Journal of Heuristics (2022) 28:259–285
https://doi.org/10.1007/s10732-022-09493-5

Finding fixed-length circuits and cycles in undirected edge-weighted graphs: an application with street networks

R. Lewis[1] · **P. Corcoran**[2]

Received: 16 November 2021 / Revised: 14 February 2022 / Accepted: 1 March 2022 /
Published online: 23 March 2022

Abstract
This paper proposes two heuristic algorithms for finding fixed-length circuits and cycles in undirected edge-weighted graphs. It focusses particularly on a largely unresearched practical application where we are seeking attractive round trips for pedestrians and joggers in urban street networks. Our first method is based on identifying suitable pairs of paths that are combined to form a solution; our second is based on local search techniques. Both algorithms display high levels of accuracy, producing solutions within just a few meters of the target. Run times for the local search algorithm are also short, with solutions in large cities often being found in less than one second.

Keywords Heuristics · Graphs · Cycles · Circuits · Location-based services

1 Introduction

A location-based service is a piece of software that provides users with a service related to their geographical location. The most common types are *routing services*, which are available in vehicle navigation systems and online applications like Google Maps and Bing Maps. As input, routing services take a source and destination location together with a required mode of transport. They then compute one or more useful paths between these locations. The paths generated by routing software will generally be constrained by the given mode of transportation. For example, if we want to walk between locations, the network in question will correspond to footpaths and streets with

✉ R. Lewis
lewisr9@cf.ac.uk

P. Corcoran
CorcoranP@cardiff.ac.uk

1 School of Mathematics, Cardiff University, CF24 4AX, Cardiff, Wales

2 School of Computer Science and Informatics, Cardiff University, CF24 4AX, Cardiff, Wales

pedestrian access. What determines the usefulness of a solution will vary depending on the user and context in question. In many cases, a user will prefer a solution that minimises travel time; in others, they may want a solution that is safe (Hannah et al. 2018; Nunes et al. 2020) or simple to follow (in terms of the instructions needed no negotiate and remember the route Duckham and Kulik 2003) while still having a reasonably short travel time.

In this paper, we consider the problem of finding *fixed-length* routes on a map that start and end at the same location. This has various practical applications such as (a) planning a jogging route, (b) organising a cycling tour, or (c) determining a round trip that allows us to complete our daily quota of steps as determined by our fitness tracker. Hypothetically, routing services can be used to generate such round trips by setting the source to be equal to the destination; however, difficulties will usually arise because, as noted, these services tend to focus on minimum-length paths, making their proposed solutions unsuitable.

From a different perspective, routing services can also assist in the formation of fixed-length round trips by making use of intermediate geographical locations known as *waypoints*. In this case, a user can plan a route by specifying a series of waypoints, with each point being linked to its predecessor by a shortest path. This forms the basis of applications such as Strava Routes, which also allows users to specify preferences for routes involving hills, dirt tracks, and popular exercise trails. In this case, however, the task of designing a route of a *specific length* is still left very much to the user who will need to adopt a trial-and-error approach until an acceptable result is achieved.

In their simplest form, exercise routes of a specific length are easy to determine. For example, we may choose to simply travel back and forth along the same street repeatedly until the required distance is covered. Similarly, we might also perform "laps" within a particular locality. In this paper, however, we want to consider ways of producing routes that avoid repetition. Specifically, we want to avoid routes that ask the user to travel along a street or footpath more than once.

To define this problem formally, it is useful to first review some standard definitions from graph theory. Let $G = (V, E)$ be an undirected edge-weighted graph with n vertices and m edges, and let $w(u, v)$ denote the weight (or length) of an edge $\{u, v\} \in E$. A street network can be represented by such a graph by using vertices for street intersections and dead ends, edges for street segments between vertices, and edge-weights for the street segment lengths.

Definition 1 A *walk* is a series of pairwise incident edges in a graph; a *trail* is a walk with no repeated edges; and a *path* is a trail with no repeated edges or vertices.

It is also usual to add the prefix u-v to the above terms to signify a walk/trail/path that starts at vertex u and finishes at vertex v. The following terms can then also be used in cases where $u = v$.

Definition 2 A u-v-walk/trail/path is considered *closed* whenever $u = v$. Closed trails are usually known as *circuits*; closed paths are usually known as *cycles*.

Similarly to the above, the term u-circuit (cycle) can also be used to denote a circuit (cycle) that contains the vertex u. We can also extend this to multiple vertices: for example, a u-v-circuit is a circuit containing vertices u and v.

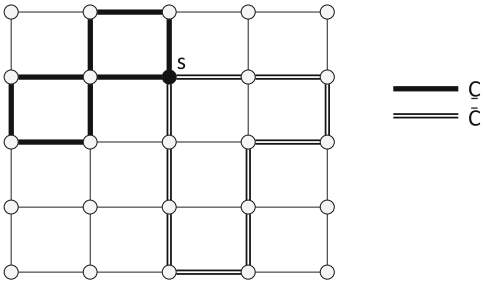

Fig. 1 An example grid graph in which all edge weights are assumed to be one. In this particular case $k = 9$, but no s-circuit with this length exists. Instead, the indicated solutions feature lengths of eight and ten

In this paper, we propose several heuristics for producing fixed-length circuits with a particular focus on tackling graphs resembling maps of roads and footpaths. For this work, we consider undirected graphs only. This is appropriate for practical applications that involve determining jogging and walking routes for pedestrians, though it is insufficient in situations involving vehicles and one-way streets.

An initial definition of this problem can be stated as follows:

Definition 3 (The k Circuit Problem) Let $G = (V, E)$ be an undirected graph with nonnegative edge weights, $s \in V$ be a *source* vertex, and k be a required length. The *k circuit problem* involves determining an s-circuit $C = (u_1, u_2, \ldots, u_l)$ in G (where $u_i \in V \ \forall i \in \{1, \ldots, l\}$ and $u_1 = u_l = s$) such that the total length (weight) of its edges $L(C) = \sum_{i=1}^{l-1} w(u_i, u_{i+1})$ is equal to k.

Note that this definition involves circuits as opposed to cycles. This means that a route can visit a vertex more than once but it cannot traverse an edge more than once. Due to our focus on street networks in this paper, here we choose to generalise this definition slightly. This is for two reasons. The first reason is that, in cases where circuits of length k cannot be found, it is desirable for two candidate solutions to be produced: $\underline{C}$, where $(k - L(\underline{C})) > 0$ is minimal; and $\bar{C}$, where $(L(\bar{C}) - k) > 0$ is minimal. The lengths of $\underline{C}$ and $\bar{C}$ can therefore be seen as lower and upper bounds on solution quality. An example is shown in Fig. 1.

Our second reason for generalising this problem is that street networks often contain many edges that are bridges.[1] By definition, circuits are not permitted to contain any bridges $\{u, v\}$ because their inclusion implies the need for a second (unavailable) u-v-trail. In street networks, however, it might be desirable to allow bridges to occur twice in a route, particularly in situations where the source s belongs to a relatively small bridge-connected component. (See Fig. 2, for example.)

This generalised version of the k circuit problem, which is the focus of this paper, is therefore stated as follows.

Definition 4 (The Generalised k Circuit Problem (GKCP)) Given the same input as Definition 3, the GKCP involves determining a closed s-s-walk $C = (u_1, u_2, \ldots, u_l)$ in G such that the total length (weight) of its edges $L(C) = \sum_{i=1}^{l-1} w(u_i, u_{i+1})$ is equal to k. Furthermore, edges can occur at most once in C except for bridges, which can occur at most twice. In cases where $L(C) = k$ cannot be achieved, two candidate

[1] Recall that a bridge is an edge in a graph that, if removed, will increase its number of components.

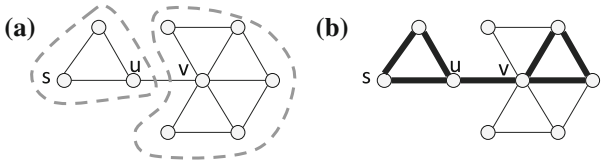

Fig. 2 Part **a** shows an example graph featuring two bridge-connected components. These occur either side of the bridge $\{u, v\}$. Only one s-circuit exists in this graph. Part **b** shows a solution in which $\{u, v\}$ is permitted to be used twice. This allows edges in the rightmost bridge-connected component to also be used

solutions are also required: $\underline{C}$, where $(k - L(\underline{C})) > 0$ is minimal; and $\bar{C}$, where $(L(\bar{C}) - k) > 0$ is minimal.

For convenience, the remainder of this paper will continue to use the term "circuit" when referring to candidate solutions of the GKCP; however, we should be mindful that they may involve using some edges twice, which does not fit the strict definition of a circuit. Note also that in cases where G has no bridges and features an s-circuit of length k the GKCP is equivalent to the k circuit problem given in Definition 3.

In the next section, we survey relevant research for this problem area. In Sect. 3, we then develop our two main methods for the GKCP: one based on identifying pairs of edge-disjoint paths, and one based on local search. In Sect. 4 we analyse the performance of these methods on a wide range of problem instances. Section 5 then shows how our methods can be adapted to cycles instead of circuits. Finally, Sect. 6 concludes the paper and makes suggestions for future research.

