Comparison of Traveling Salesman Algorithms

Austin Bomhold

Introduction

The Traveling Salesman Problem (TSP) is a classic optimization problem in computer science and operations research. Given a set of cities and the distances between them, the goal is to find the shortest possible route that visits each city exactly once and returns to the starting city. The TSP has applications in various fields, including logistics, manufacturing, and computer science. Since it can be abstracted as a graph problem, its actual uses can be found in a plethora of state-based problems. The problem is NP-hard; thus, any exact solution will have an exponential time complexity. We will compare two exact algorithms (Brute Force and Held-Karp) and two approximate algorithms (Christofides and Nearest Neighbor) to solve the TSP. The goal is to evaluate the performance of these algorithms in terms of path cost and execution time.

Algorithm Presentation

The first two algorithms are exact algorithms that guarantee an optimal solution. The Brute Force algorithm explores all possible permutations of the cities to find the shortest path. This is a naive approach that becomes intractable for large instances due to its factorial time complexity. However, it serves as a baseline for comparison with other algorithms. The heart of the algorithm is the recursive loop that generates all possible paths and calculates their total distance. The following code snippet illustrates the core logic of the Brute Force algorithm.

```
def brute_force(path: list[Point], visited: set[int]) -> tuple[float, list[Point]]:
    if len(visited) == node_count:
        return calculate_distance(path[-1], path[0]), [path[0]]
    best_distance = float('inf')
    best_path = []
    for j in range(node_count):
        if j not in visited:
            current_distance, sub_path = brute_force(path + [path[j]], visited | {j})
            current_distance += calculate_distance(path[-1], path[j])
            if current_distance < best_distance:
                best_distance = current_distance
                best_path = [path[j]] + sub_path
    return best_distance, best_path
```

The Held-Karp algorithm is another exact algorithm. However, it uses dynamic programming, in my case, memoization, for simplicity to avoid redundant calculations. The algorithm builds up a table of sub-problems and their solutions, which are then used to solve larger sub-problems. This approach reduces the time complexity from factorial to exponential. While this is a significant improvement, the algorithm is still not practical for large instances. As the next code snippet shows, the algorithm does not differ much from the Brute Force algorithm in terms of logic:

```
def held_kemp(pos: int, mask: int, node_array: list[Point], node_count: int,
              memo: dict[tuple[int, int], tuple[float, list[Point]]]) \
```

```
                                        -> tuple[float, list[Point], int]:
    if mask == (1 << node_count) - 1:
        cost += 1
        return calculate_distance(node_array[pos], node_array[0]), [node_array[0]]
    if (pos, mask) in memo:
        (best_distance, best_path) = memo[(pos, mask)]
        return best_distance, best_path, cost
    best_distance = float('inf')
    best_path = []
    for j in range(node_count):
        if not (mask & (1 << j)):
            current_distance, sub_path, cost = solve_loop(
                                                j,  # new position
                                                mask | (1 << j), # new mask
                                                node_array,
                                                node_count,
                                                memo)
            current_distance += calculate_distance(node_array[pos], node_array[j])
            if current_distance < best_distance:
                best_distance = current_distance
                best_path = [node_array[j]] + sub_path
    memo[(pos, mask)] = (best_distance, best_path)
    return best_distance, best_path, cost
```

On one more minor note, I encoded the positions and paths as integers and used bitwise operations to manipulate them. This may seem out of place compared to the rest, but it was a result of various implementation attempts, and I did not feel the need to change it for the sake of consistency.

Even though the Held-Karp algorithm is the most efficient exact algorithm, its inability to achieve sub-exponential time complexity demonstrates the difficulty of the NP problems. This is especially true for the TSP, as an NP-hard problem, where even checking the validity of a solution is NP. As such, we turn to approximate algorithms to find solutions in a reasonable amount of time. The nearest neighbor algorithm is a simple greedy approach and a good starting point for investigating heuristics. The algorithm starts at a random city and repeatedly visits the nearest unvisited city until all cities are visited. This approach is speedy, with a time complexity of O(n^2), and is the first actually tractable algorithm. However, it does not guarantee an optimal solution and, in fact, makes no promises about the quality of the solution.

```
def nearest_neighbor(node_array: list[Point]):
    route = [node_array.pop()]
    remaining: set[Point] = set(node for node in node_array)
    while remaining:
        current_city = route[-1]
        nearest_city = min(
                    [(node, calculate_distance(current_city, node)) for node in remaining],
```

```
                    key=lambda x: x[1])
        route.append(nearest_city[0])
        remaining.remove(nearest_city[0])
        route.append(route[0])
    return route
```

