Project 1 Readme Team null

Version 1 9/11/24

A single copy of this template should be filled out and submitted with each project submission, regardless of the number of students on the team. It should have the name readme_"teamname"

Also change the title of this template to "Project x Readme Team xxx"

<table>
<tr><td>1</td><td>Team Name: null</td></tr>
<tr><td>2</td><td>Team members names and netids: genesis argueta (gargueta), angel ortiz (aortiz22), jack mangione (jmangion)</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects: traveling salesman - brute force, branch and bound, and nearest neighbor</td></tr>
<tr><td>4</td><td>Overall success of the project: Overall, the project was a success, despite encountering several challenges along the way. Initially, we struggled with how to approach the test cases and develop algorithms that would produce the correct output. However, through persistent effort, we were able to overcome these obstacles. Ultimately, we succeeded in ensuring the algorithms worked as intended, correctly processing the test cases and generating accurate results.</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 15 hours</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/g3n3s1s-a/project_1_TSP</td></tr>
<tr><td>7</td><td>List of included files (if you have many files of a certain type, such as test files of different sizes, list just the folder): (Add more rows as necessary). Add more rows as necessary.
<table>
<tr><th>File/folder Name</th><th>File Contents and Use</th></tr>
<tr><td colspan="2">Code Files</td></tr>
<tr><td>code</td><td>Branch_bound_null.py - computes the best path using branch and bound algorithm

Brute_force_dumb_null.py - computes the best path using brute force

Helper_funcs_null.py - helper functions that read csv files and returns an np array, also a function that calculates total distance from a given path

Nearest_neighbor_null.py - computes</td></tr>
</table>
</td></tr>
</table>

	the best path using nearest neighbor algorithm
Test Files	
test	Data used to make sure we know our algorithms are working correctly, as well as the script to test all 3 algorithms test_cases_1.csv test_cases_2.csv test_cases_3.csv test_cases_4.csv test_cases_5.csv test_script.py
Data Files	
data	These are the different data files we used in format **$num of cities_$size_$teamname**.csv 50_medium_null.csv 100_medium_null.csv 120_medium_null.csv 30_small_null.csv 10_tiny_null.csv 12_tiny_null.csv 3_tiny_null.csv 5_tiny_null.csv 7_tiny_null.csv
Plots (as needed)	
graphs	Plots for each algorithm showing the relationship between number of cities vs time it takes to compute the best path graphs/brute_force_plot_null.png graphs/branch_bound_plot_null.png graphs/nearest_neighbor_plot_null.png
Output files	
output	Files ending in matplot lib are graphs of the paths created by earth algorithm per dataset. Output files for testing the algorithms and testing against time are below.

		branch_bound_3_tiny_matplotlib_null.png branch_bound_5_tiny_matplotlib_null.png branch_bound_7_tiny__matplotlib_null.png branch_bound_10_tiny_matplotlib_null.png branch_bound_12_tiny_matplotlib_null.png brute_force_3_tiny_matplotlib_null.png brute_force_5_tiny_matplotlib_null.png brute_force_7_tiny_matplotlib_null.png brute_force_10_tiny_matplotlib_null.png nearest_neighbors_3_tiny_matplotlib_null.png nearest_neighbors_10_tiny_matplotlib_null.png nearest_neighbors_30_tiny_matplotlib_null.png nearest_neighbors_50_medium_matplotlib_null.png nearest_neighbors_100_medium_matplotlib_null.png nearest_neighbors_120_large_matplotlib_null.png output_branch_bound_null.png output_brute_force_null.png output_nearest_neighbor_null.png test_output_null.png
8	Programming languages used, and associated libraries: Python, numpy, matplotlib.pyplot,itertools,time,math	
9	Key data structures (for each sub-project): Brute Force - arrays Branch and Bound - arrays Nearest Neighbors - arrays	
10	General operation of code (for each subproject) Brute Force - It generates all possible city paths, calculates their distances, and	

	identifies the shortest route while implementing a timeout feature to prevent excessive computation time. The script processes multiple datasets of varying city counts, measures execution times, and visualizes the results through a scatter plot that distinguishes between completed runs and timeouts. Branch & Bound - It systematically explores city paths while pruning suboptimal routes, significantly reducing the number of possibilities compared to brute force. By calculating lower bounds on the shortest possible distance at each branching step, the algorithm avoids unnecessary computation of paths that are guaranteed to be longer than the best known solution. The script includes a timeout feature to prevent excessive computation time and processes datasets with varying city counts. It measures execution times and visualizes the results with a scatter plot that highlights completed runs and timeouts, providing a comparison of efficiency across different datasets. Nearest Neighbor - This algorithm constructs a path by iteratively selecting the nearest unvisited city, starting from the first one and moving to the closest neighboring city at each step until all cities are visited. It guarantees a complete tour but doesn't ensure the optimal solution. This approach is computationally efficient and performs well on small datasets but can lead to suboptimal paths on larger datasets due to its local choice heuristic.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. We used five randomly generated test cases to evaluate our code. The randomness ensured that our solution would handle various inputs and consistently produce the correct output. By examining these cases, we confirmed that the code correctly and efficiently calculated the distance, regardless of the specific input values.
12	How you managed the code development We used a github repo and each team member managed their own files. We consistently pushed and kept the repo up to date.
13	Detailed discussion of results: The results demonstrate the limitations of brute-force solutions for the TSP, particularly as the number of cities increases. The algorithm works well for small datasets but quickly becomes impractical for larger ones, reinforcing the need for more efficient methods in real-world applications. The brute force solution becomes prohibitively inefficient to compute with graphs that contain more than 10 vertices. 12 vertices are estimated to take ~10,000 seconds. The branch and bound algorithm dramatically improves on this, with 12 vertices taking just 1.6 seconds. For the 10 vertex TSP we encountered, these two algorithms found the same solution. However, we still tested an even faster algorithm, a greedy nearest-neighbor algorithm. This finished 10 vertices in >0.01 seconds. However, the tradeoff for efficiency now becomes clear: the path it finds is about twice as long as the path that the other algorithms find.

14	How team was organized: Genesis was in charge of the Brute Force algorithm, Jack was in charge of the Nearest Neighbors algorithm, and Angel was in charge of the Branch & Bound algorithm. We each worked on our algorithms separately. Genesis found the data and made a test script. The group worked collaboratively on the remaining parts of the project.
15	What you might do differently if you did the project again: If we were able to do this project again, we would try to mess around with the algorithms more and maybe make a hybrid of all the algorithms we used to see if it would make a small difference. We probably would also try to implement more complicated algorithms.
16	Any additional material: