6-2018

Iterated Local Search Algorithms for Bike Route Generation

Aidan Pieper

Iterated Local Search Algorithms for Bike Route Generation

By

Aidan R. Pieper

* * * * * * * * *

Submitted in partial fulfillment

of the requirements for

Honors in the Department of Computer Science

UNION COLLEGE

March, 2018

Abstract

PIEPER, AIDAN R. Iterated Local Search Algorithms for Bike Route Generation. Department of Computer Science, March, 2018.

ADVISOR: Matthew Anderson

Planning routes for recreational cyclists is challenging because they prefer longer more scenic routes, not the shortest one. This problem can be modeled as an instance of the Arc Orienteering Problem (AOP), a known NP-Hard optimization problem. Because no known algorithms exist to solve this optimization problem efficiently, we solve the AOP using heuristic algorithms which trade accuracy for speed. We implement and evaluate two different Iterated Local Search (ILS) heuristic algorithms using an open source routing engine called GraphHopper and the OpenStreetMap data set. We propose ILS variants which our experimental results show can produce better routes at the cost of time.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Cycling is a popular and diverse activity enjoyed by millions of people all over the world. To some, cycling is a means of commuting to work while to others it is a recreational sport. The quality of cycling infrastructure varies across the globe. In countries like Belgium and the Netherlands where cycling is a popular recreational sport, there are vast networks of bicycle-friendly secondary roads [16]. However, many places do not have this same level of cycling infrastructure so bike riders must share highways with other road vehicles.

Route planning for recreational cyclists poses a fundamentally different problem than traditional route planning problems because the shortest route is not necessarily the preferable cycling route. Recreational cyclists generally prefer longer, more scenic, and less trafficked routes as the goal of the activity is recreation not transportation. When planning routes, recreational cyclists consider different factors such as route distance, elevation gain, maximum percent gradient, and how pleasant a road is to travel by bike. Designing a route that fits all user-specified criteria is a difficult task. Moreover, there are no set criteria which determine a "preferable" cycling route. The desirability of a given route is based on the rider's personal preferences, goals, and fitness. This research explores different algorithms for generating cycling routes for recreational road cyclists.

Most bike rides begin and end in the same location. Using this assumption, this research focuses specifically on generating preferable *circular* cycling routes. For example, a cyclist may want a 15-mile route which starts and ends at their home.

1.1 Motivations

Traditional route planning problems focus mainly on finding a path in a graph optimizing for either shortest distance or time. There exists many route planning tools such as `strava.com`, `mapmyride.com`, and `ridewithgps.com` which allow users to add points on a map and generate a route between such destinations. However, none of these tools can fully generate a route without additional user input.

1.2 Related work

In the literature, planning preferable cycling routes is modeled as an instance of the Arc Orienteering Problem (AOP), a variant of the Orienteering Problem (OP) [16]. First introduced in 1987 by Golden et al., the classical OP is a combination of node selection and determining shortest paths between nodes in a graph

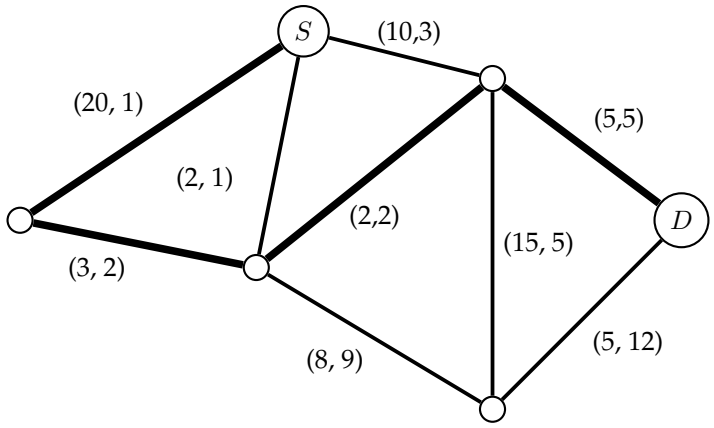


Figure 1: Undirected AOP instance with start node S and destination D. Arc label is (score, cost). Bold path is optimal for a budget of 10 (score = 30, cost = 10).

[11]. The OP is a hybrid between two classical combinatorial problems, the Knapsack Problem and the Traveling Salesman Problem[1]. In the classical OP, each node in the graph is assigned a non-negative score and a non-negative cost. Given a starting node, a destination node, and some maximum cost budget, the objective is to determine a non-repeating path which starts at the starting node, visits some subset of the graph nodes, and ends at the destination node [12]. In addition, the solution path must both maximize the total collected score, accrued from visiting a node, and keep the total collected cost under the specified budget.

The AOP is the arc variant of the OP where each arc, i.e., graph edge, is given a score and a cost. In the AOP, scores and costs are accrued from visiting an arc instead of a node. For example, Figure 1 shows an undirected AOP instance where S is the start node, D is the destination node, the budget is 10, and every edge is labeled (score, cost). The shortest path is $S \rightarrow (10, 3) \rightarrow (5, 5) \rightarrow D$ which has a cost of 8 and a score of 15. However, for the specified budget, $S \rightarrow (20, 1) \rightarrow (3, 2) \rightarrow (2, 2) \rightarrow (5, 5) \rightarrow D$ is the optimal solution with a score of 30 and a cost of 10. The optimal solution is clearly not the shortest path but rather the path with the maximal score constrained by the cost budget.

Previous research shows that both the OP and the AOP are NP-Hard problems for directed and undirected graphs [12]. No algorithms are known to *optimally* solve the AOP or OP in polynomial time. While there is considerable research into the OP and its variants, there is less research into the AOP. Gunawan et al. provide an exhaustive survey of the OP and its variants, but the AOP is clearly over shadowed by other OP variants in the literature [12].

[1]The OP may sometimes be referred to as the *Selective* Traveling Salesman Problem [13].

Gavalas et al. [8] show approximation algorithms for the AOP in both directed and undirected graphs. Approximation algorithms are algorithms to NP-Hard optimization problems that do not produce optimal answers yet have provable accuracy bounds. A polylogarithmic approximation algorithm is shown for directed graphs while a $(6 + \epsilon + o(1))$-approximation algorithm is shown for undirected graphs [8]. Moreover, they show a reduction from the AOP to the OP. Using an existing OP approximation algorithm by Nagarajan and Ravi [15], this reduction yields a $O(\frac{(\log m)^2}{\log \log m})$-approximation algorithm for solving the AOP in directed graphs where m is the number of edges.

Souffriau et al. [16] study the AOP in the context of cycle trip planning. Souffriau et al. provide an integer programming mathematical model for the AOP and a greedy randomized adaptive search heuristic algorithm for solving AOP instances to near optimality in a few seconds. To evaluate performance of their algorithm, the authors test their algorithm against a road network of bike-friendly roads in East Flanders. The East Flanders' bicycle road network covers 5 regions and is comprised of 989 nodes with 2963 arcs for a total of 3585 km of road. This model for the AOP requires that each node and edge is visited at most once by the solution path.

Verbeeck et al. [17] consider the cycle trip planning problem in a directed graph. The authors propose two heuristic algorithms for solving the AOP: A branch-and-cut algorithm and an iterated local search algorithm. See Section 2.2 for a detailed explanation of iterated local search. Unlike, Souffriau et al., this model allows the route to visit the same vertex multiple times but visiting the same arc twice is not allowed. A two-way road can be travelled exactly once in each direction since it is modeled by two separate edges in the directed graph. Both algorithms were evaluated by running them on the East Flanders road network dataset provided by Souffriau et al.

Similarly, research by Bergman and Oksanen [6] defines the "Circular Cycle Tour Problem" as a cycle trip planning problem where the start and end location are the same. Like other cycle routing problems, they model it as an instance of the AOP. Bergman and Oksanen use a popularity weighted road network graph using road popularity data from smartphone fitness tracking application `sports-tracker.com`. The authors use a tabu search heuristic algorithm for solving their AOP instance.

1.3 Research Question

As mentioned in Section 1.2, existing literature models cycle trip planning as an instance of the AOP. This research follows the existing literature and focuses on implementing and improving existing AOP algorithms for cycle route planning.

Since road networks can be quite large and because the AOP is NP-Hard, searching for the optimal route may take an infeasible amount of time. Since we care more about finding a good route than finding the best possible route, we trade off optimality for speed. However, even AOP approximation algorithms are too slow for applications where a response time on the order of milliseconds is required, e.g., 300 ms [14]. Therefore, we focus on heuristic algorithms for route generation. Both Verbeeck et al. and Lu and Shahabi propose heuristic algorithms which follow the Iterated Local Search (ILS) framework. ILS is a heuristic method for solving many optimization problems. ILS builds a sequence of locally optimal solutions through repeated applications of a heuristic search algorithm. Our research question is as follows:

To what extent can ILS algorithms be improved to generate better bike routes?

1.4 Our Contribution

In the following sections we refer to the the algorithm proposed by Verbeeck et al. as the *VVA Algorithm* and the algorithm proposed by Lu and Shahabi as the *LS Algorithm*. We implement the VVA and LS algorithms using an open source routing engine named GraphHopper. We make initial observations by running these algorithms on a subset of the New York State road network using public mapping data from the OpenStreetMap foundation. These observations lead us to create four variants of the LS algorithm. We then ran experiments on a small road network to compare the relative performance of VVA, LS, and our variants. We also pursue an absolute algorithm evaluation by attempting solve an Integer Program model of the AOP.

Our experimental results show that the LS algorithm is faster than the VVA algorithm but not substantially. However, the LS algorithm does produce substantially higher scoring routes than VVA. Our LS variants can produce even higher scoring routes than the baseline LS algorithm but at a cost of time. We are unable to validate the claimed 300 millisecond response time of the LS algorithm. VVA and LS are solving slightly different problems since VVA restricts taking a road more than once while LS does not. This may account for the scoring differences.

The remainder of this thesis explains this research in detail. Section 2 contains background information on ILS, VVA, and LS. Next, Section 3 discusses our data source, the open source software we used, and our implementation of VVA and LS including our LS variants. In Section 4 we present our experimental data and performance analysis using two different ILS stopping criteria. We also discuss an Integer Programming model for the AOP. Finally, we conclude with a summary of our results and future work.

2 Preliminaries

First, we discuss how to model preferable cycling routes and then heuristics for solving the AOP. Recall that Iterated Local Search (ILS) is a heuristic method for solving many optimization problems. We explain the abstract idea of ILS and then we explain how ILS is used in both the VVA and LS algorithms. The VVA algorithm uses a simple depth-first-search with additional checking. The LS algorithms uses a greedy search with spatial techniques to reduce the search space.

2.1 Modeling Preferability of Cycling Routes

Recreational cyclists consider many factors when designing a preferable cycling route. The following is a non-exhaustive list of such factors:

- Route distance
- Route elevation gain
- Time
- Maximum percent gradient
- Amount of traffic
- Number of intersections
- Good cellular service
- Easy parking at start
- Availability of restrooms
- Availability of rest stops
- Scenery
- Proximity to bike shops
- Proximity to mass transit
- Limited uphill at end of ride

Many of these factors can be modeled nearly identically. For example, distance, elevation gain, and time are all calculated by the sum of those weights over all roads in the route. On the other hand, maximum percent gradient can be seen as a "Boolean criterion." That is, a road's steepness is either under the maximum percent gradient or over, in which case the road does not satisfy this criterion. If these Boolean criteria must be avoided, a simple option is to initially remove all roads from the graph which do not meet these criteria.

In previous research, the cost of a particular arc is usually the length of the road and the score is some measure of the preferability of the road. Since the AOP requires a single value for the cost and score of each arc, computing costs and scores as linear combinations of different features is a one way to model multiple factors. For example, a particular road's cost might be a combination of its length, its elevation gain, and the level of traffic on the road. This allows one to give certain factors more importance by weighting them more heavily in the linear combination. Furthermore, one might want to avoid certain Boolean criteria instead of outlawing them entirely. For instance, one could avoid dirt roads by giving them higher costs.

While multiple route factors are important to recreational cyclists, the focus of this research is not to

model the preferability of roads. Hence, we assume that we have the necessary data to appropriately score roads for recreational cyclists. The focus of this research is on the route planning algorithm, not creating the graph and its associated weights which represents the AOP instance.

2.2 Iterated Local Search

Iterated Local Search (ILS) is a framework for solving optimization problems using heuristic search algorithms. A heuristic is a technique used to solve a problem quickly when exact or approximation methods are too slow. Heuristic algorithms can be thought of as "shortcuts" in that they trade optimality and completeness for speed. Heuristics are often used in search algorithms to determine which branch of the search to take but are not guaranteed to produce the best solution. A heuristic algorithm is commonly referred to as a heuristic.

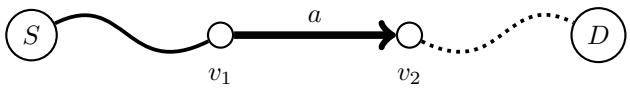
Local Search is a heuristic method for solving optimization problems. Local Search starts with a candidate solution and moves to a higher scoring solution as defined by an objective function which scores solutions in the search space [10]. Local Search can get stuck in local optima which are points in the search space that are better than all similar solutions but are not the best possible solution.

ILS is a variant of Local Search that attempts to stop it from getting trapped in local optima. Instead of repeating random trials of the heuristic algorithm, ILS builds a sequence of locally optimal solutions generated by the heuristic which is more likely to lead to a better overall solution [10]. This is done by first generating an initial solution using the search heuristic, perturbing the current solution, and applying the search heuristic again on the modified solution. The perturbation and local search steps are then repeated until some condition, usually time, is met. Algorithm 1 outlines the ILS framework.

Algorithm 1: ILS(t, $localsearch$, $score$)

Data: t: a time,
$localsearch$: a heuristic search function,
$score$: an objective function,
$perturb$: a function which modifies the solution.
Result: A solution of the $localsearch$ function.

```
S ← localsearch(empty solution)
while t seconds have not elapsed do
    S* ← perturb(S)
    S' ← localsearch(S*)
    if score(S') > score(S) then
        S ← S'
return S
```



$$(S \rightarrow v_1).cost + a.cost + ShortestPath(v_2, D) \leq Budget$$

Figure 2: Arc feasibility checking

Despite its simplicity, ILS can be challenging to implement effectively because many implementation choices are left to the developer. For example, an effective ILS implementation requires a certain level of domain specific knowledge. The main issue with ILS is that the algorithm may still get "trapped" in a local maximum over many iterations. Therefore, the modification step must modify the solution enough to make progress but not too much that the search is effectively starting with a different "random" solution upon each iteration.

2.3 VVA Algorithm

Verbeeck et al. propose an ILS algorithm which uses a modified version of depth-first search (Algorithm 2) as its local search heuristic. It is implemented as a recursive function that finds a path between two disconnected nodes in the bike route. The algorithm is allowed to "take" a road and add it to the current route as long as it has not been traversed before and the shortest path from the end of the traversed arc to the destination is less than the remaining distance budget after taking the arc (Line 4). In other words, it must be *feasible* to get from the end of the chosen arc to the desired destination after traversing the arc (Figure 2).

Since this requires many shortest path computations, the VVA algorithm assumes that all-pairs shortest path have been pre-computed before the ILS runs. In Algorithm Algorithm 2, the function $shortestPath(v_1, v_2)$ would return the pre-computed shortest path. In addition, the $maxDepth$ parameter is used to restrict the depth of the search and reduce the search space (Line 1).

Using this DFS algorithm as the local search heuristic Verbeeck et al., apply the ILS framework to create a bike route planning algorithm. Algorithm 3 first generates an initial route using the DFS heuristic and stores the path in the variable $route$ (Line 2). The ILS perturbs the solution by removing a road from the solution and invoking the DFS procedure to find a new local solution (Lines 11 to 16). In the perturbation phase, the algorithm removes R consecutive arcs starting at the arc at position A in the running solution $route$. If a new path is found after removing a path segment from the solution, then the new path is merged into the current solution (Line 17). If no new path with score improvement can be found A and R are both

Algorithm 2: DFS($route$, s, d, $dist$, $minProfit$, $maxDepth$)

Data: $route$: a temporary solution,
s: the start node of the path,
d: the end node of the path,
$dist$: the maximum cost of the route,
$minProfit$: the minimum score of the route,
$maxDepth$: the maximum number of edges allowed in the solution,
$shortestPath(v_1, v_2)$: a function which returns the shortest distance between two nodes of the graph,
$edges(v_1)$: a function which returns all edges of a node.
Result: A boolean which denotes whether a path was found. If true, the solution is contained inside of $route$.

```
1  if maxDepth < 0 then
2      return false
3  for arc ∈ edges(s) do
4      if arc ∉ route and arc.cost + shortestPath(arc.end, d) < dist then
5          Add arc to route
6          if arc.end = d and route.score > minProfit then
7              return true
8          else if DFS(route, arc.end, d, dist - arc.cost, minProfit, maxDepth - 1) then
9              return true
10         Remove arc from route
11 return false
```

incremented by 1 (Line 21). This perturbs the solution more and more in an attempt to move the search out of a local optima.

