Jeongbin Son

CS470

Programming Project 1: TSP

February 15th, 2024

Traveling Salesman Problem

This paper aims at comparing three different algorithms for the Traveling Salesman Problem: brute-force, nearest neighbor, and an original heuristic algorithm.

The Traveling Salesman Problem (TSP) stands as one of the most intensively studied puzzles in the domain of computational mathematics and operations research, encapsulating the challenge of finding the shortest possible route that a traveling salesman can take to visit each city in a list once and return to the original city. The problem is classified as NP-hard, indicating that no efficient solution exists that can solve all instances of the problem within polynomial time as the number of cities increases. Despite this complexity, brute force implementations for TSP exist and serve as a fundamental approach to understand and explore the problem's intricacies. A brute force algorithm for TSP methodically enumerates all possible permutations of the cities to determine the shortest possible path, offering a guaranteed way to find the optimal solution by sheer computational effort. Although impractical for large numbers of cities due to the exponential growth of computational time required (factorial in nature, given n! possible routes for n cities), brute force methods provide invaluable insights into the problem's structure and serve as a benchmark to assess the efficacy of more sophisticated algorithms.

Parallel to brute force methods, heuristic approaches like the Nearest Neighbor algorithm offer more practical solutions for larger instances of TSP, albeit without the guarantee of finding the optimal path. The Nearest Neighbor algorithm simplifies the decision-making process by constructing a path where the salesman always proceeds to the nearest unvisited city. This method significantly reduces

computational complexity and can quickly generate good, but not necessarily optimal, solutions, making it suitable for real-world applications where time and resources are limited.

This paper also discusses an original algorithm that modifies and improves upon the Nearest Neighbor implementation to hopefully produce a more optimal solution for a given graph of cities.

Brute Force Implementation

The implementation is designed to explore every possible permutation of cities to identify the path with the minimum traveling cost. At its core, the algorithm utilizes a recursive function, tsp, which iteratively constructs paths by visiting each unvisited city, accumulating the cost as it progresses, and updating the minimum cost and best path when a complete tour is found. The base condition for recursion is reached when all cities have been visited, signified by the path size equaling the number of cities, N.

The recursive nature of the function allows for a thorough exploration of the search space, albeit at an exponential computational cost, characteristic of brute force solutions. To facilitate this exploration, the function employs a backtracking approach, where each city is marked as visited upon inclusion in the current path, and subsequently unmarked upon retracing steps, ensuring all possible paths are considered. The graph representing the cities and the distances between them is read from a file, allowing for easy modification of the problem instance. The algorithm initializes the search from the first city, setting the initial path and marking the first city as visited. Upon completion, the algorithm outputs the minimum cost found for completing the tour and the corresponding sequence of cities that

constitute the best path, culminating in a return to the initial city.

This implementation effectively demonstrates the brute force approach to TSP, highlighting both its simplicity in concept and the limitations in scalability due to the exponential growth of possible solutions with the addition of more cities. By ensuring the examination of every potential path, the algorithm guarantees the identification of the optimal solution, making it a valuable tool for understanding the computational complexity inherent in TSP and for benchmarking the performance of more efficient heuristic or approximate algorithms designed to tackle larger instances of the problem.

Results of Brute-Force

Since a graph size of more than 100 lines is not reasonable to show, this paper will utilize the given size 100 and 1000 '.graph' files as well as the graphs given during the competition from here on out. Any graphs that the code is tested on that are below size 100, are the same as the size 100 graph omitting the lines that are not being run. Running the code for a size 3 graph, the program outputs the following image in the terminal.

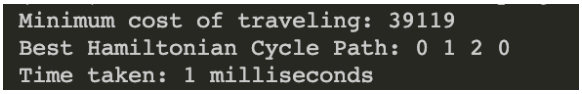

Figure A) Size3.graph; Brute Force

The time it took to solve the graph size of 3 was 1 millisecond and it computed a cost of 39,119. Calculating a size 3 graph from the size 10 graph by hand, offers the same solution as this brute force implementation. As for the size 10 graph, the program outputs:

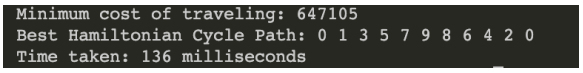

Figure B) Size10.graph; Brute Force

Here, the time it took to solve the size 10 graph using brute force was much longer, coming out to 136 milliseconds. However, the limitation for my brute force happens at above size 12, taking a while longer to finalize the computations it needs, hence the Big O (n!) time complexity.

Nearest Neighbor Implementation

This section details an implementation of the Nearest Neighbor algorithm for solving the Traveling Salesman Problem (TSP) within a graph of n cities, represented as an adjacency matrix. The algorithm begins by initializing a starting city (city 0) and iteratively selects the nearest, unvisited city as the next destination until all cities have been visited once, thus attempting to find a minimally costly Hamiltonian cycle. A Boolean array tracks visited cities to ensure each city is only visited once, and the total travel cost is accumulated as the algorithm progresses through the cities. Upon completion of the tour, the algorithm returns to the starting city, thereby completing the cycle, and updates the minimum cost and best path if the current iteration's path is more efficient than previously recorded paths.

The implementation includes a mechanism for reading the graph data from a file, ".graph," demonstrating an approach to handle external data sources in graph-related problems. This preparation phase populates the adjacency matrix with distances between cities, ensuring symmetry in the matrix to reflect the equal distances in both directions between any two cities. Following data preparation, the Nearest Neighbor algorithm is executed to approximate a solution to the TSP. The final

output presents the minimum cost of the computed Hamiltonian cycle and details the sequence of cities in the best path found, illustrating the algorithm's practical application to solve complex routing problems in a deterministic manner.

