SN Computer Science (2024) 5:868
https://doi.org/10.1007/s42979-024-03223-3

ORIGINAL RESEARCH

Fast Algorithms for Computing Fixed-Length Round Trips in Real-World Street Networks

Rhyd Lewis[1] · Padraig Corcoran[2]

Received: 1 March 2024 / Accepted: 11 August 2024
© The Author(s) 2024

Abstract
This paper proposes and evaluates algorithms for calculating round trips of a prescribed length on directed street networks. This problem has several real-world applications, such as designing jogging routes and cycling tours. In this work, we focus specifically on methods that avoid the need to download, process, and store large map databases. At the same time, we aim for our methods to be fast, accurate, and capable of handling a wide range of prescribed distances, from just a few meters to many kilometres. To achieve this, our overall strategy involves using a small number of calls to a suitable online mapping service to collect relevant structural information for the problem at hand. All remaining computations are then performed locally on the client. Empirically, we demonstrate that our suggested techniques outperform existing open-source algorithms in terms of both accuracy and runtime requirements. Our most successful approach is based on multi-objective local search, utilizing specialized neighbourhood operators that exploit the underlying graph-theoretical properties of this problem, resulting in runtimes of around 2–3 s on a typical desktop computer.

Keywords Round trip planning · Multiobjective optimisation · Edge computing · Combinatorial optimisation · Graph theory

Introduction

This paper considers the problem of constructing fixed-length routes that start and end at the same location on a map. This task has several practical applications including planning a jogging route in an unfamiliar location, organising a cycling tour, or determining a walking route that allows a user to complete their daily steps target.

In the absence of any restrictions, routes of a prescribed length are easy to create. For example, we may choose to simply travel up and down the same street repeatedly until the required distance is covered. Similarly, we could also perform laps of a small locality. In this work, however, we will consider methods of producing routes that avoid repetition. That is, where possible, users should be discouraged from having to travel along a street or footpath more than once. We will call such routes "round trips", and give a formal definition in "Problem Definition and Complexity". Two real-world examples are depicted in Fig. 1.

A common approach for producing round trips in street networks involves using online point-to-point routing services (such as Google Maps) and asking users to manually specify a series of waypoints, which are then linked by a sequence of paths. This forms the basis of the online Strava Routes service, which also allows users to specify preferences for paths involving hills, dirt tracks, and popular exercise trails. Similar schemes are also used by the popular online tools Komoot [19] and BRouter [12]. Typically, these approaches link waypoints using *shortest* paths, helping the solutions to "feel natural to the user, with no unnecessary detours" [1]. The shortest paths between two waypoints can also be calculated quickly on large maps, often in sub-linear time, using variants of the A* and Dijkstra's algorithms [4].

A feature of the above approach is that it is the users' responsibility to choose the appropriate waypoints; consequently, a trial-and-error approach may be required to achieve a round trip of the correct length. In the past five

✉ Rhyd Lewis
LewisR9@cardiff.ac.uk

Padraig 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

Published online: 09 September 2024

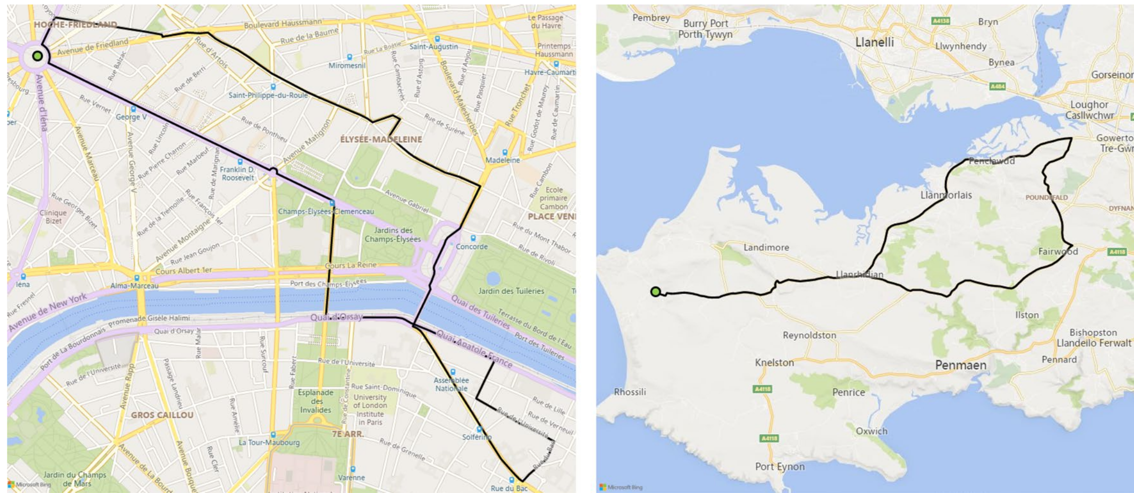


Fig. 1 Example round trips of length ≈ 10 km (left) and ≈ 50 km (right). The left figure starts and ends at the Arc de Triomphe in central Paris; the right figure starts at the western end of the Gower Peninsula, Wales. Due to the lack of alternative roads in the latter, the proposed round trip involves having to travel along some streets more than once

years or so, several tools have been made available that seek to automate this process. In April 2020, for example, Strava introduced a new feature for its paying subscribers that automatically produces round trips close to a desired length from a user-selected starting point [22]. Similar tools are now also available with Garmin Connect [13] and on websites such as RouteLoops [21] and PlotARoute [20]. As commercial enterprises, the algorithms used to produce these round trips are not in the public domain; instead, users send their input via a web request, with solutions then being calculated on the server side.

An open-source heuristic for producing round trips of a prescribed length is provided by the cloud-based application programming interface (API) OpenRouteService [14], using their "round_trip" feature. The basic idea in this algorithm is to generate an n-sided polygon that is roughly regular in shape and whose perimeter is approximately equal to the desired length. The initial bearing of the polygon is selected at random. Each vertex in this polygon is then repositioned over a street or at an intersection, and these are used as intermediate waypoints in the final round trip. When generating paths between these waypoints, efforts are also made to try to avoid using the same street segment more than once. Further details of this approach are given in "Initial Comparison and Setup".

Another website that contains tools for automatically producing round trips is TrailRouter [15]. In addition to producing fixed-length round trips, this tool also seeks to maximise the amount of green space encountered along a route. The basic version of this algorithm operates similarly to OpenRouteService's, using $n = 4$ to produce eight separate round trips with initial bearings of $0°, 45°, 90°, \ldots, 315°$. A second method is also used in which waypoints are placed on the perimeter of green spaces close to the starting point. In both cases, these waypoints are then linked using shortest paths as before. Weighting coefficients are also specified for different street segments to discourage their repetition and encourage paths that travel through green areas.

Note that all of these existing examples require significant resources to store and process the map database. OpenRouteService and TrailRouter, for example, use their servers to store the entire world map from OpenStreetMap, which contains information on nearly one billion ways [16]. The advantage of doing this is that various server-side actions can be performed that are not usually available on cloud-based APIs. OpenRouteService, for example, uses its map database in conjunction with the open-source Java-based routing engine GraphHopper to "snap" GPS coordinates to the nearest location on a street segment. GraphHopper also contains functionality for adding user-defined weights on street segments, allowing the production of paths that avoid using street segments seen in other parts of the round trip. TrailRouter similarly makes use of GraphHopper and also performs large amounts of server-side preprocessing on the map database to provide "green ratings" for each street segment. Street segments with higher green ratings are then preferred by GraphHopper's path-finding algorithms.

Related Works

From a research perspective, work in the area of round-trip production is somewhat limited in the literature. Partially related is the 1975 paper of Johnson [24], which reports an algorithm for enumerating all cycles in a graph. In theory, this method could be employed to generate a set of cycles

(round trips) on a street network, with the user then selecting a member of this set that contains the start/end point and whose length is close to the desired target length. The algorithm of Yen [35] might also be used for similar purposes by producing solutions one-by-one in increasing order of length until the target length has been reached (see [27], for example). In general, however, the number of cycles in a graph is known to grow at an exponential rate in relation to network size [10], making run times of these algorithms infeasible in most cases. As we will see in "Paper Aims and Contributions", the exclusive use of cycles in this problem is also usually unsuitable in real-world applications involving street networks. More recently, Chalupa et al. [7] have presented integer programming formulations for the problem of finding the longest cycles in unweighted networks, though similar scaling-up issues to the previous approaches are still apparent.

A heuristic for computing jogging routes on maps was also previously proposed by Gemsa et al. [9]. In their work, street networks are represented as undirected planar embeddings and a solution is built by iteratively adding the boundary edges of different faces until the desired length is reached. A second method similar to the round_trip feature of OpenRouteService is also proposed that produces solutions between a sequence of three or four waypoints, which their heuristic calculates automatically. Solutions are also evaluated according to the amount of repetition, the use of "nice" edges (such as those occurring in green areas), and the number of sharp turns.

In more recent work [26], the authors of this current paper describe two further heuristics for finding fixed-length round trips in undirected street networks. The first operates by constructing a pair of paths between the starting point s and a suitable street intersection t. These s-t-paths are then joined to form a round trip, with the aim being to identify the best option for t. In their heuristic, the authors use the algorithms of [5] to compute pairs of short edge-distinct (or vertex-distinct) s-t-paths. Other approaches might also be used, however, such as those used for finding alternative point-to-point routes in road networks [1, 29]. The second heuristic proposed in [26] is based on local search. In general, this was found to be faster and more accurate than their first heuristic, though its solutions often contained small detours that made them somewhat haphazard in appearance.

Note that the methods in [9, 26] only operate on undirected graphs, making them unsuitable in applications involving one-way streets and unidirectional footpaths. As with the other methods considered in this section, they also depend on having access to the entire street network during execution. In the experiments of [26] this was achieved by using the Python library OSMNX [6] to download and process all street segments within $k/2$ metres of the given starting point (where k is the desired length of the round trip). For higher values of k, though, this was seen to lead to long download times and large datasets. Using $k = 20{,}000$ m, for example, a location like central London was seen to involve over 90,000 street segments and 70,000 intersections. Similarly, all of the reported experiments of [9] were conducted on a single, locally stored map of Karlsruhe, Germany, which comprised approximately 120,000 street segments and 100,000 intersections.

Paper Aims and Contributions

Our aim in the remainder of this paper is to develop fast and accurate heuristics that produce round trips in *directed* street networks. We also want these methods to produce solutions from any starting point on the Earth's street network, but without needing to download and/or store large map databases. They should also be able to cope with a wide range of prescribed distances, from just a few metres to many kilometres, but still operate within timescales that are consistent with online point-to-point routing services (that is, typically a few seconds). To do this, our proposed methods will use very limited numbers of strategically chosen requests to existing cloud-based mapping services. The information collected in these requests will then be used in conjunction with bespoke graph-theoretic operators to strategically build a pool of candidate round trips. Outside of the requests, all computations, such as processes for improving and combining these round trips, are performed on the client.

