Matthew Ragoza
mtr22@pitt.edu

Assignment 1

ISSP 2160
September 15, 2021

Problem 1. Map coloring problem

a. The map coloring problem can be formulated as a search problem as follows:

- The *initial state* is the map with every country assigned a color randomly or naively, i.e. without checking the colors of bordering countries.
- The *operators* are changing the color of any country to a different color.
- The *goal condition* is when no two countries sharing a border have the same color.
- The *search space* is every possible map where each country has an assigned color.

b. Let n be the number of colors and let k be the number of countries. Since each of the k countries is assigned one of n colors, the *search space size* is n^k. In the provided example, $n = 3$ and $k = 9$, so the search space size is $19,683$.

c. An alternative formulation of the search problem is as follows:

- The *initial state* is the map with no colors assigned to any country.
- The *operators* are assigning colors to the countries one-by-one in a predetermined order, e.g. alphabetically.
- The *goal condition* is when every country has been assigned a color and no two countries that share a border have the same color.
- The *search space* is every possible map where the first i countries in the predetermined sequence have an assigned color, for each i in the range $[0, k]$.

Let S_i be the number of map states where the first i countries in the sequence have been assigned a color. Since we assign each country a color one by one, the total search space size is $\sum_{i=0}^{k} S_i$. There is only one state with no colors assigned, so $S_0 = 1$. At each step, we select from n colors and assign it to the next country in the predetermined order, so we define $S_{i+1} = S_i n$. From this recursive definition, we obtain $S_i = n^i$. Therefore, the *search space size* is $\sum_{i=0}^{k} n^i$. Given $n = 3$ and $k = 9$, this comes out to 29,524.

This is almost 50% larger than the search space size from Part b., so it is a less efficient way of setting up the problem. There are ways to make it more efficient by incorporating knowledge about the problem. For instance, when choosing colors for a country, we could select only from the colors that are not already assigned to a bordering country, which prunes the search tree to avoid exploring bad states. However, this improvement could be applied to either of the two formulations, so it still appears that the first formulation is more advantageous.

d. Figure 1 shows one possible solution to the provided map coloring problem.

Assignment 1

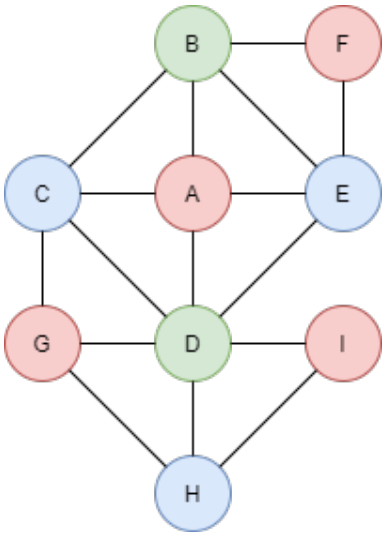

Figure 1: A solution to the map coloring problem.

Problem 2. Traveler problem

a. $S \to A \to B \to C \to D \to S \to C \to S \to A \to B$

b. $S \to B \to C \to F \to H \to E \to G$

c. $S \to A \to B \to C \to D \to C \to B \to D \to E \to F$

d. $S \to A \to B \to C \to D \to E \to F \to G$

Problem 3. A problem-solving agent for the 8-puzzle problem

a. Ran `bfs.py` successfully.

b. Implemented `bfs_stats.py`.

c. Implemented `bfs_cycles.py`.

d. Implemented `bfs_repeats.py`.

e. Out of the three breadth-first search (BFS) algorithms tested, `bfs_stats.py` performed the worst. It had the greatest number of expanded nodes, number of generated nodes, and queue length on each of Examples 1-4. In addition, it failed to find a solution to Example 5 in a tractable amount of time. It performed especially poorly on Example 4, where it generated over 87,000 nodes and reached a maximum queue length of over 55,000, as seen in Figure 5. This is an order of magnitude more than either of the other two BFS algorithms. This makes sense given that the algorithm performed no checks

for repeated states, so a significant portion of its time was spent exploring the same states repeatedly. Despite its poor search efficiency, `bfs_stats.py` found the optimal solution on each of the examples on which it finished.

