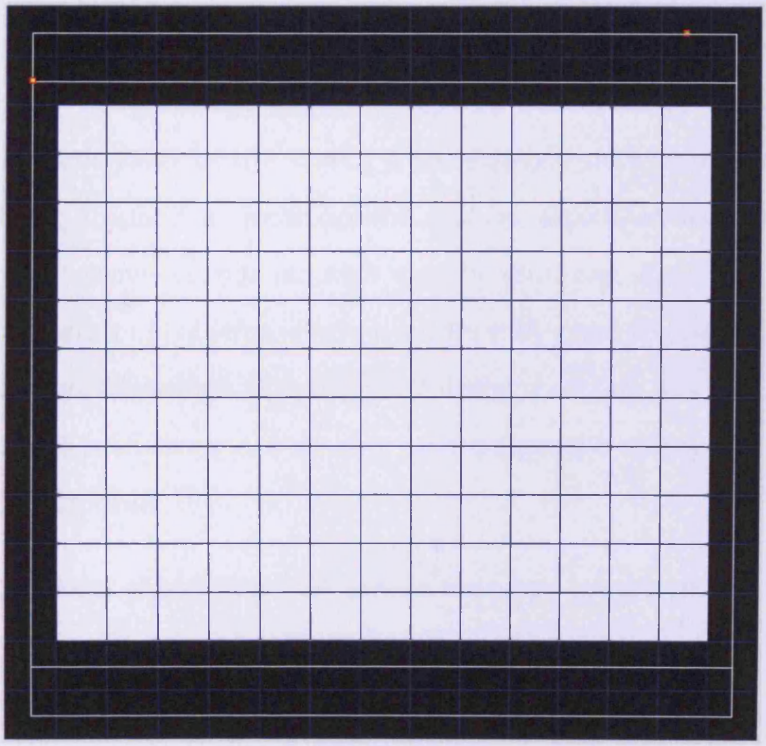


Figure 7.11 an example solution of a coarser grid found by using dynamic mapping

By using dynamic mapping, a smaller turtle graph (measured in steps) means a coarser grid. Mapping a turtle graph into a coarser grid is more likely to produce designs that have enough solid parts to provide the required value in the second moment of area, which is why the algorithm generally finds solutions that are similar to this.

It is also found that, by using dynamic mapping, the algorithm is able to locate a relatively good solution in very early generations with subsequently more gradual improvements in later generations. It is very similar to the search behaviour when static mapping is used. However, for the test problem used here, dynamic mapping generally produces better results than static mapping. Statistical data also shows better fitness improvement (Table 7.4).

Table 7.4 statistical data for dynamic mapping

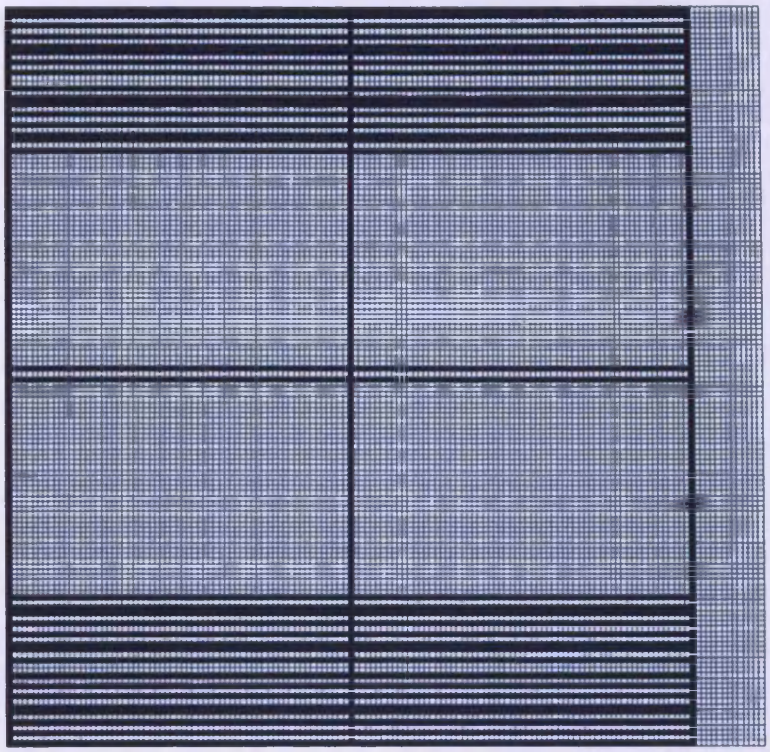
R	L	I
13.44	1978.50	15.66%

R – average times of rewrite of all individuals processed
L – average length of the compiled representation of all individual processed (in command)
I – average best fitness improvement