Our proposed strategy has three main advantages. First, unlike existing methods, it avoids the need of having to host and maintain large databases of map data. Second, it also allows the possibility of using commercial mapping services such as Google Maps which, unlike OpenStreetMaps, can provide live data on traffic conditions and road closures. Finally, adjustments for different modes of travel such as walking, cycling, and driving, can be made by simply changing the parameters of the online requests. The information returned from these requests can then be used to provide the user with information on the elevation, surface type and greenery ratings of the proposed solutions, as demonstrated in " Conclusions and Further Work".

Our main contributions are summarised as follows:

1. The paper contains an examination of the computational issues surrounding the production of round trips in street networks, particularly looking at the graph-theoretical features of the problem and its underlying computational complexity. (See "Problem Definition and Complexity".)
2. A new heuristic, the isochrone-polygon method, is proposed. Using just two or three API requests, this constructs a small pool of solutions that, on average, are superior to those returned by OpenRouteService's

"round_trip" feature. (See "The Isochrone-Polygon Method and " Initial Comparison and Setup".)
3. The design and testing of two contrasting methods for improving solutions returned by the isochrone-polygon method. The best of these uses multiobjective optimisation in conjunction with new graph-based neighbourhood operators to produce larger, improved pools of candidate solutions in short amounts of time. (See "Improving Solutions using Multiobjective Local Search", "Learning Good Routes using Regression" and "Method Comparison".)

All source code, problem instances, and results datasets used in this work can be downloaded from [28].

Problem Definition and Complexity

Let $G = (V, A, w)$ be an arc-weighted, directed graph with vertex set V, arc set A, and nonnegative weights $w(u, v)$ for each arc $(u, v) \in A$. Street networks can be represented by such graphs using the arcs for individual street segments and the vertices for street intersections and dead ends. Here, weights correspond to travel distances (in metres) between adjacent vertices.

In mapping applications, each vertex $v \in V$ is associated with a particular GPS coordinate, denoted by $GPS(v)$. The above model is also usually extended so that each arc $(u, v) \in A$ is defined by a series of GPS longitude/latitude coordinates $(x_1, y_1), (x_2, y_2), \ldots, (x_l, y_l)$, where $GPS(u) = (x_1, y_1)$ and $GPS(v) = (x_l, y_l)$. These coordinates define the curvature of each street segment (u, v) and can also be used to calculate $w(u, v)$ using:

$$w(u, v) = \sum_{i=1}^{l-1} d(x_i, y_i, x_{i+1}, y_{i+1}) \tag{1}$$

where $d(.)$ gives the geographical distance between two GPS coordinates (which can be reliably approximated using, for example, the haversine formula or Vincenty's formulae). Often, mapping applications also make use of vertices to represent these intermediate coordinates, meaning that if $(u, v) \in A$ and $(v, u) \in A$ then $w(u, v) = w(v, u)$. This also allows the vertices of a street map to be partitioned into two classes:

Definition 1 A vertex v is considered *intermediate* if either of the following applies:

(i) v has exactly one incoming arc (u_1, v), exactly one outgoing arc (v, u_2), and $u_1 \neq u_2$; or
(ii) v has exactly two incoming arcs, (u_1, v) and (u_2, v), exactly two outgoing arcs (v, u_1) and (v, u_2), and $u_1 \neq u_2$.

If neither of these cases applies, then v is classed as an *intersection* vertex.

If Case (i) of Definition 1 is satisfied, this means that v represents a coordinate occurring along a one-way street; if Case (ii) is satisfied, then v is a coordinate along a two-way street. In practice, it is possible to reduce the size of such graphs by *smoothing* the intermediate vertices. This is achieved as follows.

- For Case (i), the vertex v and arcs (u_1, v) and (v, u_2) are removed and replaced by the single arc (u_1, u_2), with $w(u_1, u_2)$ set to $w(u_1, v) + w(v, u_2)$.
- For Case (ii), the vertex v and arcs (u_1, v), (u_2, v), (v, u_1), and (v, u_2) are removed and replaced by the arcs (u_1, u_2) and (u_2, u_1), with $w(u_1, u_2)$ set to $w(u_1, v) + w(v, u_2)$ and $w(u_2, u_1)$ set to $w(u_2, v) + w(v, u_1)$.

A demonstration of this smoothing process is shown in Fig. 2. Note that when an intermediate vertex v satisfying Case (ii) is smoothed, this will prevent round trips from being able to feature an arc (u, v) followed immediately by the arc (v, u). On a street network, this would correspond to a person being asked to switch direction part-way along a two-way street segment, which may not be desirable (or possible) depending on the circumstances at hand.

Given our arc-weighted directed graph $G = (V, A, w)$, we now use the following terminology.

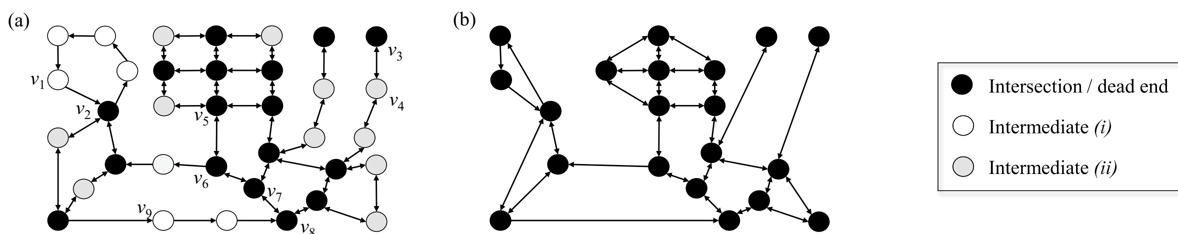


Fig. 2 Demonstration of how graphs (street networks) with intermediate vertices **a** can be smoothed to make a new, smaller graph (**b**). These smoothing actions are carried out by the Python library OSMNX, for example, [6]. Note that v_1 has not been smoothed here, despite satisfying Case (i) of Definition 1. This ensures that the structure of the one-way cycle is maintained

Definition 2 A *walk* is a series of pairwise adjacent vertices in G; a *trail* is a walk with no repeated arcs; and a *path* is a trail with no repeated arcs or vertices.

It is also useful to add a prefix u-v to these terms to denote a walk/trail/path that starts at vertex u and stops at v. In cases where $u = v$, the following terms can also be used.

Definition 3 A u-v-walk/trail/path is considered *closed* whenever $u = v$. Closed trails are also known as *circuits*; closed paths are known as *cycles*.

As mentioned earlier, our aim in this paper is to produce fixed-length round trips that avoid repetition and that start and end at the same predefined location. One possible problem definition is therefore as follows.

Definition 4 Let $G = (V, A, w)$ be a directed arc-weighted graph, $s \in V$ be our start and end vertex, $k \geq 0$ be our desired length, and let $w(u, v)$ denote a nonnegative weight (length) for each arc $(u, v) \in A$. The *k-length cycle problem* involves identifying a solution $S = (s = u_1, u_2, \dots, u_l = s)$ that is a *cycle*, and whose length $L(S) = \sum_{i=1}^{l-1} w(u_i, u_{i+1})$ minimises the cost function $f_1(S) = |k - L(S)|$.

Solutions under this problem definition naturally prohibit repetition, since cycles do not allow vertices or arcs to be used more than once. In applications with road networks, however, an insistence on using cycles is too strict because the underlying graphs are not necessarily biconnected. Examples of this can be seen in Fig. 2a. First, using $s = v_1$ we see that only one cycle is possible due to the presence of the articulation point v_2. Similarly, if $s = v_3$, then only solution (v_3, v_4, v_3) can be formed. A more flexible specification of our problem might therefore be defined as follows.

Definition 5 Given the same input as Definition 4, the *k-length circuit problem* involves identifying a solution $S = (s = u_1, u_2, \dots, u_l = s)$ that is a *circuit*, and whose length $L(S) = \sum_{i=1}^{l-1} w(u_i, u_{i+1})$ minimises the cost function $f_1(S) = |k - L(S)|$.

Here, the use of circuits instead of cycles permits vertices to be visited more than once, allowing a greater variety of solutions to be considered. Issues remain, however. First, this definition allows "out-and-back" solutions, such as $(v_5, v_6, v_7, v_8, v_7, v_6, v_5)$ in Fig. 2a which, while never using an arc more than once, certainly involves a lot of repetition. Secondly, this definition is still stricter than those used by the existing tools noted in "Introduction", which often allow street segments to be traversed more than once in the same direction. (In Fig. 2a, for example, this would occur when forming a round trip using the polygon defined by vertices (v_2, v_8, v_9).)

Our chosen problem definition, therefore, considers solutions as *closed walks* that start and end at vertex s. To avoid repetition while still allowing the necessary flexibility, a second cost function f_2 must now be used. To define this function, let $S = (s = u_1, u_2, \dots, u_l = s)$ be a closed walk, and let $E(S)$ be a multiset containing each occurrence of an arc in S, with its direction removed. For example, the solution $S = (v_5, v_6, v_7, v_8, v_7, v_6, v_5)$ shown in Fig. 2a leads to the multiset $E(S) = \{\{v_5, v_6\} : 2, \{v_6, v_7\} : 2, \{v_7, v_8\} : 2\}$, where numbers following colons represent the number of occurrences (multiplicity) of each undirected edge in $E(S)$. The function f_2 now measures the percentage of the overall round trip that involves travelling along street segments (in either direction) that have already been visited:

$$f_2(S) = 100 \times \frac{\sum_{\forall(\{u,v\}:x) \in E(S)} (x - 1) \cdot w(u, v)}{L(S)}. \quad (2)$$

Here, x defines the multiplicity of the edge $\{u, v\}$ in $E(S)$. Note that measuring the *percentage* of overlap is preferable to using the *total amount* of overlap because using the latter would encourage very short round trips. Our final problem definition is therefore:

Definition 6 Given the same input as Definition 4, the *k-length round trip (KRT) problem* involves identifying a solution $S = (s = u_1, u_2, \dots, u_l = s)$ that is a closed walk of length $L(S) = \sum_{i=1}^{l-1} w(u_i, u_{i+1})$, and that minimises the two cost functions, $f_1(S) = |k - L(S)|$ and $f_2(S)$.

Since f_1 and f_2 measure different characteristics, Definition 6 constitutes a *multiobjective* problem.

The three problem variants stated in Definitions 4–6 are all $\mathcal{NP}$-hard. Definition 4 generalises the $\mathcal{NP}$-hard problem of finding the *longest* cycle in an arc-weighted graph.[1] In turn, this latter problem generalises the $\mathcal{NP}$-hard problem of finding the longest cycle in an unweighted directed graph which, itself, is a generalisation of the $\mathcal{NP}$-hard Hamiltonian cycle problem in directed graphs. Similar reasoning can be applied with Definitions 5 and 6. Definition 5, which considers circuits, is $\mathcal{NP}$-hard since it generalises the $\mathcal{NP}$-hard problem of identifying a maximum-weight directed Eulerian subgraph in an arc-weighted directed graph [8, 33]. Basagni et al. [3] have also shown that the problem of calculating closed walks of length k is $\mathcal{NP}$-hard in arc-weighted graphs. This latter problem forms a special case of the KRT

[1] To see this, observe that Definition 4 with a sufficiently large value for k, such as $k \geq \sum_{(u,v) \in A} w(u, v)$, requires the identification of the longest cycle in a graph $G = (V, A)$.

problem of Definition 6 in which only f_1 is considered, also proving its $\mathcal{NP}$-hardness.

