Heuristics for Master Thesis

Håvard Notland

February 2020

1 Introduction

In the modern era, many jobs which were never imagined 10 years ago are starting to appear. One of these jobs is to collect and charge electric scooters.

In several major cities across the world, you can now rent an electric scooter on almost every street corner, and it's popularity has exploded in a very short amount of time. Locating a scooter and renting is easily done via an app on your mobile phone. Once your scooter has taken you to your destination, you may simply leave it on the sidewalk, as long as it is in within a designated zone. Later, someone else may pick up the scooter and continue to use it. Once it has run out of power, it needs to be collected and charged.

This logistics task can be tackled in various ways: Some companies hire designated drivers that pick up the scooters and take them to one of several charging stations throughout the city. More commonly used is a crowd-sourcing effort where anyone can sign up to become a scooter collector, and get paid per scooter collected, charged and returned to the streets. These independent contractors are commonly referred to as "Bird-catchers" or "Juicers". The names are derived from two of the larger rental companies "Birds" and "Lime".

A successful hunt for this Bird-catcher

Both solutions get the job done, but in either case one may be faced with one of two problems:
1: A company is competing with other scooter-renting companies in several cities. In order to gain the edge, they want to improve the cost and time it takes to charge their scooters, such that they can be more quickly available for use.
2: A self-employed "Bird-catcher" will be competing with other freelancers on collecting these valuable "scooter-bounties". It is first come first served, and the faster they are collected, the better.

In both cases, a solution could be to plan a route for collecting such that the distance travelled is minimized. How should you go about planning a route such that you collect as many scooters as you can in the shortest amount of distance?

1.1 Problem introduction

The task laid out in front of us has an obvious appliance in the field of algorithms. It is quite similar to other wide-studied problems, one of which is the Traveling Salesman Problem. In the Traveling Salesman Problem (TSP) you are given a set of cities that you wish to visit, and the distance between each city. In what order should you visit these cities such that the total distance covered is minimized?

TSP is usually formulated as a graph problem. In the field of mathematics and algorithms a graph is a structure amounting to a set of objects in which pairs of objects are in some way related. An object is abstracted as a vertex, or node, and the relations between vertices are known as edges, or arcs. Graphs

can be weighted on the edges, where a number is assigned to each edge. In TSP, a city is notated as vertex, and the roads between the cities are the edges. The edges are weighted such that the value each edge is assigned represents the distance between two respective cities.

Our problem is similar, but differs in some areas; The cities are replaced by electric scooters, and you are not able to "visit" every scooter in one go. There could be hundreds of scooters scattered across the city, and only a selected amount can be collected before returning to charging station is necessary. The area of focus can be one of many: One is to find the shortest trip that visits 20 scooters and returns to Depot. The focus on this thesis, however, will be on calculating all trips such that every scooter is collected in one of the trips.

1.2 Problem preliminaries

In practical terms, we will not be referencing the scooter-collecting aspect much further, but move on to a more general approach, that can solve a variety of real-life problems, one of which could be scooter-collecting. The things in common for these problems are logistics-related:

1: They have a large number of Points-of-Interest (POI) with a known location, and a primary location (Depot), which will be our starting point.
2: The POI and Depot are spread across a plane and all POI should be both pair-wise reachable and reachable from Depot.
3: The POI are stationary, and some limitation is in place such that after a certain number of POI are visited, it is necessary to return to our primary Depot location.

In order to induce a graph problem from this, we need a data set with graph-like components, such as a road map. Where roads can be represented as edges, and intersections as vertices. POI located on this road-map should have coordinates, or a direct correlation to an intersection or road.

In our heuristic we can be given any graph with a strongly connected component[1] with at least 1 POI and 1 Depot. One vertex must be marked as the Depot. We expect both the edges and vertices to be weighted. The edge-weights will represent the distance between the end-points of the edge. And the vertex-weights represent the number of POI located on any given vertex. Depending on the type of graph, some pre-processing must be done. The Depot vertex is not considered in these steps:

- If an edge in the graph is undirected, we duplicate the edge and make them directed both ways.

[1] A strongly connected component is a directed subgraph where all vertices are pair-wise reachable.

- If the graph is not connected, we induce our graph from the strongly connected component where the Depot is located.
- If the graph is not weighted on the edges, we assign each edge to have weight 1
- Vertices with weight more than 1 are split into new vertices with a new vertex for each weight value-point. The new vertices are connected with edges weighted 0 and share the same edges as the original vertex.
- If the graph is not complete[2], we calculate the shortest paths between all vertices with weight at least 1 and the Depot vertex. Then we induce a complete graph from these distances.
- If the graph is complete, but not without shortcuts[3], we do the same as the previous point.

The reason for performing these steps is to streamline our algorithm such that a simple greedy heuristic can be abstracted. The new graph obtained after these steps can be traversed much simpler than the original graph:

- Step 1 is needed so that we do not need to differentiate between directed and undirected edges.
- Step 2 ensures that our graph is strongly connected.
- Step 3 ensures that each node needs only be visited once. It removes the unpleasant predicament of arriving at a vertex where the weight on the vertex is more than the remaining capacity for the current trip.
- Step 4 gives us a much simpler starting point as the structure of the graph is similar to the preferred graph type used in TSP solving techniques. The complete graph is simply a presentation of a matrix holding all the shortest paths between vertices.
- Step 5 is just an extra insurance so that we never need to think about finding the shortest path in the graph.
- Step 6 essentially "removes" the zero-weighted nodes by not including them in our complete graph.

