Atte Karppinen D18123298

Prolog Search Strategies

Depth First Search

The default search strategy used by Prolog. Uses stack memory but may have unnecessary long solution paths. Proceeds down one branch of the branch tree at a time always taking the left option. If that fails, backtracks as far as there is another option and goes all the way down that path using the left options. $dx(k-1)$

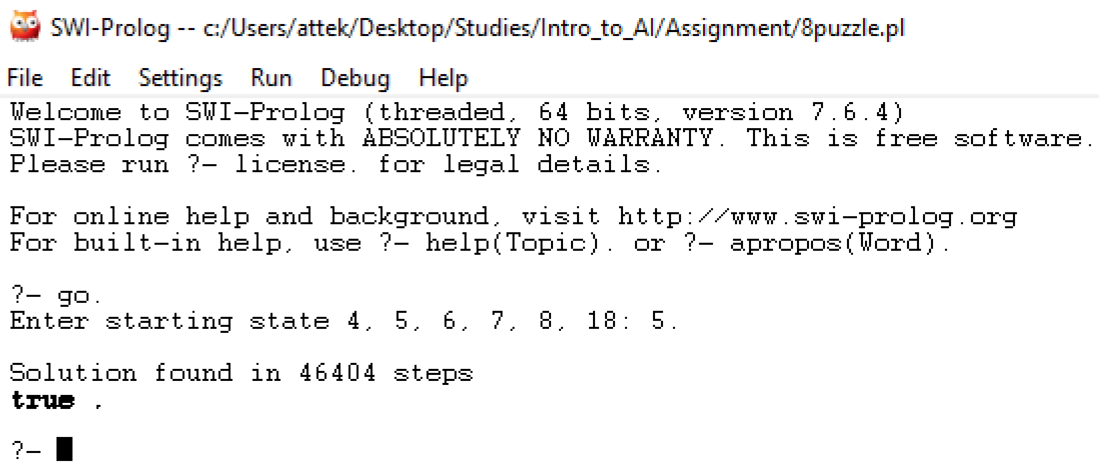
Breadth First Search

Finds the shortest solution path but may cause excessive memory usage because it grows path exponentially. Searches all the options from the first depth (from left to right) before proceeding down to next level. k^d

Iterative Deepening Depth First Search

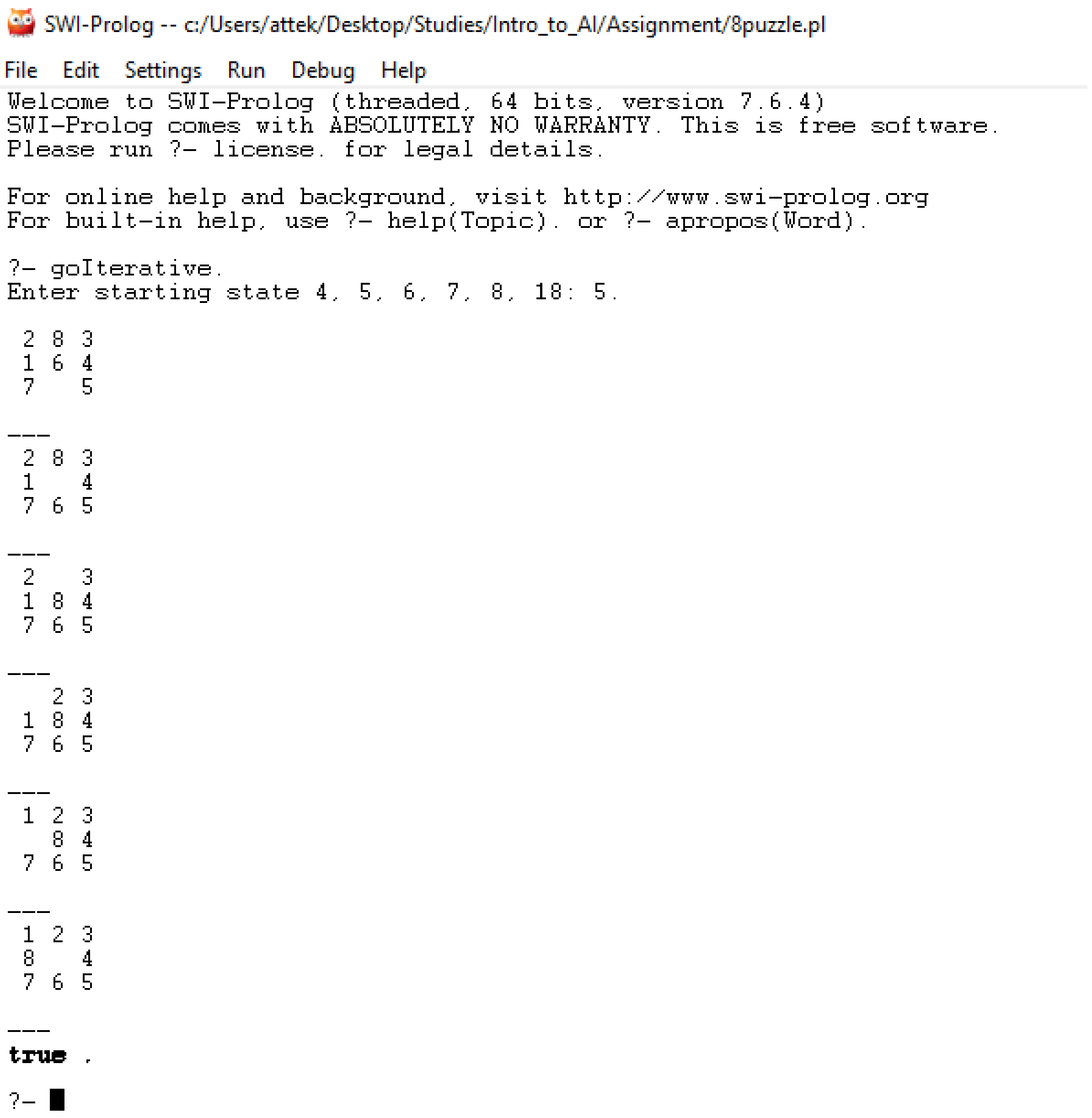
Enhanced version of depth firs search. Uses the same technique but sets maximum depth and adds to it if goal is not met. Returns shortest path and does not use too much memory but not optimal if the wanted state lies deep since instead of going everything through once, it does it as many times as there is depths.

