

# AI Assignment Report

## Daniel Tilley – C14337041

### Search Strategies

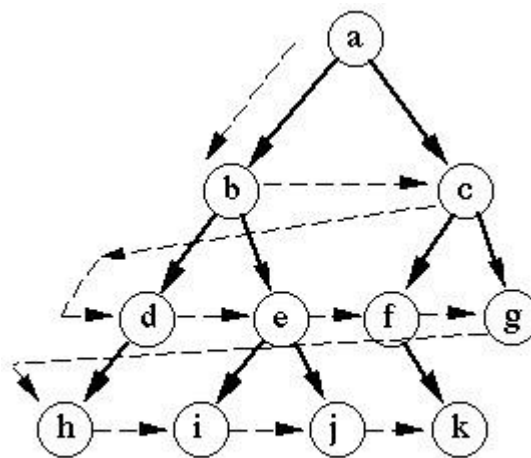
There are a few different search strategies that can be used with a program like this. The main one's I am going to discuss are:

1. Breadth First Search
2. Depth First Search
3. Iterative Deepening

I will discuss the pros and cons of each (focusing on solution distance and memory usage) and will pick a which one is best for the given problem.

#### *Breadth First Search*

This search strategy operates horizontally on a search tree. It searches through each layer (usually from left to right), dropping down a layer each time it is un successful until the correct node is found.



#### **Breadth-first search**

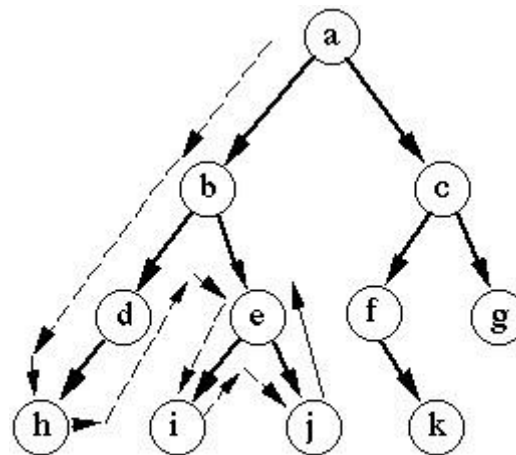
Using breadth first search means that you will always find the solution that is closest to you due to the way that BFS searches the tree. If we take C as the goal node for example and take A as our starting node, the solution will be two steps away (A -> B -> C) as opposed to other searching methods where the same solution could be many more steps away.

Although BFS is very good at finding the closest solution, one thing it is not great with is memory usage. The size of a stack using BFS could be  $K^D$  where K is the number of new states a current state can have and D is the depth of the tree. The reason for this is due to

the search methods horizontal approach meaning it will visit every node at a given depth and the width of each depth can increase exponentially as you move further down the tree.

### *Depth First Search*

This search strategy operates vertically on a search tree. It searches through each branch (usually from top to bottom), dropping down a layer each time and returning up the tree when there are no more nodes on a branch until the correct node is found.



**Depth-first search**

Unlike BFS, DFS will give you a longer solution path to what may seem like a relatively close solution. If we take the same example as previously discussed (A -> C) which only took two steps to solve using BFS, we will find that it will take us seven steps (A -> B -> D -> H -> E -> I -> J -> C). This shows us that DFS is not the most efficient at finding solutions that may in effect be quite close to the starting point.

What DFS is quite good at however, is memory usage. The size of a stack using DFS could be  $KD$  where  $K$  is the number of states a current state can have and  $D$  is the depth of the tree. This is because DFS will search the entire depth of the tree (unless of course a solution is found further up on the first branch) and the stack size will not need to be increased or decreased depending on what branch is searched.

### *Iterative Deepening*

ID is a way to take the best of both DFS and BFS and combine them into one. It takes BFS's ability to find the closest solution and merges it with the memory usage of DFS making it a clear winner for the problem given to us. ID uses a controller to restrict the depth it can travel to. This controller may be hard coded in a program or dynamically chosen by a user. The controller means that when searching, every node at a depth will be searched and the search will travel no deeper than the specified node.

## Code Explanation

Apart from the necessary predicates such as move (generates a new move) and manDist (calculates distance between tiles), there are three main predicates used for ID. They are:

1. IdDfs.
2. Solve/3
3. Solve/4

IdDfs is used to kick the program off, it calls a write statement to ask a user to enter a number, then calls a read to get that number. After the number is stored, start is called to determine which state is the starting state is. This state will be stored and passed to the solve/3 predicate (starts the solving of the problem and returns the solution state). After the solving has been done, the solution / path must be reversed as it is back to front. Once it has been reversed, it is then printed back to the user to see.

