The Traveling Salesman Problem:
A Divide and Conquer and Nearest Insertion Approach

McKay Christensen
&
Dallin Christensen

April 13, 2021
Dr. Martinez
BYU, Computer Science

ABSTRACT

In an effort to find both a unique and efficient solution to the The Traveling Salesman Problem (TSP), our team developed our own algorithm. The algorithm consists of combining the divide and conquer algorithm used to compute a convex hull and a nearest insertion algorithm. In this paper, we discuss the algorithm creation process, its Big-O complexity, performance analysis (notably percent improvement over a simple greedy approach), and future work.

1 INTRODUCTION

Our team was tasked with implementing an algorithm which can solve the largest possible city problems with the most possible accuracy in a reasonable time. We were also tasked with implementing a greedy algorithm starting from an arbitrary city.

In implementing our algorithm, we first started looking at similarities between the ideal paths we had seen from previous algorithms. It seemed that most of the ideal paths formed some sort of perimeter around the outside of the cities, ending up back at the starting city as shown below.

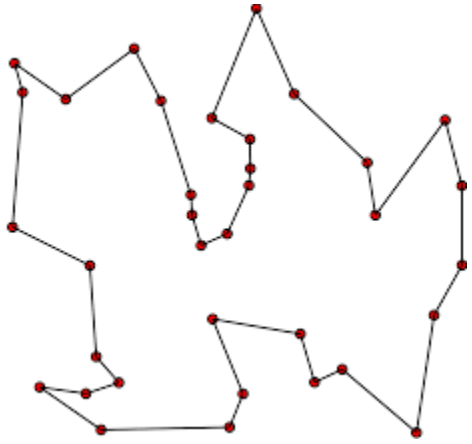


Figure 1 - Optimal path for TSP

In our algorithm, we developed a convenient way to speedily form such an outside perimeter by using principles we had practiced previously. We decided to start our algorithm by finding a perimeter that contains all of our cities. We used a similar algorithm to the Convex Hull project previously performed, to test each city and find a perimeter as shown below.

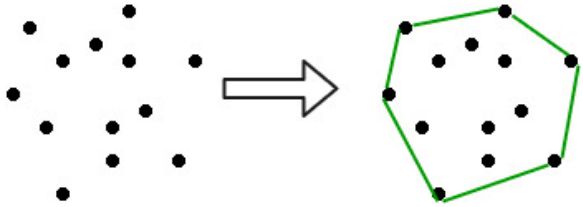


Figure 2 - Convex Hull

After finding a perimeter around the cities, we started inserting the cities left inside the perimeter one by one. We checked each city on the inside, and each possible insertion. By the time that all the cities are inserted, we have a reasonable solution to the TSP.

2 ALGORITHM EXPLANATION

2.1 Greedy

In order to demonstrate our algorithm's improved efficiency, we implemented a greedy algorithm. This algorithm has four main parts. First, a city is chosen at random as the starting point. This city is appended to a final route Python list and it is marked as the current city. Secondly, a while loop is used to establish a time limit on finding a greedy solution. Within that while loop, we iterate through each city in the list of cities. If it does not already exist in the final route list, we check the distance from the current city to that city and its cost is temporarily stored. If that cost is less than other costs from the current city to others, it is assigned as the current least expensive cost. Once the next least-cost path is found, the city is appended to the final route list and it becomes the current city. If at any point no path exists between the current city and the rest of the cities not already in the final cost path, the final route list is cleared, another city is arbitrarily picked as the starting point, and the process is run

again. This ensures that all options are considered before determining that there is no path between all the cities. Once all cities are placed in the final route list, we check if the final city can be connected to the starting city. If so, that becomes the path from the greedy approach. If not, we clear the final route list and start again as mentioned above. This greedy approach most resembles the nearest neighbor approach (Weru 2019), but rather than adding the nearest, it adds the cheapest.