Note that the existing online methods for producing round trips reviewed in "Introduction" are all heuristic-based and, while featuring polynomial-time complexities, are unlikely to produce optimal solutions for most problem instances. This compromise is nearly always necessary with $\mathcal{NP}$-hard problems. These methods also operate under slightly different problem interpretations to ours. In particular, overlap is not measured using a cost function like f_2; instead, attempts are made to avoid overlap by equally spacing out the waypoints on the generated polygon and by using additional weights on arcs to penalise their repeated use in a solution. The effects of these differences are evaluated in "Initial Comparison and Setup".

The Isochrone-Polygon Method

In this section, we describe our initial algorithm for creating a pool of candidate solutions for the KRT problem. As stated earlier, this will operate by collecting information from a small number of calls to a suitable cloud-based mapping API. All remaining computation is then carried out on the client side.

Similarly to existing approaches, our method operates by constructing polygons on the map surface. The vertices of each polygon then act as a series of waypoints in a round trip. Here, however, adaptations are required because, unlike server-side implementations, most APIs do not provide convenient methods for quickly identifying whether arbitrary GPS coordinates are close to existing streets (and not, for example, in the middle of the sea). In addition, they do not usually provide the functionality for imposing weights on street segments to help avoid their repeated use in solutions.

To cope with these issues, our method begins by generating a $(k/2)$-metre *isochrone* I from the user-defined starting location s. Many popular mapping APIs allow the creation of such isochrones in a single request, including Bing Maps, Google Maps, Mapbox, ArcGIS, and OpenRouteService. The returned information is expressed as a list of GPS coordinates that define a polygon containing the parts of the map that are accessible from s by a path of length $k/2$ metres or less. Note that I can also be used to gauge the accessibility of the region surrounding s. For example, if the area of I is small compared to the area of the $(k/2)$-radius circle centred at s, this suggests the presence of nearby regions that, while being within $k/2$ metres (as the crow flies), cannot be accessed along known streets or paths within this distance. An example isochrone is shown in Fig. 3, where s is located on a curving coastline.

Having gained the isochrone I, the next step is to generate polygons whose vertices are within the area enclosed by I. These vertices are then used as waypoints that are linked by shortest paths to form a round trip. Our strategy here, which is carried out on the client, is to generate a series of ellipses and, from each one, form a polygon by placing n evenly-spaced vertices along the circumference. The overall process is described by Algorithm 1. As shown, this method requires an initial bearing β and desired perimeter k' for the generated polygons. These are determined by identifying the most distant point on the boundary of the isochrone (as illustrated in Fig. 3), which is intended to initially encourage the formation of round trips in the most accessible bearing from s. The algorithm then generates the list of polygons L by sweeping through the different bearings surrounding s, and adding polygons to L only when all of their n vertices are contained within I. After each sweep, the generated ellipses are also progressively flattened, stretched, and reduced in size. This is intended to help fit the polygons when the isochrone is irregularly shaped. It also aids the production

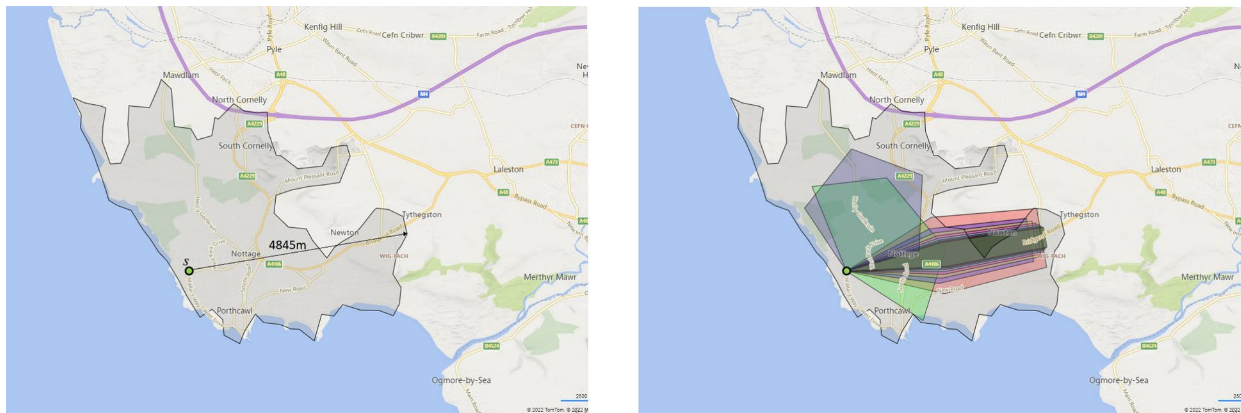


Fig. 3 Illustration of the isochrone-polygon method. Using a target of k = 10,000 m, the shaded area in the left image shows a 5000 m isochrone I generated from an address on the coast of southern Wales. The arrow indicates the initial bearing, giving $\beta = 80.3°$ and $k' = 2 \times 4845 = 9690$ m. The right image shows the first ten polygons generated by Algorithm 1, using $n = 5$

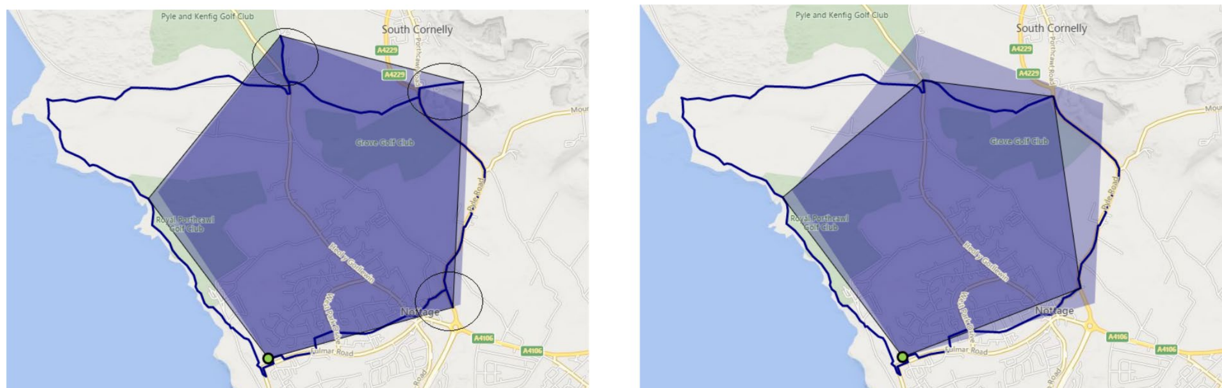


Fig. 4 The left image shows a round trip generated by one of the polygons from Fig. 3. The polygon with the black outline indicates the positions of the actual waypoints used, $P'_i = (p'_{i,1}, p'_{i,2}, \dots, p'_{i,n})$, which are slightly different to those of P_i. The three out-and-back sections that occur in this round trip are indicated by the circles. In the right image, these out-and-backs have been removed and, consequently, the waypoints of P'_i have been further adjusted

of a diverse set of polygons. An example of this process is shown in Fig. 3 where, due to the shape of the isochrone, some of the polygons are quite elongated.

Algorithm 1 Generate polygons

input : A starting vertex s; the starting perimeter k'; an initial bearing β; the number of vertices per polygon n; the isochrone I; and the desired number of polygons l.
output : A list L of n-vertex polygons.
1 **foreach** $r \in (1, \frac{9}{10}, \frac{8}{10}, \frac{7}{10}, \dots, \frac{1}{10})$ **do**
2 **foreach** $\alpha \in (1, \frac{1}{2}, 2, \frac{1}{3}, 3, \frac{1}{4}, 4, \dots, \frac{1}{10}, 10)$ **do**
3 **foreach** $b \in (0, 90, 180, 270, 30, 120, 210, 300, 60, 150, 240, 330)$ **do**
4 Set $x = (\beta + b) \pmod{360}$.
5 Produce a n-vertex polygon P at s with bearing x, perimeter rk', and aspect ratio α.
6 If all vertices of P are within the area of the isochrone I, add P to L.
7 If $|L| = l$ then end.

Once the list L has been constructed, further API calls are required to calculate the shortest paths between the successive vertices (waypoints) in each polygon. Let $L = (P_1, P_2, \dots, P_{|L|})$ be our list of polygons, and let $P_i = (p_{i,1}, p_{i,2}, \dots, p_{i,n})$ denote the vertices of each polygon (where $p_{i,1} = s$ for all $i \in \{1, \dots, |L|\}$). A round trip for polygon P_i is therefore constructed by linking each $(p_{i,j})$-$(p_{i,j+1})$-path (for all $j \in \{1, \dots, n-1\}$), followed by a final $(p_{i,n})$-$(p_{i,1})$-path. Each of these paths is expressed as a sequence of GPS coordinates that correspond to intermediate and intersection vertices in the underlying street network $G = (V, A, w)$, as described in "Problem Definition and Complexity".

It is common for mapping APIs to allow several waypoints to be specified in a single routing request. The free versions of Google Maps, Bing Maps, and MapBox, for example, allow up to $x = 25$ waypoints, while OpenRouteService allows up to $x = 50$. Here, this means that up to $\lfloor \frac{x-1}{n} \rfloor$ complete round trips can be constructed per request. In cases where waypoints do not correspond to locations on a street, these APIs also adjust the GPS coordinates to the nearest street-side location. Consequently, each polygon $P_i = (p_{i,1}, p_{i,2}, \dots, p_{i,n})$ maps to a second polygon $P'_i = (p'_{i,1}, p'_{i,2}, \dots, p'_{i,n})$, which specifies the actual waypoint coordinates used for the round trip. An example is shown in Fig. 4.

