**Traveling Salesman Problem Approximation**

Administrative:

* Group 21
* Chloe Imbusch, Kyle Lund
* GitHub: <https://github.com/chloeimbusch/DSAProject3>
* Video:

**Proposal**

Problem Statement:

In this project, we explore and compare various algorithms that aim to approximate a solution to the traveling salesman problem. The traveling salesman problem (TSP), first formulated in 1930, is one of the most extensively studied combinatorial optimization problems in computer science and is representative of many NP complete problems. This problem, which seeks to find the shortest hamiltonian cycle that connects a series of points, is applicable to a wide variety of real-world optimization problems. These problems include obvious parallels such as logistics routing, but also problems such as microchip design and DNA sequencing. This problem, and the algorithms that approximate solutions to it, are important because TSP is believed to be NP-complete. Currently, the brute force method is the only algorithm which is guaranteed to find the optimal solution, and unfortunately the traveling salesman problem is a combinatorial optimization problem. Which means that the brute force algorithm runs in O(n!) time, which quickly becomes intractable for all but the smallest of datasets.

This inefficiency means that approximations must be used. If we relax the requirement from finding the optimal solution to finding a solution that is “good enough”, there are many approximations which run in polynomial or super-polynomial time which result in a solution that is within a constant factor of the optimal solution, at least on average.

Project Features:

This project implements three main features. First, the project is built on top of an interactive GUI which allows the user to select which of the approximation algorithms to run. The second primary component is the algorithms themselves. This project implements the following algorithms: nearest neighbor, greedy heuristic, 2-Opt, genetic, simulated annealing, and ant colony optimization. We also implemented the brute force method for comparison. Finally, we implemented a visual representation of the effectiveness of the user’s chosen algorithm. We do this in two ways. First, the best cycle that has been found is displayed. And second, we graph the distances of the best route that the algorithm has found over time, showing how the algorithm improves its solution as it iterates.

Dataset:

There are two ways to look at the data used in this project. At the surface level, we utilize the geographic coordinates of the 200 highest populated cities and towns in Florida. However, these locations are simply the individual pieces of data that are used to compose our actual dataset. The actual dataset is the sample space of all possible solutions to the traveling salesman problem for our 200 cities. The sample space for the symmetric traveling salesman problem (routes are considered the same if they are the reverse of each other) contains possible solutions. For our project, with n representing the 200 cities that we try to travel between, this results in roughly different possible solutions. To put that number somewhat in perspective, the number of Planck times (the possible unit of time) since the big bang is estimated to be around , and estimates for the number of atoms in the known universe are approximately . Multiplying these two numbers together still results in a number that is only a microscopic fraction of the size of the sample space that we attempt to explore. This combinatorial explosion is the primary issue that any potential traveling salesman problem approximation algorithm needs to overcome.

Tools:

This project is written in Python. Specifically, it requires Python 3.7 or newer, and was written using 3.10.9. There were two primary external libraries that were utilized during this project. The Pygame module was used to build the GUI for the project. And matplotlib was used to plot how the calculated distance of the algorithm improves over time.

Algorithms:

* Nearest Neighbor: This approach is a greedy algorithm that begins by randomly selecting a starting city, then iteratively finding and adding to city that is closest to the last selected city. Additionally, this implementation runs this algorithm in parallel, having 100 (by default) different agents all running the same algorithm with the goal of selecting differing starting cities.
* Greedy Heuristic:
* 2-Opt:
* Genetic: Genetic, or evolutionary, algorithms attempt to solve a wide range of optimization problems by mirroring biological processes. In this algorithm, a population of individuals (in this case representing different possible solutions) have their fitness measured and are then selected into a mating pool based on their fitness. After this, individuals are “mated”, which consists of two routes swapping sections of their route. And finally, individuals undergo a mutation process, where cities are randomly swapped in each individual.
* Simulated Annealing: This algorithm is based on the metallurgical annealing process. In practice, this algorithm stochastically changes the current route and compares it to that route. If the new route is better, it is accepted as the new route. The annealing part of this algorithm occurs if the new route is worse. In this case, the new route is accepted based on the equation . The important pieces of this equation are the ∆E, which is the difference in the fitness of the new and old solutions, and t which is the temperature. This temperature variable is decreased as the algorithm is iterated, and as a result, the worse solution is selected at a higher rate early in the process, and the better solution is more likely to be retained the longer the algorithm iterates.
* Ant Colony Optimization: This algorithm is a probabilistic algorithm which utilizes the concept of pheromone trails to approximate a solution. A number of “ants” are created, which start at a random city, then select their next city based on the strength of the pheromones and the distance to the other cities. Once each ant has selected a route, each ant deposits additional pheromones on the connections it has chosen. In practice, this is similar to the nearest neighbor approach, but with an additional layer of randomness, which allows the algorithm to better explore the solution space in search of the globally optimal solution.
* Brute Force: This algorithm iterates through every possible solution until it finds the best one.

