Improving a Brute-Force of the Traveling Salesman Problem

Grant Eaton

April 2017

1 Purpose

The purpose of this paper is to analyze the Traveling Salesman problem and its known solutions. More specifically, we will explain a naive solution to the problem and how to implement it, then analyze different ways to improve this solution, as well as go through how this algorithm works, along with proving it is an improvement.

2 Introduction

The Traveling Salesman Problem is described as such: "Find the shortest route (tour) for a salesman starting from a given city, visiting each specified group of cities, and then returning to the original point of departure." (Fulkerson & Dantzig & Johnson, 1954). Typically, a collection of cities will be represented as a directed graph, where each vertex is a city and an edge represents a path from one city to another. For the purposes of this paper, we will analyze only undirected graphs, which will make implementation easier, as well as simplify problems. The problem can be generalized as: given an input of an n x n matrix where Di_j represents the distance from city i to city j, find a permutation of cities that results in the shortest sum of D_{ij} between consecutive points. For example, for the given graph with n = 4 vertexes, 1, 2, 3 and 4, the D_{ij} for all vertexes of the graph is:

	1	2	3	4
1	0.0	1.0	1.0	1.414
2	1.0	0.0	1.414	1.0
3	1.0	1.414	0.0	1.0
4	1.414	1.0	1.0	0.0

Figure 1: Vertex Distances as Table

And our matrix input would be:

$$D'_{ij} = \begin{bmatrix} 0.0 & 1.0 & 1.0 & 1.414 \\ 1.0 & 0.0 & 1.414 & 1.0 \\ 1.0 & 1.414 & 0.0 & 1.0 \\ 1.414 & 1.0 & 1.0 & 0.0 \end{bmatrix}$$

Figure 2: Vertex Distances as Matrix

Our output to the problem is a matrix of 0s and 1s, where a 0 represents that there is no edge between the vertexes and a 1 means that there is a vertex. For example, the correct output to the graph in Table 1 would be:

$$X'_{ij} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Figure 3: Optimal Output to Table 1

And the optimal path graphed would look like:

Figure 4: An Example Graph of Figure 2

The optimal distance to the problem can be represented as the minimum of the linear form

$$D(X) = \sum_{i=j=0}^{n-1} D_{ij} X_{ij}$$

For this example, D(X) = 4.

For the purposes of this paper, we will represent the output as a list

$$L = [v_1, v_2 \ldots v_k, v_1]$$

where each number represents a vertex, and each consecutive vertex represents an edge between those vertexes. For example, the graph in figure 4 would be represented as: $L = [1, 3, 4, 2, 1]$

3 Brute Force

In this section, we will discuss a naive solution to the Traveling Salesman Problem. If the goal of the problem is to find the permutation of cities, where the path through all of them is the shortest, then one solution would be to simply compute every permutation of all given cities, then keep the one with the shortest distance. The algorithm to do this is fairly trivial.
First, we set our *bestTour* to an initial route permutation and the and *bestScore* to the score associated with the Euclidean distance between the two vertexes. Then, we loop until there are no permutations left, constantly checking to see if the new tour has a more optimal path. If it does, we update the *bestScore* and *bestTour*. Finally, after all route permutations have been checked, we return our *bestScore* and *bestTour*.

Algorithm 1 Brute Force TSP

```
1: procedure BRUTEFORCETSP(CITYMATRIX[][])
2:     bestTour ← getTourPermutation(cityMatrix)
3:     bestScore ← getScore(tour)
4: while permutations are left
5:     tour ← getTourPermutation(cityMatrix)
6:     if getScore(tour) < bestScore then
7:         bestScore ← getScore(tour)
8:         bestTour ← T
9: return bestTour, bestScore
```

Since this algorithm has to generate $O(n!)$ different permutations, its run-time will be $O(n!)$. In practice, this is very slow, only allowing modern processors to generate solutions to up roughly 10 cities in under a few minutes.

For example, we are given the matrix to the following graph with the solution, $L = [1,3,4,6,9,8,7,10,5,2,1]$ shown as:

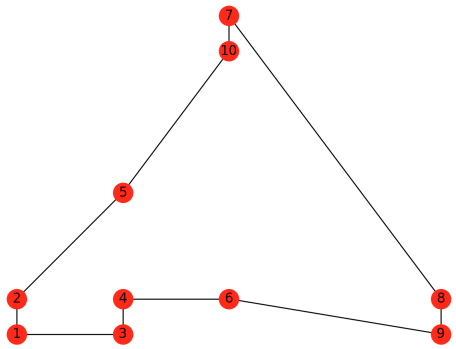


Figure 5: An Example Graph of Figure 2

Running on a 2013 Macbook Pro with a 2.6GHz i7 processor, we get a total run-time of 22.629 seconds. In these 22 seconds, the computer considered 10! = 3,628,800 route permutations. Adding extra nodes is increasing our run-time factorially.

