Methods for Finding Paths of a Prescribed Length in Weighted Graphs

Daniel Hambly[1], Rhyd Lewis[1], and Padraig Corcoran[2]

[1] School of Mathematics, Cardiff University, CF24 4AG, Wales
{HamblyDJ,LewisR9}@cardiff.ac.uk
[2] School of Computer Science and Informatics, Cardiff University, CF24 4AG, Wales
CorcoranP@cardiff.ac.uk

Abstract. This paper tackles the $\mathcal{NP}$-hard problem of finding paths of a prescribed length between a source and a target node in weighted directed and undirected graphs. To address this problem, we propose and analyse two algorithms: a heuristic method based on local search, and an exact backtracking algorithm that utilises several problem-specific operators to improve runtimes. We also present several general methods for reducing problem size. These involve removing nodes that can never be in any path from the source to the target, smoothing nodes that can be replaced with an arc, and removing nodes whose shortest path sum from the source and the target exceeds the prescribed length. Analysis on real-world street networks shows that the local search algorithm achieves solutions within a few metres of the prescribed length and consistently yields lower costs than the backtracking algorithm, even under a more restricted time limit. In contrast, some of our backtracking operators achieve lower-cost solutions on random and dense planar graphs.

Keywords: Paths · Problem reduction · Local search · Backtracking

1 Introduction

In this paper, we consider the combinatorial optimisation problem of identifying a path between two points in an arc-weighted graph, where the path length, defined as the sum of its arc weights, is as close as possible to a user-specified value k. This problem has practical applications when planning exercise routes on real-world street networks, where intersections and dead ends correspond to nodes, streets correspond to arcs, and the relevant unit of measurement (e.g., steps, distance, time, etc) corresponds to the arc weights. Existing routing services (such as Google Maps) can compute paths between two points in a street network, though they typically optimise for minimum length or travel time [22]. As such, they are not suitable for, say, a cyclist seeking a route between two points of a specified distance or time. The problem of finding paths of a prescribed length also arises in computational biology; for example, in identifying interaction pathways between proteins [18,27].

We begin by reviewing definitions and concepts relevant for this work:

Definition 1. *In a directed graph $G = (V, A)$, let V denote the set of nodes and A the set of arcs.*

In this paper, we consider directed graphs since they have immediate applications in street networks that feature one-way systems. This is not restrictive, however, because, for all problems considered, an undirected graph $G = (V, E)$ can be converted into an equivalent directed graph by assuming $A = \{(u, v), (v, u) : \forall \{u, v\} \in E\}$. We also use the following terminology:

Definition 2. *In a graph, a* walk *is a sequence of nodes and arcs; a* trail *is a walk with no repeated arcs; and a* path *is a trail with no repeated nodes.*

To indicate that a walk/trail/path begins at a node u and ends at a node v, we define it as a u-v walk/trail/path. When the start and end nodes are the same, i.e., $u = v$, we use the following:

Definition 3. *A u-v walk/trail/path is said to be* closed *if $u = v$. A closed trail is known as a* circuit*, and a closed path as a* cycle.

In this context, the terms *u-circuit* and *u-cycle* refer to circuits and cycles that start and end at u. We now formally define the main problem considered in this paper:

Definition 4. *Let $G = (V, A)$ be a directed arc-weighted graph with n nodes and m arcs. Let $k \geq 0$ be a user-prescribed path length, $w(u, v)$ be a non-negative weight for each arc $(u, v) \in A$, and s and t be defined as the source and target nodes, respectively. The* k-length path problem (KPP) *involves finding an s-t path $P = (s = v_1, v_2, \ldots, v_l = t)$ such that the total path length $L(P) = \sum_{i=1}^{l-1} w(v_i, v_{i+1})$ minimises the absolute difference $|k - L(P)|$.*

This definition is useful because it encompasses several different problems involving paths. Setting $k = 0$ corresponds to the polynomially solvable shortest s-t path problem, while choosing $k \geq \sum_{(u,v) \in A} w(u, v)$ corresponds to the $\mathcal{NP}$-hard longest s-t path problem. The KPP is also applicable to unweighted graphs simply by assuming $w(u, v) = 1$ for all $(u, v) \in A$. It can also be used to identify other structures of a prescribed length. These include

- Any path starting at node s. To do this, we simply introduce a dummy node t and add arcs (u, t) with weight $w(u, t) = 0$ for all $u \in V$. Then, finding a k-length s-t path is equivalent to finding any k-length path that starts at s.
- Any path ending at node t. To do this, we introduce a dummy node s and add arcs (s, u) with weight $w(s, u) = 0$ for all $u \in V$. Then, finding a k-length s-t path is equivalent to finding any k-length path that ends at t.
- Any path in the graph. To do this, we introduce dummy nodes s and t, and add arcs (s, u) with weights $w(s, u) = 0$ and (u, t) with weights $w(u, t) = 0$ for all $u \in V$. Then, finding a k-length s-t path is equivalent to finding any k-length path in the graph.
- s-cycles. To do this, we introduce a dummy node s' and add arcs (u, s') with weight $w(u, s)$ for all $(u, s) \in A$. Then, finding a k-length s-s' path is equivalent to finding a k-length s-cycle.

Together, these problems have a wide range of applications. For example, k-length cycles are used in designing exercise routes [34,22,23] and in network visualisation [31]. Shortest paths play a role in social networks, where nodes represent people, arcs represent relationships, and the shortest path corresponds to the minimum number of arcs to connect two individuals. They are also essential for ensuring collision-free navigation of unmanned aerial vehicles [1]. Longest paths arise in business contexts, where the total completion time of a set of jobs is equal to the longest path in a graph whose nodes represent jobs and arc weights represent processing times [3]. Additional applications can be found in peer-to-peer networks [36] and web-service selection [14].

The remainder of the paper is organised as follows: Section 2 presents a review of the complexity, theoretical results, and bounds related to path-finding and related problems, along with solution strategies for the KPP. Section 3 explores various techniques aimed at removing nodes that cannot be in any s-t path, reducing the problem size. In Section 4, we present two methods designed to address the KPP, while Section 5 evaluates the performance of these algorithms through empirical analysis. Finally, Section 6 concludes the paper and outlines potential directions for future research.

