A Shortest Path Algorithm for Graphs Featuring Transfer Costs at their Vertices

R. Lewis

School of Mathematics, Cardiff University, CF24 4AG, Wales.
LewisR9@cf.ac.uk

Abstract. This paper examines the problem of finding shortest paths in graphs that feature additional penalties – transfer costs – at their vertices. We propose a shortest path algorithm that can cope with these additional penalties without the need of first performing a graph expansion, which is the typical algorithmic strategy. While our method exhibits an inferior growth rate compared to existing approaches, we show that it is more efficient on sparse graphs.

1 Introduction

Consider the graph in Figure 1. This might depict some small public transport system with edge colours representing transport lines and weights representing travel times. Now suppose that we want to find the shortest path from vertex v_1 to v_9. By inspection, this is (v_1, v_4, v_5, v_9) with a cost of $2+1+2=5$. However, this path involves changing lines at v_4 which, in reality, might also incur some time penalty. If this penalty is more than three units, then the shortest path from v_1 to v_9 now becomes $(v_1, v_4, v_7, v_8, v_9)$ with a cost of eight.

In this paper we propose a flexible model in which colours of edges are used to help specify transfer costs at vertices. Let $G = (V, E)$ be an edge-weighted, loop-free, directed multigraph using k different edge colours. As usual, V is a set of n vertices $\{v_1, \ldots, v_n\}$ and E is a set of m coloured, directed edges taken from the set of all such edges $\{(u, v, i) : u, v \in V \wedge u \neq v \wedge i \in \{1, \ldots, k\}\}$. Hence

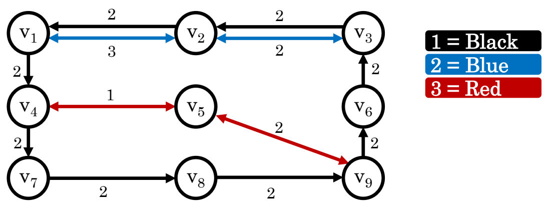

Fig. 1. A small network comprising $n = 9$ vertices, $m = 16$ edges and $k = 3$ colours. As per Definition 1 we have, for example, $E^-(v_1) = \{(v_2, v_1, 1), (v_2, v_1, 2)\}$, $E^+(v_1) = \{(v_1, v_2, 2), (v_1, v_4, 1)\}$, $C^-(v_1) = \{1, 2\}$ and $C^+(v_1) = \{1, 2\}$.

$0 \leq m \leq kn(n-1)$. The weight of an edge of colour i travelling from vertex u to vertex v is denoted by $w(u, v, i)$.

In this paper we also use the following notation, exemplified in Figure 1:

Definition 1. *Let $E^-(v) = \{(u, v, i) : (u, v, i) \in E\}$ be the set of edges whose endpoint is vertex v, and $C^-(v) = \{i : \exists (u, v, i) \in E^-(v)\}$ be the set of distinct colours that enter v. Similarly, $E^+(v) = \{(v, u, i) : (v, u, i) \in E\}$ is the set of edges whose starting point is v and $C^+(v)$ is the set of colours that leave v.*

Finally, we also need to define a set of *transfers* T. A transfer occurs when we arrive at a vertex v on an edge of colour i and leave v on an edge of colour j. Hence $T = \{(v, i, j) : v \in V \wedge i \in C^-(v) \wedge j \in C^+(v)\}$. The cost of a transfer is denoted by $t(v, i, j)$ and it is assumed that if $i = j$, then $t(v, i, j) = 0$.

Shortest path problems on graphs with transfer costs at the vertices have many practical uses. As noted, an obvious example is with public transport networks where additional costs (such as financial or time) can be incurred when switching between different lines. Another example is with the multi-modal shortest path problem where we are in interested in transporting goods efficiently between two locations using a combination of different travel modes (sea, train, road etc.), and where transfer costs represent the cost of moving the goods from one mode to another [5]. Constraints stemming from real-world road networks can also be defined using the above model by considering edges as roads and vertices as intersections. For example:

Intersection delays. Often vehicles will need to wait at an intersection due to crossing traffic and pedestrians. Such delays can be modelled using an appropriate transfer cost at the vertex.

Illegal routes and turns. On occasion, large vehicles will not be permitted to drive on particular roads or make particular turns at an intersection. In these cases we can simply change the corresponding edge weights and transfer costs to infinity.

Kerbside routing. Vehicles will often need to arrive at a location from a particular direction (e.g. if a road contains a central reservation and crossing is not permitted). In this case, the shortest path problem will be constrained so that we arrive at the destination vertex on a particular subset of edge colours. Note that this can result in shortest paths that contain cycles. For example, in Figure 1 the shortest v_1-v_4-path that also arrives on a red edge is (v_1, v_4, v_5, v_4).

Initial headway. Similarly to the previous point, vehicles may also need to leave a location in a particular direction (e.g. if they previously approached from a particular direction and turning is not possible). In this case, the shortest path should be specified as having to leave this vertex on a particular edge colour.

