CSC4120
Design and Analysis Algorithms

The Chinese University of Hong Kong, Shenzhen

Solving Approximation Algorithm of Party Together Problem (PTP)

Group

Author:	*Student Number:*
Alfonsus Rodriques Rendy	121040014
Nicholas Oh	121040006
Annabel Theodora Christabel	121040004

May 19, 2024

1 Background

1.1 Party Together Problem (PTP)

The Party Together Problem (PTP) is a challenge strongly inspired by the real life situation of picking up friends from around the city to our own home party. In this problem, we are given three inputs: an undirected graph $G = (V, E)$ with each node indexed from 0 to $|V| - 1$ representing either a point of interest in the city or a friend's home, a subset $H \subset V$ containing the nodes of our friend's homes, and a representing the relative cost of driving vs walking. We define our home to be at node 0, and the cost of going from one node to itself is equal to 0 (for friends who live at a pickup location). There are several key assumptions in this problem:

- Our car's capacity is unlimited, and picking up one or multiple friends from one node does not incur any time (we 'do not need to stop' while doing so).
- Our friends will take the shortest path from their homes to their designated pick-up location, which may be their own home or another place we decide.
- Graph G is connected, fulfills the triangle inequality rule, and we may pass through the same nodes any number of times.

Our main objective in this problem is to identify i) the pickup nodes/locations for our friends, and ii) determine a routing schedule for us to visit using our car, which minimizes the total cost for us and our friends. We can define the total cost as follows: let $L = \{p_m\}_{m \in F} \subseteq U$ be the set of locations where we pick up our friends, and $U = u_0, u_1, \ldots, u_n$ be the tour using our car, with $u_0 = u_n = v_0$ since we start and end at our own home. Let d_{ij} be the length of the shortest path between v_i and v_j. Then, the total cost is:

$$TotalCost = \sum_{i=1}^{n} w_{i,i-1} + \sum_{i}^{F} d_{h_m p_m}$$

.

1.2 Pickup From Home Problem (PHP)

The Pickup from Home Problem (PHP) is a constrained version of the PTP problem: we cannot designate pickup points for friends and must instead pick them up at their own homes. The NP-hardness of PHP can be proven by reducing the Metric Travelling Salesman Problem (M-TSP, a TSP with complete graph and triangle inequality) to PHP. One key advantage here is that we do not need to optimize the set of pickup locations. We prove below that the cost of PHP is at most twice of the optimal solution.

2 Theoretical Analysis

2.1 NP-Hardness of PTP (Q5.1)

To prove NP-Hardness of PTP, we can do it by proving the relation:

$$M - TSP \leq_p PHP \leq_p PTP$$

We can reduce M-TSP to PHP as follows: Let the graph for M-TSP instance is $G_{mtsp} = (V, E)$ to be fully connected then set $H = V$ with $G_{php} = G_{mtsp}$ and solve the PHP of the same graph. The optimal solution of PHP must be the optimal solution for M-TSP as the edges are non-negative and the graph is fully connected making PHP solution to guarantee the optimal solution of M-TSP. For a graph that is not fully connected, we can construct the fully connected counterpart by making the graph to be fully connected and add a very large weight to edges which does not belong to the original graph. If M-TSP of such graph is impossible, then the PHP will return a very large optimal as it uses the non-existent edge(s).

Second is to prove that PTP is harder than PHP. We can prove that PHP is a special case of PTP. An instance of PTP will be enforced to return a solution similar to PHP if we impose a very large penalty of walking cost. Therefore, setting $\alpha = 0$ will makes the cost of walking to be infinite. Therefore, we can reduce an instance of PHP by setting $\alpha = 0$ and solve the corresponding PTP.

Thus, since PHP is NP-hard (as it is harder than M-TSP) and PHP is a special case of PTP, then PTP must be NP-hard.

2.2 Approximation Bound of PHP on PTP (Q5.2)

Assume $\alpha = 1$.

$$C_{php} \leq T_{ptpopt} + \sum_{j \in H \setminus P_{opt}} 2 \min_{i \in P_{opt}} SP[i, j] \tag{1}$$

$$= C_{ptpopt} + \sum_{j \in H \setminus P_{opt}} \min_{i \in P_{opt}} SP[i, j] \tag{2}$$

$$\leq 2C_{ptpopt} \tag{3}$$

$$\beta = \frac{C_{php}}{C_{ptpopt}} \leq 2 \tag{4}$$

Inequality (1) holds because the optimal PHP must at least as optimal as taking tour of the optimal PTP and pick up the rest of the unpicked up friend one at a time using the shortest path to the node in the tour and go back to the node in the tour (P_{opt}). Equality (2) is because the cost of PTP is equal to the tour cost (driving cost) and the walking cost which is equivalent to the shortest path to the node in the tour. Inequality (3) is because the walking cost must be bounded by the PTP cost as driving cost is non-negative.