% This predicate is used to run the program using ID DFS

```
idDfs :- write('Start at depth 4 5 6 7 8 or 18 ? : '),  
        read( I ),  
        start( I , X ),  
        solve( X , Sol , 0 ),  
        reverse(Sol, Soln),  
        nl, showSolPath( Soln).
```

Solve/3 is our controller predicate and will be in charge of limiting the depth the program can search to. Solve/3 is passed a starting state (X), an empty list (Sol) and a depth to travel to (D). Solve/4 is called and passed the same arguments it received with the addition of an empty list which will store the path. If Solve/4 fails, then the depth is incremented and solve/3 is recursively called passing in the new depth (D1), otherwise the program, returns to IdDfs with the solution.

% control predicate

```
solve(X , Sol , D) :-  
    solve(X , [] , Sol , D),!;  
    D1 is D + 1,  
    solve(X, Sol, D1).
```

Solve/4 is the actual solving predicate and does things like generate moves, add new states to path etc. The first time Solve/4 is called, it will check if X is the goal state or where we want to finish and if it is, it will add X onto the path and return to Solve/3. Otherwise It will check that the depth is greater than 0 (So we don't get any errors or loop forever. Will return to Solve/3 if it fails). The predicate will then try to generate a move from current state (X) to new state (X1). It then check that the new state (X1) is not already on the path (P). If it is, it fails and try's to create a new move once again. If it is not, we decrease the depth by one and recursively call solve/4 passing it the new state (X1), the current state joined onto the path ([X|P]), the solution and the new depth (D1).

```
solve( X , P , [X | P] , D ) :- goal( X ).
```

```
solve( X , P , S , D ) :-
```

```
    D > 0,
```

```
    move( X , X1 ),
```

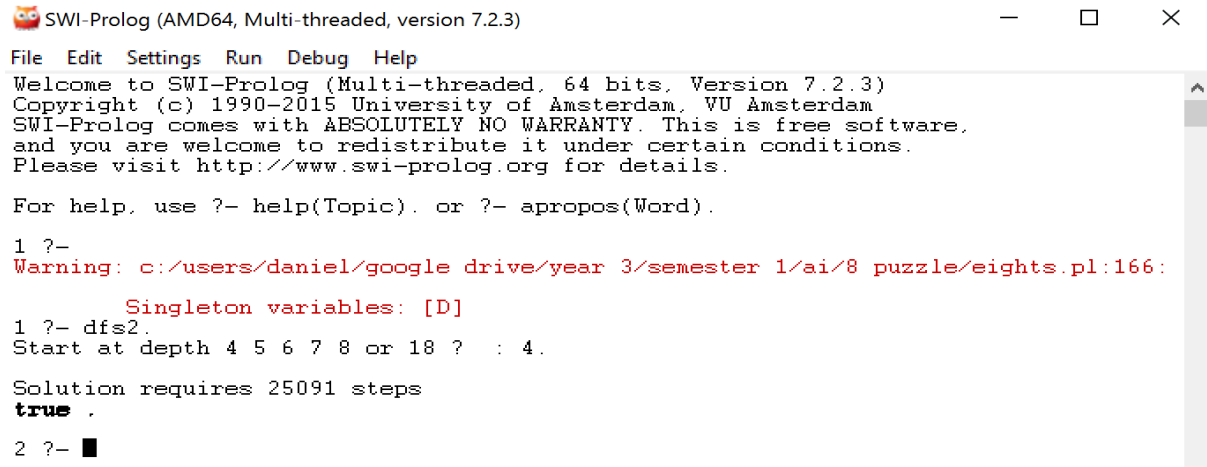
```
    not(member(X1, P)),
```

```
    D1 is D - 1,
```

```
    solve( X1 , [X|P] , S , D1).
```