In this paper we propose a shortest path algorithm that accommodates transfer costs at the vertices and also allows us to evaluate paths in which the edge colours entering the source and target vertices are specified by the user. As we will see, previous methods for this problem rely on expanding graphs, resulting in many more edges and vertices. In contrast, our algorithm avoids this and, instead, operates on the original graphs without modification. The next section of this paper reviews the shortest path problem and surveys relevant expansion methods. Section 3 then presents our algorithm and proves its correctness, while Sections 4 and 5 examine and compare asymptotic and empirical run times. Section 7 concludes the paper.

2 Identifying Shortest Paths

In general, three problems involving shortest paths on edge-weighted graphs can be distinguished: (a) the "single-source single-target" problem, which involves finding the shortest path between two vertices; (b) the "single source" problem, which involves determining the shortest path from a source vertex to all other vertices in the graph (thereby producing a shortest path tree); and (c) the "all pairs" problem, where shortest paths are identified between all pairs of vertices. A well-known algorithm for solving problems (a) and (b) is the Bellman-Ford algorithm, which operates in $\mathcal{O}(nm)$ time [4]. This is suitable for graphs featuring both positive and negative edge weights and can also be used for detecting negative cycles.

If a graph contains only nonnegative edge weights, then a more efficient alternative is to use Dijkstra's algorithm [4]. The pseudocode of this method is given in Figure 2. As shown, DIJKSTRA uses four data structures, D, L, P and Q. The first three of these contain n elements and will typically allow direct access (e.g. by using arrays). Each entry $D(v)$ is used to mark whether a vertex v is classed as "distinguished" or not. Initially, only the source vertex s is distinguished. During the run, further vertices then become distinguished one by one, and this

```
        DIJKSTRA ($s \in V$)
(1)     for all $v \in V$ do
(2)         $L(v) \leftarrow \infty$,  $D(v) \leftarrow$ false,  $P(v) \leftarrow$ NULL
(3)     $L(s) \leftarrow 0$
(4)     $Q \leftarrow \{(s, L(s))\}$
(5)     while $Q \neq \emptyset$ do
(6)         Let $(u, L(u))$ be the element in $Q$ with minimum value for $L(u)$
(7)         $Q \leftarrow Q - \{(u, L(u))\}$
(8)         $D(u) \leftarrow$ true
(9)         for all $(u, v) \in E^+(u) : D(v) =$ false do
(10)            if $L(u) + w(u, v) < L(v)$ then
(11)                if $L(v) \neq \infty$ then $Q \leftarrow Q - \{(v, L(v))\}$
(12)                $L(v) \leftarrow L(u) + w(u, v)$
(13)                $Q \leftarrow Q \cup \{(v, L(v))\}$
(14)                $P(v) \leftarrow u$
```

Fig. 2. Dijkstra's algorithm for producing a shortest-path tree from a source vertex $s \in V$. To solve the single-source single-target problem, Line (5) should be replaced by the statement "while $D(t) \neq$ true do", where $t \in V$ is the target vertex.

continues until all vertices are marked as such. Meanwhile, L is used to hold a "label" for each vertex. During execution, $L(v)$ stores the length of the shortest s-v-path that uses distinguished vertices only; hence at the end of the run, $L(v)$ will store the length of the shortest s-v-path in the graph. P then allows us to identify the shortest paths themselves by storing the predecessor of each vertex v in the shortest path tree.

The final structure used in Dijkstra is a priority queue Q. During execution this holds the label values of all vertices that have been considered by the algorithm but that have not yet been marked as distinguished. As shown on Line (6), in each iteration Q is used to identify the undistinguished vertex u with the minimal label value. In the remaining instructions, u is then removed from Q and marked as distinguished, and adjustments are made to the labels of undistinguished neighbours of u, if applicable.

The asymptotic running time of Dijkstra depends mainly on the data structure used to represent Q. A good option is to use a binary heap or self-balancing binary tree since this allows identification of the minimum label in constant time, with look-ups, deletions, and insertions then being performed in logarithmic time. This leads to an overall run time of $\mathcal{O}((n+m)\lg n)$, which simplifies to $\mathcal{O}(m \lg n)$ for connected graphs (where $m \geq n$). Asymptotically, a further improvement to $\mathcal{O}(m + n \lg n)$ can also be achieved using a Fibonacci heap for Q, though such structures are often viewed as slow in practice due to their large memory consumption and the high constant factors contained in their operations [2].

Solving the all pairs shortest path problem involves populating a matrix $\mathbf{D}_{n\times n}$, where each element D_{ij} holds the length of the shortest v_i-v_j-path. A well known approach for this problem is the Floyd-Warshall algorithm, which operates in $\mathcal{O}(n^3)$ time [4]. Another alternative – which usually gives better performance with sparse graphs – is to simply perform n applications of Dijkstra, with each application populating a single row of $\mathbf{D}$.