2 Problem analysis and existing work

From a computational perspective, very little work seems to have been conducted on the problem of finding fixed-length circuits and cycles in edge-weighted graphs. Various complexity results for related problems are known, however. For unweighted graphs, the number of walks of length k between pairs of vertices can be found by taking the binary adjacency matrix of a graph G and raising it to the kth power. Currently, the best-known algorithms for matrix multiplication operate in $\mathcal{O}(n^{\omega})$ time, where $\omega \approx 2.37369$ is the best-known exponent for the problem (Davie and Stothers 2013). The overall runtime is, therefore, $\mathcal{O}(kn^{\omega})$, which will be high for large values of k. Basagni et al. (1997) have also noted that the problem of calculating an s-t-walk of length k can be solved in polynomial time when using unweighted graphs providing that $k = n^{O(1)}$; however, for edge-weighted graphs the problem is $\mathcal{NP}$-hard. The task of calculating s-t-paths of length k in a graph is also known to be $\mathcal{NP}$-hard, though certain topologies such as trees and directed acyclic graphs can be solved in polynomial time. The problem of *counting* the number of s-t-paths in a graph is also known to be $\#\mathcal{P}$-complete. Roberts and Kroese (2007), for example, have found that random graphs of density d, have approximately

$$\sum_{i=0}^{n-2} \frac{(n-2)!}{i!} \cdot d^{n-1+(3.32/n)-(5.16/(dn))} \tag{1}$$

different paths between any two vertices on average.

For circuits and cycles, similar complexity results are known. The task of identifying a circuit in a graph G is equivalent to the problem of identifying an Eulerian subgraph in G (recall that an Eulerian graph is a connected graph in which the degrees of all vertices are even); however, the problem of identifying the *longest* Eulerian subgraph in G is known to be $\mathcal{NP}$-hard, both for weighted and unweighted graphs (Skiena 1990). This tells us that the GKCP is also $\mathcal{NP}$-hard since it is equivalent to the longest Eulerian subgraph problem whenever k is set to a sufficiently large value. Similar reasoning can also be applied to the problem of finding cycles of length k in a graph due to its relationship with the $\mathcal{NP}$-hard Hamiltonian cycle problem.

In terms of existing methods, Johnson (1975) has proposed an algorithm for enumerating all cycles in a directed graph. Each cycle is determined in $\mathcal{O}(n+m)$ steps, so the overall run time is $\mathcal{O}(c(n+m))$, where c is the total number of cycles. It is noted, however, that the growth rate of c can exceed the exponential 2^n, so run times will be infeasible in most cases. Alon et al. (1995) have also proposed exact methods for determining length-k paths and cycles in unweighted graphs. For undirected graphs, their algorithms operate in $\mathcal{O}(2^k n \log n)$ and $\mathcal{O}(2^k n^\omega \log n)$ time for paths and cycles respectively. Contrary to what is required here, these methods return *any* path/cycle of length k, as opposed to those containing specific vertices.

One recently suggested exact method for finding s-cycles of length k in edge-weighted graphs is due to Willems et al. (2018). This operates by first creating a dummy vertex s' whose set of neighbouring vertices $\Gamma(s')$ is made equal to that of s. The task is to then identify an s-s'-path of length k that contains at least three edges. In the original graph, this path corresponds to a cycle of length k. To determine a suitable s-s'-path, Willems et al. suggest using the algorithm of Yen (1971). Yen's algorithm is designed to determine the K shortest paths between any two given vertices and operates by finding the shortest path, followed by the second shortest path, the third shortest path, and so on. In Willems et al.'s case, Yen's algorithm is run until the first s-s'-path of length $\geq k$ is observed. This approach has issues, however.

- First, the complexity of Yens algorithm is $\mathcal{O}(Kn(m+n\log n))$, where $(m+n\log n)$ is an asymptotic bound of Dijkstra's shortest path algorithm (using Fibonacci heaps Cormen et al. 2009). This means that large values for K lead to long run times. In this current application, the desired K is not known beforehand, so the algorithm will be forced to continue iterating until a suitable s-s'-path has been found. For large values of k, this can lead to serious scaling-up issues, as we will see in Sect. 4.1.
- Second, as previously noted, graphs corresponding to street maps can contain bridges and articulation points that need to be used more than once. However, this method will not allow this to occur. One way of adapting to these circumstances is to add dummy vertices to the graph to raise its vertex connectivity to two. To do this, a dummy vertex v' should be added for each articulation point v in the graph,

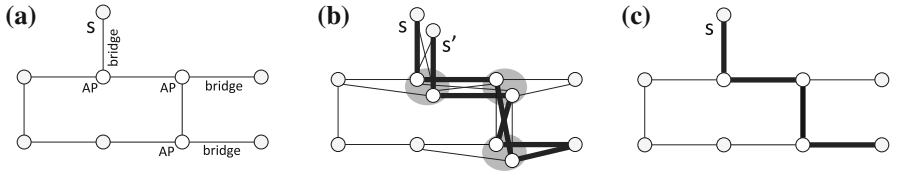

Fig. 3 Part **a** shows an example graph with three articulation points and three bridges. Part **b** shows the same graph with dummy vertices added for s and each articulation point. It also shows a valid s-s'-path. However, Part **c** shows that this path does not correspond to a valid solution because some non-bridge edges are used more than once

with $\Gamma(v')$ being set to that of $\Gamma(v)$ in all cases. However, these modifications can still lead to invalid solutions in some cases, as demonstrated in Fig. 3.

A variant of Yen's algorithm is the $\mathcal{O}(m + n \log n + K)$ algorithm of Eppstein for finding the K shortest trails between two vertices in a graph (Eppstein 1998). However, this is only designed for directed graphs and therefore allows opposing arcs (u, v) and (v, u) to both occur in a trail. This is not suitable for the GKCP where undirected non-bridging edges $\{u, v\}$ can be used at most once.

3 Algorithms for the GKCP

In this section we propose two heuristic algorithms for the GKCP: one based on generating pairs of edge-disjoint paths and one based on local search.[2]

As mentioned earlier, although repeated edges are not strictly permitted in circuits, the GKCP allows for bridging edges to be included twice in a solution. This would be necessary if we were to try and form a "circuit" containing vertex s in the graph shown in Fig. 3a, for example. One way of allowing a bridges to be used twice is to modify the graph so that, for each bridge $\{u, v\}$, we create two dummy vertices u' and v' together with the edges $\{u, u'\}$, $\{v, v'\}$ and $\{u', v'\}$ such that $w(u', v') = w(u, v)$ and $w(u, u') = w(v, v') = 0$; however, in problem instances containing many bridges this can make the underlying graph much larger. In our case, we therefore use specialised mechanisms within our heuristics that allow bridges to be used twice, but only when necessary.

Before executing our algorithms, some preprocessing can also be performed. Given an edge-weighted graph $G = (V, E)$ together with a source vertex $s \in V$, observe that any vertex v whose distance is more than $k/2$ units from s can be removed since, in such cases, all s-v-circuits will be longer than k. This results in smaller graphs for lower values of k, helping to shorten run times. This step can be achieved by generating a single shortest-path tree rooted at s using, for example, Dijkstra's algorithm.

A second optional preprocessing step is to repeatedly remove any degree-one (leaf) vertices, not including s itself. In street networks, leaf vertices correspond to dead-ends and cul-de-sacs, and it might be undesirable for users to be asked to travel up and

[2] A less developed version of the first heuristic was previously reported in Lewis (2020). This earlier version did not contain our current methods for dealing with bridges and was unable to produce circuits whose lengths exceeded k.

down such streets. Removing leaves will therefore eliminate these options, helping to reduce graph sizes and run times. On the other hand, their removal will also lead to a smaller solution space and, potentially, less accurate solutions. The implications of using this second preprocessing step are explored in Sect. 4.2 later.

3.1 Double path heuristic

Our first heuristic constructs solutions by generating a pair of edge-disjoint paths between the source vertex s and a particular target vertex t. In an undirected graph, the union of these s-t-paths forms an s-t-circuit. If these paths also happen to be vertex disjoint, then their union will be an s-t-cycle. The aim is to now identify the vertex t for which the sum of the lengths of the two generated s-t-paths is as close to k as possible.

A single path between a pair of vertices can be formed in various ways. One strategy is to use depth-first search, though this can often produce long meandering paths that, for this application, could be unattractive to the user. Another alternative is breadth-first search, which generates paths between vertices containing the minimum number of edges. Maximum flow methods such as Dinitz's $\mathcal{O}(n^2m)$ algorithm (Dinitz 1970) can also be used to find a maximally-sized set of edge-disjoint s-t-paths. Here, we choose to focus on using *shortest* paths between vertices (in terms of the sum of the edge-weights within the paths), as doing so allows various algorithm speed-ups to be achieved, as we will see in Sect. 3.1.1.

A naïve method for determining two edge-disjoint paths between s and t is to produce a single s-t-path, remove this path's edges from the graph, and then find a second s-t-path. However, this approach has faults. Figure 4a, for example, shows a small edge-weighted graph and its corresponding shortest s-t-path. Removing the edges of this path then disconnects s and t, preventing a second s-t-path from being formed. Better techniques are therefore needed.