2 Literature Review

To date, limited research has been carried out on the problem of finding s-t paths of a prescribed length in arc-weighted graphs. However, several complexity results are known for related problems. The shortest path problem has been extensively studied and can be solved in polynomial time using a range of classical algorithms, including Dijkstra's algorithm and the Bellman-Ford algorithm [29]. In contrast, the longest s-t path problem is significantly more challenging: it is known to be $\mathcal{NP}$-hard, though it admits fixed-parameter tractable solutions on unweighted graphs [6], and polynomial-time solutions in the special case of directed acyclic graphs (DAGs) [28]. As seen in the previous section, the KPP is a generalisation of the longest path problem and is therefore also $\mathcal{NP}$-hard; however, similarly to the longest path problem, polynomial-time solutions exist for the KPP on trees and DAGs [17]. In terms of counting, the problem of enumerating all k-length paths is $\#\mathcal{W}[1]$-complete [11], while counting the total number of s-t paths is known to be $\#\mathcal{P}$-complete [33].

Basagni et al. [4] have demonstrated that determining the existence of a k-length *walk* in both directed and undirected arc-weighted graphs is $\mathcal{NP}$-complete. However, in unweighted graphs where $k = n^{\mathcal{O}(1)}$, the problem is solvable in polynomial time. The number of k-length walks in unweighted graphs can be computed by raising its binary adjacency matrix A to the kth power, yielding a matrix A^k [20]. Additionally, Eppstein [10] has adapted Yen's algorithm – originally designed to compute the K shortest s-t paths in a graph – to compute the K shortest *trails*, achieving a time complexity of $\mathcal{O}(m + n \log n + K)$. In other related work, the problem of identifying a k-length *circuit* has also been shown to be $\mathcal{NP}$-hard, as it generalises the $\mathcal{NP}$-hard problem of finding the largest Eulerian subgraph in a directed arc-weighted graph [12]. Similarly, identifying a k-length *cycle* is also $\mathcal{NP}$-hard, since it extends the $\mathcal{NP}$-hard problem of determining the longest cycle in a graph [15,24].

Certain bounds on path length can be computed with relative ease. A straightforward lower bound is the length of the shortest s-t path, which, as noted, can be solved in polynomial time. For upper bounds, one trivial option is the sum of the $n-1$ heaviest arcs in the graph. Let $a_1, a_2, \ldots, a_{n-1}$ denote these arcs, ordered by decreasing weight. Then the bound is given by $\sum_{i=1}^{n-1} w(a_i)$. Another simple upper bound is based on summing the heaviest outgoing arc from each node. Using $\tau(v)$ to denote the maximum weight leaving a node v, we get $\sum_{v \in V} \tau(v)$.

In terms of algorithms in this area, Monien [25] showed how any path of length k can be detected in unweighted graphs by constructing a matrix $M^{(k)}(G)$, where each entry $m_{ij}^{(k)}$ for $i, j \in V$ is equal to k if an i-j path of length k exists, or λ otherwise. However, constructing the matrix takes exponential time. Bodlaender [6] then demonstrated how the existence of any path of length k can be decided in $\mathcal{O}(2^k k! n)$ time, and the existence of an s-t path of length k in $\mathcal{O}(2^{2k}(2k)!n + m)$ time. It was originally conjectured in [26] that paths of length $\log(n)$ could be found in polynomial time. This was later confirmed by Alon et al. [2] for unweighted graphs, who introduced the colour-coding technique to detect k-length paths in $\mathcal{O}(2^k m \log n)$ time for directed graphs and $\mathcal{O}(2^k n \log n)$ time for undirected graphs.

In recent years, a variety of algorithms have been proposed for finding k-length s-t paths in unweighted graphs. Building on the colour-coding method, Kneis et al. [19] introduced the ″divide-and-colour″ approximation algorithm, which combines colour-coding with divide-and-conquer. This algorithm achieves an exponentially small error probability and runs in $\mathcal{O}(3^{\log(k)} 4^k)$ time. It was also derandomised to yield a deterministic algorithm with time complexity $\mathcal{O}^*(16^k)$. Shortly after, Chen et al. [8] proposed an approximation algorithm with running time $\mathcal{O}(4^k k^{3.42} m)$ and a deterministic algorithm of $\mathcal{O}(4^{k+o(k)})$. Both algorithms again use divide-and-colour. For directed graphs, Koutis [21] presented an approximation algorithm with $\mathcal{O}^*(2.83^k)$ running time, which was then improved to $\mathcal{O}^*(2^k)$ [35]. In undirected graphs, Björklund et al. [5] developed an approximation algorithm with time complexity $\mathcal{O}^*(1.66^k)$. Fomin et al. [13] also developed a deterministic algorithm running in $\mathcal{O}(2.619^k n \log n)$ time on directed graphs and $\mathcal{O}(2.619^k m \log n)$ time on undirected graphs. Subsequently, Zehavi [37] applied the divide-and-colour technique to design a deterministic algorithm for directed graphs with running time $\mathcal{O}^*(2.597^k)$, which was later improved to $\mathcal{O}^*(2.554^k)$ [32].

Most recently, Hambly et al. [16] investigated three exact methods for tackling the KPP on undirected edge-weighted graphs: an integer programming formulation, a backtracking algorithm (described and extended upon in Subsection 4.2), and a variant of Yen's algorithm, modified to terminate once a path of length greater than or equal to k is found.

From the above review, it is notable that existing approaches in this area are concerned with graphs that are undirected and/or unweighted and, as such, do not fit the KPP from Definition 4. In this paper, we suggest two methods to quickly find high-quality solutions for the KPP. One of these is a local search heuristic that can accept worsening solutions, helping to explore diverse regions of the solution space. The second approach is exact, though efforts are made to improve run times by guiding the search to more promising regions of the solution space. We also describe a number of techniques that can be used to reduce the problem size. More details are provided in the next section.

3 Problem Reduction Techniques

In this section, we propose several preprocessing methods that can reduce the size of the graph. Two of the methods aim to remove nodes that cannot appear in any s-t path, and to smooth nodes without altering the overall structure of the graph. In both cases, such modifications maintain the set of all possible s-t paths. These methods can be used for any of the problem variants listed in Section 1. In contrast, the final method introduces another technique for removing nodes, albeit with the risk of eliminating good-quality solutions.

Our first method for reducing problem size involves removing *redundant* nodes from the graph, where a node is defined as redundant if and only if it cannot occur in any s-t path. In an undirected graph $G = (V, E)$, we can confirm whether a particular node u is redundant by introducing a dummy node d, together with zero-weighted edges $\{d, s\}$ and $\{d, t\}$. Then, u is not redundant if and only if there exist at least two node-disjoint d-u paths. This can be verified in $\mathcal{O}(m + n \log n)$ time using the method of Suurballe [30]. This procedure then needs to be repeated for every node $u \in V \setminus \{s, t\}$. For directed graphs $G = (V, A)$, we are unaware of any polynomial-time method for identifying redundant nodes. Instead, we suggest the following two approximative strategies, each running in $\mathcal{O}(n + m)$ time.