Although the Bellman-Ford, Floyd-Warshall, and Dijkstra algorithms all correctly calculate shortest paths in edge-weighted graphs, note that they cannot be directly applied to graphs featuring transfer penalties at the vertices. Instead, graphs are typically *expanded* to allow transfer penalties to be expressed via additional "transfer edges". This then allows shortest path methods to be applied as before. The most prominent expansion method is that of Kirby and Potts [10] who suggest using a cluster of dummy vertices for each vertex in the original graph. Specifically, using a graph $G = (V, E)$ as defined in Section 1, a new larger graph $G' = (V', E')$ is formed by creating two sets of dummy vertices for each vertex $v \in V$: one for each incoming colour in v and one for each outgoing colour in v. Transfer edges are then added between the dummy vertices in each set using edge weights equivalent to the corresponding transfer costs.

An example of the Kirby-Potts expansion method is shown in Figure 3(a). As illustrated, the transfer edges within each cluster define a directed bipartite graph. This results in a new graph $G' = (V', E')$ comprising

$$n' = \sum_{i=1}^{n} |C^-(v_i)| + |C^+(v_i)| \tag{1}$$

$$m' = m + \sum_{i=1}^{n} |C^-(v_i)| \cdot |C^+(v_i)| \tag{2}$$

vertices and edges respectively. Note that a shortest path between two vertices in G' now also specifies the starting colour and arrival colour in the original graph's path. For example, the shortest path between the vertices marked by X and Y in Figure 3(a) corresponds to the shortest v_1-v_5-path in Figure 1 in which "arrival" at v_1 is assumed on a black edge and arrival at v_5 is on a red edge.

A similar but more restricted version of the Kirby-Potts expansion has also been used in various studies regarding small bus networks [1,3,8,7,9]. In this method each vertex v of the original graph is represented by a cluster of $|C^-(v) \cup C^+(v)|$ dummy vertices. Each vertex in this cluster then corresponds to a different colour, and edges are added between these vertices using weights equivalent to the corresponding transfer costs. However, although this restricted method can result in smaller graphs than those of Kirby-Potts, we have found that it can produce illogical results when the edge weights within a cluster do not obey the triangle inequality. Consider the Kirby-Potts expansion in Figure 3(b) for example, where a cost of 3 is incurred at the vertex when transferring from blue to black. In the corresponding graph produced using the restricted expansion method (3(c)) a smaller transfer cost of 2 will be identified by transferring from blue to red to black, which is clearly inappropriate when modelling things such as transfers on public transport. (This issue is not noted in any of the above works; however, it is actually avoided due to a constant value being used for all transfers, thereby satisfying the triangle inequality at each vertex.)

One further method of graph expansion is due to Winter [12] who suggests using the line graph of G (referred to as the "pseudo-dual" in the paper) for identifying shortest paths. However, this leads to a much larger graph comprising

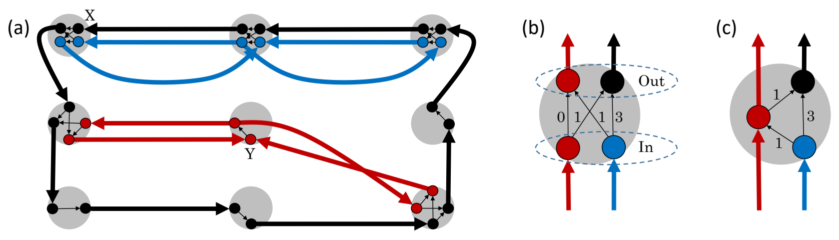

Fig. 3. (a) The graph $G' = (V', E')$ formed from Figure 1 using the Kirby-Potts expansion method; (b) an example cluster of dummy vertices produced using the Kirby-Potts method; and (c) the corresponding cluster using the restricted expansion method.

m vertices and $\sum_{v \in V} |E^-(v)| \cdot |E^+(v)|$ edges. A copy of the original graph G is also required with this method to facilitate the drawing of routes.

3 An Extension to Dijkstra's Algorithm

In this section we propose an extension to Dijkstra's algorithm that computes shortest paths in graphs featuring transfer costs at the vertices, but without the need for first performing an expansion.

The idea behind our proposed method can be explained by considering a cluster of dummy vertices in a Kirby-Potts expanded graph. As illustrated in Figures 3(a) and (b), we see that vertices in each cluster can be partitioned into two sets: in-vertices and out-vertices. Moreover, a shortest path from an in-vertex must always next pass through an out-vertex from the same cluster before moving to a different cluster. As proven in Theorem 1 below, it is therefore sufficient to simply add the cost of the corresponding transfer (edge) to the path here, rather than consider the out-vertices as separate entities within the graph.

The idea in our approach is to therefore use a pair (u, i) for each vertex $u \in V$ and incoming colour $i \in C^-(u)$, giving $\sum_{u \in V} |C^-(u)|$ pairs in total. The source is also defined by such a pair (s, l), which is interpreted as meaning that the paths should start at $s \in V$, assuming initial entry to s on an edge of colour l. Similarly to Dijkstra, during execution this algorithm stores labels, predecessors and the distinguished status of each pair using the structures L, P and D respectively. At termination, all pairs reachable from the source are marked as distinguished, and a label $L(u, i)$ holds the length of the shortest path from the source to vertex u, assuming entry at u on an edge of colour i.