Results of Nearest Neighbor

To ensure the nearest neighbor works as intended, the following image is the output for the Size3.graph. Obtaining the same path cost as the brute force implementation, the nearest neighbor implementation shown in this paper can produce a path that goes from the nearest neighbor to the next nearest neighbor correctly.

```
Minimum cost of traveling: 39119
Best Hamiltonian Cycle Path: 0 1 2 0
Time taken: 0 milliseconds
```

Figure B) Size3.graph; Nearest Neighbor

At a graph size of 10, is when the program starts to differ in results from the brute force implementation. As the program finds the nearest neighbor to traverse to, the path cost that it returns may not be the optimal shortest Hamiltonian path cost. This method is useful for TSP graphs that contain large amounts of cities, or numbers, to calculate all different paths to.

```
Minimum cost of traveling: 1628604
Best Hamiltonian Cycle Path: 0 1 2 3 4 5 6 7 8 9 0
Time taken: 1 milliseconds
```

Figure C) Size10.graph; Nearest Neighbor

As shown above, the nearest neighbor produced a longer path cost as opposed to the shortest path cost brute force found at 647,105 for the size10 graph. Moving onto a graph size of 100, 1000 and even 15000, it becomes clearer how much longer the program will take to run the more cities the program has to traverse through.

```
Minimum cost of traveling: 3179955
Time taken: 5 milliseconds
```

Figure C) Size100.graph; Nearest Neighbor

```
Minimum cost of traveling: 1780054
Time taken: 259 milliseconds
```

Figure D) Size1000.graph; Nearest Neighbor

```
Minimum cost of traveling: 796026
Time taken: 54615 milliseconds
```

Figure E) g15000.graph; Nearest Neighbor

Nearest Neighbor offers a window into returning a Hamiltonian path cost that brute force implementations may never finish on time, or in a reasonable response time. However, even NN can be improved upon, by passing the NN solution to different optimization algorithms in hopes of finding a better, shorter Hamiltonian path cost.

Original TSP Implementation

The algorithm begins with the nearest neighbor heuristic to generate an initial solution. This heuristic selects the closest unvisited city as the next city to visit, starting from an arbitrary point, until all cities have been visited. This step provides a good starting point that is significantly better than a random start but is not necessarily the optimal solution. To refine this initial solution, the algorithm employs two optimization techniques: the 2-opt swap and simulated annealing, specifically tailored here as lattice annealing. The 2-opt swap iteratively replaces two edges with two different edges if the swap results in a shorter path, effectively eliminating routes that cross over each other and, hence reducing the total distance. Lattice annealing introduces randomness into the optimization process, allowing the solution to escape local optima by accepting worse solutions with a probability that decreases over time.

This step uses a cooling schedule to gradually reduce the temperature, thereby decreasing the likelihood of accepting worse solutions as the algorithm progresses.

The combination of these strategies—nearest neighbor for an initial solution, followed by lattice annealing and the 2-opt swap for optimization—makes this algorithm superior to brute force and simple nearest neighbor approaches. Brute force, while guaranteeing the optimal solution, is computationally infeasible for all but the smallest datasets due to its exponential time complexity. The nearest neighbor method, although fast, can easily get trapped in local optima, especially in cases where the path leads to a suboptimal route due to its greedy nature. By contrast, this algorithm strikes a balance between computational efficiency and solution quality. Lattice annealing introduces a controlled randomness that allows exploration of the solution space beyond local optima, while the 2-opt swap method systematically improves the solution by removing crossings. This hybrid approach ensures a more thorough exploration of the solution space, offering a practical compromise between execution time and the quality of the solution, making it particularly suited for large instances of the TSP.

Results of Original Algorithm

As always, testing the algorithm to work for the smallest size graph of Size 3, helps to understand the legitimacy of the code. Still able to produce the same outcome as the brute force and the nearest neighbor, the original algorithm for the TSP does solve the size 3 graph in the same path and cost as shown below.

```
Optimized path: 0 1 2 0
Optimized cost: 39119
Time taken: 1 milliseconds
```

Figure F) Size3.graph; Original Implementation

Testing with the size 1000 graph gives a better outcome than the nearest neighbor implementation reducing the path cost by 827,940. Showing the improvement that the original algorithm has made on the NN, it becomes important to understand how effective certain improvement algorithms are after running the Nearest Neighbor implementation on a graph.

```
Optimized cost: 952114
Time taken: 511 milliseconds
```

Figure G) Size1000.graph; Original Implementation

As for the competition, running the g15000.graph produced a path cost of 208,688. It only took 115 seconds for the algorithm to traverse through a graph size of 15,000. The brute force implementation this paper produced could never have run the graph size of 15,000. The NN could have ran, but the solution path would most likely have been worse than the original solution.

```
Optimized cost: 208688
Time taken: 115663 milliseconds
```

Figure H) g15000.graph; Original Implementation

As this paper has shown, the original algorithm improves upon the two other implementations for the TSP, optimizing it a little further in hopes to produce a more optimal solution. It has worked for the test case graph sizes that were given for the competition and for the project itself. The original solution, albeit using the nearest neighbor algorithm, improves upon it by adding more optimization algorithms after running the nearest neighbor. As such, it gave the algorithm the better chance at outperforming brute force and nearest neighbor.