The first strategy removes redundant nodes by considering the *condensation* of G. After removing all the incoming arcs to s and the outgoing arcs from t, the condensation is constructed by contracting each strongly connected component (SCC) of G into a corresponding node. This forms a DAG, in which an arc in the DAG connects nodes in different SCCs. Now, SCCs in the DAG are redundant if and only if they cannot be reached from the source component C_s or cannot reach the target component C_t. This can be determined using a bidirectional search from C_s and C_t, with the latter via the reverse arcs of the DAG.

A second strategy considers the *block-cut tree* of the undirected version of G. The nodes in this tree correspond to the biconnected components (BCCs) or the articulation points (APs) in G. The edges in the tree connect an AP to the BCCs to which it belongs. BCCs in the tree are redundant if and only if they are not on the path from the source component C_s to the target component C_t. This can be confirmed by performing a breadth-first search from C_s.

Figure 1 illustrates these strategies. For the illustrated graph, both strategies are able to identify redundant nodes. In this case, the first identifies the nodes v_9 and v_{10} as redundant, while the second identifies the nodes v_4 and v_8 as redundant. Note, however, that these techniques may not identify all redundant nodes,

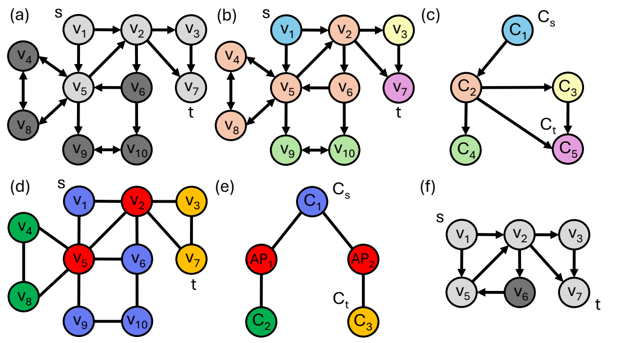

Fig. 1. Illustration of the redundant node removal process. Fig. 1(a) shows a directed graph in which light grey nodes are non-redundant and dark grey nodes are redundant. Fig. 1(b) presents the same graph with nodes coloured according to their SCC. Fig. 1(c) depicts the condensation of the graph in (b), where each node in the DAG is coloured to match its corresponding SCC. The nodes within C_4 are redundant here. Fig. 1(d) shows the undirected version of (a), with APs highlighted in red and the remaining nodes coloured by their BCC. Fig. 1(e) illustrates the block-cut tree corresponding to (d), where each node is coloured according to its associated BCC or AP. The nodes within C_2 are redundant here. Fig. 1(f) shows the resulting graph after removing the redundant nodes $\{v_4, v_8, v_9, v_{10}\}$.

as is the case with v_6 in Fig. 1(f) (although in this case v_6 can be smoothed, as discussed in the following paragraphs).

Another problem reduction technique involves smoothing, which can remove further nodes and arcs from a graph. This follows the approach outlined in [23], where a node u is classified as *intermediate* if and only if it satisfies one of the following conditions: (i) u has exactly one incoming and outgoing arc, each connected to two different nodes, or (ii) u has two incoming and two outgoing arcs, connected to the same two nodes.

Nodes that do not fit this definition are considered *intersection* nodes. Intermediate nodes can be smoothed, that is, removed and replaced with a direct arc between their neighbours, while preserving the overall graph structure. In Case (i), where u has the predecessor set $\{v_1\}$ and the successor set $\{v_2\}$, the node u and the arcs (v_1, u) and (u, v_2) are removed. A new arc (v_1, v_2) is added, with weight $w(v_1, v_2) = w(v_1, u) + w(u, v_2)$. In Case (ii), where u has the predecessor and successor sets $\{v_1, v_2\}$, the node u and the arcs (v_1, u), (v_2, u), (u, v_1), and (u, v_2) are removed. Two new arcs are then introduced: (v_1, v_2) with weight $w(v_1, v_2) = w(v_1, u) + w(u, v_2)$ and (v_2, v_1) with weight $w(v_2, v_1) = w(v_2, u) + w(u, v_1)$. In real-world street networks, nodes fitting Case (i) typically represent points occurring along a one-way street, while nodes fitting Case (ii) correspond to points on a two-way street. In both scenarios, these nodes can be removed, as they allow movement only in one direction until an intersection node is reached. Consequently, removing these nodes preserves navigational logic.

One further option for reducing problem size involves, for a given node u, considering the shortest s-u and u-t paths, assuming that all arc weights are non-negative. Let $LS(u)$ and $LT(u)$ denote the lengths of the shortest s-u and u-t paths, respectively. Now, if $LS(u) + LT(u) > k$, then any s-t path that contains u must also exceed k, perhaps justifying u's removal. The shortest paths from s and t can be obtained by constructing shortest path trees from s and from t (using reverse arcs for the latter) with Dijkstra's algorithm in $\mathcal{O}((n+m)\log n)$ time. Since the shortest path length provides only a lower bound, there may still exist s-t paths with lengths greater than k. Note, however, this approach may eliminate good-quality, or even optimal, solutions.

4 Proposed Methods

In this section, we describe our two proposed approaches for the KPP. As stated earlier, the first is a local search heuristic that uses a neighbourhood operator to explore the solution space, potentially accepting worse solutions to help escape local optima. The second is an exact, heuristically-guided backtracking algorithm whose runtime is exponential in the worst case. Both algorithms start with an initial solution. Here, this is generated by executing five path-finding methods, and taking the s-t path among these whose cost is minimal as the initial solution.

The first two methods are the standard breadth-first search (BFS) and depth-first search (DFS) algorithms. The third is a randomised version of BFS, called random-first search (RFS), where the queue structure is replaced by an unordered set from which nodes are selected at random. All three methods construct a spanning tree in $\mathcal{O}(n+m)$ time. Finally, s-t paths are also generated by Dijkstra's algorithm in $\mathcal{O}((n+m)\log n)$ time, and via a randomised version of Prim's algorithm, in which a random arc is selected from the list instead of the one with minimum weight. This randomised variant maintains a time complexity of $\mathcal{O}(n+m)$, improving upon the $\mathcal{O}(m+n\log n)$ complexity of the standard algorithm.