The pseudocode of our algorithm is shown in Figure 4 and an example solution from this method is shown in Figure 5. As shown, the main differences between this approach and Dijkstra are (a) the use of vertex-colour pairs, and (b) at Lines 11 and 13, where transfer costs $t(u, i, j)$ are added when comparing and recalculating label values. Note also that for $k = 1$ this algorithm becomes equivalent to Dijkstra, justifying our choice of the name Extended-Dijkstra here.

The correctness of Extended-Dijkstra is due to the following.

Theorem 1. *If all edge weights and transfer costs in a graph are nonnegative then, for all distinguished pairs (u, i), the label $L(u, i)$ is the length of the shortest path from the source (s, l) to (u, i).*

Proof. Proof is by induction on the number of distinguished pairs. When there is just one distinguished pair, the theorem clearly holds since the length of the shortest path from (s, l) to itself is $L(s, l) = 0$.

For the step case, let (v, j) be the next pair to be marked as distinguished by the algorithm (i.e., $L(v, j)$ is minimal among all undistinguished pairs) and let (u, i) be its predecessor. Hence the shortest (s, l)-(v, j)-path has length $L(u, i) + t(u, i, j) + w(u, v, j)$. Now consider any other path P from (s, l) to (v, j). We need to show that the length of P cannot be less than $L(u, i) + t(u, i, j) +$

$w(u,v,j)$. Let (x,a) and (y,b) be pairs on P such that (x,a) is distinguished and (y,b) is not, meaning that P contains the edge (x,y,b). This implies that the length of P is greater than or equal to $L(x,a)+t(x,a,b)+w(x,y,b)$ (due to the induction hypothesis). Similarly, this figure must be greater than or equal to $L(u,i)+t(u,i,j)+w(u,v,j)$ because, as assumed, $L(v,j)$ is minimal among all undistinguished pairs.

4 Asymptotic Analysis

In this section we consider the asymptotic complexity of Extended-Dijkstra and compare it to the process of using Dijkstra on graphs that have already been expanded using the Kirby-Potts method. As we might expect, the expense of both of these approaches increases for larger numbers of vertices and edges. In addition, they are also affected by the number of colours k used in the graph, though we avoid this variable in our analysis because it can lead to an overes-

```
       EXTENDED-DIJKSTRA ($s \in V$, $l \in C^-(s)$)
(1)    for all $v \in V$ do
(2)        for all $i \in C^-(v)$ do
(3)            $L(v,i) \leftarrow \infty$,  $D(v,i) \leftarrow$ false,  $P(v,i) \leftarrow$ NULL
(4)    $L(s,l) \leftarrow 0$
(5)    $Q \leftarrow \{(s,l,L(s,l))\}$
(6)    while $Q \neq \emptyset$ do
(7)        Let $(u,i,L(u,i))$ be the element in $Q$ with minimum value for $L(u,i)$
(8)        $Q \leftarrow Q - \{(u,i,L(u,i))\}$
(9)        $D(u,i) \leftarrow$ true
(10)       for all $(u,v,j) \in E^+(u) : D(v,j) =$ false
(11)           if $L(u,i)+t(u,i,j)+w(u,v,j) < L(v,j)$ then
(12)               if $L(v,j) \neq \infty$ then $Q \leftarrow Q - \{(v,j,L(v,j))\}$
(13)               $L(v,j) \leftarrow L(u,i)+t(u,i,j)+w(u,v,j)$
(14)               $Q \leftarrow Q \cup \{(v,j,L(v,j))\}$
(15)               $P(v,j) \leftarrow (u,i)$
```

Fig. 4. The Extended Dijkstra algorithm.

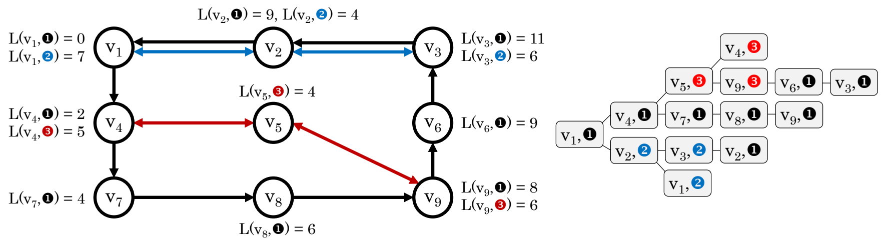

Fig. 5. Solution returned by Extended-Dijkstra using the graph from Figure 1 and source $(v_1, 1)$. All transfer costs are assumed to be 1. The shortest path tree defined by the contents of P is shown on the right.

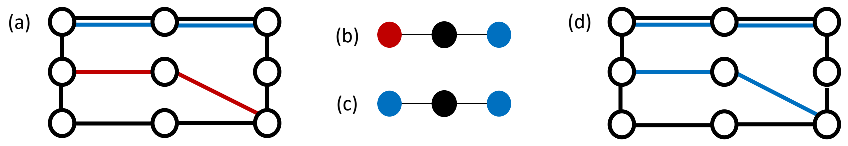

Fig. 6. (a) A graph G^* using $k = 3$ colours; (b) the underlying conflicts graph (and 3-colouring) prescribed by G^*; (c) a 2-colouring of this conflicts graph; and (d) the original graph G^* using $k = 2$ colours.

timation of complexity due to a relationship with the graph colouring problem, as we now explain.