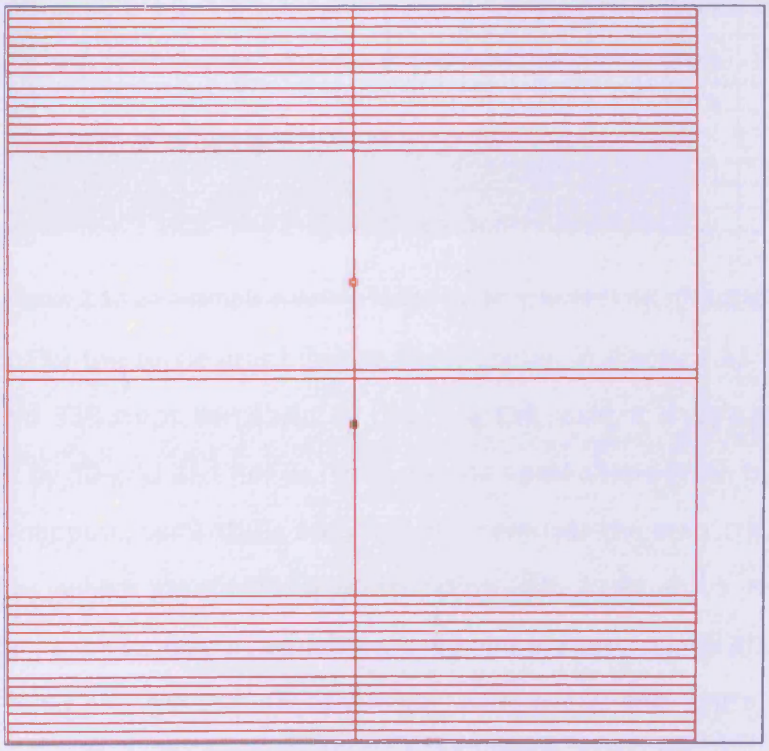
For the problem being solved here, it appears that finding moderately good solutions at the early stage of the search with relatively slow improvement later on is a sign of being trapped at local optima. For an algorithm that is designed to perform more extensive search for high quality solutions, it can be a significant drawback. However, for problems where a moderately good solution is good enough, it could become an advantage. A conceptual design problem where designers look for ideas is one of such situations as long as the algorithm is capable to provide a rich selection of solutions.

There is another kind of solution that can be found by using dynamic mapping. An example is given by Figure 7.12. Although such solutions are not found as common as the kind illustrated by Figure 7.11, it shows another property of dynamic mapping – it can produce voxel shapes with very thin parts. In contrast with the solution given by Figure 7.11, this solution actually uses a very fine grid (142 voxels by 142 voxels) because of the size of the turtle graph (141 steps by 129 steps). Mapping the turtle

graph with parallel line segments that are not close enough to each other (Figure 7.12 (b)) into a fine grid produces a voxel shape in zebra stripe style (Figure 7.12 (a)). For dynamic mapping, a turtle graph of such size can only be mapped into such a grid. If the turtle graph can be mapped into a relatively coarser grid, it will be able to produce a voxel shape with more solid parts, which may potentially result in a better solution. It can be enabled by using the semi-static mapping method which is tested in the following section.



(a) voxel shape



(b) turtle graph

Figure 7.12 an example solution of finer grid found by using dynamic mapping

7.4.3 Semi-static Mapping

For semi-static mapping, dynamic scaling is applied to turtle graphs so that turtle graphs of any size are able to fit into a static predefined grid. For the test problem used here, semi-static mapping is found to produce the best results and one of them is shown in Figure 7.13.