Suurballe (1974) and Bhandari (1999) have previously proposed methods for finding the pair of edge-disjoint s-t paths whose edge-weight sum is minimal. An example of Bhandari's method is also given in Fig. 4. As shown, the shortest s-t-path P_1 is first found. In the second step, the graph is then modified by adding directions and adjusting weights along P_1. Specifically, each arc (u, v) in the path travelling from s to t has its weight modified to $B + w(u, v)$, where B is a large constant such that $B \geq \sum_{\{u,v\} \in E} w(u, v) + 1$. The weights of the reverse arcs (v, u) are then also set to $-w(u, v)$ before calculating the shortest s-t-path P_2 in this modified graph. Finally, the graph is reset, and the two paths are "unwoven" to form the final pair of paths, as shown in Fig. 4c.

The unweaving process in this algorithm operates using the steps shown in Algorithm 1. The output is an edge multiset C that corresponds to an Eulerian multigraph. The edges of this multigraph can then be ordered into a circuit using the linear-time algorithm of Hierholzer (1873). As shown in this algorithm, if an arc (u, v) appears in P_1 and its reverse (v, u) appears in P_2, then the edge $\{u, v\}$ is not included in C. This is the case with edge $\{v_1, v_2\}$ in Fig. 4c. On the other hand, the use of the constant B in the graph modification process ensures that P_1 and P_2 are maximally edge-disjoint;

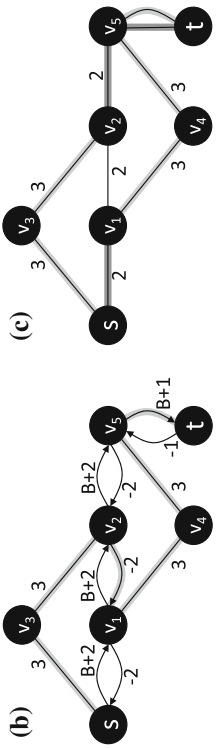

Fig. 4 Part **a** shows the shortest *s*-*t*-path in an example graph *G*. Part **b** shows a modified version of *G* and the corresponding shortest *s*-*t*-path; Part **c** shows the resultant solution formed by "unweaving" the two paths

consequently, an edge $\{u, v\}$ will occur twice in C if and only if it is a bridge relative to s and t in the original graph (Bhandari 1999). This is the case with $\{v_5, t\}$ in Fig. 4c.

Algorithm 1 Unweaving Procedure

1: Let C be an empty multiset. For all arcs (u, v) in P_1, add (u, v) to C.
2: For all arcs (u, v) in P_2, if $(v, u) \in C$ then remove (v, u) from C, else add (u, v) to C.
3: Replace each arc (u, v) in C with an undirected edge $\{u, v\}$.

Algorithm 2 Double Path Heuristic

1: Given an edge-weighted graph $G = (V, E)$, let $T = (V - \{s\})$ be the set of target vertices to check and let S be a shortest-path tree of G rooted at s. In addition, set the lower bound $\text{LB} = -\infty$ and the upper bound $\text{UB} = \infty$.
2: Select and remove a target vertex $t \in T$ and use S to determine the shortest s-t-path in G. Now use the methods of Bhandari to generate the shortest s-t-circuit C.
3: If $\text{LB} < L(C) \leq k$, then set $\underline{C} = C$ and $\text{LB} = L(\underline{C})$. If $k \leq L(C) < \text{UB}$, then set $\bar{C} = C$ and $\text{UB} = L(\bar{C})$.
4: If $\text{UB} = \text{LB}$ or $T = \emptyset$ then return $\underline{C}$ and $\bar{C}$; else go to Step 2.

Having reviewed the methods for producing an s-t-circuit, the overall double path heuristic is given in Algorithm 2. As shown, the idea is to take each vertex $t \in (V - \{s\})$ in turn and generate the shortest possible s-t-circuit. The best-observed solutions, $\underline{C}$ and $\bar{C}$, are then returned. In terms of complexity, the main expense occurs in Step 2 which, in the worst case, is executed $n - 1$ times. This step involves modifying the graph (an $\mathcal{O}(n)$ operation), determining the second s-t-path P_2 and then unweaving. Note that Dijkstra's algorithm is not suitable for computing P_2 because it cannot cope with graphs featuring negatively weighted edges. We, therefore, need to turn to more expensive alternatives such as the algorithms of Moore (1959) or Bellman and Ford (Cormen et al. 2009), which both feature a complexity of $\mathcal{O}(nm)$. This makes the overall complexity of the double-path heuristic $\mathcal{O}(n^2m)$, though we can consider this bound to be quite conservative because (a) the bounds of Bellman-Ford and Moore's algorithms are themselves known to be rather conservative (Sedgewick and Wayne 2011), and (b) further augmentations can be made to the algorithm that will shorten run times, as we now discuss.

3.1.1 Vertex filtering and heuristic selection

Run times of the double path heuristic can be shortened in two ways: first, by using information collected during a run to filter out members of T that cannot improve solution quality; second, by strategically selecting members of T that are more likely result in high-quality solutions being found earlier in a run. For the first point, consider the following theorem.

Theorem 1 *Let $G = (V, E)$ be an edge-weighted graph with no negative weights. In addition, let P_1 be the shortest s-t-path in G, and let C be the shortest s-t-circuit, determined using the methods of Bhandari (1999)).*

(i) If $u \in C$, then the shortest s-u-circuit in G has a length of at most $L(C)$.
(ii) If u is a descendant of t in the shortest-path tree rooted at s, then the length of any s-u-circuit in G is equal to or exceeds $L(C)$.

Proof Part (i) is trivial. If $u \in C$, then C also defines an s-u-circuit; hence an s-u-circuit of length $L(C)$ is known to exist. To prove Part (ii), let P_2 be the shortest t-u-path in G. The shortest s-u-path therefore has length $L(P_1)+L(P_2)$, where $L(P_2)$ is nonnegative. In addition, using the methods of Bhandari let P_1' and P_2' be the second paths generated from s to t and s to u respectively. We now need to show that $L(C) = L(P_1)+L(P_1') \leq L(P_1)+L(P_2)+L(P_2')$ or, equivalently, $L(P_1') \leq L(P_2)+L(P_2')$. To do this, assume the contrary, giving $L(P_1') > L(P_2)+L(P_2')$. This now implies that the shortest s-t-circuit has length $L(P_1) + L(P_2) + L(P_2')$, which is a contradiction. □

Theorem 1 implies that we can filter out members of T through the application of the following two rules. These should be applied between Steps 3 and 4 of the double path heuristic given in Algorithm 2. (Recall that, at this point in the algorithm, C is the shortest s-t-circuit in G).

1. If $L(C) \leq k$, then remove all vertices $u \in C$ from T. (All s-u-circuits will be equal or inferior in quality compared to C.)
2. If $L(C) \geq k$, then remove from T any descendants u of t in the shortest-path tree rooted at s. (The lengths of each s-u-circuit in G will equal or exceed $L(C)$.)

Instead of removing just one vertex from T in each iteration of the algorithm, these rules allow the removal of several vertices, thereby speeding up the algorithm while not compromising the quality of the solution produced.

A second strategy for improving the performance of the double path heuristic is to modify Step 2 of Algorithm 2 so that t is chosen according to some selection rule. We suggest three strategies here: furthest-first, where the vertex $t \in T$ furthest from the source s is selected; closest-first, where the $t \in T$ closest to s is selected; and the original random selection. Note that when using the furthest-first rule, the second filtering rule above will never be applied because descendants of the selected vertex t will already have been removed from T in previous iterations.

3.2 Local search heuristic

Our second algorithm for the GKCP is based on local search. Local search is a general-purpose methodology that seeks to identify high-quality solutions within a space of candidate solutions. It operates by moving from solution to solution within this space using a neighbourhood operator that makes small alterations to the current solution. This continues until a time limit is reached or until a solution of sufficient quality is found. Because the methods considered in this paper are intended to be fast, here our local search algorithm only accepts alterations that improve a solution. It also halts as soon as local optima are identified. While it would be simple to extend these methods to use metaheuristic frameworks such as simulated annealing or tabu search, this would also involve longer run times. Methods for forming initial solutions for this local search heuristic are discussed in Sects. 4.3 and 4.4.

To define a suitable neighbourhood operator for this problem, let $C = (s = u_1, u_2, \ldots, u_l = s)$ be a candidate solution, written as a sequence of vertices. The edges of C are therefore defined by the multiset $\{\{u_i, u_{i+1}\} : i \in \{1, 2, \ldots, l-1\}\}$. Recall that individual vertices can occur multiple times in C; on the other hand, edges of G can occur only once in the edge multiset except in cases where the edge is a bridge, in which case it can occur up to two times. The circuit shown in Fig. 4c, for example, is written as $C = (s, v_1, v_4, v_5, t, v_5, v_2, v_3, s)$ and has the edge multiset $\{\{s, v_1\}, \{v_1, v_4\}, \{v_4, v_5\}, \{v_5, t\}, \{t, v_5\}, \{v_5, v_2\}, \{v_2, v_3\}, \{v_3, s\}\}$.

An intuitive neighbourhood operator for this problem is to select two vertices $u_i, u_j \in C$, remove the current u_i-u_j-path in C, and replace it with a new u_i-u_j-path. If using shortest paths, the determination of a new path can be achieved in $\mathcal{O}(m + n \log n)$ time via Dijkstra's algorithm. Here, we use an extension of this operator that features the same asymptotic complexity. The idea is to take a single vertex $u_i \in C$ and produce a shortest-path tree (rooted at u_i) to all other vertices in C but without using any of the edges in C. The best option among these different shortest paths is then selected.