Let $G = (V, E)$ be a graph using k colours as defined in Section 1, and let $G^* = (V, E^*)$ be a copy of G with all edge directions removed. Finally let $G^*(i)$ denote the subgraph formed from G^* using edges of colour i only. Note that if two such subgraphs $G^*(i)$ and $G^*(j)$ have no common vertices, then no transfers are possible between colours i and j. In this case we have the opportunity to relabel all i-coloured edges with colour j (or vice versa), and potentially reduce the number of colours being used in the graph.

In more detail, consider a conflicts graph created using an i-coloured vertex for each component of each subgraph $G^*(i)$ (for $i \in \{1, \ldots, k\}$), with edges corresponding to any vertex pair representing differently-coloured overlapping components in G^*. Note that the colours of the vertices in this conflicts graph define a proper k-colouring, in that pairs of adjacent vertices always have different colours; however, it may be possible to colour this conflicts graph using fewer colours. If this is so, then an equivalent graph to G with fewer colours can also be created. An example of this process is shown in Figure 6. This illustrates that, while the number of edge colours k could have any value up to and including m, the minimum number of colours needed to express this graph might well be smaller. However, identifying this minimum can be difficult since it is equivalent to solving the $\mathcal{NP}$-hard chromatic number (graph colouring) problem [11].

Given these observations on k, a better alternative for analysing complexity is to consider the number of colours entering and exiting each vertex (given by $C^-(v)$ and $C^+(v)$) and, in particular, their maximum values $c^-_{\max} = \max\{|C^-(v)| : v \in V\}$ and $c^+_{\max} = \max\{|C^+(v)| : v \in V\}$.

Now reconsider the pseudocode for EXTENDED-DIJKSTRA given in Figure 4. As before, we assume the use of a binary heap for Q and direct access data structures for L, D and P. The initialisation of L, D, and P in Lines 1 to 5 has a worst-case complexity of $\mathcal{O}(nc^-_{\max})$. For the main part of the algorithm, now note that each label in L is considered and marked as distinguished exactly once and, in the worst case, we will have $nc^-_{\max}$ such labels. Once a label (u, i) is marked as distinguished, all incident edges (u, v, j) are then considered in turn and are subject to a series of constant-time and log-time operations, as shown in Lines 12 to 15. This leads to a overall worst case complexity of $\mathcal{O}\left(nc^-_{\max}\right) + \mathcal{O}\left((mc^-_{\max}) \lg\left(nc^-_{\max}\right)\right)$. Assuming graph connectivity, this simplifies to a growth

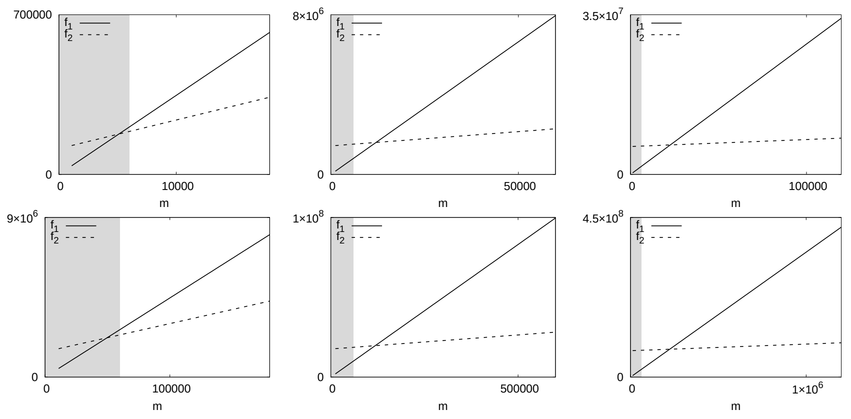

Fig. 7. Comparison of growth rates f_1 and f_2 with regards to the number of edges m. Rows show n-values of 1000 and 10000 respectively; columns consider values $c^-_{\max} = c^+_{\max}$ of 3, 10, and 20 respectively.

rate for Extended-Dijkstra of

$$f_1 = \mathcal{O}\left(\left(mc^-_{\max}\right)\lg\left(nc^-_{\max}\right)\right). \tag{3}$$

As noted, the complexity of Dijkstra using a binary heap is $\mathcal{O}(m \lg n)$. Using a Kirby-Potts expansion, in the worst case this leads to a graph G' with $n' = n(c^-_{\max} + c^+_{\max})$ vertices and $m' = m + nc^-_{\max}c^+_{\max}$ edges, giving an overall complexity of $\mathcal{O}(m' \lg n')$, or

$$f_2 = \mathcal{O}\left(\left(m + nc^-_{\max}c^+_{\max}\right)\lg\left(n\left(c^-_{\max} + c^+_{\max}\right)\right)\right). \tag{4}$$