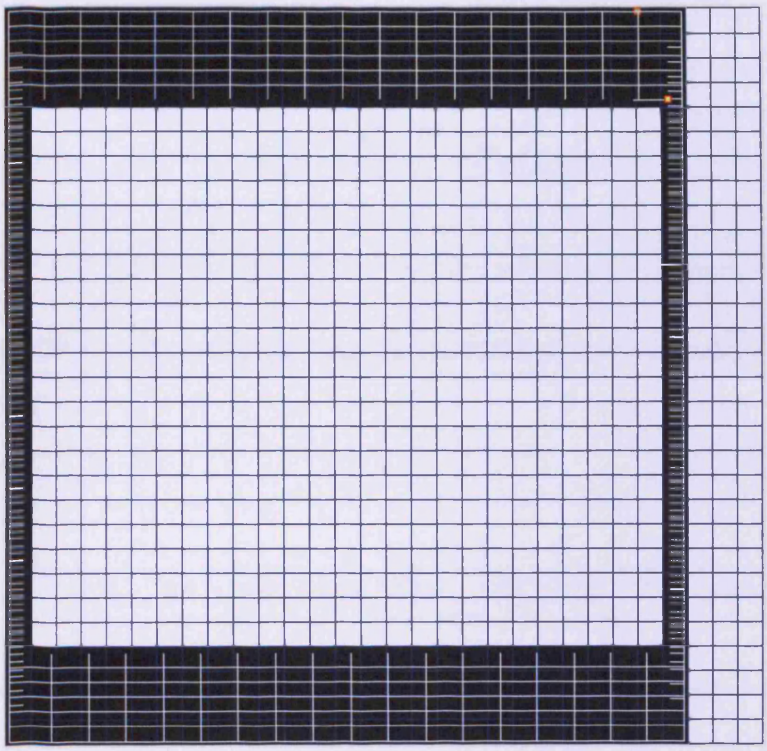


Figure 7.13 an example solution found by using semi-static mapping

The dimensions of the turtle graph (while lines) shown in Figure 7.13 are 303 steps horizontally and 338 steps vertically. By reducing the scale, it is able to be mapped into a fixed 30 by 30 grid and hence results in the voxel shape given by Figure 7.13. Like dynamic mapping, semi-static mapping also removes the restriction on the size of turtle graphs, which grants the representation with freedom to generate more versatile solutions; but unlike it, by using a fixed and predetermined grid, semi-static mapping prevents the generation of designs with overly thin parts like the one shown in Figure 7.12. A much higher level of element reuse is also observed by using semi-static mapping. It helps the representation to capture the design dependencies by replicating characteristic elements to construct different parts of a design.

In general, using semi-static mapping results in a better exploration of the search space. It allows turtle graphs of required level of complexity to be generated and deployed without being trapped at a local optimum which corresponds to a relatively simple turtle graph and an overly coarse grid. The fitness graph given by Figure 7.14 also proves this by showing steady improvement in fitness throughout the search.

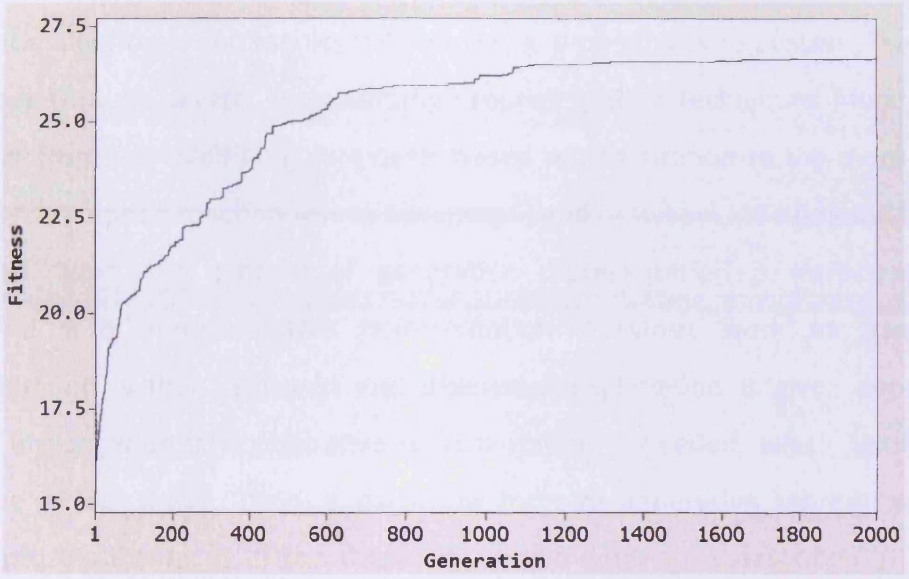


Figure 7.14 fitness graph of an example run using semi-static mapping

7.5 Summary

The experiments and results shown in this chapter demonstrate that, a suitable representation setting for one problem may not be suitable for another. For example, static mapping which was used by Hornby (2003a) had been found to produce good results for his table design problem; however, for the beam cross section design problem used in this chapter, it is far from being satisfactory. To use generative representation, one could always start with a setting of a "good guess". However, making the representation actually fit for the problem being solve often requires further considerations.

Chapter 8 Conclusions

8.1 Summary

This thesis describes an investigation of the use of a particular generative representation with a genetic algorithm for topological reasoning. First, genetic algorithms are introduced. Then the impact of representation techniques on the use of genetic algorithms for topological reasoning problems is discussed. The thesis then goes through several representative representation techniques found in the literature from the traditional parameter-based representation to the more recent topology description function whose advantages and drawbacks are pointed out and discussed. Next, the concept of generative representation is introduced and compared with non-generative representation. Previous work on generative representation is then reviewed and discussed. Explanation is given about why further investigation into generative representation is needed, which justifies the objective of the thesis. Then, a particular form of generative representation is introduced. As the focus of the thesis, its implementation is described in details, including the format of an individual as an L-system, the rewrite and that had not been mentioned and discussed in the literature, the different mapping methods. What is also new and comes next in the thesis discusses crossover and mutation in genetic algorithm in connection with the generative representation used. Finally, experiments and results are presented.

8.2 Key Results and Findings

The experimental results presented in Chapter 7 prove that generative representation is indeed a competitive representation method to be used with genetic algorithm to deal with topological reasoning problems. The test problem used in this study is a design problem for a beam cross-section under pure bending. The object is to look for an optimal quantity and distribution of material within a predefined design space that satisfies one mechanical constraint, that is, the maximum bending stress σ_{max} does not exceed the material's elastic stress limit σ_a. In literature, the same test problem had been used to study voxel representation. By

comparison, the solutions found by using generative representation have the following two major advantages over those found by using voxel representation.

