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 electric scooters that have run out of charge.

In several major cities across the world, you can now rent an electric scooter on almost every 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. 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, or nearly run out of power, it needs to be collected and charged. Your job is to collect these scooters and take them to a charging station.

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 solution 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".

A successful hunt for this bird-catcher

Both solutions get the job done, but in either case you may be faced with one of two problems:
1: Your company is competing with other scooter-renting companies in several cities and in order to get the edge, your supervisor wants you to improve the cost and time it takes to charge our scooters, such that they can be more quickly available for use. Super-chargers and larger pick-up vehicles only get you so far, though, and at a price.
2: As a self-employed ”bird-catcher” you will be competing with other free-lancers on collecting these valuable ”scooter-bounties”. It’s first come first served, so how do you get there first?

In both cases, 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, 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 distance between the cities represent the value each edge is assigned.

Our problem is similar, but differs in some areas; The cities are replaced by electric scooters, and the goal is not to "visit" every scooter in one go, but a selected few, before returning to a base. 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 our heuristic we can be given any graph with a strongly connected component is at least 2. One vertex must be marked as the base. Depending on the type of graph, some pre-processing must be done. The base vertex is not considered in these steps:

- If the graph is undirected, we duplicate all edges and make them directed both ways, such that the graph becomes directed.
- If the graph is not connected, we select the strongly connected component where the base is located.[1]
- If the graph is not weighted on the edges, we assign each edge to have weight 1
- If the graph is not complete[2], we calculate the shortest paths between all vertices and 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.
- If the graph is weighted on the vertices, we do the following:
    - 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 weighted, but not complete, when computing the complete graph, we only calculate the distance of nodes weighted no less than 1.

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:

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

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

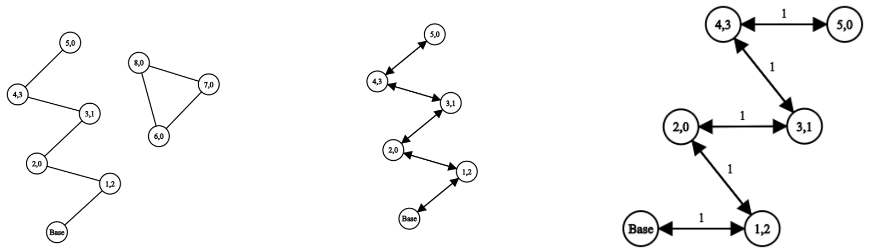
[3]A graph contains shortcuts if given 3 vertices $u, v, andw : d(u, v) + d(v, w) < d(u, w)$.

- Step 1 is needed so that we do not need to between directed and undirected graphs
- Step 2 ensures that our graph is strongly connected.
- Step 3 gives us a much simpler starting point as the shape 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 nodes.
- Step 4 is just an extra insurance so that we never need to think about finding the shortest path in a path.
- Step 5 ensures that each node needs only be visited once. It removes the unpleasant problem of arriving at a vertex where the weight on the vertex is more than we can add to our current trip, and the presence of vertices where the weight of the vertex is higher than our trip limit
- Step 6 essentially "removes" the zero-weighted nodes by not including them in our complete graph.

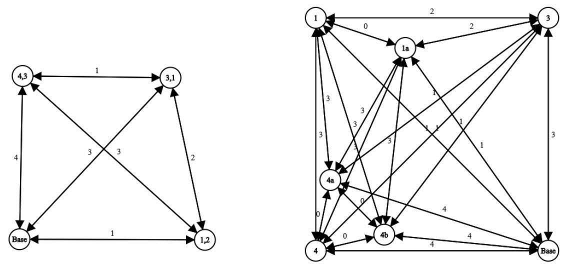
After completing these pre-processing steps we end up with a structurally identical graph no matter what the original graph input was.

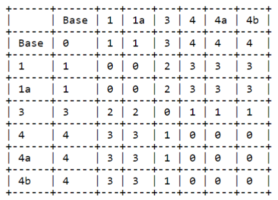
We expect the graph to be derived from a roadmap or something similar, and in order to accomodate a varying format of the graph induced from the map data, these steps are deemed as necessary. The data may be derived from a graph representing a road-network, which can be both undirected and directed (i.e. the existence of one-way streets), or it may only consist of vertices indicating the locations of our POI's. 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 connected to base, induce directed graph, assign edgeweights





```
+------+------+---+----+---+---+----+----+
|      | 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  |
+------+------+---+----+---+---+----+----+
```

Calculate complete graph, duplicate heavier nodes, induce matrix for readability

2 Problem formulation

2.1 Limit sum of length of trips, all trips are length k

Given a complete weighted graph G, a start node s, a trip-length k and bound X; Is there 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?