The time complexity of this algorithm is Big $O(n^2)$. This is the worst case since each city may have a maximum of n edges and we check each city's edges n times while attempting to find the next best insertion. The space complexity stays O(n) since we have a list of cities of size n each time.

2.2 Convex Hull w/ Divide and Conquer

While implementing the Branch-and-Bound approach during our individual TSP projects, our team realized that often the optimal solution resembled that of the solution to a divide and conquer algorithm for a convex hull. With this in mind, we decided to use that algorithm to find an initial connecting perimeter. This would ensure that the edge cases of our cities were already in the path and closely connected.

We started by finding the convex hull of the cities, regardless of whether they were all connected. After acquiring that perimeter as a list, we then iterated through the perimeter cities attempting to connect each one to the next in the perimeter list. If there was no path to an adjacent city, the next city in the list is checked, and so on. For the most part, this gave us a reasonably connected outer city path. The biggest problem that we ran into was ensuring that the final city on the perimeter path connected to the first city. When this wasn't the case, we removed the last city on the list and repeatedly checked if the updated list made up a path that was less than infinity, which was the indication that no path exists. This preliminary perimeter calculation is at worst Big O(nlogn), since that is the time complexity of the divide and conquer approach of the convex hull. Once that list is obtained, iterating through each point is Big O(n) and each city lookup and comparison is Big O(1).

2.3 Nearest Insertion

When we first started to calculate which cities to insert, we tried to just calculate by which inside cities were closest to the cities within the perimeter, however we found that we would have to change our methods as just the minimum distance didn't take into account the length of the edge between adjacent cities. After that, we decided to measure the distance of both edges, before and after the inserted city. However, this method led the smaller edges to be inserted first, causing problems with later edges being added. Our last solution that fixed our insertion problem was to consider the ratio of how much the insertion would change the length of the previous edge. This ratio is defined by the sum of the two new edges divided by the length of the previous edge. By using this criteria, we were able to procure effective insertions to produce effective TSP solutions.

3 COMPLEXITY

When calculating the complexity of our algorithm, we found the following:

- The time complexity of our complex hull algorithm to find the perimeter: **O(n*log(n))**
- The time complexity of our algorithm to determine insertion locations and cities: **$O(n^2)$**
- The time complexity of insertion into path: **O(n)**

Altogether, our algorithm is
$T(n) = O(n*log(n)) + O(n^2) + O(n)$
Big O: $O(n^2)$

