Dynamic Programming for the Traveling Salesperson Problem

David Thorne

December 14, 2023

Abstract

The Traveling Salesperson Problem (TSP) is a classical NP-hard combinatorial optimization problem with many applications in modern robotics and operations research. Soon after the introduction of Dynamic Programming (DP), two papers individually developed what is now a well-known approach for solving the TSP exactly using DP. In this project, I tried to develop my own algorithm for DPTSP with minimal knowledge of existing methods. I then compared my algorithm against an implementation of the Bellman-Held-Karp algorithm.

Introduction

The Traveling Salesperson Problem (TSP) is a classical NP-hard problem where one salesperson must visit each city in a given set exactly once, starting and ending at the same location. In 1949, a report from the RAND coorporation was the first to call this problem the 'traveling salesman problem' [Rob49], thus popularizing the term. It should not be surprising then that soon after publishing the first work on dynamic programming, Richard Bellman (also an employee at RAND coorp.) published an algorithm for solving TSP exactly using dynamic programming [Bel62]. At the same time, Michael Held and Richard Karp published a similar algorithm [HK62], leading to what is now known as the Bellman-Held-Karp (BHK) algorithm.

For this project, I wanted to use my knowledge of dynamic programming to develop an algorithm for solving the TSP exactly. The brute force search is known to have a

complexity of $O(n!)$, so I will be happy with any improvement on this. One can easily find open-source implementations of the BHK algorithm, so I lightly modified an open-source version to compare against my own algorithm. My final algorithm has a time complexity of $O(n^2 2^{2(n-1)})$ compared to the BHK algorithm which has a complexity of $O(n^2 2^{(n-1)})$. Through experiments on up to 18 cities I demonstrate the performance of my algorithm and verify my complexity analysis.

Problem Formulation

It is helpful to think of the TSP as the problem of finding the minimum length Hamiltonian cycle (defined in the following paragraph) given a graph with a set of nodes and edges with known weight $G = (S', E)$. We use S to denote a set of nodes within the graph, where S' denotes the set of all nodes. Because the edges are undirected and have known weight, we can build a symmetric cost matrix $Dist$ where the first dimension of $Dist$ denotes where the edge originates and the second dimension denotes where the edge terminates. We assign 2D coordinates to each node and build the cost matrix using the Euclidean distance between nodes. Using Euclidean distance as the cost metric means the cost matrix has a zero-diagonal with all other elements being non-negative (or strictly positive if we say no two nodes can be in the same location).

Our objective is to select an optimal set of edges (denoted as π^*) such that (1) there is a simple cycle (ie. a path that starts and ends at the same node, with no other node appearing more than once) which connects every node in S' and contains every edge in π^* and (2) the sum of the edge costs $J(\pi^*)$ in the optimal path is smaller than the sum of edge costs of any other set of edges which also satisfy condition (1). This simple cycle is also known as a Hamiltonian cycle, which is why the TSP is also sometimes called the minimum cost Hamiltonian cycle problem.

For the dynamic programming implementation of TSP, I also introduce the notation for a shortest path between two nodes which also visits every node in $S \subset S'$. We denote this optimal path as $\pi^*(i, j, S)$ and optimal cost of the path as $v^*(i, j, S)$. In practice, we are always interested in paths ending at the start node (arbitrarily set as node 0), so we use the shortened notation for optimal path and path cost $\pi^*(i, S)$ and $v^*(i, S)$.

Methodology

Disclaimer: I did read about the Bellman-Held-Karp algorithm before writing my algorithm but I did not understand or implement it until after I wrote my algorithm. I did borrow the notion of finding the shortest path from one node to another going through a set of intermediate nodes for my algorithm. Aside from knowing that concept was used in BHK, my algorithm was developed originally and I am proud of the fact that it resembles BHK. I will not cover how BHK works, for the interested reader there are many good open sources which explain the algorithm, and Bellman's paper is only three pages and quite readable despite its age [Bel62].

The pseudocode for my algorithm is given in 1. The main idea is that I want to find $(n-1)$ paths originating at node 0. Each path will end at a different node $i \in S'$ and along the way visit each node $\{S|S \in S'/i\}$ making a complete path that visits every node in S'. I then find the shortest Hamiltonian cycle by taking the minimum of these path lenghts plus the distance from $i \to 0$.