1. Solutions found by using generative representation do not have continuity issues. Solutions generated by voxel representation have two problems: first, they often contain useless isolated voxels; second, the continuity of major parts is not guaranteed and often requires additional repair to maintain. By nature, any solution given by generative representation is a single piece of continuum. There is no isolated voxel or the need to take extra care to maintain the continuity.
2. Solutions given by generative representation clearly show the reuse of element which does not exist in voxel representation. The reuse of element in representing designs indicates that generative representation is compact and capable of capturing design dependencies. This characteristic helps GA to conduct a more efficient search of a complex solution space.

Comparing to the other representation methods that are reviewed in this thesis, generative representation is the only representation method that is in compact form and is able to generate designs that do not have continuity problems.

The study also looks into the genotype formatting for generative representation and its influence on the performance of the representation method and the algorithm. The study first demonstrates that using parameter and condition-enabled L-systems is essential for representing designs of high complexity. If a non-parameterized and non-conditional L-system is used, rewriting the L-system can at the best increase the size of the resultant design; however, the complexity of the design, in terms of the variety of the elements it is composed of, remains unchanged. By enabling parameters and conditions, different production rules can get involved in the rewriting, which introduces different elements into the system to be reused to construct designs of higher complexity. For innovative design and optimisation problems, enabling parameters and conditions for a generative representation is essential.

The study then examines another two adjustable attributes for the genotype formatting, the maximum allowable length for a complied representation and the

maximum allowable length for a successor. Setting upper limits to these two values are for the consideration of computational cost and efficiency. Generative representation itself is compact, which makes it computationally cheap to operate on encoded individuals for crossover and mutation, and even to maintain the entire population pool. However, an encoded individual needs to be converted into its phenotype, the actual design, for fitness evaluation. This conversion process, which takes up the majority part of the computational cost of the entire system, involves rewriting the L-system to get the command sequence (compiled representation), generating the turtle graph according to the command sequence and mapping the turtle graph into the design space to finally produce the actual design.

Given that the L-system used is parameter and condition-enabled, the level of complexity that the representation is able to represent depends on the number of rewriting times. Each time the L-system rewrites itself, the length of command sequence grows, so does the complexity of the resultant design. Since the length for a compiled representation needs to have a limit, there are two ways to increase the number of rewriting times. The first way is to increase the maximum allowable length for a compiled representation, expecting it to be able to deal with more rewriting circles. The second way is to reduce the maximum allowable length for a success, which reduces the growth in the length of command sequence for each rewriting circle so that more rewriting circles can fit in. The study reveals that, increasing the maximum allowable length for a compiled representation increases the computational cost dramatically but has little effect on increasing the average number of rewriting times of the population. Reducing the maximum allowable length for a successor, in contrast, can effectively increase the average number of rewriting times without taxing the system. Hence, these two attributes need to be carefully considered to form a suitable genotype format that is capable to provide sufficient complexity at reasonable computational cost.

GA operators, namely crossover and mutation, are another focus of the study. The study not only provides the different ways that crossover and mutation can be implemented but also reveals what they do in generating new individuals when used with generative representation. Four crossover methods, including a new method

named as mass crossover, are introduced and discussed. The properties of these crossover methods are listed in Table 8.1.

Table 8.1 properties of different crossover methods

Method	Properties
Block-based	Production symbol NOT involved: • Command sequences (individuals in compiles form) of child individuals strongly resemble that of their dominant parents. • Child individuals have a relatively good chance to represent designs that resemble those represented by their dominant parents.
	Production symbol involved: • Resemblance in command sequence between child individuals and their dominant parents is weakened. Vague but recognisable resemblance in command sequence to the non-dominant parents emerges in child individuals. • Child individuals' resemblance to dominant parents in design and the chance that such resemblance happens are both reduced. Designs represented by child individuals have a relatively low chance to present certain characteristics of their non-dominant parents.
Successor-based	• Compared to block-based crossover, resemblance in command sequence between child individuals and their dominant parents is further weakened, whereas that between child individuals and their non-dominant parents is strengthened. • Compared to block-based crossover, child individuals' resemblance to their dominant parents in design and the chance that such resemblance happens are both further reduced. Designs represented by child individuals still have a relatively low but slightly better chance to present certain characteristics of their non-dominant parents.
Pair-based	• Compared to successor-based crossover, child individuals' resemblance to non-dominant parents in command sequence is improved due to the involvement of conditions. • Compared to successor-based crossover, the chance that designs represented by child individuals present certain characteristics of their non-dominant parents is improved.
Mass crossover	• Mass crossover seeks the balance in contribution between the parents in generating child individuals. • Command sequences of child individuals show resemblance to both parents. • Compared to the other crossover methods, mass crossover has the best chance to inherit design characteristics from both parents.

Four different mutation methods, distinguished by the different bits of information they modify, are also introduced and studied. Their properties are listed in Table 8.2.