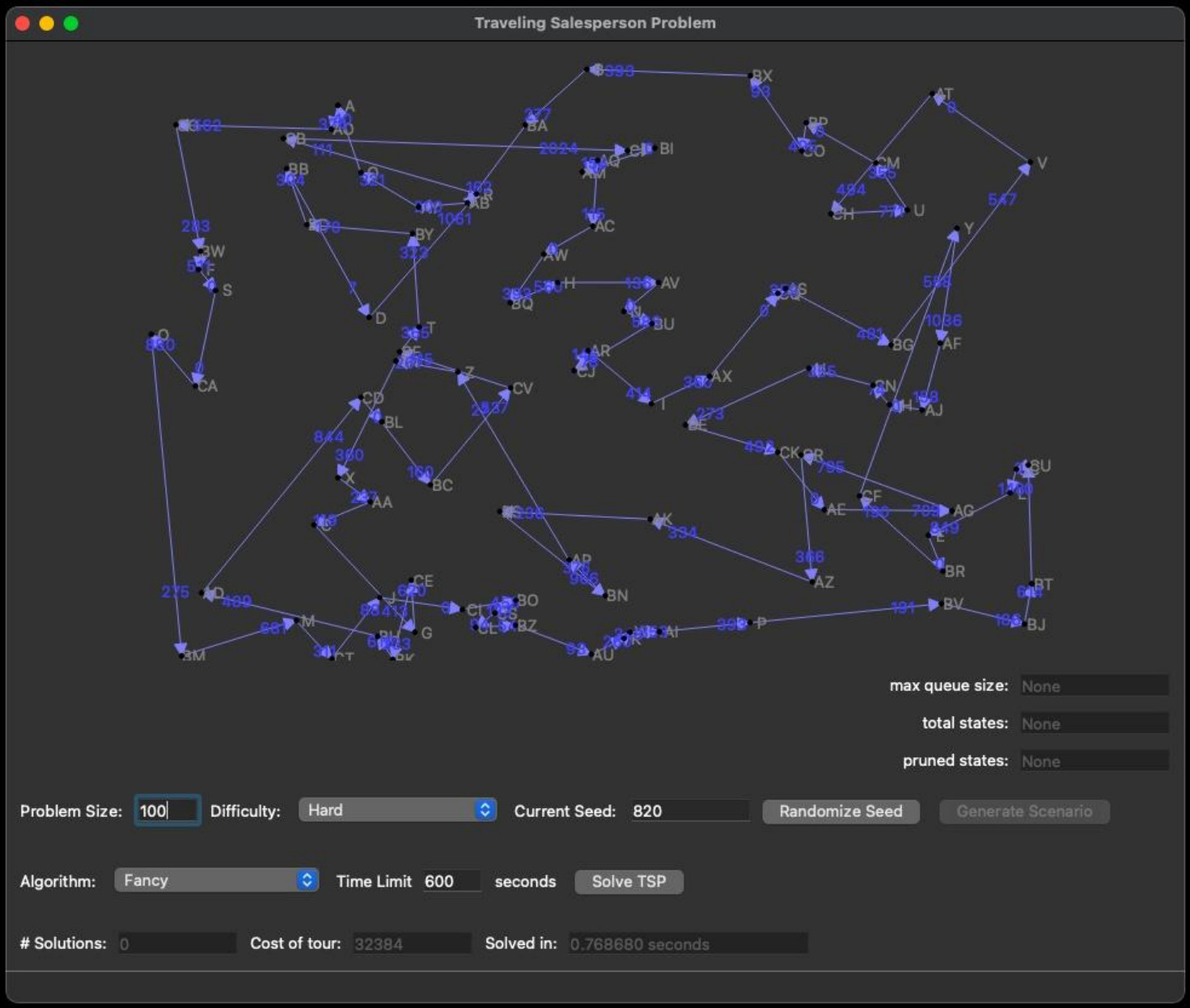
We also found the Space Complexity to be n, as each city is only stored once for each array. No matrices are necessary for computation with our algorithm.