I use dynamic programming by recursively solving for the optimal paths going through larger subsets of S'. The first recursion sees the algorithm solve for the shortest path between 0 and every other node $i \in S'$ with the only intermediate city being i (ie. find $v^*(0, i, i)\ \ i \in S'$) which is just the direct path between node 0 and every other node. We then save the sets of intermediate nodes, optimal path costs, and optimal paths as 3-tuples. On the next recursion, we look at every 3-tuple from the previous recursion and for each tuple, look to add a new step in the path for each node not in the previous set of intermediate nodes. Any repeats in intermediate nodes are resolved by only saving the path with the lower cost. By the final recursion, we should have $(n-1)$ paths ending at each node in S'.

Paths generated by my algorithm can be see in 1. It becomes obvious why I can choose any arbitrary starting point, because a minimum length Hamiltonian cycle would be identical starting from any node.

Algorithm 1 Traveling Salesperson Problem via Dynamic Programming

```
# S ↔ Set of cities in path
# S' ↔ Set of all cities
# Previous step stores sets of all cities visited between start and 0, the optimal path cost, and the optimal path
prevStep = [S = {i|i ∈ S'}, v*_k = {Dist(0,i)|i ∈ S'}, π*_k = {i|i ∈ S'}]
Dist ← euclideanDist(nCities)
while len(S)! = len(S') do
    nextStep = [[], [], []]
    for each set of [S,v*_k, π*_k] in prevStep do
        for all cities do
            if city i is not in S then
                v*_{k+1} = min_{u∈S}{Dist(u,i) + v*_k(u, S/u)}
                π*_{k+1} = arg min_{u∈S}{Dist(u,i) + v*_k(u, S/u)}
                nextStep.append([S ∪ i], [v*_{k+1}], [π*_{k+1}])
            end if
        end for
    end for
    prevStep ← nextStep
end while
v* = min_i{v*_i + Dist(i,0)}
π* = π*_i + arg min_i{v*_i + Dist(i,0)}
```

Results

I found open-source code for a Python implementation of the Bellman-Held-Karp algorithm. Both my algorithm and the open-source implementation of the Bellman-Held-Karp (BHK) algorithm are run in Python, using nodes that have randomized (x,y) coordinates and a cost matrix generated by taking the Euclidean distance between the nodes. My algorithm was tested with up to 14 nodes and BHK was tested with up to 18 nodes. Results are presented in 2a.

The time complexity of solving TSP via brute force is $O(n!)$ where n is the number of nodes. A simple goal for my algorithm was to beat the complexity of the brute force algorithm which it does with a time complexity of $O(n^2 2^{2(n-1)})$. However, as seen in 2a, BHK is faster than my algorithm and has a complexity of $O(n^2 2^{(n-1)})$.

