Marmara University
Faculty of Engineering

TSP with Penalty (Prize-Collecting TSP)

1. Introduction

TSP (Traveling Salesman Problem) is very popular problem for years. This problem says that there are a salesman and n cities, this salesman starts any of these cities and he has to visit all cities exactly one, finally he has to come to the city where he started. Visiting one city to another has some cost like distance of going from one city to the other. The aim of TSP is to find a path with minimum cost. It is an NP-Hard problem.

Finding the path with minimum cost takes exponential time even if the most powerful algorithms are used. But it is possible to approach minimum cost with polynomial times. This project's problem is TSP with Penalty which is very similar to normal TSP, but it includes some differences. In TSP with Penalty, salesman does not have to visit all cities but for cities which salesman does not visit, there is a penalty which adds to cost. Aim is finding minimum cost path in these conditions. This problem is at least as hard as normal TSP so finding the optimal path (path with minimum cost) takes exponential time. However, to approach can be done in polynomial time.

2. Purpose

The purpose of this project is to develop an algorithm to approach optimal solution of TSP with Penalty. Also, this algorithm must give the output in feasible time.

3. Other Algorithms That Used in Project

a. **Nearest Neighbour Algorithm:** This algorithm is a greedy algorithm for TSP. The salesman starts in a city and always goes to the nearest city from his current position. This is a very basic and fast approach algorithm. Although it often approaches the optimal solution, in some cases, traveling to the nearest cities results in a higher cost than the optimal solution.

b. **2-opt:** This algorithm is used to improve the current path in TSP. If a path exists, for example, one constructed with nearest neighbour algorithm, it is possible to reduce the cost of the path using 2-opt algorithm. It considers every pair of edges in the current path and reconnects these cities. If the reconnected version of path's cost is less than the old one, it applies reconnection and reverses all connections between changed cities. This algorithm is very useful and fast iterative improvement algorithm that is used in many TSP.

4. Our Algorithm for TSP With Penalty

We improved an algorithm for TSP with penalty using combination of nearest neighbour algorithm and 2-opt algorithm. Also, we used dynamic programming and iterative improvement techniques. Firstly, this algorithm reads current input file, gets

positions of every city and penalty value. Then, calculates distances of all city pairs and saves these values in a lower triangular matrix (to use less memory, it was implemented as lower triangular instead of n by n 2-dimensional matrix). After these procedures, the algorithm finds an initial path with nearest neighbour algorithm visiting all cities. After that, it uses 2-opt algorithm on the initial path to make it more optimal.
Next, it skips some cities. The *compute_penalized_cost* function optimizes a tour by minimizing the combined cost of travel distances and penalties for skipped nodes. It uses dynamic programming (dp[]) to store the minimum cost required to reach each node i, considering previous positions i - j where j is 1 to i. For each possible jump length j, it computes the cost as the sum of minimum cost to reach i - j, the distance from tour[i - j] to tour[i], and a penalty of (j - 1) * penalty for skipping j - 1 nodes. If this cost is lower than the current dp[i], it updates dp[i] and stores the corresponding previous node in prev[i]. After building the dp table, it backtracks from the last node using the prev[] array to reconstruct the optimal tour into new_tour[]. It marks which nodes are used, then reverses new_tour into the original tour[] and updates their positions. After this step, it counts how many nodes were skipped (not used in the new tour), prints that count, and updates N to reflect the new tour length. This approach balances shorter paths with the penalty of skipping nodes to produce an efficient route.
Afterwards, algorithm applies 2-opt algorithm again and final path is constructed. Finally, program creates an output file and writes results in it.

5. Ideas Behind This Algorithm

The ideas behind this algorithm are rooted in dynamic programming, iterative improvement, and problem reduction. Given the large input size, our primary focus was to make the algorithm computationally efficient. To that end, we utilized dynamic programming as a core strategy—not necessarily to find the optimal solution, but to approach it as closely as possible within practical constraints. In pursuit of improved solution quality, we also incorporated principles of iterative improvement and a problem-reduction-inspired perspective.

Dynamic programming is used to store and retrieve the minimum cost (or distance) between two nodes, and it also plays a crucial role in penalty computation. In this context, we adopt an approach inspired by the classic change-making problem. The algorithm iteratively determines how many nodes to skip in order to minimize the total cost of reaching a given node. For each node i, we update a dp array where dp[i] represents the minimum cost to reach that node. The update is performed using the formula dp[i] = min(dp[i - j] + d + (j - 1) * penalty) for all valid values of j from 1 to i. In this formula, j indicates the number of skipped nodes, d is the cost between nodes i - j and i, and (j - 1) * penalty accounts for the additional penalty incurred by skipping intermediate nodes. This approach allows the algorithm to evaluate multiple skip options and select the one with the lowest cumulative cost.

Although iterative improvement is not directly used in its classical form (such as greedy local search or random perturbations), the algorithm incorporates its spirit within the compute_penalized_cost function. This function leverages dynamic programming to explore multiple subpath combinations and chooses the one with the lowest cumulative cost, effectively pruning some nodes based on a penalty model. While this process is not iterative in the traditional sense—since the decision is made in a single global pass—it attempts to improve the solution by evaluating several alternative partial tours. Therefore, iterative improvement is not the primary mechanism, but rather an embedded strategy within a globally optimized framework driven by dynamic programming.

We do not explicitly apply problem reduction in the classical sense; however, our approach is inspired by similar principles. Instead of reducing the problem size or transforming it into a simpler version, we reuse the solution of the standard TSP as a starting point for the penalized TSP. This reflects a reduction-like mindset, where knowledge gained from a simpler version of the problem is leveraged to guide the optimization of a more complex variant. While it is not a formal problem reduction, it embodies a similar conceptual approach.