The main drawback of the VVA algorithm is that the ILS has slow iterations because it is performing DFS on every iteration. Moreover, it requires many shortest paths to be precomputed before the algorithm can run. This can be infeasible on large real-world mapping datasets. Since the algorithm assumes all pairs shortest-path is pre-computed, the feasibility checking used by the search is $O(degree^{maxDepth})$ where $degree$ is the max degree of nodes in the road network and $maxDepth$ is the maximum depth allowed in the DFS. However, since the DFS returns when it finds any better path not just the best one, this worst case performance is not typically expected.

2.4 LS Algorithm

The ILS algorithm proposed by Lu and Shahabi aims to solve many of the problems of VVA including slow iteration and large pre-computation. Instead of relying on pre-computed shortest paths, the LS algorithm uses online shortest path computations and does less feasibility checking by reducing the search space with spatial pruning techniques (See Section 2.4.4). In addition, LS uses a greedy path generation algorithm

Algorithm 3: ILS-VVA(s, d, $dist$, $maxDepth$, t)

Data: s: the start node of the path,
d: the end node of the path,
$dist$: the maximum distance of the path,
$maxDepth$: the maximum depth allowed in the DFS,
t: a time.
Result: a path.

```
route ← empty route
if not DFS(route, s, d, dist, 0, maxDepth) then
    route ← empty route
A ← 1, R ← 1
while t seconds have not elapsed do
    temp ← copy of route
    if R > temp.length then
        R ← 1
    if A + R > temp.length − 1 then
        R ← temp.length − 1 − A
    Remove R arcs from temp starting at arc at index A
    minScore ← sum of scores of removed arcs from temp
    s* ← starting node of first arc removed
    d* ← ending node of last arc removed
    new ← empty route
    if DFS(new, s*, d*, dist - temp.dist, minScore, maxDepth) then
        Merge new into temp at index A
        route ← temp
        A ← 1, R ← 1
    else
        A ← A + 1, R ← R + 1
return route
```

instead of DFS.

2.4.1 Attractive Arcs

LS models a solution route in terms of "attractive arcs" which are arcs with a positive score. A path from node v_1 to v_2 is a series of attractive arcs which starts with a_1, ends with a_n and is denoted by, $(v_1 \rightsquigarrow a_1 \rightsquigarrow a_2 \rightsquigarrow \ldots \rightsquigarrow a_n \rightsquigarrow v_2)$. The symbol $\rightsquigarrow$ denotes the shortest path in the graph between two nodes or arcs. The path between two adjacent attractive arcs ($a_i \rightsquigarrow a_{i+1}$) is known as a "blank path segment" and is the shortest path from the end vertex of a_i to the start vertex of a_{i+1}. These vertices are respectively denoted $l_i.start$ and $l_i.end$ where l_i is the shortest path. Given an attractive arc a_i from a solution path, $a_i.pre$ refers to the previous attractive arc (a_{i-1}) and $a_i.post$ refers to the next attractive arc in the path (a_{i+1}).

To build a solution, the LS algorithm connects many attractive arcs together using shortest path blank path segments. The total cost of a path is the sum of all costs of all the arcs in the path, including the arcs in the blank path segments. The score of a path is the sum of all attractive arcs *excluding* any attractive arcs which may be in blank path segments.

2.4.2 Candidate Arc Set

Every arc a in the solution route S is associated with a set of candidate attractive arcs that it could be replaced with. Arcs are taken out of these sets in order to generate a path between two vertices.

Definition 1 ([14]). *Let $a \in S$ be an arc in the solution route S. Let B be the distance budget. Then the Candidate Arc Set (CAS) of a, denoted by $a.CAS$ is the set of arcs who have a positive score and can feasibly replace a in S, i.e. $\forall a_c \in a.CAS, a_c.score > 0$ and $(a.pre \rightsquigarrow a_c \rightsquigarrow a.post).cost < B - S.cost + (a.pre \rightsquigarrow a \rightsquigarrow a.post)$.*

Lu and Shahabi show that candidate arc sets have the following inherited closure property. This allows the search space to be reduced when computing some CASs since the parent CAS can be restricted.

Lemma 1 ([14]). *Let a be an arc. $\forall a_c \in a.CAS, a_c.CAS \subseteq a.CAS$.*

To choose which candidate arcs to add to the solution, Lu and Shahabi propose a criterion called "Quality Ratio" which is defined for an arc from a candidate arc set. The intuition is that arcs with higher value and lower cost will be more likely to improve the solution. In order to determine which arcs to remove from the solution in the ILS perturbation, they propose a criteria called "Improve Potential". The intuition is that solution arcs with lower scores and more valuable nearby arcs are more likely to improve the solution. See Section A.2 for more information.

Algorithm 4 performs the feasibility checking to generate a set of candidate arcs which can be used to connect a start node v_1 to a destination node v_2. The algorithm takes in a set of possible arcs, iterates over each one, and adds the arc to the current CAS only if its score is positive and the distance of the path from v_1 to a to v_2 is within the specified budget (Lines 4 to 7). In addition, the Quality Ratio is calculated for the specified arc in the CAS (Line 6). If the CAS, A, passed into the algorithm is non-empty, then it can use the "CAS inherit" property and filter out arcs whose paths are within the new specified budget. If the CAS passed in is non-empty, then the algorithm will iterate over all arcs in the graph to find the ones which can be feasibly inserted (Lines 2 to 3).

Algorithm 4: computeCAS(G, A, v_1, v_2, $dist$)

Data: G: the road network graph,
A: a candidate arc set,
v_1: start node,
v_2: destination node,
$dist$: allowable budget.
Result: A set of candidate arcs.

```
1 CAS ← empty set
2 if A is empty then
3     A ← all arcs from G
4 for a ∈ A do
5     if a.score > 0 and (v1 ⇝ a ⇝ v2).cost ≤ dist) then // Feasibility checking
6         a.qr = QualityRatio(v1, v2, a)
7         add a to CAS
8 return CAS
```

If arcs are added to the current solution, then the route's distance changes as well as the remaining budget. For the new arcs added, computing the respective CASs using Algorithm 4 suffices. However, the previous arcs in the solution need to have their CASs changed since the remaining distance budget is now different. Algorithm 5 takes in two budget values, $newDist$ and $oldDist$. If the new budget is smaller than the old budget, there may be some arcs in our CAS whose paths are too long for the new budget. Therefore, the algorithm employs CAS inheritance and restricts the current CAS by removing the arcs which can no longer be feasibly inserted with the new budget (Lines 2 to 5). If the new budget is larger than our old budget, then the algorithm expands the CAS by checking the feasibility of all arcs in the graph (Lines 6 to 9).

Algorithm 5: updateCAS(A, a, v_1, v_2, $newDist$, $oldDist$)

Data: A: set of all arcs in the graph,
a: arc whose CAS needs to be updated,
v_1: node in current path before a,
v_2 : node in current path after a
Result: An updated set of candidate arcs

$CAS \leftarrow a.CAS$
if $newDist < oldDist$ **then** `// Restrict CAS using inherit property`
    for $e \in a.CAS$ **do**
        if $(v_1 \rightsquigarrow e \rightsquigarrow v_2).cost > newDist$ **then**
            remove e from CAS
else if $newDist > oldDist$ **then** `// Expand CAS by checking all edges from graph`
    for $e \in A$ **do**
        if $e \notin CAS$ *and* $e.score > 0$ *and* $(v_1 \rightsquigarrow e \rightsquigarrow v_2).cost \leq newDist$ **then**
            add e to CAS
return *CAS*

2.4.3 ILS Formulation

The local search method used by LS is a greedy algorithm. It continuously inserts feasible arcs from a CAS at the closest blank path segment until the budget is exhausted or there are no more CAS arcs. The LS ILS algorithm removes a random arc from the solution using the heuristic scoring metric "Improve Potential" and uses the greedy local search to fill the gap in the path. If a new path is found, it is inserted into the route and the CAS of each arc is computed or updated accordingly. See Section A.1 for details on this path generation algorithm and how it is used in the ILS.

2.4.4 Spatial Pruning Techniques

While LS relies on the "CAS inherit" property to restrict the search space, it still has to do a lot of processing to generate the initial CAS or update CASs when the budget expands. To address this issue, Lu and Shahabi propose an "ellipse pruning" technique to reduce the number of arcs which need to be checked.

An ellipse is a curve such that for every point on the curve, the sum of the distances to the two focal points is constant. Consider the scenario where there are two graph nodes v_1 and v_2 in which the desired path between the two has a budget of b. Furthermore, consider the ellipse whose focal points are the two nodes and whose sum of the distances to the two focal points is b (Figure 3). For all points p on the ellipse $(v_1 \rightsquigarrow p \rightsquigarrow v_2).cost = b$ where the shortest path is the straight line Euclidean distance. Therefore, if there is an arc a which connects v_1 to v_2 and contains a point p_o outside of the ellipse, we know that $a.cost > b$

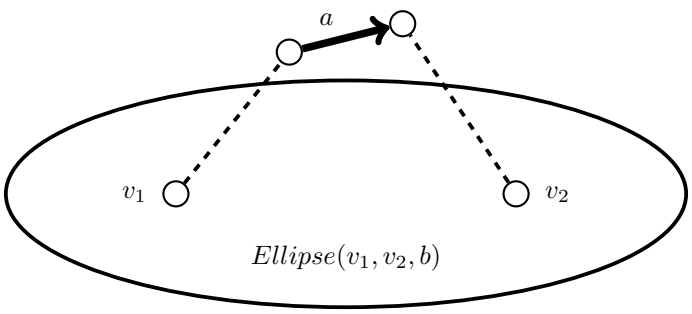


Figure 3: Illustration of Lu and Shahabi's ellipse pruning technique. The goal is to connect v_1 to v_2 with a path of budget b. The arc a is excluded from the search since it contains a point outside of the ellipse and is therefore infeasible. [14].

since $(v_1 \rightsquigarrow p_o \rightsquigarrow v_2).cost > b$. This criteria is used to prune arcs from the search space when calculating or updating CASs.

3 ILS Implementation

We implement the VVA and LS algorithms and evaluate them on real world road networks. This section discusses the technical details of our implementation. We first discuss our data source, the open source software used, and road scoring metrics. Then we explain the choices made in both our VVA and LS implementations. Finally this section concludes with observations of our implementations and we propose four new LS variants based on these observations.

3.1 OpenStreetMap

We use the crowd-sourced open mapping dataset provided by the *OpenStreetMap* (OSM) foundation[2][5]. However, the OSM map format is an XML-based schema which is not trivially translatable into a road network graph. Luckily, in addition to open data, OSM includes a collection of open source software which interface with the data. Because the goal of this research is not to translate raw OSM data into a usable graph representation, we used software that already has this parsing capability in order to implement both ILS algorithms. Because OSM is a crowdsourced dataset, its level of accuracy varies across the world which is the main drawback to using this data.

[2]OSM provides a free mapping dataset for the entire planet. A full world map is around 56GB.

Metadata Value	Meaning
-3	"Avoid at all cost"
-2	"Only use to reach your destination, not well suited"
-1	"Better take another way"
0	"As well as other ways around."
1	"Prefer"
2	"Very nice way to cycle"
3	"This way is so nice, it pays out to make a detour also if this means taking many unsuitable ways to get here."

Table 1: OSM bicycle routing hints. Taken directly from the OSM wiki [5].

3.1.1 Map Metadata

Some OSM roads (known as "ways" in OSM parlance) contain metadata used to help bicycle routing but it is not guaranteed to be available. This bicycle routing hint is a value which is used to express the desirability of a road (Table 1). However, the OSM wiki notes that these values "should not be used where other attributes[3] are considered adequate description" [5].

3.2 GraphHopper

We use GraphHopper as the starting point for our research. GraphHopper is an open source routing engine written in Java which can download and parse raw OSM data into a usable graph representation [2]. On top of data parsing, GraphHopper provides a web server and webpage front-end which are useful for visualizing and running routing algorithms (Figure 4). Internally, GraphHopper has a number of built-in pathfinding algorithms including A* and Dijkstra which can be used for routing. These algorithm implementations provide a good template for implementing other routing algorithms with GraphHopper.

Additionally, GraphHopper supports multiple "routing profiles" which modify the weights of roads based on a particular vehicle. This is used to give preference to certain roads that are more suited for a particular vehicle. GraphHopper's default bike routing profile contains code for giving the normalized "priority" value of a road. A normalized priority value is one of the 7 values contained in Table 1 normalized to a 0 to 1 scale with 1 being more preferable. When determining the priority of a road, this routing profile also considers other road metadata such as road speed and road surface. GraphHopper will use the other metadata if bicycle specific routing hints are not available. We use this normalized priority value calculated by GraphHopper as our road scoring mechanism. A road's cost is simply its distance in meters.

[3]Example attributes include number of lanes, maximum speed, and incline.

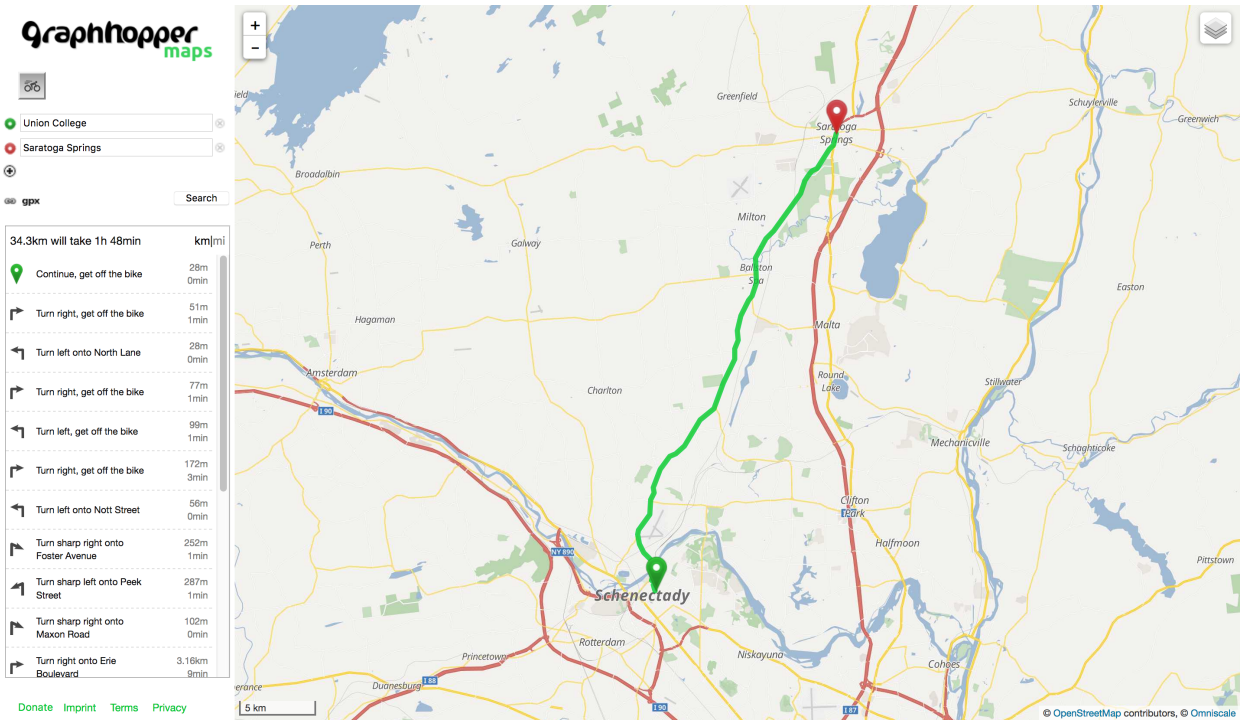


Figure 4: GraphHopper web frontend. This is OpenStreetMap data overlaid with the shortest path from Union College to Saratoga Springs.

3.2.1 Contraction Hierarchies

GraphHopper supports a special type of graph preprocessing called contraction hierarchies [2]. The goal of contraction hierarchies is to preprocess the graph such that subsequent shortest path queries can be computed more quickly but still provably correct. This is done by ordering nodes by some importance value and then iteratively "contracting" the least important node. Contracting a node v means replacing shortest paths through v with new shortcut edges [9]. A faster shortest path search can be obtained by running a bidirectional shortest-path search making sure that the forward direction only traverses edges going to more important nodes and the backward direction only traverses edges coming from more important nodes.

When running the GraphHopper server for the first time, the engine processes the raw OSM data into a graph and builds contraction hierarchies for each of the enabled routing profiles. This contraction step may take many minutes depending on the size of the graph. We use GraphHopper's built-in contraction hierarchy based shortest path algorithm[4] for calculating shortest paths in both our VVA implementation and LS implementation.

