# Implementing the Held-Karp Lower Bound Algorithm in Python

*(Final Report for CMSC 451 – Honors Option)*

*Fall 2015*

Taylor Moore

Advised by: Dr. David Mount

## 1 Introduction

The Traveling Salesman Problem is a formulation that arises in many diverse fields, from vehicle routing to genome sequencing to circuit design. Because it is an NP-Hard problem, it is primarily calculated via heuristics for graphs of non-trivial size. The Held Karp lower bound algorithm provides a lower bound for the cost of the optimal TSP tour of a graph. Having the lower bound for a particular graph is useful for checking the performance of a given heuristic. This report details an implementation of the Held Karp lower bound algorithm in Python using nearest neighbors, based on the work of Valenzuela and Jones.

## 2 Background

The Traveling Salesman Problem is to find a minimum-cost Hamiltonian cycle, given a set of points and edges, and a cost function on the edges. In its original form, the problem was to find the shortest tour of all of the state capitals in the US. Since then, countless variations have been put forward in order to solve related problems. Unfortunately, all variants of the problem are at least NP-Hard.

In the original form, the cost of the edge from capital a to capital b is equal to the Euclidean distance between them. The cost function then satisfies the triangle inequality, which states

$$c(u,v) \leq c(u,t) + c(t,v)$$

for all vertices t, u, v in V. Also, the distances are symmetric, that is

$$c(u,v) = c(v,u)$$

for pairs of distinct vertices u and v. TSP problems that satisfy these two properties are called Euclidean or geometric TSPs. Euclidean TSPs, though still NP-Hard are generally more tractable than problems that do not satisfy these constraints.

The Held Karp Bound is a lower bound for the cost of an optimal Traveling Salesman Tour, created by Michael Held and Richard Karp. Briefly, the Held Karp bound is found by assigning a weight to each vertex, which is factored into the cost of the edges incident on it. The vertex weights form a cost vector $\pi$.

For a pre-determined number of iterations, the algorithm manipulates those weights in $\pi$ using a gradient ascent method in order to generate gradually higher-cost minimum 1-trees whose cost and shape grows closer and closer to that of a true TSP tour. In essence, the algorithm attempts to force the spanning tree algorithm to produce vertices of degree exactly 2. This is accomplished by increasing the cost of vertices of degree $> 2$, which discourages the tree from using edges incident on those vertices, and decreasing the cost of vertices of degree 1 (to encourage use of those edges). The highest-cost tree found by a pre-determined stopping point is the bound.

The Held Karp bound algorithm works well for symmetric TSP problems which satisfy the triangle inequality. For large instances of these problems, calculating the Held Karp bound is a good way to check the reasonableness of an approximation result without resorting to brute-forcing the exact TSP tour.

## 3 Implementation

For my implementation I followed the work of Valenzuela and Jones, who detailed an algorithm for finding an approximation of the Held Karp lower bound. The original Held Karp bound algorithm requires working with complete graphs. Since the algorithm requires computing a minimum 1-tree and updating every edge in each iteration of the loop, this quickly becomes very computationally expensive. Using a constant number of nearest neighbors greatly improves the running time of the algorithm, while only sacrificing significant accuracy on graphs with certain difficult properties. I implemented the algorithm for finding the lower bound in Python, using the graph-tool and scipy packages. Both of these packages are based on C++ code, making them more efficient than pure Python code.

### 3.1 Generating neighborhoods

The Held Karp algorithm requires for each point p a subgraph

$$G = (V, E), \text{ where}$$

$$V = \{p\} \cup \{k \textit{ nearest neighbors of } p\}$$

$$E = \{(p, v) \mid \forall v \in V, v \neq p\}$$

I inserted each of the points into a kd-tree, and then used it to calculate distances between each of the points. The possibility of severe clustering meant that I needed to calculate distance between all pairs of points rather than the nearest 40 or 50.

