Project Readme Template

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"

1	Team Name: Czaplewski
2	Team members names and netids: jczaplew
3	Overall project attempted, with sub-projects: Hamiltonian Path and Cycle and Traveling Salesman Problems
4	Overall success of the project: Overall I was successful in my attempts to randomly create hamiltonian graphs and test to see if they have a path or not as well as keeping track of their execution times. I ran into minimal errors once I started and was able to complete the project in a timely manner
5	Approximately total time (in hours) to complete: ~5 hours
6	Link to github repository: [11jac11/HamiltonianPath_Czaplewski: In this repository is the contents of Jason Czaplewski's Project 1 Hamiltonian Path (github.com)]
7	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.

File/folder Name	File Contents and Use
Code Files	
HamiltonianCode_Czaplewski.py	In this file is the entirety of the code that runs the Hamiltonian path. It contains all the libraries, main, functions, as well as the output code for the text and graph. It is used to run the code and is where all of the work was actually done to create the project
Test Files	
TestFile_Czaplewski	This is a pdf file of what I used to test my code. I did not use csv files but instead randomly generated the graphs within the code. Essentially

	what is in here is snippets of codes that are used to check the variety of metrics within the main code. I did not use any outside tests since it all had to be done internally to make sure that it was testing what was randomly created. Since no csv files were used I thought that this would suffice as to how to show that I checked my code internally every time that it runs since it creates the graphs on its own.
Output Files	
TextOutput_Czaplewski	This is a pdf file that shows the output text files that you receive when you run the code. These are different every time so I simply picked a random execution and inserted it into the document. It tells you the size of the vertex and the average run time for that size. It also gives you one of the possibly many Hamiltonian cycles found when it ran that execution.
Plots (as needed)	
GraphOutput_Czaplewski	This is a pdf showing the graph you are given when you run the code. SI]ince the graphs are randomized each time I picked an execution and put it in this file. It puts a green dot where the Execution time for each point is at each of the vertices (4-10). It then draws a blue line through the vertices showing the average execution time for the runs and you can see the exponential growth as the vertex number gets higher.

8	Programming languages used, and associated libraries: Python is the only programming language used in this project in order to complete the requirements. Libraries used are: networkx: Used to create random graphs and manipulate graph structures. numpy: Used for numerical operations such as averaging and converting graphs to adjacency matrices. itertools: Used for generating permutations time: Used for measuring the execution time of Hamiltonian cycle finding. matplotlib: Used for plotting the performance results (execution times)

9	Key data structures (for each sub-project):
	I only did the main problem however the data structures that I used are:
	
	Adjacency Matrix (2D NumPy Array):

	The graph is represented as an adjacency matrix. This is generated using networkx's random graph function and converted to a NumPy array. Permutations (from itertools.permutations): The algorithm generates all possible vertex orderings (permutations), which are checked to see if they form a Hamiltonian cycle. Execution Time Lists: avg_execution_times stores the average execution time for each graph size. all_execution_times stores the execution time for each trial across all graph sizes.
10	General operation of code (for each subproject): Random Graph Generation: For each graph size from 4 to 10 vertices, 50 random graphs are generated using nx.gnp_random_graph. The probability of an edge between any two vertices is set to 0.5 which ensures randomness. Each graph is then converted to an adjacency matrix. Hamiltonian Cycle Search: For each graph the code checks if any permutation of vertices forms a Hamiltonian cycle. This is a brute force search and tests all possible vertex orders (n! possibilities for n vertices). If a Hamiltonian cycle is found it is stored in cycles_found. Execution Time Measurement:- For each graph the execution time of the Hamiltonian cycle search is recorded. This is done using Python's time.perf_counter() (looked up the best way to time execution). Average execution times per graph size are calculated and stored. Performance Plotting: The execution times (both individual and average) are plotted against the number of vertices, providing insight into how performance scales as the graph size increases.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. Random Graph Generation: The function random_graph(num_vertices, edge_probability=0.5) generates random graphs of different sizes. This is essentially creating test cases with varying numbers of vertices to check the Hamiltonian cycle detection. Hamiltonian Cycle Search: The function find_hamiltonian_cycle(graph) is called within the main() function to check if a Hamiltonian cycle exists in each randomly generated graph. It uses brute force by checking all permutations of vertices to see if any form a Hamiltonian cycle. Multiple Trials per Graph Size: The code performs 50 trials per graph size to gather sufficient data. This serves as a repeated test for each graph size, ensuring that the results are statistically significant rather than an outcome from a single graph instance. Execution Time Measurement: The execution time for each test (i.e., finding a

