University of Pittsburgh
CS 2710 Foundations of Artificial Intelligence — Handout 1
Professor Milos Hauskrecht — August 27, 2020

Problem assignment 1

Due: Thursday, September 3, 2020

Please note that homework assignments should be submitted in the electronic form at 4:15pm on the due date before regularly scheduled class. This concerns both the reports and programs. The reports should be submitted in the pdf format. If you include any hand-made writings or drawings in the report please make sure they fit the page and are legible. The submissions (or their parts) that are not legible with a pdf reader will be left ungraded and receive zero score. Please see Canvas for detailed instructions on how to submit the reports and programs.

Problem 1. Map coloring problem

Assume we want to solve the **map coloring problem** in Figure 1. The goal is to color a map such that no countries on the map that share a border are assigned the same color. The number of colors is limited. In this assignment assume you have three different colors: Red, Green, and Blue.

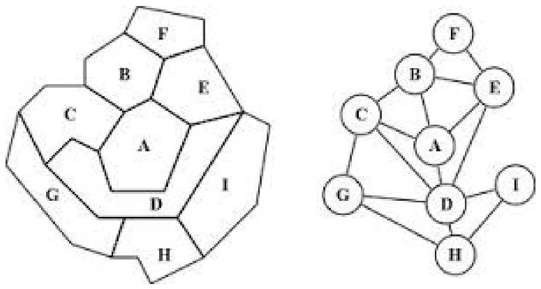


Figure 1: Map coloring problem. Left: spatial layout. Right: abstraction of the spatial layout where nodes correspond to countries and links to borders.

Part a. Formulate the map coloring problem as a (graph) search problem by defining its initial state, operators and the goal condition.

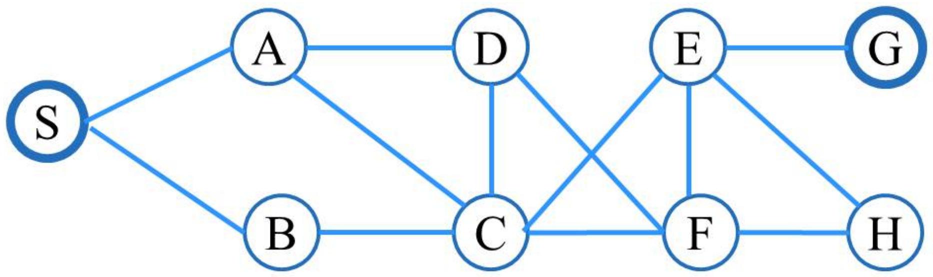
Part b. What is the search space size of the formulation in Part a? If the exact calculation of the search space size becomes hard, give a reasonable upper bound estimate.

Part c. Think of another possible formulation of the map coloring problem as the search problem. What is the search space size of this new formulation? Compare the first two formulations in terms of the search space size they define and determine which formulation appears more advantageous.

Part d. Do you think you can solve the problem? If yes, please submit the solution.

Problem 2. Traveler problem

Consider the following graph representing road connections between different cities. Let S be the initial city and G the destination.



Part a. Show how the breadth-first-search (with no state repeats checks) would search the graph. That is, give an order (of first 10 nodes) in which the nodes could be *expanded*. Use the alphabetical order to order the equivalent choices.

Part b. Show how the depth-first search with the elimination of cyclic state repeats would search the graph by giving an order of first 10 *expanded* nodes. Use the alphabetical order to break the ties (i.e., equivalent node choices).

Part c. Show how the breadth-first-search with elimination of cyclic repeats only would search the graph. Give an order of first 10 expanded nodes. Again, use the alphabetical order to break the ties (i.e., equivalent node choices).

Part d. Show how the breadth-first search that checks for all state repeats would search the graph. Give an order of first 10 expanded nodes.

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

In this problem we will implement a number of uninformed search techniques and test them on the 8-puzzle problem. The rules for submitting the programs are described in Canvas.

The 8-puzzle problem is described in the textbook (Russell and Norvig) on page 68. We have also studied the problem in lecture 2 (see lecture notes). The problem formulation of the 8-puzzle problem consists of:

- States: different tile configurations
- Operators: moves of an empty position
- Initial configuration.
- Goal configuration:

1	2	3
4	5	6
7	8	0

  where 0 represents the empty (blank) tile. Note that the goal configuration we consider is different from the configuration in the textbook!
- Solution (path) cost: the number of moves of the empty tile.

Part a. Run the plain breadth-first search algorithm.