My algorithm is slower because of how I allocate and search the saved optimal

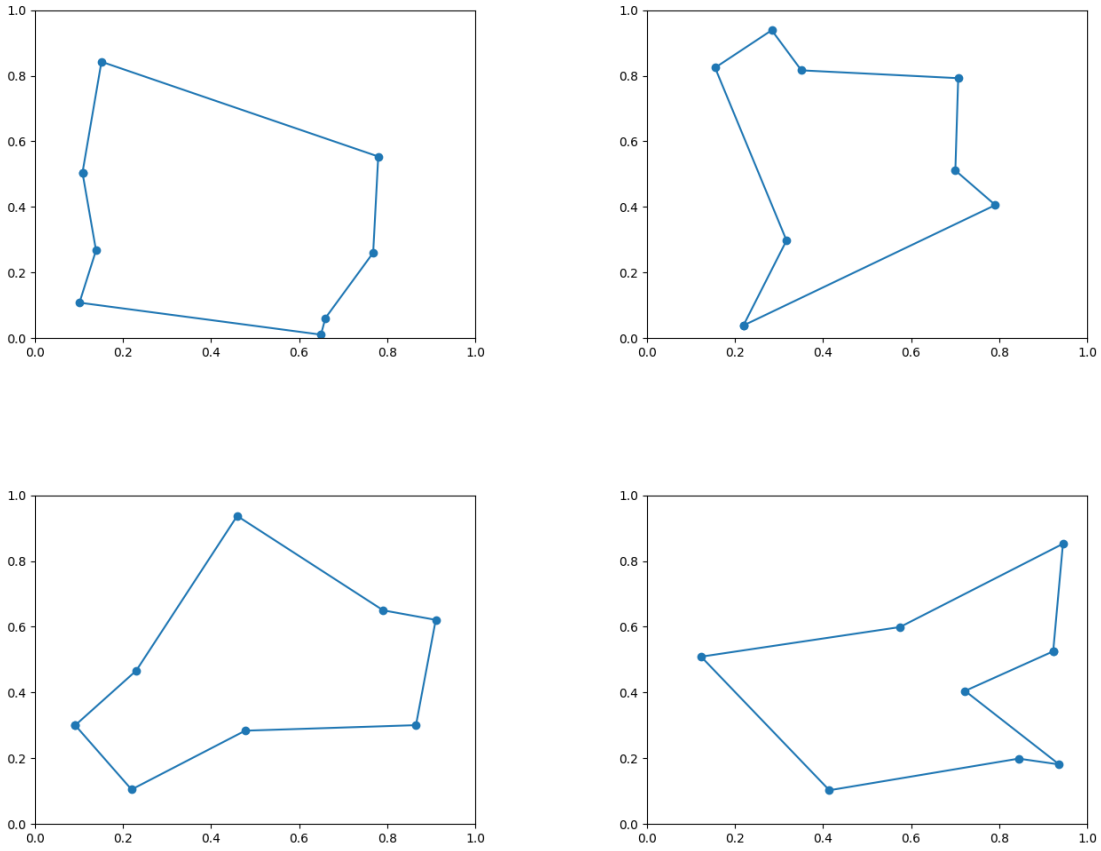
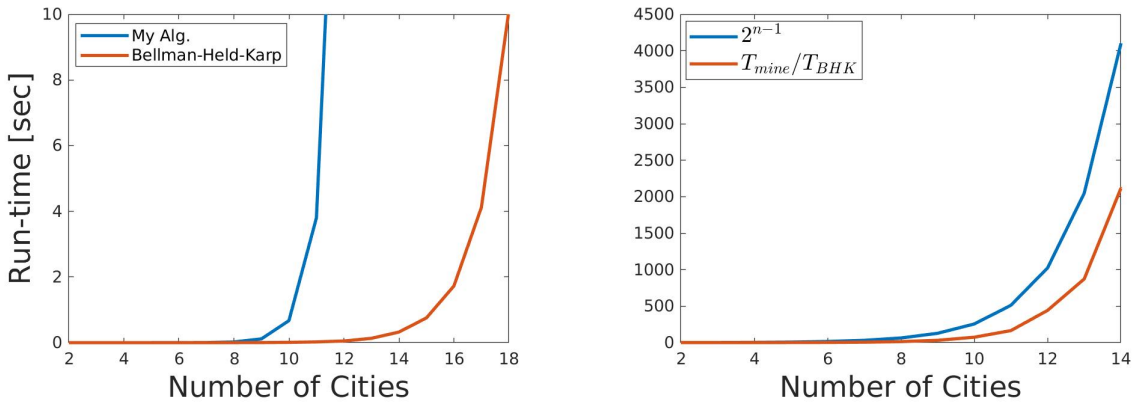


Figure 1: Solved TSP problems with 8 cities.

paths. BHK starts by allocating an array large enough for every possible combination of intermediate nodes set $S \in S'$ and starting node i. By saving optimal path costs as a pre-allocated array, BHK can search using an $O(1)$ operation. My method of dynamically saving all optimal paths from the previous step in an array of max size 2^{n-1} requires at best an $O(log(2^{n-1}))$ operation (assuming the array is sorted and a binary search is used) to search for saved paths. As I did not sort this array, I expect an additional multiplicative complexity increase of $O(2^{n-1})$ over BHK in my algorithm. To test this expectation, I plot the run-time of my algorithm divided by the run-time of BHK in 2b and show that the difference in run-time appears to have the same function form as 2^{n-1}.

As a final note, I want to compare the memory usage of my algorithm against BHK. My algorithm dynamically saves only the optimal paths from the previous step, where



(a) Number of cities for DPTSP versus run-time in seconds.

(b) Run-time of my algorithm divided by run-time of BHK. Expected a difference of 2^{n-1}.

Figure 2: Run-time and time complexity of my algorithm vs. BHK.

BHK saves everything to one pre-allocated array. In a sample case with 14 cities, my algorithm only used 1MB of RAM, where BHK used 8MB. The memory usage increases exponentially and I calculated that BHK would exceed 8GB of RAM using only 21 cities, and greatly outpace even the largest supercomputer's RAM capacity with anything above 30 cities. So BHK trades cheaper CPU performance for greater RAM usage. Therefor I conclude that my algorithm must be better and it is actually the computer hardware industry's fault for not focusing on making stronger CPUs instead of trying to needlessly increase RAM on computers.

References

[Bel62] Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM)*, 9(1):61–63, 1962.

[HK62] Michael Held and Richard M Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied mathematics*, 10(1):196–210, 1962.

[Rob49] Julia Robinson. *On the Hamiltonian game (a traveling salesman problem).* Rand Corporation, 1949.