

Problem assignment 1

Due: Thursday, September 17, 2009

Please note that homeworks should be handed in on the due date before the regulary scheduled class. This concerns both the reports and programs. For instructions on how to submit the programs see the course web page <http://www.cs.pitt.edu/~milos/courses/cs1571/>.

Problem 1

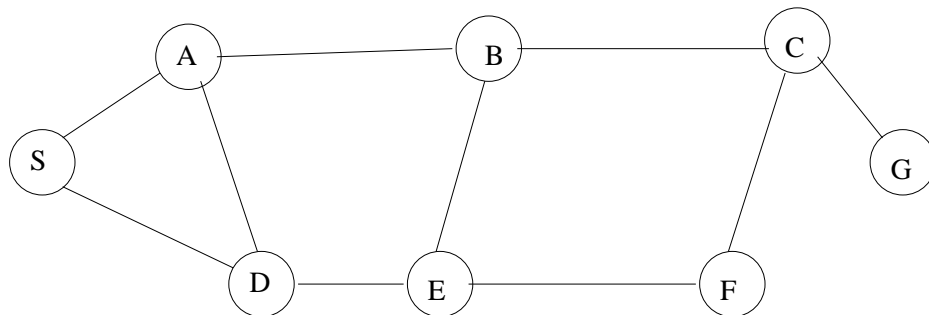
Assume we want to solve the **map coloring problem**. The goal is to color a map such that no countries on the map that share a border are assigned the same color. The number of colors is limited. For the purpose of assignment assume you have a map with 10 countries and you have three different colors: Green, Red and Blue.

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 your formulation? If the exact calculation of the search space size of your formulation becomes hard, give a reasonable upper bound estimate.

Problem 2

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 depth-first search (DFS) would search the graph. That is, give an order (of first 10 nodes) in which the nodes could be *expanded*.

Part b. Is the depth first search (DFS) threatened by cycles? That is, is it possible that the DFS program can become stuck in the cycle and hence miss the solution?

Part c. Show how the breadth-first-search would search the graph.

Part d. Can the BFS program be ever become stuck in the cycle and miss the solution?

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 programming assignments are posted at: <http://www.cs.pitt.edu/~milos/courses/cs1571/program-submissions.html>

The 8-puzzle problem is described in the textbook (Russell and Norvig) on page 65. We have also studied the problem in lecture 3 (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

Number 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 C/C++ code implementation of the breadth-first search method for 8-puzzle. You can download it from:

<http://www.cs.pitt.edu/~milos/courses/cs1570/PS/HW-1/>.

As the first step compile the programs with `g++` and run them. Use the `Makefile` to compile the code. To run the code run the `bfs` executable. More detailed instructions are provided in the `README` file.

Once you successfully compile the code and run it, you should see the solutions for two of the initial configurations we will use in the assignment – EXAMPLE1 and EXAMPLE2. Altogether, there are three examples of increasing complexity:

Example 1:	Example 2:	Example 3:																											
<table border="1"><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>4</td><td>6</td><td>0</td></tr><tr><td>7</td><td>5</td><td>8</td></tr></table>	1	2	3	4	6	0	7	5	8	<table border="1"><tr><td>4</td><td>1</td><td>2</td></tr><tr><td>7</td><td>6</td><td>3</td></tr><tr><td>0</td><td>5</td><td>8</td></tr></table>	4	1	2	7	6	3	0	5	8	<table border="1"><tr><td>4</td><td>1</td><td>2</td></tr><tr><td>7</td><td>6</td><td>3</td></tr><tr><td>5</td><td>8</td><td>0</td></tr></table>	4	1	2	7	6	3	5	8	0
1	2	3																											
4	6	0																											
7	5	8																											
4	1	2																											
7	6	3																											
0	5	8																											
4	1	2																											
7	6	3																											
5	8	0																											

Familiarize yourself with C/C++ structures and functions in *search.h*, *search.c*, and *main1a.c* files before proceeding to Part b.

Part b. Breadth-first search statistics.

Write a *bfs_stat* procedure that modifies the *bfs* procedure in *main1a.c* 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 an example is solved. Write and submit a *main1b.c* program that includes the code of the new *bfs_stat* procedure and executes it on all three test examples: EXAMPLE1, EXAMPLE2, EXAMPLE3.

Note that it may take some time to solve 'hard' configurations. If it takes too long, you may stop the program and say so in the report. However, you must be able to solve at least Example 1. Also the search tree can grow very large and it is possible your system will run out of free memory (depends on the system you use).

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 from September 14, 2006 on elimination of state repeats.

In *search.c*, define the function *check_cyclic_repeats* declared in *search.h* file. Its purpose is to check a node of the search tree for cyclic repeats. A skeleton definition is already included in *search.c*. You need to modify this definition.

Use the function *check_cyclic_repeats* you just defined to implement *bfs_cycle*, a breadth-first search procedure *bfs_cycle* that: (1) checks for and eliminates cyclic repeats, (2) collects and prints the same statistics as in Part b.

Include your *bfs_cycle* procedure in the *main1c.c* file and run it on all three test examples: EXAMPLE1, EXAMPLE2, EXAMPLE3. Do not forget to submit your *search.c*.

Hints: You need to traverse the search tree structure to correctly implement *check_cyclic_repeats*. To write *bfs_cycle* we recommend to check the node (its state) for repeats just before the expansion of the node, that is, after it is extracted from the queue. It is possible to check for repeats also earlier, for example, when the node is generated, but it is not sufficient on its own.

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

Implement a breadth-first search procedure *bfs_mark* 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 *main1d.c* file and run it on all three test examples.

To implement the elimination of all state repeats we will build upon the following *marking* scheme that effectively implements open and closed list structures mentioned in the RN textbook (page 82):

- Every expanded state is marked (a mark signals the expansion),
- Marked states are stored in a special hash table structure.
- The occurrence of a past expansion of a state is checked by checking the presence of the state in the mark structure.

The C/C++ code implementing the 'marking' routines through hash tables is in files *mark.c* and *mark.h*. The following functions implement the marking:

- *check_mark(node)* that takes a node and checks if the state has ever been marked (mark signals the expansion of a state in the past);

- *mark_node(node)* that marks the state in the node;
- *initialize_mark_structure()* that initializes the structures used by marking routines (you must run it before using marking);
- *remove_marks()* that clears the structure used to store marked states.

Hint: Similarly to the *bfs_cycle* 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 mark test subsumes the cyclic repeats test.

Part e. Analysis of the results

Analyze the performance of three methods (parts b,c,d) in terms of the collected statistics and include the analysis in the problem set report together with other problems. You should:

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