|  |
| --- |
|  |
| HIT3002 – Intro to AI |
| Assignment 1 – Tree Based Search |
|  |
| **Divyesh Prakash - 7170111** |
|  |

|  |
| --- |
|  |

Table of Contents

1. **Introduction3**

**2. Depth-First Search**4

**3. Breadth-First Search5**

**4. Greedy Best-First Search**6

**5. A\* (A Star)**7

**6. Custom Search Methods8**

6.1 Uninformed Custom Search8

6.2 Informed Custom Search8

**7. Conclusion9**

**8. References10**

1. Introduction

The NxM-puzzle problem is a variation of the 8 and 15-puzzle problems. Simply put, the puzzle involves a grid of N x M, where N and M are not necessarily equal. The grid can contain more than one blank spot, all remaining spots are numbered in ascending order from one. For example, if N = 4 and M = 6, and the puzzle included three blank spots there would be the numbers 1 to 21 randomly scattered throughout the grid. The aim of the puzzle is to rearrange the grid by sliding numbered tiles into blank spots until the desired goal state is reached.

The following is a definition of the NxM-puzzle problem. A problem can be defined formally be five components (Russel & Norvig 2010, p.66) in addition to a definition of the problems state descriptions:

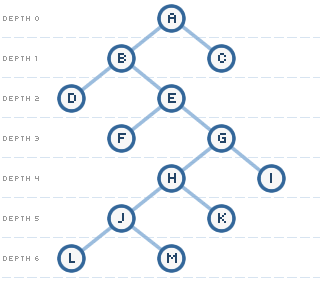
* **States**: A state description for this problem outlines the location of each tile, both numbered and blank, within the given grid.
* **Initial State**: Any valid state can be assigned as the initial state of the problem.
* **Actions**: Actions in this problem are allocated to the blank squares. The possible actions include moving *Up*, *Down*, *Left* or *Right*. However, not all of these actions are applicable at all times. For example, if the blank is on the right edge it would be unable to move right.
* **Transition Model**: When a valid Action is applied to a given State it will result in a new state comprised of the new location of each tile.
* **Goal Test**: This involves checking whether or not the current state matches the desired goal state. This test is applied after each Action to see if the resulting state of that action is the goal state.
* **Path Cost**: The path cost is used to calculate the most efficient way to traverse the resulting search tree once a solution is found. In this problem, each action costs 1 , so the path cost of a given state is equal to the number of actions taken to reach that state from the initial state.

Six algorithms and search methods were implemented to demonstrate various solutions to the NxM-Puzzle Problem. They are as follows:

* Depth-First Search implemented by Divyesh Prakash
* Breadth-First Search implemented by Charlotte Pierce
* Greedy Best-First Search implemented by Charlotte Pierce
* A\* (A Star) implemented by Divyesh Prakash
* Custom Algorithm 1 implemented by Divyesh Prakash
* Custom Algorithm 2 implemented by Charlotte Pierce

2. Depth-First Search

The Depth-First Search (DFS) algorithm is designed to always expand the deepest node in the current fringe. It continues to do so until a node no longer has any successors, at this point it will backtrack to the next deepest node which has successors that have not yet been explored. This is repeated until a solution is found. The diagram below illustrates a basic search tree.

  
(Kirupa 2010)

The order the nodes are searched in the above diagram, using depth-first search, will be: A, B, D, E, F, G, H, J, L, M, K, I, C (assuming none of those nodes contain the goal state).

The implementation of the DFS algorithm for the NxM-Puzzle uses the following pseudo code when selecting which node to expand.

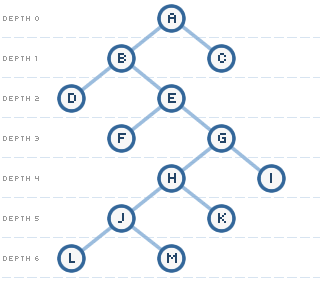
loop while (fringe\_size is greater than zero)  
 expand\_node(*last\_node\_in\_fringe)* add\_to\_search\_tree*(last\_node\_in\_fringe)* remove\_from\_fringe(*last\_node\_in\_fringe)*end loop

This method follows the principles of a DFS by always selecting the deepest node in the fringe. Once a node has been searched it is removed from the fringe, this means that the last node in the fringe will always be the deepest, unsearched node available to be searched.

Through the testing of the DFS algorithm, it was found to be extremely time and memory consuming compared to the informed search algorithms available. It would take hours and over 100mb of memory to solve the problem using the DFS algorithm.

3. Breadth-First Search

The Breadth-First Search (BFS) involves expanding all nodes at a certain depth before moving on to the next depth. In essence, this is opposite to a DFS. Rather than expanding the deepest node of the fringe as you would in a DFS, BFS expands the shallowest node in the fringe.

(Kirupa 2010)

The order the nodes are searched in the above diagram, using breadth-first search, will be: A, B, C, D, E, F, G, H, I, J, K, L, M (assuming none of those nodes contain the goal state).

The implementation of the BFS algorithm for the NxM-Puzzle uses the following pseudo code when selecting which node to expand.

loop while (fringe\_size is greater than zero)  
 expand\_node(*first\_node\_in\_fringe)* add\_to\_search\_tree*(first\_node\_in\_fringe)* remove\_from\_fringe(*first\_node\_in\_fringe)*end loop

This method is similar to that of the DFS but as explained earlier, uses the first node in the fringe. By selecting the first node of the fringe, we can navigate the fringe in order that the nodes are added. This results in a search that takes long to reach high depth.