Suppose we have an instance as follows with each edge having weight 1 and each nodes represent a friend's house:

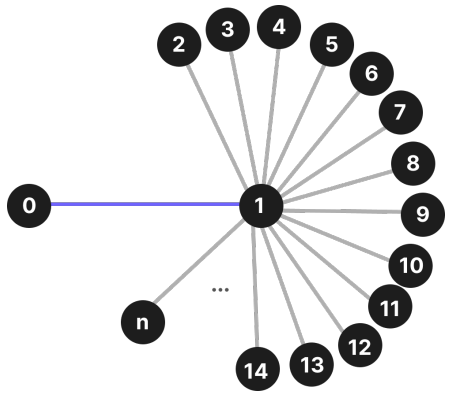

Figure 1: Instance with tight approximation bound

The optimal PTP is $T = [0, 1, 0]$ with total cost of $2+n$. But PHP will cost $2+2n$. Asymptotically we have:

$$\frac{C_{php}}{C_{ptpopt}} = \lim_{n\to\infty} \frac{2+2n}{2+n} = 2$$

3 PTP Approach

3.1 Input Generation

We decided to build a Python randomized generator for our inputs, extensively leveraging the *random* built-in library. Since the graph is continuous, we first build an MST to start the graph by adding randomly weighted (weights 10 to 40) edges between any node in the connected set (starts from root node 0 only) and the unconnected set (all nodes except root node 0). Finally, a recurring loop continually adds edges between two randomly sampled graph nodes.

The weight for each new edge is randomly chosen between the minimum bound of maximum edge weight within the entire graph, and the maximum bound of lowest value between minimum bound + weight [5, 20] and the shortest path distance between the two nodes. We bound the minimum with the max graph edge weight since this prevents the newly added edge from invalidating the triangle inequality for other node pairs. On the other hand, the maximum bound means that the new edge will be at most 5-20 weight heavier than the minimum bound, and definitely be lower than the Dijkstra shortest path distance (to satisfy the triangle inequality). If the interval between minimum and upper bounds are too tight (lower than two), we skip the step and repeat with another pair of nodes since we decide the original pair interval is too tight for randomization. We stop the edge addition loop when we have saturated the graph: something we determine by checking if the maximum graph edge weight is greater than 95% of a predetermined limit (here, we set it to 100). With these parameters in mind, our generated graph is highly connected, satisfies the triangle inequality, and all edge weights are highly randomized between 10 and 100.

3.2 Cheapest Insertion and Deletion

Cheapest insertion and deletion is a local search heuristic to obtain an approximation solution for TSP. However, we can generalize the heuristic to PTP problem by modifying the cost function of each possible tour.

Consider a possible initial tour T. The heuristic will start from T, and do a local search iteration where at each iteration it will take a decision from the following options

- Insert node i to the tour in an optimal position $T' = [T_1, \cdots, i, \cdots T_n]$ if the insertion reduce the cost.
- Delete node j from the tour $T' = [T_1, \cdots, T_{j-1}, T_{j+1}, \cdots, T_n]$ if such tour reduce the cost
- Stop the iteration when no possible insertion or deletion can be performed to reduce the cost

We formulate the cost of insertion and deletion as follows:

$$c_{insert}(i,k,j) = c + \underbrace{(d[i,k] + d[k,j] - d[i,j])}_{\text{driving cost update}} + \underbrace{\sum_{q \notin T} \min\{0, d[q.k] - \min_{p \in T} d[q,p]\}}_{\text{walking cost update}}$$

$$c_{deletion}(i,k,j) = c + \underbrace{(d[i,j] - d[i,k] - d[k,j])}_{\text{driving cost update}} + \underbrace{\sum_{q \notin T} \max\{0, \min_{p \in T \setminus \{k\}} \{d[q,p]\} - d[q.k]\}}_{\text{walking cost update}}$$

At each iteration we will iterate all possible nodes to be inserted/deleted. If a node is already in a tour, then we find the cost after deleting the node. If a node is not in a tour, then we find the location to which the node to be inserted can minimize the cost. Then, the final decision is taken at the end of iteration by finding the node to be inserted/deleted that minimizes the cost. If all possible operations do not decrease the cost then the 1-optimal solution is found.