Figure 7 compares f_1 and f_2 for a range of different parameter values. Note that f_1 grows more quickly in all cases, demonstrating that Extended-Dijkstra is less efficient with regards to increases in the number of edges m. The main reason for this is that, with Extended-Dijkstra, each outgoing edge of a vertex v is considered for each incoming colour of v. This results in the term $(mc^-_{\max})$ seen in Equation (3). In contrast, although a Kirby-Potts expansion results in a graph G' with an increased number of edges and vertices, each edge in G' is considered only once using Dijkstra, which results in a slower growth rate overall. Note, however, that each chart in Figure 7 features an intercept, suggesting that Extended-Dijkstra is more efficient with very sparse graphs. For indicative purposes, the grey rectangles in the figure show the range of values for which planar digraphs exist (i.e., the right boundary of these rectangles occur at $m = 2(3n - 6)$, which is the maximum possible number of directed edges in a planar digraph). Planar graphs are considered further in the next section.

5 Computational Comparison

We now consider the CPU times required to calculate shortest path trees on the edge-coloured graphs defined in Section 1. In our case we will seek shortest paths in which transfer costs are not incurred at terminal vertices. This is useful in applications such as public transport, where a passenger will arrive at the source vertex by means outside of the network (e.g. by foot), and will then leave the network on arrival at their destination. To make this modification with Extended-Dijkstra we can simply set all transfer costs at the source vertex s to zero before running the algorithm using an arbitrary in-colour $l \in C^-(s)$. The shortest s-v-path length in G is then indicated by the minimum value among the labels $L(v, i)$, where $i \in C^+(v)$. For a Kirby-Potts expanded graph G', a similar process is used: first, the weights of all transfer edges in the cluster defined by s are temporarily set to zero; next Dijkstra is executed from an arbitrary in-vertex within this cluster; finally, the minimum label value among all in-vertices in v's cluster is identified.

Two types of problem instances were considered in our tests. These were generated using a density parameter d that represents the average number of edges travelling from each vertex u to each vertex v. The first instance type, random graphs, were generated by randomly placing n vertices into the unit square. All potential edges (u, v, i) were then considered in turn and added to the graph with a fixed probability of d/k. During this process, care was also taken to ensure that the graph contained a random $(n-1)$-cycle, making the graph strongly connected.

The second graph type, planar graphs, were considered to give an indication of algorithm performance on transport networks. Recall that planar graphs are those that can be drawn on a plane so that no edges cross. In that sense, like road networks, they are quite sparse, with vertex degrees being fairly low. Note that when things like roads physically intersect on land, there will often be an opportunity to transfer from one to the other; hence, the underlying graph will be planar. However, this is not always the case, such as when one road crosses another via a bridge, so the analogy is not exact. Planar graphs were formed by again randomly placing n vertices into the unit square. A Delaunay triangulation was then generated from these vertices, with the edges of this triangulation being used to form a pool of potential edges for the graph (that is, for each edge $\{u, v\}$ in the triangulation, all directed and coloured edges (u, v, i) and (v, u, i) (for $i \in \{1, \ldots, k\}$) were added to the pool). Edges were then selected randomly from this pool and added to the graph until the desired graph density was reached. Again, we also ensured that the resultant graph was strongly connected: in this case by including all edges from a bidirectional minimum spanning tree.

For both random and planar graphs, edge weights were calculated using the Euclidean distances between vertices plus or minus x percent where, for each edge, x was selected randomly in the range $(-10, 10)$. This prevents edges between the same pair of vertices from having the same weight. Transfer costs were set to the average edge weight across the graph plus or minus x percent.

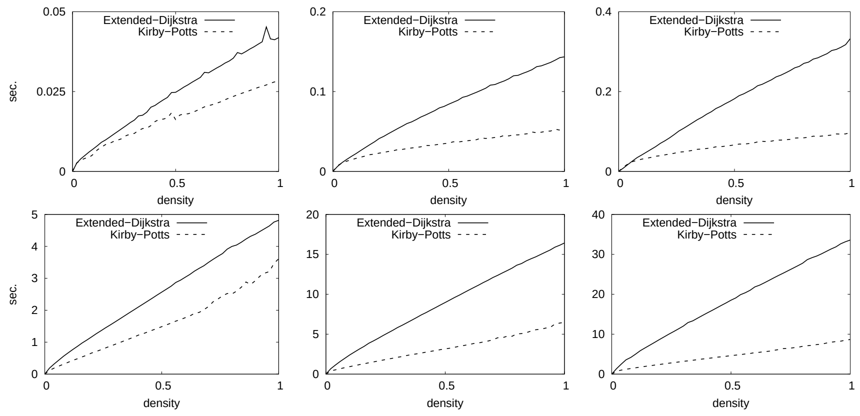

Fig. 8. CPU times required to produce a shortest path tree from a single source using random graphs of differing densities. Rows show n values of 1000 and 10000 respectively; columns consider k-values of 3, 10 and 20 respectively. The number of edges in a graph is determined by multiplying the density by $n(n-1)$.

