The Traveling Salesperson Problem: 2-Opt

Basic 2-Opt Implementation and Comparison to Branch and Bound and Nearest Neighbor

Emily Asplund

Evan Chase

Justin McKeen

Juliana Smith

10 December 2022
CS 312-003
Professor Gordon Bean
BYU, Computer Science

Abstract

The Traveling Salesperson Problem is an NP-complete problem that has held the attention of experts in both the computer science and mathematics fields. In this report, we will be exploring the 2-Opt algorithm and its practical applications in the TSP. We will compare and contrast the results of our 2-Opt implementation with the Nearest Neighbor and Branch and Bound algorithms. While 2-Opt is a tried and true solution to the TSP, implementing and evaluating the results against other solutions helps us see how 2-Opt compares in terms of accuracy and speed.

i. Introduction

The Traveling Salesperson Problem is a problem that has held the focus of experts in the computer science and mathematical fields for decades. This is due to how difficult it is to actually solve in contrast with how simple it is to explain. Because of this, research into different algorithms that can be used to solve TSP has been ongoing.

The Traveling Salesperson Problem is an NP-Complete problem. This means that TSP can be verified in polynomial time, however, no efficient solution has been found. Polynomial time is considered efficient and ideal when solving complicated problems such as TSP. However, due to the nature of the problem, no efficient implementation has been found as of yet. Due to the fact that it's NP-Complete, finding a polynomial solution would indicate that other NP-Complete problems would also have a polynomial solution. However, we are far from finding such an algorithm.

One such implementation that has become popular in recent years is 2-Opt, originally proposed by Croes in 1958 (Croes, 1958). 2-Opt takes in a current, unoptimized path and tries to uncross certain pathways to find more cost-efficient routes.

Our group decided to implement our own version of 2-Opt to compare against a greedy algorithm (Nearest Neighbor) and a previously implemented Branch and Bound algorithm prior to this project. Our goal is to compare the results given by 2-Opt to these other implementations to see how it compares in terms of speed and accuracy. We will also break down the time and space complexities of each algorithm in order to truly assess if 2-Opt is a viable algorithm that competes with other well-used algorithms.

ii. Greedy Implementation: Nearest Neighbor

ii.a. Discussion

For our greedy algorithm, we decided to implement the Nearest Neighbor greedy algorithm. This algorithm is greedy since, with each city we visit, we pick the best path available (the path with the lowest cost) and continue until each city is visited. For our algorithm and our complexities, "*n*" is the number of cities we must traverse on the map.

ii.b. Data Type

We start by creating a cost matrix that is n x n in size. Each row represents a city and every column in that row represents the cost to travel to every other city, with an infinite cost indicating that a certain path does not exist. We then populate this matrix with the calculated costs to travel between all of the cities (*space and time for creation and populating the matrix are $O(n^2)$*).

Before traversing, we initialize an array to let us store our city path in *(space is O(n) when full)*. We start by picking a city at random and storing it in the first spot in our array and building a path from there.

ii.c. Traversal

For our traversal, we start with our first city (row) and check every city (column) in the row to find the best possible cost to an unvisited city. After we have found the "best city" (the city with the best value), we store "best city" in our array and make the "best city" our current city. We then update the matrix so the former city visited can not be visited again (*time is $O(n^2)$*).

If no path exists, meaning if no city can be traversed from our current city, we store an infinite cost in our array for the city's value. This will be important for evaluations later on.

We repeat this process for each city until we have reached every city and returned to our initial city. If we hit an infinite value, we restart the traversal from a different initial city. This problem means we potentially start at every city, giving us another N iteration. (*time is $O(n^3)$*).

ii.d. Gathered Info

After we have our path, we traverse through our array to gather the values for the Nearest Neighbor. As we traverse through the array, each value must be positive. If there is an infinite cost, this means that no path exists and the greedy algorithm could not find a greedy path. We now must start our greedy algorithm again with a different initial city. *(time is O(n), space is O(n))*.

