APPROXIMATION ALGORITHMS FOR THE TRAVELING SALESMAN PROBLEM

EZRA FURTADO-TIWARI, ROGER FAN

Abstract. The Traveling Salesman Problem is a well known problem in Graph Theory and Computational Complexity Theory. It is known to be NP-Hard, so no known polynomial-time solution exists. We discuss some exact, approximation, and heuristic algorithms to approximate solutions to the problem, as well as implementations and applications to a case of the Metric TSP problem involving a tour of state capitals. All implementations and data are provided at `https://github.com/ImNotRog/TSP`.

1. Introduction

The Traveling Salesman Problem (TSP) is a famous problem in computation. It can be formulated as follows:

Definition 1.1 (Traveling Salesman Problem)**.** Given a weighted undirected graph $G = (V, E)$ and a starting node s, find the shortest path that visits all $v \in V$ exactly once and ends at s.

We consider a special case of the Traveling Salesman Problem known as Metric TSP. In Metric TSP, (V, d), where $d : V \times V \to \mathbb{R}$ is the distance function used to generate the graph, must be a metric space.

Definition 1.2 (Metric Space)**.** A metric space is a pair (X, d) where X is a set and $d : X \times X \to \mathbb{R}$ is a function satisfying the following properties:

(1) $d(x, x) = 0$ for all $x \in X$.
(2) $d(x, y) = d(y, x)$ for all $x, y \in X$.
(3) $d(x, z) \leq d(x, y) + d(y, z)$ for all $x, y, z \in X$.

The third property is called the *triangle inequality.*

We implemented three types of algorithms for Metric TSP.

The first type is called an Exact algorithm, which directly finds the shortest possible solution. A limitation of exact algorithms is that because TSP is NP-Hard, there is no known exact algorithm for it that runs in polynomial time.

The second type of algorithm is an Approximation algorithm. Approximation algorithms guarantee an upper bound on the error of the solution they find, but cannot guarantee optimality. As a result, approximation algorithms perform consistently but may not give an optimal solution.

The final kind of algorithms is a Heuristic algorithm. These algorithms give no upper bound on error, but use probabalistic and heuristic techniques to find and improve upon solutions. These algorithms can take a long time to reduce solutions but tend to produce efficient results.

Date: Applied Math H, December 2023.

2. Held-Karp Exact Algorithm

The Held-Karp algorithm is the fastest known exact algorithm to solve the Traveling Salesman Problem. A naive solution for the problem would be to enumerate all $n!$ circuits in the graph; the Held-Karp algorithm memoizes some of these transitions to avoid recomputing path weights. The algorithm can be described by the following pseudocode:

```
Initialize an array $D$ with dimensions $2^n$ by $n$.
Set $D[1][0]$ to 0.
For each $M$ between 1 and $2^n$:
    For each pair $(a,b)$ such that both $a$ and $b$ are contained
      in the mask:
        Set $D[M][b]$ = $\min(D[M][b], D[M-2^b][a]+d[a][b])$.
Return the minimum of $D[2^n-1][i]+d[i][0]$ over all $i$.
```

An important characteristic is that the answers for all submasks are computed before their corresponding mask, so the optimal path for each subset is chosen before adding an additional weight. Additionally, because the path is a circuit, it does not matter where it starts, s we may choose the starting point conveniently to be node 0. The algorithm runs in $O(n^2 2^n)$.

3. Nearest Neighbor

The Nearest Neighbor algorithm is a simple heuristic algorithm that travels to the nearest possible node at each step. The algorithm very quickly finds a route but is susceptible to large amounts of error from the optimal solution. It generally proceeds as follows:

```
For each starting node $s$ in the graph:
    Consider the current node $c$.
    While $c$ is not equal to $s$:
        Let $c'$ be the unvisited node with minimum distance
          from $c$.
        Set $c = c'$ and add the distance from $c$ to $c'$ to the
          path length.
Return the shortest path length among all posssible
  starting nodes $s$.
```

The algorithm runs in $O(n^2)$.

4. 2-Approximation

The 2-Approximation algorithm produces a solution that is at most twice the length of the optimal solution for the Metric TSP problem.

To describe the algorithm, we first need the following lemma:

Lemma 4.1. *The cost C of an optimal tour is bounded below by the weight of a minimum spanning tree.*

Proof. Consider an optimal tour T. If we delete any edge with weight W, we have a spanning tree of weight $C-W$. This spanning tree is at least the same weight as the minimum spanning tree. □

Consider a minimum spanning tree M and a graph G' made by duplicating all edges in M. We note that this graph must have an Eulerian path with twice the weight of M. Although we may not visit the same node twice, we can take advantage of the triangle inequality (assumed under the definition of a metric space); assuming we move from unvisited node A to unvisited node B by a series of visited nodes, we may instead connect the two nodes directly, which forms a path that is at least as short. Thus the weight of the path we find is at most twice the weight of the minimum spanning tree; from Lemma 5.1, we find that this is also at most twice the weight of an optimal tour.