After completing these pre-processing steps we end up with structurally speaking, homogeneous data types. Which is a directed strongly connected graph and a complete graph of all POI's located in the network.

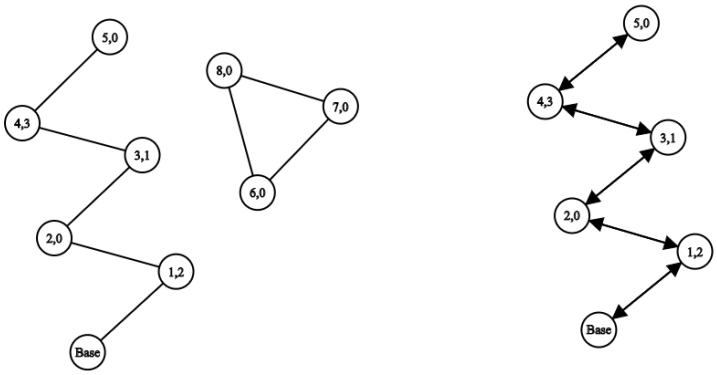
We expect the graph to be derived from a roadmap or a similar network, and in order to accomodate a varying format of the graph induced from the map data,

[2] A complete graph is a graph in which all vertices are pair-wise connected by an edge

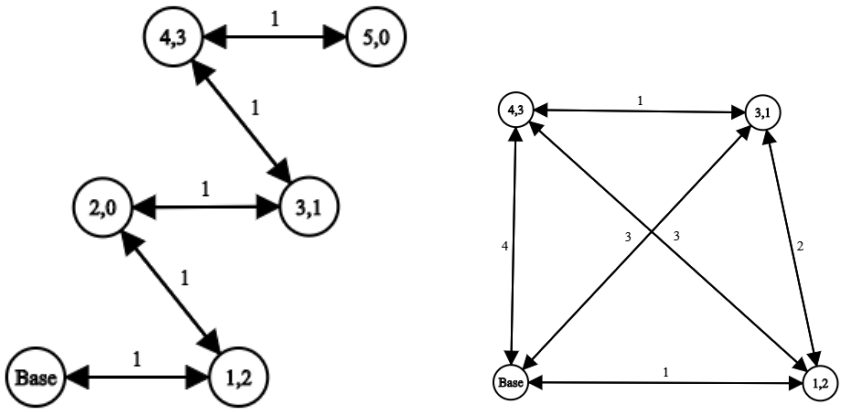
[3] A graph contains shortcuts if given 3 vertices $u, v, andw$ and a function $d(a, b)$ which indicates the distance between a and b: $d(u, v) + d(v, w) < d(u, w)$.

these steps are deemed as necessary. The data may be derived from a graph representing a road-network, which can be both undirected and directed (i.e. the existence of one-way streets), or it may only consist of vertices indicating the locations of our POI. Thus, we decrease the need to accomodate our algorithm to the input data.

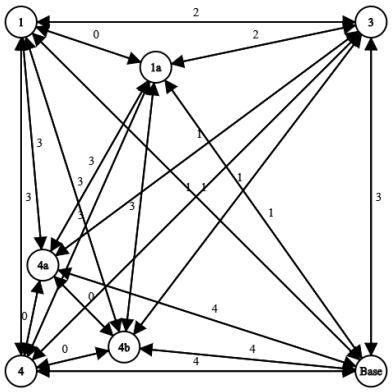
The order which these steps are performed in is somewhat arbitrary, but can affect the performance of the pre-processing heuristic. One might consider calculating the complete graph before splitting multi-weight vertices, as the distance data is identical for all duplicated vertices. The following example shows step-by-step actions taken when most cases apply:



Remove nodes not reachable from Depot, induce directed graph



assign edge-weights, Calculate complete graph,



	Base	1	1a	3	4	4a	4b
Base	0	1	1	3	4	4	4
1	1	0	0	2	3	3	3
1a	1	0	0	2	3	3	3
3	3	2	2	0	1	1	1
4	4	3	3	1	0	0	0
4a	4	3	3	1	0	0	0
4b	4	3	3	1	0	0	0

duplicate heavier nodes, induce matrix for readability

2 Problem formulation

In this section we introduce two problems that are very similar but have one key difference. The first problem wishes to minimize both the length for visiting all nodes, and the number of trips needed to visit all.

Minimizing the number of trips is trivial; it can be calculated as the size of the graph divided by the number of nodes that can be visited in a single trip. The reason for demanding this will be explained further on.

The other problem presented is identical to the first, but there is no need to minimize the number of trips. This presents an interesting dilemma. In the first problem, as long as you can continue your trip, you must. But in the second, you must consider if you should terminate your trip, or continue. Developing a heuristic for the second problem will therefore be more difficult, but will likely yield a smaller minimization than the first problem.