8-puzzle program output using depth first search


SWI-Prolog -- c:/Users/attek/Desktop/Studies/Intro_to_AI/Assignment/8puzzle.pl
File Edit Settings Run Debug Help

```
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- go.
Enter starting state 4, 5, 6, 7, 8, 18: 5.

Solution found in 46404 steps
true .

?- 
```

Using iterative deepening depth first search


SWI-Prolog -- c:/Users/attek/Desktop/Studies/Intro_to_AI/Assignment/8puzzle.pl
File Edit Settings Run Debug Help

```
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- goIterative.
Enter starting state 4, 5, 6, 7, 8, 18: 5.

 2 8 3
 1 6 4
 7   5

---
 2 8 3
 1   4
 7 6 5

---
 2   3
 1 8 4
 7 6 5

---
   2 3
 1 8 4
 7 6 5

---
 1 2 3
   8 4
 7 6 5

---
 1 2 3
 8   4
 7 6 5

---
true .

?- 
```

Predicates and explanations

```prolog
% Iterative deepening solve

% At first start from the first depth (1)
% Reverse to show answer in the bottom
iterative_solve(State, Soln) :-
    iterative_solve(State, 1, Solution),
    reverse(Solution, Soln).


% Because only the shortest path is wanted. Always start
% from the next depth with empty path
% This will go to iterative_solve/4
% and eventually fails when goal is not reached.
% When it fails, go to next iterative_solve/3 to go to next depth (NextDepth).

iterative_solve(State, Depth, Solution) :-
    % Path is always empty when starting
    iterative_solve(State, [], Depth, Solution).


iterative_solve(State, Depth, Solution) :-
    NextDepth is Depth + 1,
    iterative_solve(State, NextDepth, Solution).


% if state is the goal; State|Path the solution
iterative_solve(State, Path, _, [State|Path]) :-
    goal(State).


% Final iterative_solve/4 is only required when the goal condition is met,
% but (I don't think) there is another way
% to save it ONLY when win condition is met.

iterative_solve(State, Path, Depth, Solution) :-
    Depth > 0, % Don't allow negative depth
    move(State, NextState), % Same as DFS solve
    not(member(NextState, Path)),
    LastDepth is Depth - 1,
    iterative_solve(NextState, [State|Path], LastDepth, Solution).
```

```prolog
% Same principle as in DFS but this time use showpath to print visual aid
goIterative :-
    write('Enter starting state 4, 5, 6, 7, 8, 18: '),
    read(StartIndex),
    start(StartIndex, StartState),
    iterative_solve(StartState, Solution),
    showPath(Solution), nl.
```