Empirical Comparison of Exact and Approximate Algorithms for the Traveling Salesman Problem

Alex Keller, Ryan Leacock, Jimmy Finnegan, Graham Baker

The Traveling Salesman Problem (TSP) is a classic optimization problem in which a salesman must visit a set of n cities exactly once and return to the starting point. The goal is to minimize the total distance travelled. Cities are represented as points in a 2-dimensional plane and edge weights correspond to Euclidean distances. The problem is known to be NP-hard, which makes exact algorithms computationally infeasible for large inputs. Because of this, many heuristic and approximation methods are used to obtain high quality solutions within reasonable time frames. In this project, we implemented and evaluated three approaches:

1. Brute Force Enumeration:
   This algorithm generates every possible Hamiltonian cycle by fixing the starting city at index 0 and finding all (n−1)! permutations of the remaining cities. Because each tour's total distance is calculated, the algorithm identifies the optimal tour. However, its factorial runtime restricts its use to very small n.

2. Greedy Nearest-Neighbor Heuristic:
   This approximation method builds a tour incrementally. Starting from city 0, it repeatedly moves to the nearest unvisited city until all cities have been visited, then returns to the start. The algorithm runs in O(n^2) time and is extremely fast. But the greedy choice can lead to suboptimal tours, especially when early decisions force long detours later.

3. Christofides Approximation Algorithm:
   Based on Christofides method, this algorithm builds a minimum spanning tree (MST), identifies all odd-degree vertices, performs a greedy matching among them, and then finds a Eulerian tour in the resulting multigraph. Shortcutting repeated vertices produces a Hamiltonian cycle. The method generally yields high-quality tours and maintains O(n^2) runtime.

These three methods allow us to compare exact optimal solutions to two fundamentally different approximation strategies, one greedy and one structural. This provides insight into accuracy, runtime, scalability, and empirical behavior.

Before conducting any experiments, we established several research questions to guide our evaluation. These questions reflect both algorithmic performance and solution quality.

1. How do the runtimes of the three algorithms scale as the number of cities increases?

We expect brute force to become infeasible for relatively small instances (approximately n>14), while both nearest-neighbor and Christofides should remain fast for much larger inputs due to their polynomial time.

2. How close do the heuristic methods (NN and Christofides) come to the optimal solution on small instances?

Our expectation is that nearest-neighbor will show higher variability from the optimal solution on certain inputs, while Christofides should consistently produce tours closer to optimal.

3. Does Christofides outperform nearest-neighbor in terms of solution quality?

Given its more global structure, we expect Christofides will typically generate shorter tours than nearest-neighbor.

4. Does input structure influence heuristic performance?

Although our primary experiments use randomly generated point sets, we expect that nearest-neighbor will be more sensitive to city arrangement due to local decisions, whereas Christofides should be more stable across different layouts.

These questions form the foundation of our experimental analysis and shape the metrics, datasets, and methods chosen for this investigation.

For our experiments, all three methods were implemented in C++17 and compiled using g++ (MSYS2 MinGW-w64) with the -O2 optimization flag. Each program reads a text file containing a list of city coordinates, where each city is represented by a (x,y) point in the plane. We generated artificial datasets of varying sizes n = 5, 8, 10, 13, 20, 50, and 100 using random coordinates to simulate Euclidean TSP instances. The smaller datasets, n <= 13, were specifically selected because they are small enough for the brute force algorithm to find the optimal tour. The larger datasets, n >= 20, were used to evaluate scalability, since brute force becomes infeasible due to factorial growth runtime. Each dataset was run once per algorithm because all three algorithms are deterministic and therefore produce the same output on repeated runs.