Recall that for this problem we maintain two solutions, $\underline{C}$ and $\bar{C}$, representing the best-observed solutions either side of the target k. The aim is to therefore increase the length of $\underline{C}$ and decrease the length of $\bar{C}$ while ensuring that $L(\underline{C}) \leq k \leq L(\bar{C})$. Our strategy is to run a local search procedure that alternates between these two tasks.

Algorithm 3 Local Search Iteration (Ascending)

1: Let G' be a copy of G with the edges of $\underline{C}$ removed. Also, let $best = L(\underline{C})$.
2: Choose a vertex $u_i \in \underline{C}$, and compute a shortest-path tree S in G' rooted at u_i.
3: For $j \in (\{1, \ldots, l\} - \{i\})$ do:

   a: Let C be the circuit formed by replacing the u_i-u_j-path in $\underline{C}$ with the u_i-u_j-path in S.
   b: If $best < L(C) \leq k$ then set $i' = i$, $j' = j$ and $best = L(C)$.
   c: If $k \leq L(C) < L(\bar{C})$ then set $\bar{C} = C$. (An improvement to $\bar{C}$ has been made.)

4: If $best > L(\underline{C})$ then replace the $u_{i'}$-$u_{j'}$-path in $\underline{C}$ with the $u_{i'}$-$u_{j'}$-path in S. (An improvement to $\underline{C}$ has been made.)

A single iteration of (ascending) local search using $\underline{C}$ is described by the pseudocode in Algorithm 3. As shown, this procedure seeks to lengthen $\underline{C}$ by replacing a section of it with a different path. It does this by performing the best move among the $l - 1$ different options given by the shortest-path tree S. If, during this process, a new solution C is observed that happens to be superior to $\bar{C}$, then $\bar{C}$ is also replaced, as described in Step 3c.

A single round of (ascending) local search is achieved by applying Algorithm 3 repeatedly until improvements are not available from any of the vertices $u_i \in \underline{C}$. A similar (descending) local search is then carried out on $\bar{C}$. This is achieved in the same way except that occurrences of $\underline{C}$ and $\bar{C}$ in the pseudocode are swapped, and the inequalities are reversed. These alternating phases of ascent and descent are repeated until neither solution is seen to improve, or until $L(\underline{C}) = L(\bar{C}) = k$.

Note that when applying the above neighbourhood operator, if $u_{i'}$ and $u_{j'}$ refer to the same vertex in G, then a subcircuit of the circuit will be deleted. Similarly,

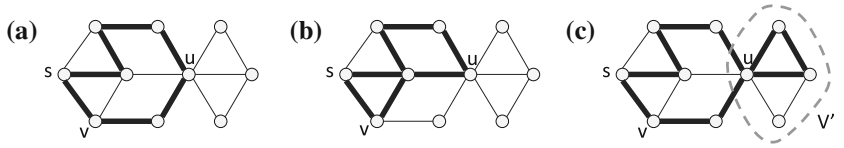

Fig. 5 Part **a** shows an example solution on a small graph. Part **b** shows how an additional subcircuit can be formed in this solution by using a different u-v-path. Part **c** shows how our second neighbourhood operator is able to form a new subcircuit to the right of the articulation point u by considering the graph induced by V'

applications of this neighbourhood operator can also introduce additional subcircuits, as demonstrated in Fig. 5a and b. On the other hand, there are also situations in which a solution will need to be lengthened through the addition of a subcircuit, but where this is not possible with the current operator due to the presence of articulation points in the graph. In Fig. 5a, for example, we see that the edges to the right of the articulation point u will never be added to the solution by this neighbourhood operator. An additional operator is therefore required.

Because it can only increase the length of a circuit, our second neighbourhood operator is applied to $\underline{C}$ only. It operates by selecting a vertex $u_i \in \underline{C}$ that is an articulation point in G. It then seeks to construct a u_i-circuit in the subgraph induced by the set V', where V' contains u_i plus all of the vertices that would be separated from $\underline{C}$ if u_i were to be removed from G. The task of forming an appropriate-length u_i-circuit in the graph induced by V' is, of course, a new (smaller) instance of the GKCP in which the target length is $k - L(\underline{C})$. In our case, we therefore apply the double path heuristic on this subgraph, before splicing the resultant circuit into $\underline{C}$. An example is provided in Fig. 5c.

Overall, our local search method operates by alternating between periods of ascending local search and descending local search as described above. Once neither of these is able to improve $\underline{C}$ and $\bar{C}$, the second neighbourhood operator is applied. Here, this is achieved by examining each articulation point in $\underline{C}$ in random order. If an improving move is found, this is then applied, and the algorithm returns to performing the alternating periods of local search. Otherwise, the process terminates, with $\underline{C}$ and $\bar{C}$ giving the final solutions of the algorithm.

4 Experimental analysis

In this section we analyse and compare the performance of our double path and local search heuristics. We also examine the scaling-up issues surrounding the use of Yen's algorithm, as described in Sect. 2. Our heuristics were implemented in C++, while Yen's algorithm was implemented in Java using publicly available code (Smock 2017). In all cases, graphs were stored using adjacency lists, and priority queues were used in conjunction with Dijkstra's algorithm. All trials were executed on 3.2 GHtz Windows 7 machines with 8 GB RAM. Source code and a full set of our results are available online at Lewis (2021). Additional charts, tables and usage instructions are also given in this paper's companion manuscript, available at Lewis and Cocoran (2021).

As noted in Sect. 3.1, the complexity of the double path heuristic is $\mathcal{O}(n^2m)$ due to the (maximum of) $n - 1$ applications of an $\mathcal{O}(nm)$ shortest path algorithm. In practice, however, we found that better run times could be achieved by replacing these shortest path algorithms with a version of Dijkstra's algorithm that is suitably modified to cope with negatively weighted edges. This algorithm operates by taking a graph $G = (V, E)$, a source vertex s, and a target vertex t. It then follows the steps given in Algorithm 4. In this pseudocode $L(v)$ is used to denote the length of the path between the source s and a vertex v, and $P(v)$ gives the vertex that precedes v in the shortest s-v-path. The method also halts as soon as the shortest s-t-path has been established. This modified version of Dijkstra's algorithm differs from the original in that vertices can be inserted and removed from the set X more than once; however, this also brings run times that are exponential in the worst case (Sedgewick and Wayne 2011). As an alternative, Bhandari (1999) has suggested a modified version of Moore's algorithm that halts as soon as the shortest s-t-path is identified. This features the more desirable complexity of $\mathcal{O}(nm)$. Despite this, in our experimentation, we still found that the modified Dijkstra's algorithm generally gave shorter run times. It is therefore used in all applications of the double path heuristic unless specified otherwise.

Algorithm 4 Modified Dijkstra's Algorithm

1: For all $v \in V$, set $L(v) = \infty$. Now let $X = \emptyset$ be the set of visited vertices, and let $L(s) = 0$.
2: Choose a vertex $u \in V$ such that (a) $u \notin X$, (b) $L(u) < \infty$, and (c) $L(u)$ is minimal among the available options. If no such vertex exists, or if $u = t$ then exit; otherwise, add u to X and go to Step 3.
3: For all neighbouring vertices v of u, if $L(u) + w(u, v) < L(v)$ then: (a) set $L(v) = L(u) + w(u, v)$, (b) set $P(v) = u$, and (c) remove v from X if present. Now return to Step 2.

As previously noted, in this paper we want to focus on graphs resembling real-world networks of roads and footpaths. However, to help analyse algorithmic behaviour, we also want to be able to alter their edge densities. We therefore started by looking at the central districts of five large cities, namely Amsterdam, Kolkata, London, Melbourne and New York. These were found to have approximately 400 vertices (intersections) per square km. We then generated a set of planar graphs that emulated these features. 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. Note that when roads and paths 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 road bridge, so the analogy is not exact.

To generate a planar graph, we started by taking a 10×10 km square and placing 40,000 vertices within it at random coordinates. A Delaunay triangulation was then generated from these vertices, and a subset of its edges was randomly selected to form a connected planar graph. Edge weights were then set to the Euclidean distances between endpoints, rounded to the nearest meter. To allow circuits in any direction, the source vertex was also placed at the centre of the square. For our experiments, we considered three types of planar graphs: sparse, medium, and dense, featuring 50,000,

Table 1 Features of the real-world street networks used in our tests (ordered by density)

City	Num. Vertices n	Num. Edges m	Density
London	70,315	92,794	0.000038
Melbourne	39,066	57,787	0.000076
Amsterdam	35,543	49,247	0.000078
New York	37,740	62,237	0.000087
Kolkata	28,667	38,222	0.000093

Fig. 6 12×12 km street networks of central London (left) and central Kolkata (right)

75,000 and 100,000 edges respectively. Twenty such graphs were generated in each case.

In addition to these planar graphs, we also tested our algorithms on the actual street networks of the above five cities. These were generated using the OSMnx library (Boeing 2017) using a 12×12 km bounding square positioned over the city centres. Our algorithms were then tested using source vertices within the central square km of these networks. Details and visualisations of these networks are shown in Table 1 and Fig. 6.