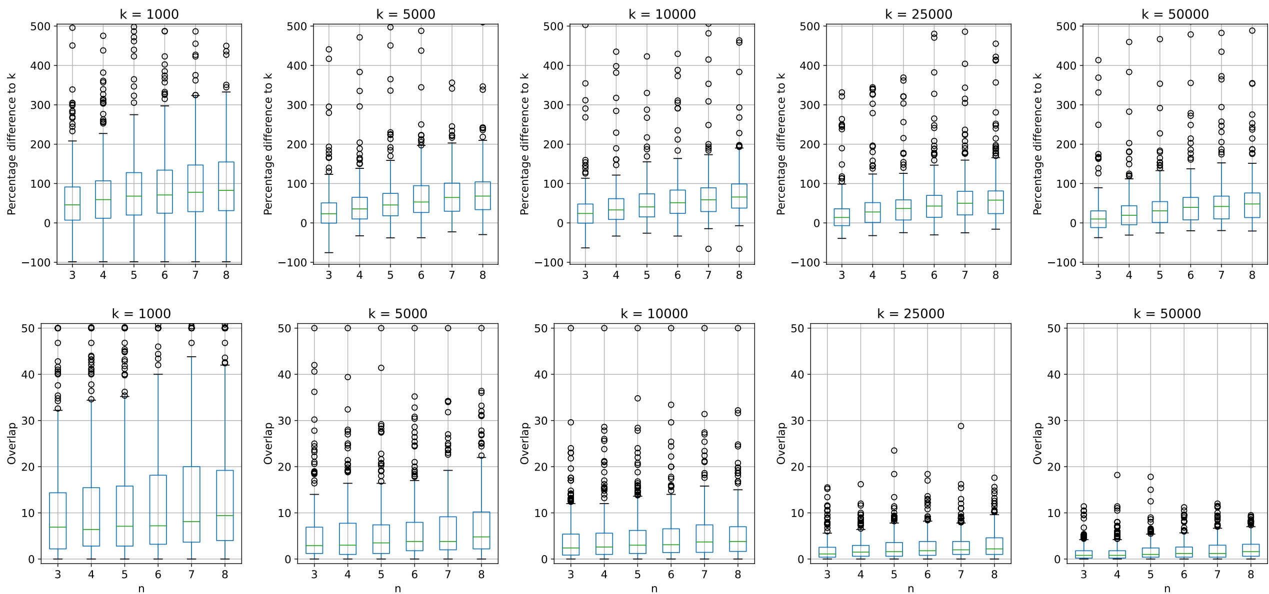


Fig. 5 Accuracy of Algorithm 2 for differing values of k and n. Each point is the mean across the solutions returned from five calls to OpenRouteService's round_trip feature

The first image of Fig. 4 also indicates a tendency for this scheme to produce "out-and-back" sections in a solution. These are often undesirable because they can involve the user having to perform a U-turn at an awkward location. They also involve the repetition of streets which, according to the KRT problem definition, we would like to avoid. To counter this, an optional step is to shorten each round trip by removing all instances of out-and-backs. This can be achieved by using the arcs of a round trip to define an undirected simple graph. Degree-one vertices in these graphs (not including s) should then be removed one by one until no such vertices remain. The second image of Fig. 4 illustrates the effects of this additional process. In this particular case, the resultant round trip has no repetition.

Initial Comparison and Setup

In this section, we examine the solutions returned by OpenRouteService's round_trip algorithm and compare their quality to those of our isochrone-polygon method. The overall behaviour of OpenRouteService's method is described in Algorithm 2.[2] As described in "Introduction", this is a server-side operation that starts by generating an n-sided polygon involving vertices $p_1, \ldots, p_n$ using a randomly selected initial bearing. A solution S is then formed by creating paths between each successive vertex in this polygon while also trying to avoid using the same street segment more than once.

[2] The open-source Java code for this can be found at [25].

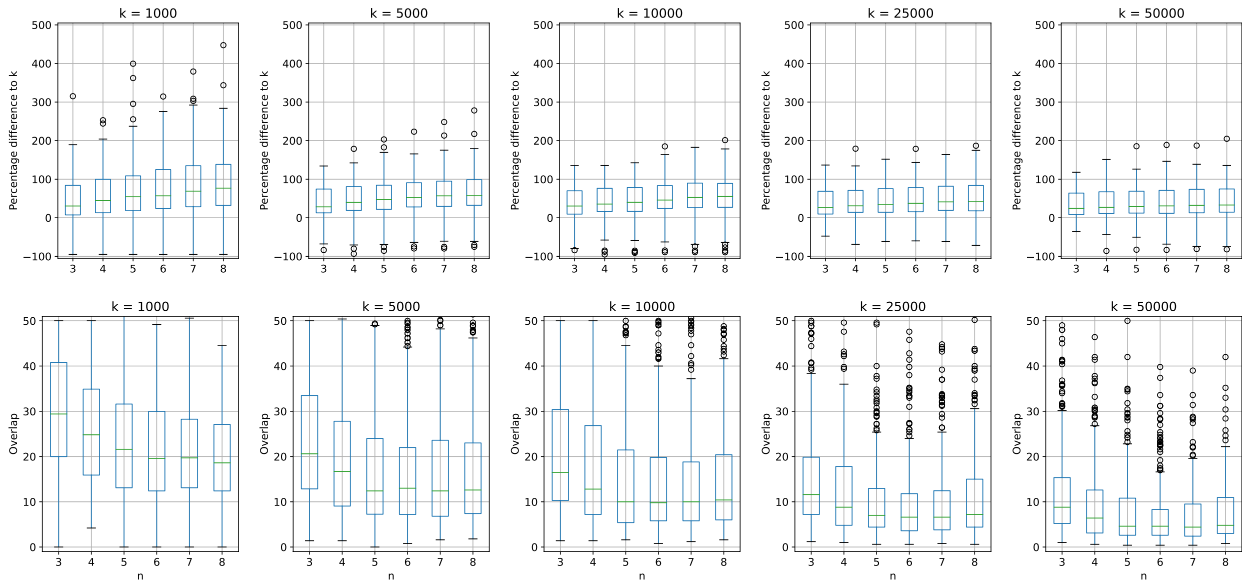


Fig. 6 Accuracy of the isochrone-polygon method (without out-and-back removal) for differing values of n and k. Each point is the mean across the first five solutions produced at each address

Algorithm 2 OpenRouteService round_trip method

input : A starting vertex s with defined GPS coordinates GPS(s); a desired distance k (in metres); the desired number of vertices on the generated polygon n; and a random seed.
output : A single solution (round trip) S.

1 Set GPS(p_1) = GPS(s), snapped to the nearest location on a street. If this snapping action fails, then halt (the algorithm has failed).
2 Set *InitialBearing* $\in \{0, 1, 2, \ldots, 359\}$ (randomly chosen), and let S be an empty round trip.
3 **foreach** $i \in \{1, 2, \ldots, n-1\}$ **do**
4   Set $b = (\mathit{InitialBearing} + \frac{i-1}{n}360) \pmod{360}$.
5   Set GPS(p_{i+1}) = GET-GPS(p_i, b, n).
6 **foreach** $i \in \{1, 2, \ldots, n-1\}$ **do**
7   Adjust GPS(p_{i+1}) by snapping to the nearest intersection along p_{i+1}'s current street.
8   Generate a path P from p_i to p_{i+1} (avoiding street segments already in S), and append P to S.
9 Generate a path P from p_n to p_1 (avoiding street segments already in S), append P to S, and then end.
10
11 **subroutine** GET-GPS(p, b, n)
12 Set $d = \frac{rk}{n}$, where r is chosen randomly from the real interval $[0.9, 1.1]$.
13 **foreach** $i \in \{1, 2, 3\}$ **do**
14   Let (x, y) be the GPS coordinates found by travelling d metres from GPS(p) on bearing b, snapped to the nearest location on a street.
15   If the previous snapping action failed, set $d = 0.95 \times d$. Otherwise, return (x, y).
16 If we reach here, then halt (the algorithm has failed).

Here, our client-side algorithms were programmed in Python 3.8 and were executed on 3.5 GHtz Windows 10 machines with 8 GB RAM [28]. In our implementation of the isochrone-polygon method, we made use of the free and open-source API OpenRouteService. Consequently, each algorithm execution involved making one request to form the isochrone, and one additional request for each set of $\lfloor \frac{50-1}{n} \rfloor$ round trips produced. For all experiments, we used a random sample of 250 real-world postal addresses from across North America and Western Europe. A single problem instance is therefore defined by an address and a target length k. The set of addresses was generated by selecting random dwellings

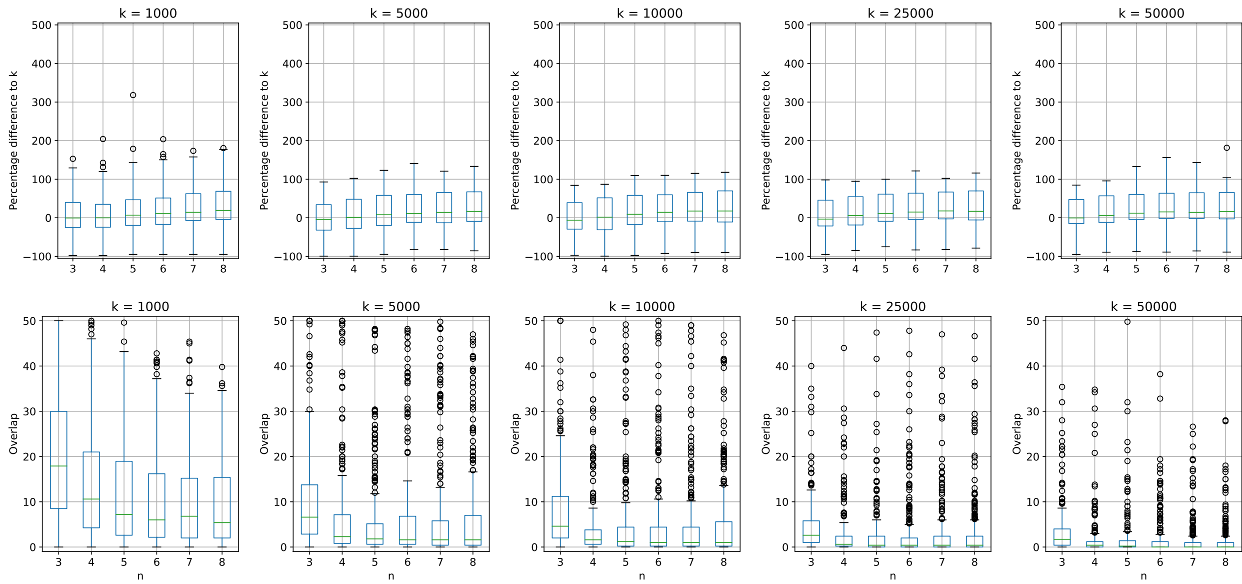


Fig. 7 Accuracy of the isochrone-polygon method (with out-and-back removal) for differing values of n and k. Each point is the mean across the first five solutions produced at each address

