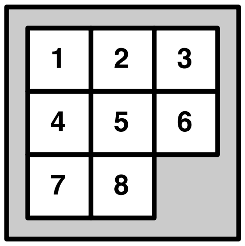
CS-330 Fall 2016, Prolog assignment

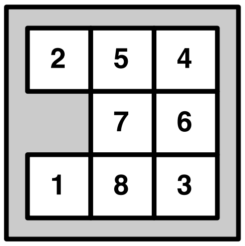
Due: November 10, 2016

Professor: Carlotta Domeniconi

The 8-Puzzle



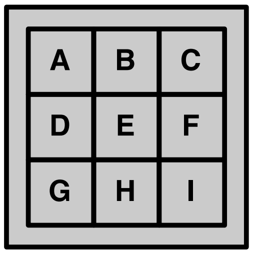
Solved Puzzle



A Jumbled Puzzle

The **8-puzzle** is a sliding block puzzle consisting of eight tiles in a 3 by 3 grid, plus an empty space into which tiles can slide. You can play the 8-puzzle at **http://mypuzzle.org/sliding/** among many other places on the web. A "bigger" version of the puzzle is the 15-puzzle, which has fifteen tiles in a 4 by 4 grid. In addition to being time-wasters, these puzzles are popular in artificial intelligence for demonstrating the value of informed or heuristic search techniques. Here we're going to tackle solving the 8-puzzle with a simple variation of depth-first search. This is hardly the most efficient way to go about it, but it's very easy for us because Prolog's approach to solving logic statements is depth-first.

The goal is to get the tiles in the configuration shown above as a **Solved Puzzle**. Note that the empty square is in the bottom right corner. Each tile will be referred to as a number, and we're going to refer to the empty square as **tile 0**, even though it's not really a tile of course. The tiles could start in any initial configuration, though note that you can't just pick random configurations because some cannot reach the solution no matter how hard you try. One valid initial configuration (among a great many) is shown above as the "Jumbled Puzzle".



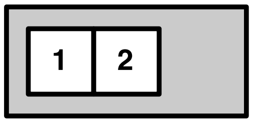
Tile Locations

There are nine possible locations for the eight tiles and "tile 0" (the empty square). These locations will be referred to by the letters A through I, as shown at right. Notice that the only tiles which can move are ones which are next to the empty square. Consider if the empty square is located at A. Then the only tiles which can move are the tile located at B (by **sliding left**) and the tile located at D (**by sliding up**). After sliding over the tile at B, the empty square is of course now located at B: essentially the tile at B and the empty square have changed positions. If the empty square is located at A, C, G, or I, there are only two possible moves. If the empty square is located at B, D, F, or H, there are three possible moves. Finally, if the empty square is located at E, there are four possible moves.

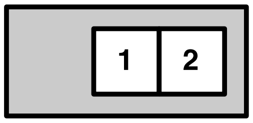
We are going to create a Prolog program which solves the 8-puzzle from any valid initial configuration. Here's how it will work. It will first consider all sequences of moves starting from the initial configuration which are only one slide long each. If it's not found the goal, it'll look for sequences which are two slides long. If it's still not found the goal, it'll look for sequences which are three slides long, and so on, until it finally finds the goal. Obviously as the possible sequences get longer, it'll start taking more and more time to find the solution. When it finally finds the solution, it will return it as a list of moves like **[up, left, up, right, right, down, up, down]**, where each move indicates how next to slide a tile into the current open space.

This is a daunting task but we'll make it easier by breaking it into stages. Note that there's a lot of sliding-block 8-puzzle Prolog stuff on the web: you **cannot** use it. The use of any code (other than yours) will constitute a violation of the GMU Honor Code. Anyway, most of the code available won't help you and will just confuse you because we're requiring that you do the puzzle solution mechanism our way.

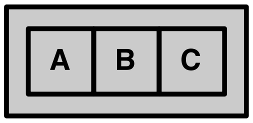
Stage 1: Solve the 2-puzzle



Solved Puzzle



"Jumbled" Puzzle



Tile Locations

The 2-puzzle is shown at right. You will agree it's not very complex. You will define a predicate called **move(Depth, A, B, C)** which receives tiles into the variables A, B, and C (representing locations). The empty square is represented by 0. If the tiles are in the right place, then the predicate should succeed. If the tiles are in the wrong place, then the predicate should recursively try all valid moves for the current configuration passed into A, B, and C.