Finally, we have the Christofides algorithm, which is another approximate algorithm. This algorithm guarantees a solution within 3/2 of the optimal solution for any instance of the TSP. It does raise the runtime complexity to O(n^4), but this is still within the realm of tractability. The algorithm consists of three main steps: finding a minimum spanning tree, finding a perfect matching, and finding an Eulerian circuit. For the minimum spanning tree, I used Prim's algorithm. I chose this because it is a simple algorithm that is easier to implement. While it is slower than other algorithms like Kruskal's, it's O(n^2) time complexity was less of a concern against the overall time complexity of O(n^4).

```
def prims_mst(node_array: list[Point]) -> list[tuple[Point, Point]]:
    unselected = set(node_array.copy())
    mst = []
    for _ in range(len(unselected) - 1):
        min_edge = float('inf')
        (p1, p2) = (None, None)
        for point in node_array:
            if point in unselected:
                for other_point in node_array:
                    if other_point in unselected and point != other_point:
                        dist_to_other = calculate_distance(point, other_point)
                        if dist_to_other < min_edge:
                            min_edge = dist_to_other
                            (p1, p2) = (point, other_point)
        mst.append((p1, p2))
        unselected.remove(p1)
    return mst
```

We simply start at a random node and add the closest node to the tree until all nodes are connected. This is very similar to the nearest neighbor algorithm, but instead of connecting the nodes directly, we add the edge to the tree. Once we have the minimum spanning tree, we need to create a path across all edges. In order to guarantee that this is possible, we need to pair up the nodes that have an odd number of edges. This is done by finding the minimum weight matching of the odd nodes. I had two different implementations of this, one using the Blossom algorithm and one using a greedy approach. Implementing the Blossom algorithm from scratch was a bit too much for this project, so I opted to use a Python library. The implementation is as follows:

```
def min_weight_matching_nx(odd_vertices: list[Point]) -> list[tuple[Point, Point]]:
    G = nx.Graph()
    for i, v1 in enumerate(odd_vertices):
        for j in range(i + 1, len(odd_vertices)):
```

```
            v2 = odd_vertices[j]
            weight = calculate_distance(v1, v2)
            G.add_edge(v1, v2, weight=weight)
    matching = nx.min_weight_matching(G)
    return list(matching)
```

The problem with this is that I am tracking the runtime of these algorithms via a custom distance function and the library does not allow for this or any other way to track the internal calculations. Thus, I also implemented a greedy approach that will allow me to track the runtime. It is less efficient, so it won't have the expected runtime or path cost, but it is a good approximation for comparison.

```
def min_weight_matching(odd_vertices: list[Point]) -> list[tuple[Point, Point]]:
    matching = []
    unmatched = odd_vertices.copy()
    while unmatched:
        min_dist = float('inf')
        min_pair = None
        for (index, p1) in enumerate(unmatched):
            for j in range(index + 1, len(unmatched)):
                p2 = unmatched[j]
                dist_to_p2 = calculate_distance(p1, p2)
                if dist_to_p2 < min_dist:
                    min_dist = dist_to_p2
                    min_pair = (p1, p2)

        if min_pair:
            matching.append(min_pair)
            unmatched.remove(min_pair[0])
            unmatched.remove(min_pair[1])
        else:
            break
    return matching
```

Once we have the minimum weight matching, we can combine the minimum spanning tree and the matching edges. This will give us a graph with all nodes having an even number of edges and allow us to find an Eulerian circuit. An Eulerian circuit is a path that visits every edge exactly once and returns to the starting node. We start with by grouping the edges into nodes and then starting at the node with the most edges. We then traverse the graph, removing the connection nodes from each edge as we go.

```
def find_complete_path(edges: list, node_array: list[Point]) -> list[Point]:
    connected_nodes = {nodes: [] for nodes in node_array}
    for (node_one, node_two) in edges:
        connected_nodes[node_one].append(node_two)
        connected_nodes[node_two].append(node_one)
```

```
    start_node = max(connected_nodes, key=lambda x: len(connected_nodes[x]))
    stack = [start_node]
    complete_path = []
    while stack:
        current_node = stack.pop()
        complete_path.append(current_node)
        for neighboring_node in connected_nodes[current_node]:
            if current_node in connected_nodes[neighboring_node]:
                connected_nodes[neighboring_node].remove(current_node)
            if connected_nodes[neighboring_node]:
                stack.append(neighboring_node)
    return complete_path
```