	Hamiltonian cycle in a random graph) is recorded using:
	`start_time = time.perf_counter()`
	`exec_time = time.perf_counter() - start_time`
	This measures the performance of the cycle detection algorithm, effectively testing its efficiency with different graph sizes.
	
	Cycle Detection Output: The Hamiltonian cycles found (if any) are stored and printed, which allows you to verify whether the algorithm correctly identifies valid cycles in the random graphs.

12	How you managed the code development: When starting the code I broke it down into a few core components; random graph generation, Hamiltonian cycle detection, performance measurement, and data visualization. I initially worked on generating random graphs using the NetworkX library and implementing the brute-force algorithm to check for Hamiltonian cycles by examining all vertex permutations. After making sure they were correct with small manually constructed graphs, I then started timing the execution using the high precision timer (time.perf_counter()). This allowed me to accurately measure the execution times as the vertex sizes increased. To ensure the results were accurate, multiple trials were run for each graph size. I then averaged the results to get rid of the variability introduced by random graph generation. Once the core functionality was in place I used Matplotlib to plot individual and average execution times. Finally, I ran multiple trials across different graph sizes in order to be able to run the code in an effective manner but still see the exponential growth in the graph.
13	Detailed discussion of results: The results from the code shows the performance and behavior of Hamiltonian cycle detection in random graphs. The brute force approach creates an exponential growth in execution time as the number of vertices increases. This was expected due to the factorial time complexity of the algorithm, where for a graph with n vertices, n! permutations are checked. For small graph sizes (4–6 vertices) the code performed efficiently and would frequently find Hamiltonian cycles when they existed within a fraction of a second. However, as the graph size grew beyond 7 vertices the execution times increased dramatically. At size 10 the average execution time for finding a Hamiltonian cycle often reached several seconds with some instances taking longer due to the increased number of possible vertex arrangements. The trials showed that not every randomly generated graph contained a Hamiltonian cycle. In smaller graphs cycles were found more frequently due to the higher probability that every vertex is connected. For larger graphs the random generation of edges with a probability of 0.5 led to sparse graphs at times, reducing the likelihood of finding a Hamiltonian cycle. In terms of execution time the scatter plot of individual execution times showed variability in the time taken across different trials for each graph size, while the line plot of average execution times highlighted the sharp increase in time required as graph size increased. Overall, the results emphasize the computational difficulty of the Hamiltonian cycle problem, especially as graph size grows.
14	How the team was organized: I decided to work with myself since I figured it would be easier if I didn't have to divide

	up the work for the code so that I would know what all the different aspects of the code do.
15	What you might do differently if you did the project again: I would definitely decide to work in a team. I think I struggled early on deciding what to pick and how to attack the problem since I did not have anyone else's opinions. I also probably would not have randomly generated the hamiltonian paths in the future I would have just used a csv file. I thought it would be cool to create the code so that it could entirely run on its own but this limits me from being able to use any outside input of a certain graph that you specifically want to run through the code to be tested.
16	Any additional material: Issues: When running the code I kept encountering zero execution time for many of the random iterations of the trials even at high vertices. I presumed that these were the non hamiltonian paths being graphed however when I tried to go back into the code and remove them it would mess up the randomization. Therefore, on the graph you see the two different y intercepts. The one on the left is the execution time for each point and the one on the right is the average execution time. I broke these up so that you can visually see the linear growth because without that the slope did not increase in an expected manner.