2.1 Minimize sum of length of trips, Minimize number of trips

Given a complete weighted graph G, a start node s, a trip-length k; Obtain a set of trips T which visits all of V(G) such that each trip (except the last trip) visits exactly k nodes (excluding s) and the total length of all trips is minimized. A trip is a subset of vertices of G, which starts and ends with s. The length of a trip is the sum of all weights of edges between all adjacent vertices in the trip subset.

$$
\begin{aligned}
&\textit{Input} : \textit{A weighted graph } G,\ \textit{vertex } s,\ \textit{trip} - \textit{length } k \\
&\\
&\textit{Problem} : \textit{Minimize the length of the set of trips} \\
&T : t_1, t_2, ..., t_n \textit{ where a trip } t_i \subseteq V(G) \\
&\textit{is an ordering of vertices } v_1, v_2, ..., v_m \\
&\textit{s.t. the following holds for any trip } t_i : \\
&1.\ \forall\ v\ \in V(G),\ v \in t_i \wedge v \notin T \setminus t_i \\
&2.\ \forall\ t_i\ \in T\ v_1 = v_m = s \\
&3.\ m\ = k + 2,\ |t_n| <= k + |v_i = s| \\
&\\
&\textit{The length of the set of trips is} \\
&\sum_{i=1}^{n} \sum_{p=1,q=2}^{m}\ w(v_p, v_q) \in t_i
\end{aligned}
\tag{1}
$$

This problem, we will call the Exact-k-Round Restricted Traveling Salesman Problem or EkRRTSP. A trip should minimize the total distance it travels. It is trivial to observe that this is an NP-Hard formulation as a specialized instance of this problem is the Traveling Salesman Problem. The problem becomes TSP if we set $k = |V(G)|$ and let the weight of all edges connected to s be weighted 0.

Note that we require each trip to visit the maximum number of nodes capable as long as there are at least k nodes left unvisited. This is a potential requirement to look at, as we may compare a heuristic that does not prohibit smaller trips and see if there is a possibility of getting a better performance.

2.2 Limit sum of length of trips, all trips are length at most k

Given a complete weighted graph G, a start vertex s, a trip-length k and bound X; Is there a set of trips T which visits all of V(G) such that the number of nodes visited excluding s is at most k and the total length of all trips is minimized?

$$
\begin{aligned}
&\textit{Input} : \textit{A weighted graph } G,\ \textit{vertex } s,\ \textit{trip} - \textit{length } k \\
&\\
&\textit{Problem} : \textit{Minimize the length of the set of trips} \\
&T : t_1, t_2, ..., t_n \textit{ where a trip } t_i \subseteq V(G) \\
&\textit{is an ordering of vertices } v_1, v_2, ..., v_m \\
&\textit{s.t. the following holds for any trip } t_i : \\
&1.\ \forall\ v\ \ \in V(G),\ v \in t_i \wedge v \notin T \setminus t_i \\
&2.\ \forall\ t_i\ \ \in T\ v_1 = v_m = s \\
&3.\ m\ \ <= k + 2 \\
&\\
&\textit{The length of the set of trips is} \\
&\sum_{i=1}^{n} \sum_{p=1,q=2}^{m}\ w(v_p, v_q) \in t_i
\end{aligned}
\tag{2}
$$

This problem we call the Restricted Traveling Salesman Problem, or RTSP, and is essentially the same as the previous with the only difference being the length of each trip no longer requires a total visit of k nodes if possible. This is likely to never yield a worse solution than EkRRTSP as it is able to give the same solution as EkRRTSP.

3 Solving techniques

For solving the problems presented we will be using some common techniques that are further explained in this section.

3.1 Greedy algorithms

For EkRRTSP we will be using an algorithm that falls under the category of greedy algorithms. Greedy algorithms are characterized by one simple logical heuristic: At any given stage in our algorithm, select the path that is the local optimum.

This reduces the need to compute exhaustively on data, but can lead to locking out an optimal solution. As such, greedy algorithms have a varying degree of success on problems. In some problems, we find the global optimum with a

greedy heuristic. Others can approximate the optimum at a lower-bound guarantee. And some may yield the worst possible result.

A problem closely resembling EkRRTSP, TSP, has a greedy heuristic: Go to the currently closest unvisited city. This heuristic does not always find the global optimum for TSP, and in fact it is possible to obtain the worst possible solution using this technique. A similar technique shall be applied for EkRRTSP, as a Depotline for other heuristics to compare to.

3.2 Clustering

In solving RTSP we will use a technique known as clustering. Clustering is commonly applied in the field of machine learning, as it is a tool for categorizing entities that have strong correlation. In our case the correlation is the distance between POI.

Clustering algorithms seek to group entities such that entities in the same group (or cluster) resemble each other more than entities of other groups. Cluster analysis may use a wide variety of measuring the likeness of entities, often more than a single measure.

For visualization purposes, the entities are placed on a plane where the closer two entities are to each other, the more they are alike. The likeness can therefore be measured as the distance between the entities. In RTSP the only measure of likeness will be the distance itself between them. So it lends an obvious appliance to the field of cluster analysis. Instead of categorizing entities in an unsupervised machine learning environment we use the technique to assign scooters to each trip.

There are many different types of clustering algorithms, which have their pros and cons. An important aspect that we need is the ability to limit the size of each cluster. As we want a cluster to be completed in a single trip. Hierarchical clustering techniques allow this.