4.1 Local Search

In this subsection, we describe our local search algorithm, which uses a problem-specific neighbourhood operator to generate new solutions from a previously observed one. The solution space of this algorithm is the set of all s-t paths in G. The complete pseudocode is given in Algorithm 1. Here, we adopt the *Great Deluge* metaheuristic [9]. For this, a boundary value B is initially set to a large value (Line 1), which allows the algorithm to sometimes accept worsening moves, enabling a greater exploration of the solution space.

Our neighbourhood operator is described in Lines 3-6 and works as follows. An illustration is also given in Figure 2. Let $P = (s = u_1, u_2, \ldots, u_l = t)$ be our current s-t path in G. A random node $u_i \in P$ is first selected, all the arcs of P are temporarily removed from G, and a spanning tree T is generated from u_i

Algorithm 1: Local Search Algorithm for the KPP

```
Input: An arc-weighted graph $G = (V, A)$, an initial $s$-$t$ path $P$, a desired
       path length $k$, and a time limit
Output: The best $s$-$t$ path observed during execution, $P_{\text{best}}$
1  Set the boundary $B = B_{\text{init}} = |k - L(P)|$ and set $P_{\text{best}} = P$
2  while $B > 0$ and $|k - L(P_{\text{best}})| > 0$ do
3      Let $P = (s = u_1, u_2, \ldots, u_l = t)$ be the sequence of nodes in the current
         path
4      Select a random node $u_i \in P$ such that $u \neq t$ , and remove the arcs of $P$
         from $G$
5      Use a path-finding method to form a tree $T$ rooted at $u_i$ to all reachable
         nodes $u_j \in P$. Reinstate the arcs of $P$ to $G$
6      Let $R$ be the set of all $u_j \in P$ reachable from $u_i$ in $T$ such that $j > i$
7      foreach $u_j \in R$ (in random order) do
8          Let $P'$ be a copy of $P$ with the $u_i$-$u_j$ path in $P$ replaced by the $u_i$-$u_j$
             path from $T$
9          if ( $|k - L(P')| \leq |k - L(P)|$ or $|k - L(P')| \leq B$ ) and $P'$ is a feasible
             path then
10             Set $P = P'$
11             if $|k - L(P)| < |k - L(P_{\text{best}})|$ then
12                 Set $P_{\text{best}} = P$
13             break
14     Update $B$ according to the remaining time (as detailed in Subsection 4.1)
```

using one of the aforementioned path-finding methods. Note that T contains no arcs in common with P. The set R is now defined as the subset of nodes in P that are reachable from u_i in T and that appear after u in P.

In Lines 7-13, each node $u_j \in R$ is considered in a random order, and a new path P' is formed by replacing the u_i-u_j path in P with the corresponding u_i-u_j path in T. As shown, P' is accepted as P if its cost is less than or equal to the cost of the current path P, or if it is less than or equal to B. Note that P' may not be a *feasible* s-t path; that is, it may contain repeated nodes and/or arcs. Therefore, it is necessary to check if the path is feasible and reject it immediately if it is not. This process continues until P' is accepted as the new solution, or all $u_j \in R$ have been considered. If the cost of this new P is better than that of the best solution seen so far P_{best}, then P is stored as the new best solution.

As the search progresses, B decreases gradually, reducing the likelihood of accepting worsening moves (Line 14). To ensure that B decreases linearly and reaches zero when the time limit is reached, we set $B = (1 - p) \cdot B_{\text{init}}$, where p denotes the proportion of elapsed time during the local search. This process is repeated until the time limit is reached or a solution with length equal to k (i.e. a cost of zero) is found (Line 2).

We chose the great deluge metaheuristic because it requires fewer parameters, unlike simulated annealing, which proved highly sensitive to the cooling

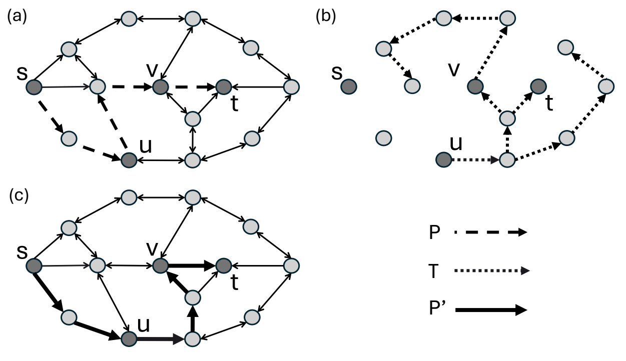

Fig. 2. Illustration of the local search neighbourhood operator. Fig. 2(a) depicts an example s-t path P in a graph, with a node $u \in P$ selected. Fig. 2(b) presents the same graph with the arcs of P removed, along with a tree T rooted at u that spans all nodes reachable from u. Of all the nodes in P that are reachable in T, a random node v is selected. Fig. 2(c) shows the resulting s-t path P', where the original u-v path in P has been replaced by the corresponding u-v path from T.

schedule and initial temperature in our initial experiments. Tabu search is another option, but evaluating the set of all neighbouring solutions is impractical given the exponential number of paths between node pairs.

4.2 Backtracking Algorithm

Our second approach extends the backtracking algorithm proposed by Hambly et al. [16] by introducing a heuristic priority-based node selection strategy that aims to identify good-quality paths earlier in the search, and additional mechanisms to further eliminate parts of the search tree. The algorithm performs a DFS-style traversal and returns the optimal solution either when a path of length k has been found or when the algorithm has backtracked to the root of the search tree. If the time limit is reached before either condition occurs, the best path found so far P_{best} (initialised as the initial solution mentioned previously) is returned. Since the total number of s-t paths considered can grow exponentially with the size of the graph, the worst-case complexity is also exponential.

During execution, let $P = (s, \ldots, u)$ be the current s-u path being considered by the algorithm. To help determine which node v should next be added to P, a stack $S = [Q(\Gamma'(s)), \ldots, Q(\Gamma'(u))]$ is maintained. Here, $\Gamma'(u)$ is a subset of the neighbours of u, in which the nodes $v \in \Gamma(u)$ are present in $\Gamma'(u)$ if and only if $v \notin P$ and $L(P) + w(u, v) + LT(v) < \overline{k}$, where $LT(v)$ is the shortest v-t path length from Section 3. Here, $\overline{k}$ is an upper bound (initialised as $k+|k-L(P_{best})|$), and prevents v from being added to $\Gamma'(u)$ if the sum of the length of the current s-u path, the weight of the arc (u, v), and the length of the shortest v-t path exceeds $\overline{k}$ (assuming non-negative arc weights). During execution, $\overline{k}$ is updated to $k + |k - L(P)|$ whenever P_{best} is updated. We also employ a counter c, which tracks the number of nodes in P that have an arc incident to t, and a constant C, which is initialised to the in-degree of t. If $c = C$, then only t is added to $Q(\Gamma'(u))$. That is, no other neighbours from u are explored, as they cannot lead to another s-t path.

