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 be comparing the time and space complexity of these implementations of these two algorithms, as well as providing empirical evidence supporting

We will first discuss our implementation of a greedy algorithm which finds a temporary solution to the TSP problem. We use this solution, as well as the solution provided by a random tour, in establishing a baseline of comparison. After we discuss our implementation of the Held-Karp, we will present a comparison of results between these three algorithms as well as our previous implementation of the Branch and Bound algorithm. After this presentation we will give a short conclusion, followed by the resources that we have used.

2.1 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 as 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.

2.2 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.

2.2.1 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.

2.2.2 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.

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