4.1. **Implementation Details.**

To build the initial minimum spanning tree, we use Kruskal's algorithm with a Disjoint-Set Union (Union-Find) data structure, which allows us to find the minimum spanning tree in $O(n^2 \log(n))$. We note that a Depth-First-Search traversal of our spanning tree produces an Eulerian tour, so we perform this traversal in $O(n^2)$ time. Thus our final complexity is $O(n^2 \log(n))$.

5. Christofides Approximation

Christofides Approximation Algorithm builds on the 2-approximation algorithm by constructing the Eulerian tour more efficiently, guaranteeing a solution that differs from the optimal solution by at most a factor of $\frac{3}{2}$.

To strengthen our algorithm, we consider the following lemma:

Lemma 5.1. *Let $V' \subseteq V$ be a subset such that $|V|$ is even. Then the cost of a minimum cost perfect matching on V' is at most half the cost of an optimal TSP tour on V.*

Proof. Consider an optimal tour T on V. Shrink the tour to a tour T' on vertices in V' by compressing all paths through nodes in $V \setminus V'$. Note that the weight of T' is at most the weight of T by the triangle inequality. Because $|V'|$ is even, T' is the union of two perfect matchings on V', constructed by selecting alternating edges in the tour. The cost of both matchings is bounded below by the weight of a minimum cost perfect matching, so the cost of an optimal tour on V is at most twice the cost of a minimum cost perfect matching on V'. □

We start by constructing a minimum spanning tree M as in the 2-Approximation. Then, we form a set V' containing all nodes with odd degree in M. It is guaranteed that $|V'|$ is even as the sum of degrees in a graph is even. We then construct a minimum cost perfect matching P from the edges between nodes in V' that are not included in M, and add these edges to M to form a graph G'. Since all nodes in G' have even degree, we can create an Eulerian tour T on G', and then compress this path using the triangle inequality. The cost of T is sum of the cost of M and the cost of P, which is at most $1 + \frac{1}{2} = \frac{3}{2}$ the cost of the optimal tour.

5.1. **Implementation Details.**

We build the spanning tree as in the 2-Approximation. We use the BLOSSOM5 [Kol09] implementation of Edmond's Blossom Algorithm for computing the minimum cost perfect matching on V', which runs in $O(n^4)$. We then compute the Eulerian circuit in $O(n^2)$, for a total algorithm runtime of $O(n^4)$.

6. Ant Colony Optimization

Ant Colony Optimization is a TSP algorithm loosely based on the pathfinding of ant colonies. Each edge in our graph is given a "pheremone" level, which represents how much that edge has been traversed before. Then, we simulate ants by starting at a random vertex and creating a random hamiltonian circuits, prioritizing edges that have a stronger pheremone level. Then, once the ant has completed its cycle, it increases the pheremone of each edge it visited by a value that depends on how short the circuit is, so that shorter circuits are emphasized. The general algorithm is as follows:

```
Set all pheremones to 1/length.
Repeat:
    Simulate 500 ants.
    Decrease all pheremones by 5%.
```

The algorithm to simulate an ant is as follows:

```
Begin at a random vertex.
Repeat:
    For each unvisited vertex $v$, let the edge pheremone
       from the current vertex to $v$ be $p$.
    Choose to move to $v$ with relative probability $p^\alpha$, where
        $\alpha$ is a constant.
```

6.1. Implementation Details.

Implementation Details. Even though this is a relatively simple algorithm, it was quite difficult to get working. We chose the pheremone decay rate of 5% ad hoc, and we slowly increased α over the course of the algorithm from 1 to 10.

7. Simulated Annealing

Simulated annealing is a *metaheuristic* method, i.e. an general strategy that works over a large range of optimization problems. The name comes from annealing, a method in metallurgy, where varying temperatures are used to alter the properties of a metal.

In simulated annealing, we attempt to minimize a *value* over the possible *states* of the system. For example, in TSP, the states are the $(n-1)!$ different tours, and the value is its length. For notation purposes, let $v(s)$ be the value of state s.

We also introduce a "temperature" variable, T, that represents how much randomness is used in our search. The general algorithm is as follows:

```
Set s to be a random state.
Repeat:
    Generate a random neighbor s' of s.
    If v(s') < v(s), then set s to s'.
    Otherwise, set s to s' with probability exp((v(s)-v(s'))/T).
    Decrease T.
```

The idea is to switch from states to "better" states, or states with a smaller value. However, there is a small probability of switching from s to a *worse* state s' as well, which is $\exp\left(\frac{v(s)-v(s')}{T}\right)$. Using this formula, the likelihood of switching is lower when s' is worse.