The initial tour can be simply an empty tour (i.e. a tour to the source node itself) or any other possible tour. We explore two different initial tour, one with only source node and one which visit every friend's home to obtain the best solution among the two.

Algorithm 1 Cheapest Insertion/Deletion

Given $G = (V, E), H, \alpha$
Given initial tour T
$d \leftarrow SP[G]$ ▷ get all pairs shortest path in G
$cost_T \leftarrow \alpha \cdot \sum_{(u,v)\in E_T} d[u][v] + \sum_{i\in H\setminus\{V_T\}} d[i][0]$ ▷ Current cost is walking + driving cost
while True **do**
  for node $k \in V \setminus \{0\}$ **do**
    if $k \in V_T$ **then** ▷ Delete heuristic
      $cost_{T'}[k] \leftarrow c_{delete}(k-1, k, k+1)$
      $T'[k] = T.delete(k)$
    else ▷ Insert heuristic
      $cost_{T'}[k] \leftarrow \min_{i\in T\setminus\{0\}} c_{insert}(i, k, i+1)$
      $T'[k] = T.insert(k, i)$ ▷ insert k at position i that minimizes the cost
    end if
  end for
  $k' = \text{argmin}_k cost_{T'}[k]$
  if $T[k'] < cost_T$ **then** ▷ Update if cost is reduced
    $T \leftarrow T'[k']$
    $cost_T = T[k']$
  else ▷ Stop at convergence
    Break
  end if
end while

The complexity of the heuristics is $O(|V|^4|H|)$ where the while loop is guaranteed to terminate in $O(|V|)$ due to triangle inequality, where in each iteration we iterate every nodes in $O(|V|)$. During insertion we search for any position to insert in $O(|V|)$. In addition, at every insertion/deletion we also calculate for walking cost update which takes $O(|H||V|)$. With careful programming and memoization, we can reduce the search for the walking distance update to $O(1)$ leading to $O(|V|^3)$.

3.3 3-OPT Final Tour Optimization

The tour obtained of the cheapest insertion/deletion heuristic might not be optimal. We can further improve the tour cost using 3-OPT heuristic in order to find the best visiting sequence that minimize the driving cost.

3-OPT is done by checking every possible 3 edge combinations in the current tour at every iteration. At each iteration, the chosen three edges is removed leaving four unconnected segments of the tour. Then we connects every combination segment (excluding the beginning and end of the tour) resulting in $2^3 = 8$ combinations of tour connections. For each combination, we check

the update cost of changing the edge connection by:

$$cost_{update} = \sum_{(u,v)\in E'} d[u][v] - \sum_{(u,v)\in E} d[u][v]$$

where E' is the new edge connection added and E to be deleted connection from the current tour. We find the minimum update cost over all combination of three edges as c_{min}. $c_{min} < 0$ implies we can reduce the cost of the tour by updating the edge to the new 3-edge connection. If $c_{min} \geq 0$, then we already reach a local minima.

The complexity of 3-OPT is $O(|V|^3)$ for edge search at each iteration. Overall complexity is $O(n|V|^3)$ where n is the number of iterations until convergence.

Algorithm 2 Cheapest Insertion/Deletion

```
Given initial tour T, all-pairs distance d
minCost = 0
bestUpdate = T
while True do
    for E_1, E_2, E_3 ∈ E_T do                    ▷ Iterate combinations of three edges in E_T
        for T' = connect(E_T − {E_1, E_2, E_3}) do    ▷ Iterate any combinations of connection
            c ← cost_update(T')
            if c < minCost then
                bestUpdate ← T'
                minCost ← c
            end if
        end for
    end for
    if minCost < 0 then
        T ← T'
    else
        Break
    end if
end while
```

4 PHP Approach

4.1 PHP Reduction to TSP

We can reduce PHP to TSP by constructing a new graph $G' = (H, E')$ given original graph $G = (V, E)$ and list of houses $H \subset V$. We want G' to be fully connected so TSP is guaranteed to return a tour (as fully connected graph is guaranteed to have a hamiltonian path). To construct such graph G' we connect every houses to any other houses with weight approximated by the

shortest distance between each house, i.e.

