Project 1 Readme Team hflick

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: hflick
2	Team members names and netids: Hunter Flick (hflick)
3	Overall project attempted, with sub-projects: DumbSAT for Hamiltonian Paths
4	Overall success of the project: Successfully checked graphs up to eleven nodes
5	Approximately total time (in hours) to complete: 12 hours
6	Link to github repository: https://github.com/huntflick/TOC-Hamiltonian-Paths-hflick
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). Add more rows as necessary.

File/folder Name	File Contents and Use
Code Files	
hamiltonianPathDumbSAT_hflick.py	Hamiltonian path checker
Test Files	
dataCheckHamiltonianPath_hflick.cnf	Contains graphs in cnf form including whether each graph is Hamiltonian or not Used for timing and validation
Output Files	
outputHamiltonianPath_hflick.txt	Contains an output for

	each graph and whether it contains a Hamiltonian path
Plots (as needed)	
plotHamiltonianPath_hflick.png	A plot of times for each graph

8	Programming languages used, and associated libraries: Language: Python Libraries: itertools (permutations), csv, time, sys, matplotlib (pyplot, patches)
9	Key data structures (for each sub-project): The key data structures I use are dictionaries, lists, and tuples. The graphs are stored in dictionaries, where the keys represent nodes. The values are lists that contain every node to which the key node is connected. The timing data is stored in a dictionary where the key represents the number of nodes in the graph. The values are lists that contain tuples, each of which contains the time to compute the graph and whether or not there was a Hamiltonian path.
10	General operation of code (for each subproject) The program begins by determining what input file to use by either reading an argument or using the hardcoded name. Then, it parses the cnf file into a dictionary, which represents the graph by illustrating the connectivity of each node. The program then calculates all the permutations of nodes, regardless of whether an edge exists or not. Each path is checked one at a time. As a node is visited, it is removed from a list and the path. If the list of nodes and path are both empty at the same time, then the program has found a Hamiltonian path, which it then returns. If a Hamiltonian path is found, the program stops checking paths and outputs a success message. If not, then the program checks every path and outputs a failure message. The time to compute the graph is appended to a list that is contained in a dictionary organized by the number of nodes in a graph. This process repeats for every graph in the input file. Once every graph is computed, the time to compute each graph is plotted and the maximum times are connected via a line. The times and success of each graph are printed for the user to check the program's validity.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. I included a modified version of the provided cnf file for Hamiltonian paths from the Canvas page. This file contained a large sample size for testing and included information about what graphs contained Hamiltonian paths. I was able to use this information to check my program as well as time the performance. I modified the file by adding a line between each graph description for clarity. I also removed any graphs above eleven nodes, as my program ran for upwards of thirty minutes on a twelve-node graph without checking a single graph. To ensure I had a good curve, I modified certain graphs to ensure every number of nodes had at least one non-Hamiltonian graph.
12	How you managed the code development

	I handled the code development incrementally. I began by making the path generator and the recursive Hamiltonian checker as they are the most important parts of the program. Once it was working with a hardcoded graph, I expanded to include timing and printed outputs. After verifying the outputs again, I implemented the ability to parse a cnf file and command line arguments. Finally, I added the capability to create a plot of the results. Testing was conducted using a modified version of the provided input file on Canvas.
13	Detailed discussion of results: The time taken to check a graph increased as the number of nodes increased. This is visible when three nodes as the time for a non-Hamiltonian graph was three to four microseconds longer than previously. This trend increases with every additional node added, eventually ending with ten nodes taking approximately 3.3 seconds and eleven nodes taking 39 seconds. Based on the shape of the graph, the increasing rate of change, and the increase in number of paths as nodes increase, the time complexity of the program is likely factorial (O(n!)). This is because the possible number of paths is the factorial of the number of nodes. A non-Hamiltonian graph will cause the program to test all paths, dramatically increasing the number of operations which fits with the prediction of O(n!).
14	How team was organized I completed the project on my own and was in charge of all development and testing.
15	What you might do differently if you did the project again I would carefully read the prompt multiple times to reduce the time I spent rewriting code to handle different requirements I missed on the first read.
16	Any additional material: N/A