A Clustering Approach to Approximate Solutions for TSP

Benjamin Song
Benjamin1225bs@gmail.com

Gregory Knapp
Gregory.c.knapp@gmail.com

Ethan Low
Ethanthelow@gmail.com

Megan Dahl
Megan@dahlmart.com

9 December 2021
CS 312-003
Professor David R. Grimsman
BYU, Computer Science

Abstract

The Traveling Salesman Problem is a frequently studied problem in the field of computer science. The authors suggest a clustering approach to solving the problem to overcome the time efficiency problems in the traditional Branch and Bound Algorithm. Results show that the algorithm succeeds in producing fast and reasonable solutions when the problem sizes are smaller (city count < 300), but falls increasingly short to the greedy algorithm when problems sizes are larger.

1 Introduction

The Traveling Salesman Problem is a challenge to find the most cost-efficient path to travel all of given cities. Although the problem has a rather simple nature, it has grabbed the attention of many scientists because of the difficulty of finding an algorithm that would produce the best solution. Although the Branch and Bound Algorithm is one of the more widely known algorithms that attempt to solve the problem, the algorithm suffers from a lack of time efficiency. To overcome this shortcoming, the authors suggest an algorithm involving K-means clustering, a machine learning method, that clusters cities into smaller groups, and a variation of the Branch and Bound algorithm which is applied to connect the clusters and the cities in each cluster (Sharma 2018).

2 The Greedy Approach

2.1 Implementation of the Algorithm

The greedy algorithm builds a path by first selecting a starting city. The next city added to the path is the city with the cheapest edge from the last city added to the path. The algorithm repeats this process until a path containing each city is make or until a city that has no outgoing edges is reached. If after creating a solution and the path does not contain all cities, then the solution is rejected. If a complete path is created, it then checks if there is an edge coming from the last city in the path going to the first city in the path. If there is not an edge the path is rejected. The algorithm runs this process with each city are its starting city. It compares the cost of each valid path made to the best path it has created. If the path it is comparing to the best path so far has a lower cost, then the best path is replaced with the that path, otherwise the path is rejected.

2.2 Big-O Complexity Analysis

The Greedy algorithm implements the following non-trivial functions for its execution:

- Node.get_greedy_path() **-** It loops through a row of the cost matrix to determine which city to travel to next, O(n)
- Node.test() - Determines if the length of the path of a Node is equal to the number of cities, O(1)
- Node.GenerateCostMatrix() **-** Generates a n * n (n is the number of cities) sized cost matrix, O(n^2)

The Node.get_greedy_path() algorithm is called n-times which is equivalent to the number of cities it has to visit in order to be a valid solution. This process is repeated n-times for each starting city. The Big-O Time Complexity of this algorithm reaches O(n^3).

3 The Authors TSP Approach

3.1 Origin of the Algorithm

The concept of this algorithm originates from the idea of building a convex hull using a divide and conquer algorithm (Guttage 2017). In the convex hull problem, large problems are broken into smaller subproblems which are solved and combined with other subproblems to produce the solution to the main problem. Similarly, the authors used an approach that would divide the TSP problem into subproblems which would combine with other subproblems to become the solution of the initial problem.

In order to achieve this notion, the algorithm implements two major algorithms: K-means, an unsupervised clustering method, and the branch and bound algorithm (Sharma 2018). Contrary to a traditional divide and conquer algorithm, the problem set is only divided once into a set number of subproblems or clusters with the exception of special occasions where the cluster becomes too big. The authors implemented a K-means clustering method to efficiently assign each city to a K-number of clusters. This way, it is expected that the edges that connect the cities of the same cluster would cost less than an edge from a city to a city of a different cluster. The algorithm also utilizes a variation of the branch and bound algorithm to connect the clusters and the cities within the clusters. The approach differs from a more traditional branch and bound algorithm in that only the top three children of a node are inserted into the priority queue and the search terminates once a full path is found, instead of when the queue is depleted.

3.2 Implementation of the Algorithm

The algorithm can be divided into the following three steps: 1) Assigning each city to a cluster, 2) Building the cluster connection and cost matrix, and 3) Applying the Branch and Bound Algorithm to connect the clusters and the cities within each cluster.

