Roan Urquhart
COMP 560
730090295

Assignment 1

Implementation:

Input Parsing

The code used to take input from the shell exists in two locations. The python script execution logic is within the **mapcolor.py** module. That modules imports the **utilityfuncs.py** module and uses the **parse_input(data)** function to store the data located in the .txt file for later use in the script.

Backtracking Search

All of the core logic of the backtracking search exists in the **backtrack.py** module. There are a variety of helper functions which can be more easily understood through the comment documentation within the file, but the most important functions for the search are **backtrack_search()** and **color_states(state)**. The strategy I used for backtracking search is a type of recursive filtering and forward checking. I will first give a high level overview of the logic here, more explicit detail on each function can be found in the comments within the source code:

Brief Overview

The **backtrack_search()** function is used to set up the search and make the first call to **color_states(state)**. The recursion technique that **color_states(state)** uses resembles a depth first search. On each call, a color is assigned to the state from a filtered list of colors, and **color_states(state)** is called on the “parent’s” next connected state. If the assignment of colors was successful on the first path of the depth first search, **color_states(state)** would return true and the search continues with the other states that are connected to the “parent”. As soon as the filtered list is empty and there are no more possible colors to assign, the function returns false, resets the state’s filtered list of colors and returns to its “parent”. All colors that would violate the rule are removed from the list as violations are found.

This search strategy is effective because it uses both filtering and forward checking to eliminate false positives when searching for a solution. The possible color assignments for each state are forward checked with the recursive depth first search to make sure that the assignment does not cause problems with connected states further in the search. In addition, after each depth search is completed the list of possible colors for a state are filtered for all future depth search paths. This limits the amount of color options that need to be check, which makes the search faster. The amount of nodes that are searched during an execution of my implementation ranges from 48 – 65 nodes on average.

Local Search

All of the core logic of the backtracking search exists in the **local.py** module. There are a variety of helper functions which can be easily understood through the comment documentation within the file, but the most important functions for the search are **local_search()** and **resolve_violations(state)**. The strategy I used for local search is to establish multiple starting positions to give the function a better chance at finding a solution. I will first give a high level overview of the logic here, more explicit detail on each function can be found in the comments within the source code:

Brief Overview

The **local_search()** function has three main parts. The first section calls several helper functions to set up the environment for the local search. During this set up, each state is assigned a random color so that local search can be used to find a solution. The second section is the iterative local search. A global variable **num_violations** tracks the number of violations that exist within the current assignment of colors to the states. This iterative local search runs until there are either no more violations or a set timeout has been reached (initially set to be 15 seconds). The final section deals with print formatting.

One of the most important parts of the local search I implemented is the order with which the nodes are traversed. Within the helper function **init_heap()**, I create a max heap where each state is ordered by how many color violations they have with the surrounding states. I convert this to a list with which to iterate through during the local search. This serves the purpose of having multiple starting locations with which to search. Within the iterative local search section of **local_search()**, the **resolve_violations(state)** function is used. This goal of this function is to resolve the most violations while minimizing the amount of new violations created. It analyzes the surrounding states colors, if there is no way to resolve all of the violations, it chooses to reassign the color to the least used color among the surrounding states.

This search strategy is effective because it utilizes many different starting locations. These starting locations are also ordered by how many violations each state has. I implemented the search in this way for two reasons. I ordered the states by their violations so that the states with the most violations would be evaluated first, which will reduce the greatest number of violations at each iteration. I decided to create many starting positions to avoid situations where I reach a local maximum during the search. This means that my search consistently makes less changes and fails less than a normal local search. The number of changes made by my implementation of local search ranges from 28 – 200 on average.

Contribution:

I, Roan Urquhart, worked alone on this project so I wrote all of the code by myself.