3.3 VVA Implementation

Our Java implementation of the VVA algorithm differs very little from the pseudocode provided by Verbeeck et al. Our implementation does not have a set of starting locations nor does it retain the four best initial solutions. Rather, the starting location is fixed and only the single highest scoring solution is retained between iterations. These choices both simplify our implementation and make the ILS closer to that of the LS algorithm. Our local search heuristic is still a recursive DFS with a maximum depth parameter that performs arc feasibility checking.

Another difference in our implementation is how we check arc feasibility. Instead of pre-computing all-pairs shortest path, we use GraphHopper's built in contraction hierarchies and do an online shortest-path computation. This is slower than assuming all shortest paths have been pre-computed. However, this requires far less computation before our algorithm starts. In addition, we can leverage GraphHopper's fast and correctly implemented shortest-path algorithms without writing our own pre-processing code. Since we are routing on a contraction hierarchy graph, we need to make sure to ignore the special "shortcut" edges added in the contraction phase. Our road scoring mechanism is GraphHopper's normalized priority value and road costs are distances in meters.

[4] GraphHopper's default algorithm is bidirectional Dijkstra's algorithm.

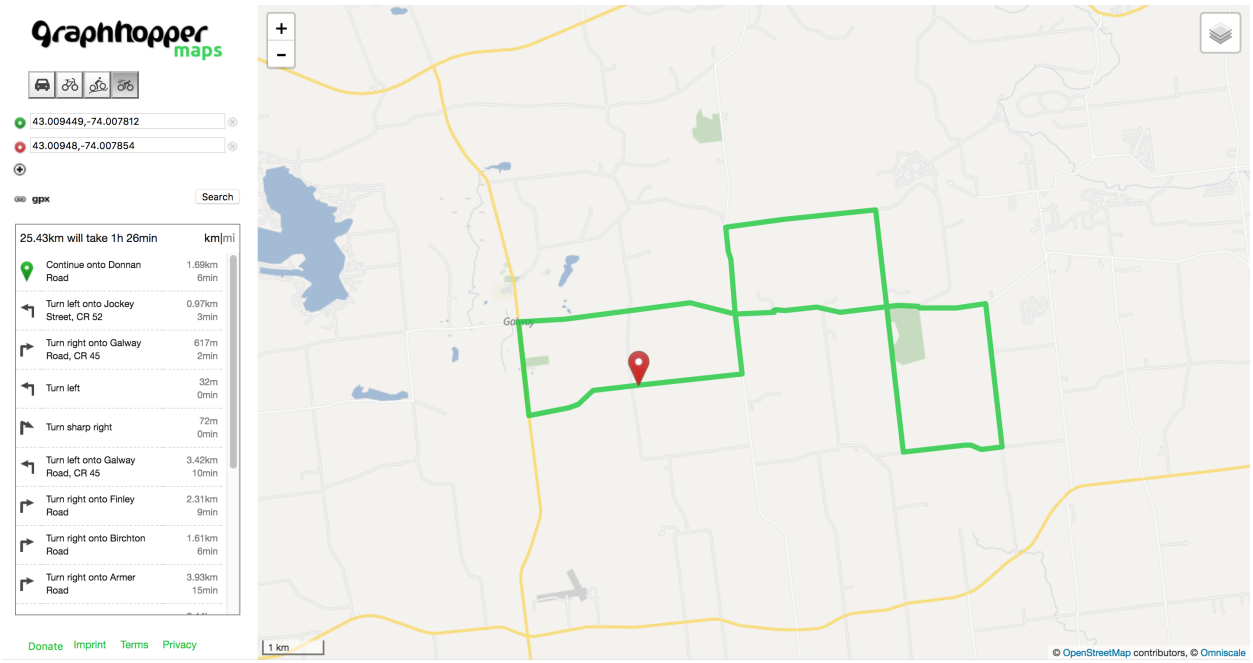


Figure 5: Example route produced by our implementation of the VVA algorithm.

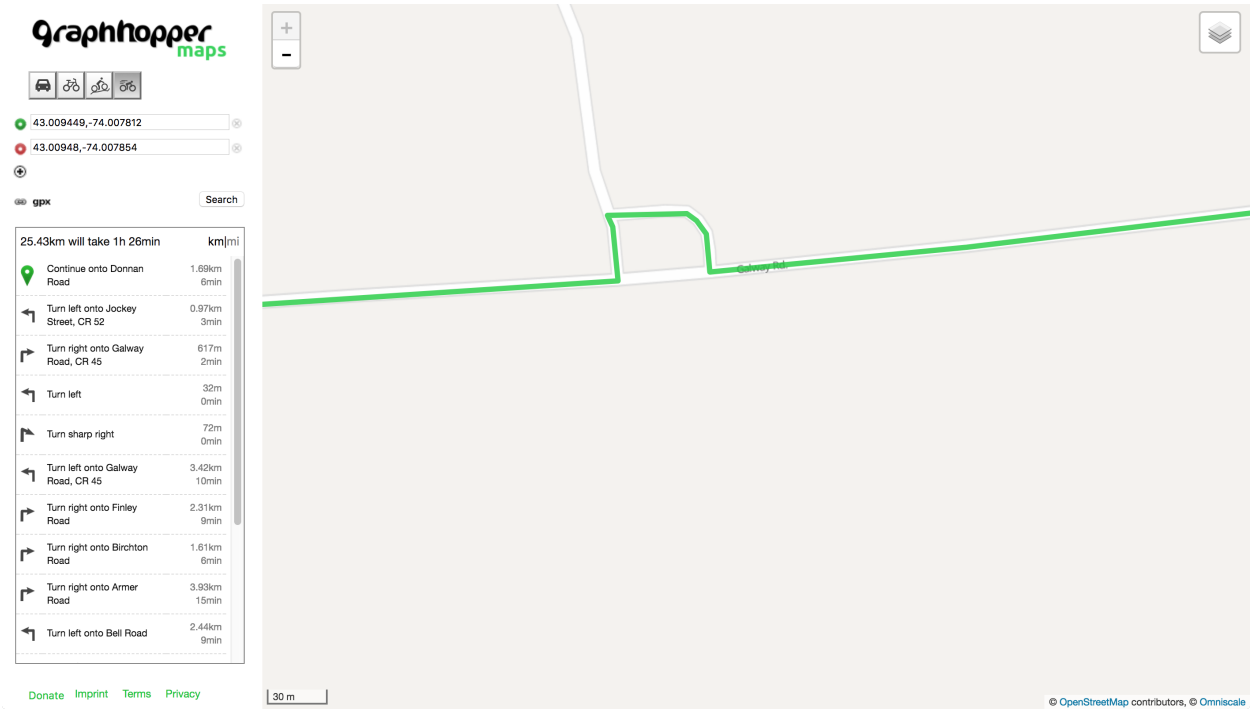


Figure 6: Quick turns in VVA example route. Inset in Figure 5

3.4 LS Implementation

Compared to the the VVA algorithm, the LS algorithm is more complex and so our implementation differs more from the pseudocode provided by Lu and Shahabi. There are many more implementation choices to be made. Recall that this algorithm works by connecting together attractive arcs with shortest paths known as blank path segments.

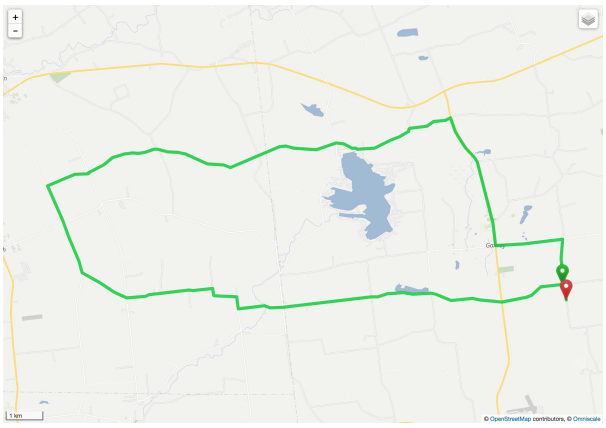
Implicitly defined in the LS algorithm is an object which represents the solution built up through iterations. We call this object a "route" and provide a unified interface for adding and removing arcs from this path. When adding and removing arcs, internally the object maintains the blank path segment invariant by calculating shortest paths and storing these paths. When it is time to return the actual path to GraphHopper, the Route object simply iterates over stored attractive arcs and shortest paths (blank path segments) in order. Since we are using a contraction hierarchy shortest path algorithm to compute the blank path segments, we recursively "unpack" any shortcuts (to get the original roads) before returning the solution to GraphHopper.

Our Candidate Arc Set (CAS) computation also differs in the way we spatially fetch arcs. We perform a breath first search starting at our start node only continuing our search outward if a given road is inside of our pruning ellipse. When the search returns, we have a list of all arcs that are contained solely inside of the pruning ellipse. We compute CAS feasibility of these arcs using the same contraction hierarchy shortest path algorithm used to calculate blank path segments. Our scores and costs are identical to those used in our VVA implementation.

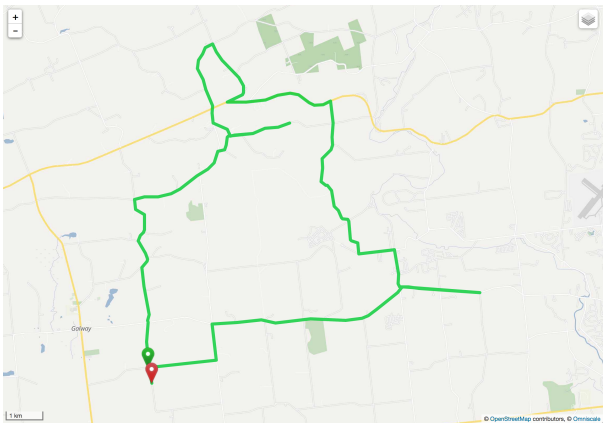
3.4.1 Implementation Observations

We ran our ILS implementations on OSM data of upstate New York to examine the generated routes. Figure 5 shows an example route generated by the VVA algorithm. Since this algorithm is deterministic, running the same query multiple times gives the same result. In this case, the generated route contains three quick turns in succession which can be dangerous for cyclists as they need to cross traffic lanes (Figure 6).

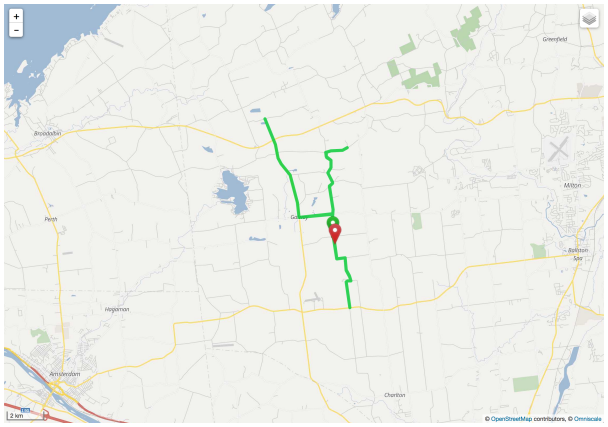
However, since the LS algorithm is randomized, running the same query multiple times produces different routes. Figure 7a shows a circular route. However, a route with these characteristics is not always generated by the algorithm. Running the same query may produce a route such as Figure 7b. The route in Figure 7b contains two subpaths which extend outward and return on the same path like cul-de-sacs. In the most extreme case, shown in Figure 7c, the route is solely composed of these "backtracking" subpaths. Backtracking occurs because attractive arcs are glued together by shortest paths. The shortest path back



(a) Circular route



(b) Route with some backtracking



(c) Route with excess backtracking

Figure 7: Example routes generated by our LS implementation with GraphHopper.

after taking an attractive arc may be the same path taken to get to the arc's start.

This backtracking shown in Figure 7 may be undesirable for cyclists. While riding on the same road more than once is not inherently undesirable for recreational cyclists, this can pose a safety issue. Following a route with excess backtracking may result in U-turns which can be dangerous for cyclists. However, not all backtracking creates U-turns.

Our implementation the LS algorithm in GraphHopper lead us to the following observations about the algorithm:

1. LS does not avoid backtracking when creating blank path segments or when computing arc feasibility.

2. LS tries to get as close to the cost budget as possible.

3. LS puts very few restrictions on what is considered an attractive arc.

4. LS does not penalize turns.

3.5 Our LS Variants

We introduce a few variants of LS to address the observations explained in Section 3.4.1.

3.5.1 Budget Allowance

The LS local search algorithm makes the greedy choice to insert a candidate arc at the smallest blank path segment in the route. This function continuously inserts candidate arcs until the CAS is empty or the cost budget is exhausted. This means that the path returned by LS will normally be very close to the maximum cost.

The intuition is if the initial route generated by LS is very close to the budget, then there may not be enough budget remaining to make big changes to the route. Therefore, the ILS may get stuck in a local optimum. The "budget allowance" variant aims to solve this by leaving more budget for later iterations of the search. Given a fixed percentage $0 < p < 1$, this variant ensures that LS is only allowed to use $p \cdot RemainingBudget$ when constructing the path at any given iteration. This variant addresses observation 2.

3.5.2 Incremental Budget

The "incremental budget" variant is similar to the "budget allowance" variant and aims to solve the same problem. However, instead of using a fixed budget percentage, it has a minimum budget percentage p_{min}. Over the course of the ILS iterations, the allowed budget scales from p_{min} to 1 in increments of $(1 - p_{min})/iterations$. The intuition is that while we want to save budget for later iterations, we shouldn't heavily restrict the budget as we near the termination of the ILS. This variant addresses observation 2.

3.5.3 Arc Restrictions

This variant changes how arcs are chosen to be included in a CAS to address observation 3. In the baseline LS implementation, attractive arcs are arcs whose score is greater than zero. This variant takes in two parameters $minRoadLength$ and $minRoadScore$. An arc is only considered attractive and added to the CAS if its distance in meters is greater than $minRoadLength$ and its score is greater than $minRoadScore$.

The intuition behind these restrictions is that the algorithm should not route to an attractive arc that is very short and has a meager score. Similarly, for a particular arc, the distance spent to traverse it should be worthwhile and this is generally true of longer arcs with higher scores.

3.5.4 No Backtracking

This variant attempts to address observation 1 shown in Figure 7 by "blacklisting" roads from the shortest path computations. As the algorithm builds up intermediate solutions, we keep track of all the arcs currently in the solution using a HashSet. We stop the shortest path algorithm from using any blacklisted roads during its search. When calculating the blank path segment away from the attractive arc, we need to blacklist not only the roads in the solution but the roads in the first blank path segment as well. This approach has two key implementation details which we address.

First, blacklisting roads may break the shortest path computation. In a connected graph there is always some shortest path between any two nodes. However, if we restrict which roads are allowed in the search then it is possible that we may have no shortest path. For example, consider an attractive arc at the end of a dead end road. Computing the first blank path segment to the arc will succeed but we cannot take the same path back so we have no return segment. In the case where we have no available blank path segment, we set the total path cost to infinity. This means that the arc will no longer be included in the CAS since it cannot feasibly update any arcs.

Second, this blacklisting process does not work well with contraction hierarchies. Recall that a contraction-hierarchy shortest-path algorithm traverses over contracted "shortcut" edges in the graph. The actual returned shortest path is recreated by finding the original roads which these shortcuts skip over. When determining if a road is blacklisted in our shortest path traversal, we need to check whether our current road is a shortcut and if it is, make sure none of the roads it skips are also blacklisted. This is quite challenging since a shortcut may skip multiple roads and may skip other shortcut edges. This means that we need to recursively "unpack" a shortcut (and any skipped shortcuts) before we can determine if we should avoid the arc. This is effectively undoing all the pre-computation that is done when the contraction hierarchy is initialized. In order to avoid this problem, our no-backtracking variant does not use a contraction hierarchy shortest-path algorithm and instead uses bidirectional Dijkstra's algorithm which is slower.

4 Data

We ran a series of experiments to evaluate the performance of the VVA algorithm, the LS algorithm, and our LS variants. This section discusses our data set and data collection process.

4.1 Map Data

Our mapping data set is a OpenStreetMap file[5] corresponding to 350 square kilometers centered around Galway, NY. We chose this dataset because it is relatively small and is moderately road dense. The size of the road network was also chosen to make an Integer Programming evaluation feasible. See Section 6 for more information on this approach. The free online tool *BBBike Extract* [1] was used to obtain the data. When parsed by GraphHopper, the internal graph representation contains 2425 directed arcs and 982 nodes.

4.2 Data Collection

We ran 500 trials of each algorithm configuration (Table 2) fixing the start location, the cost budget, and the number of ILS iterations. To achieve circular routes, the start and end location was fixed at GPS position $(43.009327, -74.009166)$, the center of our OSM data set. The cost budget was fixed at 40 kilometers. The number of iterations was fixed at 100. Our experiments use unit scoring. Thus, if two roads of different length have the same priority value by GraphHopper then they have the same score which is its priority value directly. These choices are mostly arbitrary. The iteration number and cost budget were chosen to be similar to the experimental tests run by Verbeeck et al. and Lu and Shahabi.