Table 8.2 properties of different mutation methods

Method	Properties
Starting symbol	Starting production: • Highly aggressive mutation as call map is changed at the start of the rewriting.
	Starting production parameter: • Less aggressive than mutating starting production as call map change may happen in a delayed manner (in later rewriting circles). • Certain topology characteristics may be preserved.
Rewriting times	• Topology characteristics are very likely to be preserved. • Aggressive mutation method as it modifies the complexity of the resultant topology.
Condition	• Aggressive mutation. • Change in call map can happen in either a prompt or a delayed manner.
Successor symbol	Production symbol involved: • Aggressive mutation as call map is changed. • Certain topology characteristics may be preserved as call map change may happen in a delayed manner.
	Production symbol NOT involved: • The least aggressive mutation of all.

Using synergy among different crossover and mutation methods can aid the navigation in the search space. It is considered a good practice to make all these different methods available to GA and leave the option open to adjust the probabilities for them to happen.

The study also shows it is important to choose a proper mapping method for the problem being dealt with. Three different mapping methods are studied. One of them has appeared in the literature and is referred to as static mapping in this thesis. The other two methods are new and are referred to as dynamic mapping and semi-static mapping. Properties of these mapping methods are listed in Table 8.3.

Table 8.3 properties of different mapping methods

Method	Properties
Static	• It places a strong constraint on the representation as the size of the turtle graph is restricted. • It effective limits the number of rewriting times and reduces the complexity that the representation is able to represent.
Dynamic	• No restriction on turtle graph size. • It is able to find moderately good designs very fast. • Solutions found are either in a coarse grid (local optimum, lack of details), or in a very fine grid (local optimum, too many thin parts).

Semi-static	• No restriction on turtle graph size. • It produces designs with required complexity without being trapped at local optima.

It should be noted that, the choice of mapping method depends on the problem. For the test problem used in this study, semi-static mapping is found to produce the best designs. But it does not mean it is the best method for any problem. It should be considered wise to carry out experiment to determine which method is the best for the specific problem to be solved.

8.3 Future Work

Using generative representations with GAs to solve topological reasoning problems is still a relatively new area of research which requires further studies. Based on this study, four possible directions for future work are listed below.

1. Three dimensional problems:

   The results and findings presented in this thesis are based on a two dimensional structural design problem. Whether or not these principles still apply to three dimensional problems is a question to be answered. Since many real-world design problems are three dimensional, progressing the study of generative representations with GAs to an additional dimension is a worthy step to move forward.

2. GA operators:

   Both crossover and mutation can be implemented in different ways. This study shows the properties of different crossover and mutation methods when used to generative representations. However, the strategy used in this study to achieve synergy among these different methods is still rather intuitive. Different crossover methods are set to happen in equal probability. The same applies to different mutation methods with their overall probability to be adjusted based on human observation and judgement. For a better synergy among these different methods, future work should look into the arrangement of these methods and its influence on the search. A better synergy among different crossover and

mutation methods can help to form more efficient search and to find better solutions.

3. Real-world design problems:

   The structural design problem used as the test problem in this study is rather simple. The value of generative representations in real-world applications is yet to be proved. In order to do this, studies need to be carried out to test their capability on problems of real-world complexity. Finite element analysis may be used to provide more accurate evaluations of designs.

4. Regularity and irregularity:

   The optimised designs found by using generative representations show high degree of regularity. The test problem used and the representation's reuse of element can both contribute to this result. It raises a series of challenging questions about generative representations. Is generating designs of high regularity one of the characteristics of generative representations? If yes, is this characteristic an advantage or drawback of the representation? Are generative representations capable of dealing with problems that favour irregular designs? To answer these questions, future work can start from testing generative representations on problems where the known optima solutions contain irregularity.

In a word, the studies of generative representations are still not extensive. A considerable amount of future work is needed to improve the understanding of generative representations and their applications on real-world problems.

References

ABELSON, H. & DISESSA, A. A. (1981) *Turtle geometry : the computer as a medium for exploring mathematics,* Cambridge, Mass, MIT Press.

AZID, I. A. & KWAN, A. S. K. (1999) A layout optimisation technique with displacement constraint. *Optimization and Control in Civil and Structural Engineering,* 71-77.

BARON, P., FISHER, R., TUSON, A., MILL, F. & SHERLOCK, A. (1999) A voxel-based representation for evolutionary shape optimization. *Ai Edam-Artificial Intelligence for Engineering Design Analysis and Manufacturing,* 13, 145-156.

BEASLEY, D. (1997) Possible applications of evolutionary computation. IN BÄCK, T., FOGEL, D. B. & MICHALEWICZ, Z. (Eds.) *Handbook of Evolutionary Computation.* Institute of Physics Pub and Oxford University Press.

BEKIROGLU, S., DEDE, T. & AYVAZ, Y. (2009) Implementation of different encoding types on structural optimization based on adaptive genetic algorithm. *Finite Elements in Analysis and Design,* 45, 826-835.

BELEGUNDU, A. D. & ARORA, J. S. (1985) A study of mathematical programming methods for structural optimization. Part I: Theory. *International Journal for Numerical Methods in Engineering,* 21, 1583-1599.

