

**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 through one state and using that path, 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. 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 state and try to pick which one has the least potential steps from its immediate step. This means it takes more time 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 two approaches have their advantages and disadvantages in their attempt to search for the most optimal path.

Iterative Deeping takes the best from both and searches for the most optimal path using both approaches. 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 IDDFS:

The main predicates for my ID\_DFS is the id\_solve(), as well as ids\_dfs(). These two methods work together to get to the goal state through the best features of breath first search and depth first search. This method is known as Iterative Deeping Depth first search. The rule id\_solve takes a state, a certain depth to go and passes these to the ids\_dfs as well as an empty list in order to find the solution to the path. If the depth is not sufficient, it will call itself to extend the depth, so it can continue to search for the most optimal path.

Through the algorithm, it will call on the other rules in the system to run the programme, so it can go through the states. Since the rules of the search method is independent of the other rules inside the system, the algorithm can be implemented into any programme and will run as behaved. After finding all the states from the start through the path to the solution, it will allow for visual representation of the states since the efficiency of the algorithm allows for solving of the state to be done within seconds, whereas Depth first search could take minutes to complete.

***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

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

% %

% Eights Puzzle - Specific Predicates %

% Eight.pl %

% %

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

% State represented by list of tile positions

% [t0, t1, t2, t3, t4, t5, t6, t7, t8]

%

% --->x

% | 1 2 3

% | 8 4

% | 7 6 5

% y

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

% 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

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

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 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: 4, 5, 6, 7, 8, 18"),

nl, read(I), nl,

start(I,A), id\_solve(A,I,B),

reverse(B,B1),showPath(B1).

A screenshot of a cell phone

Description generated with very high confidence

A screenshot of a cell phone

Description generated with very high confidence