Readme_hflick

Version 1 10/17/24

1. Team name:
   hflick
2. Names of all team members:
   Hunter Flick
3. Link to github repository:
   https://github.com/huntflick/TOC-Hamiltonian-Paths-hflick
4. Which project options were attempted:
   DumbSAT for Hamiltonian Paths
5. Approximately total time spent on project:
   12 hours
6. The language you used, and a list of libraries you invoked
   Language: Python
   Libraries: itertools (permutations), csv, time, sys, matplotlib (pyplot, patches)
7. How would a TA run your program (did you provide a script to run a test case?)
   A TA can run the program by entering *python3 HamiltonianPathChecker_hflick.py* into the command line. A test file can be included in the command as an argument, but it must be in the same cnf format as the input file I included. If no file is specified, the program uses the provided input file. The output looks best when captured in an output file instead of the terminal.
8. A brief description of the key data structures you used, and how the program functioned.
   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.
   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.

9. A discussion as to what test cases you added and why you decided to add them (what did they tell you about the correctness of your code). Where did the data come from? (course website, handcrafted, a data generator, other)

    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.

10. An analysis of the results, such as if timings were called for, which plots showed what? What was the approximate complexity of your program?

    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!).

11. A description of how you managed the code development and testing.

    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.

12. Did you do any extra programs, or attempted any extra test cases

    I did not do any extra programs or test cases.