BENDSØE, M. P. (1995) *Optimization of structural topology, shape, and material,* Berlin ; New York, Springer-Verlag.

BENDSØE, M. P., GUEDES, J., PLAXTON, S. & TAYLOR, J. E. (1996) Optimization of structure and material properties for solids composed of softening material. *International Journal of Solids and Structures,* 33, 1799-1813.

BENDSØE, M. P. & KIKUCHI, N. (1988) Generating optimal topologies in structural design using a homogenization method. *Computer Methods in Applied Mechanics and Engineering,* 71, 197-224.

BENDSØE, M. P. & RODRIGUES, H. C. (1991) Integrated topology and boundary shape optimization of 2-D solids. *Computer Methods in Applied Mechanics and Engineering,* 87, 15-34.

BEYER, H.-G. (2001) *The theory of evolution strategies,* Berlin ; New York, Springer.

BOHNENBERGER, O., HESSER, J. & MÄNNER, R. (1995) Automatic design of truss structures using evolutionary algorithms. *Proceedings of the Second IEEE International Conference on Evolutionary Computation (ICEC' 95).* Perth, Australia.

BORKOWSKI, A., GRABSKA, E., NIKODEM, P. & STRUG, B. (2003) Searching for innovative structural layouts by means of graph grammars and evolutionary optimization. *System-Based Vision for Strategic and Creative Design, Vols 1-3,* 475-480.

CHAPMAN, C. D., SAITOU, K. & JAKIELA, M. J. (1994) GENETIC ALGORITHMS AS AN APPROACH TO CONFIGURATION AND TOPOLOGY DESIGN. *Journal of Mechanical Design,* 116, 1005-1012.

CHENG, F. Y. & LI, D. (1997) Multiobjective optimization design with Pareto genetic algorithm. *Journal of Structural Engineering,* 123, 1252-1261.

DE RUITER, M. J. & VAN KEULEN, F. (2000) Topology optimization: Approaching the material distribution problem using a topological function description. IN

TOPPING, B. H. V. (Ed.) *5th International Conference on Computational Structures Technology/2nd International Conference on Engineering Computational Technology.* Leuven, Belgium, Civil Comp Press.

DE RUITER, M. J. & VAN KEULEN, F. (2004) Topology optimization using a topology description function. *Structural and Multidisciplinary Optimization,* 26, 406-416.

DEB, K. (2000) An efficient constraint handling method for genetic algorithms. *Computer Methods in Applied Mechanics and Engineering,* 186, 311-338.

DEB, K. (2001) *Multi-objective optimization using evolutionary algorithms,* Chichester ; New York, John Wiley & Sons.

DEB, K. & GOEL, T. (2001) A Hybrid Multi-objective Evolutionary Approach to Engineering Shape Design. *Evolutionary Multi-Criterion Optimization.*

DEB, K. & GULATI, S. (2001) Design of truss-structures for minimum weight using genetic algorithms. *Finite Elements in Analysis and Design,* 37, 447-465.

DIMOU, C. K. & KOUMOUSIS, V. K. (2003) Genetic algorithms in competitive environments. *Journal of Computing in Civil Engineering,* 17, 142-149.

DORIGO, M. & STÜTZLE, T. (2004) *Ant colony optimization,* Cambridge, Mass. ; London, MIT Press.

EBERHART, R. & KENNEDY, J. (1995) A new optimizer using particle swarm theory. *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science.* Nagoya, Japan, Ieee.

EDELSBRUNNER, H. (2001) *Geometry and topology for mesh generation,* Cambridge ; New York, Cambridge University Press.

EIBEN, A. E. & SMITH, J. E. (2003) *Introduction to evolutionary computing,* New York, Springer.

FOGEL, L. J., OWENS, A. J. & WALSH, M. J. (1966) *Artificial intelligence through simulated evolution,* New York, Wiley.

FOLEY, J. D. (1997) *Computer graphics : principles and practice,* Boston ; London, Addison-Wesley.

GOLDBERG, D. E. (1989) *Genetic algorithms in search, optimization, and machine learning,* Reading, Mass ; Wokingham, Addison-Wesley.

GOLDBERG, D. E. & SAMTANI, M. (1986) Engineering optimization via genetic algorithm. *The Ninth Conference on Electronic Computation.* University of Alabama, Birmingham.

GRIFFITHS, D. R. & MILES, J. C. (2003) Determining the optimal cross-section of beams. *Advanced Engineering Informatics,* 17, 59-76.

HAJELA, P. & LEE, E. (1995) Genetic Algorithms in Truss Topological Optimization. *International Journal of Solids and Structures,* 32, 3341-3357.

HAMDA, H., JOUVE, F., LUTTON, E., SCHOENAUER, M. & SEBAG, M. (2002a) Compact unstructured representations for evolutionary design. *Applied Intelligence,* 16, 139-155.

HAMDA, H., ROUDENKO, O. & SCHOENAUER, M. (2002b) Multi-objective evolutionary topological optimum design. IN PARMEE, I. C. (Ed.) *Proceedings of the Fifth International Conference on Adaptive Computing Design and Manufacture (ACDM 2002).* University of Exeter, Devon, UK.