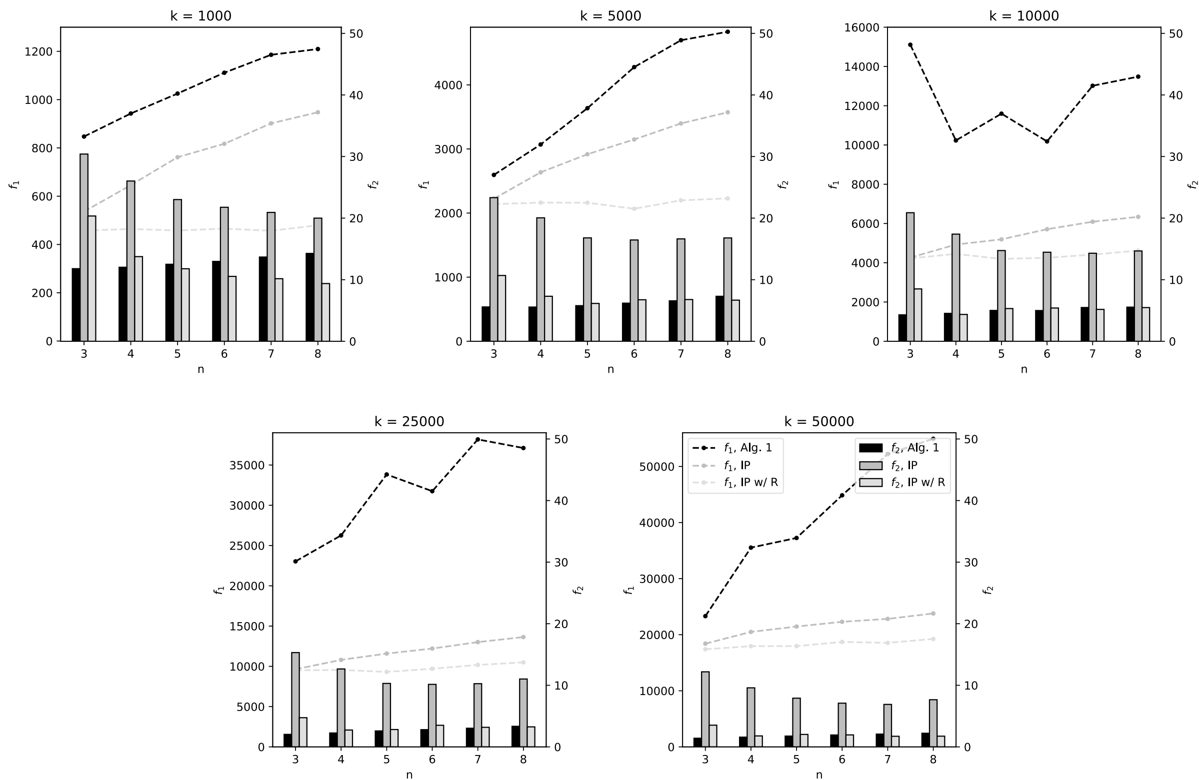
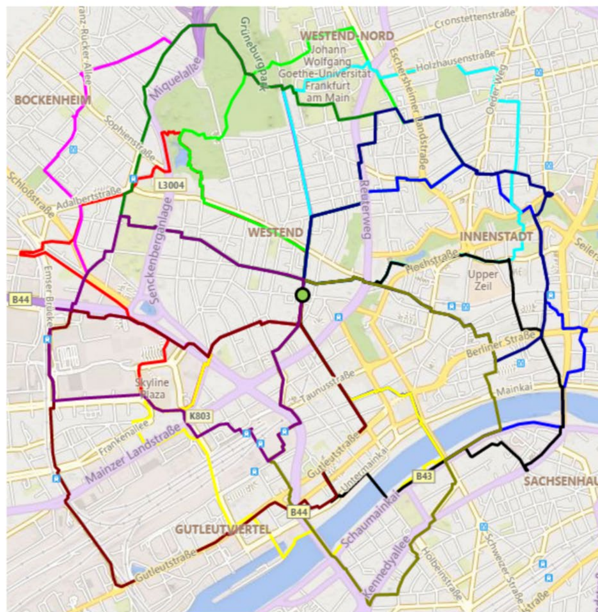


Fig. 8 Mean accuracy of Algorithm 2, the isochrone-polygon method (IP) and the isochrone-polygon method with out-and-back removal (IP w/ R). Note the differences in scales for cost function f_1

from the USA, France, UK, Netherlands and Canada, with the number of addresses per country weighted by their relative number of households. This led to 155, 37, 33, 14, and 11 addresses from each country respectively. Addresses for the UK were generated using the tool at [18], those for the USA and Canada came from [23], and the remainder were



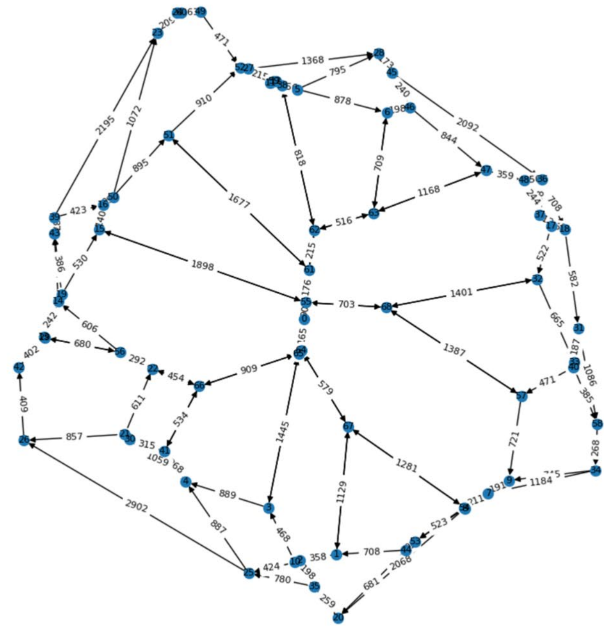


Fig. 9 (Left) Twelve round trips starting in central Frankfurt using $k = 5000$ m. (Right) The corresponding smoothed, arc-weighted directed graph $G = (V, A, w)$

taken from [17]. The selected travel mode in all experiments was "foot-walking", though solutions for hikers, drivers and cyclists can be achieved by simply changing the relevant input parameters.

Results

To assess the performance of Algorithm 2, trials on our random sample were performed using values of $k \in \{1000, 5000, 10{,}000, 25{,}000, 50{,}000\}$ (metres) and $n \in \{3, 4, \ldots, 8\}$. For each (k, n) pair, five calls per address were then made to OpenRouteService's round_trip feature to produce up to five different round trips. Figure 5 summarises the results of these trials. The top row shows the lengths of the returned solutions, expressed as the percentage difference to the target k; the bottom row shows the corresponding values for the percentage overlap (f_2, Eq. (2)). Each box is formed from 250 values, corresponding to the mean of the five values for each address.

Figure 5 indicates a tendency for Algorithm 2 to return solutions that exceed the target length k. This is quite natural since the distances between successive vertices in the polygon are calculated as straight lines whereas the physical *paths* between these are nearly always longer. In this sense, the perimeter of the polygon can be seen as a lower bound on the length of the resultant round trip. We also see that this effect worsens for higher values of n. In these cases, the algorithm must try and produce edge-distinct paths between a greater number of waypoints, which tends to result in lengthened round trips.

The bottom row of Fig. 5 shows how the amount of overlap also tends to increase for larger values of n. This is because, with a greater number of paths being generated, the potential for overlap increases. On the other hand, larger values of k result in less overlap, because the resultant round trips span a larger geographical area. Also observe that, on rare occasions, overlap costs of 50% are returned. In these cases, all streets are visited twice, indicating that the round trip is an out-and-back or (more generally) a tree.

Figure 6 shows the corresponding results for our isochrone-polygon method (without out-and-back removal). As with Algorithm 2, there is also a tendency for solutions to be too long here, with higher values of n worsening this effect. Larger amounts of overlap are also apparent compared to Fig. 5 because, unlike Algorithm 2, this method contains no mechanism for avoiding the repetition of street segments. Overlap is also seen to worsen for smaller values of n, because the acuter angles that occur in the generated polygons tend to encourage more out-and-backs in a solution.

Figure 7, however, demonstrates that the removal of out-and-backs from the isochrone-polygon method's solutions alleviates these issues. Specifically, their removal shortens round trips, helping to counter the overestimation seen in the previous results. Simultaneously, this removal also reduces the amount of overlap, helping to improve f_2. To show these differences more clearly, Fig. 8 shows a direct comparison of the mean values achieved for f_1 and f_2 by each method. For f_1, the isochrone-polygon method with out-and-back removals is clearly favourable, returning the

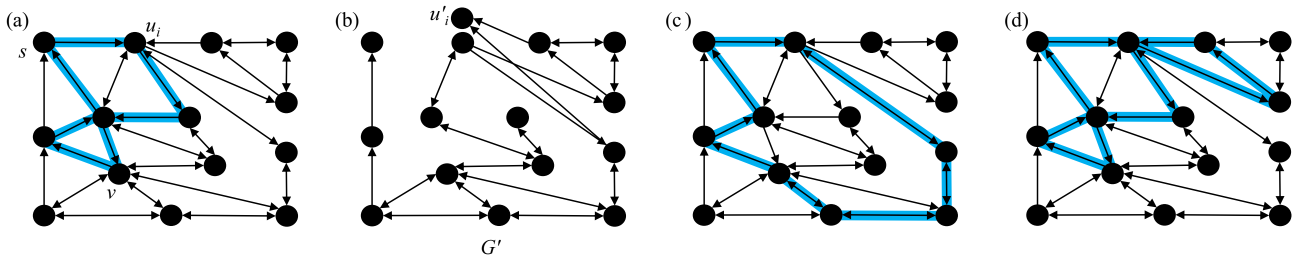


Fig. 10 Part **a** shows an example graph G with solution S (in blue) and a selected vertex $u_i \in S$. Part **b** shows the graph G' formed with respect to G, S, and u_i. Part **c** shows a solution S', formed by replacing the u_i-v path in S by a u_i-v in G'. Part **d** shows the solution formed by adding to S the cycle defined by a u_i-u_i' path in G'

most accurate solutions across all tested values of k and n. For f_2, the isochrone-polygon method (without out-and-back removal) is by far the worst, while no significant difference was observed between the remaining methods.[3]

In addition to the greater accuracy of the isochrone-polygon method (with out-and-back removal), also observe that it operates using fewer calls to the API. As described, Algorithm 2 requires one call per solution. On the other hand, following the isochrone request, our method can produce up to $\lfloor \frac{50-1}{n} \rfloor$ different solutions per call. Consequently, all results from the isochrone-polygon method presented here were found using just two calls per run.

Improving Solutions Using Multiobjective Local Search

In the previous section, we saw that the isochrone-polygon method (with out-and-back removal) can produce several high-quality candidate solutions to the KRT problem using only a small number of API calls. In this section we show how parts of these candidate solutions can be combined by the client to form further solutions, helping to increase algorithm accuracy.

To illustrate this, Fig. 9 shows twelve different round trips generated by the isochrone-polygon method. Each of these solutions is represented by a sequence of GPS coordinates; that is, intermediate and intersection vertices in the underlying street network. Consequently, we can combine these and use the smoothing actions from “Problem Definition and Complexity” to construct a corresponding arc-weighted, directed graph $G = (V, A, w)$, as shown in the figure. This means that attempts can now be made by the client to find new high-quality solutions for the KRT problem using G. Note that G does not contain *all* streets and intersections within $k/2$ metres of the starting vertex s; instead, it only includes the streets and intersections used in the supplied set of round trips. This leads to smaller graph sizes but, potentially, less accuracy. On the other hand, doing this eliminates the need for storing and/or downloading large street network databases. Here, we make use of both smoothing actions seen in “Problem Definition and Complexity”. In particular, the application of Case (ii) is desirable as it prevents the formation of solutions that will users to perform U-turns midway along a street. This helps to prevent the formation of out-and-backs in a solution.