$$(u, v) \in E' \qquad w_{uv} = SP[u][v] \qquad \forall\, u, v \in H$$

Given the solution of $TSP(G', 0)$ where G' is the fully connected graph of houses and 0 to be the original node (home), and let the solution to be T we can get the solution of PHP by connecting each nodes of the tour with the shortest path between each nodes. We have

$$T_{PHP} = [0, SP[0, T_1], T_1, \cdots, T_n, SP[T_n, 0], 0]$$

4.2 TSP with Dynamic Programming

We formulate the TSP with dynamic programming to solve PHP. Specifically, we are given a fully connected graph meaning a hamiltonian tour always exists. Suppose $DP[0 \to j, S]$ be the shorest path that visits all nodes $v \in S$ once and starts at i and ends at node j. The formulation of TSP is

$$TSP = \min_j \{DP[0 \to j, \{0, \cdots, n-1\}] + d[j][0]\}$$

where n is the number of nodes. For each sub-problem we have a substructure as follows:

$$DP[0 \to j, S] = \min_{i \in S, i \notin \{1, j\}} \{DP[1 \to i, S - \{j\} + d[i][j]\}$$

To model S in the code, we use a bit mask where 1 represents visited node (i.e. already removed from the set of nodes to be visited).

We have $2^n \cdot n$ subproblems as we have 2^n possible subsets of the nodes to be visited and n number of intermediates. Each subproblem has n choices. Therefore the complexity is $O(2^n \cdot n^2)$.

5 Results

We compare the solution and runtime of PHP and PTP implementation. To be specific, we compare the result of implementing cheapest insertion/deletion heuristics starting from (1) no pickup (i.e. the starting tour only consists of only home) (2) PHP pickup (i.e., starting tour consists of a tour to all of friend's home). The PHP pickup is approximated using 3-OPT heuristics from sequential tour (i.e. $T = [h_0, h_1, \cdots, h_{|F|-1}]$). We also compare the result of further approximation of the final tour using 3-OPT.

Testcase	PHP	C-ID (H)	C-ID (A)	+3-OPT(H)	+3-OPT(A)	+C-ID(R-PHP) (Multi-Iter)
1	4.0	3.33	3.33	**3.33**	**3.33**	**3.33**
2	**69.0**	76.8	72.9	**69.0**	72.9	**69.0**
3	142.0	145.0	142.0	145.0	142.0	**138.0**
4	**130.2**	157.8	138.6	132.3	138.6	**130.2**
5	271.0	264.0	273.0	264.0	273.0	**255.0**
6	**252.9**	298.8	252.0	298.8	**252.9**	**252.9**
7	886.0	900.0	886.0	900.0	886.0	**723.0**
8	**1743.0**	1937.1	1881.0	**1743.0**	1881.0	**1743.0**
9	7644.0	5961.0	6038.0	5961.0	5972.0	**5770.0**
10	18192.0	12686.0	11787.0	12686.0	**11775.0**	**11775.0**
Average	2933.41	2242.983	2147.473	2220.243	2139.673	**2085.943**

Table 1: Comparison of PTP approximation of Cheapest Insertion/Deletion (C-ID) with initial tour of home (C-ID (H)) and all friends home (C-ID (A)) and with or without 3-OPT. We also explored randomization by taking random initial tour based on PHP of random subset of houses, in addition with random tour optimization within greedy iteration (C-ID(R-PHP))

Our heuristic manage to achieve similar and even better performance compared to PHP. With performance on all of the testcases better or on par with PHP (improvement on 6 testcases).

In addition, we also find that 3-OPT have the potential to reduce the cost of the tour found in th cheapest insertion/deletion heuristic as the sequential decision might be still not optimized. Randomization also helps the greedy algorithm by exploring multiple local optimum. However, the improvement from insertion-deletion heuristic with 3-OPT is not significant while it requires significant additional time. Therefore, the heuristic is suitable for time-sensitive usage while with additional time we can improve it using randomization.

6 Conclusion

In this project, we have successfully demonstrated our strong understanding of PTP and PHP problems, NP-hardness, and the ability to design extensively heuristics-optimized solutions to computationally expensive problems. We have theoretically proven the NP-hardness of PTP, the approximation bound of PHP on PTP, reduced PHP to TSP, and formulated TSP with Dynamic Programming. Our PTP input generator is capable of generating highly random, connected, and triangle-inequality fulfilling graphs. We obtained promising results which demonstrate a highly-optimized solution to the provided project problems.