Finally, we also considered random graphs. These were generated by taking an empty graph on n vertices and then adding $\lfloor d\binom{n}{2} \rfloor$ randomly selected edges, where d is the desired density. The weight of each edge was set to a randomly selected value from the set $\{0, 1, \ldots, 10000\}$.

4.1 Performance of Yen's algorithm

Our first set of experiments examines the implications of using the approach of Willems et al. (2018) with our planar graphs. The bars in Fig. 7 show how, due to the preprocessing step described at the start of Sect. 3, the number of vertices in the resultant graphs increases for larger values of k. For the sparse graphs, we also see that significant

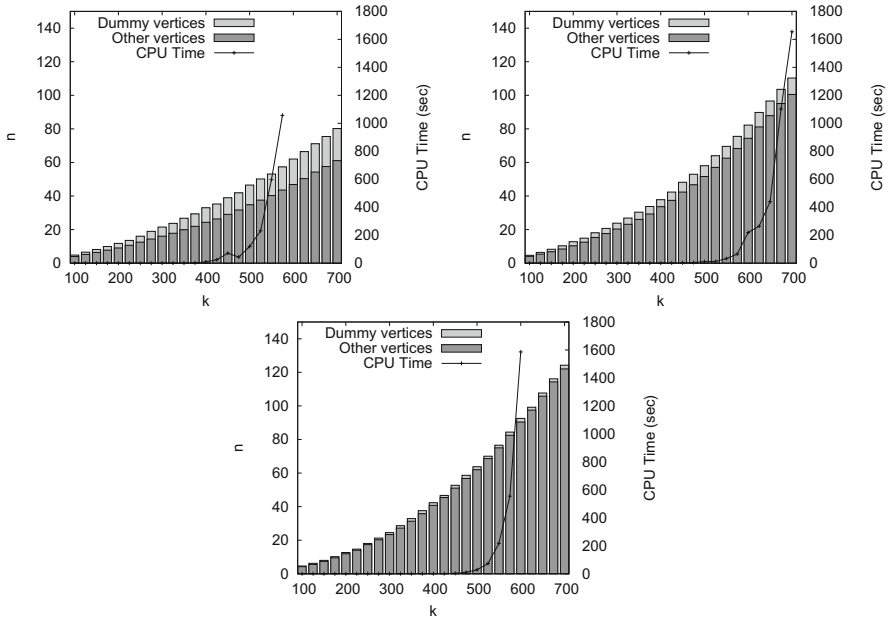

Fig. 7 Graph sizes and execution times of Yen's algorithm for various values of k using, respectively, sparse, medium and dense planar graphs. Each point in the charts is the mean across twenty problem instances

numbers of dummy vertices need to be added to eliminate bridges and articulation points.

For values of k up to around 500 meters, we see that the execution of Yen's algorithm is quite fast; however, beyond these values, the required run times increase rapidly due to the very large numbers of paths that need to be produced. Indeed for $k > 700$ meters, solutions were never found within our imposed 30-minute time limit, suggesting that this method is unsuitable for practical-sized problems.

4.2 Double path heuristic performance

We now assess the performance of the double path heuristic on our planar and real-world graphs. Figure 8 shows the accuracy of the heuristic on planar graphs in two ways: the *gap* in quality between the two returned solutions $\underline{C}$ and $\bar{C}$; and the *success rate*, calculated as the proportion of runs where solutions of length k are found.

For larger values of k we see that solution lengths are closer to the target. This is because the corresponding graphs are larger, meaning that more solutions are generated for the double path heuristic to choose between. For similar reasons, accuracy also seems to increase slightly with the denser graphs because the additional edges result in a greater number of vertices being within $k/2$ meters of the source.

On the other hand, increases in k (and therefore graph size) also give lengthier runs. In Fig. 9 we show the execution times corresponding to the results of Fig. 8. Here, five variants of the double path heuristic are presented: each of the three heuristics

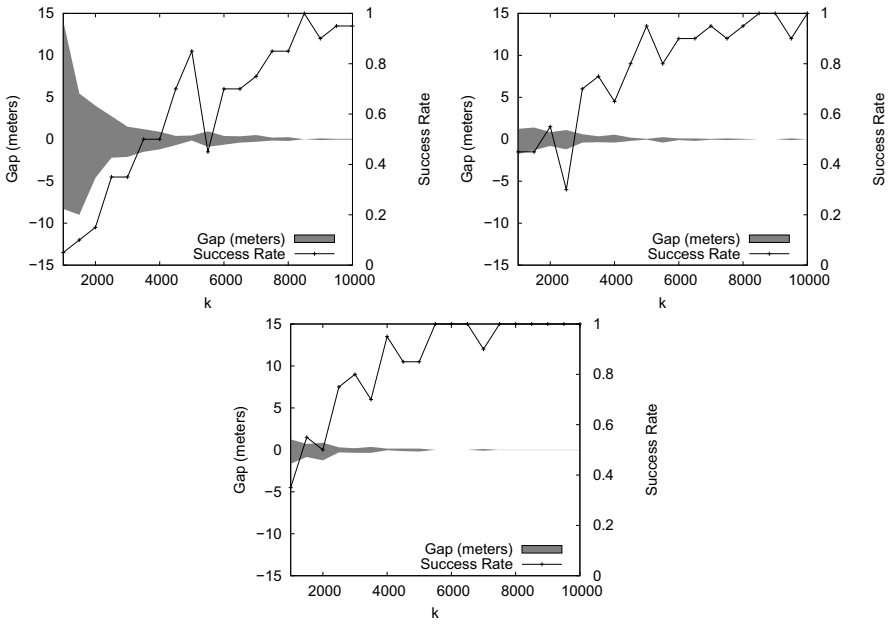

Fig. 8 The shaded areas show the gap (in meters) between the two solutions returned by the double path heuristic for differing values of k. The lines show the corresponding success rates. Each point is the mean across twenty problem instances for, respectively, sparse, medium, and dense planar graphs

given in Sect. 3.1.1; one using random selection with no vertex filtering; and one using random selection, no filtering, and Bhandari's $\mathcal{O}(nm)$ adaptation of Moore's algorithm for calculating the second shortest path P_2 (Bhandari 1999). To reduce noise in these timings, note that we also executed the double path heuristic until $T = \emptyset$ in all cases. That is, the algorithm did not halt early if a solution of length k was achieved.

Figure 9 demonstrates that the use of vertex filtering shortens run times. The quickest runs occur when this is used in conjunction with furthest-first and, for sparse graphs, random selection. With the furthest-first rule, short run times occur because, in the early stages of runs, it tends to produce circuits with many vertices. This allows larger numbers of elements to be removed from T. On the other hand, the closest-first rule is the least favourable because, in early iterations, the solutions it produces are too short and contain few vertices; consequently, less filtering takes place. The results also suggest that the modified version of Dijkstra's algorithm gives better run times than Moore's algorithm, despite its exponential worst-case complexity.

The differences between these five variants of the double path heuristic are more obvious in Fig. 10, where we show the times at which the best-observed solutions were found. (In many cases, these gaps were of size zero, and would therefore result in early termination.) This further demonstrates the superiority of using vertex filtering in conjunction with the furthest-first rule, where mean run times have fallen to less than ten seconds in all cases.

Table 2 shows the performance of the double path heuristic (using vertex filtering and the furthest-first rule) with our five chosen cities. Here, CPU times again indicate

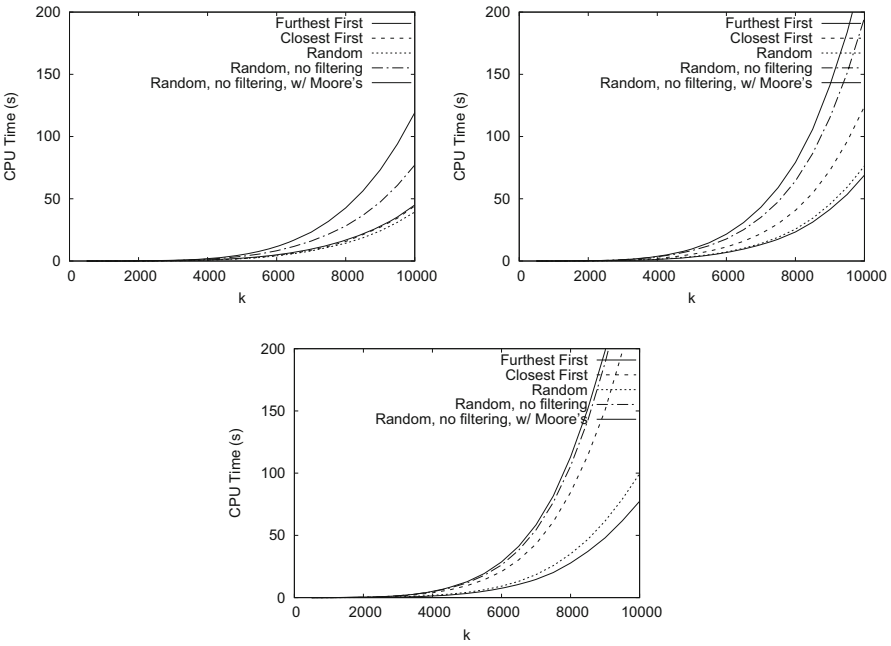

Fig. 9 Mean run times for differing variants of the double path heuristic using various values of k for (respectively) sparse, medium, and dense planar graphs. All points are the mean across twenty problem instances

entire runs; however, we note that when solutions of length k *were* achieved, this tended to occur in the early stages, so run times will be much shorter in these cases. The results show that the solutions returned by the double path heuristic are consistently within just a few meters of the target k. Patterns consistent with the previous figures are also evident, with accuracy generally increasing with denser graphs and larger values for k. Higher values of k and higher network densities also result in longer run times, with London being the most notable example.