3.2.1 Greedy clustering

As a benchmark, we will use a greedy clustering heuristic. Given a cluster size limit k we create clusters by repeatedly selecting the unclustered POI furthest away from depot, p and then selecting the closest k POIs from p. By selecting the POI furthest away, we find the area where all remaining POI are located. The line between depot and p, L becomes the radius of a circle where all POI are contained. We can then separate the POI's into two hemispheres of the circle, by the perpendicular line of $\emptyset$ through the circle. If any of the k closest POI's from p are located on the opposite hemisphere, we are left with two scenarios: Since we can choose to end a tour before visiting k POI's, we do not have to visit a POI, h located in the other hemisphere, and instead do it in a different tour.

This will often yield a worse performance by the triangle inequality property, as $dist(depot, p) + dist(p, h) + dist(h, depot) \leq dist(depot, p) + dist(p, depot) + dist(depot, h) + dist(h, depot)$. There is a counter example, though. Where choosing to visit h is a bad idea. If there are POI located in groups on opposite sides of depot. If the most remote group is of size k-1 and the other containing h is of size k, we end up travelling to the larger group twice, as we are unable to collect the remaining POI located near h.

To decide which one is the better rule to follow, we can to examine the worst-case outcomes for both, and see which has the worse performance, in relation to the other rule. Looking back on the circle created by depot and p, the closest possible POI not on the same hemisphere, t creates a right triangle by drawing a straight line between depot, p and t. We notate the length of the line a between depot and p, b between depot and t and c between p and t. t can be placed anywhere perpendicular to a, but the maximum difference between a and c occurs when t is equally distanced from depot as p. In this case, the triangle is an isosceles right triangle, where $b = a$ and $c = \sqrt{2}a$. If we choose to add t to the cluster the path for visiting p and t is a, c, b. Otherwise we return to depot first and travel a, a, b, b. The difference in distance travelled becomes $2a + 2b - a - b - c = a + b - c$. In an isosceles right triangle the ratio between the hypothenus and any other side is $\sqrt{2}$ or roughly 1.414. Thus, the increase in distance travelled if not choosing to add t to the same cluster is at most $a + b - c = 2a - c = 2a - 1.414a = 0.586a$.

Now, consider the possibility that we chose to visit t. Our current tour is maxed out upon visiting t so we return to depot. However, there were $k - 1$ other POI's located on the same position as t. Then, we need to travel b twice more to collect those POI's. If we chose to ignore t we could travel a, a, b, b and visit all POI's instead of a, c, b, b, b. The total increase then becomes $a + c + 3b - 2a - 2b = c + b - a = 1.414a + a - a = 1.414a$.

From this, we can argue that the former rule of not visiting POI's in the opposite hemisphere gives a better worst-case outcome. However, many more factors come into play that make it less straight forward. There is a mucher higher likelihood of the first example happening, as it occurs whenever there are less than $k - 1$ POI's in the same hemisphere, while the second example is much more specific. Also, we are not working with a pure euclidean distance model, but one based on a graph or some other form of map, with rules to follow. Thus, the characteristics of the isosceles right triangle seldom apply.

A way to combine these two choices is to use a branching technique whenever the choice occurs, and use dynamic programming to select the branch that yielded the best performance. Branching does significantly increase the complexity, though. Also, it is no longer a greedy heuristic once branching comes into play. As the thesis focuses on a non-euclidean distance model, we will not be applying the "opposite-hemisphere" rule in the heuristic.

3.2.2 Hiearchical clustering

Hierarchical clustering (HC) is a technique commonly used in machine learning. It is useful due to the fact that it does not need a specified number of clusters, as opposed to the more commonly used k-means clustering.

HC can implemented in a top-down approach, where the initial cluster is all elements, and is continuously split into smaller clusters in a tree-like fashion until we are left with a single element in each cluster.
Or it can be implemented bottom-up, which is the complete opposite. We start with a single element cluster and then continuously merge clusters until we are left with a single cluster.

Both methods create a tree of clusters that applies itself to data of hierarchical properties. This is useful when categorizing entities that can be placed into several subcategories, such as animals (split/merge for each trait) or similar data elements. Although there are usually no categories for our POI, it is still possible to apply HC to our POI data.

In order to utilize HC for our problem, we will be starting with clusters containing a single POI and the Depot. Then, we merge clusters that have the lowest cost function (the closest ones) as long as the size of the merged cluster does not exceed our given limit value for size of each cluster. This will usually not yield a complete tree as usually the limit value is lower than the total number of POI.

Instead, we are left with several subtrees where each root is a cluster of size at most the limit. The idea is to then compute a TSP result on each root, to obtain the tours for each cluster and the total distance metric.

3.3 2-opt

In order to compute the optimal tour for a given cluster we need to solve the Traveling Salesperson Problem. This is notoriously difficult to solve exactly and thus we need to look for efficient solvers that yield an acceptable performance when considering run-time and optimal results.

2-opt is a simple and elegant technique for solving TSP. The idea for 2-opt is to create an initial permutation for the tour and then repeatedly reverse the order of all sub-tours, until the reversing a sub-tour no longer improves the total tour-lenght. A sub-tour is for any tour with k elements $t : p_1, p_2, ..., p_k$ a subset where the order is the same. So, for sub-tour st_i, any two consecutive elements in the order p_q, p_r, $r = q + 1$. In our case we exclude the first and final elements from all sub-tours as we always start and end with the Depot. When reversing an arbitrary sub-tour of length j, $st_i : p_u, p_u + 1, ..., p_u + j - 1, p_u + j$ the reversal of st_i is $p_u + j, p_u + j - 1, ...p_u + 1, p_u$. In each iteration of 2-opt we try to reverse all sub-tours and keep the reversal that yields the lowest cost

function for the tour. We keep iterating until no reversal of a sub-tour yields a lower cost function.

