**Solving the Traveling Salesman Problem Using Local Search with Simulated Annealing**

Steven Rummler

Brandon Scott

Caleb Sly

Brigham Young University

April 13th 2021

**Abstract**

This paper puts forth an algorithm for approximating a solution to the traveling salesman problem. A greedy approach is first examined. Then a fancier algorithm that includes depth first search, local search, simulated annealing, and a retry clause is examined. An empirical analysis shows that the fancier algorithm is better at finding lower cost solutions than the greedy approach in every case. We end with a discussion on the pros and cons of the fancy algorithm.

*Keywords:* traveling salesman problem, simulated annealing, local search, greedy, algorithm

# Greedy Algorithm

The greedy algorithm has a time complexity of O(n^2). The algorithm starts from an initial city, picks the shortest path available from that city to an unvisited city, and continues this process until all the cities have been visited. Each city is considered exactly once. The complexity of considering a single city is O(n) because each outgoing edge must be considered to find the smallest path. This results in an overall running time of O(n^2). We did add one enhancement, so the greedy algorithm did not get stuck. If it ever has an infinite cost for the path, we just restart at a new arbitrary start city. There is no complex data structure used within the greedy algorithm, so the space complexity is simply the space needed to store the problem, which is O(n).

# Fancy Algorithm

Our process of creating a good algorithm for approximating the traveling salesperson problem had four main phases:

* Optimizing the initial solution
* Local Search
* Simulated Annealing
* Adding a retry clause

We will explore each of these phases of our solution.

## Initial solution – Depth First Search

For our fancy algorithm we took a local-search-based approach. To fulfill local search’s requirement of always having an initial route, we had to design a better greedy algorithm that would always arrive at a solution.

The updated greedy algorithm works similarly to the basic greedy algorithm described above but includes backtracking. It keeps track of which routes have been tried from a node in a n by n table (n is the number of cities) and tries new paths in order of shortest distance. If there is not a valid complete tour, this algorithm will try every possible route. This is similar to how a depth first search works.

It has space complexity of O(n2), best-case time complexity of O(n2), and worst-case time complexity of O(n!\*n2). It is possible to reduce the worst-case complexity to O(n!\*n) by first checking for a valid path without taking the time to compare distances, but the graphs we’re working with are sufficiently dense that we never found one without at least one complete tour. We also felt it was best to leave that feature out because for best-case runs, which are common, it doubles the time taken.

We spent a decent amount of time optimizing the updated greedy algorithm, and it finds routes for around 7000 cities within the 60 second maximum time. City counts below 900 almost always take less than a second to process.

We labeled the updated greedy algorithm as dfs in our code and use it to initialize our local search algorithm. In the table, we run the old greedy algorithm that simply took the shortest path.

## Local Search

With this speedy initial solution, we wrote a local search algorithm that sought to incrementally improve the cost. The algorithm would compare the current solution to neighbors in the solution space. A neighbor was defined as a solution that could be obtained by switching the order of two cities of the current solution.

If the neighboring solution resulted in a better cost, that became the current solution. This was repeated until no changes were made for a length of time (specifically 1000 \* the number of cities). At this point, the best solution so far was returned.

While this approach was quick, it did not improve on the greedy costs as much as we would have liked, so we add simulated annealing to the algorithm.

## Simulated Annealing

Simulated annealing allowed our algorithm to possibly take a worse neighbor based on the current temperature. As the algorithm continues, the temperature dropped, making it less likely to take a worse neighbor. This allowed the algorithm to avoid local minimums and cover more of the global search space.

For the initial temperature, we used the depth first search’s initial cost. The temperature cooled by a factor of .003 every iteration of the algorithm. The possibly for accepting a worse neighbor was

1 / (1 + e^(delta / current temperature))

where delta is the difference in costs between the two neighbors. We pulled from the following sources for these formulas:

* <http://www.eng.uwaterloo.ca/~sjayaswa/projects/MSCI703_project.pdf>
* <https://www.theprojectspot.com/tutorial-post/simulated-annealing-algorithm-for-beginners/6>
* <https://www.mathworks.com/help/gads/how-simulated-annealing-works.html>

This approach generally improved the cost found by our local search algorithm, but sometimes the cost became worse.

## Retry

Sometimes, because of the annealing, our algorithm would get stuck down a less optimal solution space than our current best solution so far. In order to account for this, we allowed our algorithm to run on the best solution so far with a reset cooldown and a cool temperature. This meant that our algorithm would not take worse neighbors and would seek to find a local optimum using our best solution so far. This caused the local search with simulated annealing to consistently generate a better cost than local search without annealing. The downside of adding the retry and simulated annealing is that the algorithm became much slower, sometimes by a factor of 10 times.

# Results Table

# Discussion of Table

We took the average of 5 trials for city sizes of 15, 30, 60, 100, 200, 500, 100, 1500, and 2000 for each algorithm (Random, Greedy, Branch and Bound, and Local Search). Of all the algorithms, Greedy was by far the fastest across all city sizes. Greedy finished with an average run time of 10 seconds for 2000 cities. The only other algorithm that could finish a solution before the 10-minute limit was the Local Search algorithm with an average run time of 424 seconds. Nonetheless, the Local Search algorithm had the best average cost of tour for each tried city size. The Local Search algorithm on average had a cost of tour score 6% lower than Greedy. Therefore, Local Search is the more effective algorithm at finding the best minimum cost solution while Greedy finds good, low-cost solutions quickly.