The ILS algorithms were modified to record the current solution score and elapsed time at each iteration. These values were written to a single CSV file for each algorithm configuration. For each configuration, average scores and average times were calculated using pivot tables with Python and Pandas[6]. The experimental runs were performed on a computer with 4 Intel Xeon E5620 processors and 16GB of RAM running Ubuntu 16.04.03 LTS Desktop.

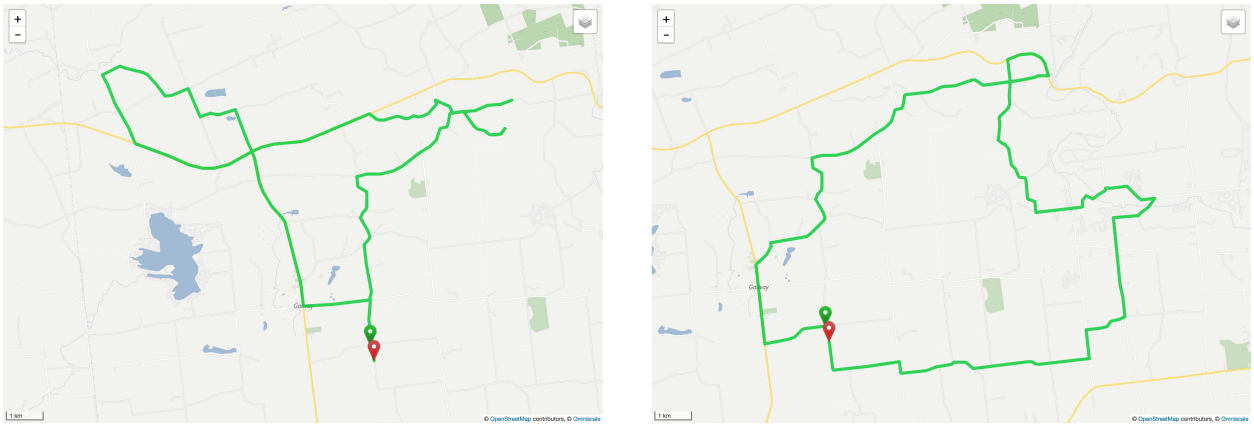
5 Experimental Results

First, we present images of example routes generated by our LS variants (Figure 8). In Section 5.1 we present our performance data using the iteration number as the cutoff. Lastly, in Section 5.2 we analyze a

[5] In Protocolbuffer Binary Format (.pbf). PBF is an alternative to the XML format which provides better compression.
[6] A Python package for data manipulation and analysis.

Algorithm	Extra Parameters
VVA	$maxDepth = 20$
LS	N/A
LS + (Budget Allowance)	$p_{min} = 0.5$
LS + (Incremental Budget)	$p_{min} = 0.5$
LS + (Arc Restrictions)	$minRoadLength = 1km$, $minRoadScore = 0.5$
LS + (No Backtracking)	N/A
LS + (Budget Allowance) + (Arc Restrictions)	$minRoadLength = 1km$, $minRoadScore = 0.5$, $p_{min} = 0.5$

Table 2: Experimental algorithm configurations



(a) Example route generated by LS + (Arc Restrictions)

(b) Example route generated by LS + (No Backtracking)

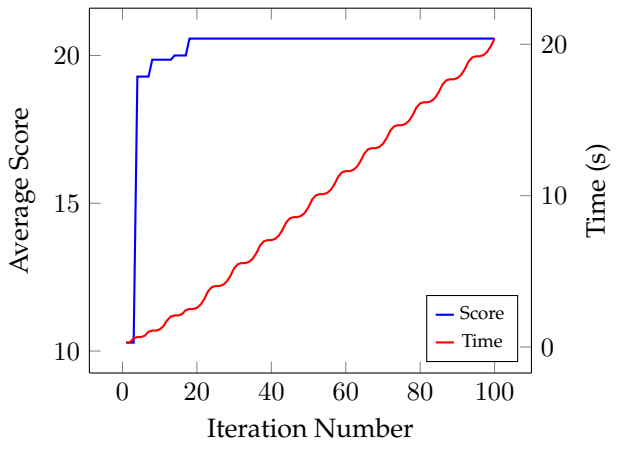
Figure 8: Example routes generated by our LS variants implemented with GraphHopper.

more efficient ILS stopping criterion.

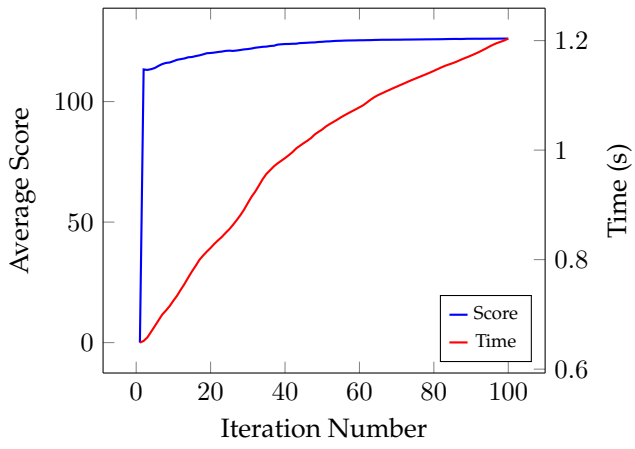
5.1 Iteration Cutoff

Table 3 shows the average times and scores of each algorithm configuration after 100 iterations. Independent plots of score and time versus iteration number are shown in Figure 9b. A combined plot of variant score versus log of time is shown in Figure 10.

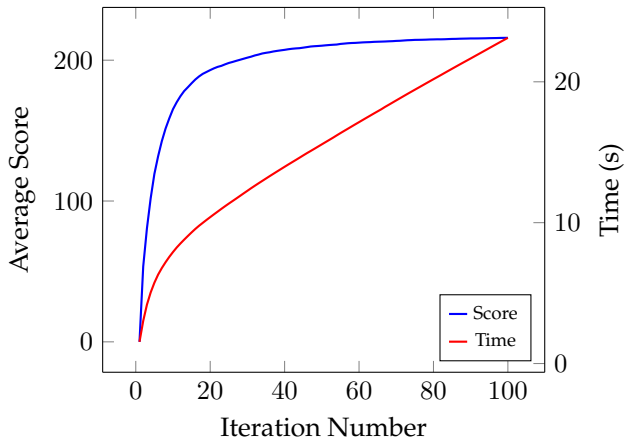
Our results validate the performance of the baseline LS algorithm compared to the VVA algorithm. VVA has a number of substantial limitations. First, VVA has small score improvement at each iteration because it completes a number of idle iterations with no improvement (Figure 9a). The "wavy" time plot is a consequence of the ILS perturbation phase. Recall that if no route improvement is found, VVA increases the number of contiguous arcs that it removes. This means that on these iterations, the DFS search must find a longer path which takes more time. VVA has small score improvement because it simply runs DFS and



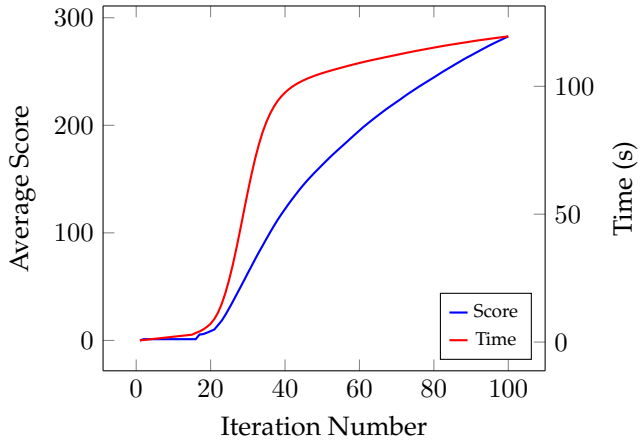
(a) VVA



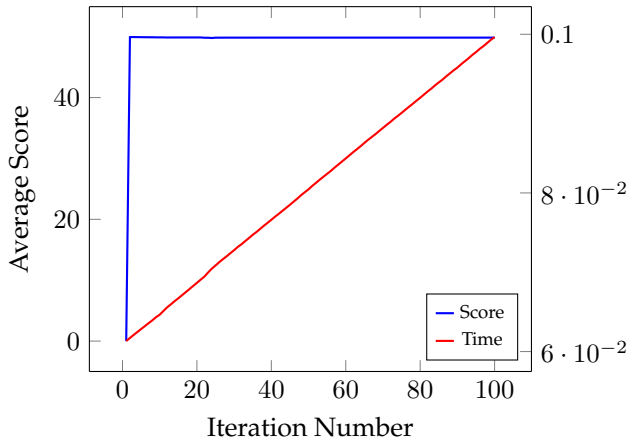
(b) LS



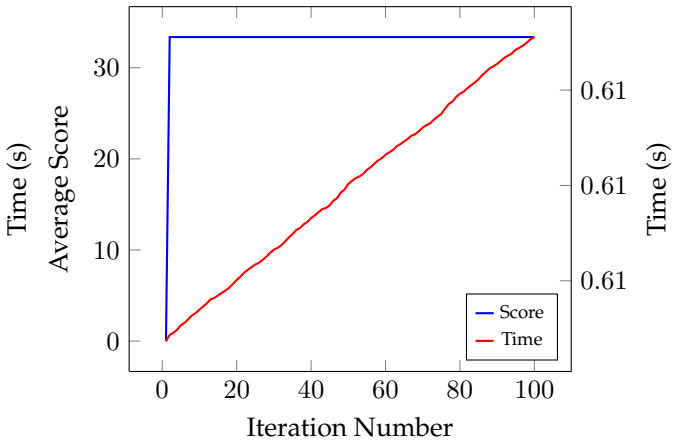
(c) LS + (Budget Allowance)



(d) LS + (Incremental Budget)



(e) LS + (Arc Restrictions)



(f) LS + (No Backtracking)

Figure 9: Algorithm performance with unit scoring.

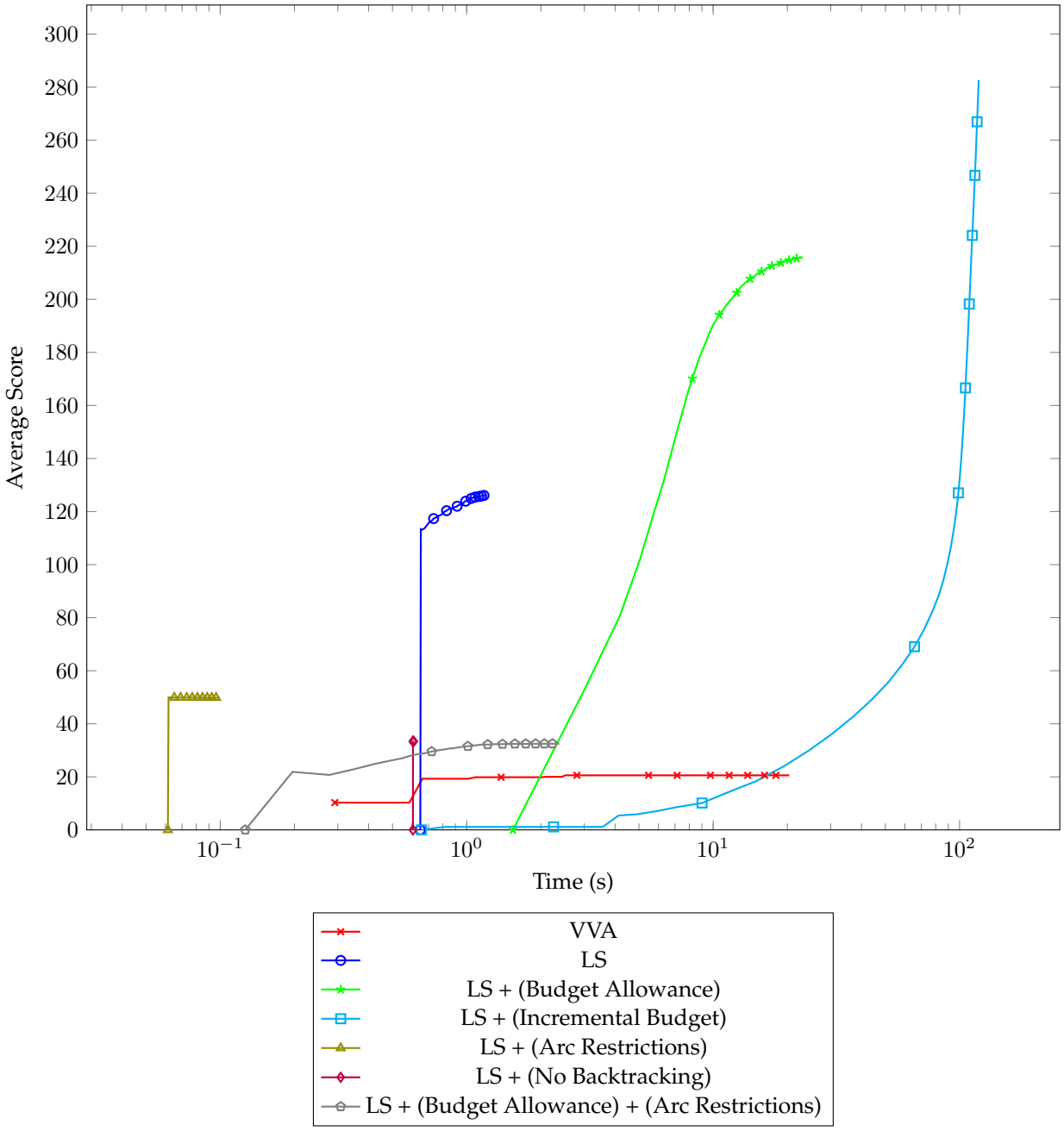


Figure 10: Performance of all algorithms using unit-scoring.

Algorithm	Score	Time (s)	Score/Time Ratio
VVA	20.57	20.37	1.00
LS	126.13	1.20	105.10
LS + (Budget Allowance)	215.87	23.12	9.33
LS + (Incremental Budget)	282.66	119.52	2.36
LS + (Arc Restrictions)	49.85	0.09	553.88
LS + (No Backtracking)	33.36	0.60	55.6
LS + (Budget Allowance) + (Arc Restrictions)	32.49	2.37	13.70

Table 3: Algorithm performance after 100 iterations with unit scoring.

checks to see if the solution after inserting an arc has improved score and is still within the cost budget. It does not consider the value or cost of the path segment being removed. Secondly, VVA has slow iteration because it has a large DFS search space. Even with a maximum depth search parameter, substantial feasibility checking is required, especially for road dense areas. At the end of 100 iterations, our VVA implementation produces a route in 20 seconds (Table 3). Third, VVA requires all-pairs shortest path to be precomputed which can be infeasible for large graphs. This is not an issue with our implementation since we are using contraction hierarchies which requires less pre-computation.

Our LS baseline implementation produces a route with 6 times the score of VVA in less than 1.5 seconds (Table 3). In addition, it performs very few idle iterations (Figure 9b). This shows that the spatial pruning techniques and heuristics for modifying the solution work well together to both improve overall score and reduce the time required. LS finds an initial solution which is better than VVA's final route and slowly improves over the subsequent iterations.

The data shows that our intuition behind the budget allowance variant was correct. Saving cost budget for later iterations generates a route with 70% score improvement when compared to the baseline algorithm. However, this variant pays a big penalty in time because the route generation time is nearly 20 times longer. The incremental budget variant produces an even higher score than budget allowance, but the time required to produce such a route is 100 times that of the LS baseline. Since the remaining budget is not decreasing sharply after the first iteration, these variants have to spend time computing and updating larger CASs.

The two final LS variants, arc restrictions and no backtracking, have the same fault. They run much faster than the baseline but this is because after the first iteration they do no work. Both variants pose strict limitations on which arcs are allowed in the CAS. After the first iteration, the initial CAS gets pruned to the empty set so no route changes can be made. Both of these variants beat VVA's score at a minuscule fraction of the required time. However, neither comes close to LS baseline's score. Combining budget allowance

Algorithm	Score	Time (s)	Score/Time Ratio
VVA	19.28	1.01	19.08
LS	113.93	0.67	170.0
LS + (Budget Allowance)	192.95	10.39	18.57
LS + (Incremental Budget)	1.14	1.29	0.88
LS + (Arc Restrictions)	49.92	0.06	832
LS + (No Backtracking)	33.37	0.61	54.70
LS + (Budget Allowance) + (Arc Restrictions)	30.80	0.88	35

Table 4: Algorithm performance with unit scoring and score-cutoff.

and arc restrictions variants does not lead to a variant with high score that runs fast. While its score is above the VVA algorithm, it is even lower than the no backtracking variant.