Our total **time complexity** comes out to $On(n+n)^2 + O(n) = \mathbf{O(n^3)}$.

Our total **space complexity** comes out to $O(n^2) + O(n) = \mathbf{O(n^2)}$.

iii. 2-Opt Implementation

iii.a. Discussion

For our chosen algorithm, we decided to implement a variation of the 2-Opt algorithm. Our 2-Opt algorithm begins with an initial path, found via the previously discussed greedy algorithm. This solution path is converted to a linked list to allow for greater ease in performing the 2-Opt swap. We then iterate over every city in the route, our n value, checking if there is a valid 2-Opt swap within the next k cities. If such a swap exists, we compare its cost to the current path and perform the 2-Opt swap if there is an improvement. We then continue iterating through the remaining cities in the updated route. If any swaps were performed, we once more iterate through all the cities looking for improved paths, or until we have performed the swap search n times.

iii.b. Data Type

We start by creating a cost matrix that is n by n in size once more. Each row represents a city, and every column in that row represents the cost to travel to every other city. We then populate this matrix with the cost values used to travel between all the cities (*space and time for creation and populating the matrix are $O(n^2)$*).

We then create a circular linked list of size n, storing only a city index and a reference to the next node in the list. "*n*" is the total number of cities we must traverse on the map.

iii.c. 2-Opt Swap

The basis of the 2-Opt algorithm is the 2-Opt swap, which is used to untangle crossed paths. This is done by replacing the crossed connecting edges of two nodes and reversing the direction of every edge between the two (Croes, 1958). Figure 1 demonstrates this swap between nodes B and D on a 5-city TSP.

To check if there is a swap available, we search through the k nodes involved in the swap, checking if a reverse path exists, and calculating its cost. Performing and checking the viability of the swap is an *$O(k)$ time and $O(1)$ space* operation; we perform checks over k cities, choosing the best possible swap for a total complexity of *$O(k^2)$* time and *$O(k)$* space.

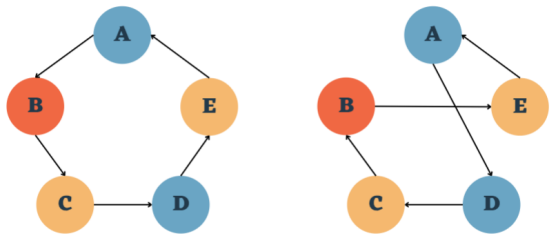


Figure 1: Example 2-Opt swap with B and D

iii.d. Traversal

Once an initial solution is found, we begin traversing over every city within the solution looking for possible 2-Opt swaps that will improve our solution. Our version of 2-Opt limits the number of cities traversed while looking for another city to swap with to a maximum of k cities away. This swap search is repeated until no new swaps have been made or for a maximum of n iterations. This traversal gives us a total time complexity of *$O(n^2k^2)$* and a space complexity of *$O(k)$*.

iii.e. Gathered Info

Once we have exhausted all possible swaps, we return the newly found solution *(time $O(n)$ space $O(n)$)*.

Our total **time complexity** comes out to $O(n^2) + O(n^2) + O(n) + O(n^2k^2) + O(n) =$ **$O(n^2k^2)$**.

Our total **space complexity** comes out to $O(n^2) + O(n^2) + O(n) + O(k) + O(n) =$ **$O(n^2)$**.

iv. Pros and Cons of 2-Opt

iv.a. Pros

- 2-Opt algorithm has incredibly fast run times, even on larger problem sizes
- The algorithm is relatively space-efficient, especially when compared to other solutions like Branch and Bound
- 2-Opt is a relatively simple solution to implement so it allows for testing of different variations

iv.b. Cons

- The produced circuits aren't always clean

- Because 2-Opt is a local search algorithm, optimal paths are not guaranteed
- 2-Opt only had small reductions on our Greedy approach
- 2-Opt doesn't work at all if Greedy doesn't return a valid path