Also, the likelihood decreases as the temperature T decreases: when T is high, there is a lot of randomness in our search, which helps us break out of local minima. After a while, though, T becomes low, and we then don't have as much randomness, which allows us to more carefully narrow in on a minimum.

7.1. Implementation Details.

For our TSP problem in particular, we generated random neighbors by swapping random vertices, and reversing random segments of the current tour. We ran $100,000$ steps, and decreased T exponentially with a rate of .99 at each step, which seemed to produce the best resutls on our problem.

In our experience, the performance of Simulated Annealing was finnicky, as there was a lot of randomness, and we had to determine the number of steps and the rate of decrease ad hoc. In the end, running the program multiple times seemed to consistently produce strong solutions.

8. Lin-Kernighan Heuristic

The Lin-Kernighan Heuristic algorithm (LKH) is widely accepted to be one of the best methods of approximating TSP.

8.1. *k*-Opt.

LKH is an extension of the k-opt method, where we optimize an existing tour by removing k edges from it and replacing them with k new edges. We then check if this makes the tour shorter. A 3-opt move is shown in Figure 1: we remove the three red edges and add the three blue ones. (We say an edge is red if it is in the current tour, and blue otherwise.)

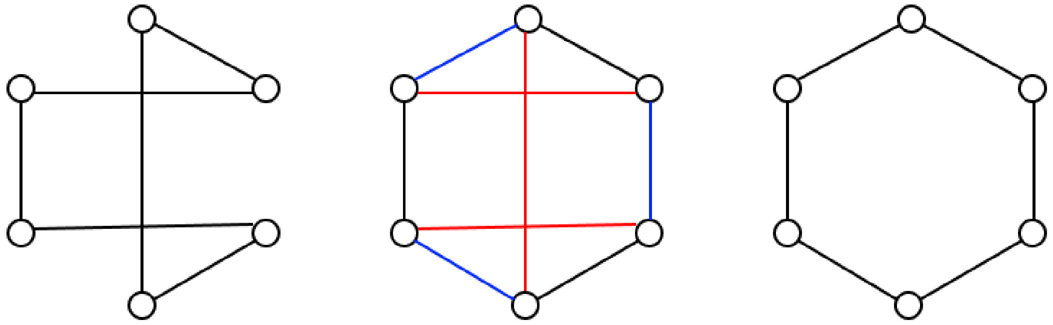

Figure 1. A 3-opt move.

The degree of each vertex before and after our k-opt move must be 2, since we begin and end with a tour. This implies that for every red edge we remove from a vertex, there must be a blue edge we add to a vertex, which implies that the blue and red edges form a circuit of their own! Instead of searching through all possible sets of edges to add and remove, we simply have to search through these alternating red and blue circuits, where the blue edges are part of our previous tour and the red edges are not.

Let our previous tour be T, and let our alternating path be A. We will write $T \triangle A$ to denote our modified tour after deleting the red edges of A and adding its blue edges. It turns out, given some tour T, not all alternating paths A yield valid tours for $T \triangle A$, since our modified tour may end up disconnected, as shown in Figure 2. As a result, we always manually check that $T \triangle A$ is actually a tour.

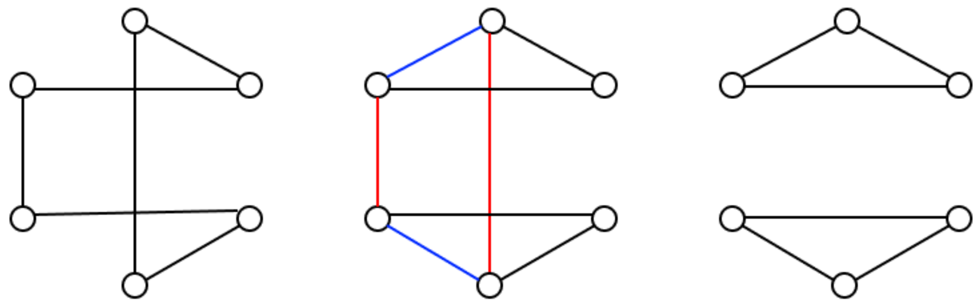

Figure 2. A failed 2-opt move, resulting in two disconnected components.