The algorithm initiates with assigning each of the cities to a cluster. Each of the clusters contains a randomly generated centroid value which is used to determine which centroid each of the cities is closest to. The city then would be assigned to a cluster that contains the centroid closest in distance. Depending on whether a cluster contains more than 20 cities or less than 5 cities, the cluster is either split into smaller clusters or dissolved with its cities absorbed to neighboring clusters.

Upon clustering each city, the algorithm builds a cluster cost matrix and a cluster connection matrix. The cluster cost matrix is used in the branch and bound algorithm when establishing the order of the clusters. The cluster cost matrix contains the average cost to travel from a city in a cluster to another city in a different cluster while the cells in the cluster connection matrix contain a priority queue of all the costs from one city of a cluster to a city in a different cluster. The cluster connection matrix is used when selecting the edges connecting the clusters once the branch and bound algorithm has found the ordering of the clusters.

Finally, the branch and bound function is called on each cluster in the cluster route to connect the cities within each cluster. All cities are connected and yield a full solution after this last step.

3.3 Big-O Complexity Analysis

In the following analysis n represents the number of cities in a TSP problem. This algorithm implements the following non-trivial functions in its execution:

- Kmeans() – It takes as input the list of n cities. It outputs a list between n/20 to n/5 clusters of unique cities. This function does this by running K-means algorithm to cluster close cities together with a maximum cluster size of 20 cities or a minimum of 5 cities. K-means has a time complexity of $O(n^2)$.
- find_children_connections() – It takes as input the list of clusters created by Kmeans(). It creates a cluster cost matrix and a cluster connection matrix. Both matrices have size $O(n^2)$. The values in these matrices are assigned by looping over each cluster and computing the average cost to every other cluster. This process has an $O(n^2)$ time complexity.
- branch_and_bound_algorithm() – Branch and bound is run on the clusters acting as nodes. In general branch and bound has a complexity of $O(n^2 * b^n)$. The complexity of this implementation is $O(n^2 * 3^{(n/10)})$. The branching factor equals 3 because the number of child-states each state can add to the priority queue is limited to the three with the lowers lower bound. The exponent is n/10 because throughout testing the number of clusters made average to 1/10th of the size of n.
- select_connection_tuple() – It takes as input the cluster route decided by the branch and bound function, and the cluster connection matrix. It outputs the list of edges used to connect the clusters in the order specified in the cluster route. This is accomplished by iterating over each cluster in the cluster route and selecting the lowest cost edge to the next cluster that does not contain any cities in any edge already selected. The edges are chosen from a priority queue of edges stored in the cluster connection matrix. The time complexity to select an edge is O(1) because the queue has a max size of 20^2 due to each cluster having a max size

of 20 cities. The total time complexity of this process is O(n).

The above functions are each executed once before the algorithm enters a for loop iterating over each cluster. In this for loop, branch and bound is run on each cluster to determine the path in the cluster, which will complete the full path. Because each cluster has a maximum size of 20 cities, the complexity of running branch and bound on each cluster is O(1). This is because 20^2*3^{20} is a constant. Thus, the time complexity of the for loop is O(n).

The total worst case time complexity of the full algorithm can be broken into $O(n^2) + O(n^2) + O(n^2 * 3^{(n/10)}) + O(n) + O(n) = O(n^2 * 3^{(n/10)})$. The space complexity also matches the time complexity because branch and bound has the same time complexity and space complexity. Since the largest component in the time complexity comes from the branch and bound connecting the clusters, the space complexity is also $O(n^2 * 3^{(n/10)})$.