Finally, we have a complete path that visits every edge exactly once. All that is left is to remove the duplicates from the path. This is done by traversing the path and adding each node to a list if it is not already in the list. Then, finish by appending the starting node to the end of the list. This will give us the final path that visits every node exactly once and returns to the starting node.

```
def trim_path(completed_path: list[Point]) -> list[Point]:
    trimmed_path = []
    visited = set()
    for node in completed_path:
        if node not in visited:
            trimmed_path.append(node)
            visited.add(node)
    trimmed_path.append(trimmed_path[0])
    return trimmed_path
```

Experimental Design

The goal of this project is to compare the solutions of four TSP algorithms: Brute Force, Held-Karp, Nearest Neighbor, and Christofides. During the test, I aim to evaluate both the accuracy of the path and the calculations required. To judge the accuracy of the path, I will look at the spread of the path costs across 1000 consistent instances. Each instance will have the same number of nodes, fourteen, but the positions of the nodes will be randomly generated and guaranteed to be unique. The idea is to then compare the spread of the approximations to the spread of the optimal path costs using the Held-Karp algorithm as a baseline. This will tell us what the expected variation in path costs is for an exact solution and hopefully give us a better understanding of the quality of the approximations.

I will also compare the accuracy of the two different minimum weight matching implementations in the Christofides algorithm. These are the Blossom algorithm provided by the Networkx library and the greedy approach I implemented. To evaluate the runtime of the algorithms, I will measure the amount of calls to the distance function. Each algorithm implements the same distance function provided by the graph script. This

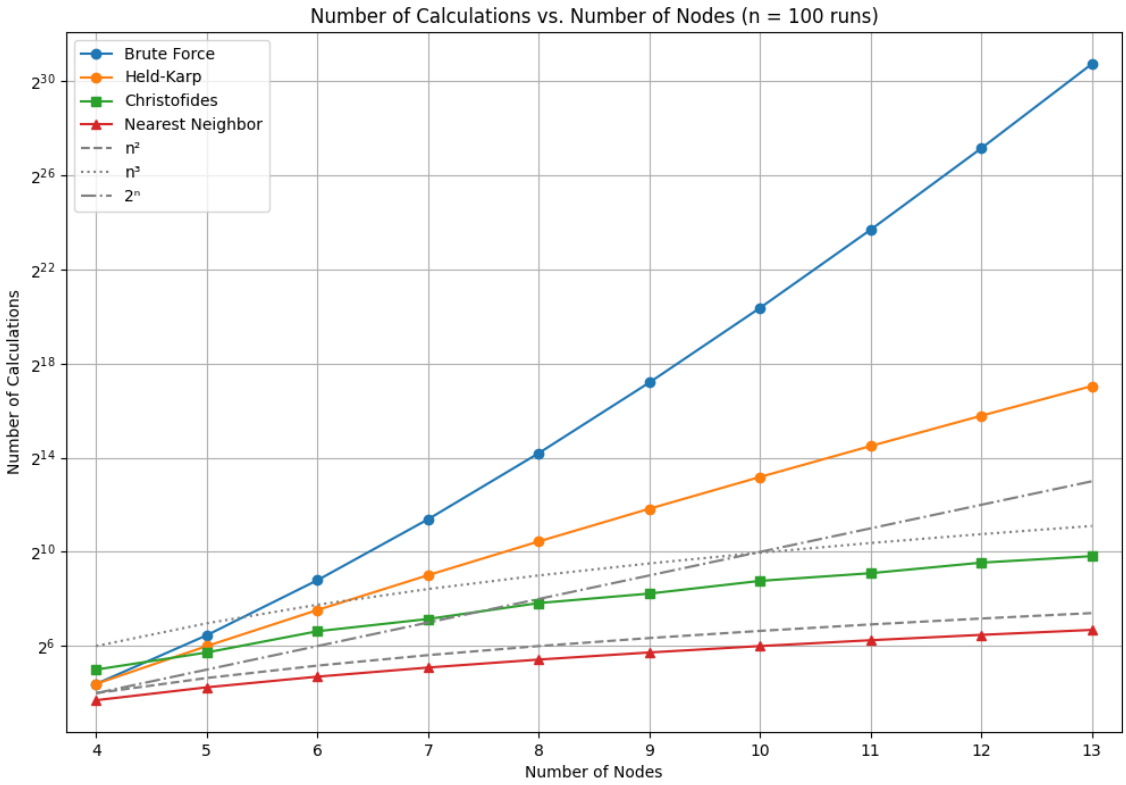
ensures that the calculations are consistent across all algorithms. Interestingly, during testing, I often used a pre-calculated distance matrix to speed up the computations allowing me to test more instances in a shorter amount of time while still maintaining the consistency of the calculations. As the function is used as a counter, even if the distance is pre-calculated, the counter will still increment.

```
def calculate_distance(point1, point2) -> float:
    global calculations
    calculations += 1
    p1_x, p1_y = point1
    p2_x, p2_y = point2
    return math.sqrt(
        math.pow(p1_x - p2_x, 2) +
        math.pow(p1_y - p2_y, 2))
```