v. Modifications to 2-Opt

v.a. The 'k' value

Our version of the 2-Opt algorithm had several additions we used to make the algorithm more efficient and to lower the expected time for the algorithm to run. The most notable of these changes was the addition of a "k" value to the algorithm to limit the distance that possible swaps would be searched for. Our reasoning for this was that since 20% of edges are set to infinity in the TSPs we were testing, the likelihood of a reverse path existing for long stretches of cities becomes very small as the distance from the current city increases. In our brief testing, we found that running the algorithm over large k values, like n, made no noticeable improvement to the execution and the expected length of the path. As such, we originally chose a very small value, 5, but while running on increasingly large n values, we found that the value of k was too small for noticeable improvements, so we increased it slightly.

v.b. Limiting additional iterations

In the original 2-Opt algorithm, when a change is found during iteration through the cities, you must repeat the search for possible swaps until no such improvements are found. This can have the potential to iterate up to n! times. Rather than searching until no improvements are found, we limit the number of times the algorithm can run to n, while still keeping the possibility of ending early if no such improvement can be found. In practical cases, this has no effect as the algorithm only loops through a couple of times before there are no effects. Fortunately, it does severely decrease the maximum possible run time.

vi. Results

All results ran for 10 minutes. If a solution was not found within that time, TB is listed. The tables below are the averaged results from five test trials for each algorithm of each city size. The "k" value we tested with was set to 50 cities for all trials.

Random

# Cities	Time (sec)	Path Length
15	0.00	21401
30	0.06	42670
60	28.65	81346

100	TB	TB
200	TB	TB
1,000	TB	TB
10,000	TB	TB

Greedy

# Cities	Path Length	% of Random
15	13189	61.63
30	21336	50.01
60	27842	34.23
100	37309	N/A
200	58241	N/A
1,000	161957	N/A
10,000	704816	N/A

Branch and Bound

# Cities	Time (sec)	Path Length	% of Greedy
15	1.14	10359	78.55
30	TB	TB	N/A
60	TB	TB	N/A
100	TB	TB	N/A
200	TB	TB	N/A
1,000	TB	TB	N/A
10,000	TB	TB	N/A

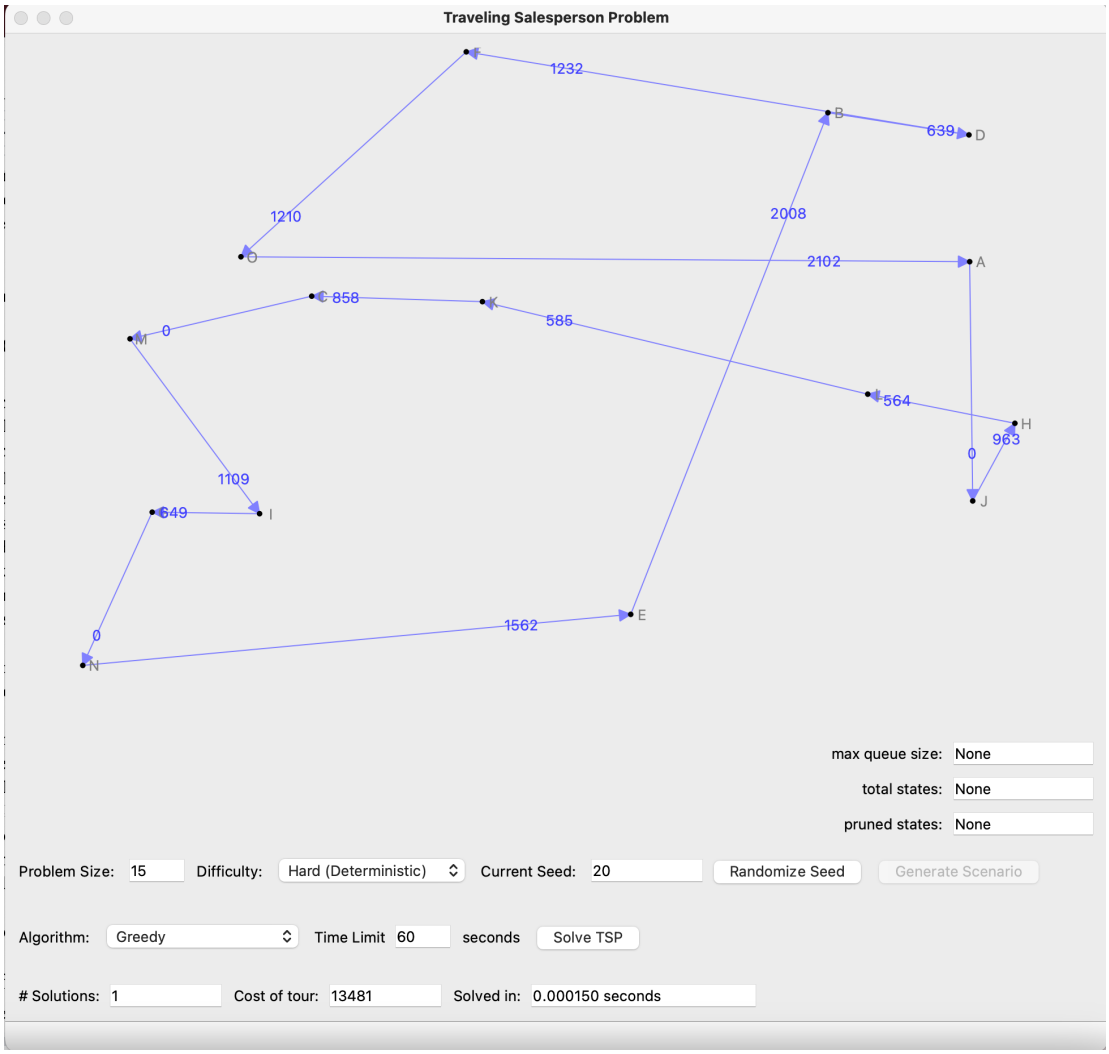
2-Opt