HINTON, E. & SIENZ, J. (1995) Fully stressed topological design of structures using an evolutionary procedure. *Engineering computations,* 12, 229-244.

HOEFFLER, A., LEYSNER, U. & WEIDERMANN, J. (1973) Optimization of the layout of trusses combining strategies based on Mitchel's theorem and on biological principles of evolution. *Symposium on Structural Optimization.* Milan, Italy.

HOLLAND, J. H. (1975) *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence,* Ann Arbor, University of Michigan Press.

HOLLAND, J. H. (1986) *Induction : processes of inference, learning, and discovery,* Cambridge, Mass ; London, MIT Press.

HORNBY, G. S. (2003a) Generative representations for evolutionary design automation. Brandeis University.

HORNBY, G. S. (2003b) Generative representations for evolving families of designs. IN CANTUPAZ, E., FOSTER, J. A., DEB, K., DAVIS, L. D., ROY, R., OREILLY, U. M., BEYER, H. G., STANDISH, R., KENDALL, G., WILSON, S., HARTMAN, M., WEGENER, J., DASGUPTA, D., POTTER, M. A., SCHULTZ, A. C., DOWSLAND, K. A., JONOSKA, N. & MILLER, J. (Eds.) *5th Annual Genetic and Evolutionary Computation Conference (GECCO 2003).* Chicago, Illinois, Springer-Verlag Berlin.

HORNBY, G. S. (2004) Functional scalability through generative representations: the evolution of table designs. *Environment and Planning B-Planning & Design,* 31**,** 569-587.

HORNBY, G. S., LIPSON, H. & POLLACK, J. B. (2001) Generative representations for the automated design of modular physical robots. *IEEE International Conference on Robotics and Automation.* Seoul, South Korea, Ieee-Inst Electrical Electronics Engineers Inc.

HORNBY, G. S. & POLLACK, J. B. (2002) Creating high-level components with a generative representation for body-brain evolution. *Artificial Life,* 8**,** 223-246.

KAVEH, A., HASSANI, B., SHOJAEE, S. & TAVAKKOLI, S. M. (2008) Structural topology optimization using ant colony methodology. *Engineering Structures,* 30**,** 2559-2565.

KICINGER, R., ARCISZEWSKI, T. & DE JONG, K. (2005a) Generative Representations in Structural Engineering. *Computing in Civil Engineering 2005.* 40794 ed. Cancun, Mexico, ASCE.

KICINGER, R., ARCISZEWSKI, T. & JONG, K. D. (2005b) Evolutionary computation and structural design: A survey of the state-of-the-art. *Computers & Structures,* 83**,** 1943-1978.

KOUMOUSIS, V. K. & GEORGIOU, P. G. (1994) Genetic algorithms in discrete optimization of steel truss roofs. *Journal of Computing in Civil Engineering,* 8**,** 309-325.

KOZA, J. R. (1992) *Genetic programming : on the programming of computers by means of natural selection,* Cambridge, Mass ; London, MIT Press.

LAWO, M. & THIERAUF, G. (1982) Optimal design for dynamic stochastic loading: a solution by random search. *Optimization in Structural Design.* University of Siegen, FR Germany, Bibliographisches Institut Mannheim/Wien/Zürich, B. I. - Wissens chaftsverlag, Germany.

LINDENMAYER, A. (1968) Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of Theoretical Biology,* 18**,** 280-299.

LUH, G.-C. & LIN, C.-Y. (2009) Structural topology optimization using ant colony optimization algorithm. *Applied Soft Computing,* 9, 1343-1353.

MILES, J. C., SISK, G. M. & MOORE, C. J. (2001) The conceptual design of commercial buildings using a genetic algorithm. *Computers & Structures,* 79, 1583-1592.

MURAWSKI, K., ARCISZEWSKI, T. & DE JONG, K. (2000) Evolutionary Computation in Structural Design. *Engineering with Computers,* 16, 275-286.

OKABE, A. (2000) *Spatial tessellations : concepts and applications of Voronoi diagrams,* Chichester, Wiley.

OLHOFF, N., BENDSØE, M. P. & RASMUSSEN, J. (1991) On CAD-integrated structural topology and design optimization. *Computer Methods in Applied Mechanics and Engineering,* 89, 259-279.

PEREZ, R. E. & BEHDINAN, K. (2007) Particle swarm approach for structural design optimization. *Computers & Structures,* 85, 1579-1588.

PRUSINKIEWICZ, P. & LINDENMAYER, A. (1990) *The algorithmic beauty of plants,* New York ; London, Springer.

RAMASAMY, J. V. & RAJASEKARAN, S. (1996) Artificial neural network and genetic algorithm for the design optimizaton of industrial roofs --A comparison. *Computers & Structures,* 58, 747-755.

ROSENMAN, M. A. (1996) A growth model for form generation using a hierarchical evolutionary approach. *Microcomputers in Civil Engineering,* 11, 163-174.

