Brandon Nguyen
brandonn@sfu.ca
Student #: 301219561

CMPT 310 – Assignment 2 Written Document

The general architecture of my program is as follows:

- I have a **solve()** function that returns either a proper colouring or an empty list if no colouring is possibly
- solve() calls a recursive algorithm called **recursiveBacktrackSearch()** which starts with an empty solution = []
    1. checks if the length of our current solution is finished, if it is return, else continue
    2. finds a node to colour, using **MRV** with **DH** as a tie-breaker
    3. finds a colour for that node, using **LCV**
    4. if a legal colour can be assigned then
        - adds (node, colour) to our solution
        - recursively call recursiveBacktrackSearch() with our current solution
    5. if no legal colour can be assigned then
        - return an empty list
- recursiveBacktrackSearch() uses helper functions to make the code more readable, modular, and reusable:
    1. **MRV – Minimum Remaining Values heuristic**
        - Picks the node with the fewest legal moves available
        - Uses a helper function called **getMoves**()
            - **getMoves()** returns a set of legal colours for a given node
            - We store the number of legal colours for each remaining node in a list, and pick the smallest one as our next node to colour
    2. **DH – Degree heuristic**
        - Tie-breaker for MRV, picks the node with the most constraints on neighbouring unassigned nodes
        - Given a list of nodes with the same number of legal moves
            - Get the degrees of each of these nodes
            - Pick the node with the biggest degree, i.e. the node with the most constraints on neighbouring unassigned nodes
        - Uses a helper function called **getDeg()**
            - **getDeg()** returns an integer representing the number of unassigned nodes adjacent to some target node
    3. **LCV – Least Constraining Value**
        - Given a node to colour, picks a colour that is least constraining on neighbouring unassigned nodes
        - Also uses getMoves() as a helper function
        - Iterate through legal colours for a given node and choose the colour that allows for the most number of colourings for remaining unassigned nodes
    4. getCol() – is a function that may be found all over my program
        - checks if a node is in our solution, returning its colour if it is or False if not

Brandon Nguyen
brandonn@sfu.ca
Student #: 301219561

I based my implementation of backtrack search on the algorithm found in the Constraints lecture slides for our course. All the other heuristics implementations, such as MRV, DH, and LCV, were derived by myself but based on the ideas provided in lecture.

I ran tests on my program using both the small example found in the assignment description and the larger example found in the test data file provided to us. The tests were successful for both examples and I'm confident that my program will be robust with any other larger inputs as well.

While my program is well commented and should be easy to follow, I feel like some refactoring could do it good. Adding consistency to variable names and tuple forms should make my code easier to understand.

A common strategy that I reused a couple of times to find a count of something was to start out with a full list and subtract from that list. I did this as opposed to adding to an empty list as I felt it was easier but in retrospect either way would have been fine.

Overall, I feel like my program is solid and complete, but I did not use any clever tricks or abstract ideas. My algorithm is very straight forward and intuitive.

Upon observing time for execution of the algorithm, I found some of my results were unexpected. Execution with the added heuristics was actually slower than without the added heuristics. I'm not sure why this is, given that my heuristic functions seem to be correct.

Time w/ no heuristics: 0.0000441 seconds
Time w/ MRV: 0.0001610 seconds
Time w/ MRV+DH: 0.0002141 seconds
Time w/ all heuristics: 0.0003493 seconds

Using the time library to time my program:

```
Import time
Start_time = time.time()
Solve(n,k,g)
print("--- %s seconds ---" % (time.time() - start_time))
print(solution)
```