All algorithms used in our tests were written in C++ and executed on 3.2 GHtz Windows 7 machines with 8 GB RAM. Our implementations used red-black trees for the priority queues Q and adjacency lists for storing edges, colours and weights.

Figure 8 shows the average CPU times required by Extended-Dijkstra for random graphs with $n \in \{1000, 10000\}$ and $k \in \{3, 10, 20\}$. In all cases, five graphs were generated for each density $d \in \{0, 0.02, 0.04, \ldots, 1.0\}$. Extended-Dijkstra was then run using each of the n vertices as a source in turn. Each point in the figure is therefore a mean of $5n$ different values. The figures also show the times required by Dijkstra on the corresponding Kirby-Potts expanded graphs. Note that the cost of performing the expansions is not included in these figures.

As expected, Figure 8 shows that the run times of both algorithms grow for increases in n, k, and m. We also see that the Kirby-Potts method shows more favourable run times overall, particularly for large dense problem instances. Indeed, the most extreme difference occurs with graphs with $n = 10000$, $d = 1$, and $k = 20$, where an average difference of over twenty seconds per run is observed.

To contrast these results, Figure 9 shows the average CPU times required for planar graphs. Here, for each n and k, graphs were generated for 25 different values for m, using an upper limit of $m = 2k(3n - 6)$ (this is the maximum number of edges in a planar, loop-free, multi-digraph using k edge colours). As before, the right boundaries of the grey rectangles in these charts indicate $m = 2(3n - 6)$ (the maximum number of edges in a planar digraph). All other details are the same as the previous experiments.

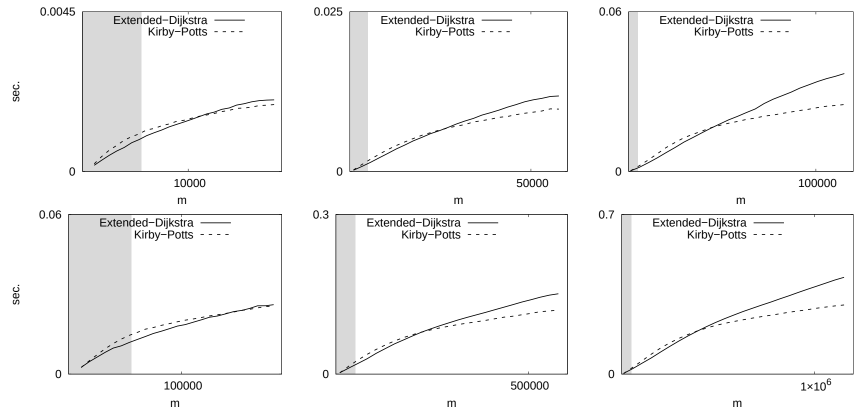

Fig. 9. CPU times required to produce a shortest path tree from a single source using planar graphs of differing densities. Rows show n values of 1000 and 10000 respectively; columns consider k-values of 3, 10 and 20 respectively.

Figure 9 reveals similar patterns to our analysis in Section 4. For sparse graphs, including all of those within the grey rectangles, EXTENDED-DIJKSTRA requires less run time. However, as previously seen in Figure 7, these two lines eventually intersect, with EXTENDED-DIJKSTRA then requiring more run time for denser graphs.

6 Expansion Times

In the results of the previous section we have chosen not to include the time taken to perform Kirby-Watts expansions. This is because the decision on how a graph is represented and stored will often be made by the user beforehand and might therefore be considered a separate process. However, this will not always be the case. For example, Ahmed et al. [1] and Heyken Soares et al. [7] have proposed methods for optimising public transport systems that use heuristics to produce a whole series of graphs, each that is then expanded before evaluation. In these cases, it is therefore appropriate to consider the overheads consumed by these expansions.

In our case, an expansion takes a graph G and produces a corresponding adjacency list for the expanded graph G'. This involves stepping through each edge and each transfer cost in G and then adding an appropriate edge to G', leading to an overall complexity of $\mathcal{O}(m + nc^{-}_{\max}c^{+}_{\max})$. Note that this has a slightly lower growth rate compared to executing DIJKSTRA on G', shown in Equation (4).

Table 1 shows the conversion times for random and planar graphs of differing parameters. We see that these times increase for graphs featuring more edges, vertices, and colours, as we would expect. For graphs with $n = 1000$, conversion

