Project 1 Readme Team Gryffindor

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: Gryffindor</td></tr>
<tr><td>2</td><td>Team members names and netids: Jack Keller, jkeller7</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects: Hamiltonian Path</td></tr>
<tr><td>4</td><td>Overall success of the project: Complete Success</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 8 hours</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/hokagejack/Hamiltonian_Path/tree/main</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><td>File/folder Name</td><td>File Contents and Use</td></tr>
<tr><td colspan="2">Code Files</td></tr>
<tr><td>simple_solver_Gryffindor.py</td><td>1. read_graphs_from_csv function: This function reads a CSV file that contains multiple graphs, extracting the vertices (v lines) and edges (e lines) of each graph, and yields each graph as a dictionary with two keys: 'v' for vertices and 'e' for edges.
2. hamiltonian_path_exists function: For each graph, this function checks if a Hamiltonian path exists. It does so by generating all possible permutations of the vertices and checking if each consecutive pair of vertices in the permutation is connected by an edge. If a valid path is found, the function returns True.
3. Main script:
- The script reads each graph from the input CSV (test_graphs.csv) and processes it individually.
- For each graph, it calculates whether a Hamiltonian path exists and measures the</td></tr>
</table>
</td></tr>
</table>

	time taken for this computation. - The results are stored in a list that contains the number of vertices, the time taken, and whether a Hamiltonian path was found (1 for yes, 0 for no). 4. **Results**: The results are written to an output CSV file (hamiltonian_path_results.csv), including the number of vertices, the time taken, and the Hamiltonian path result for each graph. The code uses itertools.permutations for brute-force checking of Hamiltonian paths.
Test Files	
data_test_graphs.csv check_1.csv check_2.csv	data_test_graphs.csv: I used the test file of 40 graphs provided on Canvas to test my code's functionality as well as graph the data. check_1.csv: testing functionality with a predetermined hamiltonian path graph check_2.csv: testing functionality with a predetermined non hamiltonian path graph
Output Files	
output_terminal_Gryffindor.txt output_hamiltonian_path_results_Gryffindor.csv	output_terminal_Gryffindor.txt: This txt file contains the entirety of the output printed to the terminal when the code is executing. output_hamiltonian_path_results_Gryffindor.csv: this is a csv file of the data collected from running data_test_graphs.csv
Plots (as needed)	
plots_hamiltonian_path_results_Gryffindor.pdf	This plot has time on the y axis and #vertices in the graph on the x axis. It plots the time taken to determine if a Hamiltonian Path exists for each graph. The green dots are graphs where a path was found, and the red dots are graphs where paths were not found. There is also an exponential curve plotted to represent the exponentially increasing time complexity that comes with searching through graphs of greater complexity (in this case # of

	vertices)
8	Programming languages used, and associated libraries: Programming Language: Python Libraries used: csv, itertools, time
9	Key data structures (for each sub-project): 1. **Dictionary** (current_graph): - Stores each graph's vertices ('v') and edges ('e'). 2. **Set** (edges): - Used to store edges as a set of tuples to enable fast lookup when checking if two vertices are connected. 3. **List** (results): ○ Stores the result for each graph, including the number of vertices, time taken, and whether a Hamiltonian path was found. 4. **Tuple** (perm): ○ Represents a permutation of vertices while checking for a Hamiltonian path.
10	General operation of code (for each subproject): **Reading Graph Data from CSV**: - The function read_graphs_from_csv reads the graph data from the CSV file test_graphs.csv. - Each graph in the file is represented by two types of lines: - 'v' lines: Represent the vertices of the graph. - 'e' lines: Represent the edges between vertices - When a new graph is detected (indicated by a 'c' line), the current graph's vertices and edges are stored in a dictionary (current_graph), which is yielded one graph at a time for further processing. **Checking for Hamiltonian Path**: - For each graph, the function hamiltonian_path_exists checks if a Hamiltonian path exists. - The function generates all permutations of vertices using itertools.permutations. A Hamiltonian path exists if every consecutive pair of vertices in any permutation is connected by an edge. - The edges are stored in a set for fast lookup, and for each permutation, the function verifies whether all consecutive vertex pairs are connected. - If a valid path is found, the function returns True; otherwise, it returns False. **Processing and Timing Each Graph**: - The script processes each graph one by one, using the generator from read_graphs_from_csv. - For each graph, the function hamiltonian_path_exists is called to check for a

	Hamiltonian path. - The time taken to perform this check is recorded using the time module, and the result (whether a Hamiltonian path was found or not) is stored. **Recording and Writing Results**: - The results for each graph (number of vertices, time taken, and whether a Hamiltonian path exists) are stored in the list results. - Once all graphs are processed, the results are written to a CSV file (hamiltonian_path_results.csv), with each row representing the outcome for one graph.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. I used two test cases (check_1.csv and check_2.csv) to ensure that the functionality of my code was complete before running the larger data_test_graphs.csv with 40 test cases. Check 1 and 2 contained a predetermined graph with a hamiltonian path and without a hamiltonian path respectively. They ensured that all my functions worked in conjunction as I worked. I would run these checks to make sure the output was correct for hamiltonian and non hamiltonian paths.
12	How you managed the code development: I started development with brainstorming a brute force method for checking hamiltonian paths. I came up with the idea to generate all the possible permutations of ordered vertices in the specified graph, and then iterate through them one by one, checking if valid edges exist to connect the permutation. If the edges did not exist, then I would check the next permutation, if they did I would stop iteration and return True. My first step was making a function that effectively parsed through a csv file containing graphs and store the vertices and edges in a dictionary data structure. I created a read_graphs_from_csv function that reads through the csv line by line and does this. My next step was to create the hamiltonian_path_exists function. I did research and found that the itertools library could automatically create all the permutations of vertices so I would not have to do this manually. Finally, I created a for loop to iterate through each graph in the csv file, and worked with some print statements to make sure the code was functional / spot errors for debugging. The last thing I added was the time library and variables to calculate the time taken for each graph, as well as a results array where I stored the results and then wrote into a csv file.
13	Detailed discussion of results: The results, especially the resulting graph “plots_hamiltonian_path_results_Gryffindor.pdf”, gave very informative insight into the exponential growth in the time complexity of NP problems. By analyzing the graphs, we

	can see that initially, when the code is checking for hamiltonian paths in graphs with vertices under 9 nodes, the time needed to confirm whether a path does or does not exist is very similar. However, as the nodes per graph increases, this difference dramatically changes. At 11 vertices, the time difference between finding a hamiltonian path and confirming the path does not exist is approximately 30 seconds. Moving onto 12 vertices, it took the code 131 and 340 seconds to determine no paths exist for two examples. This is a significant jump from the 30 seconds taken at 11 vertices. Even moving onto the graphs containing 13 and 14 vertices, the time taken to confirm a path existed was much less, presumably because the code did not have to iterate through every single permutation. Because the number of permutations grows exponentially with each additional vertex, the time to iterate through all the permutations will as well, resulting in the exponential curve we see in the graph. It is clear from these results that calculating hamiltonian paths for graphs with a much larger number of vertices would be next to impossible using this brute force method. It would simply take too much time.
14	How team was organized: As I was the only member on my project, I did not have to organize any team dynamics.
15	What you might do differently if you did the project again: Initially I wrote my code to only accept one graph per csv submission, which was inefficient when attempting to check many graphs at a time. Also, if I were to do the project again, I would add a quick check in my hamiltonian paths exist function that would automatically return false if the total number of edges is less than the total number of vertices, because this would make a hamiltonian path impossible. There is no use checking over all the permutations if this case is true
16	Any additional material: None