4 Analyzing Potential Improvements

The Traveling Salesman Problem has seen many improvements beyond the brute force since it was first published in a paper. Some solutions are complete, finding the optimal path (Held-Karp algorithm), some find approximate solutions to the problem, giving a range with the accurate solution (Christofide's algorithm), and others use heuristics to find good, but sub-optimal solutions to the problem. Since the Traveling Salesman problem is an NP problem (it cannot be solved under a polynomial time), we know that there are currently no solutions that will give us the optimal solution without a polynomial run-time. In this section we will analyze the optimization algorithms, notice pros and cons, and finally, pick one to further analyze.

Christofides Algorithm

Christofides Algorithm is an approximation algorithm that guarantees a solution within 50 - 150% the true value. This means that it can be used to accurately predict a range of values the solution will be in (Christopfide, 1976). The algorithm works by using minimum spanning trees, using the handshaking lemma, creating a minimum-weight perfect matching subgraph, forming a Eulerian circuit, and finally getting the Hamiltonian circuit of the graph.The output is guaranteed to be 50% away from the actual optimal solution. The upside of this algorithm is its speedy run-time compared to brute force: $O(n^3)$.(Christofides, 1976) The downside of the algorithm is that its useless other than certain scenarios where you want to predict a range of values that the solution is within.

Nearest Neighbor Heuristic

The Nearest Neighbor heuristic is a simple, yet effective approach to getting a sub-optimal solution the Traveling Salesman Problem. The idea is to:

1. Select a random city, r
2. find the next, nearest city, then go there.
3. Repeat step 2 until there are no unvisited cities left.
4. Return to r

(Nilsson)

Since this algorithm essentially is a loop nested inside an identical loop, both iterating from 1 to n, we end up with an $O(n^2)$ solution. Unfortunately, though, our solution is not always going to be optimal. This is because we are relying on a heuristic (a good guess) for picking the next node. This leads us to only know that our solution is a decent one, but likely non-optimal.

Held-Karp Algorithm

The Held-Karp Algorithm is a dynamic programming solution to the problem. The idea is to take a top-down approach, generating all possible sub-sets to the problem, then saving them in a table. Each sub-set can be constructed based off previously solved subsets, saving time compared to the brute force. This results in an exact solution with a run-time of $O(2^n n^2)$ algorithm, which at first glance appears to be a very substantial improvement, however it requires $O(2^n n)$ space, which means our algorithm still can only go up to around 30 cities before maxing out 16gb of ram (Michael Held & R.M.K, 1962). Since this algorithm can be implemented relatively easily, it solves an exact solution, and we should be able to see a speed increase that is thousands of times faster.

5 The Held-Karp Algorithm

The Held-Karp algorithm works as such: let

let $S = [v_1, v_2, ..., v_n]$ be a subset of $1, 2, ..., n$, $l \in S$, where v_1 is our starting & ending vertex, and D_{l,v_1} represents the distance from l to v_1. For every l in S, we find the minimum cost path starting with v_1, going through all other vertexes exactly once, and ending at v_1. Let the cost of this path be $cost(l)$, so the cost of the corresponding cycle would be $cost(l) + D_{l,v_1}$. After computing the distance of all subsets, we return the minimum of all $cost(l) + D_{l,v_1}$ values.

Computing the $cost(l)$ is where the top-down approach of dynamic programming comes into play. We will create a map, call it C(S,l), which represents the minimum cost path, starting and ending at vertex l, going through each vertex in S once.