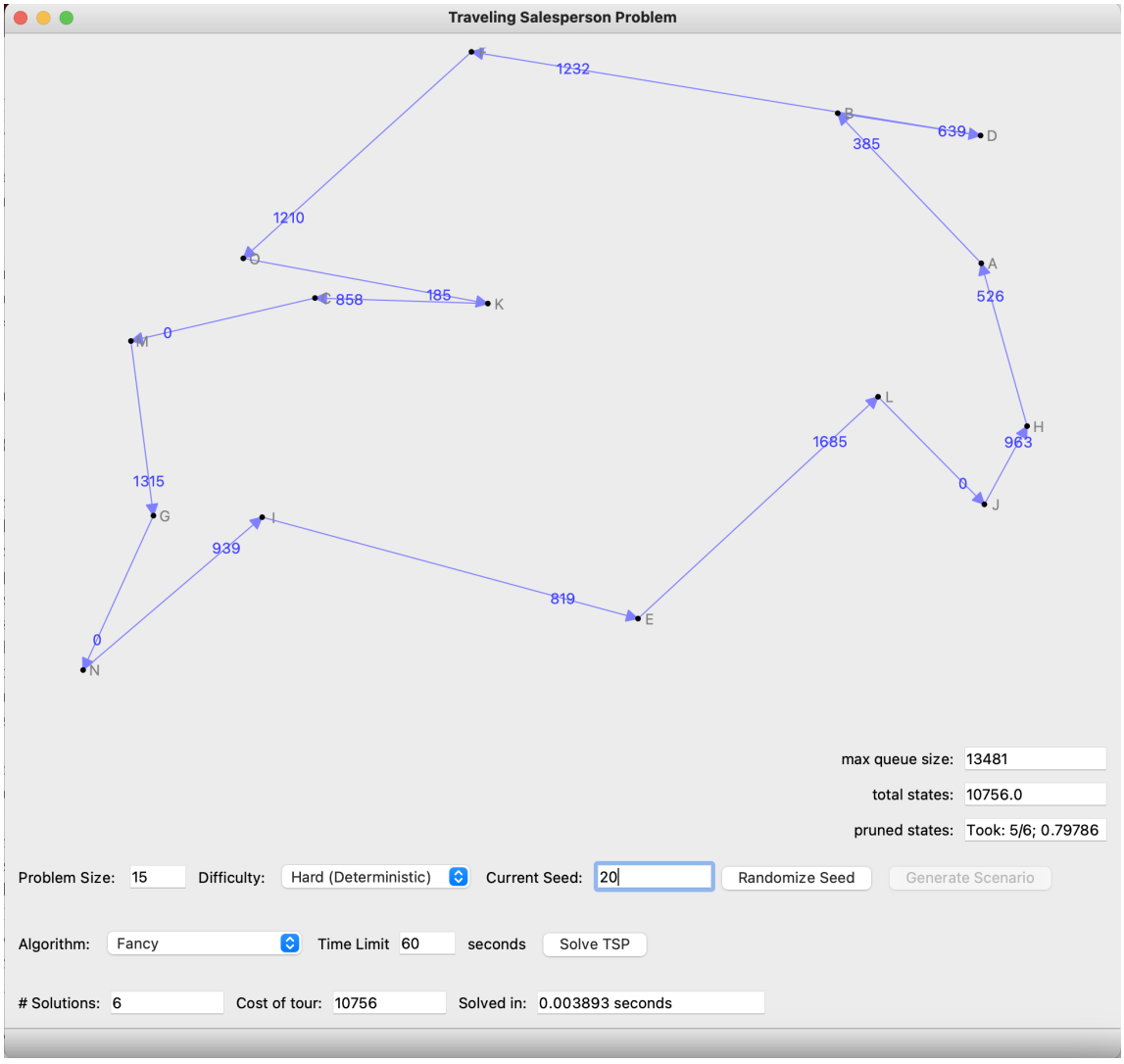
# Cities	Time (sec)	Path Length	% of Greedy