Table 2 also shows that run times with these instances can be approximately halved by using the additional preprocessing step of removing leaf vertices (as discussed at the start of Sect. 3). However, the removal of these vertices also results in fewer candidate solutions being considered by the algorithm. This brings slightly less accurate solutions, as shown.

4.3 Local search heuristic performance

In this section, we compare the performance of the double path and local search heuristics. In all cases, the double path heuristic variant used vertex filtering together with the furthest-first selection rule.

Three different variants of the local search heuristic are considered here. The first takes as an initial solution the best solution returned by the double path heuristic. The second uses the s-t-circuit generated by Bhandari's method, where t is the vertex in

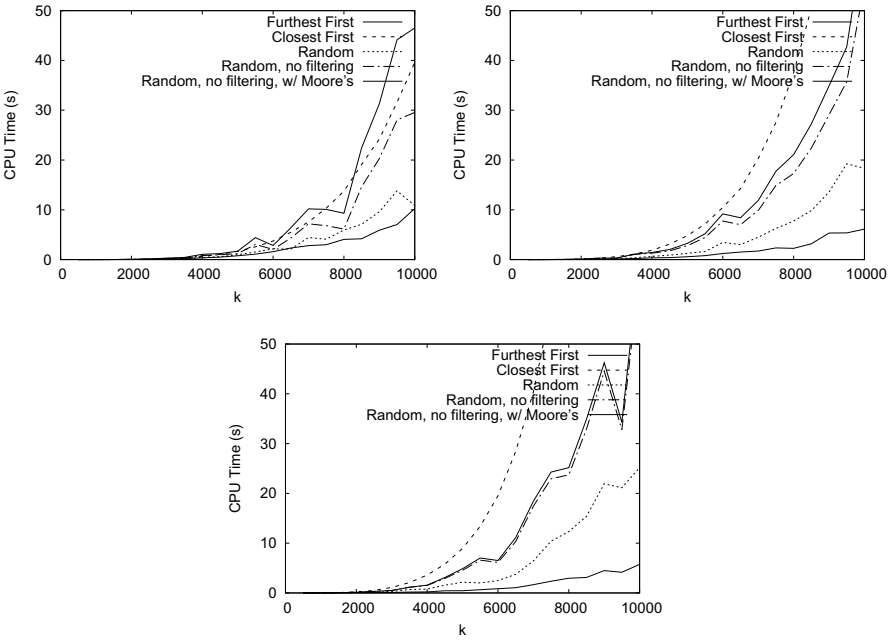

Fig. 10 Times at which the best gap was achieved for differing variants of the double path heuristic. Other experimental details are the same as those in Fig. 9

G whose distance from the source s is closest to but not exceeding $k/2$. The final method is identical to the previous except that, instead of generating a shortest-path tree in each iteration of the local search procedure, a breadth-first search (BFS) tree is used instead. Note that BFS trees are generated in $\mathcal{O}(n + m)$ time as opposed to the $\mathcal{O}(m + n \log n)$ time required by Dijkstra's algorithm. On the other hand, paths generated by BFS do not consider the weights of edges; instead, they simply minimise the *number of edges* in the paths. The changes in length caused by applications of the BFS neighbourhood operator therefore tend to vary more widely.

Figure 11 demonstrates the accuracy of these four algorithms on planar graphs. For all values of k we see that the addition of local search to the double path heuristic brings the best solutions on average; hence, local search is consistently able to make improvements on the solutions returned by this heuristic. The solutions returned by the other two variants of the local search heuristic are also mostly superior to the double path heuristic, the exception being for small values of k with sparse planar graphs.

The execution times of these runs are given in Fig. 12. In these cases, runs of the double path heuristic were permitted to halt if a solution of length k was generated. This allows more meaningful comparisons with the local search variants and brings slightly shorter run times than those seen in Fig. 9. We see that, despite providing much better solutions, the addition of local search to the double path heuristic brings a negligible increase in time. That said, it is also obvious that the other two variants of local search are much faster, with average run times taking less than one second, even for the largest graphs.

Table 2 Accuracy and speed of the double path heuristic on five cities. Each figure is a mean across 50 runs using randomly selected source vertices within 1 km of the city centre. CPU times (in seconds) are stated as the mean across 50 runs, plus/minus the standard deviation

City	$k = 1000$			$k = 5000$			$k = 10,000$		
	LB $- k$	UB $- k$	CPU Time (s)	LB $- k$	UB $- k$	CPU Time (s)	LB $- k$	UB $- k$	CPU Time (s)
London	−1.2	1.8	0.046 ± 0.016	−0.1	0.1	11.835 ± 0.867	0.0	0.0	193.484 ± 9.511
Melbourne	−1.4	2.1	0.031 ± 0.016	−0.1	0.1	8.285 ± 1.591	−0.2	0.2	63.800 ± 7.236
Amsterdam	−3.1	3.0	0.010 ± 0.003	−0.7	0.9	1.997 ± 0.264	−0.4	0.4	21.682 ± 2.009
New York	−5.7	4.9	0.004 ± 0.001	−0.4	0.5	2.116 ± 0.253	−0.2	0.2	25.934 ± 2.099
Kolkata	−6.1	9.2	0.002 ± 0.001	−1.3	1.0	0.976 ± 0.176	−0.5	0.5	13.038 ± 1.388
(Remove degree-1 vertices)									
London	−2.2	3.8	0.022 ± 0.008	−0.7	0.6	3.518 ± 0.324	−0.3	0.3	48.219 ± 3.425
Melbourne	−3.4	4.9	0.014 ± 0.008	−0.3	0.3	3.564 ± 1.376	−0.4	0.4	22.935 ± 5.215
Amsterdam	−5.7	5.3	0.006 ± 0.002	−1.1	1.2	0.893 ± 0.132	−0.6	0.8	7.924 ± 0.791
New York	−9.7	8.4	0.003 ± 0.001	−0.4	0.7	1.089 ± 0.193	−0.3	0.3	13.983 ± 1.534
Kolkata	−13.9	14.3	0.001 ± 0.001	−1.8	1.8	0.485 ± 0.107	−1.3	1.2	4.604 ± 0.622

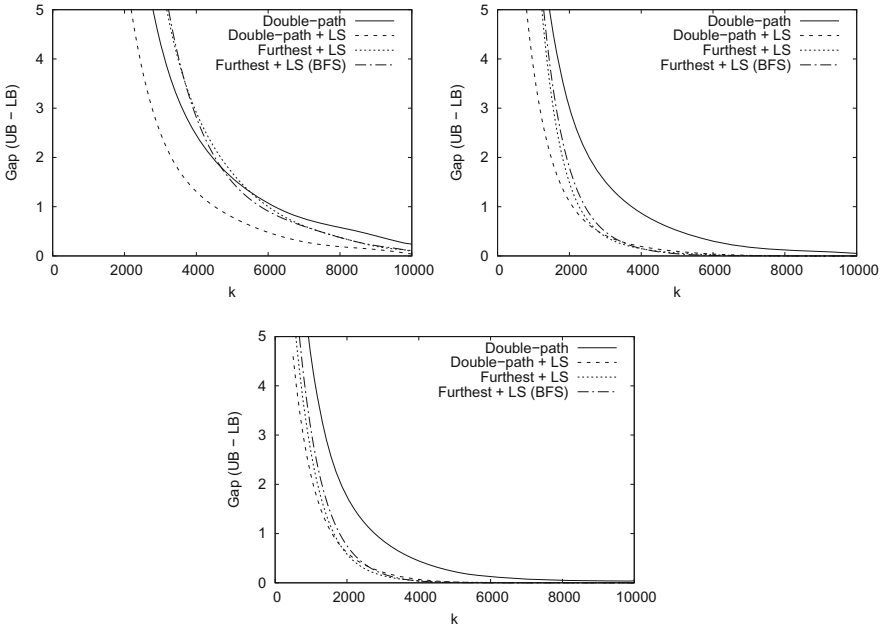

Fig. 11 Accuracy of the double path and three local search heuristics for differing values of k using (respectively) sparse, medium, and dense planar graphs. All points are the mean across 100 problem instances