The vanilla algorithm performs poorly for graphs with obvious clusters. In order to mitigate this, I used the technique specified by the Valenzuela and Jones. Instead of forming a neighborhood of the nearest neighbors, for each point the algorithm divides the plane into quadrants with the point as the origin. For each quadrant, select the nearest seven points in the quadrant, (or all points in the quadrant if fewer than seven) and add them to the neighborhood for a total of up to 28. Then fill the remaining space in the neighborhood with the nearest remaining points, without regard to quadrant, for a total of 40 neighbors.

This method produced a list of 40 neighbors for each vertex.

I created the subgraph by filtering out the vertices not in the neighborhood (except the vertex itself) and creating a directed edge from the vertex to each of its neighbors. Two directed edges (u,v) and (v,u) were used to describe each undirected edge for programming convenience.

## 3.2 Running loop

Next I implemented the substance of the algorithm. The details of the algorithm can be found in Valenzuela, 1997.

The most time-consuming step was the updating of the edge costs, since each edge in each subgraph needed to be updated. After the edge costs were updated, a minimum spanning tree was created using graph-tool, which relies on Kruskal's algorithm. Then it was simple to select the two smallest edges to form the 1-tree.

I updated the step size using the formula

$$t^m = (m-1)\left(\frac{2M-5}{2(M-1)}\right)t^1 - (m-2)t^1 + \frac{1}{2}\frac{(m-1)(m-2)}{(M-1)(M-2)}t^1$$

where

$$t^1 = \frac{1}{2n}L(e_{ij}, T)$$

$L(e_{ij}, T)$ is the minimum 1-tree generated by the edge weights.

I updated the vertex weights using the formula $\pi_i^{(m+1)} = \pi_i^m + t^m(d_i^m - 2), 1 \le i \le n$

## 4 Results

I tested the algorithm on examples from TSPLIB, a database containing various graphs, many of which were included for being in some way pathological or difficult to find approximate tours for. The following shows my implementation execution on symmetric TSP graphs lin318 and dsj1000.