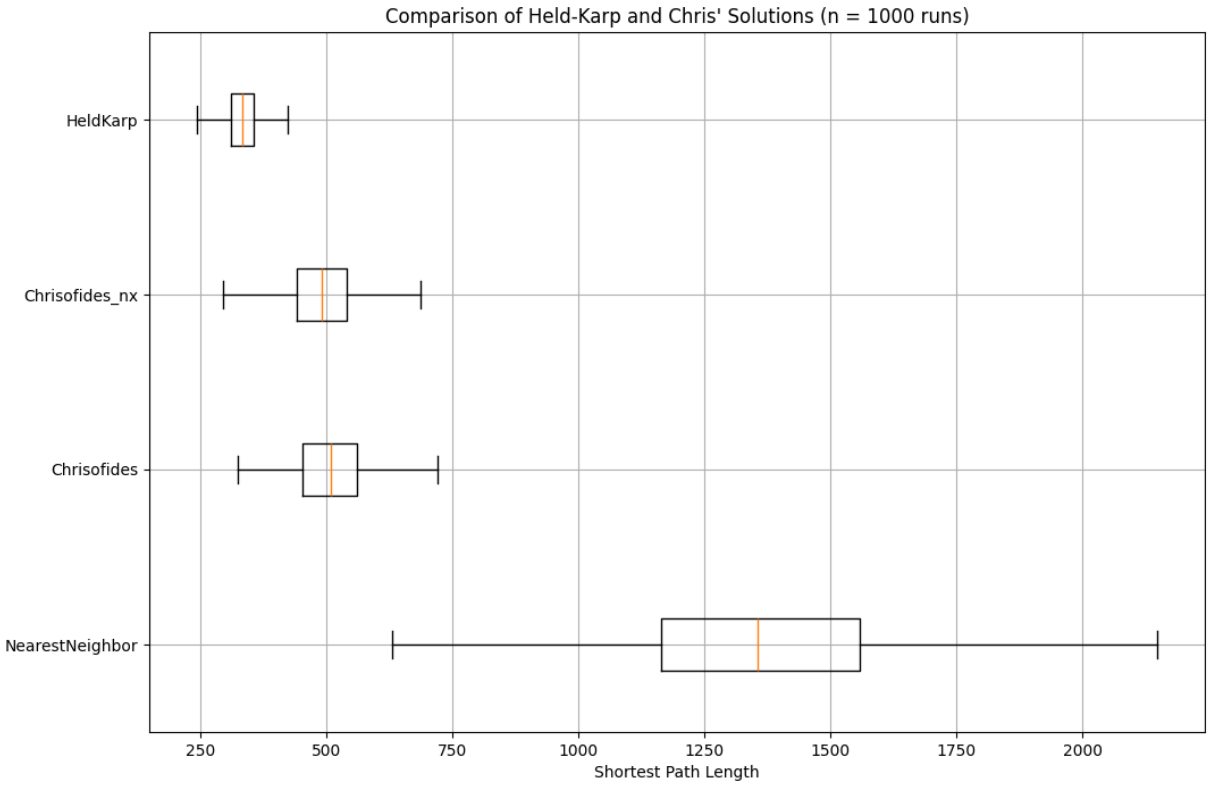
There are a couple of differences between the runtime test and the accuracy test. First, the runtime test will use random test instances and apply them to each of the algorithms. Second, the runtime test will track the worst result of all test instances, whereas the accuracy test will focus on the spread of the results. This will give us an idea of the upper bound of the runtime for each algorithm, which is more advantageous for practical applications. In contrast to the accuracy test, which will use instances of the same size, the runtime test will use instances of increasing size, from 4 to 13 nodes. In the accuracy test, we want to see how the algorithms perform on consistent instances, while in the runtime test, we want to see how the algorithms scale with the number of nodes. Finally, the runtime test will only run ten instances per algorithm per 100 runs, while the accuracy test will run 1000 instances per algorithm during a single run. In total, this should give us two different perspectives on the performance of the algorithms. Each has relatively good rigor and consistency while testing in different ways to paint a complete picture.