As with the DFS algorithm, the BFS algorithm was found to be extremely time and memory consuming compared to the informed search algorithms available. It would take hours and over 100mb of memory to solve the problem using the BFS algorithm.

4. Greedy Best-First Search

Greedy Best-First Search (GBFS) is an informed search algorithm which makes use of a heuristic in order to determine which node is the ‘best’ to expand. Greedy Best-First Search attempts to expand the node that is ‘closest’ to the goal based on the calculations made according to the heuristic used.

A number of heuristics can be used to solve the NxM-Puzzle Problem, the two most popular are:

1. Missing Tiles: The total number of grid tiles that are not in their goal state.
2. Manhattan Distance: Using the sum of the Manhattan distance for each tile. The Manhattan distance is the grid-based distance between the tiles current position and its goal position. This is essentially the difference between the x and y co-ordinates of the current and goal position of the tile.

The heuristic chosen for this implementation is the Manhattan Distance as it provides a much more accurate representation of the current state of the problem and truly reflects the minimum number of moves required to solve the problem.

The implementation of GBFS uses the following pseudo code when selecting which node to expand.

loop while (fringe\_size is greater than zero)  
 expand\_node(*first\_node\_in\_fringe)* add\_to\_search\_tree*(first\_node\_in\_fringe)* remove\_from\_fringe(*first\_node\_in\_fringe)*  
 sort\_fringe // according to total cost  
end loop

The only difference between this implementation and that of a BFS implementation is the addition of the ‘sort fringe’ procedure. It is here where the heuristic calculations are applied to determine the ‘quality’ of each node and then sorting them based on which node is the best to search.

Greedy Best-First Search is notably faster than the previous two uninformed searches. It takes a few seconds to find a solution to a problem of similar complexity to the test input provided. The speed at which it solves the problem can be attributed to the fact that it takes into consideration an informed calculation.

5. A\* (A Star)

A\* (A Star) is an informed search algorithm, much like GBFS, which makes use of a heuristic in order to determine which node is the ‘best’ to expand. The key difference between A\* and GBFS is that A\* takes into consideration the path cost of the node. As a result, nodes with are easier to get to are generally searched before nodes that have a higher cost.

As with GBFS, the heuristic chosen for this implementation is the Manhattan Distance as it provides a much more accurate representation of the current state of the problem and truly reflects the minimum number of moves required to solve the problem.

The implementation of A\* uses the same pseudo code as GBFS when selecting which node to expand.

loop while (fringe\_size is greater than zero)  
 expand\_node(*first\_node\_in\_fringe)* add\_to\_search\_tree*(first\_node\_in\_fringe)* remove\_from\_fringe(*first\_node\_in\_fringe)*  
 sort\_fringe // according to total cost  
end loop

In addition to this, A\* also incorporates the path cost into the calculation for the total cost. Because of the nature of the NxM-Puzzle, where each step has a cost of 1, the path cost is simply equal to the depth of the search tree.

A\* solves the solution in exactly the same time as GBFS. This is because each step has a cost of one, as a result there is no real difference between the result of the heuristic calculations of each algorithm. Theoretically speaking, A\* would actually perform slightly slower than GBFS in this situation as it has to calculate the path cost for each node.

6. Custom Search Methods

# 6.1 Uninformed Custom Search:

The uninformed Custom Search algorithm implemented is similar to the breadth and depth-first searches as it does not use any heuristic to determine which node to expand (hence being uninformed). In fact, unlike breadth and depth-first searches, this algorithm uses no logical method for selecting which node to expand. Rather, it selects a completely random node to expand and search. As a result, it is nearly impossible to determine how long a given problem (assuming similar complexity to the test input provided) will take to solve as it is completely random.   
  
The pseudo code for selecting which node to expand is as follows:

loop while (fringe\_size is greater than zero)  
 generate random number  
 expand\_node(*random\_number index of fringe)* add\_to\_search\_tree*(random\_number index of fringe)* remove\_from\_fringe(*random\_number index of fringe)*   
end loop

This search is much too random and unpredictable to be considered useful by any measure. it does, however, have the possibility to solve the solution quicker than any other algorithm but this occurring is extremely unlikely.

# 6.2 Informed Custom Search:

The Informed Custom Search algorithm implemented is very similar to the A\* algorithm. They vary in that the custom algorithm uses a different heuristic to calculate the total cost. The custom algorithm uses a heuristic which calculates the total number of tiles that are not in their goal position. For example, if 8 tiles are in place and 7 are not then the total cost will equal 7 + the path cost for the current node.

7. Conclusion

After developing and testing numerous algorithms to solve the NxM-Puzzle Problem, it can be concluded that the GBFS and A\* methods are the most efficient as solving the problem in terms of both time and memory complexity.   
  
These two informed searches perform drastically better in that they are capable of providing a solution in a reasonable amount of time without consuming large amounts of memory. Both GBFS and A\* perform at the same level for the problem.

Given the problem, GBFS in reality performs slightly better than A\* as it is not required to calculate the path cost. If, however, the path cost was variable and not fixed at 1, then A\* would most likely outperform GBFS.

8. References

Kirupa 2010*, Depth First and Breadth First Search,* viewed 21/04/2011, <<http://www.kirupa.com/developer/actionscript/depth_breadth_search.htm>>

Russel, S & Norvig, P 2010, *Artificial Intelligence: A Modern Approach,* 3rd edn, Pearson Education, New Jersey, USA.