Execution for graph dsj1000

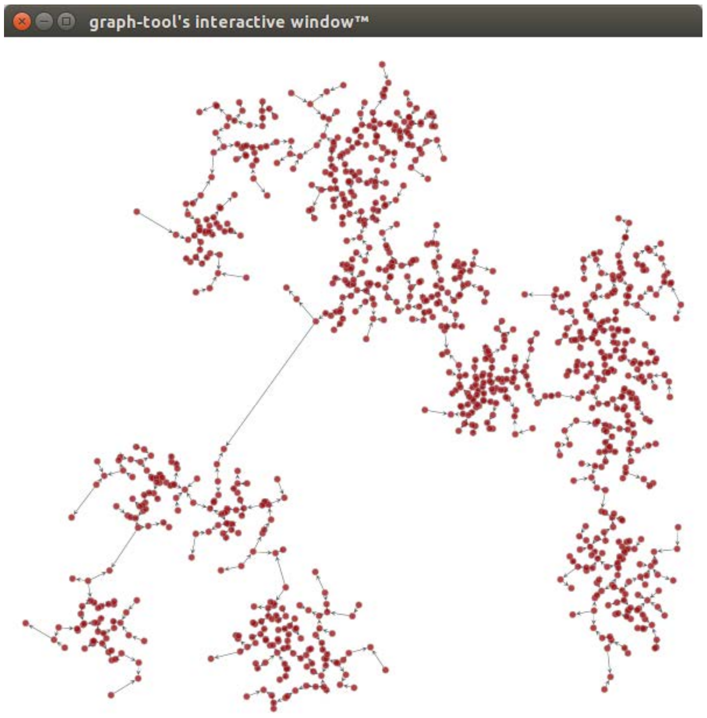


Fig 1. Dsj1000 initial spanning tree

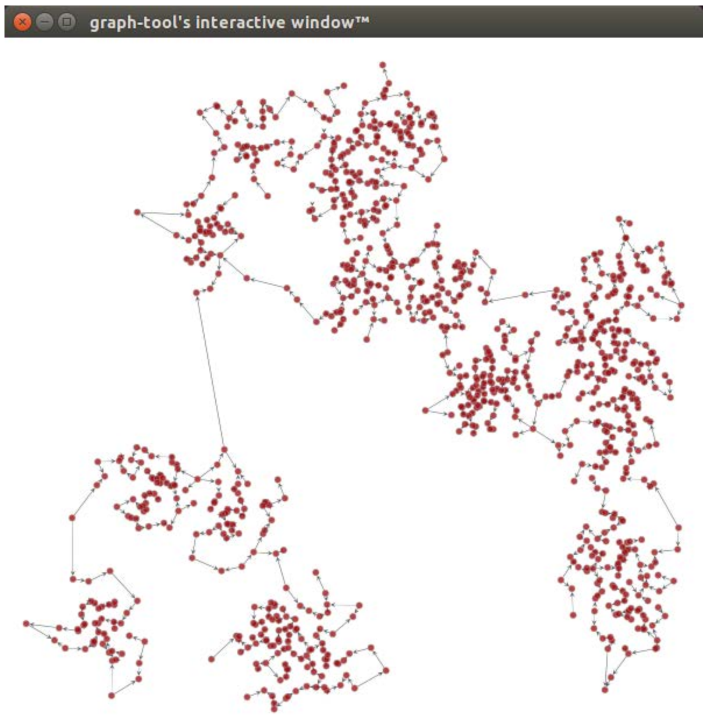