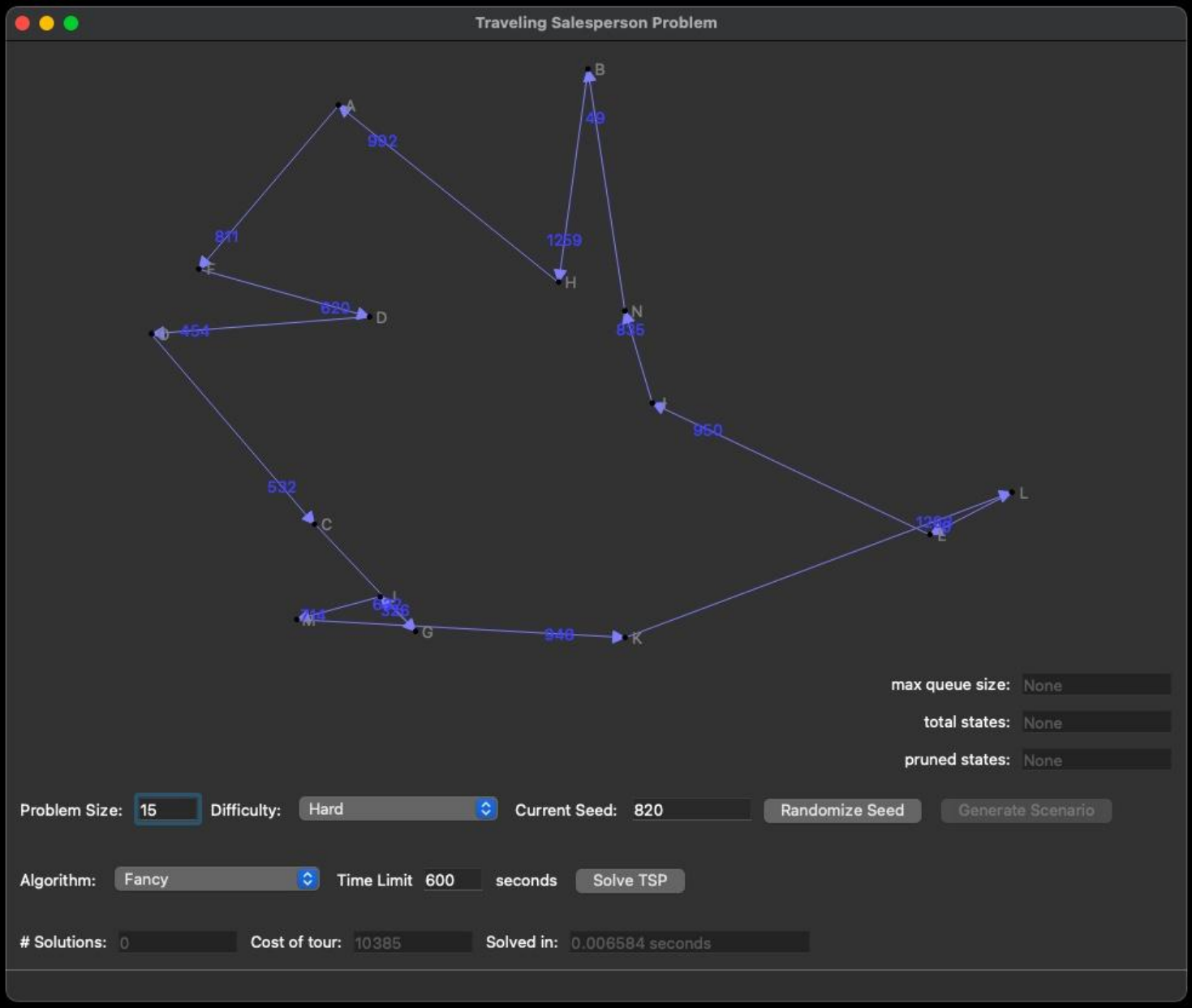
4 EMPIRICAL RESULTS AND ANALYSIS

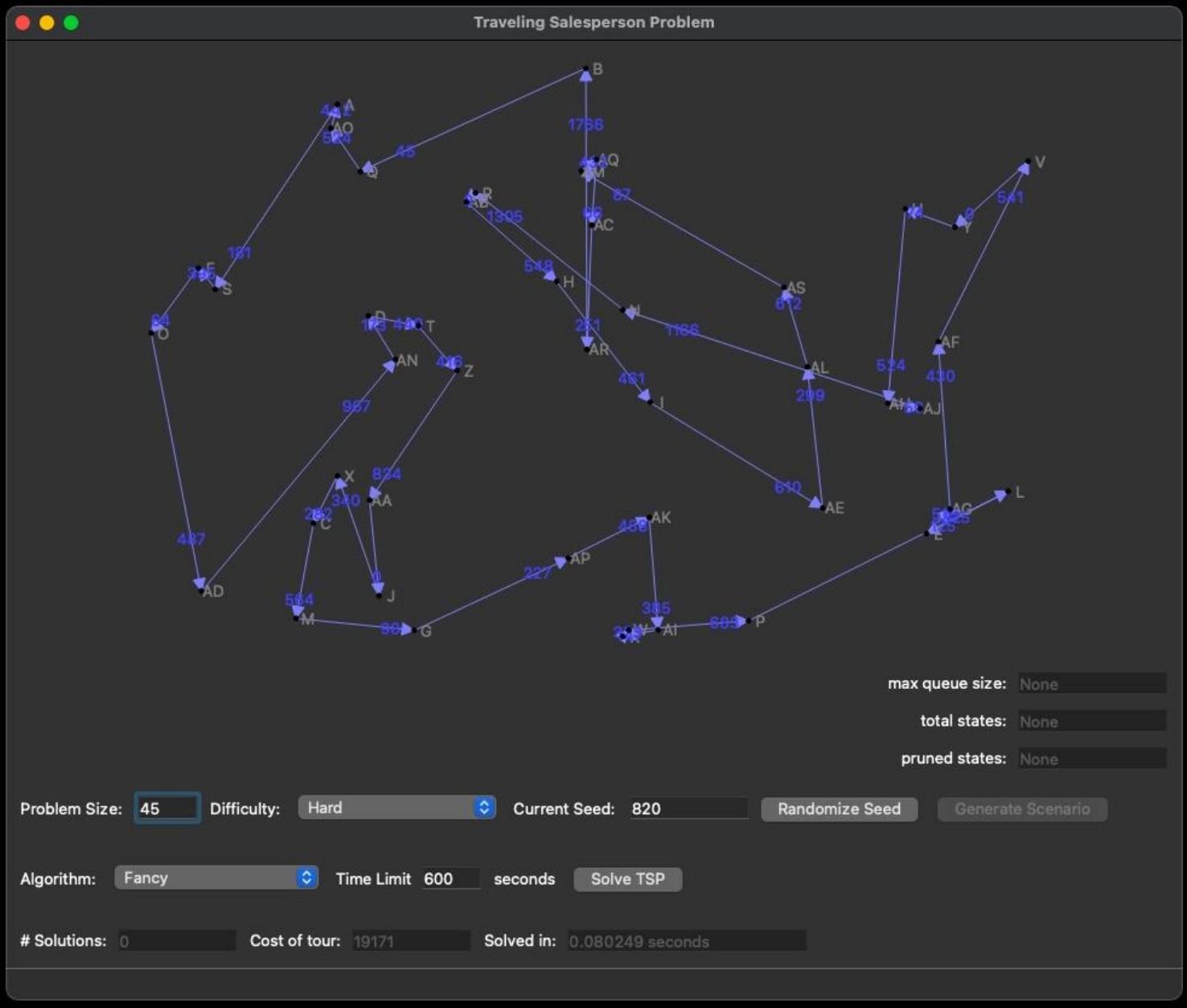
To be able to visualize the improvement that our algorithm had over the simple greedy implementation, several empirical tests were run. In Figure 3, a variety of situations each with different city sizes were used. In regard to the random insertion algorithm, once we tried 100 cities, the time it took to compute exceeded 10 minutes. This was to be expected since the random algorithm has a complexity of Big O(n!). The greedy algorithm performed the fastest during each test. Given that the Big O time complexity of this algorithm is at worst $O(n^2)$, this was also to be expected. The Branch and Bound has a particular time complexity since the implementation we used pruned off any state that wasn't lower than the calculated lower bound. This resulted in a complexity of $O(n^2 * b^n)$, 'b' being some amount of branches. This complexity meant that after approximately 20 cities, it couldn't handle computing the optimal solution in a reasonable time. Our algorithm consistently performed better than the greedy, and since its overall time complexity is also Big $O(n^2)$. In Figure 3, with # Cities 15, 30, and 60, it is shown that as the number of cities doubles, the time to compute for our algorithm increases by a factor of approximately 5. We also observed that while the greedy and our algorithm both have a theoretical Big $O(n^2)$, our algorithm has a constant k which is approximately 20, i.e. our complexity would be closer to $O(20n^2)$ compared to the greedy approach. The maximum that our algorithm could handle in a reasonable time was 800 cities. Although it took nearly 9 minutes, it still produced a solution that was better than the greedy approach. For all of our calculations, we found that our algorithm was consistent in its path length improvement. You will also find below our table examples of our algorithm with 100, 15, and 45 cities respectively.

