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 base. 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 (base), which will be our starting point.
2: The POI and base are spread across a plane and all POI should be both pair-wise reachable and reachable from base.
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 base 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 base. One vertex must be marked as the base. 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 base 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 base 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 base 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 a structurally identical graph no matter what the original graph input was.

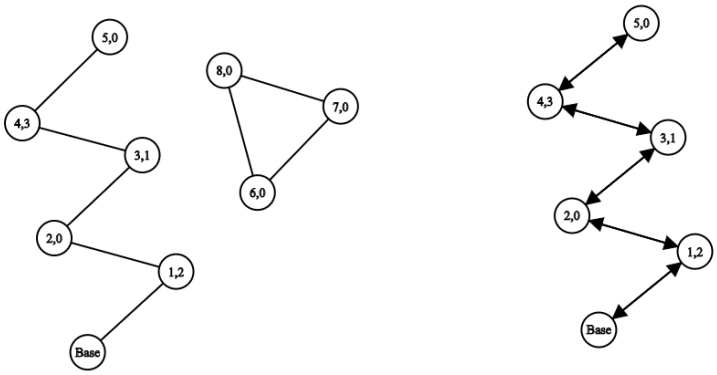
We expect the graph to be derived from a roadmap or a similar structure, and in order to accomodate a varying format of the graph induced from the map data, these steps are deemed as necessary. The data may be derived from

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

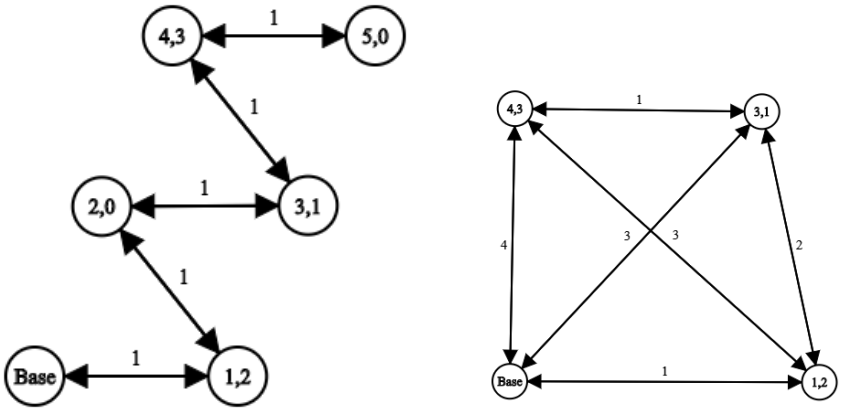
[3] A graph contains shortcuts if given 3 vertices $u, v, andw$ and a function $d(a, b)$ which indicates the distance between a and b: $d(u, v) + d(v, w) < d(u, w)$.

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.

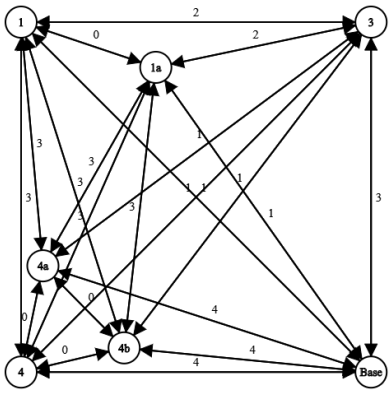
The order which these steps are performed in is somewhat arbitrary, but can affect the performance of the pre-processing heuristic. One might consider calculating the complete graph before splitting multi-weight vertices, as the distance data is identical for all duplicated vertices. The following example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



Remove nodes not reachable from base, 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.

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

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?

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

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 baseline 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 Hiearchical clustering

Hiearchical clustering (HC) is a technique that continuously splits clusters into smaller subsets. This is useful when categorizing entities that can be placed into several subcategories, such as animals. HC creates a tree of clusters that split the data into hierarchical categories. Although there are usually no categories for our POI, it is still possible to apply HC to our POI data.

Rather than attempting to create a tree of subcategories we only focus on a single level of the tree. After applying a complete HC to our data, we can select the level of the tree that has no clusters larger than the limit. Thus we are given a divide of our POI that can each be completed in a single trip. If we make this divide be as efficient as possible when completing each trip, we have an excellent start for our heuristic.

3.3 Dynamic Programming - rough draft

Once we have our clusters there still remains to calculate the shortest distance to visit each POI in the cluster. There is no way around accepting that this is the classic Traveling Salesman Problem mentioned several times in this thesis. The only difference is the base which we must start and stop out trip at. TSP is a notoriously difficult to solve, but there exists some techniques that outperform a brute force computation.

But let's first discuss how to consider our base node such that we guarantee that it is the first and last node in our trip. A simple way to do this is to duplicate the base and add a dummy node to each trip that have 0 weight on edges connected to its base and infinite weight to all other nodes. This will force the algorithm to start and end on the dummy nodes and thus, the base node. This may be a risky tactic as many TSP solvers do not handle the occurrence of 0 and infinite edges well.

