Held-Karp Algorithm to Approximate Solutions to the Traveling Salesperson Problem

Drake Foltz, Kailean O'Keefe, Tsung-Chiang Johnny Huang, Carson Hinckley

Received 16 April 2019
CS 312-001
Professor Ryan Farrell
BYU, Computer Science

Abstract

The Traveling Salesperson Problem has long been one of the most visible NP-Complete problems in the fields of computer science and mathematics. What follows is an examination and implementation of a Held-Karp algorithm for quickly approximating solutions to this problem. This dynamic programming algorithm will be compared to a greedy approximation, random path, and a branch-and-bound algorithm. Our intention is to show that the dynamic Held-Karp algorithm's relative optimization in terms of accuracy and time/space complexity.

1. Introduction

The Traveling Salesperson Problem (TSP) is especially confounding to scientists, because of the cognitive gap between understanding the problem and being able to efficiently find the true solution. Included in this report will be a explanation of the greedy algorithm for approximating a solution to the traveling salesperson problem, and a further exploration into the Held-Karp dynamic programming algorithm which attempts a more accurate approximation. We will compare the time and space complexity of our implementations of these two algorithms alongside our previous implementation of the Branch and Bound algorithm. We will afterwards provide empirical evidence supporting our comparison.

2. Greedy algorithm

In order to properly run our greedy algorithm, we first obtain the result from the provided random tour algorithm. We use this result as an upper bound on updating our current, lowest cost result. Then, for each city in the graph and in the time allotted, we build a path, always utilizing the edge which requires the least cost. The only other requirement for building the path is that it makes a Hamiltonian circuit, or that no cities are repeated in the path except the city used as the start and end position. Once the path is completely calculated, we then update our best solution so far if the new path has a lower cost than the best solution. After all this has run for all cities and within the time allotted, then we return our best solution so far. This function uses two nested loops, each bounded by the number of cities, and then a function call inside the inner loop which also runs through all cities. These all result in a complexity of $O(n^3)$, with n being the number of cities.

3. Held-Karp

Our implementation of the Held-Karp algorithm starts in much the same way as our greedy algorithm, except in this case we initialize the best solution so far to the answer provided by the greedy algorithm. Once that estimate is in place, we use a function to create an adjacency matrix from the cities. With these preparation steps complete, we were ready to implement the body of the Held-Karp algorithm.

At first, we had a hard time wrapping our heads around how Held-Karp was supposed to work. We did some research through a few quick Internet searches [1]. After much discussion and experimentation, we came to our current implementation.

We used the framework of "layers" and "subproblems" to understand and store our intermediate results [2]. We define a subproblem as a collection of a "cost" or length of a path, and the path itself. Layers are used to group series of subproblems together in a way to make the intermediate answers accessible for later calculations. Entries in a layer are accessed by providing a layer index as well as a tuple describing the index of the city being considered and a set of visited cities.

3.1 The 0th layer

We populated our 0^{th} layer by selecting an arbitrary starting city and storing subproblems relating the cost and route of all the connections coming from that city. In this case the set used to store the result in the layer is empty.

3.2 The 1st layer

The 1^{st} layer is populated by iterating over all of the cities and as we do so we retrieve each entry in the 0^{th} layer. If the entry already includes the city being considered, then that entry is skipped. Otherwise, a new subproblem is added to the 1^{st} layer. This subproblem consists of the path from the city under consideration to the previous entry's city, as well as the added cost of the new connection. This new subproblem is accessed by the tuple of the considered city and the union of the previous entry's set and the previous entry's city.

3.3 The remaining layers

The Held-Karp algorithm uses as many layers as there are cities, so at this point 2 of those layers have been populated. For each of the remaining layers, each city is considered in combination with the all of the entries on the previous layer. If the city has already been referenced in that entry, then that combination is skipped.

Assuming the city-entry combination is not skipped, then a tuple is made of the city under consideration and the union of the entry's city and set, in much the same manner as the 1^{st} layer. This tuple is then stored in a temporary array. Right after constructing the tuple, the same city-entry combination is used to create a new subproblem. This subproblem uses the cost and route of the entry's subproblem, and then is stored in a separate temporary array. Then both temporary arrays are iterated over in parallel in order to make entries with the lowest possible cost. It is these entries that are stored in the current layer.

3.4 Full path

Once all of the layers have been created, we iterate over the last layer and "manually" add the start city and the cost to get there. We store these subproblems in a separate list, which could be perceived as one last layer. Then the list is processed to find the subproblem that contains the Hamiltonian circuit of the least cost, and that is considered our answer. After converting indices to cities in order to create a dictionary of results, we return those results and our implementation of the Held-Karp algorithm is complete.

3.5 Complexity analysis

In our implementation of each of our algorithms, we need to return an answer within some specified amount of time, usually 60 seconds. Competing with this time limit is how long it takes the program to process the provided data. Just as in the greedy algorithm, the input for the Held-Karp algorithm is some number of cities.

Based on how our implementation iterates over the cities, we evaluate our function to run in $O(n^2 2^n)$ time, with the step described in 3.3 using the most time. Our algorithm also uses a lot of space, resulting in a $O(2^n n)$ space complexity. This is again because of step 3.3, where we create as many subproblems as there are combinations, each with has a path whose length is approaching n. While we trim these combinations in order to make the layer, that space may still be in use.

4. Empirical analysis

In our analysis, we compare the results across four algorithms (Random, Greedy, Branch and Bound, Held-Karp) for an increasing number of cities. It should be noted that, because we use answers from the other algorithms as initial solutions for some algorithms, this data may have some interesting interconnectedness. As the table on the following page shows, the complexity of the TSP problem led to some of the larger problems being left unsolved. The implementation of the Random algorithm was able to calculate approximate solutions until our 60-city test, but after that it was only Greedy that was able to give an approximation for 100 and 200 cities. The other algorithms, Branch and Bound and Held-Karp, were each able to give more accurate solutions for the smaller values but ultimately their complexity led to them taking too much time for the larger problem sets. The shocking fact of the results was the ability of the Greedy algorithm to return results that were not too far off from our more accurate results, but in a fraction of the time. For example, at 25 cities the Greedy algorithm returned a result that was 20.88% higher than the approximation from the Held-Karp Algorithm in $1/32{,}165^{th}$ the amount of time! It wasn't until our 200-city test that the speed of the Greedy algorithm seemed to show any signs of slowing.

The results fall in line with what we discovered in our complexity analysis outlined in section 2 (Greedy) and 3.5 (Held-Karp). Greedy has a time complexity of $O(n^3)$ which seems far from ideal, but it is dwarfed by the complexity of our Held-Karp algorithm at $O(n^2 2^n)$. The trade-off that comes into play is between speed and accuracy. Since the Greedy runs through the set of points and tests each as a start point and

makes locally ideal decisions, the set of solutions that is scanned is much smaller than the possibilities parsed by our Held-Karp algorithm. Thus, the Held-Karp is able to return more precise distances (closer to the ideal), but it takes substantially more time. As the city set expands, it would take much more computational power for the Held-Karp to even return a result in a reasonable amount of time. Thus, for most cases with large city sets it is understood that one would have to resort to using the Greedy result and recognize that the solution will be about (from our testing) 10-25% smaller, but the path unknown.