3.4 Pros and Cons

Pros

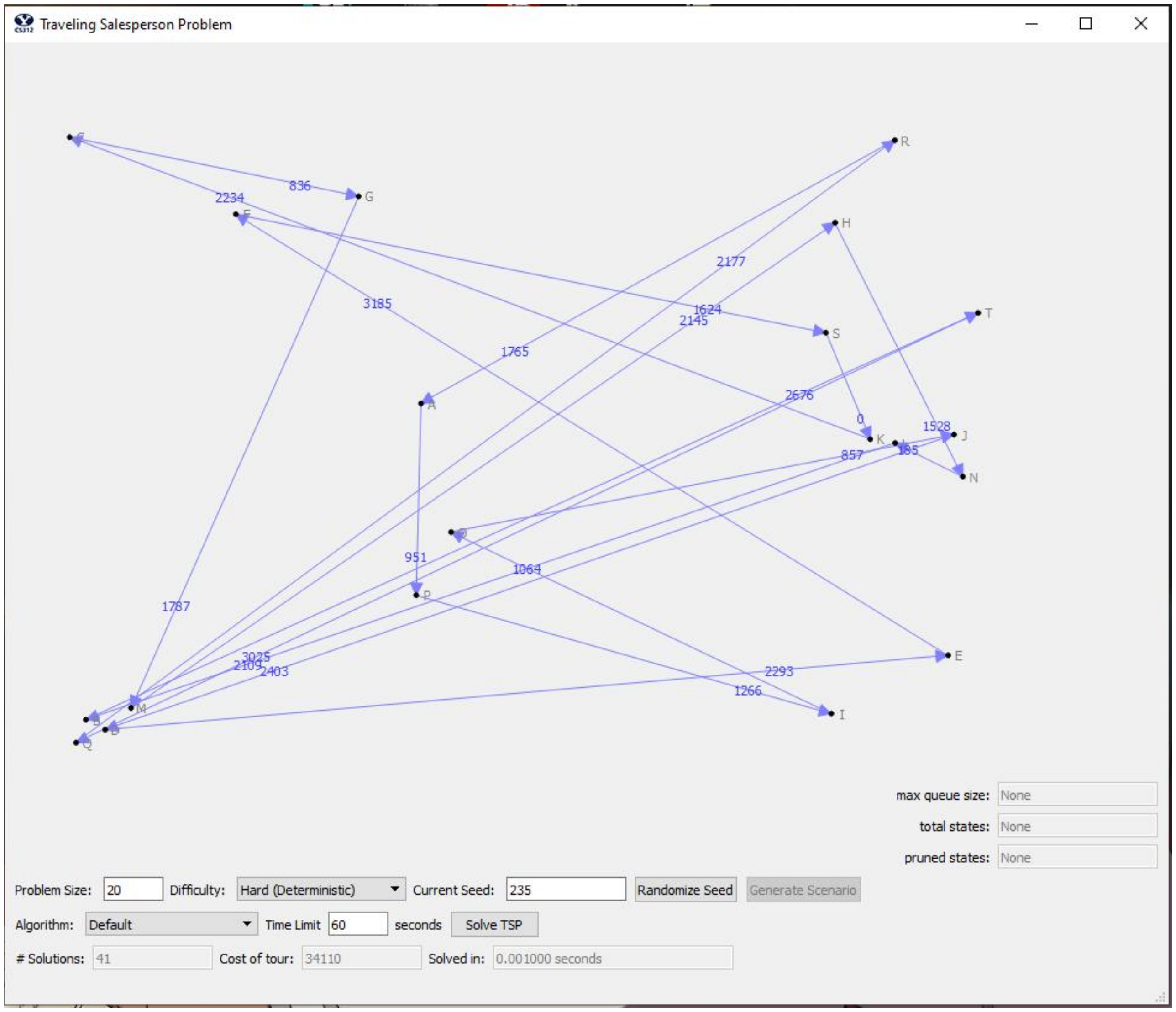
- The algorithm yields an improved solution compared to the greedy approach for problems involving more than 300 cities.
- In empirical testing the algorithm runs considerably faster than the greedy algorithm.

Cons

- It falls short in finding the optimal solution in smaller problem sets.
- Depending on where the centroids are placed in the map, the clustering of the cities vary, resulting in occasional less optimal solutions.