With this, we can now define the following recurrence relations

$$C(S,l) = \begin{cases} 0 & \text{if } length(S) \leq 1 \\ D_{lv_1} & \text{if } length(S) = 2 \\ min_{l \in s}[C(\{S-l\},\mathrm{l}) + \mathrm{D}_{l,j_S}] & \text{if } length(S) > 2 \end{cases}$$

To show this is correct, we will consider the following situation:

Suppose that for a given subset, starting at vertex v_1 and visiting every vertex in S and ending at v_1, vertex v_n precedes v_1. Assuming that all other cities were visited in optimal order, the cost for the subset would be $C(\{S-l\}, v_n) + D_{v_n,j_S}$.(Michael Held & R.M.K, 1962) Since we will accept the minimum cost over all other choices of v_n, we obtain this cost. Finally, if we let G denote the minimum cost of a complete tour, then

$$G = min_{l \in \{2,3,\ldots,n\}}[C(\{2,3,\ldots,n\}, l) + D_{v_n,j_S}]$$

(Michael Held & R.M.K, 1962)

For the following pseudo code, to keep a consistent naming convention of variables, we will represent C as $minCostMap$. For each path in $minCostMap$ there is a corresponding parent vertex, which we will store in another map, $parentMap$. Additionally, we will represent the distance from vertex i and j not as D_{ij} but instead as $distance(i, j)$.

Algorithm 2 Held-Karp

```
1:  procedure TSPHELDKARP(CITYMATRIX[][]
2:      minCostMap ← Map<Index,Integer>
3:      parentMap ← Map<Index,Integer>
4:      allSets ← getAllSets(CityMatrix)
5:      i ← patlen
6:  For set in allSets:
7:      For vertex = 1 → all Vertices in cityMatrix:
8:          if set contains vertex then continue
9:          index ← getIndex(vertex,set)
10:         minCost ← ∞
11:         For preVertex in set
12:             cost = distance(preVertex,vertex)
13:             + cost(startVertex → all vertexes → vertex)
14:             if cost < minCost then
15:                 minCost = cost
16:                 minPreVertex = preVertex
17:             if set.length is 0 then minCost = distance(0,vertex)
18:             minCostMap[index] ← minCost
19:             parentMap[index] ← minPreVertex
20:     set = [1,2,...,CityMatrix.length]
21:     min ← ∞, preVertex ← −1
22: For k in set:
23:     cost = distance(k,0)
24:     + cost(startVertex →all vertexes in set → k)
25:     ifcost < min then
26:         min = cost
27:         PreVertex = k
28: parentMap[index] ← preVertex
29: return min
```

Finally, we will prove by induction that the Held Karp algorithm is an improvement on the brute force by showing the algorithm has a run time of $O(n^2 2^n)$. At the same time, we will prove that the algorithm requires $O(n2^n)$ space.

We begin by analyzing the algorithm to notice that we must compute every subset in S in order to compute a solution.

Proof: Let $P(n)$ denote the claim that for any set with length n, we have 2^n number of subsets.
Basis: since an empty set has 1 subset, itself, then a set containing 0 elements has 2^0 subsets.
Induction: let k = n so that every k-element set has a total of 2^k subsets. Suppose that $P(k)$ is true. We must show that $P(k+1)$ is also true. $P(k+1)$ is the claim that: for any set with length $k+1$, we have 2^{k+1} subsets.

Let a be an element of S, with $k+1$ length, where $S' = S - a$. This means that S' must have k elements. We now will section S into two groups: I = subsets that have a and II = subsets that do not have a.

This means that $I = S'$ exactly; the subsets are identical. Since the length of A is k, we can apply the inductive hypothesis to see there must be 2^k elements in I.

The subsets of II are equal to $B' \cup a$ where B' is a subset of A'. By the inductive hypothesis, B' must have exactly 2^k sets. Since the total amount of sets in II+ total amount of sets in$I = 2^k + 2^k = 2^{k+1}$, and $II \cup I = S$, we know that we have 2^{k+1} elements in S.
Since S was just an arbitrary set, we have proved that any $(k+1)$-element set has 2^{k+1} subsets, proving $P(n)$ true. ■

Finally, since we know that the computation of the subsets gives us a run-time of 2^k, we can get our final run-time by noticing that for each permutation, we have a nested loop where we must run through each vertex inside of a loop where we must run through each vertex, giving us our $O(n^2)$. The $O(2^k)$ space comes from the fact we must store each permutation in memory.

6 Analysis of Brute Force vs Held-Karp

In this section we will compare brute force and the Held-Karp algorithm.

# Vertexes	2	4	6	8	10	12	14	16	18	20	22
H-K	$1.59e^{-5}$	$5.6e^{-5}$	.0001	.0009	.007	.049	.21	1.12	5.31	28.36	70.60
BF	$2.217e^{-5}$	.0001	0.003	0.21	22.629	-	-	-	-	-	-

Figure 6: Run-time, in seconds, of Held-Karp and Brute Force

In figure 6, we can see a comparison of run times of the brute force and the Held-Karp algorithm, in seconds, on various graphs with increasing number of vertexes. The algorithms were implemented in Python and ran on a 2013 Macbook Pro with a 2.6GHz i7 processor. The Held-Karp algorithm allows us to solve graphs almost twice the size in the same amount of time. Brute force tests that went beyond 10 vertexes took too long to wait for, as did Held-Karp tests over 22. It is worth noting that although Held-Karp is much faster, it is also using up incredibly large amount of memory to achieve such speed, where brute force works with $O(n)$ space.