Random Graphs			Planar Graphs		
	Conversion Time			Conversion Time	
Parameters	Mean	Std. dev.	Parameters	Mean	Std. dev.
$n = 1000$:			$n = 1000$:		
$d = 0.05$, $k = 3$	0.006	< 0.001	$m = 1998$, $k = 3$	< 0.001	0.001
$d = 0.50$, $k = 3$	0.058	0.002	$m = 9661$, $k = 3$	0.003	0.001
$d = 0.95$, $k = 3$	0.113	0.001	$m = 17325$, $k = 3$	0.003	< 0.001
$d = 0.05$, $k = 10$	0.017	< 0.001	$m = 1998$, $k = 10$	0.001	< 0.001
$d = 0.50$, $k = 10$	0.063	0.001	$m = 29781$, $k = 10$	0.016	0.002
$d = 0.95$, $k = 10$	0.113	0.001	$m = 57564$, $k = 10$	0.017	0.001
$d = 0.05$, $k = 20$	0.055	0.001	$m = 1998$, $k = 20$	0.001	0.001
$d = 0.50$, $k = 20$	0.108	0.002	$m = 58523$, $k = 20$	0.059	0.002
$d = 0.95$, $k = 20$	0.162	0.016	$m = 115049$, $k = 20$	0.068	0.004
$n = 10000$:			$n = 10000$:		
$d = 0.05$, $k = 3$	0.602	0.006	$m = 19998$, $k = 3$	0.008	< 0.001
$d = 0.50$, $k = 3$	6.640	0.161	$m = 96781$, $k = 3$	0.028	0.001
$d = 0.95$, $k = 3$	38.369	6.452	$m = 173565$, $k = 3$	0.034	0.001
$d = 0.05$, $k = 10$	0.714	0.004	$m = 19998$, $k = 10$	0.009	0.001
$d = 0.50$, $k = 10$	6.550	0.229	$m = 298341$, $k = 10$	0.203	0.006
$d = 0.95$, $k = 10$	44.437	8.754	$m = 576684$, $k = 10$	0.232	0.008
$d = 0.05$, $k = 20$	1.193	0.014	$m = 19998$, $k = 20$	0.010	0.001
$d = 0.50$, $k = 20$	6.768	0.092	$m = 586283$, $k = 20$	0.689	0.002
$d = 0.95$, $k = 20$	53.360	7.904	$m = 1152569$, $k = 20$	0.770	0.016

Table 1. Number of seconds to perform a Kirby-Potts expansion for random and planar graphs of differing parameters. Figures show the mean and standard deviation across twenty problem instances.

times are very small, coming in at less than 0.2 seconds in all cases; however, for larger graphs much longer times are sometimes required. Note that the largest Kirby-Watts graphs seen here (produced from random graphs with $n = 10000$, $d = 0.95$ and $k = 20$) required over 5 GB of RAM, so significant amounts of memory management (such as paging) may also be needed as part of this expansion process.

7 Conclusions

In this paper we proposed a new shortest path algorithm that copes with vertex transfer costs without having to first perform a graph expansion, a process that can sometimes be quite costly with large graphs. While our method exhibits an inferior growth rate compared to using Dijkstra's algorithm on Kirby-Potts expanded graphs, we have seen that it can exhibit shorter run times with sparse problem instances such as planar graphs.

In the future it would be useful to see if Extended-Dijkstra might also be converted into a modified version of the A* algorithm, perhaps giving superior performance with the single-source single-target shortest path problem. This would involve modifying Line 7 of Figure 4 so that a heuristic rule is used for selecting the pair (u, i). Properties of suitable heuristics are outlined in the work of Hart et al. [6].

References

1. L. Ahmed, C. Mumford, and A. Kheiri. Soving urban transit route design problems using selection hyper-heuristics. *European Journal of Operational Research*, 274:545–559, 2019.
2. R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. In C. McGeoch, editor, *Experimental Algorithms*, volume 5038 of *Lecture Notes in Computer Science*, pages 303–318. Springer, Berlin, Heidelberg, 2008.
3. I. Cooper, M. John, R. Lewis, C. Mumford, and A. Olden. Optimising large scale public transport network design problems using mixed-mode parallel multi-objective evolutionary algorithms. In *Proceedings of the 2014 IEEE Congress on Evolutionary Computation*, pages 2841–2848. IEEE, 2014.
4. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 3rd edition, 2009.
5. E. Demir, W. Burgholzer, M. Hrusovsky, E. Arikan, W. Jammernegg, and T. Van Woensel. A green intermodal service network design problem with travel time uncertainty. *Transportation Research Part B*, 93:789–807, 2016.
6. P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.
7. P. Heyken Soares, C. Mumford, K. Amponsah, and Y. Mao. An adaptive scaled network for public transport route optimisation. *Public Transport*, 11:379–412, 2019.
8. M. John. *Metaheuristics for designing efficient routes and schedules for urban transportation networks.* PhD thesis, School of Computer Science and Informatics, Cardiff University, 2016.
9. M. John, C. Mumford, and R. Lewis. An improved multi-objective algorithm for the urban transit routing problem. In *Evolutionary Computation in Combinatorial Optimisation*, volume 8600 of *Lecture Notes in Computer Science*, pages 49–60. Springer-Verlag, 2014.
10. R. Kirby and R. Potts. The minimum route problem for networks with turn penalties and prohibitions. *Transportation Research*, 3:397–408, 1969.
11. R. Lewis. *A Guide to Graph Colouring: Algorithms and Applications.* Springer International Publishing, 2016.
12. S. Winter. Modeling costs of turns in route planning. *GeoInformatica*, 6(4):345–361, 2002.