4 Typical Screenshots of Each Algorithm

Default Random Tour:



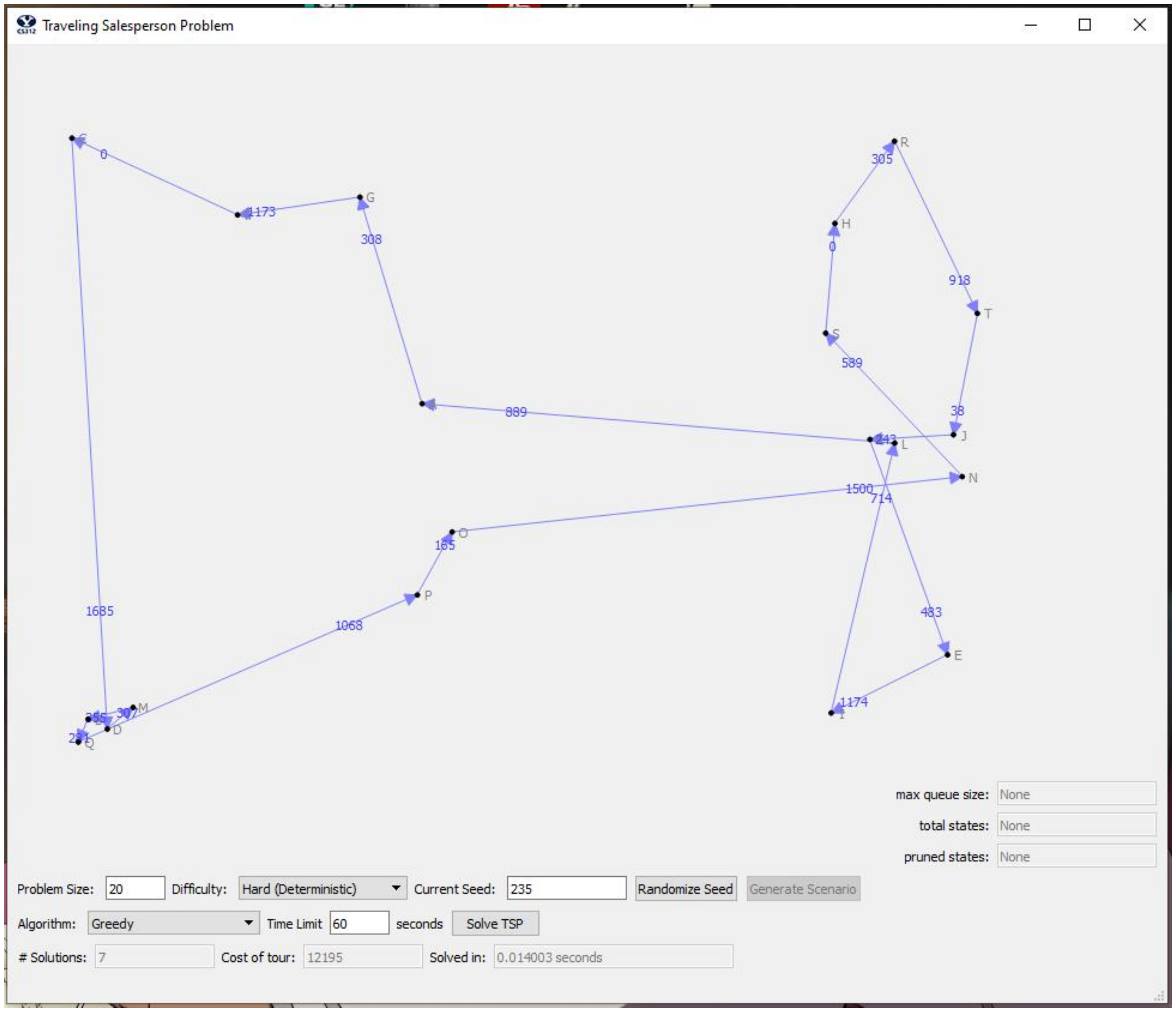
max queue size: None
total states: None
pruned states: None