15	0.01	10772	81.68
30	0.02	20861	97.77
60	0.09	26313	94.51
100	0.12	35927	96.3
200	0.25	56483	96.98
1,000	3.19	158265	97.72
10,000	245.39	693006	98.32

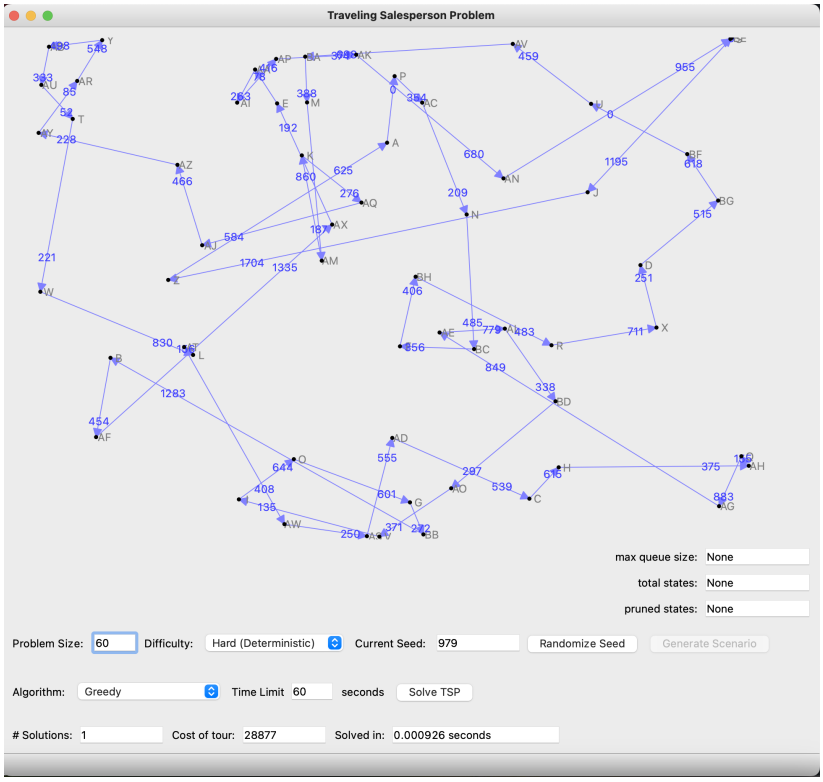
vii. Screenshots of Greedy vs 2-Opt

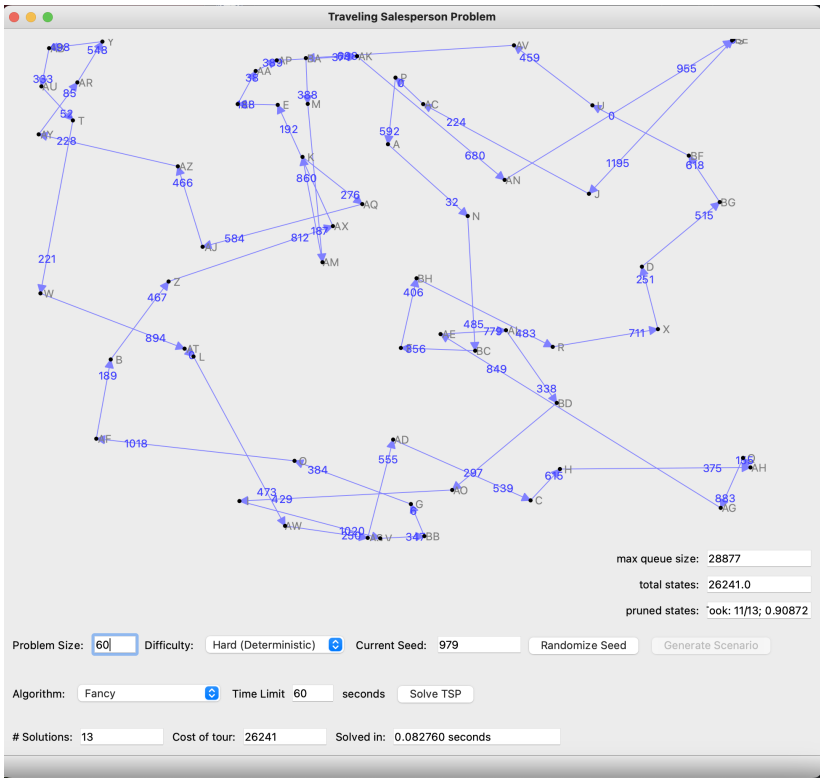
of Cities: 15 - Seed: 20:



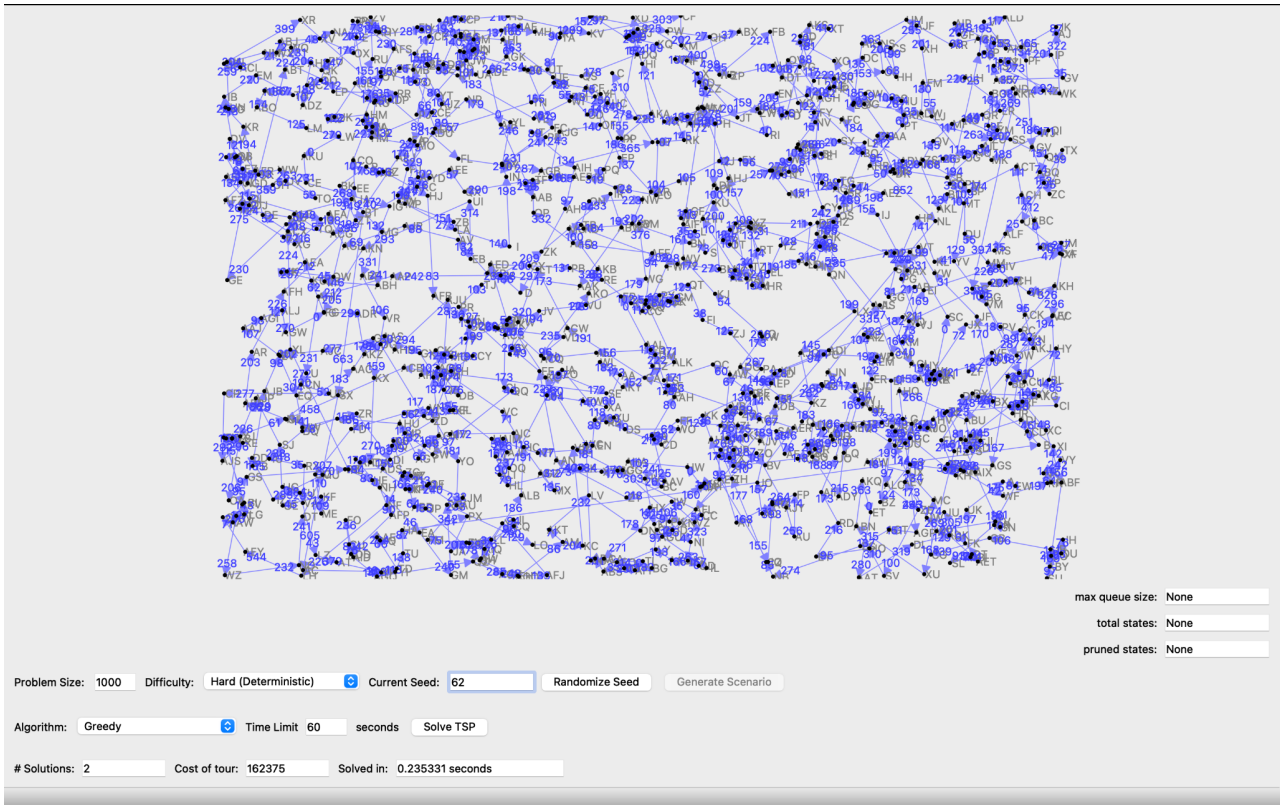

Traveling Salesperson Problem
1232
639
385
1210
526
858
185
0
1685
963
1315
0
939
819
0
max queue size: 13481
total states: 10756.0
pruned states: Took: 5/6; 0.79786
Problem Size: 15
Difficulty: Hard (Deterministic)
Current Seed: 20
Randomize Seed
Generate Scenario
Algorithm: Fancy
Time Limit 60 seconds
Solve TSP
Solutions: 6
Cost of tour: 10756
Solved in: 0.003893 seconds