Each queue $Q(\Gamma'(u))$ defines an ordering of the elements of $\Gamma'(u)$ according to one of three operators. In our case, these operators rely solely on precomputed or readily available information, enabling constant-time decision-making. The first operator uses the shortest v-t path length $LT(v)$. The nodes in $Q(\Gamma'(u))$ are then ordered according to how close $L(P) + w(u,v) + LT(v)$ is to k. The second operator adopts a greedy strategy, ordering nodes in $Q(\Gamma'(u))$ based on how close $L(P) + w(u,v)$ is to k. In the third operator, the nodes in $Q(\Gamma'(u))$ are randomly ordered. In any case, if $v = t$, it is put to the front of the queue.

Each loop of the algorithm operates as follows. If $Q(\Gamma'(u))$ is empty, then it is removed from the stack S and u is removed from P. If u has an arc incident to t, then c is decreased by one. If $Q(\Gamma'(u))$ is non-empty, then the node v at the front of the queue is popped and added to P. If v has an arc incident to t, then c is incremented by one. If $v = t$ and P has a cost lower than P_{best}, then P_{best} is set to P, $\overline{k}$ is updated (as described above), and t is removed from P. If $v \neq t$ and $c \neq C$, then $Q(\Gamma'(v))$ is formed (as described above). If $c = C$, then only t is added to $Q(\Gamma'(v))$. Finally, if $Q(\Gamma'(v))$ is non-empty, it is pushed onto S; otherwise, v is removed from P.

5 Experimental Analysis

In this section, we evaluate the performance of our local search and backtracking algorithms. The aim is to identify which method produces lower-cost solutions across varying graph topologies and values of k. Here, both algorithms were implemented in C++ using adjacency lists to represent the graphs. The shortest path sum reduction technique was also implemented in C++, while the redundant node removal and smoothing techniques were developed in Python using the NetworkX library. Experiments were conducted on Windows 10 machines with 3.5 GHz processors and 8 GB of RAM. Both algorithms were tested using a 300-second time limit per instance. Local search was also evaluated under a 10-second time limit to assess its performance when time is more restricted.

In the local search algorithm, we employed the randomised Prim's algorithm to construct the tree in Line 5 of Algorithm 1. In preliminary experiments, we found that this method produced the best solutions, as it benefitted from a strong randomisation element. For the backtracking algorithm, we also tested the shortest path (BSP), greedy (BG), and randomised (BR) operators described in Subsection 4.2. A complete listing of our source code and problem generator is available at https://doi.org/10.5281/zenodo.18257021.

We consider three graph topologies in our experiments. First, we use street networks from five real-world cities: Barcelona, Berlin, London, New York, and Paris. These were obtained using the OSMnx library [7], with a 5×5 km bounding box centred over each city. The details of these graphs are summarised in Table 1.

Second, we consider planar graphs with Euclidean arc weights. For our tests, these were generated by placing 25,000 nodes uniformly at random in a $10,000 \times 10,000$ unit square. A Delaunay triangulation was then formed among these nodes to generate a maximal planar graph. From this, a directed spanning tree was extracted, and additional arcs from the triangulation were added at random until the graph contained the user-prescribed number of arcs, m. Each arc was then assigned a weight equal to the Euclidean distance between its endpoints. Here, we look at sparse, medium, and dense planar graphs with 60,000, 90,000, and 120,000 arcs, respectively[3].

Finally, we generated random directed graphs using the Erdős-Rényi $G(n, p)$ model, where each possible arc between distinct node pairs is included independently with probability p. Arc weights were then sampled uniformly at random from the set $\{1, 2, ..., 10000\}$. Although random graphs with randomly assigned arc weights are not representative of real-world road networks, we include them in our experiments primarily to stress test the algorithms.

Table 1 presents the performance of the local search algorithm and the BSP backtracking operator across the five cities for a small, medium, and large value of k. Here, we also see the effects of the redundant node removal and smoothing problem reduction techniques. On average, approximately half of the nodes were removed in two of the five cities, while the remaining three experienced average reductions of 36%, 27%, and 11%, respectively. The results for the other backtracking operators are omitted here due to their consistently poor performance. In general, both algorithms tend to produce higher solution costs as k increases. For large values of k, paths of length k become more rare in the graph, making them more difficult and time-consuming to identify. For the backtracking algorithm, at larger k values, the upper bound $\overline{k}$ is rarely updated, forcing the algorithm to explore more of the search tree and increasing the runtime. For the local search algorithm, when k is large, the initial solution is far from the target length, requiring larger moves to obtain solutions closer to k.

On average, local search consistently produced lower-cost solutions across the five cities, even under the more restricted time limit. This highlights its robustness and overall superiority on real-world street networks. In contrast, BSP occasionally found the optimal solution. In all cases, this is because it found a solution with zero-cost, rather than backtracking to the root of the search tree. BSP often found optimal solutions quickly but was inconsistent; when no zero-cost solution was found, it rarely improved on the initial solution, resulting in high average costs.

Figure 3 illustrates the performance of the algorithms on planar graphs for values of k up to 1,500,000. On average, the sparse, medium, and dense planar graphs saw node reductions of 37%, 2%, and 0.1%, respectively due to the redundant node removal and smoothing problem reduction techniques. As with our previous results, the costs of the solutions increase for larger k's with both algorithms. For all levels of graph density, the BG and BR backtracking operators consistently produce solutions with the highest costs. This is perhaps

[3] In any directed planar graph, the maximum number of arcs is always $6n - 12$.

Table 1. Average costs and percentage of zero-cost solutions obtained by the local search algorithm and the BSP backtracking operator across the five cities. Each cost value represents the mean over 20 runs with randomly selected source and target nodes, plus/minus the coefficient of variation (CV). For BSP, the mean runtime of successful runs is also reported. Next to the name of each city are, respectively, the number of nodes and arcs in the original graph, and the mean percentage of nodes removed due to the redundant node removal and smoothing problem reduction techniques.