Problem Size: 20 Difficulty: Hard (Deterministic) Current Seed: 235

Algorithm: Default Time Limit 60 seconds

Solutions: 41 Cost of tour: 34110 Solved in: 0.001000 seconds

Greedy Algorithm:



max queue size: None

total states: None

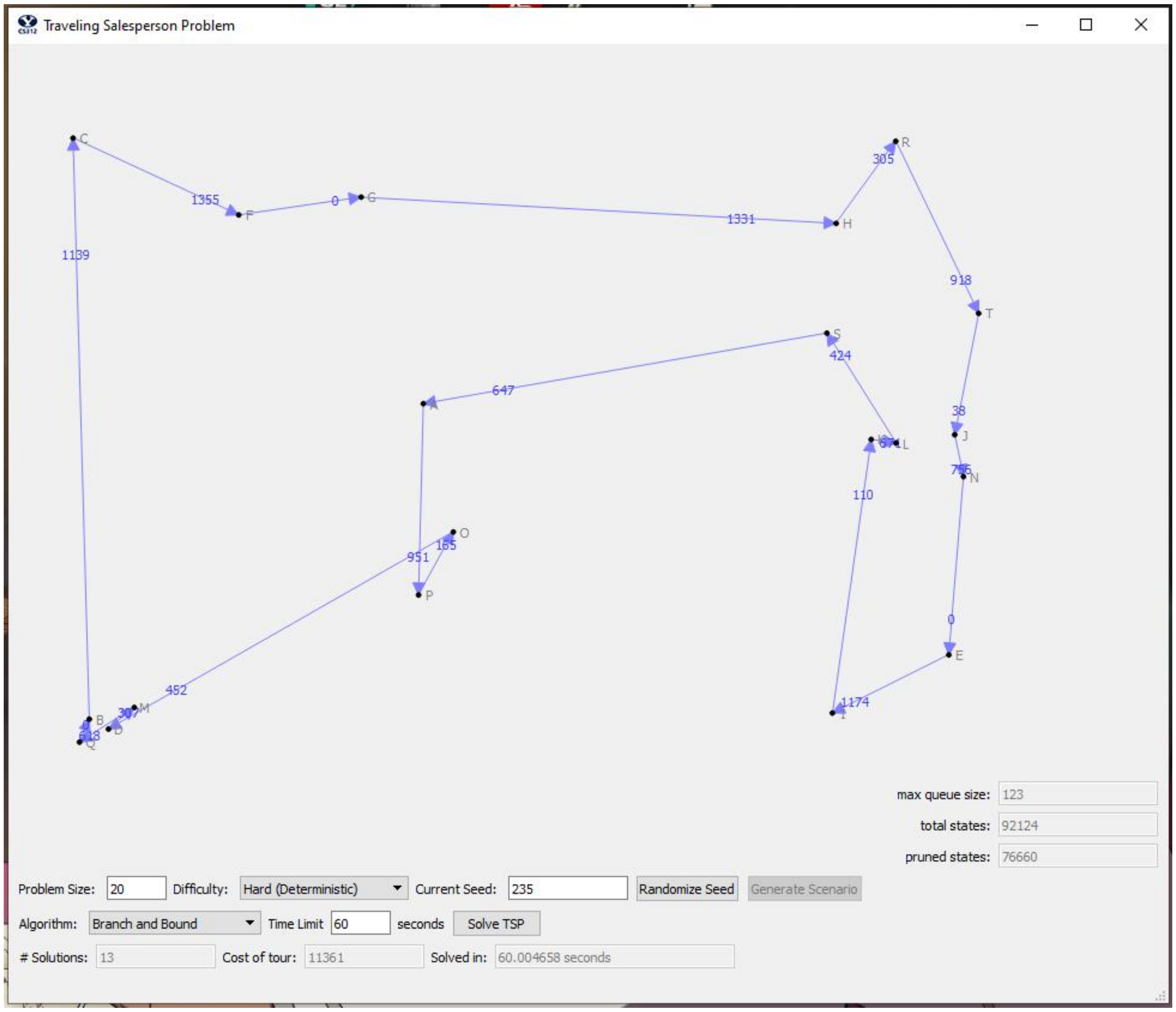
pruned states: None

Problem Size: 20 Difficulty: Hard (Deterministic) Current Seed: 235 Randomize Seed Generate Scenario