Note that this could lead to infinite recursion, and you'll run out of stack space. For example, consider the "Jumbled" puzzle at right. The predicate tries moving tile 1 to the left. Then it next tries moving tile 1 back again, and so on, resulting in an infinite loop. For this reason, you will start with a high **Depth** value and decrease it as you make deeper moves, and ultimately stop further search (failing) if Depth reaches zero.

Let's say we're willing to search 5 moves deep. We could start the puzzle-solving session, for the "Jumbled" Puzzle, like this:

```
move(5, 0, 1, 2).
```

Hints: move(...) will consist of five rules. You can use **trace** to see if the solver is searching right. A trace might look like:

```
   ?- move(5, 0, 1, 2).
          (1)    call:move(5,0,1,2) ?
          (2)    call:move(5-1,1,0,2) ?
          (3)    call:move(5-1-1,1,2,0) ?
?         (3)    exit:move(5-1-1,1,2,0) ?
?         (2)    exit:move(5-1,1,0,2) ?
?         (1)    exit:move(5,0,1,2) ?
yes
```

Stage 2. Show the answer

Stage 1 does not provide a very informative answer: it just tells you if a solution was found, and the trace is cumbersome to follow. Instead we're going to create a variable called **Trace** which collects the answer into a list and returns it. The valid moves are **left** and **right**, and obviously for some empty square positions only one or the other is valid at a given time. Modify the move predicate to look like this: **move(Trace, Depth, A, B, C).** Now if you do a query:

```
move(Trace, 5, 0, 1, 2).
Trace = [left,left] ?
yes
```

And of course, the correct answer is to move **left** and then **left** again.
A trace might look like:

```
   ?- move(Trace, 5, 0, 1, 2).
          (1)    call:move(_131111,5,0,1,2) ?
          (2)    call:move(_131211,4,1,0,2) ?
          (3)    call:move(_131302,3,1,2,0) ?
?         (3)    exit:move([],3,1,2,0) ?
?         (2)    exit:move([left],4,1,0,2) ?
?         (1)    exit:move([left,left],5,0,1,2) ?
Trace = [left,left] ?
yes
```

Stage 3. Add Iterated Deepening

Iterated Deepening is the strategy of increasingly doing deeper depth-first searches until you find the answer. Create a predicate called **search(Trace, Depth, A, B, C)** which calls **move(Trace, 0, A, B, C)**, then **move(Trace, 1, A, B, C)**, and so on forever. Each time it prints out the current depth. You start it with a depth of 0:

```
   ?- search(Trace, 0, 0, 1, 2).
0
1
2
Trace = [left,left] ?
yes
```

Then create a cover predicate **solve(Trace, A, B, C)**, so you don't have to pass in 1 to **search**:

```
   ?- solve(Trace, 0, 1, 2).
0
1
2
Trace = [left,left] ?
yes
```

Stage 4. Solve the 8-Puzzle

Change to the following predicates:

- **move(Trace, Depth, A, B, C, D, E, F, G, H, I)**
- **search(Trace, Depth, A, B, C, D, E, F, G, H, I)**
- **solve(Trace, A, B, C, D, E, F, G, H, I)**

You might use the initial configuration found on the first page as a test. Or maybe something simpler, like one or two slides wrong.

SUBMISSION: Submit four (ascii) files, one for each Stage of the project. You must send your files electronically to our TA Ivan Avramovic at iavramo2@masonlive.gmu.edu, and to me at carlotta@cs.gmu.edu. You must also submit a hardcopy of your code at the beginning of class on Thursday, November 10.

This assignment must be performed individually. Group work is NOT allowed. Any deviation from this policy will be considered a violation of the GMU Honor Code. Be advised that submitting somebody else's code (including code found on the web) is considered plagiarism, and therefore in violation of the GMU Honor Code.

The deadline to submit your code electronically is **11AM Thursday, November 10**. Late submissions will be penalized **10 points a day**.