Appendix A Implementations

```
import mat
import parser
import networkx as nx
import matplotlib.pyplot as plt
import heldKarp as hk
import time

#from plot import plotTSP

G=nx.Graph()
graphTups = parser.get_cities_tups();
dictNodes = parser.get_cities_dict();
graphNodes = {}

count = 0
for node in dictNodes:
    #for some reason the api I used gives me the points twice, so dont add if < len()
    if(count < len(dictNodes)/2):
        graphNodes[node[0]] = node[1]
    count+=1

count = 1
for point in graphTups:
    if(count-1 < len(dictNodes)/2):
        G.add_node(count, pos = (point[0],point[1]))
    count+=1
```

```
G.add_path(optimalPath)

nx.draw(G, nx.get_node_attributes(G, 'pos'), with_labels=True, node_size=300)
plt.show()
_
```

A way to graph a TSPLib file. Note that the actual parsing code is not shown, as an api was used to do this portion for me.

```
def getDistancesMatrix(nodes):
    distances = [[ 0 for i in range(len(nodes))] for j in range(len(nodes))]
    i = 0
    for curNode,xy in nodes.iteritems():
        j = 0
        for node,xy in nodes.iteritems():
            dist = math.sqrt(((xy[0] - nodes[curNode][0]) * (xy[0] - nodes[curNode][0])) +
                    ((xy[1] - nodes[curNode][1]) * (xy[1] - nodes[curNode][1])))
            distances[i][j] = dist
            j+=1
        i+=1
    return distances
```

A way to modify TSP input (X,Y coordinates of each vertex) into an actual matrix of distances.

```
47 def bruteForceTSPHelper (nodes, start, curNode, path, score, visited):
48
49     global bestScore
50     global bestPath
51
52
53     for node,xy in nodes.iteritems():
54         if(len(visited) is 0):
55             start = node
56         #skip visited nodes
57         if(node in visited):
58             continue
59
60         #compute distance
61         dist = math.sqrt(((xy[0] - nodes[curNode][0]) * (xy[0] - nodes[curNode][0])) +
62                 ((xy[1] - nodes[curNode][1]) * (xy[1] - nodes[curNode][1])))
63         # print "computing dist from a: %s and b: %s dist is: %d"% ( node, curNode, dist)
64         #update values
65         prevNode = curNode
66         curNode = node
67         path.append(node)
68         score += dist
69         visited[node] = 1
70
71         #did we visit them all?
72         if(len(visited) == len(graphNodes)):
73             #compute distance
74             distToStart = math.sqrt(((xy[0] - graphNodes[start][0]) * (xy[0] - graphNodes[start][0])) +
75                     ((xy[1] - graphNodes[start][1]) * (xy[1] - graphNodes[start][1])))
76             score += distToStart
77
78             if(score < bestScore): #update score if new best
79                 print("best",bestScore)
80                 bestPath = path
81                 bestScore = score
82         else:
83             #recursively call function to get all permutations
84             bruteForceTSPHelper(nodes, start, curNode, path, score, visited)
85
86         del visited[node]
87         #print("deleted : ", node)
88         del path[-1]
89         score -=dist
90         curNode = prevNode
```

A Brute Force solution. Note that the code differs from the pseudo code because recursion was used to get every possible permutation of vertexes.

```
92 def bruteForceTSP(nodes):
93
94     startNode = next (iter (nodes.keys()))
95
96     bruteForceTSPHelper(nodes = nodes,
97             start = 0,
98             curNode = startNode,
99             path = [],
100            score = 0,
101            visited = {})
```