If we consider score to time ratio, then the arc restrictions variant wins out among all the algorithms. This ratio can be interpreted as the efficiency of the algorithm given its time usage. The arc restrictions variant has a score to time ratio of over 500, 5 times better than the baseline LS. While the variant's final score is not as good as the baseline, it can produce a route roughly half as good in a small fraction of the time.

5.2 Score Cutoff

The ILS trials in Section 5.1 are naive because they use a fixed iteration number as the algorithm stopping criterion. Many of these algorithms spend later iterations idle with no score improvement. Instead of using a fixed stopping criterion, we can terminate the algorithm based on score improvement over time.

We use the data from Section 5.1 to simulate the stopping point of the algorithms with this halting method. At each iteration, we calculate the percent change of the score from the previous iteration. If the score improvement is less than 1% for three consecutive iterations, then the algorithm terminates.

This technique culls large periods of wasteful time. The VVA algorithm's time drops from 20 seconds to 1 second while retaining 94% of its score (Table 4). Similarly, the LS baseline algorithm time is nearly halved while retaining 90% of its score. With this stopping criterion the VVA algorithm now runs runs roughly 300 milliseconds slower than the LS baseline. While there is no substantial time difference between the two, the score of the LS baseline is nearly 6 times as great. This suggests that the heuristics which LS uses to choose arcs are effective at producing a high scoring solution.

The assumption in this stopping criterion is that small score improvement over successive iterations is likely to continue. Therefore, it is not effective with ILS variants which have small value improvement at

the start. For example, this criterion quickly halts the LS Incremental Budget variant before it can get any meaningful score improvement. This can be seen in the small score slope at early iterations in Figure 9d. The short cutoff suggests that either the our chosen cutoff is not good or that too small a budget is not productive at improving the route.

With this stopping criterion, the score to time ratio improves for all but the incremental budget and no backtracking variants. This means that generally the algorithms are spending less idle time with no score improvement. The VVA ratio improves from 1 to 19 and the arc restriction variant ratio improves from 553 to 832.

6 Integer Programming Evaluation

In the previous section, we evaluate performance of the ILS algorithms using relative scores and times. Since these are heuristic algorithms, this approach is required because we do not know the optimal route given our data. Integer Programming (IP) is a model for finding *exact* answers to optimization problems such as the AOP. Given an exact solution to our AOP instance, we can assign our heuristic algorithm an absolute accuracy measure which shows how close the heuristic is to the best possible answer.

6.1 Integer Programming Definition

Many optimization problems maximize or minimize an objective given limited resources and competing constraints. If the objective can be written as a linear function of variables and the constraints written as equalities or inequalities on those variables then we have a Linear Programming (LP) problem [7]. The goal of the LP is to find some assignment to the variables that satisfies all the constraints while maximizing or minimizing the objective function. LP is used to model many problems such as planning, routing, and scheduling.

IP is a special case of LP where all the variables are constrained to use integer values [4]. While LP can be solved efficiently, the IP variant is NP-Hard. Therefore it is challenging to efficiently find exact answers to optimization problems using IP.

6.2 Integer Programming model for the AOP

Like many optimization problems, the AOP can be modeled using IP. Verbeeck et al. introduce an IP model for solving the AOP [17]. We modified the IP model introduced by Verbeeck et al. to only use a single

starting node rather than a set and removed the minimum score constraint. This is consistent with the choices made in our ILS implementations.

In the IP model, we are given a directed graph $G = (V, A)$, a start vertex $d \in V$, and a distance budget $B \in \mathcal{R}$. Each arc $a \in V$ has a cost $c_a \in \mathcal{R}$, a profit $p_a \in \mathcal{R}$ and a complementary arc $\bar{a} \in A \cup \{\emptyset\}$. If two arcs are available in two directions between a pair of vertices then they are complementary arcs. In addition, define $\delta(S)$ as the set of outgoing arcs from S to $V \setminus S$ and let $\lambda(S)$ be the set of incoming arcs to S from $V \setminus S$.

The decision variables variables of the IP are $x_a \in \{0, 1\}, \forall a \in A$ and $z_v \in \mathcal{Z}^{\geq}, \forall v \in V$. If $x_a = 1$ then arc a is chosen in the route otherwise it is 0. z_v represents the number of times a vertex v is visited by the route. The following is the IP model formulation:

$$\text{Maximize} \sum_{a \in A} p_a \cdot x_a$$

subject to:

$$\sum_{a \in A} c_a \cdot x_a \leq B \tag{1}$$

$$\sum_{a \in \lambda(v)} x_a - \sum_{a \in \delta(v)} x_a = 0 \quad \forall v \in V \tag{2}$$

$$\sum_{a \in \delta(v)} x_a = z_v \quad \forall v \in V \tag{3}$$

$$\sum_{a \in \delta(S)} x_a \geq \frac{\sum_{v \in S} z_v}{\sum_{v \in S} |\delta(v)|} \quad \forall S \subseteq V \setminus \{d\} \tag{4}$$

$$z_d = 1 \tag{5}$$

$$x_a + x_{\bar{a}} \leq 1 \quad \forall a \in A : \exists \bar{a} \in A \tag{6}$$

The objective maximizes the total collected score while Equations (1) to (6) are constraints. Equation (1) ensures that the total route cost is within the specific budget B. Equations (2) and (3) ensure that for vertices in the solution, the number of outgoing and incoming arcs are equal in the route and equal to the number of times a vertex is visited. These can be thought of as "flow constraints" limiting the route to contiguous arcs. With these constraints thus far, a valid solution to the IP may produce two disconnected loops. We want a single contiguous route. Equation (4) is a sub-tour constraint ensuring that there are no disconnected components of the route. This constraint operates on all *subsets* of the vertex set. Equation (5) ensures the the start vertex d is visited exactly once and Equation (6) ensures that an arc is taken in exactly one direction.

With Equation (6), the model is comparable to the no backtracking LS variant.

6.3 Gurobi Implementation

Given our road graph, we want to solve this IP model to find an exact solution to our AOP instance. We implement a Java program to model and solve these constraints using Gurobi [3], a commercial optimization solver. GraphHopper was used as a Java library to read the graph data. The graph data was then used to create the variables and constraints using the Gurobi Java API.

Equation (4) was the most challenging constraint to implement since it is a constraint on all subsets of vertices. Because V may be quite large and there are 2^n subsets of a set of size n, there are too many subsets to enumerate all possible constraints. Luckily, many optimization solvers have "lazy" constraints to address this problem. Lazy constraints operate differently than normal constraints because they are not immediately evaluated. Gurobi will ignore a lazy constraint until it finds a solution which satisfies the other constraints then it checks to see if its solution violates the lazy constraint. If so, then a new constraint based on the lazy constraint is added to the IP and Gurobi continues searching. This process continues until Gurobi finds a solution which does not violate the lazy constraint.

We implemented Equation (4) with a lazy constraint in Gurobi. This is done by giving a callback function to Gurobi which will be called whenever an feasible solution is found. The callback must determine if the lazy-constraint is violated and if so, add a new constraint to the model. This constraint is violated whenever we have a disconnected sub-tour. To check if we have a sub-tour, we find the vertices that we can reach from the start vertex using only the arcs chosen by the decision variables x_a by performing a DFS. If the reachable vertices are not all the vertices in the solution, then we have a disconnected sub-tour. The non-reachable vertices form the $S \subseteq V$ which violates Equation (4). This set is converted into a new constraint for the IP inside of the lazy constraint callback.

For the problem instances we experimented on, we were unable to get our Gurobi program halt and produce a valid optimal answer after multiple days of computation. We believe that there is some subtle bug in our implementation rather than lacking enough computation power because our Gurobi program succeeded for small test graphs. Verbeeck et al. solve their IP model using CPLEX, another optimization solver, and their graph takes about 6 hours. Since our graph is of comparable size, we would expect Gurobi to take around that long to solve our IP assuming similar computation power.

7 Conclusion

This research studied algorithms for generating bike routes for recreational road cyclists. We followed existing literature and formulated the problem as an instance of the AOP, a NP-Hard optimization problem. We focused on implementing and evaluating two ILS heuristic algorithms [17] [14] for the AOP using open source mapping tools. When using naive ILS stopping criteria, our experimental results validate previous work by [14] by showing that spatial techniques are effective at reducing the search space and speeding up the route generation time. When using smarter ILS stopping criteria, our results show that spatial techniques may not drastically speed up the search. However, the other heuristics proposed by [14] do lead to much higher scoring routes. Some of our proposed ILS variants lead to higher scoring routes but at the penalty of longer generation time. When comparing score to time ratio, our arc restrictions variant is substantially better than that of either baseline ILS algorithm. While our variant's route score is not the best, this ratio shows that it is very efficient at producing a good scoring route given its time usage.

7.1 Future Work

With more time, we hope to run additional experimental tests of the algorithm variants. In our tests, the road graph, starting location and cost budget are fixed. More tests should be run varying all three of these parameters to see if our results generalize. We hope to continue work on our Gurobi Integer Program solution to get an optimal route score as a baseline. This will allow us give absolute accuracy measurements of the algorithms as opposed to relative comparisons.

Road scoring mechanisms have much room for improvement. In our research, we used GraphHopper's built-in bike preferability value as the road score. This choice was practical as it allowed us to focus on the algorithms themselves instead of building the road graph. However, since this scoring relies on metadata from OpenStreetMaps, this scoring may be inaccurate. Further research could work on improving road scoring by using other datasets such as road popularity among cyclists. In addition, changing the scoring metrics may change how these algorithms perform.

All of our experimental tests were performed on powerful desktop computers. Future research could work on implementing these ILS algorithms on a mobile phone. With some performance tuning, we think that it is possible to generate routes in real time on a phone.

7.2 Acknowledgements

I would like to thank David Frey of the Union College Computer Science Department for helping me set up the computing resources to run my experiments. I would like to thank Robin ("boldtrn") of the GraphHopper open source project for answering my questions online.

References

[1] OpenStreetMap extracts — BBBike.org. https://extract.bbbike.org/. Visited Mar 16, 2018.

[2] GraphHopper Routing Engine. https://github.com/graphhopper/graphhopper. Visited Nov 2, 2017.

[3] Gurobi Optimization - The State-of-the-Art Mathematical Programming Solver. http://www.gurobi.com/. Visited Mar 17, 2018.

[4] IBM Knowledge Center: What is integer programming? https://www.ibm.com/support/knowledgecenter/SSSA5P_12.6.1/ilog.odms.ide.help/OPL_Studio/opllanguser/topics/opl_languser_shortTour_IP_what.html. Visited Mar 9, 2018.

[5] OpenStreetMap Wiki. http://wiki.openstreetmap.org/wiki/Develop. Visited Nov 13, 2017.

[6] Cecilia Bergman and Juha Oksanen. Optimization of circular cycling routes based on mobile sports tracking application data. *Computing*, 1(2):190–206, 2015.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

[8] Damianos Gavalas, Charalampos Konstantopoulos, Konstantinos Mastakas, Grammati Pantziou, and Nikolaos Vathis. Approximation algorithms for the arc orienteering problem. *Information Processing Letters*, 115(2):313–315, 2015.

[9] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. *Experimental Algorithms*, pages 319–333, 2008.

[10] Michel Gendreau and Jean-Yves Potvin. *Handbook of Metaheuristics*, volume 2. Springer, 2010.

[11] Bruce L Golden, Larry Levy, and Rakesh Vohra. The orienteering problem. *Naval research logistics*, 34 (3):307–318, 1987.

[12] Aldy Gunawan, Hoong Chuin Lau, and Pieter Vansteenwegen. Orienteering problem: A survey of recent variants, solution approaches and applications. *European Journal of Operational Research*, 255(2): 315–332, 2016.

[13] Gilbert Laporte and Silvano Martello. The selective travelling salesman problem. *Discrete Applied Mathematics*, 26(2-3):193–207, 1990.

[14] Ying Lu and Cyrus Shahabi. An arc orienteering algorithm to find the most scenic path on a large-scale road network. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 46. ACM, 2015.

[15] Viswanath Nagarajan and R Ravi. The directed orienteering problem. *Algorithmica*, 60(4):1017–1030, 2011.

[16] Wouter Souffriau, Pieter Vansteenwegen, Greet Vanden Berghe, and Dirk Van Oudheusden. The planning of cycle trips in the province of East Flanders. *Omega*, 39(2):209–213, 2011.

[17] Cédric Verbeeck, Pieter Vansteenwegen, and E-H Aghezzaf. An extension of the arc orienteering problem and its application to cycle trip planning. *Transportation Research Part E: Logistics and Transportation Review*, 68:64–78, 2014.

Appendices

A LS Algorithm

This appendix discusses the details of the LS algorithm not covered in Section 2.4. Section A.1 explains the path generation algorithm and how it in the LS ILS algorithm. Section A.2 explains the heuristic metrics used to determine which arcs should be added and removed when performing the ILS.

A.1 Path Generation

Algorithm 6 is the local search heuristic used by the LS algorithm (Algorithm 7). Its goal is to produce a path which connects the start vertex s with the destination vertex d whose total cost is within the budget $dist$ and total score is greater than $minProfit$. The algorithm builds the path by choosing candidate arcs from the CAS A.

Algorithm 6 first instantiates a fake arc starting and ending at the specified endpoints with a cost and score of 0 (Line 1). This fake arc is used to instantiate the solution to return, $route$ (Line 2). It then obtains a set of arcs to insert by filtering the CAS A by choosing arcs whose quality ratio is higher than the average (Line 3). While there are still possible arcs left to insert and the path has budget left, arcs are continuously removed from the CAS and inserted into the current solution $route$ (Lines 4 to 15). The algorithm inserts these candidate arcs into the path using a greedy approach. It chooses the closest blank path segment in the solution to insert the arc into the path (Lines 6 to 12).

Algorithm 6: generatePath(s, d, $dist$, $minProfit$, A)

Data: s: a start node of the path,
d: the end node of the path,
$dist$: the path's budget,
$minProfit$: minimum score of the path,
A: candidate arc set to choose arcs from.
Result: a path which fits the specified criteria.

1 $a_f \leftarrow (s, d, 0, 0)$ `// Arc with endpoints s & d with cost & score of 0`
2 $route \leftarrow \{a_f\}$
3 $arcs \leftarrow$ all arcs from A whose quality ratio is above the average
4 **while** *$arcs$ is not empty and $route.cost < dist$* **do**
5 $e \leftarrow$ remove random arc from $arcs$
6 $l \leftarrow$ empty blank path segment
7 $minDist \leftarrow 0$
8 **for** $l_i \in$ *blank path segments of* $route$ **do**
9 $dist \leftarrow (l_i.start \rightsquigarrow e \rightsquigarrow l_i.end).cost$
10 **if** $dist < minDist$ **then**
11 $l \leftarrow l_i$
12 $minDist \leftarrow dist$
13 $path \leftarrow (l.start \rightsquigarrow e \rightsquigarrow l.end)$
14 **if** $path.cost \leq dist - route.cost + l.cost$ **then** `// Our path can feasibly replace` l
15 insert $path$ into $route$ at blank path segment l
16 **if** $route.score > minProfit$ **then**
17 **return** *route*
18 **else**
19 **return** *empty route*

Algorithm 7 uses the ILS framework and generates the final bike route. First, the algorithm checks to see if the shortest path from the start to the destination is within the budget and if so then it runs the ILS. If not, it returns an empty solution (Lines 1 to 2). The ILS first initializes a fake arc with endpoints s & d and a cost of $dist$ and a score of 0 (Line 4) then computes the CAS of this arc (Line 5). This arc is used to initialize the temporary solution (Line 6).

While the time limit t has not elapsed, the algorithm chooses arcs from the solution to be removed based on their improve potential, removes them from the solution, then uses $generatePath$ to find a new path which closes the gap (Line 7-Line 11). If $generatePath$ can find a path to close the gap, then it needs to update the CAS of all the arcs in the solution. For the new arcs from $generatePath$ being added to the solution, the candidate arc sets must be computed (Line 18). On the other hand, arcs already in the solution must have their CASs updated (Line 20) since the remaining budget will have changed by adding the new path segment.

A.2 Arc Choice Heuristics

These heuristic scoring metrics are used by the LS algorithm to guide the perturbation and path generation phases of the ILS. Quality Ratio is defined for an arc from a CAS of a ILS solution. It is used to determine which arcs from the CAS will be chosen to add to the route to better improve its score. Improve Potential is used to determine which arcs to remove from the current solution so that new paths are more likely to improve the score of the route.

B ILS Implementation Code

This appendix contains Java code of our GraphHopper implementation.