Fig 2. Best tree found by step 250

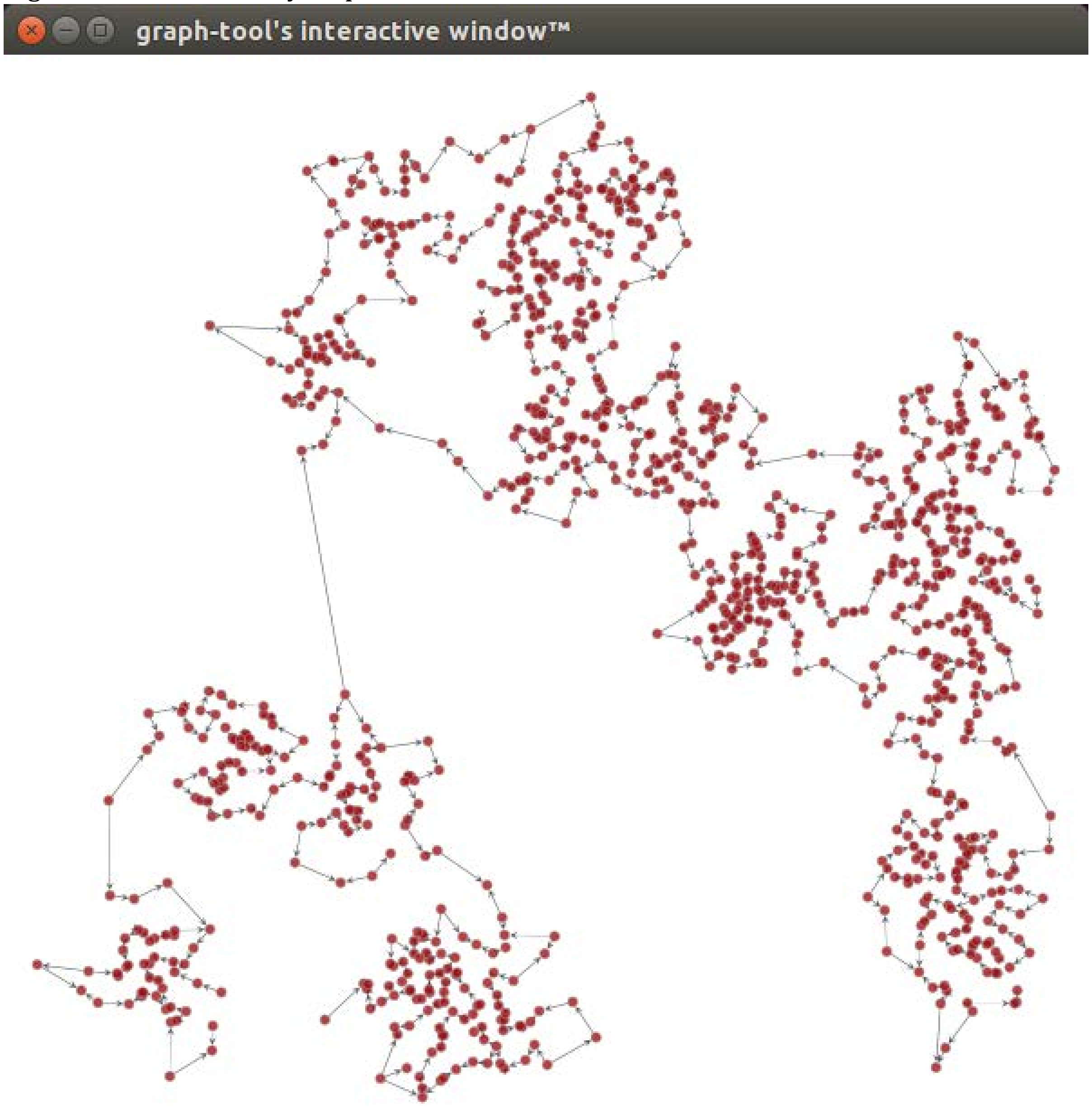
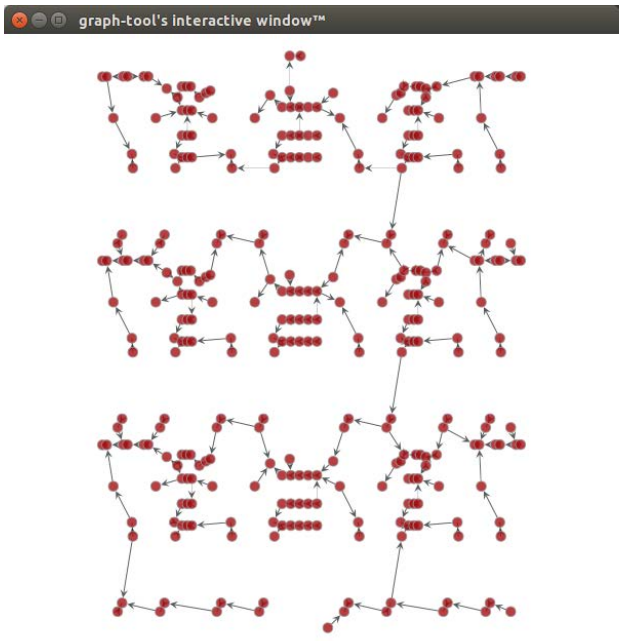


