DUBLIN INSTITUTE of TECHNOLOGY

Institiúid Teicneolaíochta Bhaile Átha Cliath

State Search Strategies

Intro to Artificial Intelligence

Prolog AI Assignment

Completed By:

Student Number: C16315253

Author: William Carey

Course Code: DT228

Year: 3

Date Due: 12 - 11 - 2018

Date Completed: 6 – 11 – 2018

The different types of State Space Searches:

State Space searches is a mechanism that takes in one state, known commonly as the start state, and try to find the goal state from that state. This could be from a few states away to thousands or billions of steps away. Through test and documentation, we find there is three approaches to solve this scenario. They are Depth First Search, Breath First Search and a combination of the two known as Iterative Deeping Search. Through State searching, they take in the form of Predicate Calculus of Knowledge Representation and Reasoning in addition to recursively iterating through the lists to find the desired state.

Through the combination of iterative and recursively searching, we will find that we will constantly go back over similar steps that we came across before, so we store every new step in our path. This is so we can go back over the states in case we need to go back to any previous state in order to try advance to the goal state. It also so we don't run in an infinite loop in our searches.

Depth First Search is often going the adjacent of one desired state and keeps using that path to try to find the solution or goal state. If it leads to problems, it will backtrack and try the next step. The State search tree could end up being very large on one side as a result and if the node is relatively close in level time, the algorithm will use up a lot of unnecessary time to try find the node. The speed of this algorithm is O(D) where D is the depth is the tree.

Another way to search is using Breath Space search. This approach differs from depth as it looks at all the possible paths from its current level and try to pick which one has the least potential steps from its immediate step. This means it takes more space to look at the states, where as Depth looks at the first one it sees, and if it didn't take that path yet it will take it. The speed of the Breath First Search is O(N) where N is the number of nodes traversed.

The two approaches have their advantages and disadvantages in their attempt to search for the most optimal path. One search method that solves the problems of either search method is Iterative Deeping Depth First Search. This method takes the best from both and searches for the most optimal path using both approaches. Through this, each level is visited according to their depth in level. For example, the last level needs only to be visited once, whereas the second last is visited twice. As a result, it is more efficient then the search methods described above. Where any of the above methods could take thousands of steps to solve the predicate, the Deeping method would solve the problem in double digits steps in comparison (Eg Ten steps in opposed to 40000).

Explanation of Prolog Predicate for Iterative Deeping Depth First Search:

The main predicate for my Iterative Deeping search is defining the depth to go. Also to solve the puzzle through a few prefined rules unique to eqch programme that the Iterative deeping will use to aid in its search. The main rules for this that applies to every programme that runs the Iterative deeping method are defining the state, create a path way, create a depth to go as far and using all that find the solution. In the searching part of the algorithm, it checks if the it already took the path and continues if it did not, ensuring it adds it to the current list of paths taken already so it does not take this again.

The main part of the programme is the feature to do a breath first search in the depth first search algorithm for more efficient searching to be completed. It also checks the levels closer to the top level more frequent to find the goal state which might be closer to the start state.

```
id_solve(State, Depth, Sol) :- ids_dfs(State, [], Depth, Sol),
                    write("Goal is at depth: "),write(Depth),nl,!.


id_solve(State, Depth, Sol) :-
                    Depth1 is Depth + 1,
                    id_solve(State,Depth1,Sol).


ids_dfs(X,P,_,[X|P]) :- goal(X).


ids_dfs(X,P,D,Sol) :- D > 0,
                                move(X,Y),
                                not(member(Y,P)),
                                D1 is D - 1,
                                ids_dfs(Y,[X|P],D1,Sol).
```