Here, our approach is to generate new solutions for G using *local search*, a general-purpose methodology that seeks to identify high-quality solutions within a space of potential solutions. Local search operates by moving among solutions within this space using a *neighbourhood operator* that makes small alterations to the current solution. Because the KRT problem is multiobjective, here we choose to make use of the Pareto local search framework of Paquete et al. [31] given in Algorithm 3. As with most multiobjective optimisation algorithms, this uses the concept of *dominance* for distinguishing between solutions. For the KRT problem, which uses the two cost functions f_1 and f_2, this is defined as follows.

Definition 7 Let S and S' be two candidate solutions to the KRT problem. Also, assume (as is the case here) that we are seeking to minimise the cost functions f_1 and f_2. Then S is said to *dominate* S', written $S \preceq S'$, if and only if:

- $f_1(S) \le f_1(S')$ and $f_2(S) < f_2(S')$, or
- $f_1(S) < f_1(S')$ and $f_2(S) \le f_2(S')$.

If neither $S \preceq S'$ nor $S' \preceq S$, then solutions S and S' are said to be *mutually non-dominating*.

[3] According to a two-tailed Wilcoxon signed-rank test (at the 0.05 level) using sample sizes of 30, corresponding to the mean values for f_2 at each tested value of k and n.

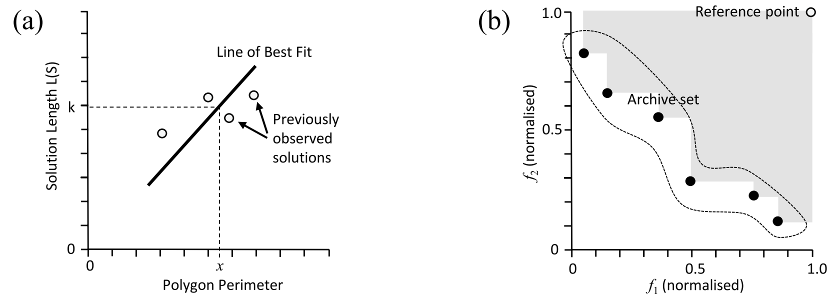


Fig. 11 Part **a** demonstrates how the perimeter x for a polygon P_i (where $i \geq 2$) is calculated in Algorithm 5. As shown, the previously-observed solutions (along with the origin (0, 0)) are used to generate a line of best fit. This is then used with k to determine the required value x. Part **b** demonstrates the calculation of the normalised S-metric using an archive set of mutually non-dominating solutions. The value of the S-metric corresponds to the area of the shaded part

Algorithm 3 Pareto local search

```
input  : An arc-weighted graph G = (V, A, w), and an archive set 𝒜 of mutually non-dominating solutions.
output : An archive set 𝒜 of mutually non-dominating solutions.
1  Mark each solution S ∈ 𝒜 as unvisited.
2  while there exists a solution S ∈ 𝒜 that is unvisited do
3      Mark S as visited.
4      foreach S′ ∈ N(S) do
5          Update-Archive(S′, 𝒜).
6
7  subroutine Update-Archive(S′, 𝒜)
8  foreach S ∈ 𝒜 do
9      if S′ ≤ S then remove S from 𝒜
10     else if S ≤ S′ then exit.
11 Mark S′ as unvisited and add it to 𝒜.
```

As shown, Algorithm 3, starts with an archive set $\mathcal{A}$ of mutually non-dominating solutions. Each of the solutions in this archive is then marked as unvisited. In each iteration, an unvisited solution S is selected from the archive set, and new solutions $S' \in N(S)$ are generated using a neighbourhood operator (see below). These new solutions are then added to $\mathcal{A}$ if they are not dominated by a solution already in the archive. In addition, if the new solution is seen to dominate any solutions currently in the archive, these are deleted from $\mathcal{A}$. This process continues until all solutions in $\mathcal{A}$ have been visited. The output $\mathcal{A}$ is a set of non-dominating solutions that are equal to, or an improvement on, the input set. Note that, under just one cost function, Algorithm 3 is equivalent to the steepest descent local search algorithm. Like steepest descent, the total number of iterations performed by the algorithm also depends on the number of improving moves that are carried out, which cannot be predicted.

Neighbourhood Operator

The solution space for the KRT is the set of all walks that start and end at vertex s. Neighbourhood operators for this problem should therefore take a member S of this space and make alterations that result in a new, different solution S'. In this sense, we say that S' is a member of the set of solutions that are *neighbours* of S, written $S' \in N(S)$.

To define the neighbourhood operator, let $S = (s = u_1, u_2, u_3, \dots, u_l = s)$ be a candidate solution, written as a sequence of vertices in G. The arcs of S are therefore defined by the multiset $\{(u_i, u_{i+1}) : i \in \{1, 2, \dots, l-1\}\}$. An intuitive neighbourhood operator is to now take two vertices $u_i, u_j \in S$ (where $i < j$), remove the corresponding u_i-u_j-walk in S, and then form a new (neighbouring) solution S' by adding a new, different u_i-u_j-walk. This new walk can be generated using path-finding algorithms such as breadth-first search, depth-first-search, or randomised variants of these.

Our proposed neighbourhood operator extends this idea by allowing several new solutions to be evaluated following a single application of a path-finding algorithm. The idea

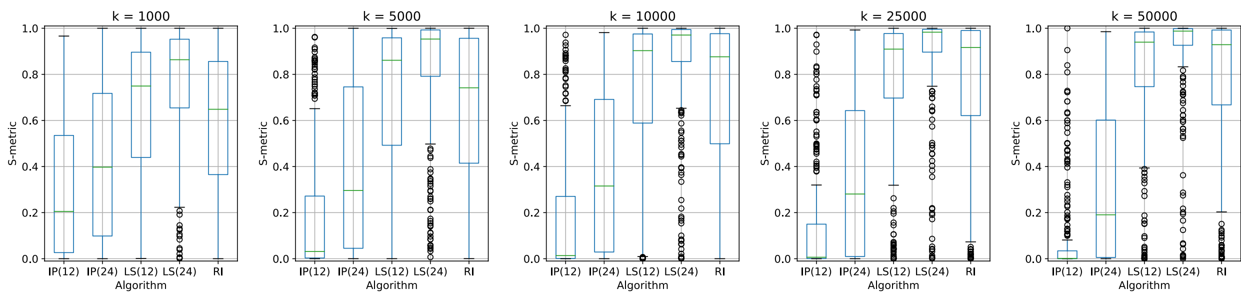


Fig. 12 S-metric values for the five algorithm variants for differing values of k. Each box plot is formed from 250 data points, one per address

is to select a vertex $u_i \in S$ and try to produce paths from this to all other vertices $u_{j>i} \in S$ without using the arcs currently in S. It proceeds as follows. Given G, S, and a vertex $u_i \in S$, a new graph G' is first constructed. This is simply a copy of G with all the arcs of S removed. Next, an additional dummy vertex u'_i is added to G', and all incoming arcs of u_i are redirected to u'_i (that is, all arcs of the form (v, u_i) are replaced by (v, u'_i)). Finally, a path-finding algorithm is used to form a directed tree T in G' that defines paths from u_i to all reachable vertices in the set $\{u_{i+1}, u_{i+2}, \ldots, u_l, u'_i\}$. Figure 10 shows example applications of this neighbourhood operator. In particular, Part (d) illustrates the effects of adding a new u_i-u'_i-path to the solution. This equates to adding a new subcycle at vertex u_i.

Note that Line 4 of Algorithm 3 involves evaluating all members of the set $N(S)$. This involves applying our neighbourhood operator for each vertex $u_i \in S$. A full description of how this is done is given by Algorithm 4.

Algorithm 4 Evaluate neighbourhood set

input : An arc weighted graph $G = (V, A, w)$, and the current solution S written as a series of vertices $(s = u_1, u_2, u_3, \ldots, u_l = s)$.
output : Cost values for all solutions $S' \in N(S)$.

1 **foreach** $u_i \in S$ (in a random order) **do**
2 Create the graph G' with respect to G, S, and u_i.
3 Form a directed tree T in G' that defines u_i-v-paths to all reachable vertices $v \in \{u_{i+1}, u_{i+2}, \ldots, u_l, u'_i\}$.
4 **foreach** $v \in (u_{i+1}, \ldots, u_l)$ **do**
5 Let S' be the solution formed by replacing the u_i-v-walk in S by the u_i-v-path in T.
6 Evaluate S' w.r.t f_1 and f_2.
7 Let S' be the solution formed by splicing the cycle defined by the u_i-u'_i-path in T at position i in S.
8 Evaluate S' w.r.t f_1 and f_2.

Learning Good Routes Using Regression

An alternative to using local search with the KRT problem is to employ a method that makes a series of calls to the mapping API. At each iteration, this method should then use information gained in previous calls to try to identify better waypoints for the construction of solutions. Recall from "The Isochrone-Polygon Method" that the isochrone-polygon method operates by first creating a polygon $P = (p_1, \ldots, p_n)$, where each vertex p_i is at a particular GPS coordinate. P is then used with the API to construct a round trip solution S. In turn, this process gives a second polygon $P' = (p'_1, \ldots, p'_n)$ where each p'_i corresponds to the *actual* waypoints used for S. At this point, we might now choose to use S and P' to inform the production of a new polygon for use with the API. Specifically, if the current solution S is seen to be too short (that is, $L(S) < k$), then this new polygon

Table 1 S-metric values, percentage deviations between k and f_1, and final archive set sizes from the five algorithm variants for differing values of k

	IP(12)	IP(24)	LS(12)	LS(24)	RI
$S-metric(mean \pm s.d.)$					
$k = 1000$	0.295 ± 0.283	0.428 ± 0.327	0.654 ± 0.297	***0.763** $\pm$ 0.257	0.597 ± 0.304
$k = 5000$	0.186 ± 0.271	0.388 ± 0.346	0.704 ± 0.323	***0.818** $\pm$ 0.272	0.649 ± 0.332
$k = 10{,}000$	0.172 ± 0.264	0.369 ± 0.328	0.741 ± 0.320	***0.850** $\pm$ 0.256	0.710 ± 0.326
$k = 25{,}000$	0.142 ± 0.241	0.350 ± 0.343	0.767 ± 0.306	***0.880** $\pm$ 0.232	0.746 ± 0.329
$k = 50{,}000$	0.089 ± 0.198	0.326 ± 0.339	0.800 ± 0.287	***0.892** $\pm$ 0.226	0.756 ± 0.330
Percentage deviation between k and the minimum value for f_1 in the archive (mean $\pm$ s.d.)					
$k = 1000$	12.77 ± 16.85	9.502 ± 15.45	3.346 ± 11.73	**2.429** $\pm$ 10.35	2.736 ± 9.736
$k = 5000$	17.52 ± 20.75	9.821 ± 15.52	4.193 ± 14.34	**1.779** $\pm$ 10.14	2.693 ± 10.71
$k = 10{,}000$	17.96 ± 20.31	11.87 ± 16.98	3.666 ± 12.88	1.824 ± 9.711	**1.523** $\pm$ 6.092
$k = 25{,}000$	14.51 ± 17.90	9.542 ± 13.14	1.645 ± 7.210	**0.233** $\pm$ 0.890	1.644 ± 7.364
$k = 50{,}000$	15.40 ± 20.34	10.56 ± 15.50	1.293 ± 7.978	**0.518** $\pm$ 4.641	1.519 ± 7.654
Final archive size $\|\mathcal{A}\|$ (mean $\pm$ s.d.)					
$k = 1000$	2.057 ± 1.025	2.195 ± 1.307	4.159 ± 3.555	***4.195** $\pm$ 3.638	3.268 ± 1.921
$k = 5000$	1.616 ± 0.774	1.728 ± 0.930	3.320 ± 3.494	***3.416** $\pm$ 3.783	2.284 ± 1.514
$k = 10{,}000$	1.540 ± 0.755	1.684 ± 0.965	2.848 ± 2.595	***2.904** $\pm$ 2.945	2.092 ± 1.438
$k = 25{,}000$	1.448 ± 0.658	1.456 ± 0.739	2.368 ± 1.869	***2.568** $\pm$ 2.956	1.676 ± 1.170
$k = 50{,}000$	1.238 ± 0.480	1.262 ± 0.532	***2.044** $\pm$ 1.796	2.020 ± 2.494	1.419 ± 0.675