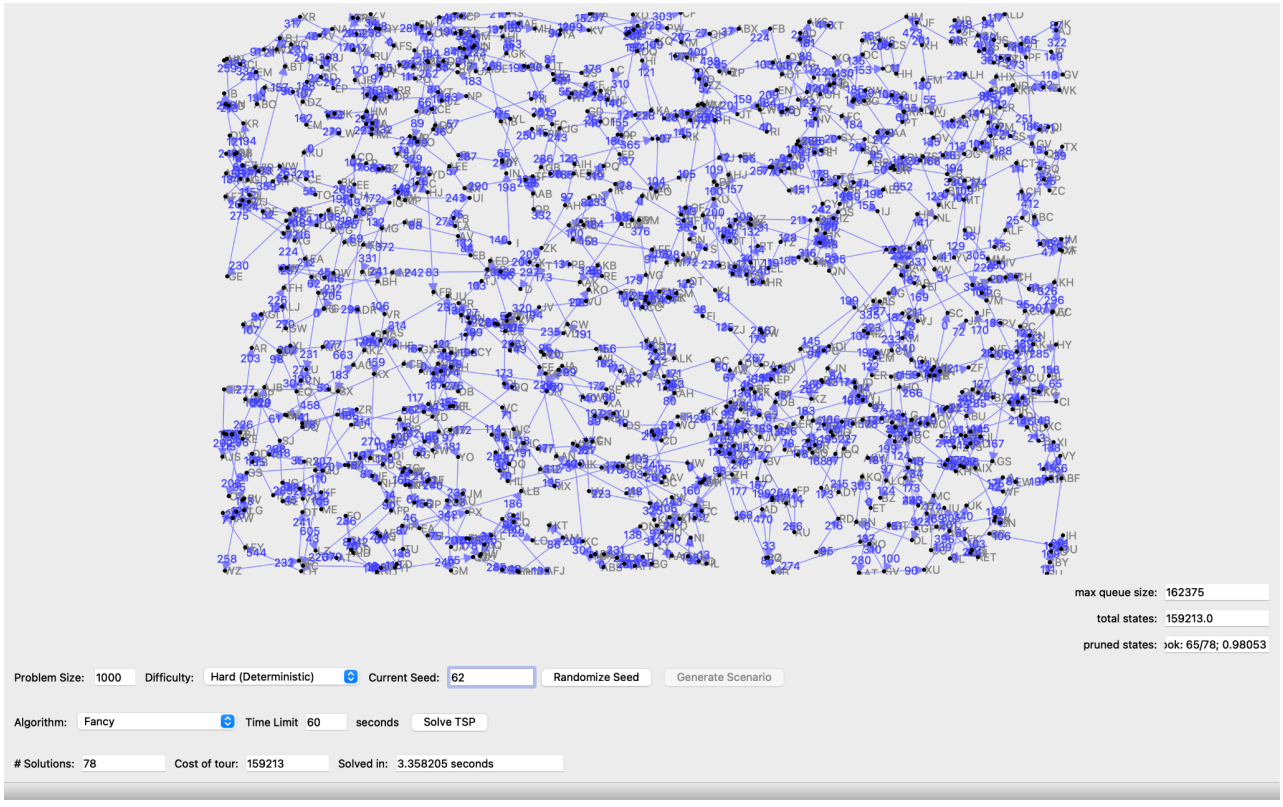
# of Cities: 60 - Seed: 979





# of Cities: 1000 - Seed: 62





viii. Analysis

We learned from these tests that the 2-Opt and Greedy approaches both have good time efficiency. We were able to go up to sample sizes in the 10,000 city size range and still find a solution in a timely manner. However, this does drastically affect 2-Opt's optimality.

2-Opt was typically able to reduce the cost of the greedy algorithm by about 2-8%. While this is still a reduction, it's not as significant as the branch and bound algorithm, which saw larger reductions in small city sizes. In medium and large city sizes, however, the Branch and Bound algorithm was unable to find an optimal solution within the 10-minute time period. With large numbers of cities, the greedy and 2-Opt approaches are able to find a solution much faster than the branch and bound approach, although it may come with a higher cost of travel.

This matches what we predicted from our theoretical time complexity, as Branch and Bound has a factorial worst case while 2-Opt and greedy have a worst case of O(n^3), which is significantly lower than a factorial time complexity. This allows 2-Opt and greedy to test much larger sample sizes than Branch and Bound.

ix. Conclusions

Overall, our implementation of the 2-opt algorithm is a highly efficient implementation for the traveling salesperson problem. We were able to test sample sizes of up to 10,000 cities without significant time loss. The efficiency of the algorithm does have a trade-off with optimality. While the Branch and Bound algorithm will return a more optimal solution, it's also too slow to return a solution in scenarios with large city sizes.

Our results show that 2-Opt and the Nearest Neighbor greedy algorithm are great options when a solution is needed quickly and when there are too many cities for Branch and Bound to run in a reasonable time frame. The "k" value that we implemented to limit how far our swaps reached was successful in helping the algorithm run quickly in large sample sizes.

In the future, we would like to continue to explore what distances are most efficient and see if we can continue to speed up the algorithm.

References

Croes, G. A. (1958). A Method for Solving Traveling-Salesman Problems. *Operations Research*, *6*(6), 791–812. http://www.jstor.org/stable/167074

Lin, S., & Kernighan, B. W. (1973). An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research*, *21*(2), 498–516. http://www.jstor.org/stable/169020