CSE 6140 - ALGORITHMS

Lukas Ott: lott7

General Framework. The purpose of this project is to investigate different ways to solve the traveling salesman problem (TSP). I chose to create a general framework in C++ that enables seamless connectivity and possibilities for extension. An abstract class `TravelingSalesmanSolver` contains the general variables and functions that are useful to process the datasets and handle the TSP. The function `TravelingSalesmanSolver::solve()` is a virtual function that needs to be implemented in the children's class of each specific solver. This interface allows to building of the three different algorithms on its own and coherence in the end product. The benefit is also seen in the main function, which is thus greatly simplified.

Brute Force. In my *Brute Force* implementation, I meticulously followed the instructions and crafted an algorithm that exhaustively explores all possible combinations for a given set of nodes. I stored the minimum path while iteratively navigating through these combinations. Recognizing the algorithm's increasing complexity with 11 cities, I developed one version tailored for solving problems with sizes $N \leq 10$ and another for $N > 10$. To heighten the chances of obtaining the best result within a short timeframe, I introduced a random shuffling of permutations. Starting from the base case [0, 1, 2, ..., N-1, N], I shuffle the permutations randomly, necessitating a random seed. It's important to note that this implementation is most effective when a cutoff time is specified. Otherwise, a pure *Brute Force* algorithm, commencing from the base case, would be sufficient.

Approximation. For the *Approximation Algorithm*, I implemented a 2-approximate solution to the TSP problem using a *Minimum Spanning Tree*. I began by building an *undirected graph* of all the city's locations as an $N \times N$ Adjacency Matrix A, storing the distances of cities. Each row or column $A[i], A[j]$ corresponds to a vector of edges $e_i = (ID_i, ID_j), e_j = (ID_j, ID_i)$, for each of the city's location IDs $ID_i = i+1, ID_j = j+1$. Using Kruskal's Algorithm to find an MST ($\mathcal{O}(N^2)$ time), I created a vector of edges to then start building an MST matrix while connecting the shortest distances without building a cycle (cut property). For ID_i, $MST[ID_i - 1] = (min_dist, ID_j)$, with ID_i being the MST predecessor.

Given this MST matrix, I can perform the *Depth First Search* algorithm starting from a chosen initial node. To find the ideal tour, I compared all possible nodes as a starting point, using the minimum path length from *Kruskal's MST* ($\mathcal{O}(N)$). Throughout the traversal, I update a vector of booleans, storing the visited nodes in order. Once the tour is completed, I add the source node to the end of the vector and compute the tour length through the initial adjacency matrix A, allowing for $\mathcal{O}(1)$ access. Thus, the algorithm runs in $\mathcal{O}(N^2)$ time.

Local Search. For my *Local Search Algorithm*, I implemented the Simulated Annealing method, which helps me explore the solution space without getting stuck in local minima. The algorithm starts by sampling a random solution S, which is randomly altered at each iteration. Two types of modifications are implemented: the first one randomly swaps 2 nodes of S, and the second one moves a single node to a random place in the tour S. Following suggestions in Ref.,[?] 90% of the modifications will be a pair swap, and the single node displacement represents the remaining 10%. If the modified tour S' has a lower cost than the tour S, then it is accepted ($S \leftarrow S'$). If not, then it is accepted with a probability

$$p = \exp\left(-\frac{c(S') - c(S)}{c(S)T}\right), \tag{1}$$

where I divide by $c(S)$ to have a unitless error that generalizes better for different problems. This probability is lower for a higher cost gap and higher for higher values of temperature T, which are parameters. The temperature exponentially decreases every 5 iterations with $T \leftarrow \alpha T$. The algorithm terminates if 1) the maximum number of iterations is reached (10,000,000), 2) the cutoff time is reached, or 3) the solution has not improved for n_{nodiff} iterations.

I tune two sets of parameters (α, T, n_{nodiff}) by hand on `UKansasState.tsp` ($N = 10$) and `Berlin.tsp` ($N = 52$). Using these, I create a linear fit for the parameters as a function of the problem size. More datasets could be used in the training for parameters, but this may lead to complicated rules for parameters that overfit the data. This linear rule allows me to mitigate the observed tradeoff in solving low-order and high-order problems. Low-order problems can afford to have faster convergence parameters (lower T, lower α) that yield good results, while high-order problems need a slower cooling process to find decent results.

Results. The runtimes and solution quality for the different datasets are presented in the table contained in the GitHub repository *results.csv*. It can be observed that the brute force only terminates when $N \leq 10$, where it finds the global optimum. For greater problem sizes, solving the problem with brute force becomes computationally intractable.

The approximation algorithm does not find the global optimum but always finds a satisfactory result in polynomial time. It is consistently the fastest algorithm, and I can observe near $\mathcal{O}(N^2)$ time complexity by looking at the runtimes between Toronto and Roanoke.

Finally, simulated annealing consistently finds the path with the lowest cost among all methods. Given that the brute force algorithm converged for `Cincinnati`, `UKansasState`, and that the solution for `Berlin` is online,[?] I can confirm that it was able to find the global optimum several times (over 5 trials) on these datasets. Simulated annealing consistently finds a path that is 10 to 20% shorter than the approximation. However, it can be several orders of magnitude slower than the approximation algorithm. Further study could determine how to improve the speed of such an algorithm, for

example, by starting SA from the approximation instead of from a random tour. Visual results of the estimated tours are available in Figs 2, 3, and 4.

Convergence Analysis. Finally, let's examine the convergence of brute force compared to simulated annealing (SA) in Fig. 1. At the beginning, both algorithms explore the solution space randomly. As the temperature decreases in SA, it starts to reject points with higher costs. Only then does the algorithm transition into the exploitation phase. This observation suggests that simulated annealing should not be stopped too early, as it may not have completed enough exploitation.

