

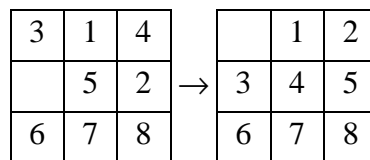
CSIS0270 Artificial Intelligence, 2002–2003

Assignment 1

Deadline Feb 14, 2003, 5:00pm.

This assignment contains both written parts (Questions 1–3) and programming part (Question 4). For the written parts, hand-in your answers to the assignment box (R2). For the programming part, hand-in the programs that you write via the hand-in system of the department.

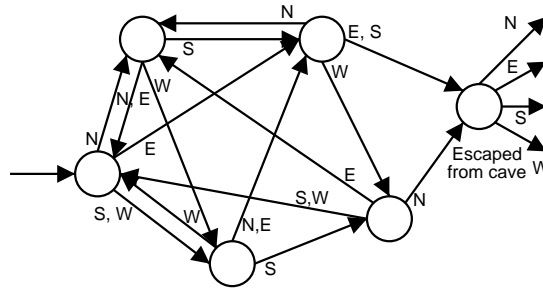
1. **Search Strategies (20%)** (Textbook question 4.3.) Prove each of the following statements:
 - a. Breadth-first search is a special case of uniform-cost search.
 - b. Breadth-first search, depth-first search, and uniform-cost search are special cases of best-first search.
 - c. Uniform-cost search is a special case of A* search.
2. **A* search (20%)** “Dry-run” (i.e., using paper and pens, rather than a computer) the A* search algorithm on the following 8-puzzle problem. Use number of misplaced tiles as the heuristic, and avoid duplicated states by not going back to the state where you come from.



3. **Dungeon (20%)** You are playing a text mode adventure game. In such a game, you type commands like “north”, “south”, “east”, “west”, etc., to “walk around” a world. After a while, you find that you are in a cave, looking exactly alike. You start typing random commands hoping that you will escape from it by chance:

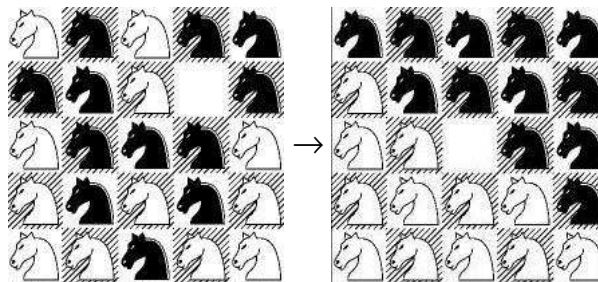
```
You are at a twisty little passage, all alike.
> north
You are at a twisty little passage, all alike.
> north
You are at a twisty little passage, all alike.
> north
You are at a twisty little passage, all alike.
> east
You are at a twisty little passage, all alike.
> west
You are at a twisty little passage, all alike.
...
```

After trying that for half an hour, you decide that too much is too much. Instead of continuing randomly, you look into the game files to see what the “world” looks like. You find that the cave has 24 nodes, with a rather arbitrary structure. You also find that the world is “safe”: there is a way to escape from any state in the cave. So there should be no problem escaping, except that you no longer know where you are! You are a bit regret that you tried so many random moves, and a bit unhappy that you don’t have a time machine. Since playing the game again from the start would waste months of efforts, and since your other friends will also have the same problem, you decide to seek a sequence of moves that guarantees to escape from the cave. E.g., if the cave looks like this:



Then the sequence “south, south, north, south, south” will definitely escape the cave, no matter where you were at the beginning. Of course, it would be better if the sequence is shorter (there are some in this example cave).

- a. Pose the problem as a state space search problem. (Hint: read section 3.6, sensorless problem, of the textbook). Clearly describe the states that you will use, the successor function, and the goal, in a way that makes it easy to implement.
 - b. After you try solving the problem with IDS, you find that the search uses up the memory of your computer and cannot terminate. Suggest a heuristic which allows you to apply A* search.
4. ***Knight moves (40%)*** On a 5×5 chessboard, 24 knights are placed, 12 in black and 12 in white. They can move in the normal way in which knights move in chess (2 steps forward and 1 step side-way, resulting in 8 possible movements). We want a program to find what is the shortest sequence of moves that bring the knights to a fixed state:



The program would read a description of the initial location of the knights, in the following form:

```
01011
110 1
01110
01010
00100
```

Then it should perform searching, and prints out a sequence of location in form (row, col) specifying the locations of knights to move (here, row starts from top, column starts from left, and both have values between 0 to 4).

The course web page contains a partial program (`partial.cc`), which has a sample implementation of the state representation (the struct `config`), how to read the state above (the function `read_config`), and how to make a move (using `make_move`). You may base your work on that program.

- a. Implement such a program using IDS, avoiding repeated states by not going back to

the previous state.

- b. Repeat (a) using BFS, avoiding repeated states by not going to any expanded state.
- c. Repeat (a) using bidirectional search (BFS from both sides), avoiding repeated states by not going to any expanded state.

NB: For part (b) and (c), you might end up using a huge amount of memory, especially for buggy programs. If you do it in a server computer (e.g., `virtue`) or in Linux, please limit your virtual memory size to a reasonable value (e.g., by running `ulimit -v 40960`) before testing your program, or the computer will use up its memory and start swapping memory to disk forever. Also, the initial state shown above requires 18 steps to solve, and will overwhelm many of the simpler search strategies. You probably want to start testing your program using a much simpler initial state.