Logic Programming and Artificial Intelligence

Prolog Lab Problem Sheet 3

The goal of this question is to understand the various search methods by writing Prolog implementations of some of them and using them for problem solving. The 8-puzzle will be used as an example throughout.

To simplify the task you are given, at the end of this question sheet and on the LPA web page as `eightpuzzle.pl`, Prolog code that implements all the movement operators for the 8-puzzle. The movement operators are defined using the predicate: `apply(Op, Before, After)` which is true if and only if the result of operator `Op` is applicable to state `Before` and the result of doing so is the state `After`.

The four operators that can be applied are named left, right, up and down. You can think of the operators as moving the empty square since moving the empty square is equivalent to moving a tile (by swapping the two) in the opposite direction. So, left means moving the empty square to the left i.e. the tile to the left of the empty square is moved onto the previous position of the empty square.

The states are encoded as lists of nine elements, where the empty square is represented by the number 0. As an example, the board configuration shown on the right-hand side in Figure 1 is encoded as the list [1,2,3,8,0,4,7,6,5]. This configuration is the goal configuration. The initial state (also given at the end of the question sheet and online) is [8,1,3,7,0,2,6,5,4].

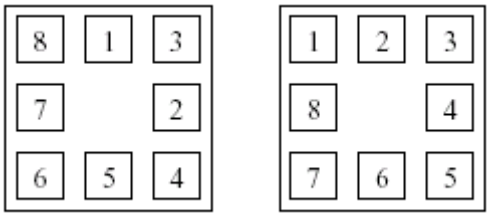

Figure 1: The 8-puzzle (left=initial state, right=goal state).

As an example, the Prolog call:

```
:- apply(left, [1,2,3,8,0,7,6,5], X).
```

has exactly one answer: `X = [1,2,3,0,8,4,7,6,5]`.

To get you started, you are also given (at the end of this file) a Prolog implementation of breadth-first search along with definitions for the initial state (shown in Figure 1, left) and goal state (shown in Figure 1, right). The implementation of breadth-first search is generic in that it can operate on any search space and it also maintains a path-to-goal list. Once goal and apply are defined it will search for a goal from any given state.
You should try using it to solve the 8-puzzle by using the definition of apply that you have been given. You can run the program by:

```
:- bfs([8,1,3,7,0,2,6,5,4],P).
```

The solution you should see (after a few seconds) is:

```
P = [right,down,left,left,up,up,right,down]
```

Using the given implementation of breadth-first search as a guide, implement the following search algorithms:

(a) Depth-bounded depth first search
(b) Iterative Deepening Search
(c) Greedy Best First search
(d) A* search

NOTE: It may be useful for you to try your algorithm implementations out with simpler examples than the initial state. Remember that the shallowest solution that `bfs/2` found for the example above was at depth 8!

All of these should be able to return the path to the goal in the same manner as the implementation of breadth-first search. For the search algorithms that require you to implement heuristic functions, at the end of this file you are given Prolog code for two such heuristics:

- Number of displacements: the total number of tiles that are not in the position they need to be in the goal state.
- Manhattan distance: the sum of the distances of the tiles from their goal positions.

```
initial([8,1,3,7,0,2,6,5,4]).
goal([1,2,3,8,0,4,7,6,5]).
operators([left, right, up, down]).
% main procedure for breadth first search
bfs(Start,Path) :-
     bfs_path([node(Start,[])], Path).
% breadth_first search
bfs_path([node(State,Path) | _], Path) :-
     goal(State).
bfs_path([node(State, Path) | Queue], GoalPath) :-
     findall(node(Child,PathToChild),
               (
                 apply(Operator, State, Child),
                 append(Path,[Operator],PathToChild)
                ), ChildNodes),
     append(Queue, ChildNodes, NewQueue),
     bfs_path(NewQueue, GoalPath).
```

```
%=====================================================
%Implementation of 8-Puzzle Operators
%=====================================================
% move_left in the top row
move_left([X1,0,X3, X4,X5,X6, X7,X8,X9],
          [0,X1,X3, X4,X5,X6, X7,X8,X9]).
move_left([X1,X2,0, X4,X5,X6, X7,X8,X9],
          [X1,0,X2, X4,X5,X6, X7,X8,X9]).

% move_left in the middle row
move_left([X1,X2,X3, X4,0,X6, X7,X8,X9],
          [X1,X2,X3, 0,X4,X6, X7,X8,X9]).
move_left([X1,X2,X3, X4,X5,0, X7,X8,X9],
          [X1,X2,X3, X4,0,X5, X7,X8,X9]).

% move_left in the bottom row
move_left([X1,X2,X3, X4,X5,X6, X7,0,X9],
          [X1,X2,X3, X4,X5,X6, 0,X7,X9]).
move_left([X1,X2,X3, X4,X5,X6, X7,X8,0],
          [X1,X2,X3, X4,X5,X6, X7,0,X8]).

% move_right in the top row
move_right([0,X2,X3, X4,X5,X6, X7,X8,X9],
           [X2,0,X3, X4,X5,X6, X7,X8,X9]).
move_right([X1,0,X3, X4,X5,X6, X7,X8,X9],
           [X1,X3,0, X4,X5,X6, X7,X8,X9]).

% move_right in the middle row
move_right([X1,X2,X3, 0,X5,X6, X7,X8,X9],
           [X1,X2,X3, X5,0,X6, X7,X8,X9]).
move_right([X1,X2,X3, X4,0,X6, X7,X8,X9],
           [X1,X2,X3, X4,X6,0, X7,X8,X9]).

% move_right in the bottom row
move_right([X1,X2,X3, X4,X5,X6, 0,X8,X9],
           [X1,X2,X3, X4,X5,X6, X8,0,X9]).
move_right([X1,X2,X3, X4,X5,X6, X7,0,X9],
           [X1,X2,X3, X4,X5,X6, X7,X9,0]).

% move_up from the middle row
move_up([X1,X2,X3, 0,X5,X6, X7,X8,X9],
        [0,X2,X3, X1,X5,X6, X7,X8,X9]).
move_up([X1,X2,X3, X4,0,X6, X7,X8,X9],
        [X1,0,X3, X4,X2,X6, X7,X8,X9]).
move_up([X1,X2,X3, X4,X5,0, X7,X8,X9],
        [X1,X2,0, X4,X5,X3, X7,X8,X9]).
```

```
% move_up from the bottom row
move_up([X1,X2,X3, X4,X5,X6, 0,X8,X9],
        [X1,X2,X3, 0,X5,X6, X4,X8,X9]).
move_up([X1,X2,X3, X4,X5,X6, X7,0,X9],
        [X1,X2,X3, X4,0,X6, X7,X5,X9]).
move_up([X1,X2,X3, X4,X5,X6, X7,X8,0],
        [X1,X2,X3, X4,X5,0, X7,X8,X6]).

% move_down from the top row
move_down([0,X2,X3, X4,X5,X6, X7,X8,X9],
          [X4,X2,X3, 0,X5,X6, X7,X8,X9]).
move_down([X1,0,X3, X4,X5,X6, X7,X8,X9],
          [X1,X5,X3, X4,0,X6, X7,X8,X9]).
move_down([X1,X2,0, X4,X5,X6, X7,X8,X9],
          [X1,X2,X6, X4,X5,0, X7,X8,X9]).

% move_down from the middle row
move_down([X1,X2,X3, 0,X5,X6, X7,X8,X9],
          [X1,X2,X3, X7,X5,X6, 0,X8,X9]).
move_down([X1,X2,X3, X4,0,X6, X7,X8,X9],
          [X1,X2,X3, X4,X8,X6, X7,0,X9]).
move_down([X1,X2,X3, X4,X5,0, X7,X8,X9],
          [X1,X2,X3, X4,X5,X9, X7,X8,0]).

% Applying an operator
apply(left,S1,S2) :- move_left(S1,S2).
apply(right,S1,S2) :- move_right(S1,S2).
apply(up,S1,S2) :- move_up(S1,S2).
apply(down,S1,S2) :- move_down(S1,S2).

%=======================================================
%Implementation of 8-Puzzle Heuristic Functions
%=======================================================
% displacement heuristic
displaced(State, Number) :-
     goal(Goal),
     misplaced(State,Goal,Number).

% misplaced returns the number of tiles found in the
% wrong position
misplaced([],[],0).
misplaced([0|T1],[0|T2],Number) :- !,
     misplaced(T1,T2,Number).
misplaced([H|T1],[H|T2],Number) :- !,
     misplaced(T1,T2,Number).
misplaced([H1|T1],[H2|T2],Number) :- !,
     H1\==H2,
     misplaced(T1,T2,N),
     Number is N+1.
```

```
% Manhattan Distance heuristic
manhattan(State, Number) :-
     manh(State,State,0,Number).
manh([], _, X, X).
manh([H|T], State, Acc, Result) :-
     nth1(Position, State, H),
     NewPos is Position - 1,
     Xaux1 is NewPos mod 3,
     X1 is integer(Xaux1),
     Y1 is NewPos // 3,
     goal(Goal),
     nth1(GoalPosition, Goal, H),
     NewGPos is GoalPosition - 1,
     Xaux2 is NewGPos mod 3,
     X2 is integer(Xaux2),
     Y2 is NewGPos // 3,
     S1 is abs(X1-X2),
     S2 is abs(Y1-Y2),
     N is S1+S2,
     NewAcc is Acc+N,
     manh(T, State, NewAcc, Result).
```