Heuristics for Master Thesis

Håvard Notland

February 2020

1 Introduction

We assume our graph is a complete graph. If it is not a complete graph, we induce a complete graph by computing the distances between all nodes. We assume a graph is located on a plane, such that given 3 nodes u, v, and w: $d(u,w) \leq d(u,v) + d(v,w)$. This implies that there are no shortcuts to be taken in the complete graph, and the shortest path is always the direct one.

The original graph may be weighted both on the nodes and on the edges. The nodeWeights indicate the number of POI's located in the node, while the edgeWeights indicate the distance between the given nodes. We induce a non-node-weighted graph from the complete graph by deleting all nodes with 0 weight and for all nodes with weight $n > 1$ we create a set of n nodes with edgeWeights set to 0 between them, and connect them to all other nodes. Thus any connected graph can be induced to a complete non-node-weighted graph.

1.1 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 at most k nodes (excluding s) and the total length all trips is at most X?

$$
\begin{aligned}
&\textit{Input} : G,\ w(E(G)), \textit{start } s,\ \textit{trip} - \textit{length } k,\ \textit{constraint } X,\\
&\textit{and} \ \forall\ u,\ v,\ w\ \in V(G),\ w(u,w)\ \leq w(u,v)\ + w(v,w)
\end{aligned}
$$

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

This problem, we will call the k-Round-based Traveling Salesman Problem or kRbTSP. 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:

1.2 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 Round-based Traveling Salesman Problem, or RbTSP, 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 RBTSP will yield a better solution than KRBTSP.

2 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.

2.1 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
```

2.2 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.

2.3 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
```

2.4 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.