A helper function for the brute force algorithm.

```
47 def bruteForceTSPHelper (nodes, start, curNode, path, score, visited):
48 
49     global bestScore
50     global bestPath
51 
52 
53     for node,xy in nodes.iteritems():
54         if(len(visited) is 0):
55             start = node
56         #skip visited nodes
57         if(node in visited):
58             continue
59 
60         #compute distance
61         dist = math.sqrt(((xy[0] - nodes[curNode][0]) * (xy[0] - nodes[curNode][0])) +
62                 ((xy[1] - nodes[curNode][1]) * (xy[1] - nodes[curNode][1])))
63         # print "computing dist from a: %s and b: %s dist is: %d"% ( node, curNode, dist)
64         #update values
65         prevNode = curNode
66         curNode = node
67         path.append(node)
68         score += dist
69         visited[node] = 1
70 
71         #did we visit them all?
72         if(len(visited) == len(graphNodes)):
73             #compute distance
74             distToStart = math.sqrt(((xy[0] - graphNodes[start][0]) * (xy[0] - graphNodes[start][0])) +
75                     ((xy[1] - graphNodes[start][1]) * (xy[1] - graphNodes[start][1])))
76             score += distToStart
77 
78             if(score < bestScore): #update score if new best
79                 print("best",bestScore)
80                 bestPath = path
81                 bestScore = score
82         else:
83             #recursively call function to get all permutations
84             bruteForceTSPHelper(nodes, start, curNode, path, score, visited)
85 
86         del visited[node]
87         #print("deleted : ", node)
88         del path[-1]
89         score -=dist
90         curNode = prevNode
```

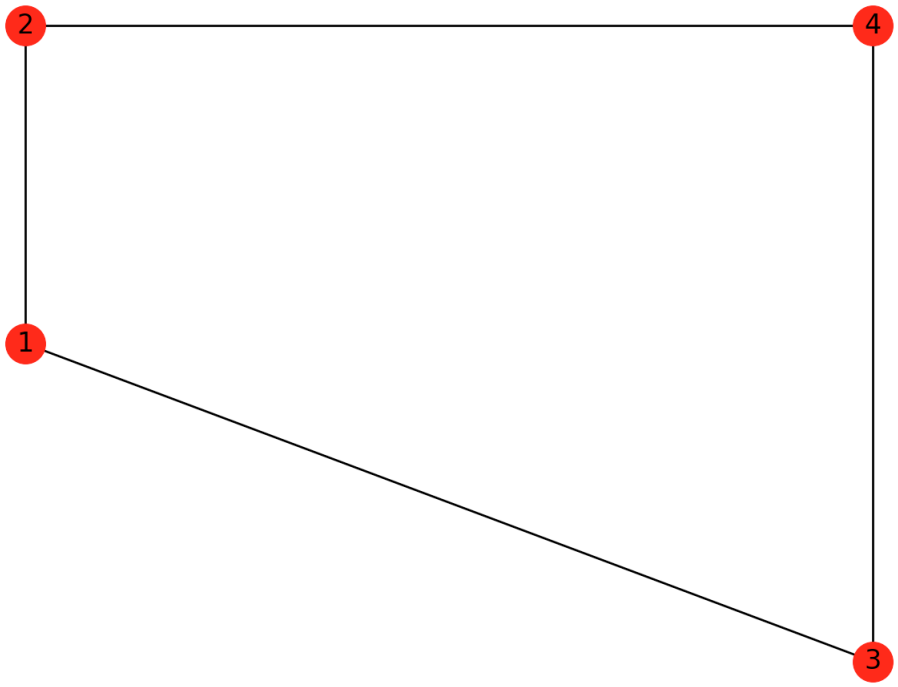
A Brute Force solution. Note that the code differs from the pseudo code because recursion was used to get every possible permutation of vertexes.

```python
def heldKarp(distMatrix):
    n = len(distMatrix)

    # <key -> subset, value -> cost to read subset>
    # also takes into accout what node was used before subset, in order
    # to reconstruct the final path
    costs = {}

    #set costs for all vertexes to dist(startVertex, vertex)
    for k in range(1, n):
        costs[(1 << k, k)] = (distMatrix[0][k], 0)

    # iterate through, get costs and put them in costs matrix
    for subset_size in range(2, n):
        #get all subsets possible
        for subset in itertools.combinations(range(1, n), subset_size):
            # represent the subset as bits to be used for the map key
            bits = 0
            for bit in subset:
                bits |= 1 << bit

            # get lowest cost for subset
            for k in subset:
                prev = bits & ~(1 << k)

                res = []
                for i in subset:
                    if (i == 0 or i == k):
                        continue
                    res.append((costs[(prev, i)][0] + distMatrix[i][k], i))
                costs[(bits, k)] = min(res)

    bits = (2**n - 1) - 1

    # get optimal cost
    res = []
    for k in range(1, n):
        res.append((costs[(bits, k)][0] + distMatrix[k][0], k))
    opt, parent = min(res)

    path = []
    for i in range(n - 1):
        path.append(parent)
        newBits = bits & ~(1 << parent)
        _, parent = costs[(bits, parent)]
        bits = newBits

    path.append(0)

    return opt, list(reversed(path))
```