4 Solution

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

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 Pre-processing

```
Data: graph G, base vertex B
Result: A distance matrix
initialization;
Matrix[V(G)][V(G)]
—
/* Make all edges directed                                    */
for each edge e in E(G) do
    if e is undirected then
        e1 := e
        make e, e1 directed in opposite directions
    end
end
—
/* Remove all vertices not strongly connected to B            */
reachable := DFS(G, B)
for each node u in V(G) do
    if u in reachable then
        connected := DFS(G, u)
        if B in connected then
            continue
        end
        else
            remove u from G
        end
    end
    else
        remove u from G
    end
end
—
/* Calculate distance between all nodes in G that have
   weight at least 1, using a path-finding algorithm e.g.
   Dijkstra.                                                  */
for each node u in V(G) do
    if weight(u) > 0 then
        Matrix[B][u] := shortest distance from B to u
        Matrix[u][B] := shortest distance from u to B
        for each node v in G do
            if weight() > 0 then
                Matrix[u][v] = shortest distance from u to v
            end
        end
    end
end
—
/* Duplicate plural weighted nodes                            */
for each node u in G do
    if weight(u) > 1 then
        for i=1; i<weight(u); i++ do
            new node u-i := u
            weight(u-i) := 1 dist(u-i, u) := 0
            dist(i, all new nodes) := 0
            add i to Matrix
        end
        weight(u) := 1
```

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 base once k nodes are visited. The distance matrix is sorted by distance to easily find the closest vertex. The pseudocode below only returns the total distance travelled to solve, while our source code also returns the subsets for each trip and their respective distance to travel.

```
Data: A sorted distance matrix M, a start key B, limit k
Result: totalDist, the distance traveled
initialization;
totalDist := 0
visited[M[0]] := all(False)
visited[B] := True
while any(not visited) do
    trip := 0
    currentNode := B
    while trip < k do
        if all(visited) then
            break
        end
        for i := 0; i < |M[currentNode]|; i++ do
            if not visited[i] then
                next := i
                visited[i] := True
                break
            end
        end
        totalDist += M[currentNode][next]
        currentNode := next
        trip++
    end
    totalDist += M[currentNode][s]
end
return totalDist
```

4.3 Clustering heuristic for RBTSP

RBTSP may require a more sophisticated solution in order to utilize the variable trip length. Thus, we will refrain from presenting a greedy heuristic for this problem. Instead we will utilize a technique that has a broad use in the field of algorithms.

This heuristic we call a clustering solution: First use the distance matrix in a clustering algorithm to get a grouping where the size of each group is at most k. Each cluster is grouped such that the average distance between members of

the group, including the base, is minimized.

In the clustering step the grouping constraint takes the double of the distance to base, as it must be visited twice in a trip. This is a key element that may allow the clustering solution to beat the traditional greedy solution, as it does not take the distance to base into account. However, it is not yet determined if it makes a difference in the computation.

The following heuristic describes a agglomerative hierarchical clustering technique. We start by copying the start node and grouping each other node with the start node, and then merging them as long as the average distance decreases. The distance from start node needs to be doubled to emphasize the fact that we need to visit start node twice. Note that there are several subroutines used here that has not yet been described:

- c.avgDist(): Each cluster has an average distance between pairwise elements.
- avgDist(c,d): The subroutine can also give the avgDist for pairwise elements in two clusters.
- maxDist(c,d): Yields the distance between the two elements from each cluster that are furthest away
- merge(c,d): Merges two clusters into one, start node is not duplicated. Ideally the subroutine deletes the latter cluster as it is merged.

```
Data: A distance matrix M, a start index s, limit k
Result: A set of C of clusters dividing all members of M into cluster
initialization;
C := empty set of clusters
optimized[M[0]] := all(False)
for each node u in M != s do
 | cluster c := [s,u]
end
while any !optimized do
 | for each cluster c where !optimized[c] do
 | | minMaxDist := Inf
 | | selfDist := c.avgDist()
 | | mergeCluster := null
 | | for each other cluster d do
 | | | if d.size() + c.size() - 2 <= k && !optimized[d]
 | | | | then
 | | | | dist := maxDist(c,d)
 | | | | avgdist := d.avgDist()
 | | | | avgcomp := avgDist(c,d)
 | | | | if avgcomp < selfDist + avgdist && dist <
 | | | | | minMaxDist then
 | | | | | minMaxDist := dist
 | | | | | mergeCluster := d
 | | | | end
 | | | end
 | | end
 | | if mergeCluster != null then
 | | | c := merge(c,d)
 | | end
 | | else
 | | | optimized[c] := True
 | | | C.insert(c)
 | | end
 | end
end
return C
```

4.4 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.

Hopefully the clustering solution yields a smaller approximation factor than the proposed greedy solution while staying polynomial in time.

5 Results

5.1 Greedy heuristic

First we implemented a greedy heuristic with a simple logic. Once the pre-processing 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 base. 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 base, 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.