ROSENMAN, M. A. (1997) The generation of form using an evolutionary approach. IN DASGUPTA, D. & MICHALEWICS, Z. (Eds.) *Evolutionary Algorithms in Engineering Applications.* Southampton, Springer-Verlag.

ROSENMAN, M. A. & GERO, J. (1999) Evolving designs by generating useful complex gene structures. IN BENTLEY, P. J. (Ed.) *Evolutionary Design by Computers.* San Francisco, Morgan Kaufmann.

ROZVANY, G. I. N. (1992) *Shape and layout optimization of structural systems and optimality criteria methods*, Springer.

RUSSELL, S. J., NORVIG, P. & CANNY, J. (2003) *Artificial intelligence : a modern approach,* Upper Saddle River, N.J., Prentice Hall.

SAKAMOTO, J. & ODA, J. (1993) Technique for optimal layout design for truss structures using genetic algorithms. *Proceedings of the 34th AIAA/ASCE/ASME/AHS Structural Dynamics and Material Conference AIAA/ASME Adaptive Structures Forum.* New York, NY.

SANDGREN, E., JENSEN, E. D. & WELTON, J. (1990) Topological design of structural components using genetic optimization methods. *Proceedings of the Winter Annual Meeting of the American Society of Mechanical Engineers, Sensitivity analysis and optimization with numerical methods.* Dallas.

SARMA, K. C. & ADELI, H. (2001) Bilevel parallel genetic algorithms for optimization of large steel structures. *Computer-Aided Civil and Infrastructure Engineering,* 16, 295-304.

SCHWEFEL, H.-P. (1997) Advantages and disadvantages of evolutionary computation over other approaches. IN BÄCK, T., FOGEL, D. B. & MICHALEWICZ, Z. (Eds.) *Handbook of Evolutionary Computation.* Institute of Physics Pub and Oxford University Press.

SHANKAR, N. & HAJELA, P. (1991) Heuristics driven strategies for near-optimal structural topology development. IN TOPPING, B. H. V. (Ed.) *Artificial intillegence structural engineering.* Civil-Comp Press, Oxford, UK.

SOH, C. K. & YANG, J. (1996) Fuzzy controlled genetic algorithm search for shape optimization. *Journal of Computing in Civil Engineering,* 10, 143-150.

STEVEN, G., QUERIN, O. & XIE, M. (2000) Evolutionary structural optimisation (ESO) for combined topology and size optimisation of discrete structures. *Computer Methods in Applied Mechanics and Engineering,* 188, 743-754.

SUZUKI, K. & KIKUCHI, N. (1991) A homogenization method for shape and topology optimization. *Computer Methods in Applied Mechanics and Engineering,* 93, 291-318.

TAI, K. & AKHTAR, S. (2005) Structural topology optimization using a genetic algorithm with a morphological geometric representation scheme. *Structural and Multidisciplinary Optimization,* 30, 113-127.

TAI, K. & CHEE, T. H. (2000) Design of structures and compliant mechanisms by evolutionary optimization of morphological representations of topology. *Journal of Mechanical Design,* 122, 560-566.

TAI, K., CUI, G. Y. & RAY, T. (2002) Design synthesis of path generating compliant mechanisms by evolutionary optimization of topology and shape. *Journal of Mechanical Design,* 124, 492-500.

TANSKANEN, P. (2002) The evolutionary structural optimization method: theoretical aspects. *Computer Methods in Applied Mechanics and Engineering,* 191, 5485-5498.

TOPPING, B. H. V. & LEITE, J. P. B. (1998) Parallel genetic models for structural optimization. *Engineering Optimization,* 31, 65-99.

WANG, S. Y. & TAI, K. (2005) Structural topology design optimization using Genetic Algorithms with a bit-array representation. *Computer Methods in Applied Mechanics and Engineering,* 194, 3749-3770.

XIE, Y. M. & STEVEN, G. P. (1993) A simple evolutionary procedure for structural optimization. *Computers & Structures,* 49, 885-896.

YANG, Y. & KIONG SOH, C. (2002) Automated optimum design of structures using genetic programming. *Computers & Structures,* 80, 1537-1546.

YANG, Y. W. & SOH, C. K. (2002) Automated optimum design of structures using genetic programming. *Computers & Structures,* 80, 1537-1546.

ZHANG, Y. (2004) Searching for optimal beam cross sections using a genetic algorithm. *School of Engineering.* Cardiff, UWC.

ZHANG, Y., MILES, J., KWAN, A. (2008) Using Generative Representations with Genetic Algorithms for Topological Search. IN PARMEE, I. (Ed.) *ACDM 2008.*

ZHANG, Y., WANG, K., SHAW, D., MILES, J., PARMEE, I., KWAN, A. (2006) Representation and its Impact on Topological Search in Evolutionary Computation. IN RIVARD, H., MIRESCO, E., MELHEM, H. (Ed.) *Joint International Conference on Computing and Decision Making in Civil and Building Engineering.* Montreal, Canada.