To get you started in the assignment, you are given a python code implementation of the breadth-first search method for 8-puzzle. You can download the code using the assignment link in Canvas. The two files given are:

- *Puzzle8.py* that defines the Puzzle 8 problem, search tree nodes, a hash table, and 5 different game configurations to solve labeled as Example 1, ..., Example 5
- *bfs.py* that implements the basic breadth first search procedure and runs it in on 4 out of 5 initial game configurations.

Once you run the code in *bfs.py*, you should see the solutions for four initial configurations. Three of the initial configurations are shown below:

Example 1:

1	2	3
4	6	0
7	5	8

Example 3:

4	1	2
7	6	3
0	5	8

Example 4:

4	1	2
7	6	3
5	8	0

Familiarize yourself with the python code given to you before proceeding to Part b.

Part b. Breadth-first search statistics.

Write a *breadth_first_search_stats* procedure that modifies the breadth first search procedure given to you in file *bfs.py* such that it is able to collect and print the following statistics:

- the total number of nodes expanded;
- the total number of nodes generated;
- the maximum length of the queue structure;
- the length of the solution path (number of moves)

The statistics should be printed after the example is solved and should be followed by the solution (move) sequence. Include the *breadth_first_search_stats* procedure in the *bfs_stats.py* file and use it to execute it on the first four initial configurations similarly to *bfs.py*.

Part c. Breadth-first search with the elimination of cyclic state repeats.

The basic breadth-first search procedure does not check for and eliminate state repeats. In general, there are two strategies to eliminate state repeats:

- Elimination of cyclic state repeats: Do not expand the node if its state is the same as in one its ancestors in the search tree.
- Elimination of all state repeats: Do not expand the node if its state has been expanded before.

Note: Please see lecture notes for Lecture 3 on the elimination of state repeats.

To implement the bfs with cycling check repeats you will need to write two pieces of code:

- function *check_cyclic_repeats*(*node*) that takes a tree node and checks if the state linked to that node is also associated with one of its parent nodes in the search tree. The function should return true if the cyclic repeat indeed occurred.
- function *breadth_first_search_cycles* that modifies your code in Part b, such that prior to the node expansion it calls *check_cyclic_repeats*(*node*) to check if we entered the cycle.

Include the above two functions in the *bfs_cycle.py* file and run it on the same set of four examples we included in *bfs.py* file. You may also try to solve the initial configuration in Example5. The function should collect and print the same statistics as in Part b.

Part d. Breadth-first search with the elimination of all state repeats.

Implement a *breadth_first_search_repeats* procedure that: (1) checks for and eliminates all state repeats, (2) collects and prints the same statistics as in Part b. Include the procedure in the *bfs_repeats.py* file and run it on all five test examples.

To implement the elimination of all state repeats please use the hash table implemented in *Puzzle*8*.py* file.

Hint: Similarly to the breadth first search with cyclic state repeats we recommend to check the node (its state) for repeats just before the node is expanded, that is, after it is extracted from the queue. Note that you do not have to check for cyclic repeats since the all repeats test subsumes the cyclic repeats test.

Part e. Analysis of the results

Analyze the performance of all three bfs methods (parts b,c,d) in terms of the collected statistics and include the analysis in the report. More specifically you should:

- Summarize the results of the different bfs methods in different tables, one table for every example tried.
- Compare the methods in terms of the respective statistics. Which one is the best? Explain why.

Part f. Depth-limited depth-first search

Implement a depth-limited depth-first search procedure $depth\_first\_search\_limit(problem, limit)$. Please note that the depth of the node is stored in the g field of the Treenode structure, so you can easily check if the node satisfies the depth limit. Your procedure should return the optimal solution if it can be reached within the search limit. Also your limited-depth depth-first search procedure should try to check for and avoid the branches of the tree that are suboptimal given the previously seen nodes and states. To do so please use the hash table and its ability to keep an arbitrary positive value for each key. Briefly, for each explored state always keep in the hash the length of the minimum path to that state as observed during the search process. Finally, the procedure should calculate and keep the same search statistics as used in Parts b, c, d.

Include the depth-limited depth-first search procedure in the *dfs_limit.py* file and run it using the limit 10 on Examples 1, 2 and 3. Analyze the results of your dfs procedure and compare it to results obtained for the different versions of bfs in Parts b, c, d.

Programs to be submitted with your assignement. In addition to the report you should submit the folowing python files implementing Parts b,c,d,f: *bfs_stats.py*, *bfs_cycles.py*, *bfs_repeats.py* and *dfs_limit.py*. These files and the code in them will be run by the TA to check for the consistency with your results.