All figures are taken from 250 data points, one per address. The bold typeface indicates the best-observed values; asterisks indicate where these values were seen to be significantly different to the second-best values at the 0.001 level

Table 2 Run times (in seconds) of the three optimisation algorithms for differing values of k

Method	$k = 1000$	$k = 5000$	$k = 10{,}000$	$k = 25{,}000$	$k = 50{,}000$
Run times, including latency (mean $\pm$ s.d.)					
LS(12)	***0.492** $\pm$ 0.118	***0.700** $\pm$ 0.396	***1.226** $\pm$ 1.446	***2.526** $\pm$ 5.516	***3.261** $\pm$ 7.368
LS(24)	0.798 ± 0.217	1.126 ± 0.431	1.856 ± 1.594	3.757 ± 6.776	4.792 ± 7.405
RI	2.932 ± 0.959	3.669 ± 1.044	4.646 ± 1.560	7.320 ± 5.271	10.279 ± 7.557
Run times without latency (mean $\pm$ s.d.)					
LS(12)	***0.038** $\pm$ 0.038	***0.177** $\pm$ 0.411	***0.620** $\pm$ 1.474	***1.655** $\pm$ 5.431	***1.751** $\pm$ 7.330
LS(24)	0.067 ± 0.065	0.279 ± 0.432	0.887 ± 1.667	2.414 ± 6.737	2.686 ± 7.399
RI	0.143 ± 0.050	0.509 ± 0.243	1.246 ± 1.336	3.180 ± 5.191	4.793 ± 7.158

All figures are the mean plus/minus the standard deviation taken from 250 data points, one per address. The bold typeface indicates the best-observed values; asterisks indicate where these values were seen to be significantly different to the second-best values at the 0.001 level

should be an enlarged version of P'. Similarly, if $L(S) > k$, the new polygon should be a shrunken version of P'.

A description of our proposed method is given in Algorithm 5. As shown, the process starts by using an initial polygon P_1 to produce a solution S_1 and its actual waypoints P'_1. At each iteration $i \geq 2$, the information gained in iterations $1, \ldots, i-1$ is then used to build a regression model that is used to calculate the perimeter x for the new polygon P_i. This polygon is then used to produce S_i and P'_i. Here, our proposed method uses a simple linear (least-squares) regression model for determining the perimeter of new polygons. A demonstration of how x is calculated with this linear model is shown in Fig. 11a. Other regression models may also be used in practice, though we must bear in mind that the number of data points in these models will be rather small due to the limited runtimes being imposed.

Algorithm 5 Regression-based improvement

1 Use a polygon P_1 to determine a solution S_1 and its actual waypoints P'_1.
2 Let $R = \{(0,0), (\text{perimeter}(P'_1), L(S_1))\}$.
3 **foreach** $i \in \{2, \dots, l\}$ **do**
4 Determine the least-squares line of best fit for R and calculate the desired perimeter x (see Figure 11(a)).
5 Select a previously-observed polygon P'_j at random, where $j \in \{1, \dots, i-1\}$.
6 Let P_i be a copy of P'_j. Now rescale P_i so that its perimeter is of length x.
7 Use P_i to determine the solution S_i and its actual waypoints P'_i.
8 Add the element $(\text{perimeter}(P'_i), L(S_i))$ to R.

As noted previously, our chosen API allows the production of several solutions per call. Here, we therefore use Algorithm 1 to produce up to $\lfloor \frac{50-1}{n} \rfloor$ initial polygons. A separate regression model is then maintained for each of these initial polygons, with $\lfloor \frac{50-1}{n} \rfloor$ new solutions being produced at each iteration. As with our local search method, each solution produced is then considered for inclusion in the final archive set $\mathcal{A}$ using the dominance criteria given in Definition 7.

Method Comparison

In this section, we compare the performance of the isochrone-polygon method, multiobjective local search, and the regression-based improvement algorithm. In all trials reported here, out-and-back removal was employed, and $n = 4$ vertices were used in the generated polygons, allowing up to twelve separate round trips to be generated in each API call. The remaining experimental details are the same as "Initial Comparison and Setup".

Two variants of the isochrone-polygon method are considered here. After determining the isochrone, the first of these, IP(12), uses just one further call to produce twelve solutions. These are then used to form the archive set $\mathcal{A}$ of mutually non-nominating solutions. Similarly, the second variant, IP(24), uses two calls to produce 24 solutions for the construction of $\mathcal{A}$. Our local search algorithm extends this approach by using these solutions to form the arc-weighted graph $G = (V, A, w)$ as described in "Improving Solutions using Multiobjective Local Search". This graph, together with $\mathcal{A}$, is then used as input for Algorithm 3. These algorithms are labelled LS(12) and LS(24) in the following tables and figures. Note that the local search method makes no further calls to the API during execution.

In our trials, all applications of local search used breadth-first search (BFS) for producing the tree T (as seen in Line 3 of Algorithm 4). This path-finding algorithm identifies paths between vertices that contain the fewest possible number of arcs. In our trials, we also experimented with a randomised version of BFS in which vertices were removed from random positions in the BFS queue (as opposed to just the head of the queue); however, no significant differences in performance were observed between these variants.[4] Hence, only results from the BFS variant are reported here. Using our arc-weighted graph $G = (V, A, w)$, the complexity of BFS is $\mathcal{O}(|V| + |A|)$; consequently, each application of Algorithm 4 (and iteration of Algorithm 3) has complexity $\mathcal{O}(|S|(|V| + |A|))$.

Finally, for the regression-based improvement algorithm (RI), an iteration limit of $l = 10$ was used in all trials. This means that, including the production of the isochrone, eleven calls to the API are required per run. (Additional iterations of this algorithm were not seen to make further improvements to solutions in general.)

Algorithm Accuracy

To compare the quality of the solutions produced by our five algorithm variants, we use the well-known S-metric [2]. This measures the quality of an archive set $\mathcal{A}$ by calculating the volume of the space existing between $\mathcal{A}$ and a single reference point. A demonstration is provided in Fig. 11b. To be valid, the reference point must be dominated by all solutions in the archive. Here, reference points for each problem instance were calculated by examining the returned archive sets of each algorithm and then taking the maximum observed values for both cost functions. As shown in Fig. 11b, volumes were then normalised to assume a value between zero and one, with larger values reflecting a higher-quality archive set.

Figure 12 summarises the S-metric values achieved by the five algorithms for various values of k. As might be expected, for all values of k the isochrone-polygon method

[4] In this section, all claims of statistical significance are made according to two-tailed paired t-tests ($p \leq 0.001$) using results collected across all 250 problem instances.

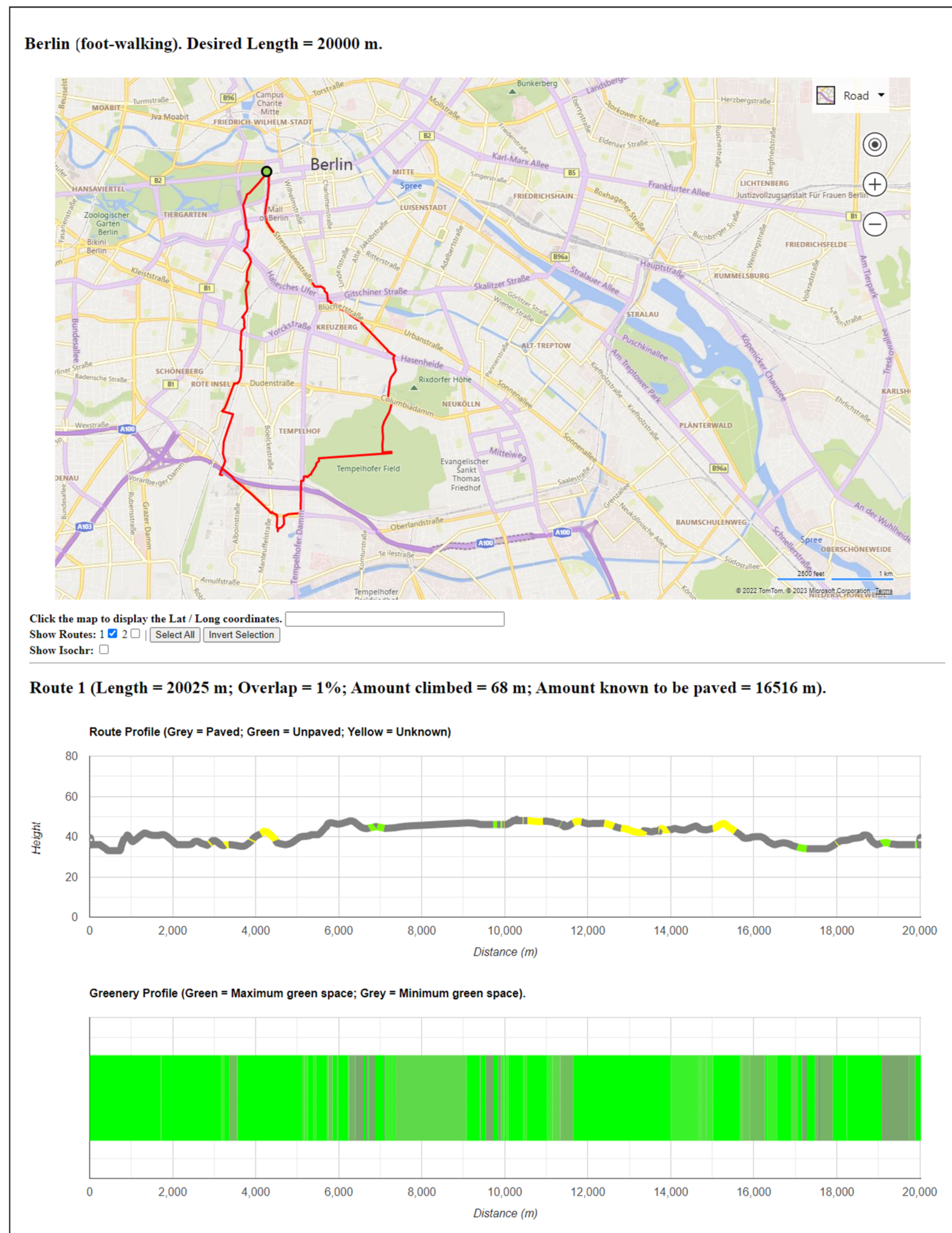


Fig. 13 Screen shot of the output produced by our implementation [28]

gives significantly better archive sets when the number of solutions it initially produces is doubled from 12 to 24. The addition of local search also makes large improvements to these results, with the 24-solution variant producing significantly better results than the twelve-solution variant for all values of k. In our trials, the 24-solution variant of local search was also seen to produce significantly better results than the regression-based improvement method for all values of k, although the latter was also seen to be a significant improvement on both isochrone-polygon variants. A summary of these S-metrics is also shown in Table 1, indicating the superiority of the archive sets returned by the 24-solution variant of the local search algorithm.[5]

Note that, unlike the local search algorithm, the regression-based improvement method does not optimise f_2 directly during execution. It is therefore relevant to also compare algorithm accuracy according to cost function f_1 only. To do this, Table 1 also gives information on the minimum observed value for f_1 in the archive sets. (Specifically, for each algorithm this is expressed as a percentage deviation between the f_1 value and k, averaged across the problem

[5] A full listing of the results achieved by each algorithm on each instance can be found in [28].

instances.). This reveals no significant differences between LS(24) and RI for any of the tested values of k, suggesting that, while these algorithms have similar capabilities for reducing f_1, the local search method is more effective at also reducing f_2.