## Screen Captures

### Depth First Search

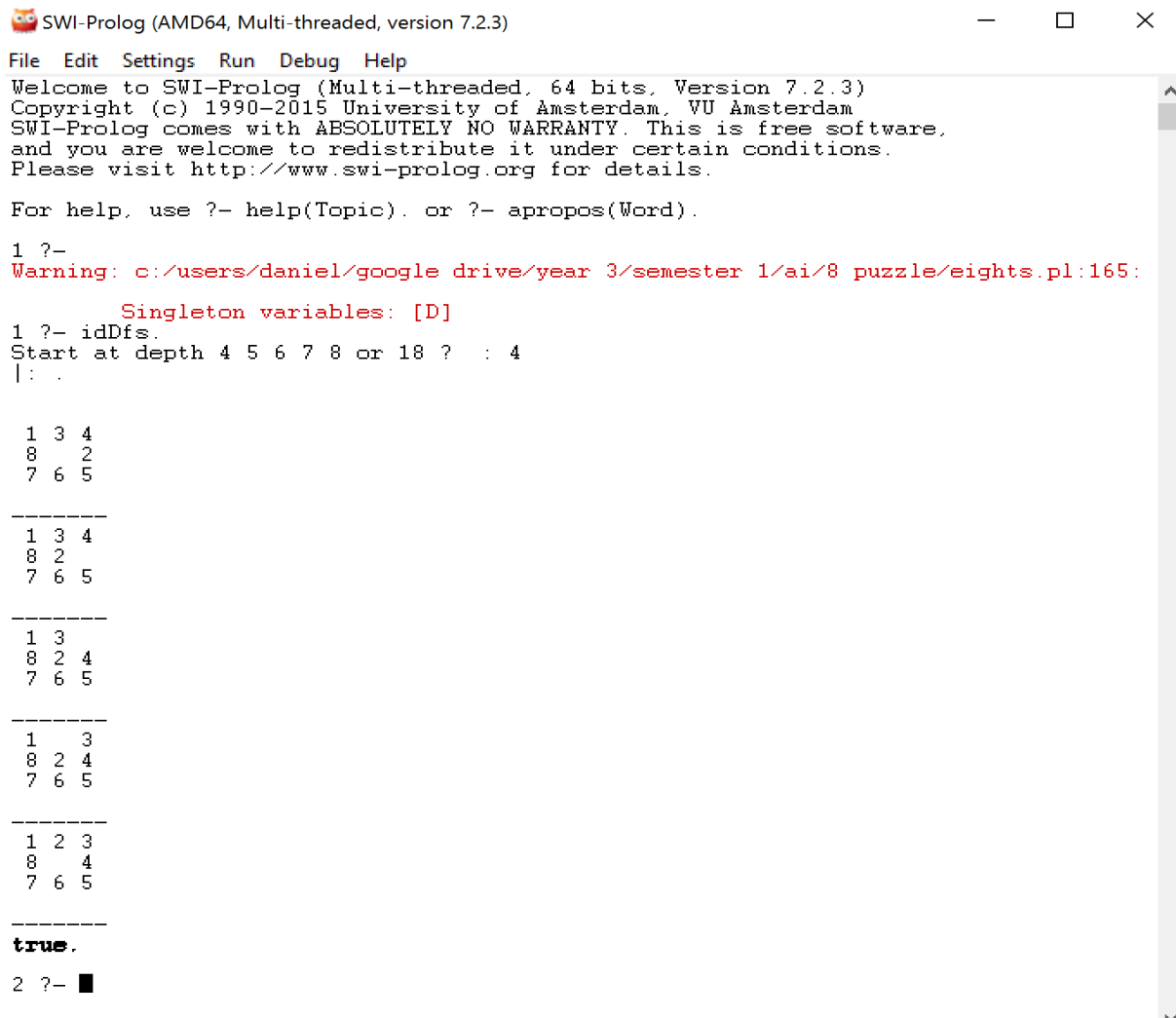


```
SWI-Prolog (AMD64, Multi-threaded, version 7.2.3)
File Edit Settings Run Debug Help
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.2.3)
Copyright (c) 1990-2015 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?-
Warning: c:/users/daniel/google drive/year 3/semester 1/ai/8 puzzle/eights.pl:166:
Singleton variables: [D]
1 ?- dfs2.
Start at depth 4 5 6 7 8 or 18 ? : 4.
Solution requires 25091 steps
true .
2 ?- █
```

### Iterative Deepening



```
SWI-Prolog (AMD64, Multi-threaded, version 7.2.3)
File Edit Settings Run Debug Help
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.2.3)
Copyright (c) 1990-2015 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?-
Warning: c:/users/daniel/google drive/year 3/semester 1/ai/8 puzzle/eights.pl:165:
Singleton variables: [D]
1 ?- idDfs.
Start at depth 4 5 6 7 8 or 18 ? : 4
|: .

1 3 4
8 2
7 6 5

-----
1 3 4
8 2
7 6 5

-----
1 3
8 2 4
7 6 5

-----
1 3
8 2 4
7 6 5

-----
1 2 3
8 4
7 6 5

-----
true .
2 ?- █
```