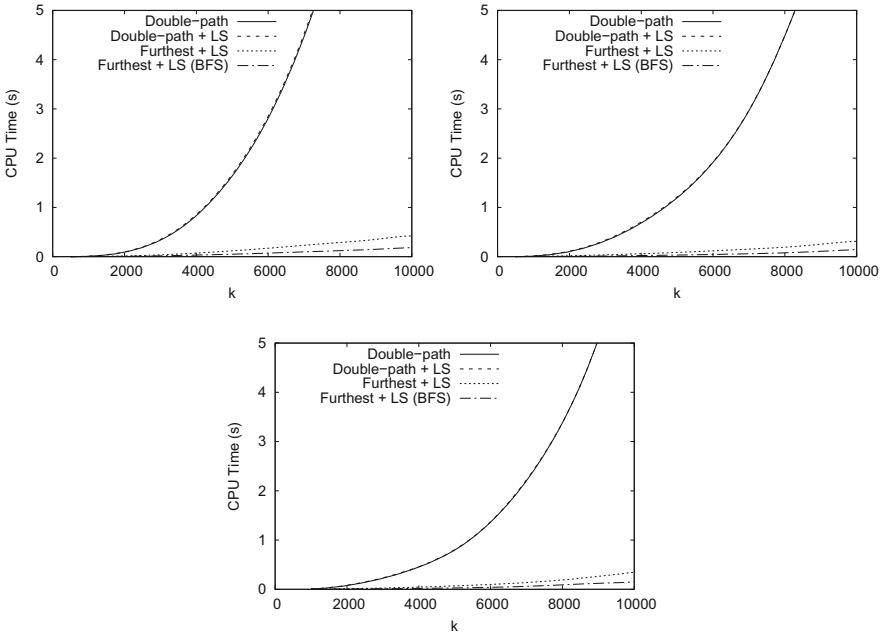

Fig. 12 Execution times of the trials given in Fig. 11 for (respectively) sparse, medium, and dense problem instances

Similar patterns can also be seen with the real-world street networks, as shown in Table 3. Here, the accuracy of the results are better than those of the double path heuristic seen earlier in Table 2. The run times of the latter two local search variants are also superior to the first, though for small values of k, its results are marginally less accurate.

4.4 Random graphs

We now consider random graphs. Although these seem to have fewer practical applications than our previous graphs, they are useful for stress testing our algorithms. Equation (1) seen earlier characterises the number of paths between pairs of vertices in random graphs. In our case, the (uniformly distributed) random assignment of weights to these edges means that, in all but the sparsest of graphs, very short paths are likely to exist between all pairs of vertices. This means that vertices are nearly always within $k/2$ units from the source, rendering the associated preprocessing step redundant. For larger values of k this also makes our double path heuristic unsuitable, because the circuits it produces are much too short. We therefore only consider the local search heuristic here.

In our trials, initial solutions were produced by selecting a random vertex $t \in (V - \{s\})$ and then forming an s-t-circuit using Bhandari's methods. Because this uses shortest paths, these are usually too short and need to be lengthened by the local search procedure. In initial tests, we found that, when using shortest-path trees with our first neighbourhood operator, this lengthening took considerable time because each accepted move would only increase the length of the circuit by a small amount, meaning that long series of moves were required to gain high-quality solutions. A better alternative is to therefore use BFS trees with this neighbourhood operator because the proposed changes in solution length are subject to fluctuate more widely, meaning fewer changes will be needed overall.

Figure 13 shows the accuracy of solutions using random graphs with 2000 vertices, densities ranging from 0.0 to 1.0, and values of k up to 100,000. Two algorithm variants are considered: one that uses BFS with the local search heuristic as described, and one that also applies a second stage of local search using shortest-path trees once the BFS stage has reached a local optimum. The rationale for using this second stage is that it will make the final refinements to solutions using an operator that brings small changes in length. A selection of these results are also tabulated in Table 4, and the corresponding execution times are shown in Fig. 14.

Figure 13 shows that this refinement stage does indeed improve the quality of the solutions produced, though this comes at the expense of slightly longer run times. We also see that lengthier runs occur with high values of k and high densities. This is because higher k's require more steps of local search, and because the larger number of edges in these graphs increases the run times of BFS and Dijkstra's algorithm. That said, all mean times reported in these figures are less than ten seconds.

Table 3 Accuracy and speed of the LS heuristic on five cities. Each figure is a mean across 50 runs using a randomly selected source vertex within 1 km of the city centre. CPU times (in seconds) are stated as the mean across 50 runs, plus/minus the standard deviation

City	$k = 1000$			$k = 5000$			$k = 10,000$		
	LB	UB	CPU Time (s)	LB	UB	CPU Time (s)	LB	UB	CPU Time (s)
Double Path + LS									
London	−0.4	1.0	0.034 ± 0.020	0.0	0.0	2.947 ± 3.071	0.0	0.0	23.795 ± 12.616
Melbourne	−0.7	0.6	0.014 ± 0.016	0.0	0.0	2.080 ± 2.586	0.0	0.0	13.234 ± 20.040
Amsterdam	−0.8	0.7	0.010 ± 0.005	0.0	0.0	0.983 ± 0.763	0.0	0.0	8.913 ± 8.031
New York	−2.1	1.5	0.005 ± 0.003	0.0	0.0	0.830 ± 0.862	0.0	0.0	7.708 ± 8.425
Kolkata	−4.3	4.6	0.003 ± 0.001	−0.2	0.2	0.622 ± 0.429	0.0	0.0	6.209 ± 5.448
Furthest + LS									
London	−0.8	0.5	0.010 ± 0.007	0.0	0.0	0.150 ± 0.169	0.0	0.0	0.641 ± 1.004
Melbourne	−0.9	2.4	0.006 ± 0.005	0.0	0.0	0.129 ± 0.130	0.0	0.0	0.676 ± 1.015
Amsterdam	−1.2	1.3	0.004 ± 0.002	0.0	0.0	0.071 ± 0.074	0.0	0.0	0.217 ± 0.212
New-York	−8.6	2.3	0.003 ± 0.002	0.0	0.0	0.061 ± 0.063	0.0	0.0	0.146 ± 0.113
Kolkata	−16.1	11.6	0.002 ± 0.001	−0.1	0.1	0.060 ± 0.058	−0.1	0.1	0.246 ± 0.248
Furthest + LS using BFS									
London	−0.9	0.5	0.004 ± 0.003	0.0	0.0	0.053 ± 0.046	0.0	0.0	0.254 ± 0.234
Melbourne	−0.7	2.0	0.003 ± 0.002	0.0	0.0	0.044 ± 0.041	0.0	0.0	0.235 ± 0.324
Amsterdam	−1.9	1.1	0.002 ± 0.001	0.0	0.0	0.023 ± 0.017	0.0	0.0	0.084 ± 0.092
New-York	−5.2	2.7	0.002 ± 0.001	0.0	0.0	0.022 ± 0.019	0.0	0.0	0.076 ± 0.048
Kolkata	−15.4	13.5	0.001 ± 0.001	−0.2	0.2	0.023 ± 0.020	−0.1	0.1	0.088 ± 0.074

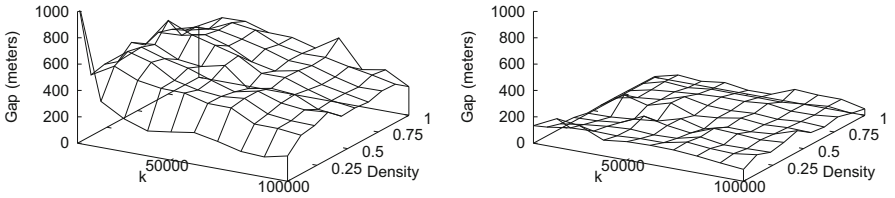

Fig. 13 Accuracy of the local search algorithms using random graphs of varying densities and different values of k. Accuracy is measured using the difference between the lower and upper bound, averaged across 20 instances. The first chart shows the local search algorithm based on BFS; the second shows the algorithm augmented with a second step of local search using shortest paths

Table 4 Selection of the results shown in Fig. 13. The left half of the table shows the local search algorithm based on BFS; the right shows the algorithm augmented with a second step of local search using shortest paths

k	Local Search (BFS)				Local Search (BFS + SP)			
	$d = 0.25$	0.5	0.75	1.0	0.25	0.5	0.75	1.0
1000	974.35	725.90	526.00	540.40	4.95	3.90	5.45	0.00
25000	629.05	562.40	416.00	403.35	23.40	54.10	2.90	15.00
50000	397.15	459.40	400.90	430.85	41.95	46.60	30.80	29.45
75000	322.30	382.30	449.30	420.25	65.50	140.50	138.85	152.20
100000	303.80	372.85	314.40	217.65	101.05	135.00	119.25	49.25

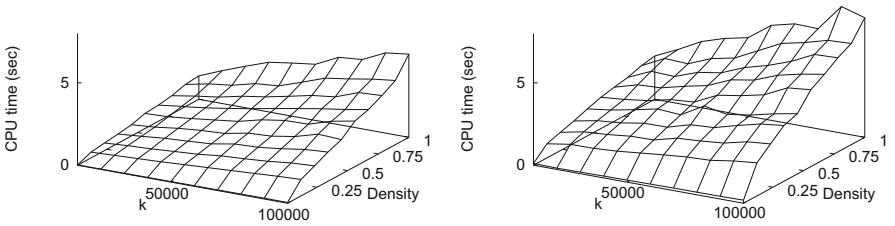

Fig. 14 Mean run times of the local search algorithms using random graphs of varying densities and different values of k. Other details are the same as Fig. 13

5 Adaptation to cycles

In this section, we now describe how our double path and local search heuristics can be adapted to produce *cycles* instead of circuits (refer to Definition 2). For the GKCP, recall that it is sometimes necessary to allow bridging edges to occur twice in a solution. Similarly, for cycles it is also necessary for articulation points to sometimes occur more than once in a solution. Mechanisms that allow this must therefore be built into our methods.