	Random		Greedy			B&B			Div & Con Convex Hull + Nearest Insertion		
# Cities	**Time (sec)**	**Path Length**	**Time (sec)**	**Path Length**	**% of Random**	**Time (sec)**	**Path Length**	**% of Greedy**	**Time (sec)**	**Path Length**	**% of Greedy**
15	0.001	19099	0.001	11680	0.61	6.44	9113	0.78	0.004	9916	0.85
30	0.032	38474	0.002	17845	0.46	TB	TB	N/A	0.03	16277	0.91
60	17.46	79183	0.01	27027	0.34	TB	TB	N/A	0.17	22486	0.83
100	TB	TB	0.03	36917	N/A	TB	TB	N/A	0.75	31803	0.86
200	TB	TB	0.12	59351	N/A	TB	TB	N/A	6.77	49393	0.83
80	TB	TB	0.02	34830	N/A	TB	TB	N/A	0.41	27592	0.79
150	TB	TB	0.12	48627	N/A	TB	TB	N/A	2.98	41166	0.85
Biggest - 800	TB	TB	7.14	137755	N/A	TB	TB	N/A	494.00	121540	0.88

Figure 3 - Empirical analysis and comparison of each algorithm


Traveling Salesperson Problem
max queue size: None
total states: None
pruned states: None
Problem Size: 100
Difficulty: Hard
Current Seed: 820
Randomize Seed
Generate Scenario
Algorithm: Fancy
Time Limit 600 seconds
Solve TSP
Solutions: 0
Cost of tour: 32384
Solved in: 0.768680 seconds


Traveling Salesperson Problem
B
A
992
49
871
1259
H
D
N
950
L
532
C
G
K
max queue size: None
total states: None
pruned states: None
Problem Size: 15
Difficulty: Hard
Current Seed: 820
Randomize Seed
Generate Scenario
Algorithm: Fancy
Time Limit 600 seconds
Solve TSP
Solutions: 0
Cost of tour: 10385
Solved in: 0.006584 seconds


Traveling Salesperson Problem
max queue size: None
total states: None
pruned states: None
Problem Size: 45
Difficulty: Hard
Current Seed: 820
Randomize Seed
Generate Scenario
Algorithm: Fancy
Time Limit 600 seconds
Solve TSP
Solutions: 0
Cost of tour: 19171
Solved in: 0.080249 seconds

5 FUTURE WORK

If we were to consider continuing research with our algorithm, we would focus on optimization in the following areas:

- **4.Perimeter Determination.** Our current method of finding a working perimeter is limited, as we implemented a method that relies on edges existing between all nodes. In the Hard and Hard (Deterministic) modes, this is not the case. Therefore, we were sometimes forced to delete cities from the originally determined perimeter when the perimeter included non-existent edges. Our perimeter determination also didn't take into account 3D Space in elevation, which is also a factor in Medium, Hard and Hard (Deterministic) modes. If we were to optimize our algorithm further, we would implement a way to replace instead of delete perimeter cities with nonexistent edges, and take into account elevation difference between perimeter cities.

- **Insertion.** Although we are proud of our current method of insertion, we also believe that there is significant room for optimization. If we were to spend more time optimizing our algorithm, we would focus on faster ways to parse through cities that are not yet included in the path, like potentially sectioning the map into quadrants and only parsing through cities in each relevant quadrant. We believe that some sort of method could be implemented to allow our insertions to be faster.

- **Post-Insertion Optimization.** We believe that although the initial perimeter can provide a great framework for insertions, it would possibly be beneficial to check the order in the final path against neighboring cities. For example, after the final path is calculated, we could follow the path taking cities in sets of 3 or 4 and randomly change the order to see if it is decreased in any other order. If no decrease is found, the path would be left alone, but if a decrease is found it would be applied to the final path. This would only add an order of O(n), and may offer valuable increases in distance optimization.

Github Repo with our code

References:

1. **Weru, L. (2019, December 28).** ***11 Animated Algorithms for the Traveling Salesman Problem*****. STEM Lounge.**