Fig3. Best tree found by step 500

Execution of algorithm for graph lin318



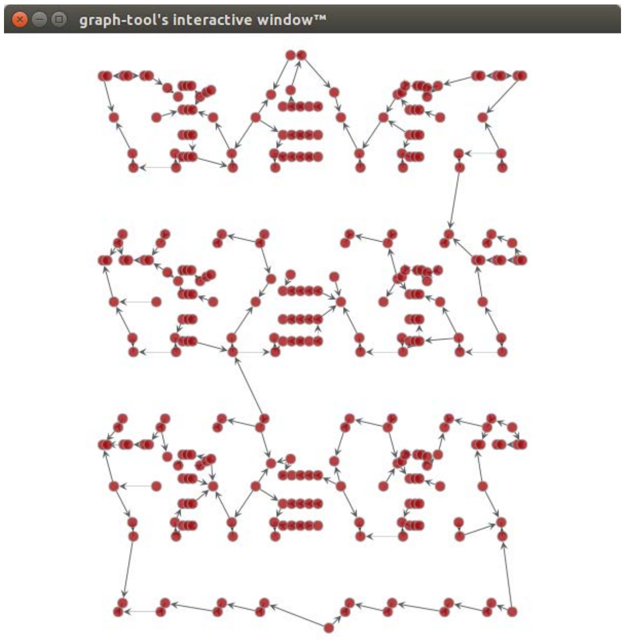


Fig. 1: Left - Initial 1-tree. Right: Best 1-tree found by step 250

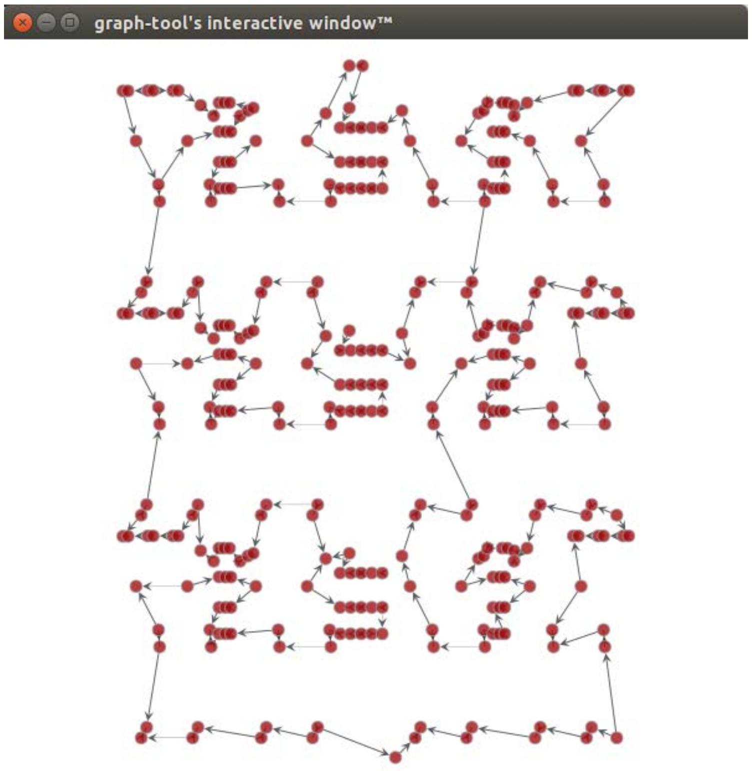


Fig. 2 Best 1-tree found by the end of the algorithm

With the modifications specified, the bounds that my implementation returned matched those of Valenzuela and Jones exactly.

The running time of the subgraph creation is dominated by the nearest neighbor query performed on the k-d tree. It amounts to sorting n lists, where each list is the distance of all points in the graph to a single point. Thus at worst it runs in $O(n^2 \log n)$ time.

The running time of each iteration of the loop is mainly contributed by the edge cost updating. Each edge in each subgraph must be updated, for a total of $80n$ updates. Finding the minimum 1-tree has time complexity $O(m \log m)$ for Kruskal's algorithm, which is $O(n \log n)$ in this case. The other update steps run in either $O(n)$ or $O(1)$ time. Thus, each iteration of the loop runs in $O(n \log n)$ time. For the relatively small graphs that I tested on, the execution time was dominated by the edge updating.

This total running time, while large, is far smaller than the running time of a dynamic programming ($O(2^n n^2)$) or branch and bound algorithm ($O(2^n)$) needed to find an exact answer.

## 5 Conclusion

The implementation presented here can be used to find an approximation of the Held Karp lower bound for Euclidean TSPs, including those involving clustered data. This is a faster solution than calculating the bound outright, however, the implementation still does not scale well.

There are two areas for performance improvement – initial subgraph creation and edge cost updating. Both aspects have some repetition in computation, and perhaps more efficient data structures could improve their running time.

## 6 Bibliography

Cook, William J. *In Pursuit of the Traveling Salesman*. Princeton, NJ: Princeton University Press, 2012. Print.

Jones, Eric, Travis Oliphant, Pearu Peterson, et al. SciPy: Open Source Scientific Tools for Python. 2001- http://www.scipy.org/

Peixoto, Tiago P., "The graph-tool python library", figshare. (2014) DOI: 10.6084/m9.figshare.1164194

Valenzuela, Christine L. and Antonia J. Jones. "Estimating the Held-Karp lower bound for the geometric TSP." *European Journal of Operational reseaerch* (1997): 157-175.

Wikipedia Contributors. *Travelling salesman problem*. 13 November 2015. 19 December 2015.