City	$k = 15{,}000$		$k = 50{,}000$		$k = 250{,}000$	
	Cost	% zero-cost	Cost	% zero-cost	Cost	% zero-cost
Barcelona	(74,474, 237,512, 11.416%)					
LS 300s	0.1 ± 4.5	95%	0.6 ± 1.5	60%	7.1 ± 1.6	5%
LS 10s	4.4 ± 1.1	20%	543.0 ± 1.8	0%	74,319.9 ± 1.1	0%
BSP	7.9 ± 4.5	95% (0.1)	2,485.3 ± 4.2	90% (1.9)	92,552.6 ± 1.1	40% (10.4)
Berlin	(83,648, 214,737, 53.463%)					
LS 300s	0.0 ± 0.0	100%	0.1 ± 3.1	90%	2.1 ± 1.1	5%
LS 10s	0.8 ± 1.4	50%	14.8 ± 1.0	5%	17,032.9 ± 1.2	0%
BSP	349.6 ± 3.0	85% (0.4)	8,498.7 ± 2.1	80% (11.5)	101,688.1 ± 0.9	35% (68.4)
London	(68,146, 179,454, 46.017%)					
LS 300s	0.0 ± 0.0	100%	0.2 ± 2.4	85%	1.2 ± 1.8	45%
LS 10s	1.3 ± 1.3	45%	13.3 ± 1.0	0%	5,675.7 ± 1.8	0%
BSP	13.3 ± 4.5	95% (2.9)	18,874.4 ± 1.0	45% (4.5)	87,712.0 ± 1.0	30% (19.6)
New York	(37,724, 120,260, 36.411%)					
LS 300s	0.0 ± 0.0	100%	0.1 ± 4.5	95%	2.0 ± 1.3	35%
LS 10s	0.7 ± 2.4	75%	7.2 ± 1.1	10%	23,199.3 ± 1.0	0%
BSP	2.6 ± 4.5	95% (0.1)	8,128.1 ± 2.1	75% (5.6)	68,884.6 ± 0.8	25% (9.5)
Paris	(79,054, 224,014, 27.184%)					
LS 300s	0.0 ± 0.0	100%	0.5 ± 1.2	55%	4.0 ± 1.9	25%
LS 10s	2.1 ± 1.1	25%	19.5 ± 0.9	5%	37,497.5 ± 0.9	0%
BSP	29.4 ± 3.2	90% (0.2)	8,362.7 ± 2.1	80% (0.5)	148,786.4 ± 0.6	15% (0.5)

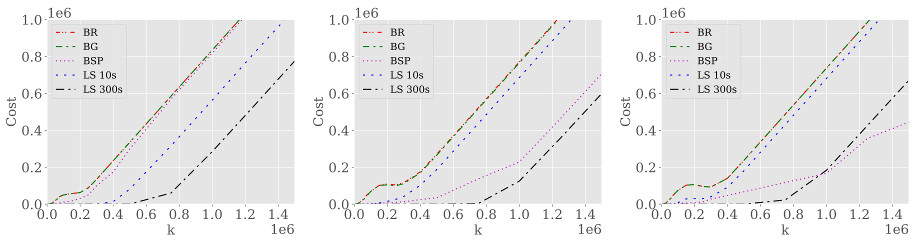

Fig. 3. Costs achieved for the KPP for various values of k on sparse, medium, and dense planar graphs, respectively. Each point is the mean across 20 graphs.

unsurprising, since BR selects the next node without any guidance, whilst BG follows a short-sighted strategy that does not reliably lead toward high-quality solutions. Indeed, as k increases, these two operators often struggle to improve on the initial solution within the time limit, resulting in poor solution costs.

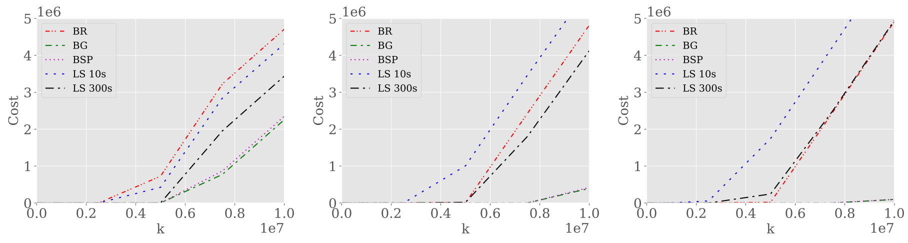

Fig. 4. Costs achieved for the KPP for various values of k on sparse, medium, and dense random graphs, respectively. Each point is the mean across 20 graphs.

Considering the remaining methods, Figure 3 shows that the local search algorithm achieves the lowest cost solutions on sparse planar graphs, consistently outperforming all backtracking operators, even with the shorter 10-second time limit. It also delivers superior performance in medium planar graphs and, for small to medium values of k, with dense planar graphs. In contrast, for the largest values of k on dense planar graphs, the performance of local search seems to decline compared to the BSP backtracking operator. This seems to be due to the neighbourhood operator, which often produces paths that are not feasible and are subsequently rejected.

In contrast, BSP produces the lowest-cost solutions for larger values of k in dense planar graphs. These results highlight the ability of BSP to effectively guide the search toward more promising regions of the solution space compared to the other backtracking operators. Notably, BSP achieves lower-cost solutions on dense graphs than on sparse ones. This is likely due to the greater number of arcs in the graph and, therefore, a larger number of paths of length k in denser graphs, enabling the faster discovery of optimal solutions.

Figure 4 shows the solution costs of the algorithms on random graphs with 1000 nodes, and probabilities of 0.01, 0.1, and 0.5 for values of k up to 10,000,000. The redundant node removal and smoothing problem reduction techniques had a negligible impact on the number of nodes reduced in random graphs. As with our previous results, the costs of the solutions increase for larger k's with both algorithms. For all backtracking operators and the local search algorithm with a 300-second time limit, these algorithms are generally able to find solutions with zero cost for small to medium values of k.

In contrast to previous results, for these graphs we see that, for larger values of k, the BSP and BG backtracking operators produce the lowest-cost solutions, with BG performing slightly better than BSP across all three graph densities. As the density of the graph increases, the solution costs obtained by these two operators also decrease. This is due to the greater number of arcs in denser graphs, which provides more s-t paths with lengths equal to or near k, allowing these operators to effectively guide the search toward promising regions of the solution space and, in some cases, identify paths close to the lengths of the longest paths. In contrast, solutions with the highest costs are produced by the BR backtracking operator on sparse random graphs and the 10-second local search algorithm on medium and dense random graphs. BR lacks any mechanism to guide the search toward promising regions of the solution space and therefore performs worse than the other two backtracking operators, while local search with a 10-second time limit does not have sufficient time to find low-cost solutions.