B.1 VVA Code

Listing 1: `code/ils/vva/Arc.java`

```
1 package com.graphhopper.routing.ils.vva;
2
3 import java.util.Objects;
4
5 /**
```

Algorithm 7: ILS-LS(t, s, d, $dist$, G)

Data: t: a time,
s: the start node of the path,
d: the end node of the path,
$dist$: the maximum cost of the route,
G: the graph of the road network.
Result: a path

if $(s \rightsquigarrow d).cost > dist$ **then**
    return *empty route*
else
    $a_f \leftarrow (s, d, dist, 0)$ // Arc with endpoints s & d with cost $dist$ and score 0
    $a_f.CAS \leftarrow computeCAS(G, \{\}, s, d, dist)$
    $solution \leftarrow \{a_f\}$
    while *t seconds have not elapsed* **do**
        $arcs \leftarrow$ all arcs from $solution$ whose improve potential is above the average
        $e \leftarrow$ remove a random arc from $arcs$
        $b_1 \leftarrow solution.cost + e.cost$ // Budget after removing e from solution
        $path \leftarrow generatePath(e.pre, e.post, b_1, e.score, e.CAS)$
        if $path$ *is not empty* **then**
            remove e from $solution$
            insert $path$ into $solution$ between $e.pre$ and $e.post$
            for $a \in route$ **do**
                $b_2 \leftarrow solution.cost + a.cost$ // Budget after removing a from solution
                if $a \in path$ *or* $a = e.pre$ *or* $a = e.post$ **then**
                    $a.CAS = computeCAS(G, a.CAS, a.pre, a.post, b_2)$
                else
                    $a.CAS = updateCAS(G, a.CAS, a.pre, a.post, b_1, b_2)$
    return *route*

Algorithm 8: QualityRatio($a.pre$, $a.post$, a_c)

Data: a_c: arc from candidate arc set,
$a.pre$: previous arc in solution,
$a.post$: next arc in solution.
Result: a number.

$score \leftarrow (a.pre \rightsquigarrow a_c \rightsquigarrow a.post).score$
$cost \leftarrow (a.pre \rightsquigarrow a_c \rightsquigarrow a.post).cost$
return $score/cost$

Algorithm 9: ImprovePotential(a)

Data: a: a solution arc
Result: a number.

$score \leftarrow 0$
$maxDist \leftarrow 0$
$dist \leftarrow (a.pre \rightsquigarrow a \rightsquigarrow a.post).cost$
for $e \in a.CAS$ **do**
    $score \leftarrow score + (e.score - a.score)$
    $maxDist \leftarrow max(maxDist, (a.pre \rightsquigarrow e \rightsquigarrow a.post).cost)$
return $score/(maxDist - dist)$

```
 * Class which contains metadata about a particular edge in the graph. Used by
 * {@link Route} and {@link VVAIteratedLocalSearch}
 */
class Arc {
    final int edgeId, baseNode, adjNode;
    final double cost, score;

    Arc(int edgeId, int baseNode, int adjNode, double cost, double score) {
        this.edgeId = edgeId;
        this.baseNode = baseNode;
        this.adjNode = adjNode;
        this.cost = cost;
        this.score = score;
    }

    @Override
    public String toString() {
        return "Arc{" +
                "edgeId=" + edgeId +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if(this == o) return true;
```

```
        if(o == null || getClass() != o.getClass()) return false;
        Arc arc = (Arc) o;
        return edgeId == arc.edgeId &&
                baseNode == arc.baseNode &&
                adjNode == arc.adjNode;
    }

    @Override
    public int hashCode() {

        return Objects.hash(edgeId, baseNode, adjNode);
    }
}
```

Listing 2: `code/ils/vva/Route.java`

```
package com.graphhopper.routing.ils.vva;

import com.carrotsearch.hppc.IntHashSet;
import com.graphhopper.routing.ils.IlsPath;
import com.graphhopper.routing.weighting.Weighting;
import com.graphhopper.storage.Graph;

import java.util.ArrayList;
import java.util.List;

/**
 * Object which represents a path created by the {@link VVAIteratedLocalSearch} algorithm.
 */
final class Route {
    private List<Arc> arcs;
    private IntHashSet edges;
    private double cost;
    private double score;

    Route() {
        arcs = new ArrayList<>();
        edges = new IntHashSet();
    }

    // Copy constructor
    private Route(Route route) {
        cost = route.cost;
        score = route.score;
        arcs = new ArrayList<>(route.arcs);
```

```
30          edges = route.edges.clone();
31      }
32
33
34      void addEdge(int edgeId, int baseNode, int adjNode, double cost, double score) {
35          arcs.add(new Arc(edgeId, baseNode, adjNode, cost, score));
36          edges.add(edgeId);
37          this.cost += cost;
38          this.score += score;
39      }
40
41      void removeEdge(int edgeId) {
42          for(int i = arcs.size() - 1; i >= 0; i--) {
43              Arc arc = arcs.get(i);
44              if(arc.edgeId == edgeId) {
45                  arcs.remove(i);
46                  edges.remove(edgeId);
47                  cost -= arc.cost;
48                  score -= arc.score;
49                  break;
50              }
51          }
52      }
53
54      Arc removeEdgeIndex(int index) {
55          Arc arc = arcs.remove(index);
56          edges.remove(arc.edgeId);
57          cost -= arc.cost;
58          score -= arc.score;
59          return arc;
60      }
61
62      void clear() {
63          arcs.clear();
64          edges.clear();
65          cost = 0;
66          score = 0;
67      }
68
69      Route copy() {
70          return new Route(this);
71      }
72
73      boolean containsEdge(int edgeId) {
74          return edges.contains(edgeId);
75      }
```

```
76
77      void insertRoute(Route other, int index) {
78          arcs.addAll(index, other.arcs);
79          edges.addAll(other.edges);
80          cost += other.cost;
81          score += other.score;
82      }
83
84      void blacklist(Route other) {
85          edges.addAll(other.edges);
86      }
87
88      IlsPath getPath(Graph graph, Weighting costWeighting, Weighting scoreWeighting, int s, int d) {
89          IlsPath path = new IlsPath(graph, costWeighting, scoreWeighting);
90          for(Arc arc : arcs) {
91              path.processEdge(arc.edgeId, arc.adjNode, arc.edgeId);
92          }
93          return (IlsPath) path
94                  .setEndNode(d)
95                  .setFromNode(s)
96                  .setFound(!arcs.isEmpty());
97      }
98
99      public double getCost() {
100         return cost;
101     }
102
103     public double getScore() {
104         return score;
105     }
106
107     public int length() {
108         return arcs.size();
109     }
110
111 }
```

Listing 3: `code/ils/vva/VVAIteratedLocalSearch.java`

```
1  package com.graphhopper.routing.ils.vva;
2
3  import com.graphhopper.routing.AbstractRoutingAlgorithm;
4  import com.graphhopper.routing.DijkstraBidirectionCH;
5  import com.graphhopper.routing.Path;
6  import com.graphhopper.routing.RoutingAlgorithm;
```

```
7  import com.graphhopper.routing.ils.BikePriorityWeighting;
8  import com.graphhopper.routing.ils.IlsAlgorithm;
9  import com.graphhopper.routing.ils.IlsPath;
10 import com.graphhopper.routing.ils.Iteration;
11 import com.graphhopper.routing.util.DefaultEdgeFilter;
12 import com.graphhopper.routing.util.EdgeFilter;
13 import com.graphhopper.routing.util.TraversalMode;
14 import com.graphhopper.routing.weighting.Weighting;
15 import com.graphhopper.storage.Graph;
16 import com.graphhopper.util.EdgeExplorer;
17 import com.graphhopper.util.EdgeIterator;
18 import com.graphhopper.util.PMap;
19 import com.graphhopper.util.Parameters;
20
21 import static com.graphhopper.util.Parameters.Routing.*;
22
23 /**
24  * Routing Algorithm which implements the bike route Iterated Local Search algorithm from the following paper:
25  * https://www.sciencedirect.com/science/article/pii/S1366554514000751
26  */
27 public class VVAIteratedLocalSearch extends AbstractRoutingAlgorithm implements IlsAlgorithm {
28
29     private final double MAX_COST;
30     private final double MIN_COST;
31     private final int MAX_DEPTH;
32     private final int MAX_ITERATIONS;
33
34     private Graph CHGraph; // CH Dijkstra search
35     private EdgeFilter levelEdgeFilter; // Used for CH Dijkstra search
36     private Weighting scoreWeighting;
37
38     private boolean isFinished = false;
39     private int s, d;
40     private Iteration[] iterations;
41     private EdgeFilter bikeEdgeFilter;
42
43     /**
44      * @param graph specifies the graph where this algorithm will run on
45      */
46     public VVAIteratedLocalSearch(Graph graph, Weighting weighting,
47                                   EdgeFilter levelEdgeFilter, PMap params) {
48         super(graph.getBaseGraph(), weighting, TraversalMode.EDGE_BASED_1DIR);
49
50         CHGraph = graph;
51         this.levelEdgeFilter = levelEdgeFilter;
52         scoreWeighting = new BikePriorityWeighting(flagEncoder);
```

```
53          bikeEdgeFilter = new DefaultEdgeFilter(flagEncoder);
54
55          MAX_COST = params.getDouble(MAX_DIST, DEFAULT_MAX_DIST);
56          MIN_COST = params.getDouble(MIN_DIST, DEFAULT_MIN_DIST);
57          MAX_DEPTH = params.getInt(SEARCH_DEPTH, DEFAULT_SEARCH_DEPTH);
58          MAX_ITERATIONS = params.getInt(Parameters.Routing.MAX_ITERATIONS, DEFAULT_MAX_ITERATIONS);
59
60          iterations = new Iteration[MAX_ITERATIONS];
61      }
62
63      @Override
64      public Path calcPath(int from, int to) {
65          checkAlreadyRun();
66          s = from;
67          d = to;
68          return runILS();
69      }
70
71      private Path runILS() {
72          Route solution = initialize();
73          solution = improve(solution);
74          isFinished = true;
75          return getPath(solution);
76      }
77
78      private IlsPath getPath(Route solution) {
79          return solution.getPath(graph, weighting, scoreWeighting, s, d);
80      }
81
82      private Route improve(Route solution) {
83          long start = System.currentTimeMillis();
84          Route newPath = new Route();
85          int a = 1, r = 1, count = 0;
86          while(count < MAX_ITERATIONS) {
87              double score = getPath(solution).getScore();
88              Route temp = solution.copy();
89              int size = temp.length();
90
91              if(r > size) {
92                  r = 1;
93              }
94
95              if(a + r > size - 1) {
96                  r = size - 1 - a;
97              }
98
```

```
99              // Remove arcs a - r
100             double minScore = 0;
101             int startId = s, endId = d;
102             for(int i = 0; i < r; i++) {
103                 Arc arc = temp.removeEdgeIndex(a - 1);
104                 minScore += arc.score;
105
106                 if(i == 0) {
107                     startId = arc.baseNode;
108                 }
109
110                 if(i == r - 1) {
111                     endId = arc.adjNode;
112                 }
113             }
114
115             // Don't allow search to traverse roads already in our path
116             newPath.blacklist(temp);
117             if(localSearch(newPath, startId, endId, MAX_COST - temp.getCost(),
118                     minScore, MAX_DEPTH)) {
119                 temp.insertRoute(newPath, a - 1);
120                 solution = temp;
121                 a = 1;
122                 r = 1;
123             } else {
124                 a++;
125                 r++;
126             }
127
128             long elapsed = System.currentTimeMillis() - start;
129             iterations[count] = new Iteration(score, elapsed / 1000.0);
130
131             // Clear temp path so we can use it again
132             newPath.clear();
133             count++;
134         }
135
136         return solution;
137     }
138
139     private Route initialize() {
140         Route route = new Route();
141
142         if(!localSearch(route, s, d, MAX_COST, 0, MAX_DEPTH)) {
143             route.clear();
144         }
```

```
145
146         return route;
147     }
148
149     private boolean localSearch(Route route, int s, int d, double dist,
150                                 double minProfit, int maxDepth) {
151         if(maxDepth == 0) {
152             return false;
153         }
154
155         // Using edgeExplorer from baseGraph for traversal (non-CH version)
156         EdgeExplorer explorer = graph.createEdgeExplorer(bikeEdgeFilter);
157         EdgeIterator edgeIterator = explorer.setBaseNode(s);
158
159         while(edgeIterator.next()) {
160             int currentEdge = edgeIterator.getEdge();
161
162             if(route.containsEdge(currentEdge)) {
163                 continue;
164             }
165
166             double edgeCost = edgeIterator.getDistance();
167             int nextNode = edgeIterator.getAdjNode();
168
169             double remainingDist = dist - edgeCost;
170             double shortestDist = shortestPath(nextNode, d);
171
172             if(shortestDist >= remainingDist) {
173                 continue;
174             }
175
176             double edgeScore = scoreWeighting
177                     .calcWeight(edgeIterator, false, nextNode);
178
179             route.addEdge(currentEdge, s, nextNode, edgeCost, edgeScore);
180
181             if(nextNode == d &&
182                     route.getCost() >= MIN_COST &&
183                     route.getScore() > minProfit) {
184                 return true;
185             } else if(localSearch(route, nextNode, d, remainingDist,
186                     minProfit, maxDepth - 1)) {
187                 return true;
188             }
189
190             route.removeEdge(currentEdge);
```

```
191         }
192
193         return false;
194     }
195
196     /**
197      * Returns the shortest distance in meters between two nodes of the graph.
198      */
199     private double shortestPath(int s, int d) {
200         RoutingAlgorithm search =
201                 new DijkstraBidirectionCH(CHGraph,
202                         weighting, TraversalMode.NODE_BASED)
203                         .setEdgeFilter(levelEdgeFilter);
204
205         Path path = search.calcPath(s, d);
206         return path.getDistance();
207     }
208
209     // Unused
210     @Override
211     public int getVisitedNodes() {
212         return 0;
213     }
214
215     @Override
216     protected boolean finished() {
217         return isFinished;
218     }
219
220     // Unused
221     @Override
222     protected Path extractPath() {
223         return null;
224     }
225
226     @Override
227     public Iteration[] getIterationInfo() {
228         return iterations;
229     }
230 }
```

B.2 LS Code

Listing 4: `code/ils/ls/Arc.java`

```java
package com.graphhopper.routing.ils.ls;

import com.graphhopper.util.PointList;

import java.util.ArrayList;
import java.util.List;

/**
 * Class which contains metadata about a particular edge in the graph. Used by
 * the ILS algorithms.
 * <p>
 * In the ILS-CAS algorithm this represents an "attractive arc".
 */
public class Arc {
    public static final int FAKE_ARC_ID = -1;

    public final int edgeId, baseNode, adjNode;
    public final double cost, score;
    public final PointList points; // Points along the arc

    public double improvePotential, qualityRatio; // Metrics used by ILS algorithm
    private List<Arc> cas; // Candidate Arc Set of this arc

    /**
     * Constructor for creating a new Arc object.
     *
     * @param edgeId   The ID of the current edge in the graph.
     * @param baseNode The node ID of the first node which this arc connects.
     * @param adjNode  The node ID of the second nod which this arc connects.
     * @param cost     The distance of the road, in meters.
     * @param score    The score of the arc.
     * @param points   Points on the map of the arc.
     */
    public Arc(int edgeId, int baseNode, int adjNode, double cost, double score, PointList points) {
        this.edgeId = edgeId;
        this.baseNode = baseNode;
        this.adjNode = adjNode;
        this.cost = cost;
        this.score = score;
        this.points = points;
        improvePotential = -1;
        qualityRatio = -1;
        cas = new ArrayList<>();
    }

```

```
46      @Override
47      public String toString() {
48          return "Arc{" +
49                  "edgeId=" + edgeId +
50                  '}';
51      }
52
53      /**
54       * Gets the Candidate Arc Set of the current Arc.
55       *
56       * @return CAS
57       */
58      public List<Arc> getCas() {
59          return cas;
60      }
61
62      /**
63       * Updates the Candidate Arc Set of the current Arc.
64       *
65       * @param cas Candidate Arc Set to update.
66       */
67      public void setCas(List<Arc> cas) {
68          this.cas = cas;
69      }
70
71      @Override
72      public boolean equals(Object o) {
73          if(this == o) return true;
74          if(o == null || getClass() != o.getClass()) return false;
75
76          Arc arc = (Arc) o;
77
78          return edgeId == arc.edgeId && baseNode == arc.baseNode && adjNode == arc.adjNode;
79      }
80
81      @Override
82      public int hashCode() {
83          int result = edgeId;
84          result = 31 * result + baseNode;
85          result = 31 * result + adjNode;
86          return result;
87      }
88  }
```