Additional Data Structures and Algorithms:

Several of the implemented algorithms utilize additional data structures and algorithms under the hood. The nearest neighbor algorithm utilizes a series of linear searches to find the closest city. The strength of pheromones between cities in the ant colony optimization is represented by a weighted adjacency matrix.

Roles and Responsibilities:

We utilized [agile methodology](https://trello.com/b/c0Zj9Dev/dsa-project-3) to divide the workload. We had an initial meeting where the course of the project was planned out, various tasks were added to the linked tracker, after which we each claimed and worked on a task when we had time available. Over the course of the project, Chloe implemented the greedy heuristic algorithm, the 2-Opt algorithm, created the video, and provided the foundation of the workflow by maintaining the git repo and the agile storyboard. Kyle implemented the GUI in addition to the nearest neighbor, genetic, simulated annealing, ant colony optimization, and brute force algorithms.

**Analysis**

Changes:

The project mostly proceeded as initially planned. The main changes were that the planned 3-Opt algorithm was dropped due to being too similar to the 2-Opt algorithm, and additional algorithms were added (simulated annealing and ant colony optimization) simply because we found the implementations to be interesting.

Time Complexity:

* Nearest Neighbor: O(m\*n2), where n is the number of cities in the route, and m is the number of parallel nearest neighbor agents being run simultaneously.
* Greedy Heuristic:
* 2-Opt:
* Genetic: O(m\*n\*p), where m is the number of generations, n is the number of cities in the route, and p is the number of individuals in the population.
* Simulated Annealing: O(m\*n), where m is the number of operations that are run, and n is the number of cities in the route. This algorithm is complete when a better solution has not been found after a certain number of iterations, which is why the definition for m is inexact.
* Ant Colony Optimization: O(a\*m\*n2), where a is the number of simulated ants, m is the number of iterations the algorithm runs, and n is the number of cities in the route.
* Brute Force: O(n!) where n is the number of cities in the route, as discussed in the dataset section.

**Reflection**

Overall Experience:

Challenges:

One challenge that we faced was that neither of us had ever written a GUI for a project of this size before and we had a lot of early struggles with getting the various required pieces to fit together smoothly. It eventually came together, and we were able to rewrite various components so that they were more interchangeable, which made further modifications and additions simpler.

An additional challenge was that several of the algorithms, namely the genetic, ant colony optimization, and an unincluded particle swarm optimization, experienced pre-mature convergence. Roughly speaking, all these algorithms work by creating a bunch of solutions in the sample space and moving the various solutions towards the current best solution. This allows these algorithms to be useable in a wide variety of situations, but in order to be effective, these algorithms require that enough of these solutions must be utilized to effectively cover the sample space. Pre-mature convergence occurs when these solutions all gravitate towards a local minimum and become similar to each other. A general recommendation that we encountered was that the number of sample solutions (individuals in the genetic algorithm, ants in the ant colony, and particles in the particle swarm) should greatly exceed the number of cities in our route. This was somewhat possible to achieve for the computationally simpler genetic algorithm, but was not achievable in the ant colony, and was so bad in the particle swarm optimization that we chose not to include it. It appears that this is the price that these algorithms pay for their generality.

A final challenge was scope creep. While researching this problem, we encountered a significant number of interesting sounding algorithms that distracted us from focusing on the core project. In addition to the ones that we did eventually implement, we attempted to implement particle swarm optimization, and looked into state transition, tabu search, artificial bee colony, and black hole optimization.

Hindsight Changes:

Based on the challenges we encountered, we would spend more time upfront planning how the various pieces of the GUI will fit together, in order to reduce the coupling between the components. Additionally, prior to beginning to work on the various algorithms, we should have spent more time researching what requirements they have for working well. This could have saved us the time spent working on the particle swarm optimization algorithm and may have dissuaded us from attempting the ant colony optimization.

Lessons Learned:

* Chloe:
* Kyle: Learning how the various algorithms worked was a beneficial experience, both from simply experimenting with these solutions in general, and also comparing how the different algorithms approached the same problem. I’ve hopefully added a few algorithms tricks to the proverbial tool chest. The GUI implementation was also surprisingly helpful. There were a lot of pieces to fit together into a cohesive whole, and seeing what eventually worked to fit them all together is something I can take away for the next project of this size.