Also, we can easily compute whether an alternating path A would make our tour T shorter. Let the *gain* of A be the the total length of the red edges minus the total length of the blue ones. This is the length that would be subtracted from T, so if the gain is positive, then we will end up with a shorter tour. Using this, we now have the following k-opt algorithm:

```
Set $T$ to be a random tour.
Repeat:
    For every alternating circuit $A$ with $k$ red edges and $k$
       blue edges:
        If $A$ has positive gain and $T \triangle A$ is a tour:
            Set $T$ to $T \triangle A$.
    If no alternating circuit $A$ improves $T$, terminate
       program.
```

8.2. **LKH.** The LKH algorithm improves on k-opt by choosing the value k on the go. We instead search through possible alternating *paths* A with positive gain, which we eventually attempt to close into alternating circuits. (An alternating path is simply a path with alternating red and blue edges, and we write $P \cup E$ to denote the alternating path P extended by the edge E. The gain of P is again the total length of the red edges minus the blue edges, and we write $|P|$ to denote the number of edges in P.) We now present a sketch of the basic algorithm:

```
Set $T$ to be a random tour.
Repeat:
    Set $S$ to be the stack of alternating paths to explore.
    For each vertex $v$, push length-1 paths starting at $v$ to
       $S$.
    Repeat:
        Pop path $P$ off the stack.
```

```
If $|P|$ is odd, we wish to add a red edge:
    If $P$ can be closed by a red edge $R$:
        If $P \cup R$ has positive gain and $T \triangle (P \cup R)$ is a
            tour:
            Set $T$ = $T \triangle (P \cup R)$.
            Break loop.
    For every red edge $R$ extending $P$:
        If $P \cup R$ has positive gain:
            Push $P \cup R$ onto $S$.

If $|P|$ is even, we wish to add a blue edge:
    For every blue edge $B$ extending $P$:
        If $P \cup B$ has positive gain:
            Push $P \cup B$ onto $S$.
```

To curb our search, notice we only explore along paths that *always* have a positive gain.[1] Note also that there are always only 2 blue edges that extend a path P, since every vertex only has two neighbors in the current tour.

8.3. **Restrictions and Optimizations.** The LKH algorithm above explores too many paths and does not run efficiently. To further curb our search, we introduce the following restrictions:

(1) We never add an edge E that visits an already visited vertex of P.
(2) If $|P|$ is even and greater than $c_1 = 4$, we only add a blue edge B to our path P if $P \cup B$ can be immediately closed with a red edge R, and $(P \cup B \cup R) \triangle T$ is a tour.
(3) If $|P|$ is greater than $c_2 = 7$, we only consider one way of extending the trail with an edge. Instead of adding all possible extensions to the stack, we only add the one with the most gain.
(4) If $|P|$ is greater than $c_3 = 5$, we only search through the 10 nearest neighbors to the last vertex of P when considering new red edges to add.

We chose the values of c_1, c_2, and c_3 ad hoc.

9. Application

We ran each algorithm on two datasets from [Jas], one consisting of locations of 20 United States capitals, and one consisting of all 50. We used the Haversine formula to compute distances between points given by latitude and longitude. This assumes that one can travel directly between capitals through great circles on the earth, which may not be possible due to waterways, mountains, and road configurations.

[1]For every alternating circuit with positive gain, there is always a vertex v such that you can start at v and traverse through the circuit while always maintaining positive gain along the edges you have visited. Thus, even with our restriction, our LKH algorithm will eventually visit all alternating circuits with positive gain.

Algorithm	20 Capitals (km)	50 Capitals (km)
Nearest Neighbor	21600.3	40666.6
2-Approximation	20644.4	33197.1
Christofides Approximation	20475.9	29094.0
Ant Colony Optimization	19765.7	28142.7
Simulated Annealing	19319.8	27376.7
Lin-Kernighan Heuristic	19252.9	26346.4
Held-Karp	19252.9	—

The LKH algorithm provided the most optimal path in both cases, matching the optimal solution for 20 capitals. Simulated Annealing also provided near-optimal solutions in both cases.

10. Conclusions

Heuristic algorithms generally performed the best. Although approximation algorithms guarantee an upper bound on the solution weight, they do not perform especially well. However, an approximation algorithm could provide an initial solution to improve upon using a heuristic algorithm in order to reduce the amount of time necessary to compute an efficient solution.

Given more time, we could implement hybrid algorithms derived from combining multiple algorithms used or allow the heuristic algorithms we implemented to complete more iterations.

References

[Hel00] Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European journal of operational research*, 126(1):106–130, 2000.

[Jas] Github - US State Capitals. `https://github.com/jasperdebie/VisInfo/blob/master/us-state-capitals.csv`. Accessed: 2023-12-14.

[Kol09] Vladimir Kolmogorov. Blossom 5: A new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation*, 1:43–67, 2009.

[Ton14] Chaoxu Tong. Approximation algorithms for the traveling salesman problem. *ORIE 6300 Mathematical Programming I*, 2014.