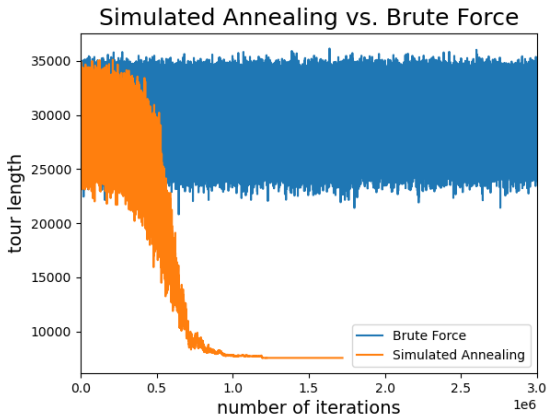

Figure 1. Quality of the solution per iteration for Berlin.

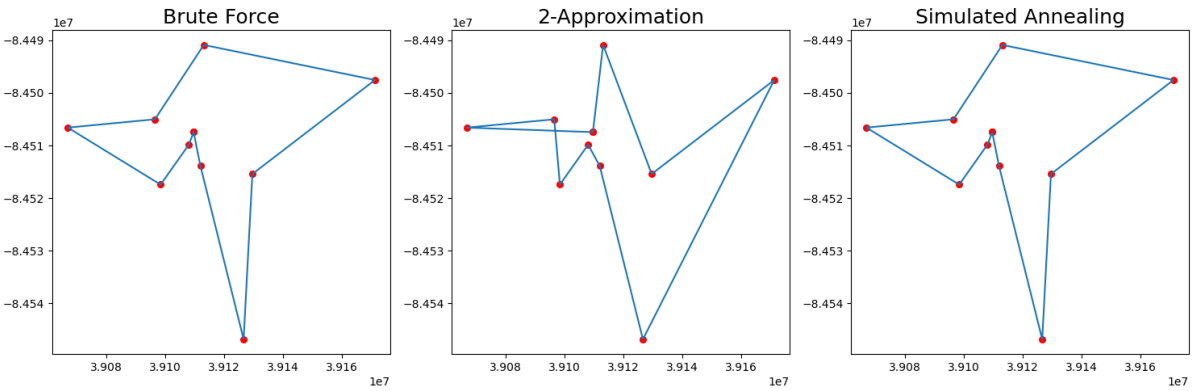

Figure 2. Estimated shortest paths for Cincinnati.

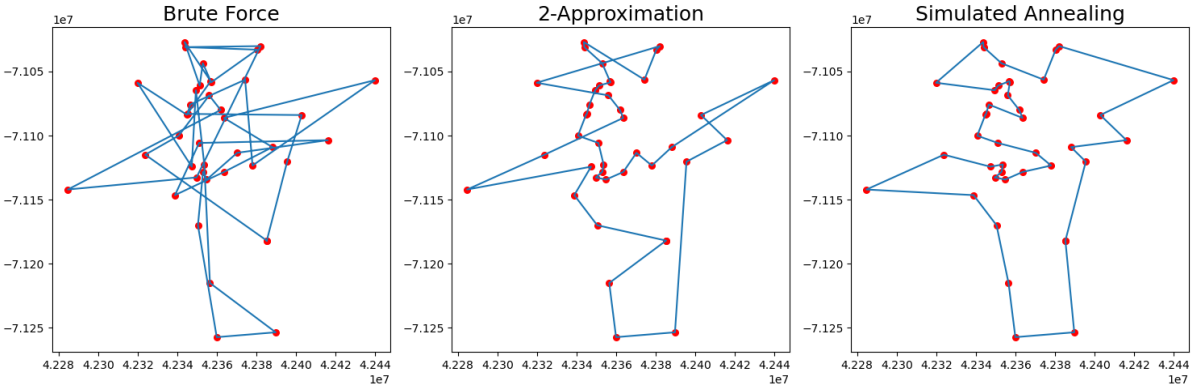

Figure 3. Estimated shortest paths for Boston.

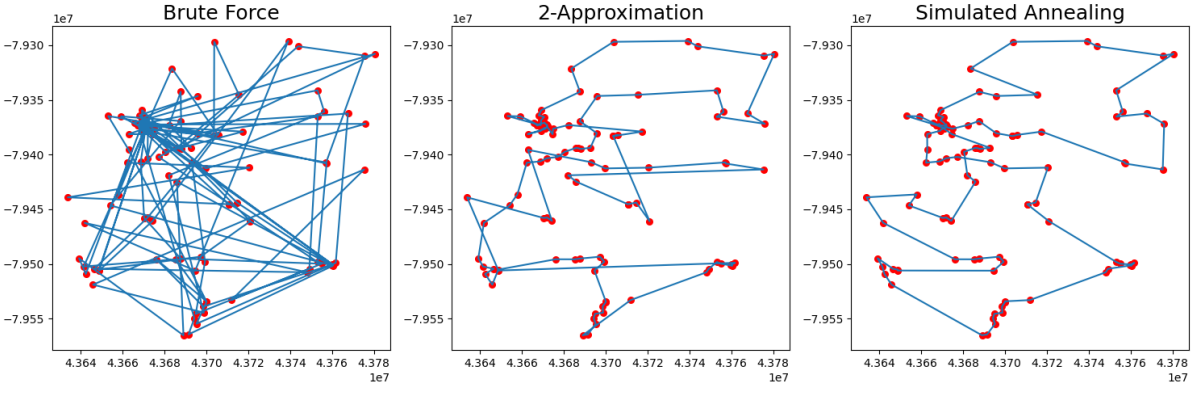

Figure 4. Estimated shortest paths for Toronto.