An implementation of the Held-Karp algorithm. (Ekerot, 2016)

```
NAME : xqf131
COMMENT : Bonn VLSI data set with 131 points
COMMENT : Uni Bonn, Research Institute for Discrete Math
COMMENT : Contributed by Andre Rohe
TYPE : TSP
DIMENSION : 25
EDGE_WEIGHT_TYPE : EUC_2D
NODE_COORD_SECTION
1 0 1
2 0 2
3 1 0
4 1 2
EOF
```

An example input TSPLib file.



The example input TSPLib file as a graph.

Grants-MacBook-Pro:Traveling-salesman-problem eatongl$ python TSP.py **//run file**

[[0.0, 1.0, 1.4142135623730951, 1.4142135623730951], **// distances matrix**
[1.0, 0.0, 2.23606797749979, 1.0],
[1.4142135623730951, 2.23606797749979, 0.0, 2.0],
[1.4142135623730951, 1.0, 2.0, 0.0]]
best: 999999999 **//print the best solution as we find a new best**
best: 6.650281539872885
best: 5.414213562373095 **//final, best path cost**
— Brute Force ran in 0.0001540184021 seconds —
— Held-Karp ran in 5.60283660889e-05 seconds —
[1, 3, 4, 2, 1] **//print optimal path of held-karp**

The example output of the Brute Force and Held Karp to the TSPLib file.

All code can be found at:
https://github.com/GrantEaton/travelling-salesman-problem

Appendix B Peer Reviews

A special thanks to Brennan Hoeting and Luke Artnak, who helped me immensely in the editing and criteria meeting process.

Artnak gave very helpful tips on how to properly cite my in-line sources in APA format (dates, not page numbers). Artnak also pointed out that my pseudo-code for my paper did not match my implementation. I chose not to change this, because the pseudo-code was very clear and concise, yet I had already implemented my algorithm in this way. Artnak also helped me with formatting my paper properly, just general Latex tips. Artnak helped me fix up my recurrence, which had a few mistakes to it

Hoeting read through to make sure I met the criteria. He noticed that I only compared one other improvement to the TSP, so I added two more algorithm comparisons. Hoeting also pointed out that I probably did not need to explain the 0/1 matrix output of the problem, but I felt it was necessary for explaining a technical way to describe an optimal solution. Hoeting pointed out that my proof may have proved the 2^n part but possibly not the n^2 part properly.

I gave tips to Hoeting about what sources to use, as well as how to go about his proof. Having written a similar paper previously for the graduate assignment, I was able to give him tips about how to apacite properly with latex. I helped Artnak use an APA citation package in Latex.

References

Christofides, N. (1976). *Worst-case analysis of a new heuristic for the travelling salesman problem*. Graduate School of Industrial Administration, Carnegie Mellon University. Retrieved from `http://www.dtic.mil/dtic/tr/fulltext/u2/a025602.pdf`

Dantzig, F. R. . J. S., G. (1954). *Solution of a large-scale traveling-salesman problem.* Journal of the Operations Research Society of America. Retrieved from `http://books.google.com/books?id=W-xMPgAACAAJ`

Ekerot, H. (n.d.). *A pure-python held-karp implementation*. Github. Retrieved from `https://github.com/CarlEkerot/held-karp`

Hassan Ismkhan, K. Z. (n.d.). *Developing improved greedy crossover to solve symmetric traveling salesman problem*. Computer Department, University of Bonab. Retrieved from `https://arxiv.org/pdf/1209.5339.pdf`

Michael Held, R. M. K. (1962). *A dynamic programming approach to sequencing problems*. Society for Industrial and Applied Mathematics.

Nilsson, C. (n.d.). *The traveling salesman problem*. Research Gate. Retrieved from `file:///Users/eatongl/Downloads/ch10%20THE%20TRAVELING%20SALESMAN%20PROBLEM.pdf`

Roy, T. (1962). *Traveling salesman problem dynamic programming held-karp*. Video File. Retrieved from `https://www.youtube.com/watch?v=-JjA4BLQyqE`

(Dantzig, 1954) (Michael Held, 1962) (Roy, 1962) (Christofides, 1976) (Nilsson, n.d.) (Hassan Ismkhan, n.d.) (Ekerot, n.d.)