For these instances, unlike previously, the local search algorithm with a 300-second time limit produces worse solution costs than BSP and BG on sparse and medium random graphs, and higher solution costs than all backtracking operators on dense random graphs. This occurs for two main reasons. First, like previously, the neighbourhood operator produces many paths that are not feasible and therefore rejected for large values of k. Second, the use of random arc weights means that there can be large jumps in the lengths of newly generated paths, which may be rejected if their costs exceed the current solution or are greater than the boundary value B.

6 Conclusion

This paper has presented two methods for tackling the $\mathcal{NP}$-hard problem of finding k-length s-t paths in arc-weighted graphs. The first is a local search algorithm; the second is a heuristically-guided exact backtracking algorithm. We used non-negative arc weights in our experiments; however, the local search algorithm is applicable for graphs with negative weights, provided that Dijkstra's algorithm is not used to build the tree. We also proposed several problem reduction techniques to reduce the graph size.

On real-world networks and sparse and medium planar graphs, the local search algorithm consistently delivered solutions with the lowest costs, even when using a reduced limit of 10 seconds. However, on dense random graphs, the local search algorithm performed worse than all the backtracking operators. Among the backtracking operators, BR consistently produced higher solution costs. BSP achieved solutions of lower costs on real-world networks and planar graphs, whilst on random graphs, BG produced the lowest-cost solutions.

There are a number of directions for future research. One could consider the practical aspects of paths in the real world. For example, ensuring that paths are not on steep inclines or declines, avoiding poorly lit or isolated areas, avoiding high-traffic roads, or bypassing routes known to be unsafe for pedestrians or cyclists. Another direction could involve scenarios in which a user wishes to make multiple stops between the source and the target. This can be modelled as having multiple intermediate nodes, treating them as required waypoints. The objective is then to find an s-t path that visits each waypoint and has a length as close to k as possible.

With regard to the algorithms presented in this paper, further research could also explore the design of alternative neighbourhood operators for our local search approach, or the development of new operators for the backtracking algorithm to more effectively navigate promising regions of the solution space. It would also be worthwhile to investigate whether evolutionary algorithms, commonly applied to $\mathcal{NP}$-hard problems, are also suitable for the KPP.

Acknowledgments. This research is funded by EPSRC.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Aggarwal, S., Kumar, N.: Path planning techniques for unmanned aerial vehicles: A review, solutions, and challenges. Computer Communications **149**, 270–299 (2020). https://doi.org/https://doi.org/10.1016/j.comcom.2019.10.014, https://www.sciencedirect.com/science/article/pii/S0140366419308539
2. Alon, N., Yuster, R., Zwick, U.: Color-coding: a new method for finding simple paths, cycles and other small subgraphs within large graphs. In: Leighton, F.T., Goodrich, M.T. (eds.) Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada. pp. 326–335. ACM (1994). https://doi.org/10.1145/195058.195179, https://doi.org/10.1145/195058.195179
3. Azaron, A., Katagiri, H., Kato, K., Sakawa, M.: Longest path analysis in networks of queues: Dynamic scheduling problems. European Journal of Operational Research **174**(1), 132–149 (2006). https://doi.org/https://doi.org/10.1016/j.ejor.2005.02.018, https://www.sciencedirect.com/science/article/pii/S0377221705002122
4. Basagni, S., Bruschi, D., Ravasio, F.: On the difficulty of finding walks of length k. RAIRO Theor. Informatics Appl. **31**(5), 429–435 (1997). https://doi.org/10.1051/ita/1997310504291, https://doi.org/10.1051/ita/1997310504291
5. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Narrow sieves for parameterized paths and packings. J. Comput. Syst. Sci. **87**, 119–139 (2017). https://doi.org/10.1016/j.jcss.2017.03.003, https://doi.org/10.1016/j.jcss.2017.03.003
6. Bodlaender, H.L.: On linear time minor tests with depth-first search. J. Algorithms **14**(1), 1–23 (1993). https://doi.org/10.1006/jagm.1993.1001, https://doi.org/10.1006/jagm.1993.1001
7. Boeing, G.: Modeling and analyzing urban networks and amenities with osmnx. Geographical Analysis (2025). https://doi.org/10.1111/gean.70009, published online ahead of print
8. Chen, J., Lu, S., Sze, S., Zhang, F.: Improved algorithms for path, matching, and packing problems. In: Bansal, N., Pruhs, K., Stein, C. (eds.) Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007. vol. 7, pp. 298–307. Citeseer, SIAM (2007), http://dl.acm.org/citation.cfm?id=1283383.1283415
9. Dueck, G.: New optimization heuristics: The great deluge algorithm and the record-to-record travel. Journal of Computational Physics **104**(1), 86–92 (1993). https://doi.org/https://doi.org/10.1006/jcph.1993.1010, https://www.sciencedirect.com/science/article/pii/S0021999183710107
10. Eppstein, D.: Finding the k shortest paths. SIAM J. Comput. **28**(2), 652–673 (1998). https://doi.org/10.1137/S0097539795290477, https://doi.org/10.1137/S0097539795290477
11. Flum, J., Grohe, M.: The parameterized complexity of counting problems. SIAM J. Comput. **33**(4), 892–922 (2004). https://doi.org/10.1137/S0097539703427203, https://doi.org/10.1137/S0097539703427203