$$\begin{aligned}
&Input: G,\ w(E(G)), start\ s,\ trip-length\ k,\ constraint\ X,\\
&and\ \forall\ u,\ v,\ w\ \in V(G),\ w(u,w)\ \leq w(u,v)\ + w(v,w)\\
&\\
&Problem: \exists\ set\ of\ trips\ T: t_1, t_2, ..., t_n\ where\ a\ trip\ t_i \subseteq V(G)\\
&with\ an\ ordering\ of\ nodes\ v_1, v_2, ..., v_m\ s.t. the\ following\ holds: \qquad (1)\\
&1.\ \forall\ v\ \in V(G),\ v \in t_i \wedge v \notin T \setminus t_i\\
&2.\ \forall\ t_i\ \in T\ v_1 = v_m = s\\
&3.\ m\ = k+2,\ |t_n| <= k + |v_i = s|\\
&4.\ \textstyle\sum_{i=1}^{n} \sum_{p=1,q=2}^{m}\ w(E(v_p, v_q))v_p, v_q \in t_i \leq X
\end{aligned}$$

This problem, we will call the k-Rounds Traveling Salesman Problem or kRTSP. 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 $k = |V(G)|$. 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 solution that does not prohibit smaller trips and see if there is a possibility of getting a smaller solution:

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

Given a complete weighted graph G, a start node 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 all trips is at most X?

$$\begin{aligned}
&Input : G,\ w(E(G)), start\ s,\ trip - length\ k,\ constraint\ X,\\
&and\ \forall\ u,\ v,\ w\ \in V(G),\ w(u,w)\ \leq w(u,v)\ + w(v,w)\\
&\\
&Problem : \exists\ set\ of\ trips\ T : t_1, t_2, ..., t_n\ where\ a\ trip\ t_i \subseteq V(G)\\
&with\ an\ ordering\ of\ nodes\ v_1, v_2, ..., v_m\ s.t.\ the\ following\ holds :\\
&1.\ \forall\ v\ \in V(G),\ v \in t_i \wedge v \notin T \setminus t_i\\
&2.\ \forall\ t_i\ \in T\ v_1 = v_m = s\\
&3.\ m\ <= k + 2\\
&4.\ \textstyle\sum_{i=1}^{n} \sum_{p=1,q=2}^{m}\ w(E(v_p, v_q)) v_p, v_q \in t_i \leq X
\end{aligned} \tag{2}$$

This problem we call the At-most-k-Rounds Traveling Salesman Problem, or AmkRTSP, 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. Important to note, is the constraint of distance between nodes. By not having any "shortcuts", it is not yet clear if AmkRTSP will yield a better solution than kRTSP.

3 Solution

The problems can be solved using various techniques, though it is uncertain if they yield a minimum solution. First, in order to obtain a complete graph we must obtain a distance matrix between all members of G using a path-finding algorithm. From this matrix we can build a complete graph.

3.1 Pre-processing

```
Data: Connected weighted graph G
Result: A distance matrix
initialization;
Matrix[V(G)][V(G)]
for each node u in G do
    /* Calculate distance to all other nodes in G using a
       path-finding algorithm e.g.  Dijkstra                */
    for each node v in G do
        Matrix[u][v] = shortest distance to v from u
    end
end
/* Do additional pre-processing if nodes are weighted      */
if V(G) is weighted then
    for each node u in G do
        if weight(u) = 0 then
            remove u from Matrix
        end
        if weight(u) > 1 then
            for i=1; i<weight(u); i++ do
                new node i := u
                dist(i, u) := 0
                dist(i, all new nodes) := 0
                add i to Matrix
            end
        end
    end
end
return Matrix
```

3.2 Heuristic for greedy solution

KRBTSP has a simple greedy solution: Simply visit the closest non-visited node at each step, and then return to base once k nodes are visited. This is an approximation algorithm, in which the approximation factor is currently unknown.

```
**Data:** A distance matrix M, a start index s, limit k
**Result:** totalDist, the distance traveled
initialization;
totalDist := 0
visited[M[0]] := all(False)
visited[s] := True
while any !visited do
    trip := 0
    currentNode := s
    while trip <= k do
        if all(visited) then
            break
        end
        next := closest unvisited node from currentNode
        totalDist := totalDist + M[currentNode][next]
        visited[next] := True
        currentNode := next
        trip++
    end
    totalDist := totalDist + M[currentNode][s]
end
return totalDist
```

Note that finding the closest neighbour for each step can be handled using a priority queue of some sort such that the closest neighbour is always easily obtainable. A naive solution is described here which makes finding the closest neighbor take linear time rather than logarithmic.

3.3 Heuristic for clustering solution

RBTSP may require a more sophisticated solution in order to utilize the variable trip length. This solution 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. This step can be approximated in polynomial time, but is NP-hard to optimize.

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
```

3.4 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.