Eight Piece Puzzle

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%The goal state and some starting states%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
goal( [(2,2), (1,1), (2,1), (3,1),
 (3,2), (3,3), (2,3), (1,3), (1,2)] ) .
%depth 4
start4( [(2,2), (1,1), (3,2), (2,1),
 (3,1), (3,3), (2,3), (1,3), (1,2)] ) .
%depth 5
start5( [(2,3), (1,2), (1,1), (3,1),
 (3,2), (3,3), (2,2), (1,3), (2,1)] ) .
%depth 6
start6( [(1,3), (1,2), (1,1), (3,1),
 (3,2), (3,3), (2,2), (2,3), (2,1)] ) .
%depth 7
start7( [(1,2), (1,3), (1,1), (3,1),
 (3,2), (3,3), (2,2), (2,3), (2,1)] ) .
%depth 8
start8( [(2,2), (1,3), (1,1), (3,1),
 (3,2), (3,3), (1,2), (2,3), (2,1)] ) .
%depth 18
start18( [(2,2), (2,1), (1,1), (3,3),
 (1,2), (2,3), (3,1), (1,3), (3,2)] ) .
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%predicate to help you choose one of the  %
%starting states whose solution paths are %
%at different depths - start(depth, State)%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
start( I , X ) :-
	I == 4 , start4( X ) , !
	;
	I == 5 , start5( X ) , !
	;
	I == 6 , start6( X ) , !
	;
	I == 7 , start7( X ) , !
	;
	I == 8 , start8( X ) , !
	;
	I == 18, start18( X ) .
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%move( State1 , State2 ) generates%
%a successor state                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
move( [E | Tiles] , [T| Tiles1] ):-
	swap( E , T , Tiles , Tiles1 ) .
swap( E , T , [T | Ts] , [E | Ts] ):-
	mandist( E , T , 1 ) .
swap( E , T , [T1 | Ts] , [T1 | Ts1] ):-
	swap( E , T , Ts , Ts1 ) .
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Manhattan Distance -                   %
%mandist( TilePos1 , TilePos2, Dist )    %
%is the distance between two tile positions%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
mandist( (X,Y) , (X1,Y1) , D ):-
        diff( X , X1 , Dx ) ,
        diff( Y , Y1 , Dy ) ,
        D is Dx + Dy .
diff( A , B , D ):-
    D is A - B , D > 0 , !
        ;
        D is B - A .
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% code for pretty printing the solution path and states  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
showPath( [] ) .
showPath( [P | L] ) :-
        showState( P ),
    nl, write('---'),
    showPath( L ).
showState([P0, P1, P2, P3, P4,
 P5, P6, P7, P8]) :-
        member( Y , [1, 2, 3] ),
        nl,
        member( X , [1, 2, 3] ),
        member( Tile-(X,Y),
                [' '-P0, 1-P1, 2-P2, 3-P3, 4-P4,
                5-P5, 6-P6, 7-P7, 8-P8] ) ,
        write(' '), write( Tile ) ,
        fail
        ;
```

```
        nl, true .
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%The code for solving the state%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%iterative deeping
id_solve(State, Depth, Sol) :-
        ids_dfs(State, [], Depth, Sol),
        write("Goal is at depth: "),
        write(Depth),nl,!.
id_solve(State, Depth, Sol) :-
        Depth1 is Depth + 1,
        id_solve(State,Depth1,Sol).
ids_dfs(X,P,_,[X|P]) :- goal(X).
ids_dfs(X,P,D,Sol) :- D > 0,
        move(X,Y),
        not(member(Y,P)),
        D1 is D - 1,
        ids_dfs(Y,[X|P],D1,Sol).
% adding the solving state
solve(N,Sol) :- solve(N,[],Sol).
solve(Node,Path,[Node | Path]) :-
 goal(Node).
solve(Node, Path, Sol) :-
        move(Node,Successor),
        not(member(Successor,Path)),
        solve(Successor,[Node| Path],Sol).
%%%%%%%%%%%%%%%%%%%%
%To run the programme%
```

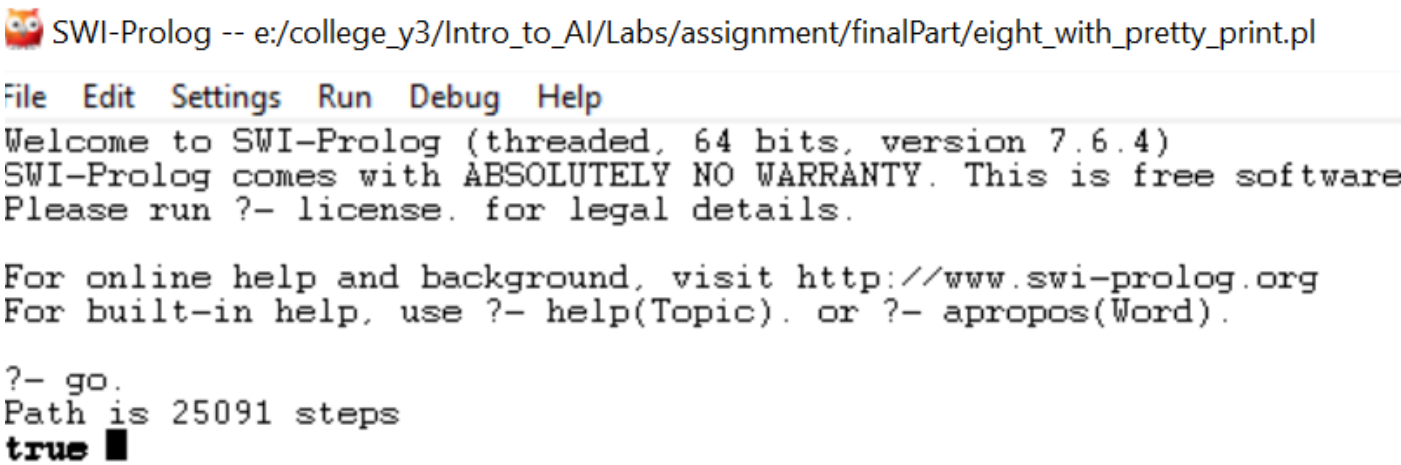
```
%%%%%%%%%%%%%%%%%%%%%%

go :- start4(X),solve(X,A),
      write('Path is '),
      length(A,P),write(P),
      write(' steps').

