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

Drake Foltz, Kailean O’Keefe, Tsung Huang, Carson Hinckley

Received 15 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(n³), 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. So we did some research through a few quick Internet searches [ INSERT REFERENCES ] . 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. 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 0th 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 1st layer is populated by iterating over all of the cities and as we do so we retrieve each entry in the 0th layer. If the entry already includes the city being considered, then that entry is skipped. Otherwise, a new subproblem is added to the 1st 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 1st 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 paralell 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 percieved 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 indecies 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(n²2ⁿ) time, with the step described in 3.3 using the most time. Our algorithm also uses a lot of space, resulting in a O(2ⁿn) 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 usea nswers from the other algrotihms as initial solutions for some algorithms, this data may have som interesting interconnectedness. As you can see in the graph………...

|  | Random | |  | |  | |  | | |  | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  | 2ⁿ |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |

Acknowledgements

Proin pharetra nonummy pede. Mauris et orci. Aenean nec lorem. In porttitor. Donec laoreet nonummy augue. Suspendisse dui purus, scelerisque at, vulputate vitae, pretium mattis, nunc. Mauris eget neque at sem venenatis eleifend. Ut nonummy.

References

1. Surname A, Surname B and Surname C 2015 *Journal Name* **37** 074203
2. Surname A and Surname B 2009 *Journal Name* **23** 544