Results

After running the experiments, including offloading some of the exponential calculations to a Google Colab instance, I was able to gather the following results.



Firstly, the calculations are represented by the y-axis, and the number of nodes in the instance is represented by the x-axis. The y-axis is scaled logarithmically to better visualize the exponential growth, and to keep with conventions, a base of 2 was chosen. As we can see, each algorithm is represented by a different colored line. Additionally, the graph includes three general runtime categories to help orient the viewer on the logarithmic scale. These are the squared, cubed, and exponential time complexities, indicated by the grey dotted lines. This brings us to the accuracy comparison.



Here, a box plot is used to show the path costs spread by each algorithm. Each of the approximate algorithms is compared to the optimal solution provided by the Held-Karp algorithm. They are then displayed in order ---spread. The y-axis represents each of the algorithms, and the x-axis represents the path cost. Additionally, to compare the effect of the minimum weight matching implementation in the Christofides algorithm, the Blossom algorithm is compared to the greedy approach. The implementation of the Christofides algorithm using the Blossom algorithm is notated as **`Christofides_nx`**, as it uses the networkx library. The graph was left relatively simple to make it easier to pinpoint the critical figures represented in the box plot (median, IQR,…).

Discussion

The runtime results were mostly in line with expectations. The Brute Force algorithm was by far the slowest, with an exponential time complexity. We expected this was going to happen, but it added validity to the testing process. Similarly, the Held-Karp algorithm was the most efficient exact algorithm but still had a greater than exponential time complexity. The approximate algorithms were much faster.

The Nearest Neighbor algorithm was the fastest, with a time complexity of O(n^2). This one was interesting as it is increasingly fast as the number of nodes increases and has plenty of room for optimization. I think that for practical applications, there is an effective and relatively simple solution that lies somewhere between the Nearest Neighbor and Christofides algorithms. Finally, the Christofides algorithm was the slowest of the approximate algorithm and was the only one to have an unexpected time complexity. The time complexity was expected to be O(n^4) but was actually O(n^3). I believe this was due to the implementation of minimum weight matching. As mentioned earlier, the runtime test utilized a ready approach to the minimum weight matching problem in the Christofides algorithm. The greedy approach was around a factor of n faster than computing an optimal solution and considerably less complex.

Unlike with the runtime test, I didn't have much in the way of expectations for the accuracy test. I knew that the Christofides algorithm was supposed to be at most 3/2 times the optimal solution. I also thought the Nearest Neighbor algorithm was going to be the worst, though I didn't know by how much. After some initial struggles where I seemed to have gotten my data mixed up, I was able to get reliable results. In the end, however, the accuracy test provided some unexpected results. Firstly, the Nearest Neighbor algorithm was far worse than I expected. If you look at the box plot, you can see that even in the instances where all other algorithms were close to the optimal solution, the Nearest Neighbor algorithm was far off. In fact, the minimum path cost of the Nearest Neighbor algorithm was close to 1.5 times the optimal solution's maximum path length. Even more surprising was that, in some instances, the Nearest Neighbor algorithm seemed to generate a path that was twice the optimal solution. Clearly, the Nearest Neighbor algorithm is not a good choice if accuracy is a concern.

This brings us to the Christofides algorithm. The surprise here is just how little the Christofides algorithm deviated from the optimal solution. The Christofides algorithm is much closer to the optimal solution than the nearest neighbor, which demonstrates why it was such an important algorithm. Another notable revelation was just how insignificant the difference was between the minimum weight matching with greedy and Blossom implementations. The greedy approach was a factor of n faster than expected, but the path cost was almost identical. On top of that, the greedy approach to the minimum weight matching problem was much less complex than the Blossom algorithm. There is one big caveat here, as the design of the test instances embedded an assumption that the triangle inequality would hold. This is because the points were generated on a 2D plane, and the distance function was the Euclidean distance. This is a valid assumption in a lot of cases. However, given instances where the cost of traveling between two points is not symmetric, the Christofides algorithm would not be able to guarantee a solution within 3/2 of the optimal solution.

In conclusion, this project was a great exercise in algorithm implementation and testing. I was able to implement four different algorithms to solve the TSP and compare their performance. As well as practice with some of the ideas and concepts that were covered in the course. I would say that I'm probably 80% satisfied with the results. Given more time, I would try to remove this bias from my tests and see how the algorithms perform in a more general case, especially against the nearest neighbor algorithm. Finally, I would have liked to track the runtime of the Blossom algorithm to see if I can confirm the missing factor of n.