%%%%%%%%%%%%%%%%%%%%%%

%To run the programme%

%%%%%%%%%%%%%%%%%%%%%%

go2 :-
      write("What number to start at: \c
      4, 5, 6, 7, 8, 18"),
      nl, read(I), nl,
      start(I,A), id_solve(A,I,B),
      reverse(B,B1),showPath(B1).
```

SWI-Prolog -- e:/college_y3/Intro_to_AI/Labs/assignment/finalPart/eight_with_pretty_print.

File Edit Settings Run Debug Help

```
?- go2.
What number to start at: 4, 5, 6, 7, 8, 18
|: 6.

Goal is at depth: 6

 2 8 3
 1 6 4
   7 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.

?-
```

Iterative Deeping Search



The eight piece puzzle is an algorithm of moving all eight tiles to the right position by sliding an of the available ones to the open space inside the tile position. It does this through the Manhattan Distance. This algorithm checks for possible swappable tiles, which they have to be right beside each other which is checked by finding the distance between them to be 1, in ordered to be swapped and to get to their necessary tile position is the largest distance of all the tiles.

It does the swapping and checking all the distances by calling the move/2 method which is called from the ids_dfs itself. The ids_dfs then checks if the current path has already being navigated through. If it has, it will not proceed Once all the objects had moved and swapped with each other to get to the correct positions, the list will stop growing a be enabled to be shown.

Farmer Wolf Goat Cabbage

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%The rules to do with the fwgc algorithm%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
opp(e,w).
opp(w,e).
unsafe(F,X,X,C) :- opp(F,X).
unsafe(F,W,X,X) :- opp(F,X).
% move(State1, State2).
% move the wolf
move(state(X,X,G,C), state(Y,Y,G,C)) :-
    opp(X,Y),
    not(unsafe(Y,Y,G,C)).
% move the goat
move(state(X,W,X,C), state(Y,W,Y,C)) :-
    opp(X,Y),
    not(unsafe(Y,W,Y,C)).
% move the cabbage
move(state(X,W,G,X), state(Y,W,G,Y)) :-
 opp(X,Y), not(unsafe(Y,W,G,Y)).
% move self only
move(state(X,W,G,C), state(Y,W,G,C)) :-
 opp(X,Y), not(unsafe(Y,W,G,C)).
%%%%%%%%%%%%%%%%%%%%%
%End of the rule set%
%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Pretty printing code for FWGC%
```

```
%solution path and states     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% showPath(Path)
showPath([]) :- nl.
showPath([S|Path]) :-
    nl,showState(S),
    showPath(Path).
showState(S) :-
    showWest(S),
          write('|~~~~~~~|'),
          showEast(S),nl,
    write('     |~~~~~~~|').
showWest(state(F,W,G,C)) :-
    (F == w, write('F'), !; write(' ')),
    (W == w, write('W'), !; write(' ')),
    (G == w, write('G'), !; write(' ')),
    (C == w, write('C'), !; write(' ')).
showEast(state(F,W,G,C)) :-
    (F == e, write('F'), !; true),
    (W == e, write('W'), !; true),
    (G == e, write('G'), !; true),
    (C == e, write('C'), !; true).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%End of the printing part%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%hardcoding the goal state
%into the programme
goal(state(e,e,e,e)).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Solving the state of the alogithm%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%iterative deeping
id_solve(State, Depth, Sol) :-
 ids_dfs(State, [], Depth, Sol),
                    write("Goal is at depth: "),
                    write(Depth),nl,!.
id_solve(State, Depth, Sol) :-
 Depth1 is Depth + 1,
 id_solve(State,Depth1,Sol).
ids_dfs(X,P,_,[X|P]) :- goal(X).
ids_dfs(X,P,D,Sol) :- D > 0,
 move(X,Y), not(member(Y,P)),
 D1 is D - 1, ids_dfs(Y,[X|P],D1,Sol).
% adding the solving state
solve(N,Sol) :- solve(N,[],Sol).
solve(Node,Path,[Node | Path]) :-
 goal(Node).
solve(Node, Path, Sol) :-
          move(Node,Successor),
          not(member(Successor,Path)),
          solve(Successor,[Node| Path],Sol).
%%%%%%%%%%%%%%%%%%%%
%To run the programme%
%%%%%%%%%%%%%%%%%%%%
go :- solve(state(w,w,w,w),A),
 reverse(A,A1), showPath(A1).
```

