CSC4120 Spring 2024 - Project PTP Report

Yohandi 120040025
Andrew Nathanael 120040007

May 19, 2024

Project Party Together

We and our friends are going to have a party at our house this weekend to celebrate the end of the semester. we, being an excellent driver with a nice car, offer to pick up all our friends near or at their homes and drive them to our house. Can we come up with a transportation plan so that everyone gets to the party as efficiently as possible?

Formally, we are given an instance of the Party Together Problem (PTP) with inputs (G, H, α). $G = (V, E)$ is an undirected graph where V is the set of nodes indexed from 0 to $|V| - 1$ and each node represents a location in the city. The weight of each edge (u, v) is the length of the direct road connection between location u and v, which is non-negative. our house is at location 0. $H \subset V$ is the set of distinct locations that correspond to our friends' homes. If $F = \{0, 1, \ldots, |F| - 1\}$ is the set of friends, then each friend $m \in F$ has a house location $h_m \in H$ and each house location corresponds to exactly one friend. The constant α refers to the relative cost of driving vs walking.

A possible pickup schedule specifies the locations where we will pick up our friends. More specifically, it will specify for each friend m a pickup location $p_m \in V$, and we will need to drive our car starting from our home to pass from all the pickup locations to collect our friends and bring them back home (assume that the car has enough capacity to carry all our friends).

Cost structure: Each friend incurs a walking cost equal to the distance traveled to get from his home to his pickup location. we incur a driving cost equal to the distance traveled by our car multiplied by the constant α, $0 \leq \alpha \leq 1$. It is in general more efficient to travel by car than walking, and the cost of the car does not depend on how many people it carries.

Consider the Party Together Problem (PTP) with inputs (G, H, α). Here, $G = (V, E)$ is an undirected graph where V represents locations in the city and each edge $(u, v) \in E$ has a non-negative weight representing the road length between u and v. our house is at location 0. The set $H \subset V$ represents the homes of our friends. The constant α represents the relative cost of driving compared to walking.
The objective is to develop a transportation plan to minimize the total cost of driving and walking, where the driving cost is scaled by α. The solution involves the following heuristics:

First, compute the shortest paths between all pairs of nodes using the Floyd-Warshall algorithm. This ensures efficient distance calculations, which are crucial for evaluating different pickup strategies.

Two heuristics are used for selecting initial pickup locations for friends:

1. Distance-Based Selection
   - Calculate the shortest path distance from the starting point (our house) to each home in H.
   - Sort the homes by their distance from the house.
   - Select a subset of homes to visit based on this sorted list, focusing on the closest locations first.

2. Random Selection

   - Randomly select a subset of homes from H. This heuristic introduces variability and explores different configurations that may not be considered by a purely distance-based approach.

A local search heuristic is also used to iteratively improve the tour:

1. Tour Modification:

   - Evaluate the cost of removing or adding nodes to the tour.
   - Select the configuration that results in the lowest total cost.

2. Solidify Function:

   - Further refine the tour by exploring possible insertions and deletions of nodes.
   - Select the most cost-effective configuration by comparing the cost before and after each modification.

Moreover, an approach similar to a genetic algorithm is used by maintaining the top k tours and performing crossover and mutation operations on them. Each tour is considered as a species that will undergo crossover and mutation:

1. Crossover:

   - Select pairs of tours from the top k tours.
   - For each pair, combine parts of both tours to create new offspring tours. This can be done by randomly selecting a crossover point and swapping segments of the tours.

2. Mutation:

   - Introduce random changes to some tours to explore new configurations. This can include swapping nodes, reversing segments, or randomly inserting/removing nodes.

3. Selection:

   - Evaluate all tours (both original and newly generated) and select the top k tours based on the lowest total cost.
   - These selected tours will form the basis for the next iteration of crossover and mutation.

Lastly, we assign specific pickup points to minimize walking distances:

- For each home $h_m \in H$, find the closest node on the tour.
- Create a dictionary mapping each pickup location to the friends picked up at that location, ensuring minimal walking distance for all friends.

This approach combines various strategies to effectively solve the PTP, balancing exploration and exploitation to find a practical and efficient transportation plan.

Constrained Version of Project Party Together

In this part, we consider a simpler version of PTP, namely Pickup from Home Problem (PHP). The problem has the additional constraint that we must pick up our friends at their homes (so we don't need to worry about optimizing over the set of pickup locations).

To solve the M-TSP using dynamic programming, we implement a function `mtsp_dp`(G). Here, G is a complete graph with nodes and edges representing the city, where the triangle inequality holds. The DP approach involves:

1. Defining a DP table `dp` where `dp`$[$`mask`$][u]$ represents the minimum cost to visit all nodes in the subset represented by `mask` ending at node u.
2. Initializing `dp` such that `dp`$[1][0] = 0$ (starting at node 0).
3. Iterating over all subsets of nodes and updating the DP table based on the shortest paths between nodes.
4. Extracting the minimum cost tour by backtracking through the DP table.