Despite it's simplicity, 2-opt achieves great results on real-world problems, despite only finding a local optimum. Contrasting this simple technique, the runtime complexity of 2-opt finding the local optimum is a subject that has been studied extensively. The reason for this is the nature of polynomial local search (PLS) problems. PLS is the complexity class that measures the difficulty of finding a locally optimal solution to an optimization problem. We will not be discussing PLS further in this thesis, 2-opt will be used as the final tool to measure the performance of an obtained cluster system.

3.4 Optimal solution

In order to reliably obtain an optimal solution, we need to exhaustively try all permutations of tours and select the one with the lowest distance total. A set of n elements has $n!$ possible permutations. This is made even more complicated by adding the choice of returning to Depot and start a new tour. This means that on top of the $n!$ permutations of visiting all POI's, we also have $2^n - n/k$ possibilities of when to return to Depot (n/k is the minimum times we need to return to Depot as the minimum number of tours is n/k) giving a runtime complexity of $O(n! * 2^n)$. This is extremely exponential and makes it near impossible to obtain an optimal solution once $n > 10$. There is therefore little use in comparing our heuristic to an exact solution.

It is possible to achieve an exact solution for sets much larger than 10 however. For normal TSP, an exact solution can be obtained by using the Held-Karp algorithm which has a runtime complexity of $O(n^2 * 2^n)$. This is still an exponential measure, and thus solving TSP for sets larger than 30, using Held-Karp becomes exceedingly impossible. However, there have been presented solutions for sets of several thousand data points. There is a continuous race for obtaining an exact solution for the largest set of datapoints. A famous result for solving TSP appeared in Dantzig, Fulkerson, and Johnson's classic paper from 1954 where they present a solution for a 49-city problem consisting of one city from each of the 48 main-land states in the U.S.A. and Washington D.C. This paper started the steady progress of solving larger and larger sets for TSP. Today, researchers are working on sets the size upwards of 1 million!

3.4.1 The Cutting-Plane method

The paper presented by Dantzig et al. disserts a technique for solving large TSP's that is still in use today, a linear programming method known as the Cutting-Plane method. To describe the Cutting-Plane method, we must first describe TSP as a linear programming.

GJENGITT FRA KILDE An instance of TSP with n cities can be described as a complete graph with $n(n-1)/2$ weighted edges. These edges can be specified into two vectors. The first vector c contains the weight of each edge. The second vector x assigns each edge the value 1 if it is in the solution, 0 otherwise. Not all possible assignments of x yields a valid solution as the assignment must denote a tour that visits each city once. We denote set of valid assignments of x as S. The problem can then be stated as the following linear programming problem:

$$minimize\ c^T x\ subject\ to\ x \in S \tag{3}$$

This is not a linear inequality however, and can not be solved using techniques such as the simplex method. A relaxation of (3) is needed, and can be obtained by swapping $x \in S$ for a system $Ax \leq b$ that is satisfied by any x. The optimal solution to this provides a lower bound on the optimal value of (3). However, this can help with more than just finding a lower bound. If the optimal solution x^* is in S, then we have solved the problem. If it is not in S then a linear inequality can be obtained that is satisfied by all the points in S, but not by x^*. Such an inequality is known as a cut or a cutting plane. This cut can then be added to the system $Ax \leq b$, further tightening the relaxation of (3). We continue this process until we find an optimal solution of the relaxation that is in S.

This technique has been used since the 1950's to solve TSP instances of exceedingly larger size, but unfortunately computational limits are still a hinder to overcome, and in theory The fact of optimally solving TSP instances of tens of thousands of cities is remarkable.

4 Solution

In order to obtain the input given to our heuristic, we must perform some pre-processing steps that were explained in section 1.2

4.1 Pre-processing

The following pseudo-code tries to illustrate the pre-processing algorithm in a short concise matter. This should not be considered a direct representation of the logic in the source code.

4.1.1 Induce directed graph

First we simply want to be working with a directed graph. In the case of undirected edges in the graph we convert them into two opposite directed edges. After this procedure we consider all edges in the graph as directed.

Algorithm 1 Construct graph as directed

```
1: procedure DIRECTED GRAPH(Graph G, node b)
2:     for each edge (u, v) in E(G) do
3:         if undirected(u, v) then
4:             G.add_edge(v, u)
5:         end if
6:     end for
7:     return G
8: end procedure
```

Runtime analysis:
We assume $undirected(u, v)$ is an $O(1)$ which gives us a run-time of $O(n)$ for n nodes.

4.1.2 Strongly Connected Component of Depot node

Next, we consider graphs that are not strongly connected, and induce the subgraph that is a SCC containing our Depot node. This is because we only want vertices that are strongly connected to the Depot vertex. To determine the SCC of b, we select the nodes which are reachable from b, and b is reachable from each respective node.