Algorithm: Greedy Time Limit 60 seconds Solve TSP

Solutions: 7 Cost of tour: 12195 Solved in: 0.014003 seconds

Branch and Bound Algorithm:



max queue size: 123

total states: 92124

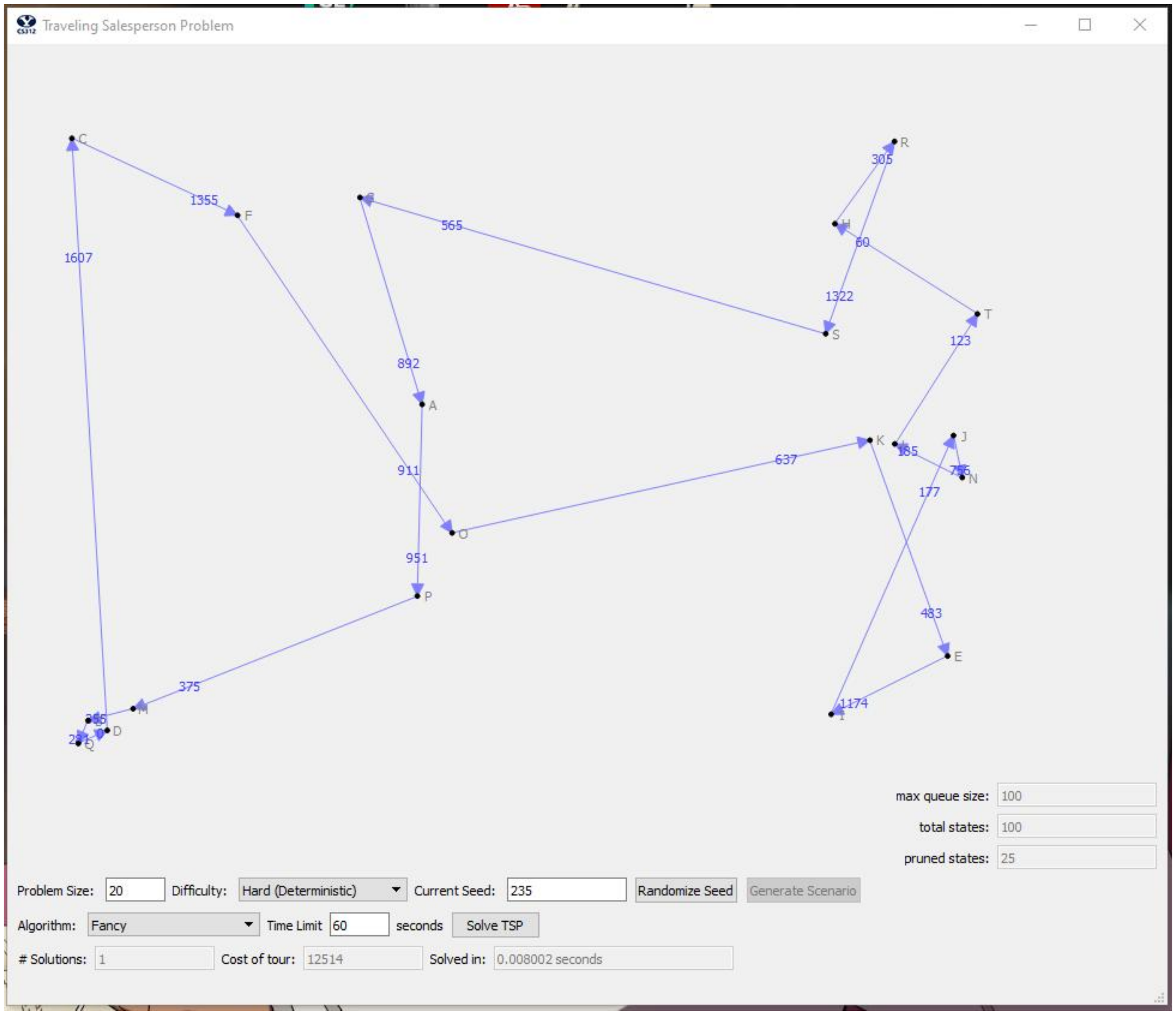
pruned states: 76660

Problem Size: 20 Difficulty: Hard (Deterministic) Current Seed: 235

Algorithm: Branch and Bound Time Limit 60 seconds

Solutions: 13 Cost of tour: 11361 Solved in: 60.004658 seconds

Clustering (Fancy) Algorithm:



max queue size: 100
total states: 100
pruned states: 25

Problem Size: 20 Difficulty: Hard (Deterministic) Current Seed: 235

Algorithm: Fancy Time Limit 60 seconds

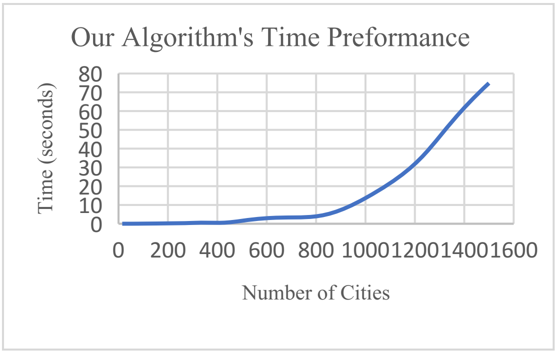
Solutions: 1 Cost of tour: 12514 Solved in: 0.008002 seconds

5 Empirical Results Table of TSP Algorithms

	Random		**Greedy**		**Branch and Bound**			**Own Algorithm**		
# Cities	**Time (sec)**	**Path Length**	**Path Length**	**% of Random**	**Time (sec)**	**Path Length**	**% of Greedy**	**Time (sec)**	**Path Length**	**% of Greedy**
15	0.00022	23027	11081	.48	51.524961	10534	.95	0.003768	11369	1.03
30	0.04	42092	17165	.41	60	15395	.90	0.014	16237	.95
60	31.90	78683	25357	.32	60	23625	.93	0.02	24134	.95
100	TB	TB	35459	N/A	TB	TB	TB	0.07	33885	.95
200	TB	TB	55290	N/A	TB	TB	TB	0.21	53874	.97
20	0.011	29909	12596	.42	177.64	10721	.851143	0.0076	12371	.98
300	TB	TB	72924	N/A	TB	TB	TB	0.45	73050	1.0017
500	TB	TB	101321	N/A	TB	TB	TB	1.58	109792	1.084
1000	TB	TB	157918	N/A	TB	TB	TB	13.66	179368	1.14
1500	TB	TB	201504	N/A	TB	TB	TB	74.87	245761	1.22
3000	TB	TB	322315	N/A	TB	TB	TB	TB	TB	TB

5.1 Discussion and Analysis of Results

In the above table each algorithm was run with a time limit of 10 minutes. If the algorithm could not find a path within that time the problem size was considered too big and marked with TB in the table. The data from the table illustrates the exponential growth rate of our algorithms time requirement.



It is clear from graphing the data that the growth rate of the algorithm is exponential, which matches the expected trend calculated in the theoretical Big-O analysis. Even though the growth rate is exponential like branch and bound, the algorithm can handle a larger number of cities because of its clustering method, which is reflected in the exponent being n/10 in the time complexity.

In comparison to the greedy algorithm, our algorithm found higher cost paths for problem sizes of less than to 15 cities and more than 300 cities. However, between 15 and 300 cities our algorithm slightly out preformed the greedy algorithm by an average of 96%. For smaller problem sizes our algorithm most likely preformed worse then the greedy because it is not optimized for those situations. The maximum number of clusters that the algorithm can make for 15 cities is three, but it is more likely that it only made one. This meant that the algorithm couldn't make use of its clustering. For 300 cities or more, the algorithm most likely runs into issues with how the branch and bound implementation selects the first complete path it finds. As the number of cities increase the number of clusters created also increase. So, as the number of clusters increase the likelihood that the branch and bound will return a less than optimal result also increases.

5.2 Future Work

The next step to improving this algorithm is to increase the number of levels of clustering for larger problem sizes. This would entail adapting the algorithm to cluster clusters whenever the number of formed clusters gets too high. This would allow the runtime to stay low while handling an increasing number of cities. Implementing this change to the algorithm should improve the run time exponentially for larger problem sizes.

References

Guttag, J. (2017, May 19). *Introduction to Computational Thinking and Data Science: Clustering* [Video file] Retrieved from https://youtu.be/esmzYhuFnds

Sharma, P. (2018, July 3). *Chan's algorithm to find convex hull*. OpenGenus IQ: Computing Expertise & Legacy. Retrieved December 2, 2021, from https://iq.opengenus.org/chans-algorithm-convex-hull/.