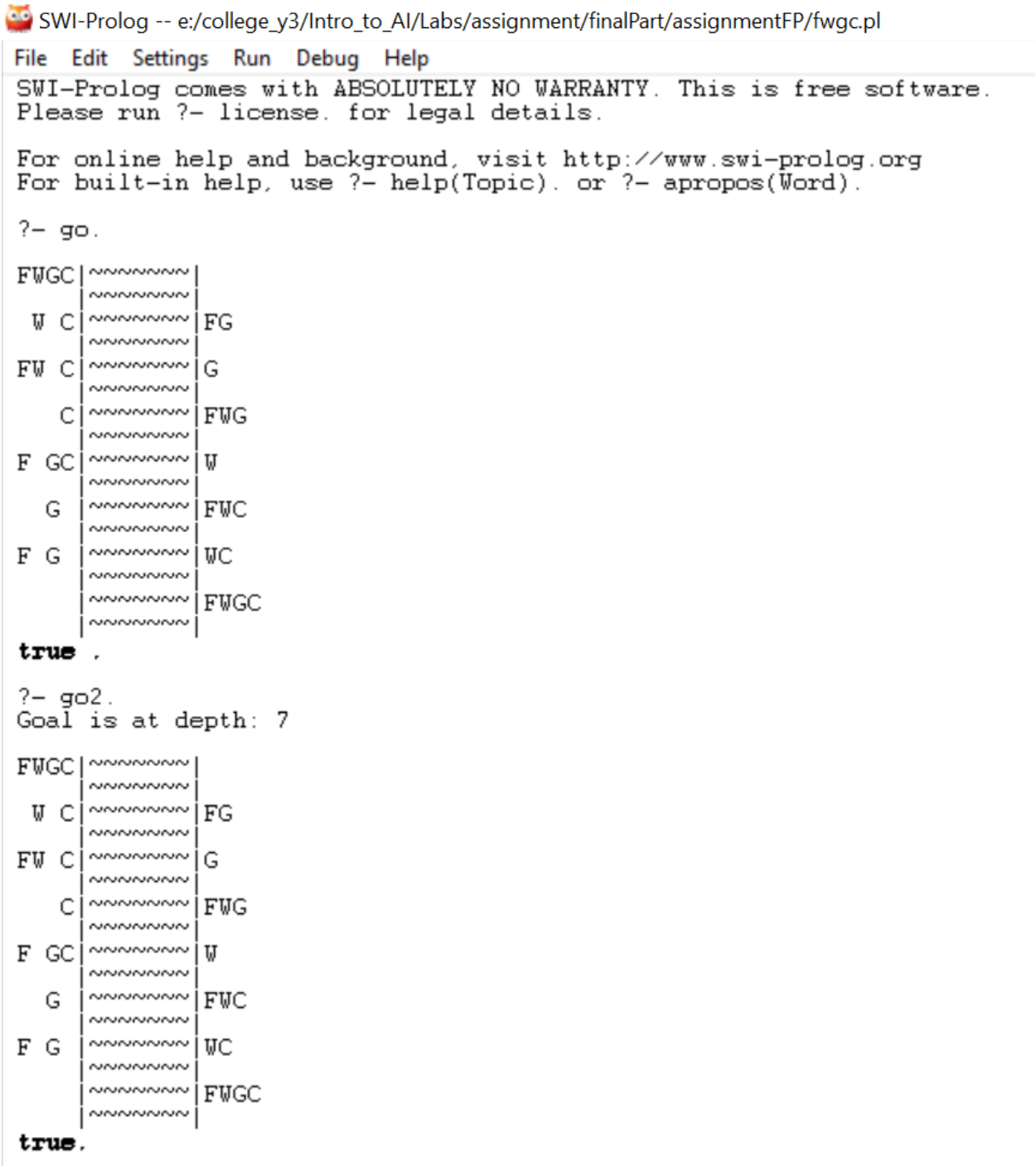
%%%%%%%%%%%%%%%%%%%%%

%To run the programme%

%%%%%%%%%%%%%%%%%%%%%

go2 :- id_solve(state(w,w,w,w),6,A),

reverse(A,A1),showPath(A1).



The farmer Wolf Goat Cabbage algorithm is a taking parts in a state and bring them over to the other side of the programme. We do this through the predicate of making sure the farmer is with all the danger parts of the states. These are the wolf can't be left alone with the goat or the goat cannot be alone with the cabbage. So, we set these rules in place. We then move all the objects in order. The farmer can be moved alone or can take one of the object with him. In our example we are trying to get all the objects to go from the west side to the east. We solve this method by getting all the objects to move from the west to the east, ensuring all the safe paths are took to get to that part. Once we traverse through each part that was safe, we store the path into the list, so we know what states we have visited already and what is yet to be visited.

Once all four objects get from the east bank to the west bank, the algorithm is considered complete.