12. Fomin, F.V., Golovach, P.A.: Long circuits and large euler subgraphs. SIAM Journal on Discrete Mathematics **28**(2), 878–892 (2014). https://doi.org/10.1137/130936816, https://doi.org/10.1137/130936816
13. Fomin, F.V., Lokshtanov, D., Panolan, F., Saurabh, S.: Efficient computation of representative families with applications in parameterized and exact algorithms. J. ACM **63**(4), 29:1–29:60 (2016). https://doi.org/10.1145/2886094, https://doi.org/10.1145/2886094
14. Gao, Y., Na, J., Zhang, B., Yang, L., Gong, Q.: Optimal web services selection using dynamic programming. In: Proceedings of the 11th IEEE Symposium on Computers and Communications. p. 365–370. ISCC '06, IEEE Computer Society, USA (2006). https://doi.org/10.1109/ISCC.2006.116, https://doi.org/10.1109/ISCC.2006.116
15. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, USA (1979)
16. Hambly, D., Lewis, R., Corcoran, P.: Determining Fixed-Length Paths in Directed and Undirected Edge-Weighted Graphs. In: Liberti, L. (ed.) 22nd International Symposium on Experimental Algorithms (SEA 2024). Leibniz International Proceedings in Informatics (LIPIcs), vol. 301, pp. 15:1–15:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2024). https://doi.org/10.4230/LIPIcs.SEA.2024.15, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SEA.2024.15
17. Hetland, M.: Python Algorithms: Mastering Basic Algorithms in the Python Language. Professional and Applied Computing, Apress (2011), https://books.google.co.uk/books?id=4cytGpIPYsAC
18. Kelley, B.P., Sharan, R., Karp, R.M., Sittler, T., Root, D.E., Stockwell, B.R., Ideker, T.: Conserved pathways within bacteria and yeast as revealed by global protein network alignment. Proceedings of the National Academy of Sciences **100**(20), 11394–11399 (2003). https://doi.org/10.1073/pnas.1534710100, https://www.pnas.org/doi/abs/10.1073/pnas.1534710100
19. Kneis, J., Mölle, D., Richter, S., Rossmanith, P.: Divide-and-color. In: Fomin, F.V. (ed.) Graph-Theoretic Concepts in Computer Science, 32nd International Workshop, WG 2006, Bergen, Norway, June 22-24, 2006, Revised Papers. Lecture Notes in Computer Science, vol. 4271, pp. 58–67. Springer, Springer (2006). https://doi.org/10.1007/11917496_6, https://doi.org/10.1007/11917496_6
20. Kocay, W., Kreher, D.: Graphs, Algorithms, and Optimization, second edition. CRC Press (11 2016). https://doi.org/10.1201/9781315372563
21. Koutis, I.: Faster algebraic algorithms for path and packing problems. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games. Lecture Notes in Computer Science, vol. 5125, pp. 575–586. Springer, Springer (2008). https://doi.org/10.1007/978-3-540-70575-8_47, https://doi.org/10.1007/978-3-540-70575-8_47
22. Lewis, R., Corcoran, P.: Finding fixed-length circuits and cycles in undirected edge-weighted graphs: an application with street networks. J. Heuristics **28**(3), 259–285 (2022). https://doi.org/10.1007/s10732-022-09493-5, https://doi.org/10.1007/s10732-022-09493-5
23. Lewis, R., Corcoran, P.: Fast algorithms for computing fixed-length round trips in real-world street networks. SN Computer Science **5** (09 2024). https://doi.org/10.1007/s42979-024-03223-3

24. Lewis, R., Corcoran, P., Gagarin, A.V.: Methods for determining cycles of a specific length in undirected graphs with edge weights. J. Comb. Optim. **46**(5), 29 (2023). https://doi.org/10.1007/s10878-023-01091-w, https://doi.org/10.1007/s10878-023-01091-w
25. Monien, B.: How to find long paths efficiently. In: Ausiello, G., Lucertini, M. (eds.) Analysis and Design of Algorithms for Combinatorial Problems, North-Holland Mathematics Studies, vol. 109, pp. 239–254. North-Holland (1985). https://doi.org/https://doi.org/10.1016/S0304-0208(08)73110-4, https://www.sciencedirect.com/science/article/pii/S0304020808731104
26. Papadimitriou, C.H., Yannakakis, M.: On limited nondeterminism and the complexity of the V-C dimension. J. Comput. Syst. Sci. **53**(2), 161–170 (1996). https://doi.org/10.1006/jcss.1996.0058, https://doi.org/10.1006/jcss.1996.0058
27. Scott, J., Ideker, T., Karp, R.M., Sharan, R.: Efficient algorithms for detecting signaling pathways in protein interaction networks. J. Comput. Biol. **13**(2), 133–144 (2006). https://doi.org/10.1089/cmb.2006.13.133, https://doi.org/10.1089/cmb.2006.13.133
28. Sedgewick, R., Schidlowsky, M.: Algorithms in Java, Part 5: Graph Algorithms. Addison-Wesley Longman Publishing Co., Inc., USA, 3 edn. (2003)
29. Sedgewick, R., Wayne, K.: Algorithms, 4th Edition. Addison-Wesley (2011)
30. Suurballe, J.W.: Disjoint paths in a network. Networks **4**(2), 125–145 (1974). https://doi.org/https://doi.org/10.1002/net.3230040204, https://onlinelibrary.wiley.com/doi/abs/10.1002/net.3230040204
31. Tamassia, R.: Handbook of Graph Drawing and Visualization. Chapman & Hall/CRC, 1st edn. (2016)
32. Tsur, D.: Faster deterministic parameterized algorithm for *k*-path. Theor. Comput. Sci. **790**, 96–104 (2019). https://doi.org/10.1016/j.tcs.2019.04.024, https://doi.org/10.1016/j.tcs.2019.04.024
33. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM J. Comput. **8**(3), 410–421 (1979). https://doi.org/10.1137/0208032, https://doi.org/10.1137/0208032
34. Willems, D., Zehner, O., Ruzika, S.: On a technique for finding running tracks of specific length in a road network. In: Kliewer, N., Ehmke, J.F., Borndörfer, R. (eds.) Operations Research Proceedings 2017, Selected Papers of the Annual International Conference of the German Operations Research Society (GOR), Freie Universiät Berlin, Germany, September 6-8, 2017. pp. 333–338. Operations Research Proceedings, Springer, Springer (2017). https://doi.org/10.1007/978-3-319-89920-6_45, https://doi.org/10.1007/978-3-319-89920-6_45
35. Williams, R.: Finding paths of length k in $\mathrm{o}^*(2^\mathrm{k})$ time. Inf. Process. Lett. **109**(6), 315–318 (2009). https://doi.org/10.1016/j.ipl.2008.11.004, https://doi.org/10.1016/j.ipl.2008.11.004
36. Wong, W.Y., Lau, T.P., King, I.: Information retrieval in p2p networks using genetic algorithm. In: Special Interest Tracks and Posters of the 14th International Conference on World Wide Web. p. 922–923. WWW '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1062745.1062799, https://doi.org/10.1145/1062745.1062799
37. Zehavi, M.: Mixing color coding-related techniques. In: Bansal, N., Finocchi, I. (eds.) Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9294, pp. 1037–1049. Springer, Springer (2015). https://doi.org/10.1007/978-3-662-48350-3_86, https://doi.org/10.1007/978-3-662-48350-3_86