The BFS algorithms that checked for cyclic repeated states (`bfs_cycles.py`) or all repeated states (`bfs_repeats.py`) were more efficient at solving the puzzle. They were also the only algorithms tested that found the optimal solution to all puzzles, including Example 5. They performed identically on Examples 1 and 2, but `bfs_repeats.py` had an increasingly large edge on Examples 3-5. For instance, it generated 10 fewer nodes than `bfs_cycles.py` on Example 3, 248 fewer nodes on Example 4, and 75,826 fewer nodes on Example 5, as seen in Figures 4, 5, and 6. On these harder problems, this is an indication of the degree to which `bfs_cycles.py` searched through non-cyclic repeated states, which were avoided by `bfs_repeats.py`. Since the first time that BFS encounters a state is guaranteed to be the shallowest occurrence of that state in the search tree, `bfs_repeats.py` finds the optimal solution in less time by eliminating sub-optimal paths. Therefore, it was the best BFS search algorithm overall.

f. The depth-limited depth-first search (DFS) algorthim, `dfs_limit.py`, used far less memory than any of the BFS algorithms. This can be seen in its maximum queue length, which was never more than 17 on any of the problems it successfully solved. This reflects the lower space complexity of DFS, which is linear in the maximum depth, compared with BFS, which is exponential in the solution depth. In terms of number of generated and expanded nodes, the DFS algorithm performed better than BFS with no repeat checking, but worse than BFS with repeat checking, on average. However, it failed to find a solution on Examples 4 and 5, due to their optimal solutions occurring at depth 11 and 19, respectively. This was beyond the maximum depth limit, which was set to 10.

   One interesting observation is that `dfs_limit.py` performed better on Example 2 than Example 1, despite that it was a harder problem in terms of optimal solution length. This reflects how depth-first search can occasionally get lucky by choosing a good branch and finding a solution relatively quickly, but it can also get unlucky by searching very deeply on a bad search bramch. On the other hand, breadth-first search is guaranteed to find the optimal solution if it exists. But by searching the space evenly, BFS spends a lot of time trying branches that will not turn out to be solutions.

	# expanded	# generated	Max. queue length	Solution length
`bfs_stats.py`	29	81	53	4
`bfs_cycles.py`	16	44	22	4
`bfs_repeats.py`	16	44	22	4
`dfs_limit.py`	164	467	17	4

Figure 2: Search statistics for Example 1.

	# expanded	# generated	Max. queue length	Solution length
`bfs_stats.py`	500	1,382	883	7
`bfs_cycles.py`	84	234	105	7
`bfs_repeats.py`	84	234	105	7
`dfs_limit.py`	106	274	16	7

Figure 3: Search statistics for Example 2.

	# expanded	# generated	Max. queue length	Solution length
`bfs_stats.py`	1,962	5,622	3,661	9
`bfs_cycles.py`	176	500	226	9
`bfs_repeats.py`	173	490	216	9
`dfs_limit.py`	692	1,863	17	9

Figure 4: Search statistics for Example 3.

	# expanded	# generated	Max. queue length	Solution length
`bfs_stats.py`	31,658	87,542	55,885	11
`bfs_cycles.py`	808	2,198	934	11
`bfs_repeats.py`	723	1,950	775	11
`dfs_limit.py`				

Figure 5: Search statistics for Example 4.

	# expanded	# generated	Max. queue length	Solution length
`bfs_stats.py`				
`bfs_cycles.py`	59,524	160,638	67,175	19
`bfs_repeats.py`	31,591	84,812	30,611	19
`dfs_limit.py`				

Figure 6: Search statistics for Example 5.