Algorithm 2 Remove vertices not strongly connected to node b

```
1: procedure SCC OF B(Directed graph G, node b)
2:     reachable ← DFS(G, b)   ▷ Run DFS to get all nodes reachable from b
3:     for each node u in V(G) do
4:         if u in reachable then
5:             connected ← DFS(G, u)
6:             if b in connected then
7:                 continue
8:             else
9:                 G ← G.remove(u)
10:            end if
11:        else
12:            G ← G.remove(u)
13:        end if
14:    end for
15:    return G
16: end procedure
```

Runtime analysis:
DFS is $O(n + m)$ and we potentially run DFS for each node, which yields a runtime of $O(n(n + m))$

4.1.3 Linkage matrix construction

The bulk of our pre-processing is to construct a linkage matrix for our clustering technique. We only consider nodes that are weighted, i.e. nodes that contain one or more POI's. This is also the computationally slowest part of the pre-processing as we need to make use of Dijkstra's algorithm on each weighted node to construct the matrix. It is possible to construct a linkage matrix that is sorted for each row. This is due to the nature of Dijsktra visiting nodes that are closest first. This will be helpful for our greedy heuristic. Our source code includes a non-sorting method as well, to maintain a structure of readability, i.e. the 4th element in any particular row always corresponds to the same POI.

Algorithm 3 Create linkage matrix for nodes that are weighted and b

```
1:  procedure LINKAGE MATRIX(Directed graph G, node b)
2:      M ← Matrix[V(G)][V(G)]
3:      dists ← djikstra(G, b)                    ▷ dists[u]: distance from b to u
4:      for each node u in V(G) do
5:          if w(u) > 0 then
6:              M[b][u] ← dists[u]
7:              udists ← djikstra(G, u)
8:              for each node v in V(G) do
9:                  if w(v) > 0 or v == b then
10:                     M[u][v] ← udists[v]
11:                 end if
12:             end for
13:         end if
14:     end for
15:     return M
16: end procedure
```

Runtime analysis:
The same principle as SCC applies, except we compute Dijkstra on each weighted node. Dijkstra's runtime complexity depends on the implementation, but optimally it will have a runtime of $O(nlogn)$. Which ultimately yields a runtime of $O(wnlogn)$ when constructing the linkage matrix.

4.1.4 Create a matrix row for each individual POI on each node

Finally, to simplify the clustering step, we seek to create individual rows for each POI. This means we need p duplicates for a row corresponding to a node weighted p. We don't need the duplicates in the graph, but they should be included in the linkage matrix.

Algorithm 4 Duplicate nodes equal to their weight, update M

```
1: procedure DUPLICATE NODES(Graph G, linkage matrix M, node b)
2:     for each node u in V(G) do
3:         if w(u) > 1 then
4:             for (i ← 1; i < w(u); i ← i + 1) do
5:                 M[ui] ← M[u]          ▷ Update linkage matrix with new node
6:                 for each row v in M do
7:                     v[ui] ← v[u]
8:                 end for
9:             end for
10:            w(u) ← 1
11:        end if
12:    end for
13:    return M
14: end procedure
```

Runtime analysis:
Although the concept seems rather simple, the current heuristic contains three nested for-loops. This is due to the fact that we must update the entire linkage matrix for each new row created. This procedure adds another $O(n^2W)$ to the total runtime for our pre-processing heuristic.

4.1.5 Runtime analysis of pre-processing

The challenge of the pre-processing is how to deal with the duplication of rows. The current heuristic will have a runtime of $O(n^2 log n + n^2 W)$. The total weight of the nodes play a role both in the pre-processing part as well as the later clustering procedure. It is currently unknown if there are methods to decrease the runtime of the pre-processing.

4.2 Greedy heuristic for EkRRTSP

ekRRTSP has a simple greedy heuristic: Simply visit the closest non-visited node at each step, and then return to Depot once k nodes are visited. The distance matrix is sorted by distance to easily find the closest vertex. The pseudocode below only returns both the total distance travelled to solve and the order of which the POI's are visited.

Algorithm 5 Greedy heuristic for EkRTSP

```
1: procedure GREEDY SOLVER(Sorted Linkage Matrix M, Size k, Start Node b)
2:     totalDist ← 0
3:     trips ← [b]                                  ▷ Order of nodes visited when solving
4:     visited[M[0]] ← all(False)
5:     visited[b] := True
6:     cnt ← 0
7:     while cnt < visited.size() do
8:         tripLength ← 0
9:         currentNode ← b
10:        while trip < k do
11:            if cnt >= visited.size() then
12:                break
13:            end if
14:            for (i ← 0; i < M[currentNode].size(); i ← i + 1) do
15:                if not visited[i] then
16:                    visited[i] ← True
17:                    cnt ← cnt + 1
18:                    next ← i
19:                    break
20:                end if
21:            end for
22:            trips ← trips.push(next)
23:            totalDist ← totalDist + M[currentNode][next]
24:            currentNode ← next
25:            tripLength ← tripLength + 1
26:        end while
27:        trips ← trips.push(b)
28:    end while
29:    trips ← trips.push(b)
30:    totalDist ← totalDistM[currentNode][s]
31:    return (totalDist, trips)
32: end procedure
```

4.2.1 Runtime analysis of greedy heuristic

While there are two nested while loops as well as a for loop the runtime is actually $O(nlogn)$. As each iteration of the inner for loop marks exactly one POI as visited. And the linkage matrix is sorted, so we need only find the closest node that is not visited. This is a very fast way to solve a TSP-like problem, but sadly not an effective one.