5. Future work

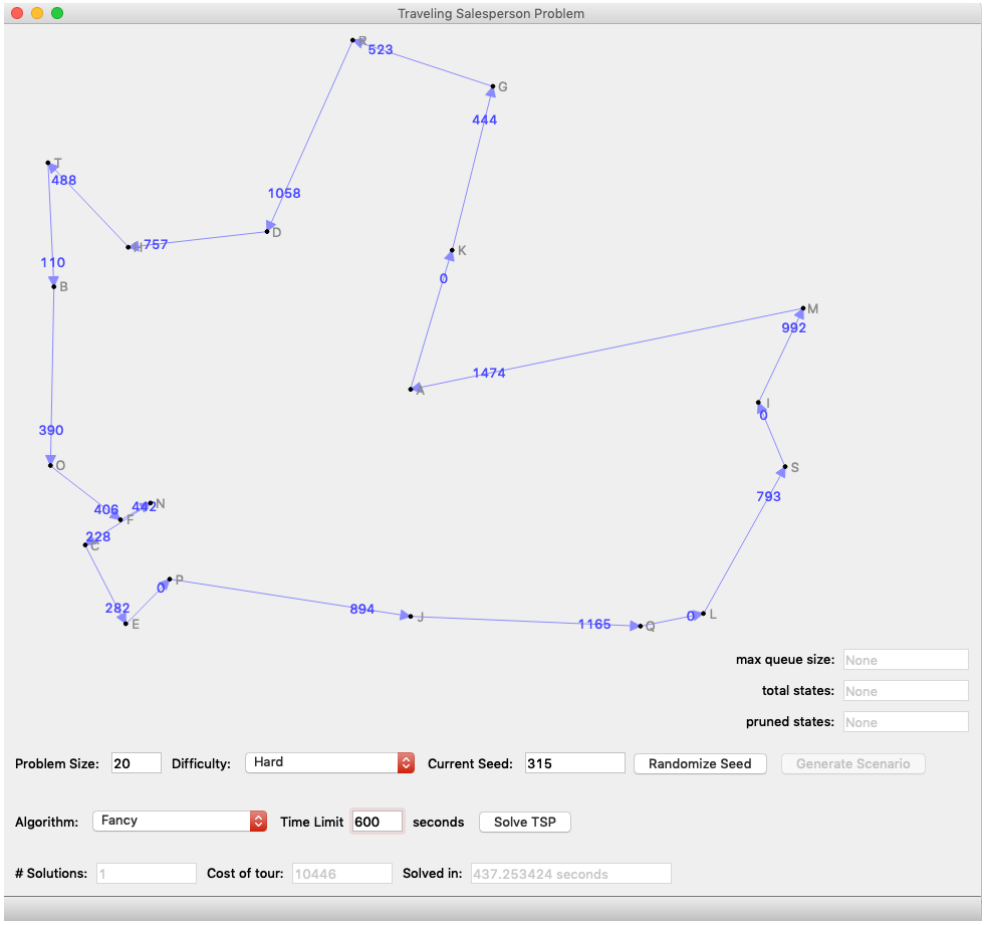
During the course of our testing, we made sure to run our algorithms on a variety of machines with differing levels of performance to accurately judge how things would turn out. Due to the design of our algorithms, they are best suited to machines with high levels of RAM and high single-core clock speeds. A few times we ran into issues with not enough RAM when we got to our larger city sets. Those two limitations pointed us to parallelization and better memory management as the areas that would allow us to see additional gains in our processor dependant calculations. Using systems like OpenMP or OpenGL we could identify portions of the code that can be parallelized or take advantage of a graphics card to perform the repetitious matrix operations. [3]