All experiments were executed on the same machine, to ensure consistent performance measurement. They were done on a laptop with an Intel Core i7 processor with 32 GB of RAM and running on Windows 10. Execution time for each algorithm was measured using C++'s std::chrono::high_resolution_clock and reported in milliseconds. For every trial we recorded two main metrics: total tour length and runtime. The total tour length reflects the quality of the solution and the runtime reflects algorithmic efficiency and scalability. For the datasets where brute force was feasible, its output served as the optimal solution. We used this to calculate how close the Greedy and Christofides algorithms came to optimal by comparing their tour lengths to that of brute force. For larger datasets, where no optimal solution could be computed, we compared Greedy and Christofides directly against each other in terms of runtime and solution quality. Overall, the experiment setup clearly captures both accuracy and performance differences between the algorithms as problem size increases.

Results:

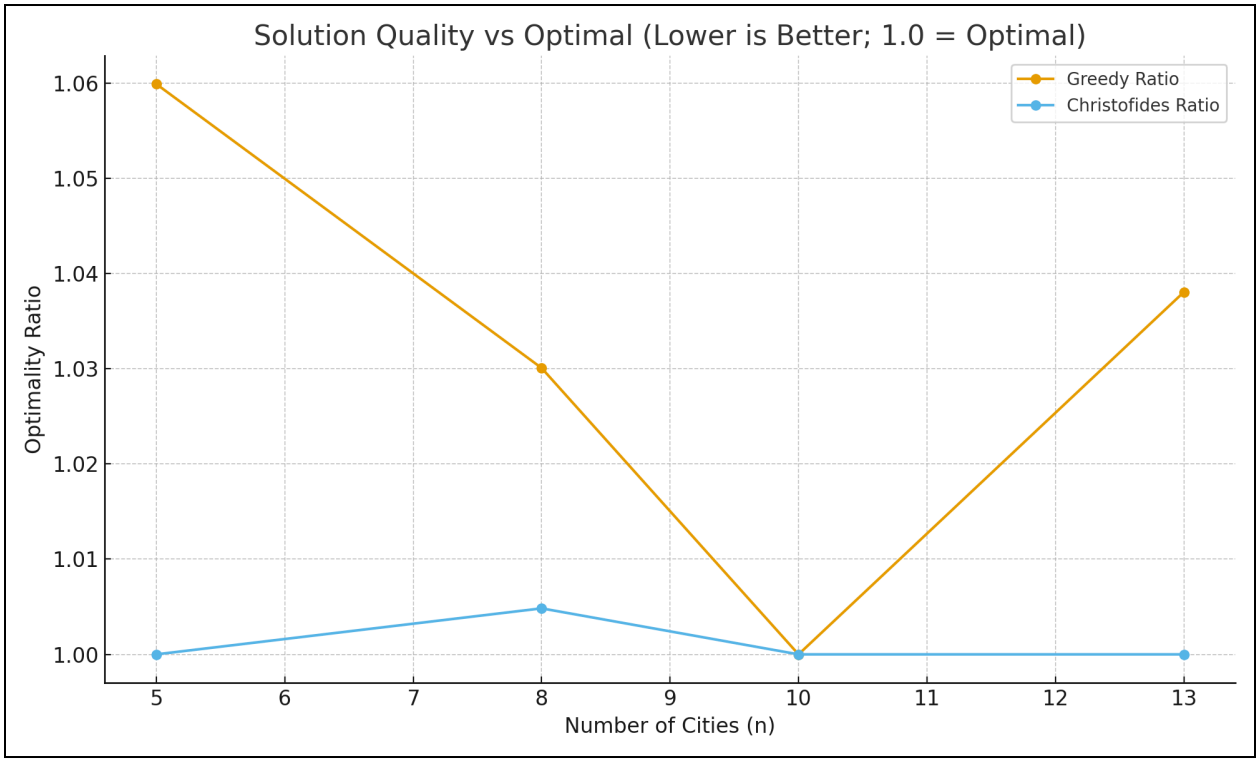
The table below summarizes the raw experimental results, including the number of cities in each dataset as well as the tour length and runtime produced by each algorithm.