4.3 Greedy clustering heuristic for RTSP

The greedy heuristic for obtaining a cluster set for RTSP has a simple implementation that is easy to explain. Given a sorted linkage matrix M we find the next element to cluster on by the last non-assigned element p in the row in M corresponding to the depot. From there we select the k first elements in the row corresponding to p that have not been assigned to a cluster. Once the cluster is filled we add the cluster to the list of clusters. Once we have gone through all elements in the row corresponding to depot, we have assigned all elements to exactly one cluster and can return the list of clusters. The pseudocode describes in detail the logic of the implementation.

Algorithm 6 Greedy clustering for RTSP

```
1: procedure GREEDY CLUSTERING(Sorted Linkage Matrix M, Start Node d, limit k)
2:     clusterArray ← Array < cluster >    ▷ Create empty array for clusters
3:     clustered ← HashMap < string, bool > (M, False)    ▷ track which elements are clustered
4:     for p in M[d].reverse() do
5:         if !clustered[p] then
6:             elemList ← [d, e]
7:             clustered[p] ← True
8:             size ← 1
9:             for qinM[p] do
10:                if size >= k then
11:                    break
12:                end if
13:                if !clustered[q] then
14:                    elemList ← elemList + q
15:                    clustered[q] ← True
16:                    size ← size + 1
17:                end if
18:            end for
19:            cluster ← cluster(elemList)
20:            clusterArray ← clusterArray + cluster
21:        end if
22:    end for
23:    return clusterArray
24: end procedure
```

Runtime analysis:
Although there are two nested for-loops in the implementation, the outer for-loop skips all elements that are processed by the inner for-loop. Thus the runtime is simply $O(n)$ as 1 element is assigned to a cluster for each iteration of either for loop.

4.4 Agglomerative clustering heuristic for RBTSP

The heuristic for solving RBTSP using hierarchical clustering is split into three parts, where the two first parts can be considered pre-processing. The pseudocode snippets describe a slightly modified agglomerative clustering heuristic (ACH). We continuously select the cluster-pair that have the lowest cost function value and merge them. The cost-function is most commonly Depotd on the distance-metric between clusters, we will be utilizing three of these in our experimentation:

- single linkage: The smallest distance between any two elements from each cluster. For our clusters we need to exclude the Depot vertex to be selected, as it is present in both clusters. This means that the absolute smallest distance between elements would be 0 (Depot element in both clusters). Single linkage tends to produce "loose" clusters.
- complete linkage: The largest distance between any two elements from each cluster. Complete linkage tends to create more compact clusters, and is the most common of the three.
- average linkage: The average distance between all pair-wise elements from each cluster.

Overall, complete linkage is the most likely to produce better results, but we include the other linkage methods for comparison's sake.

A regular ACH will keep merging clusters until a single cluster is obtained. However, in our heuristic we do not merge clusters if their combined size exceeds the imposed limit on cluster size. Thus, we are left with several trees, instead of a single one.

4.4.1 Initializing the clusterset

Our starting input is the linkage matrix obtained in the pre-processing section. We create a unique cluster for each row entry in the linkage matrix, which contains the respective vertex and the Depot vertex from the input graph. The Depot vertex is part of the linkage matrix, but does not get its own cluster.

Algorithm 7 Construct initial clusters

```
1: procedure INITIAL CLUSTERS(Linkage Matrix M, Start Node b)
2:     clusterArray ← Array < cluster > ▷ Create initial clusters containing a single node and b
3:     for e in M do
4:         if e ≠ b then
5:             elemList ← [b, e]
6:             cluster ← cluster(elemList)
7:             clusterArray ← clusterArray.push(cluster)
8:         end if
9:     end for
10:    return clusterArray
11: end procedure
```

Runtime analysis:
This procedure is completed using a single for loop over the rows of M, and so it is an $O(n)$ operation.

4.4.2 Initialize priority queue

Agglomerative clustering merges the two clusters with the lowest cost function. A fast way of finding these two clusters is to use a priority queue, or a heapqueue. We store the two clusters in the heap and sort them by the computed cost function belonging to the pair. We do not implement our own priority queue framework, instead using the library function included with Python 3.

Algorithm 8 Construct Priority Queue for pair-wise cluster distances

```
1: procedure PRIORITY QUEUE(list clusterArray, size k)
2:     PQ ← priorityQueue()
3:     for c1 in clusterArray do
4:         for c2 in clusterArray do
5:             if c1.size() + c2.size() ≤ k then
6:                 res ← CostFunc(c1, c2)
7:                 PQ ← PQ.push(res, (c1, c2))
8:             end if
9:         end for
10:    end for
11:    return PQ
12: end procedure
```

Runtime analysis:
Adding elements to a priority queue takes O(log n). As we need the pair-wise distances of all clusters we need to add elements n^2 times, yielding a run-time of $O(n^2 logn)$

4.4.3 Agglomerative clustering

Finally, when the priority queue is ready, we can start building the trees. The clusters have a child and parent parameter that keeps track of the tree as it is being built. The parameters are updated in the *merge* function, where two clusters are merged into one. To make sure that each cluster is merged at most once, we include a lookup array that keeps track of which clusters have been merged. Once two clusters are merged, we need to recalculate the cost functions of the new cluster and all other non-merged clusters, and add them to the priority queue.