Listing 5: `code/ils/ls/normal/Route.java`

```java
package com.graphhopper.routing.ils.ls.normal;

import com.graphhopper.routing.Path;
import com.graphhopper.routing.ils.IlsPath;
import com.graphhopper.routing.ils.ls.Arc;
import com.graphhopper.routing.weighting.Weighting;
import com.graphhopper.storage.Graph;
import com.graphhopper.util.EdgeIteratorState;
import com.sun.istack.internal.NotNull;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

/**
 * Object which represents a path created by the {@link LSIteratedLocalSearch}
 * algorithm.
 */
class Route implements Iterable<Arc> {

    private final Logger logger = LoggerFactory.getLogger(getClass());

    private Graph graph;
    private Weighting timeWeighting;
    private Weighting scoreWeighting;
    private ShortestPathCalculator sp;
    private final int s, d; // Start & End Node IDs
    private final double MAX_COST;

    private List<Arc> arcs; // List of "attractive arcs" in the Route
    private List<Path> blankSegments; // List of shortest paths connecting non-contiguous attractive arcs.
    private double cost, score; // Current

    private Route(ShortestPathCalculator shortestPathCalculator, Graph graph, Weighting timeWeighting,
                  Weighting scoreWeighting, int s, int d, double maxCost) {
        sp = shortestPathCalculator;
        arcs = new ArrayList<>();
        blankSegments = new ArrayList<>();
        cost = 0;
        score = 0;
        this.s = s;
        this.d = d;
        this.graph = graph;
```

```
46          this.timeWeighting = timeWeighting;
47          this.scoreWeighting = scoreWeighting;
48          MAX_COST = maxCost;
49      }
50
51      /**
52       * Static factory method for creating a new Route instance.
53       *
54       * @param sp              Interface which can calculate Shortest Paths.
55       * @param graph           Graph.
56       * @param weighting       Weighting used to calculate distance of added arcs.
57       * @param scoreWeighting  Weighting used to calculate score of added arcs.
58       * @param s               Start Node ID.
59       * @param d               End Node ID.
60       * @return New Route Instance.
61       */
62      static Route newRoute(@NotNull ShortestPathCalculator sp, @NotNull Graph graph,
63                            @NotNull Weighting weighting, @NotNull Weighting scoreWeighting,
64                            int s, int d, double maxCost) {
65          return new Route(sp, graph, weighting, scoreWeighting, s, d, maxCost);
66      }
67
68      /**
69       * Adds the specified Arc to the Route at the specified index.
70       * Throws {@link IndexOutOfBoundsException} if index <= 0 or index > {@link Route#length()}.
71       *
72       * @param index Index to insert Arc.
73       * @param arc   Arc to insert.
74       */
75      void addArc(int index, @NotNull Arc arc) {
76          int length = length();
77          if(index < 0 || index > length) {
78              throw new IndexOutOfBoundsException(String.format("index %d, length %d", index, length));
79          }
80
81          updatePathSegments(index, arc, arc);
82          arcs.add(index, arc);
83          cost += arc.cost;
84          score += arc.score;
85      }
86
87      /**
88       * Removes the first instance of the specified Arc from the Route.
89       *
90       * @param a Arc to remove.
91       * @return Index of removed Arc. Returns -1 if Arc was not in the current Route.
```

```
92          */
93         int removeArc(@NotNull Arc a) {
94             int index = arcs.indexOf(a);
95 
96             // Short circuit if Arc is not present in Route
97             if(index == -1) {
98                 throw new IllegalArgumentException("Arc is not in route!");
99             }
100 
101            // Remove two path segments surrounding Arc
102            Path segment1 = blankSegments.remove(index);
103            Path segment2 = blankSegments.remove(index);
104            cost -= segment1.getDistance();
105            cost -= segment2.getDistance();
106 
107            // If we have more than 1 arc we need to add a new path segment to join the Route
108            int length = length();
109            if(length > 1) {
110                int start = s;
111                int end = d;
112 
113                // Calculate start/end points for the new blank path segment
114                int prevIndex = index - 1;
115                if(prevIndex >= 0 && prevIndex <= length - 1) {
116                    start = arcs.get(prevIndex).adjNode;
117                }
118 
119                int nextIndex = index + 1;
120                if(nextIndex <= length - 1) {
121                    end = arcs.get(nextIndex).baseNode;
122                }
123 
124                // Calculate and add new path segment
125                Path segment = sp.shortestPath(start, end);
126                blankSegments.add(index, segment);
127                cost += segment.getDistance();
128            }
129 
130            arcs.remove(index);
131            cost -= a.cost;
132            score -= a.score;
133 
134            return index;
135        }
136 
137        /**
```

```
138         * Adds the specified Route to the current Route at the specified index.
139         * Throws {@link IndexOutOfBoundsException} if index <= 0 or index > {@link Route#length()}.
140         *
141         * @param index Index to insert Route.
142         * @param route Route to insert.
143         */
144        void insertRoute(int index, @NotNull Route route) {
145            int length = length();
146            if(index < 0 || index > length) {
147                throw new IndexOutOfBoundsException(String.format("index %d, length %d", index, length));
148            }
149
150            // Only add Route if it is non-empty
151            if(!route.isEmpty()) {
152                Arc first = route.arcs.get(0);
153                Arc last = route.arcs.get(route.length() - 1);
154
155                updatePathSegments(index, first, last);
156
157                // We need to remove the inserted routes starting and ending path segments
158                // We recalculate the new path segments below
159                Path head = route.blankSegments.remove(0);
160                Path tail = route.blankSegments.remove(route.blankSegments.size() - 1);
161                route.cost -= head.getDistance();
162                route.cost -= tail.getDistance();
163
164                score += route.score;
165                cost += route.cost;
166                arcs.addAll(index, route.arcs);
167                blankSegments.addAll(index + 1, route.blankSegments);
168            }
169        }
170
171        /**
172         * Updates the blank path segments at the specified index. Used when adding a new Arc to the route.
173         * <p>
174         * 1-2 --> 1-3-2
175         *
176         * @param index Index of blank path segments to update.
177         * @param left  Left bound of the Arc to be inserted.
178         * @param right Right bound of the Arc to be inserted.
179         */
180        private void updatePathSegments(int index, Arc left, Arc right) {
181            int length = length();
182            int start = s, end = d;
183
```

```
184             int startIndex = index - 1;
185             if(startIndex >= 0 && startIndex <= length - 1) {
186                 start = arcs.get(startIndex).adjNode;
187             }
188 
189             if(index <= length - 1) {
190                 end = arcs.get(index).baseNode;
191             }
192 
193             Path segment1 = sp.shortestPath(start, left.baseNode);
194             cost += segment1.getDistance();
195 
196             Path segment2 = sp.shortestPath(right.adjNode, end);
197             cost += segment2.getDistance();
198 
199             // If non-empty, remove the previous blank path segment before inserting the two new ones
200             if(length > 0) {
201                 Path removed = blankSegments.remove(index);
202                 cost -= removed.getDistance();
203             }
204 
205             blankSegments.add(index, segment2);
206             blankSegments.add(index, segment1);
207         }
208 
209 
210         /**
211          * Returns the current cost (distance) of the route in meters.
212          *
213          * @return Sum of edge distances in the Route.
214          */
215         double getCost() {
216             return cost;
217         }
218 
219         /**
220          * Returns the total score of the route.
221          *
222          * @return Sum of all attractive arc scores in the Route.
223          */
224         double getScore() {
225             return score;
226         }
227 
228         /**
229          * Returns the leftover budget after subtracting the current Route's cost.
```

```
230         *
231         * @return Remaining cost left in budget.
232         */
233        double getRemainingCost() {
234            return MAX_COST - cost;
235        }
236
237        /**
238         * Converts the Route into a Path object which GraphHopper can display on a map.
239         *
240         * @return Fully connected Path object
241         */
242        IlsPath getPath() {
243            IlsPath path = new IlsPath(graph, timeWeighting, scoreWeighting);
244
245            // If we have a fake arc return no path
246            if(contains(new Arc(Arc.FAKE_ARC_ID, s, d, 0, 0, null))) {
247                path.setFound(false);
248                return path;
249            }
250
251            for(int i = 0; i < blankSegments.size(); i++) {
252                Path blank = blankSegments.get(i);
253                for(EdgeIteratorState edge : blank.calcEdges()) {
254                    path.processEdge(edge.getEdge(), edge.getAdjNode(), edge.getEdge());
255                }
256
257                if(i < arcs.size()) {
258                    Arc arc = arcs.get(i);
259                    path.processEdge(arc.edgeId, arc.adjNode, arc.edgeId);
260                }
261            }
262
263            path.setEndNode(d)
264                    .setFromNode(s)
265                    .setFound(!isEmpty());
266
267            logger.debug("Route dist: " + path.getDistance() + " Route score: " + path.getScore());
268
269            return path;
270        }
271
272        private int length() {
273            return arcs.size();
274        }
275
```

```
276     /**
277      * Returns whether the Route has any arcs in it.
278      *
279      * @return True if contains arc, else false.
280      */
281     boolean isEmpty() {
282         return length() == 0;
283     }
284 
285     /**
286      * Returns a list of Arcs from the Route whose Improve Potential scores are above the average.
287      *
288      * @return Arc list.
289      */
290     List<Arc> getCandidateArcsByIP() {
291         List<Arc> result = new ArrayList<>();
292         double avgIP = 0;
293         for(Arc ca : arcs) {
294             calcImprovePotential(ca);
295             avgIP += ca.improvePotential;
296         }
297         avgIP /= arcs.size();
298 
299         for(Arc ca : arcs) {
300             if(ca.improvePotential >= avgIP) {
301                 result.add(ca);
302             }
303         }
304 
305         return result;
306     }
307 
308     /**
309      * Calculates the Improve Potential score of a given arc.
310      *
311      * @param arc Arc to calculate
312      */
313     private void calcImprovePotential(@NotNull Arc arc) {
314         int v1 = getPrev(arc);
315         int v2 = getNext(arc);
316 
317         double score = 0;
318         double maxDist = 0;
319 
320         double dist = sp.getPathCost(v1, v2, arc);
321 
```

```
322             for(Arc e : arc.getCas()) {
323                 score += e.score - arc.score;
324                 maxDist = Math.max(maxDist, sp.getPathCost(v1, v2, e));
325             }
326 
327             double result = score / (maxDist - dist);
328 
329             // Hacky fix for NaN values
330             if(Double.isNaN(result) || result < 0) {
331                 result = 0;
332             }
333 
334             arc.improvePotential = result;
335         }
336 
337         /**
338          * Returns the Node ID before the specified Arc in the Route.
339          *
340          * @param a Arc
341          * @return Node ID
342          */
343         int getPrev(@NotNull Arc a) {
344             if(!contains(a)) {
345                 throw new IllegalArgumentException("Arc is not in route!");
346             }
347 
348             int index = arcs.indexOf(a);
349             return (index - 1 >= 0) ? arcs.get(index - 1).adjNode : s;
350         }
351 
352         /**
353          * Returns the Node ID after the specified Arc in the Route.
354          *
355          * @param a Arc
356          * @return Node ID.
357          */
358         int getNext(@NotNull Arc a) {
359             if(!contains(a)) {
360                 throw new IllegalArgumentException("Arc is not in route!");
361             }
362 
363             int index = arcs.indexOf(a);
364             return (index + 1 <= length() - 1) ? arcs.get(index + 1).baseNode : d;
365         }
366 
367         /**
```

```
368         * Returns whether the specified Arc is in the Route.
369         *
370         * @param a Arc to query
371         * @return True if arc is in Route, else false.
372         */
373        boolean contains(@NotNull Arc a) {
374            return arcs.contains(a);
375        }
376
377        /**
378         * Adds the specified arc to the Route at the smallest blank path segment as long as it does not go over
                the
379         * specified budget.
380         *
381         * @param arc Arc to insert.
382         */
383        void insertArcAtMinPathSegment(@NotNull Arc arc) {
384            // We have at least 1 arc and 2 blank path segments
385            if(!isEmpty()) {
386                int pathIndex = -1;
387                double minPathValue = Double.MAX_VALUE;
388                // Find smallest blank path segment
389                for(int i = 0; i < blankSegments.size(); i++) {
390                    double value = blankSegments.get(i).getDistance();
391                    if(value < minPathValue) {
392                        minPathValue = value;
393                        pathIndex = i;
394                    }
395                }
396
397                int start = pathIndex == 0 ? s : arcs.get(pathIndex - 1).adjNode;
398                int end = pathIndex == length() ? d : arcs.get(pathIndex).baseNode;
399
400                if(sp.getPathCost(start, end, arc) <=
401                        getRemainingCost() + minPathValue) {
402                    addArc(pathIndex, arc);
403                }
404
405            } else if(sp.getPathCost(s, d, arc) <= getRemainingCost()) {
406                addArc(0, arc);
407            }
408        }
409
410        @Override
411        public Iterator<Arc> iterator() {
412            return arcs.iterator();
```

```
413     }
414 }
```

Listing 6: `code/ils/ls/Ellipse.java`

```
1  package com.graphhopper.routing.ils.ls;
2
3  import com.graphhopper.routing.ils.ls.normal.LSIteratedLocalSearch;
4  import com.graphhopper.util.DistanceCalc;
5  import com.graphhopper.util.Helper;
6  import com.graphhopper.util.shapes.BBox;
7  import com.graphhopper.util.shapes.GHPoint;
8  import com.graphhopper.util.shapes.Shape;
9  import sun.reflect.generics.reflectiveObjects.NotImplementedException;
10
11 /**
12  * Class which represents an Ellipse on the map. Used by the {@link LSIteratedLocalSearch} algorithm for
        restricting
13  * the search space.
14  * <p>
15  * Note: This does not fully implement the Shape interface!
16  */
17 public class Ellipse implements Shape {
18
19     private static DistanceCalc calc = Helper.DIST_EARTH;
20
21     private GHPoint focus1;
22     private GHPoint focus2;
23     private double radius;
24
25     public Ellipse(GHPoint focus1, GHPoint focus2, double radius) {
26         this.focus1 = focus1;
27         this.focus2 = focus2;
28         this.radius = radius;
29     }
30
31     @Override
32     public boolean intersect(Shape o) {
33         throw new NotImplementedException();
34     }
35
36     @Override
37     public boolean contains(double lat, double lon) {
38         return calc.calcDist(lat, lon, focus1.lat, focus1.lon) +
39                 calc.calcDist(lat, lon, focus2.lat, focus2.lon) <= radius;
```

```
40      }
41
42      @Override
43      public boolean contains(Shape s) {
44          throw new NotImplementedException();
45      }
46
47      @Override
48      public BBox getBounds() {
49          throw new NotImplementedException();
50      }
51
52      @Override
53      public GHPoint getCenter() {
54          throw new NotImplementedException();
55      }
56
57
58      @Override
59      public double calculateArea() {
60          throw new NotImplementedException();
61      }
62  }
```

Listing 7: `code/ils/ls/normal/ShortestPathCalculator.java`

```
1   package com.graphhopper.routing.ils.ls.normal;
2
3   import com.graphhopper.routing.Path;
4   import com.graphhopper.routing.ils.ls.Arc;
5   import com.sun.istack.internal.NotNull;
6
7   public interface ShortestPathCalculator {
8
9       /**
10       * Returns the shortest distance in meters between two nodes of the graph.
11       */
12      Path shortestPath(int s, int d);
13
14      /**
15       * Returns the total distance in meters of the path s --> arc --> d where "-->" is shortest path.
16       *
17       * @param s    Start node ID.
18       * @param d    End node ID.
19       * @param arc Arc.
```

```
20          * @return Distance in meters
21          */
22         double getPathCost(int s, int d, @NotNull Arc arc);
23
24     }
```