Archive Sizes and Execution Times

In comparing these algorithms, it is also useful to look at the sizes of the archive sets produced, since larger values for $|\mathcal{A}|$ will give users a wider choice of solutions. Table 1 summarises this information. As shown, the best results are again returned by the local search methods—indeed, across all values of k, both variants of local search were seen to produce significantly larger archive sets than the regression-based improvement method.

Table 2 summarises the run times of the three optimisation algorithms with and without latency (here latency refers to the time taken between sending the API request and receiving the response). As shown, in both cases the regression-based improvement method takes the longest to execute due to the larger number of requests made and its associated bookkeeping. The 24-solution variant of local search also takes slightly longer than the twelve-solution variant in both cases because (a) it makes an additional call to the API, and (b) the larger number of solutions leads to graphs $G = (V, A, w)$ that have more vertices and arcs. That said, across our experiments with local search, 96% of runs were seen to terminate in less than six seconds, including latency.

Table 2 also demonstrates that run times lengthen when longer round trips are required. Some of these increases are due to the API, which requires more server-side computation time for calculating paths between distant waypoints. An extreme result was also witnessed for one address in our sample, located in rural Alberta, Canada. This particular case featured a very irregularly shaped isochrone whose outline tended to closely follow the small number of roads in the locality. As a result, it took Algorithm 1 over 100 s to identify suitable polygons for this case. One way of rectifying this problem would be to simply reduce the number of increments in each for-loop of the procedure.

Conclusions and Further Work

This paper has proposed several fast-acting heuristics for the $\mathcal{NP}$-hard k-length round trip (KRT) problem. Unlike existing approaches, these algorithms involve only small amounts of data transfer and do not need to store large map databases; however, they are still able to produce accurate solutions in short amounts of time (typically a few seconds), even over large geographical areas. "Initial Comparison and Setup" has demonstrated that our isochrone-polygon method produces solutions of greater accuracy than OpenRouteService's round_trip algorithm. "Method Comparison" has also shown that solution quality can be further improved using a fast, bespoke local search algorithm (and to a lesser extent, a regression-based improvement method).

During our research, we explored the idea of replacing the local search procedure with a variant of Johnson's algorithm that enumerates all cycles in the directed graph G that contain s. Given excess time, this approach is guaranteed to find the optimal (Pareto) archive set of cyclic solutions. However, it suffers from two drawbacks. First, the method only considers cycles which, as noted in "Problem Definition and Complexity", is usually too restrictive for practical applications. Second, the run times of this approach are often excessive because the number of cycles usually rises exponentially with respect to graph size.

The screenshot in Fig. 13 shows an example output from our downloadable implementation [28]. This output is written in html and viewed in a web browser. In addition to the interactive visualisations of round trips, other information is displayed in this output including elevation profiles, surface types, and greenery ratings. In future work, our multiobjective algorithm could be modified so that these features are also optimised. For example, additional cost functions might be used to help identify round trips that minimise the amount climbed or maximise the amount of paved surfaces. Other factors, such as the amount of lighting [30], the required number of street crossings [11], or the predicted traffic density, might also be considered, providing that this real-world information can be reliably determined.

Related to this, it would also be useful to further consider the aesthetics of a solution. Rossit et al. [34], for example, have noted that visual attractiveness is an important factor in vehicle routing problems, with users often preferring solutions that are judged to be *compact* and *logical*. Again, such factors could be incorporated into our optimisation methods using additional cost functions. These might involve comparing a solution's similarity to a standard geometric shape, or measuring how easy it is for a user to memorise its directions. It would also be useful to constrain round trips so that they do not stray too far from their designated starting point. This could be used during pandemics, for example, when governments urge the public to stay within limited distances of their homes.

As noted in "Initial Comparison and Setup", the algorithms presented here have been implemented in Python. We imagine that it be straightforward to reimplement these methods in JavaScript so that clients could download and execute these algorithms directly in a web browser. Shorter run times are also to be expected if compiled programming languages such as Java, Go, or C++ were to be used [32].

Author Contributions All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by R. Lewis. The first draft of the manuscript was written by R. Lewis and both authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding Not applicable.

Data Availibility Statement All source code and datafiles are available from https://zenodo.org/doi/10.5281/zenodo.8154412.

Declarations

Conflict of Interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

Research Involving Human Not applicable.

Informed Consent Not applicable.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Abraham I, Delling D, Goldberg A, and Werneck R. Alternative routes in road networks. ACM J Exp Algorithmics. 2013; 18.
2. Audet C, Bigeon J, Cartier D, Le Digabel S, Salomon L. Performance indicators in multiobjective optimization. Eur J Oper Res. 2020;292(2):397–422.
3. Basagni S, Bruschi D, Ravasio S. On the difficulty of finding walks of length k. Theor Inform Appl. 1997;31(5):429–35.
4. Bast H, Delling D, Goldberg A, Müller-Hannemann M, Pajor T, Sanders P, Wagner D, Werneck R. Route planning in transportation networks. In: Kliemann L, Sanders P, editors. Algorithm engineering: selected results and surveys. Cham: Springer International Publishing; 2016. p. 19–80.
5. Bhandari R. Survivable networks. Kluwer Academic Publishers; 1999.
6. Boeing G. OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. Comput Environ Urban Syst. 2017;65:126–39.
7. Chalupa D, Balagan P, Hawick K, Gordon N. Computational methods for finding long simple cycles in complex networks. Knowl-Based Syst. 2017;125:96–107.
8. Fomin F, Golovach P. Long circuits and large Euler subgraphs. SIAM J Discret Math. 2014;28(2):878–92.
9. Gemsa A, Pajor T, Wagner D, Zündorf T. Efficient computation of jogging routes. In: Bonifaci V, Demetrescu C, Marchetti-Spaccamela A, editors. Experimental algorithms. Berlin: Springer; 2013. p. 272–83.
10. Giscard P, Kriege N, Wilson R. A general purpose algorithm for counting simple cycles and simple paths of any length. Algorithmica. 2019;81:2716–37.
11. Hannah C, Spasić I, Corcoran P. A computational model of pedestrian road safety: the long way round is the safe way home. Accident Anal Prevent. 2018;121:347–57.
12. https://brouter.de/brouter-web/. Accessed 2024-2-1.
13. https://connect.garmin.com/. Accessed 2024-7-2.
14. https://openrouteservice.org/. Accessed 2024-2-1.
15. https://trailrouter.com/. Accessed 2024-2-1.
16. https://wiki.openstreetmap.org/wiki/Way. Accessed: 2024-02-01.
17. https://www.bestrandoms.com/. Accessed 2024-2-1.
18. https://www.doogal.co.uk. Accessed 2024-2-1.
19. https://www.komoot.com/plan/. Accessed 2024-2-1.
20. https://www.plotaroute.com/routeplanner. Accessed 2024-2-1.
21. https://www.routeloops.com/. Accessed 2024-2-1.
22. https://www.runnersworld.com/uk/news/a31913463/stravas-routes-tool/. Accessed 2024-7-2.
23. https://www.worldnamegenerator.com. Accessed 2024-2-1.
24. Johnson D. Finding all the elementary circuits of a directed graph. SIAM J Comput. 1975;4(1):77–84.
25. Karich P. https://github.com/GIScience/graphhopper/blob/4bb48323f2dfa4583292667558332269d2cc83ac/core/src/main/java/com/graphhopper/routing/RoundTripRouting.java. Accessed 2022-10-01.
26. Lewis R, Corcoran P. Finding fixed-length circuits and cycles in undirected edge-weighted graphs: an application with street networks. J Heuristics. 2022;28:259–85.
27. Lewis R, Corcoran P and Gagarin A. Methods for determining cycles of a specific length in undirected graphs with edge weights. J Combin Optimiz. 2023;46(29).
28. Lewis R. Source code, addresses, and data sets, https://zenodo.org/doi/10.5281/zenodo.8154412, 2023. Accessed 1 Feb 2024.
29. Li L, Cheema M, Lu H, Ali M, Toosi A. Comparing alternative route planning techniques: a comparative user study on Melbourne, Dhaka and Copenhagen road networks. IEEE Trans Knowl Data Eng. 2022;34(11):5552–7.
30. Nunes P, Moura A, and Santos J. Evolutionary approach for the multi-objective bike routing problem. In Computational Logisitics, volume 12433 of Lecture Notes in Computer Science. Springer, 2020; pp. 311–25.
31. Paquete I, Chiarandini M, Stutzle T. Pareto local optimum sets in the biobjective traveling salesman problem: an experimental study. In: Gandibleux X, Sevaux M, Sorensen K, T'kind V, editors. Metaheuristics for Multiobjective Optimisation, volume 353 of Lecture Notes in Economics and Mathematical Systems. Berlin: Springer; 2004. p. 177–200.
32. Pereira R, Couto M, Ribeiro F, Rua R, Cunha J, Fernandes J, and Saraiva J. Energy efficiency across programming languages: How do energy, time, and memory relate? In Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, New York, NY, USA, 2017. Association for Computing Machinery, pp. 256–67.
33. Richey M, Parker R. A cubic algorithm for the directed Eulerian subgraph problem. Eur J Oper Res. 1991;50(3):345–52.
34. Rossit D, Vigo D, Tohme F, Frutos M. Visual attractiveness in routing problems: a review. Comput Oper Res. 2019;103:13–34.
35. Yen J. Finding the K shortest loopless paths in a network. Manage Sci. 1971;17(11):661–786.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.