The implementation of `mtsp_dp`(G):

```python
import networkx as nx
from itertools import combinations

def mtsp_dp(G):
    N = G.number_of_nodes()
    dp = [[float('inf')] * N for _ in range(1 << N)]

    dp[1][0] = 0
    node_list = list(G.nodes())

    edge_weight = {}
    for u in node_list:
        for v in node_list:
            if u != v:
                edge_weight[(u, v)] = G[u][v]['weight']
    edge_weights = [[edge_weight[(node_list[u], node_list[v])] if u
    != v else 0 for v in range(N)] for u in range(N)]

    for mask in range(1 << N):
        u_mask = mask
        v_mask_init = mask ^ ((1 << N) - 1)
        while u_mask:
            u_idx = u_mask.bit_length() - 1
            u_mask ^= (1 << u_idx)

            v_mask = v_mask_init & ~(1 << u_idx)
            while v_mask:
                v_idx = v_mask.bit_length() - 1
                v_mask ^= (1 << v_idx)

                new_mask = mask | (1 << v_idx)
                new_cost = dp[mask][u_idx] + edge_weights[u_idx][v_idx]
                if new_cost < dp[new_mask][v_idx]:
                    dp[new_mask][v_idx] = new_cost

    min_cost = float('inf')
    last_node = -1
    for i in range(1, N):
        cost = dp[(1 << N) - 1][i] + edge_weights[0][i]
        if cost < min_cost:
            min_cost = cost
            last_node = i

    mask = (1 << N) - 1
    tour = [0]

    while last_node != 0:
        tour.append(node_list[last_node])
        next_node = -1

        v_mask = mask & ~(1 << last_node)
        while v_mask:
            v = v_mask.bit_length() - 1
            v_mask ^= (1 << v)
            if dp[mask][last_node] == dp[mask ^ (1 << last_node)][v] + edge_weights[last_node][v]:
                next_node = v

        mask = mask ^ (1 << last_node)
        last_node = next_node

    tour.append(0)
    tour.reverse()

    return tour
```

To solve PHP by reducing it to M-TSP, we implement `pthp_solver_from_tsp`(G, H) that involves:

1. Constructing a complete graph G' consisting of nodes in $H \bigcup \{0\}$ where each edge weight is the shortest path distance in G.

2. Solving the M-TSP on G' using mtsp_dp(G').

3. Reconstructing the tour in the original graph G based on the solution from G'.

The implementation of pthp_solver_from_tsp(G, H):

```python
import networkx as nx
from mtsp_dp import mtsp_dp
from student_utils import *

def reconstruct_tour(G, mtsp_tour):
    actual_tour = [0]
    for i in range(len(mtsp_tour) - 1):
        path = nx.shortest_path(G, source=mtsp_tour[i], target=mtsp_tour[i+1], weight='weight')
        actual_tour.extend(path[1:])
    return actual_tour

def create_complete_graph(G, nodes):
    G_complete = nx.complete_graph(nodes)
    for u in G_complete.nodes():
        for v in G_complete.nodes():
            if u != v:
                G_complete[int(u)][int(v)]['weight'] = nx.shortest_path_length(G, source=int(u), target=int(v), weight='weight')
    return G_complete

def pthp_solver_from_tsp(G, H):
    nodes = list(set([0] + H))
    G_prime = create_complete_graph(G, nodes)
    mtsp_tour = mtsp_dp(G_prime)
    php_tour = reconstruct_tour(G, mtsp_tour)
    print(php_tour)
    return php_tour

if __name__ == "__main__":
    pass
```

The correctness of the mtsp_dp algorithm for solving the M-TSP (Multi-Traveling Salesman Problem) is ensured by the dynamic programming approach which systematically evaluates all possible subsets of nodes and their corresponding paths. The algorithm initializes the DP table with the starting node and iterates through all subsets of nodes, updating the DP table with the minimum cost required to visit all nodes in each subset. By iterating over all possible subsets and ending nodes, the algorithm guarantees that the minimal cost path to visit all nodes exactly once is found. The final extraction of the tour by backtracking through the DP table ensures that the optimal path is constructed, preserving the minimal cost property.

The reduction of the PHP (Pickup from Home Problem) to M-TSP and its solution using the pthp_solver_from_tsp algorithm relies on constructing a complete graph G' from the original graph G. This graph G' includes only the nodes that are required (homes of the friends and the starting node). The weights of the edges in G' represent the shortest path distances in G. By solving the M-TSP on G' and then reconstructing the tour in the original graph G, the algorithm ensures that the shortest possible paths are used between the required nodes, while the overall tour remains optimal in terms of cost. The correctness is thus guaranteed by the optimality of the shortest paths used to construct G' and the correctness of the M-TSP solution on G'.