To form a cycle, the double path heuristic can be applied in the same way as before. The only difference lies in the way that, given a shortest s-t-path P_1, the graph is modified for the generation of P_2. Figure 15 illustrates this graph modification process using the graph and path from Fig. 4a as an example. As shown, the process involves

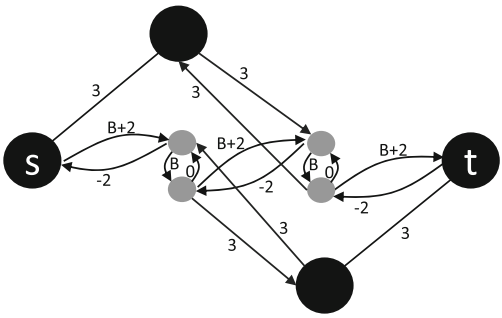

Fig. 15 Illustration of how graphs are modified to produce the shortest pair of minimally-vertex disjoint paths. Here, B assumes the same value as described in Sect. 3.1

splitting each internal vertex of P_1 into two, and then adding and adjusting edges incident to these vertices. Once P_1 and P_2 have been unwoven in the same fashion as before, the result is the shortest possible s-t-cycle in G. This "cycle" may contain repeated vertices and edges, but only if they are articulation points or bridges relative to s and t in the original graph G. Note that because cycles are special cases of circuits, Theorem 1 also holds, meaning that vertex filtering can be applied in the same way as before.

For the local search heuristic, the only modification required is in Step 3 of the algorithm given in Sect. 3.2. Now, before evaluating the cost of replacing the u_i-u_j-path in C with the u_i-u_j-path from S, it is first necessary to check whether this change will result in multiple occurrences of an inappropriate vertex in a solution. This is done by scanning the internal vertices of the u_i-u_j-path in S and checking to see if they are already present in C. If this is true for a vertex that is not an articulation point in G, then the move is immediately rejected as it would result in an illegal solution.

For space reasons, charts and tables showing the results of our experiments with cycles are not given here; instead, we refer the reader to this paper's companion manuscript for a full listing (Lewis and Cocoran 2021). In summary, the differences in performance between circuits and cycles with the double path heuristic were negligible, with virtually identical patterns to those reported above. For the local search heuristic, the solution space of cycles is a subset of that of circuits, leading to lower-quality solutions in general; however, these differences were again small, with average differences being less than two meters for our planar graphs with $k > 1000$. Due to the extra checks required by the neighbourhood operator, run times with cycles were also slightly higher, ranging from approximately 20% increases in time with $k = 1000$ to approximately 100% increases with $k = 10,000$.

6 Conclusions and further work

This paper has proposed two fast-acting heuristics for the $\mathcal{NP}$-hard generalised k circuit problem (GKCP). With minimal adjustments, these methods can also be applied to the more restricted problem of finding cycles of length k. We have seen that these heuristics regularly produce solutions of the required length. When this is not achieved, the lengths are usually within a small number of meters from the target.

Fig. 16 Ten km solutions from central London (left) and Kolkata (right) determined using the double path heuristic

There are several avenues for further research. The first of these concerns the aesthetics of a solution. Figure 16, for example, shows two 10 km solutions generated using our double path heuristic. While these are quite regular in shape, the way that this heuristic uses pairs of shortest paths also means that solutions can be rather elongated. On the other hand, our local search algorithm can sometimes result in solutions that look illogical or have many turns, particularly if many neighbourhood moves are performed from the initial solution. One option in this regard is to introduce a term into the objective function that measures the visual appeal of a solution. This could involve measuring a solution's similarity to a convex polygon (Arkin et al. 1991), and/or avoiding routes with too many sharp turns. As noted by Rossit et al. (2019), our understanding of visual attractiveness in routing problems is still rather vague, however.

Several other practical factors may also influence solution viability. These can include:

- Avoiding undesirable streets, such as those with high traffic densities, inadequate footpaths, steep hills, or too much street furniture;
- Prioritising certain streets such as those popular with runners, or those that travel through parkland or along the coast;
- Ensuring that users are not asked to stray too far from their starting point;
- Making routes that are easy to memorise.

Some of these requirements might be incorporated by augmenting the objective function and/or by modifying the weights of specific edges in the graph. One relevant example here is the work of Hannah et al. (2018), who use additional weights and graph expansion techniques to generate short paths for pedestrians that avoid too many road crossings. Similar techniques have also been used by Nunes et al. (2020) for creating safe paths for cyclists.

A further practical issue involves the incorporation of other compulsory locations in a route. For example, a user may want to leave their home, visit a supermarket, a pharmacy, and a cafe (in some order) and then return home while still travelling a specific distance. This situation can be considered a generalisation of the GKCP in which there are multiple source vertices $s_1, s_2, \ldots, s_l$. There are also connections with the travelling salesman problem here; indeed, an optimal TSP tour that visits all

vertices in $\{s_1, \ldots, s_l\}$ will provide a lower bound on the length of any solution for this problem. Our suggested methods would need to be modified here to ensure that all vertices in $\{s_1, \ldots, s_l\}$ are always present in a solution.

References

Alon, N., Yuster, R., Zwick, U.: Color-coding. J. ACM **42**(4), 844–856 (1995)
Arkin, E., Chew, L., Huttenlocher, D., Kedem, K., Mitchell, J.: An efficiently computable metric for comparing polygonal shapes. IEEE Trans. Pattern Anal. Mach. Intell. **13**(3), 209–216 (1991)
Basagni, S., Bruschi, D., Ravasio, S.: On the difficulty of finding walks of length k. Theor. Inform. Appl. **31**(5), 429–435 (1997)
Bhandari, R.: Survivable networks. Kluwer Academic Publishers, Netherlands (1999)
Boeing, G.: OSMnx: new methods for acquiring, constructing, analyzing, and visualizing complex street networks. Comput. Environ. Urban Syst. **65**, 126–139 (2017)
Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to algorithms. The MIT Press, Cambridge (2009)
Davie, A., Stothers, A.: Improved bound for complexity of matrix multiplication. Proc. Royal Soc. Edinburgh **143**(2), 351–369 (2013)
Dinitz, Y.: Algorithm for solution of a problem of maximum flow in a network with power estimation. Doklady Akademii Nauk SSSR **11**, 1277–1280 (1970)
Duckham, M., Kulik, L.: "Simplest" paths: Automated route selection for navigation. In International Conference on Spatial Information Theory, volume 2825, pages 169–185, 05 (2003)
Eppstein, D.: Finding the k shortest paths. SIAM J. Comput. **28**(2), 652–673 (1998)
Hannah, C., Spasić, I., Corcoran, P.: A computational model of pedestrian road safety: the long way round is the safe way home. Accid. Anal. Prev. **121**, 347–357 (2018)
Hierholzer, C.: Ueber die möglichkeit, einen linienzug ohne wiederholung und ohne unterbrechung zu umfahren. Mathematische Annalen (in German) **6**, 30–32 (1873)
Johnson, D.: Finding all the elementary circuits of a directed graph. Siam J. Comput. **4**(1), 77–84 (1975)
Lewis, R.: A heuristic algorithm for finding attractive fixed-length circuits in street maps In computational logistics number 12433 in Lecture notes in computer science. Springer, Berlin (2020)
Lewis, R.: Source code and results. http://rhydlewis.eu/resources/kcircuit.zip, (2021) Accessed 2021-09-31
Lewis, R., Cocoran, P.: Additional materials. http://rhydlewis.eu/resources/kcircuita.pdf, (2021) Accessed 2021-11-01
Moore, E.: The shortest path through the maze. In Proceedings of the International Symposium on the Theory of Switching, 1957, Part II, pages 285–292. Harvard University Press, (1959)
Nunes, P., Moura, A., Santos, J.: Evolutionary approach for the multi-objective bike routing problem. In Computational Logisitics, volume 12433 of Lecture Notes in Computer Science, pages 311–325. Springer, (2020)
Roberts, B., Kroese, D.: Estimating the number of s-t paths in a graph. J. Graph Algorithms Appl. **11**(1), 195–214 (2007)
Rossit, D., Vigo, D., Tohme, F., Frutos, M.: Visual attractiveness in routing problems: a review. Comput. Op. Res. **103**, 13–34 (2019)
Sedgewick, R., Wayne, K.: Algorithms. Pearson Education, 4th edition, isbn: 9780 321 573513 (2011)
Skiena, S.: Implementing discrete mathematics: Combinatorics and Graph Theory with Mathematica, chapter Eulerian Cycles, pages 192–196. Addison-Wesley, Reading, MA., (1990)

Smock, B.: K-shortest-paths. https://github.com/bsmock/k-shortest-paths, (2017) Accessed 2021-09-31
Suurballe, J.: Disjoint paths in a network. Networks **4**, 125–145 (1974)
Willems, D., Zehner, O., Ruzika, S.: On a technique for finding running tracks of specific length in a road network. In N. Kliewer, J. Ehmke, and R. Borndörfer, editors, Operations Research Proceedings 2017, pages 333–338, Cham, Springer International Publishing (2018)
Yen, J.: Finding the K shortest loopless paths in a network. Manag. Sci. **17**(11), 661–786 (1971)