Listing 8: code/ils/ls/normal/LSIteratedLocalSearch.java

```
1   package com.graphhopper.routing.ils.ls.normal;
2
3   import com.carrotsearch.hppc.IntHashSet;
4   import com.graphhopper.routing.AbstractRoutingAlgorithm;
5   import com.graphhopper.routing.DijkstraBidirectionCH;
6   import com.graphhopper.routing.Path;
7   import com.graphhopper.routing.RoutingAlgorithm;
8   import com.graphhopper.routing.ils.BikePriorityWeighting;
9   import com.graphhopper.routing.ils.IlsAlgorithm;
10  import com.graphhopper.routing.ils.Iteration;
11  import com.graphhopper.routing.ils.ls.Arc;
12  import com.graphhopper.routing.ils.ls.Ellipse;
13  import com.graphhopper.routing.util.EdgeFilter;
14  import com.graphhopper.routing.util.TraversalMode;
15  import com.graphhopper.routing.weighting.Weighting;
16  import com.graphhopper.storage.Graph;
17  import com.graphhopper.util.*;
18  import com.graphhopper.util.shapes.GHPoint;
19  import com.graphhopper.util.shapes.GHPoint3D;
20  import com.graphhopper.util.shapes.Shape;
21  import com.sun.istack.internal.NotNull;
22  import com.sun.istack.internal.Nullable;
23  import org.slf4j.Logger;
24  import org.slf4j.LoggerFactory;
25
26  import java.util.ArrayList;
27  import java.util.List;
28  import java.util.Random;
29
30  import static com.graphhopper.util.Parameters.Routing.*;
31
32  /**
33   * Routing Algorithm which implements the bike route Iterated Local Search algorithm from the following paper:
34   * https://dl.acm.org/citation.cfm?id=2820835
35   */
36  public class LSIteratedLocalSearch extends AbstractRoutingAlgorithm implements ShortestPathCalculator,
        IlsAlgorithm {
```

```
37
38      private final Logger logger = LoggerFactory.getLogger(getClass());
39
40      // Constants passed in as parameters
41      private final double MIN_ROAD_SCORE;
42      private final int MIN_ROAD_LENGTH;
43      private final double MAX_COST;
44      private final int MAX_ITERATIONS;
45      private final long SEED;
46
47      private Graph CHGraph; // Graph used for CH Dijkstra search
48      private EdgeFilter levelEdgeFilter; // Used for CH Dijkstra search
49      private Weighting scoreWeighting; // Used for scoring arcs
50      private int s, d; // Start and End Node IDs
51      private Random random;
52      private Iteration[] iterations; // Keep track of score at each iteration
53
54      private boolean isFinished = false;
55
56      //////////////////////////////////////////
57      // TEST CODE
58      //////////////////////////////////////////
59      private final Mode MODE;
60      private final double BUDGET_PERCENTAGE;
61
62      private enum Mode {
63          NORMAL,
64          FIXED_PERCENTAGE_BUDGET,
65          INCREMENTAL_BUDGET,
66          NORMALIZED_SCORES;
67
68          static Mode getMode(int value) {
69              if(value >= 0 && value < values().length) {
70                  return values()[value];
71              }
72              throw new RuntimeException("Invalid mode specified!");
73          }
74      }
75
76      /**
77       * Creates a new ILS algorithm instance.
78       *
79       * @param graph           Graph to run algorithm on.
80       * @param weighting       Weighting to calculate costs.
81       * @param levelEdgeFilter Edge filter for CH shortest path computation
82       * @param params          Parameters map.
```

```
83          */
84         public LSIteratedLocalSearch(Graph graph, Weighting weighting,
85                                      EdgeFilter levelEdgeFilter, PMap params) {
86             super(graph.getBaseGraph(), weighting, TraversalMode.EDGE_BASED_1DIR);
87
88             CHGraph = graph;
89             this.levelEdgeFilter = levelEdgeFilter;
90             scoreWeighting = new BikePriorityWeighting(flagEncoder);
91
92             MAX_COST = params.getDouble(MAX_DIST, DEFAULT_MAX_DIST);
93             MAX_ITERATIONS = params.getInt(Parameters.Routing.MAX_ITERATIONS, DEFAULT_MAX_ITERATIONS);
94             MIN_ROAD_SCORE = params.getDouble(Parameters.Routing.MIN_ROAD_SCORE, DEFAULT_MIN_ROAD_SCORE);
95             MIN_ROAD_LENGTH = params.getInt(Parameters.Routing.MIN_ROAD_LENGTH, DEFAULT_MIN_ROAD_LENGTH);
96             SEED = params.getLong(Parameters.Routing.SEED, System.currentTimeMillis());
97
98             random = new Random(SEED);
99             iterations = new Iteration[MAX_ITERATIONS];
100
101            //////////////////////////////////////////
102            // TEST CODE
103            //////////////////////////////////////////
104            MODE = Mode.getMode(params.getInt(Parameters.Routing.MODE, DEFAULT_MODE));
105            BUDGET_PERCENTAGE = params.getDouble(Parameters.Routing.BUDGET_PERCENTAGE,
106                    Parameters.Routing.DEFAULT_BUDGET_PERCENTAGE);
107            if(MODE.equals(Mode.NORMALIZED_SCORES)) {
108                double SCORE_CUTOFF = params.getDouble(Parameters.Routing.SCORE_CUTOFF, DEFAULT_SCORE_CUTOFF);
109                scoreWeighting = new NormalizedBikePriorityWeighting(flagEncoder, SCORE_CUTOFF);
110            }
111
112            if(params.getBool(USE_SCALED_SCORES, false)) {
113                scoreWeighting = new ScaledBikePriorityWeighting(flagEncoder);
114            }
115        }
116
117        /**
118         * Calculates a route between the specified node IDs.
119         *
120         * @param from Start Node ID.
121         * @param to   End Node ID.
122         * @return Path
123         */
124        @Override
125        public Path calcPath(int from, int to) {
126            checkAlreadyRun();
127            s = from;
128            d = to;
```

```
129         return runILS();
130     }
131
132     /**
133      * Main algorithm loop
134      */
135     private Path runILS() {
136         long start = System.currentTimeMillis();
137         Route solution;
138         if(shortestPath(s, d).getDistance() > MAX_COST) {
139             solution = Route.newRoute(this, graph, weighting, scoreWeighting, s, d, MAX_COST);
140         } else {
141             solution = initializeSolution();
142             logger.info("Seed: " + SEED);
143             for(int i = 1; i <= MAX_ITERATIONS; i++) {
144                 double score = solution.getPath().getScore();
145                 logger.debug("Iteration " + i);
146                 List<Arc> arcRemovalPool = solution.getCandidateArcsByIP();
147                 logger.debug("Possible arcs to remove from solution: " + arcRemovalPool.size());
148
149                 int randomIndex = random.nextInt(arcRemovalPool.size());
150                 Arc arcToRemove = arcRemovalPool.remove(randomIndex);
151                 List<Arc> inheritedCas = arcToRemove.getCas();
152
153                 // Remaining budget after removing "arcToRemove" from solution
154                 double pathBudget = solution.getRemainingCost() + arcToRemove.cost;
155
156                 ////////////////////////////////////////
157                 // TEST CODE
158                 ////////////////////////////////////////
159                 if(MODE.equals(Mode.FIXED_PERCENTAGE_BUDGET)) {
160                     pathBudget = pathBudget * BUDGET_PERCENTAGE;
161                 } else if(MODE.equals(Mode.INCREMENTAL_BUDGET)) {
162                     double percent = ((double) i / MAX_ITERATIONS) * (1 - BUDGET_PERCENTAGE);
163                     pathBudget = (pathBudget * percent) + BUDGET_PERCENTAGE;
164                 }
165
166                 Route path = generatePath(solution.getPrev(arcToRemove), solution.getNext(arcToRemove),
167                         pathBudget, arcToRemove.score, inheritedCas);
168
169                 if(!path.isEmpty()) {
170                     logger.debug("Found path with with dist " + path.getCost());
171                     int index = solution.removeArc(arcToRemove);
172                     solution.insertRoute(index, path);
173                     for(Arc arc : solution) {
174                         // Remaining budget after removing "arc" from solution
```

```
175                         double newBudget = solution.getRemainingCost() + arc.cost;
176
177                         int startCAS = solution.getPrev(arc);
178                         int endCAS = solution.getNext(arc);
179
180                         if(path.contains(arc) || arc.adjNode == startCAS || arc.baseNode == endCAS) {
181                             // Using removed arc's CAS to compute next CAS (inherit)
182                             computeCAS(arc, inheritedCas, startCAS, endCAS, newBudget);
183                         } else {
184                             double oldBudget = solution.getRemainingCost() + arcToRemove.cost;
185                             updateCAS(arc, inheritedCas, startCAS, endCAS, newBudget, oldBudget);
186                         }
187                     }
188                 }
189
190                 long elapsed = System.currentTimeMillis() - start;
191                 iterations[i - 1] = new Iteration(score, elapsed / 1000.0);
192             }
193         }
194
195         isFinished = true;
196
197         return solution.getPath();
198     }
199
200     /**
201      * Creates a new Route, adds a fake arc, and computes first CAS.
202      *
203      * @return Route.
204      */
205     private Route initializeSolution() {
206         Route route = Route.newRoute(this, graph, weighting, scoreWeighting, s, d, MAX_COST);
207         // Add fake edge to start solution
208         Arc arc = new Arc(Arc.FAKE_ARC_ID, s, d, MAX_COST, 0, PointList.EMPTY);
209         computeCAS(arc, null, s, d, MAX_COST);
210         route.addArc(0, arc);
211
212         return route;
213     }
214
215     /**
216      * Computes the Candidate Arc Set for the specified start, end, and cost parameters.
217      *
218      * @param arc  Arc to set CAS on.
219      * @param cas  Current CAS. May be null.
220      * @param s    Start Node ID.
```

```
221         * @param d     End Node Id.
222         * @param cost  Cost allowance.
223         */
224        private void computeCAS(Arc arc, @Nullable List<Arc> cas, int s, int d, double cost) {
225            List<Arc> result = new ArrayList<>();
226
227            GHPoint focus1 = new GHPoint(nodeAccess.getLatitude(s), nodeAccess.getLongitude(s));
228            GHPoint focus2 = new GHPoint(nodeAccess.getLatitude(d), nodeAccess.getLongitude(d));
229            Ellipse ellipse = new Ellipse(focus1, focus2, cost);
230
231            // If we don't have a CAS yet
232            // Fetch arcs from the graph using spatial indices
233            if(cas == null) {
234                // Since s is one of the foci of our ellipse, it will always be contained in it.
235                // Use s as the node which we start our search
236                cas = getAllArcs(ellipse, s);
237            }
238
239            logger.debug("Starting to compute CAS! num arcs: " + cas.size() + " cost: " + cost);
240
241            outer:
242            for(Arc e : cas) {
243
244                // Basic restrictions on attractive arcs
245                if(e.score > MIN_ROAD_SCORE && e.cost > MIN_ROAD_LENGTH) {
246                    // Spatial-based feasibility checking
247                    for(GHPoint3D ghPoint3D : e.points) {
248                        if(!ellipse.contains(ghPoint3D.lat, ghPoint3D.lon)) {
249                            continue outer;
250                        }
251                    }
252
253                    // Check arc feasibility
254                    if(getPathCost(s, d, e) <= cost) {
255                        calcQualityRatio(e, s, d);
256                        result.add(e);
257                    }
258                }
259            }
260
261            logger.debug("Finished computing CAS! size: " + result.size());
262
263            arc.setCas(result);
264        }
265
266        /**
```

```
267         * Fetches all Arcs from the graph which are contained inside of the specified Shape.
268         *
269         * @param shape      Shape.
270         * @param startNode Node to start search from.
271         * @return Arc list.
272         */
273        private List<Arc> getAllArcs(final Ellipse shape, int startNode) {
274            logger.debug("Fetching arcs from graph!");
275            final List<Arc> arcs = new ArrayList<>();
276
277            BreadthFirstSearch bfs = new BreadthFirstSearch() {
278                final Shape localShape = shape;
279                final IntHashSet edgeIds = new IntHashSet();
280
281                @Override
282                protected boolean goFurther(int nodeId) {
283                    return localShape.contains(nodeAccess.getLatitude(nodeId), nodeAccess.getLongitude(nodeId));
284                }
285
286                @Override
287                protected boolean checkAdjacent(EdgeIteratorState edge) {
288                    if(localShape.contains(nodeAccess.getLatitude(edge.getAdjNode()), nodeAccess.getLongitude(edge.
                            getAdjNode()))) {
289                        int edgeId = edge.getEdge();
290                        if(!edgeIds.contains(edgeId)) {
291                            arcs.add(getArc(edge));
292                            edgeIds.add(edgeId);
293                        }
294                        return true;
295                    }
296                    return false;
297                }
298            };
299
300
301            bfs.start(outEdgeExplorer, startNode);
302
303            logger.debug("Got all arcs inside of ellipse! num: " + arcs.size());
304
305            return arcs;
306        }
307
308        /**
309         * Returns an Arc object instance from the specified EdgeIterator from the Graph.
310         *
311         * @param edgeIterator Edge
```

```
312         * @return Arc
313         */
314        private Arc getArc(EdgeIteratorState edgeIterator) {
315            int edge = edgeIterator.getEdge();
316            int baseNode = edgeIterator.getBaseNode();
317            int adjNode = edgeIterator.getAdjNode();
318            double edgeCost = edgeIterator.getDistance();
319
320            double edgeScore = scoreWeighting
321                    .calcWeight(edgeIterator, false, baseNode);
322
323            return new Arc(edge, baseNode, adjNode, edgeCost, edgeScore, edgeIterator.fetchWayGeometry(0));
324        }
325
326        /**
327         * Updates the Candidate Arc Set for the specified Arc.
328         *
329         * @param arc        Arc to update.
330         * @param s          Start Node Id.
331         * @param d          End Node Id.
332         * @param newBudget New allowable budget.
333         * @param oldBudget Old allowable budget.
334         */
335        private void updateCAS(@NotNull Arc arc, @NotNull List<Arc> cas, int s, int d, double newBudget, double
                oldBudget) {
336            // Restrict CAS using inherit property
337            if(newBudget < oldBudget) {
338                List<Arc> newCas = new ArrayList<>();
339                for(Arc e : cas) {
340                    // Remove any arc whose path is too big
341                    if(getPathCost(s, d, e) <= newBudget) {
342                        newCas.add(e);
343                    }
344                }
345                arc.setCas(newCas);
346            } else if(newBudget > oldBudget) {
347                computeCAS(arc, null, s, d, newBudget);
348            }
349        }
350
351        /**
352         * Computes the Quality Ratio for the specified Arc.
353         *
354         * @param arc Arc.
355         * @param s   Start Node ID.
356         * @param d   End Node ID.
```

```
357         */
358         private void calcQualityRatio(@NotNull Arc arc, int s, int d) {
359             Path sp1 = shortestPath(s, arc.baseNode);
360             Path sp2 = shortestPath(arc.adjNode, d);
361 
362             double value = 0;
363 
364             List<EdgeIteratorState> edges = sp1.calcEdges();
365             edges.addAll(sp2.calcEdges());
366 
367             for(EdgeIteratorState edge : edges) {
368                 value += scoreWeighting.calcWeight(edge, false, edge.getBaseNode());
369             }
370 
371             value += arc.score;
372             value /= (sp1.getDistance() + arc.cost + sp2.getDistance());
373 
374             if(Double.isNaN(value)) {
375                 value = 0;
376             }
377 
378             arc.qualityRatio = value;
379         }
380 
381         /**
382          * Returns a list of Arcs from the specified CAS whose Quality Ratio scores are above the average.
383          *
384          * @param cas CAS
385          * @return Arc list.
386          */
387         private List<Arc> getCandidateArcsByQR(List<Arc> cas) {
388             List<Arc> arcs = new ArrayList<>();
389             double avgQR = 0;
390             for(Arc ca : cas) {
391                 avgQR += ca.qualityRatio;
392             }
393             avgQR /= cas.size();
394 
395             for(Arc ca : cas) {
396                 if(ca.qualityRatio >= avgQR) {
397                     arcs.add(ca);
398                 }
399             }
400 
401             return arcs;
402         }
```

```
403
404     /**
405      * Generates a Route from the specified start and end nodes whose distance is less than the specified
             budget and
406      * total score is above the specified.
407      *
408      * @param s          Start Node Id.
409      * @param d          End Node Id.
410      * @param dist       Allowable budget.
411      * @param minProfit Minimum required score.
412      * @param cas        CAS
413      * @return Route. May be empty!
414      */
415     private Route generatePath(int s, int d, double dist, double minProfit, List<Arc> cas) {
416         logger.debug("Generating path! dist: " + dist + " minProfit: " + minProfit + " cas size: " + cas.size()
                 );
417         Route route = Route.newRoute(this, graph, weighting, scoreWeighting, s, d, dist);
418
419         List<Arc> arcs = getCandidateArcsByQR(cas);
420         while(!arcs.isEmpty() && route.getCost() < dist) {
421             int randomIndex = random.nextInt(arcs.size());
422             Arc e = arcs.remove(randomIndex);
423             route.insertArcAtMinPathSegment(e);
424         }
425
426         if(route.getScore() > minProfit) {
427             return route;
428         } else {
429             return Route.newRoute(this, graph, weighting, scoreWeighting, s, d, dist);
430         }
431
432     }
433
434     @Override
435     public double getPathCost(int s, int d, @NotNull Arc arc) {
436         return shortestPath(s, arc.baseNode).getDistance() + arc.cost +
437                 shortestPath(arc.adjNode, d).getDistance();
438     }
439
440     @Override
441     public Path shortestPath(int s, int d) {
442         RoutingAlgorithm search =
443                 new DijkstraBidirectionCH(CHGraph,
444                         weighting, TraversalMode.NODE_BASED)
445                         .setEdgeFilter(levelEdgeFilter);
446
```

```
447             return search.calcPath(s, d);
448         }
449 
450         // Unused
451         @Override
452         public int getVisitedNodes() {
453             return 0;
454         }
455 
456         @Override
457         protected boolean finished() {
458             return isFinished;
459         }
460 
461         // Unused
462         @Override
463         protected Path extractPath() {
464             return null;
465         }
466 
467         // Used for tracking progress of iterations
468         @Override
469         public Iteration[] getIterationInfo() {
470             return iterations;
471         }
472 }
```