Project 1 Readme Team JT-JR

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: JT-JR
2	Team members names and netids: Joshua Tighe – jtighe3 Jack Roberts – jrober27
3	Overall project attempted, with sub-projects: Graph coloring – brute force and backtracking
4	Overall success of the project: We completed all components of the project successfully. We implemented the brute force and backtracking functions and a plotting function.
5	Approximately total time (in hours) to complete: 10
6	Link to github repository: https://github.com/joshuatighe/Project1-TOC.git
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	
Here are the code files that we edited/created (does not include starter code): - graph_coloring.py - plots_JT-JR.py	The graph_coloring.py file contains the finalized brute force and backtracking functions. The plots_JT-JR.py file contains the plotting algorithm. Use uv run main.py to implement

	the graph_coloring.py file. Go into the src folder in terminal and enter uv run plots_JT-JR.py to generate the graph output. We edited the return value of our graphing functions to be of the form Tuple[bool, Optional[List[int]]], instead of Tuple[bool, Optional[List[int, int]]], as we found easier to just return a list, where the list index corresponds to the vertex, and the value at that index being the color assigned to said vertex. We found it made our code neater as the expected output only required the color assignments.
Test Files	
graph_input_JT-JR.cnf	This is a test file containing several different graph implementations. As we developed our code, we used uv run main.py to access these test cases.
Output Files	
brute_force_graph_input_JT-JR_graph_coloring_results.csv btracking_graph_input_JT-JR_graph_coloring_results.csv	These files are the output files generated by executing the code and associated test cases.
Plots (as needed)	
graph_input_graph_coloring_results_bf_JT-JR.png	These files are the graphs generated by running the

	graph_input_graph_coloring_results_bt_JT-JR.png — plots_JT-JR.py file.
8	Programming languages used, and associated libraries: Python: used the itertools, typing, and matplotlib libraries
9	Key data structures (for each sub-project): Brute force: used lists, dictionaries, and tuples Backtracking: used lists, dictionaries, and tuples
10	General operation of code (for each subproject) Brute force: First, the algorithm adds all vertices to a list by utilizing the edges list, which is a list of tuples. Then, the algorithm tries assigning all possible color combinations to the vertices. For each attempt, a dictionary (called “attempt”) is created, which maps each vertex to a corresponding color. For each item in the attempt dictionary, the inputted edges list is accessed. If any of the tuples within the edges list have been assigned the same color, the attempt fails and the next attempt is generated. If a mapping works, a return list (called return_list) is generated that outputs the color of each vertex. A tuple of (True, return_list) is outputted. If all attempts are tried and no valid mapping was found, a tuple of (False, []) is outputted. Back tracking: An adjacency list of vertices is created initially by defining a dict with each vertex (key) pointing to an empty set (value). The adjacency list is populating by looping through all edges and appending connections to the corresponding vertices in the dict. For coloring assignments, an list of length n (where n is the number of vertices) is defined with each value initially being -1, meaning no color assignment yet. The index of this array corresponds to the respective vertex - 1 (as we are using 0-based indexing). We then define our backtrack function where it does the following: - Checks if we have a complete solution by comparing the current vertex number to the number of vertices - For the current vertex, we loop through all possible colors and check if they are valid assignments (i.e. no neighbors of the vertex have the same color). - If the current color is a valid assignment for the current vertex, we assign that color and and recursively call backtrack for the next vertex, if the next vertex cannot be assigned a color we backtrack and remove the current vertex’s assignment and keep looping through other colors We then check if there is a valid coloring solution by calling our backtrack function on vertex 0, if it backtrack returns true, it has found a solution, and so we return True and our coloring solution list Else, we return false and an empty list to indicate no valid solution was found

11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. We did not use pytest or other Python testing libraries in our code development. Instead, we created other test cases to verify the accuracy of our outputs and our graphs. Our test cases are found in the graph_input_JT-JR.cnf file. This file contains several different test graphs, which test edge cases and verify the accuracy of our results.
12	How you managed the code development We developed our code over the course of about a week. On the first day of development, we spent about an hour reading over the project description and associated files and began understanding the functions and interactivity of the starter code. We then divided the work: Josh was responsible for the backtracking function, and Jack was responsible for the brute force function. We both contributed to the graphing algorithm, and Josh created the test cases.
13	Detailed discussion of results: Overall, our results were as expected. The code worked as intended, with incompatible graphs generating no output and compatible graphs generating a valid output. The brute force algorithm demonstrated exponential growth in solve time as the number of vertices increased. Backtracking demonstrated non-exponential growth, which was to be expected. Backtracking only grows exponentially in the worst-case scenario.
14	How team was organized We divided up responsibilities evenly between the two of us. Throughout the course of the past week, we met several times to discuss progress, and we also coded alongside each other. Josh was primarily responsible for the development of the backtracking algorithm and the test cases. Jack was primarily responsible for the development of the brute force algorithm and the README file. We both contributed to the graphing function.
15	What you might do differently if you did the project again If doing this problem again, we would work to develop a very strong understanding of the starter code before coding anything. It took us a while to fully grasp the interactivity of the files within the starter code. Additionally, we would like to create a basic GUI to demonstrate valid graph outputs. In future iterations of the project, we would also like to explore other NP problems aside from graph coloring.
16	Any additional material: **Citations:** - Jack - I used the itertools.product() function in the brute force algorithm to generate the Cartesian product of the vertices and corresponding colors. I then created a dictionary with the help of the zip function. I used

	ChatGPT and Google to help me research itertools.product() and zip in order to most efficiently create all (vertex, color) combinations. - Josh - To implement the backtracking assignment, I adapted the following pseudocode https://codeiiest-dev.github.io/Algorithms/Backtracking/Pseudocode/Pseudocode.html to work with our implementation, as I was not too familiar with the backtracking algorithm before this project. - I also made reference to the matplotlib documentation which we used to implement our plots https://matplotlib.org/stable/api/index