n (cities)	algorithm	tour length	runtime (ms)
5	brute	1142.95	2
	greedy	1211.4	1
	christofides	1142.95	1
8	brute	1426.29	12
	greedy	1469.17	1
	christofides	1433.18	1
10	brute	1653.38	18
	greedy	1653.38	2
	christofides	1653.38	2
13	brute	1303.74	4679
	greedy	1353.34	3
	christofides	1303.74	2
20	brute	N/A	N/A
	greedy	2828.28	3
	christofides	2447.67	3
50	brute	N/A	N/A
	greedy	3913.34	6
	christofides	3987.21	7
100	brute	N/A	N/A
	greedy	4933.26	11
	christofides	4591.5	11

The following table shows the expected compute time for the brute-force algorithm for various values on n. It demonstrates how quickly the method becomes computationally infeasible as n approaches fourteen and beyond.

Cities (n)	Tours $(n-1)!$	Typical time
11	3.6 million	fractions of a second – a few sec
12	39.9 million	~4 sec to ~40 sec
13	479 million	~50 sec to ~8 min
14	6.2 billion	~10 min to ~1.7 hours
15	87 billion	~2.4 hours to ~1 day
16	1.3 trillion	days

The graph below shows how close each heuristic comes to the optimal tour. Christofides stays at or near a perfect optimality ratio, while Greedy produces slightly longer tours.



Our empirical analysis supports the expected behavior of all three algorithms. Brute force showed clear factorial growth, jumping from milliseconds at ten cities to nearly five seconds at thirteen cities, thus becoming unusable beyond that. This confirms the O(n!) complexity studied in class. Both Greedy and Christofides algorithms scaled smoothly with a roughly quadratic growth pattern, remaining below fifteen ms even at 100 cities. In terms of solution quality, Christofides consistently produced optimal or near optimal tours for all datasets where brute-force results were available (n = 5–13). Our Greedy algorithm, however, was typically 3–6% worse than optimal. For larger datasets, Christofides generally produced shorter tours than Greedy, as expected, with the exception of the fifty city case where Greedy slightly outperformed Christofides. Our study has a few limitations. The Christofides implementation used a greedy matching rather than exact minimum weight matching. We also only used randomly generated Euclidean sets, which may not reflect performance on real world datasets. If we repeated the experiment, we would add an exact matching algorithm, test more input types, and evaluate additional heuristics such as 2-opt improvements. These results suggest further questions however, including how these algorithms behave on non-Euclidean graphs, whether Christofides consistently beats Greedy, and how post-processing heuristics could further improve solution quality.

We used several resources to help us understand the Traveling Salesman Problem and the algorithms we used. Our main reference was the course textbook, which provided a background on TSP, brute force searching, greedy heuristics, and approximation algorithms. We also used AI in a few ways. First, we used AI to gather possible prompts to decide the route we would be taking for this project. We used AI to help clarify certain algorithmic steps and to guide part of the coding for the Christofides method. We also used AI for figuring out how to make the SVG files, minor debugging, and guidance on using MSYS2 for our testing process. We found general information about TSP variations through Google searches as well, although we did not keep specific links. These sources helped us better understand the theory behind the algorithms and how to implement and compare them effectively.

Not only was coding the algorithms a challenge, but coordinating everything as a group was just as difficult. It was tough finding times when everyone was available to meet since most of the project was done over Thanksgiving break. We had to allocate a lot and trust each other in our abilities to complete the work. Overall, with the timing, scheduling, and just getting all of our code to work,it was definitely harder than we initially expected.

	Member(s)
Brainstorming	Alex / 50%, Graham / 50%
Coding	Alex / 40%, Graham / 60%
Experiment design	Jimmy / 50%, Graham / 50%
Testing	Jimmy / 60%, Alex / 20%, Graham / 20%
Visualization/results	Jimmy / 50%, Alex / 50%
Communication	All
Submission	Alex / 100%
Report	Jimmy / 100%
Presentation	All