Algorithm 9 Pseudocode for Agglomerative clustering

```
1: procedure AGGLOMERATE(array clusterArray, priority queue PQ, size k)
2:     merged ← [false for c in clusterArray]
3:     while !PQ.isEmpty() do
4:         (dist, (c1, c2)) ← PQ.pop()
5:         if !merged[c1] and !merged[c2] then
6:             newC ← merge(c1, c2)
7:             merged[newC] ← false
8:             merged[c1] ← true
9:             merged[c2] ← true
10:            for c in clusterArray do
11:                if !merged[c] then
12:                    res ← costFunc(c, newC)
13:                    PQ ← PQ.push(res, (c, newC)
14:                end if
15:            end for
16:            clusterArray ← clusterArray.push(newC)
17:        end if
18:    end while
19:    return clusterArray
20: end procedure
```

runtime analysis:
Priority queue is the aspect of importance when analyzing the runtime of the heuristic. Selecting and removing the smallest element of a priority queue is an $O(logn)$ operation, as well as inserting new elements into the PQ. The initial size of PQ is n^2 so each of these operations become $O(log^2 n)$. For each loop iteration, the number of clusters that remain unmerged is reduced by 1 (two clusters are merged, a new one is added to the pool). Also, the PQ does not shrink in size until the clusters are sufficiently large enough. Further analysis is required, but we know that the overall runtime is $O(n^2 logn)$, for an nxn matrix which is slightly faster than not using a priority queue where the runtime would be $O(n^3)$.

4.4.4 Merge subroutine

To show that the set of clusters contain a tree-like structure we include a description of the *merge* sub-routine. Here we create the new cluster from two child-clusters and connect them with the child/parent parameters.

Algorithm 10 sub-routine for merging two clusters into one

```
1: procedure MERGE(cluster c1, cluster c2)
2:     elemList ← c1.elems() ∪ c2.elems()
3:     cluster newC ← cluster(elemList)
4:     newC ← newC.setChildren((c1, c2))
5:     c1 ← c1.setParent(newC)
6:     c2 ← c2.setParent(newC)
7:     merged(newC) ← false
8:     return newC
9: end procedure
```

runtime analysis:
The subroutine is a simple operation of connectors and object initiating, and has a runtime complexity of $O(1)$

4.5 Heuristic for solving TSP within each cluster, TBD

Finally we could use a traditional TSP solving technique for each group. Although the TSP solving step is exponentially hard to compute, so long as k is of a relative small size ($k < 30$) it should not be too resource intensive. As k is a constant it does not grow as the problem size grows so it is not considered an exponential solution for that matter.

5 Results

5.1 Greedy heuristic for EkRTSP

First we implemented a greedy heuristic with a simple logic. Once the preprocessing was complete, the solving was simple to implement. One obvious flaw with the heuristic that became apparent is that it does not take advantage of the split nodes. There is a high likelihood of having to visit the same split cluster several times to collect all the POI that are located there. This is something a more elegant solution may take better advantage of.

```
Input:
maximum trip: 2
B: B : 0.00, 0 : 1.00, 1 : 2.00, 2 : 3.00, 3 : 3.00, 4 : 2.00, 5 : 1.00,
0: B : 1.00, 0 : 0.00, 1 : 1.00, 2 : 2.00, 3 : 3.00, 4 : 2.00, 5 : 1.00,
1: B : 2.00, 0 : 1.00, 1 : 0.00, 2 : 1.00, 3 : 2.00, 4 : 3.00, 5 : 2.00,
2: B : 3.00, 0 : 2.00, 1 : 1.00, 2 : 0.00, 3 : 1.00, 4 : 2.00, 5 : 3.00,
3: B : 3.00, 0 : 3.00, 1 : 2.00, 2 : 1.00, 3 : 0.00, 4 : 1.00, 5 : 2.00,
4: B : 2.00, 0 : 2.00, 1 : 3.00, 2 : 2.00, 3 : 1.00, 4 : 0.00, 5 : 1.00,
5: B : 1.00, 0 : 1.00, 1 : 2.00, 2 : 3.00, 3 : 2.00, 4 : 1.00, 5 : 0.00,

Output:
total Distance: 15.0
trip 0 | length: 4.0 ['B', '0', '1', 'B']
trip 1 | length: 4.0 ['B', '5', '4', 'B']
trip 2 | length: 7.0 ['B', '2', '3', 'B']

Process finished with exit code 0
```

The image above is code input and output of the heuristic. 6 nodes excluding Depot. The input is the distance between nodes as well as the maximum trip possible.

The input table was derived from a graph that was originally an undirected cycle of 6 nodes including Depot, and a couple disconnected nodes that were removed in the pre-processing. The POI were generated at random, and it was possible for more POI to be assigned to the same node, leading to the node being split. However, in this example that was not the case, as there was a single POI for each node in the CC.

Output gives us the total distance covered and the trips performed, and the length of each trip. The total distance will be used as a benchmark to compare the different heuristics performance.

5.2 Greedy heuristic for RTSP

The greedy clustering heuristic performs surprisingly well on most graphs, given its simplicity.