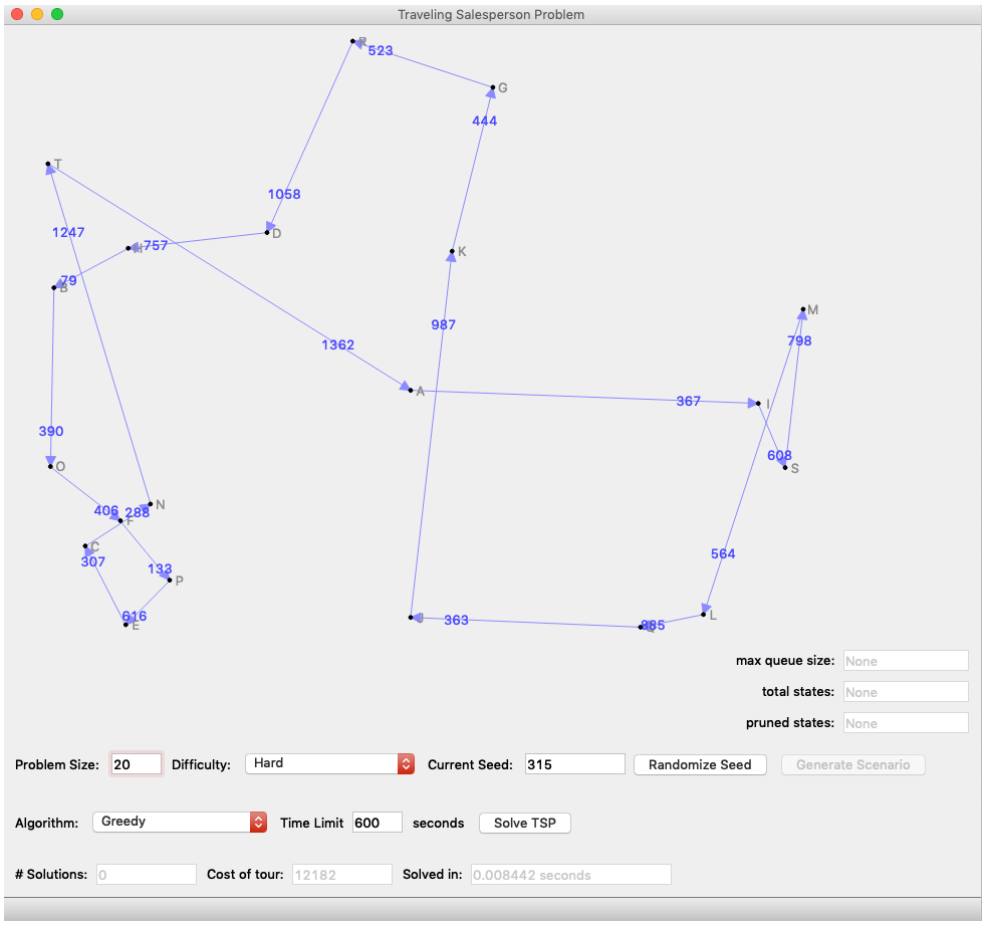
6. Conclusion

The strength of the Held-Karp algorithm lies in its ability to return accurate results. However, when compared to a well-designed Greedy algorithm, the difference in path length doesn't seem to justify the extra time and space required to fully run the algorithm. The additional complexity, O(n22n) of the Held-Karp means that for large data sets it takes far too long to return any results. We were able to only use it in problem sets that were the same size as the largest we were able to use for the Branch and Bound. The maximum difference we were able to see between those algorithms was 11.74% with the Held-Karp being lower, but again the issue of time comes into play. It is difficult to recommend the use of the Held-Karp algorithm given the results that we found. Possibly the additional future work we do will allow us to create a Held-Karp algorithm that holds up, but until then those hoping to come to a quick and accurate approximation of the Traveling Salesperson Problem should look elsewhere.

	Greedy		**Random**		**Branch and Bound**			**Held-Karp Algorithm**		
# **Cities**	Time (sec)	Path Length	Path Length	% Improve	Time (sec)	Path Length	% Improve	Time (sec)	Path Length	% Improve
10	0.0013	10123	13486	-33.22%	0.061	9109	10.02%	0.027	9109	10.02%
15	0.0018	11392	19419	-70.46%	42.33	10901	4.31%	10.07	9564	16.05%
20	0.008	13335	28409	-113.04%	203.53	11357	14.83%	351.65	10194	23.55%
25	0.016	16675	29408	-76.36%	470.21	13602	18.43%	514.65	13193	20.88%
30	0.02	17007	41341	-143.08%	TB	TB	-	TB	TB	-
60	0.15	26547	79839	-200.75%	TB	TB	-	TB	TB	-
100	0.92	34776	TB	-	TB	TB	-	TB	TB	-
200	17.91	55920	TB	-	TB	TB	-	TB	TB	-

7. Screenshots





References

[1] 'A dynamic programming approach to sequencing problems', Michael Held and Richard M. Karp, *Journal for the Society for Industrial and Applied Mathematics* 1:10. 1962

[2] GeeksforGeeks. "Travelling Salesman Problem | Set 1 (Naive and Dynamic Programming) | GeeksforGeeks." *YouTube*, YouTube, 5 July 2017, www.youtube.com/watch?v=hvDx7q6vcWM.

[3] amoffat. "Solving the TSP with WebGL and Gpgpu.js." *TSP with WebGL*, amoffat.github.io/held-karp-gpu-demo/.