Theoretical Questions

Show that PTP is NP-hard.

Let's consider a known NP-hard problem, Metric Travelling Salesman Problem (M-TSP).

Given an instance of M-TSP with a complete graph $G_e = (V_e, E_e)$ where the triangle inequality holds, we construct an instance of the Pickup from Home Problem (PHP) with the same vertex and edge sets representing home locations. In this case, the optimal PHP solution must visit each node exactly once, mirroring the M-TSP solution. Since the transformation is polynomial in time, PHP inherits the NP-hardness from M-TSP.

Conversely, PHP can be reduced to M-TSP. Any instance of PHP can be transformed into an instance of M-TSP by creating a complete graph where the weight of each edge corresponds to the shortest path distance in the original graph. Solving the M-TSP instance and converting the tour back to the original graph provides a solution for PHP. This process is also polynomial in time.

To show that PTP is NP-hard, we can use a reduction from PHP to PTP.

First, recall that a problem X is NP-hard if every problem in NP can be reduced to X in polynomial time. Given that PHP (Pickup from Home Problem) is NP-hard, we need to show that PTP (Party Together Problem) is at least as hard as PHP.

Consider an instance of PHP. In this instance, we have a fixed set of pickup locations (the friends' homes) and must find the optimal route for picking up all friends and returning to the starting point (our home). Let's denote this instance as (G, H, α), where G represents the graph, H represents the set of friends' homes, and α represents the relative cost of driving versus walking.

We can reduce PHP to PTP by setting $\alpha = 0$. In this scenario, the driving cost is irrelevant since α scales the driving cost to zero. Therefore, the optimal solution for PTP when $\alpha = 0$ is to pick up each friend directly at their home, which is exactly the PHP problem. This reduction shows that solving PTP with $\alpha = 0$ is equivalent to solving PHP.

Since PHP is NP-hard and we have shown that solving PHP can be reduced to solving PTP with $\alpha = 0$, it follows that PTP is also NP-hard. This proves that $x \in \texttt{PHP} \iff f(x) \in \texttt{PTP}, \forall x \in \{0,1\}^*$. The transformation involves setting α to zero, which can be done in constant time. Hence, f is a polynomial time transformation, which proves that PTP inherits NP-hardness from PHP.

Show that the cost of PHP is at most twice that of the optimal solution (which we don't know). That is, $\beta = \frac{C_{php}}{C_{ptpopt}} \leq 2$. Also show that this bound is tight, i.e., there is an instance where $\beta = 2$ (at least asymptotically). We can assume $\alpha = 1$ for simplicity.

To show that the cost of the Pickup from Home Problem (PHP) is at most twice that of the optimal Party Together Problem (PTP) solution, we let C_{php} be the cost of the solution to the PHP and C_{ptpopt} be the cost of the optimal solution to the PTP. Assume $\alpha = 1$, meaning driving cost equals walking cost.

In PHP, we start from our home, drive to each friend's home to pick them up, and return home. In PTP, all friends meet at a single point (optimal meeting point) and then go to the final destination together.

In PHP, the cost is the sum of the distances from our home to each friend's home and back. Let's denote the driving distances as d_i for each friend i:

$$C_{php} = 2\sum_i d_i$$

In PTP, the optimal meeting point minimizes the total cost. If we denote the optimal meeting point as M, the cost C_{ptpopt} is:

$$C_{ptpopt} = \sum_i d'_i + d(M, \text{home})$$

, where d'_i is the distance each friend i travels to M and $d(M, \text{home})$ is the distance from M to the final destination (home).

Notice that in the worst case, $d'_i \leq d_i$ because the optimal meeting point M minimizes the distances. Therefore:

$$C_{ptpopt} \leq \sum_i d_i + d(M, \text{home})$$

Since $d(M, \text{home}) \leq \sum_i d_i$ (as M is chosen optimally and could be closer to home):

$$C_{ptpopt} \leq \sum_i d_i + \sum_i d_i = 2\sum_i d_i$$

Thus:

$$\frac{C_{php}}{C_{ptpopt}} \leq \frac{2\sum_i d_i}{C_{ptpopt}} \leq 2$$

Consider a scenario where there are two friends located at the same distance d from the starting point (our home). Let the home be at point A, and the two friends' homes be at points B and C, each at distance d from A and equidistant from each other.

In PHP, we drive from A to B, then to C, and back to A:

$$C_{php} = 2d + 2d = 4d$$

The optimal meeting point M is the midpoint between B and C, at distance $\frac{d}{2}$ from both B and C. In PTP, the total cost is driving each friend to M, then from M to A:

$$C_{ptpopt} = d + d = 2d$$

The ratio β is:

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

This example shows that the bound is tight as there exists an instance where the cost of PHP is exactly twice the cost of the optimal PTP solution. Therefore, $\beta = 2$ is both an upper bound and a tight bound.