Introduction to AI Assignment 1

Tree-Based Search : The NxM Puzzle

Charlotte Pierce

7182139

Table of Contents

1.0 The NxM Puzzle

The NxM puzzle is a generalised representation of the well-known ‘sliding block’ or ‘sliding tile’ puzzle. The puzzle is in the form of a grid, with a number of sliding tiles. In the generalised version, this grid has the dimensions NxM, and thus can have NxM minus 1 sliding tiles (at least one tile must be removed so that the other tiles can be manipulated by the player).

The aim of the puzzle is to rearrange the tiles on a scrambled grid into some specific order, usually the numbers being placed in ascending order from left to right, with the empty tile(s) in the bottom right-hand side of the grid. The player does this by sliding the tiles on the grid into the empty space(s) until they have reached the goal configuration. The tiles cannot be lifted off the grid.

The original form of the puzzle was the 15 puzzle, where players attempted to rearrange a 4x4 grid with one blank square, and 15 numbered tiles (hence the name ’15 puzzle’), but is more commonly seen now as a 3x3 grid with either the numbers 1 – 8 (and one blank tile), or a picture for the player to unscramble.

The NxM puzzle, as solved in this document’s accompanying application, is able to solve the general NxM puzzle problem, with any number of blank tiles. This can be accomplished with any one of six search algorithms, as specified by the user, each of which is discussed below.

2.0 Search Tree

A search tree is a data structure fundamental to search algorithms and artificial intelligence in general. A tree is a structure made of a number of nodes. In a search tree a node represents a possible state in that which is being searched – in the NxM puzzle solver this would be a particular state of the tiles in the puzzle grid. The search tree begins with the root node, from which other nodes are created, or ‘branch’. Each node created is referred to as a ‘child’ of the node which created it (with the originating node being referred to as its ‘parent’).

Each node can have any number of children (with the exception of a binary tree, where each node must have exactly two children), but each node can only have one parent. In a search tree, this characteristic makes it simple to find the solution path, once a goal node (i.e. a node representing the goal state) has been found. When a node is created, it has a certain depth value. The root node has a depth value of 0, with each of its children having a depth value of 1, with each ‘layer’ of children having a depth increased by one from their parent node. Depth is an essential value for some search algorithms, as it can determine the order in which nodes are searched (for example with breadth-first search).

The search tree is a highly useful structure for solving problems, because each problem state can be easily represented as a node. Beginning at the root node, each node can ‘spawn’ a number of children, representing the possible moves from that node, and the resulting states, thus expanding

the tree. This process can be repeated, with generated nodes being created and searched according to the algorithm chosen, until the goal node is found and the solution path can be created.

When using a search tree to solve a problem such as the NxM puzzle, a similar structure is used regardless of the search algorithm. For any search, a fringe must be obtained. The fringe is a list of unexpanded nodes, which (depending on the search algorithm) may have already been checked and found to not contain the goal state, or not yet checked for the goal state. Regardless, the fringe remains the central point of the search algorithm. The main difference between different search approaches is the way in which nodes within the fringe are sorted (thus the order in which they are chosen for expansion), and on what grounds nodes are added to the fringe. These and further differences between each algorithm used in the NxM Puzzle Solver are discussed below.

3.0 Algorithms

Six different search algorithms have been implemented in the NxM puzzle solver. An overview of each algorithm and the differences between them is provided below.

3.1 Breadth-First Search (BFS)

Implementation By: Charlotte Pierce.

Breadth-first search is a simple, inefficient search algorithm. As with all tree search algorithms, search begins with the root node. The algorithm then searches the root node's children, before expanding each of them in turn, and exploring their children. Search operates on a 'layer-by-layer' basis, with the algorithm searching and expanding all nodes of the same depth, before moving on to the next depth. Figure 1 below shows the order in which nodes would be searched in a breadth-first search.

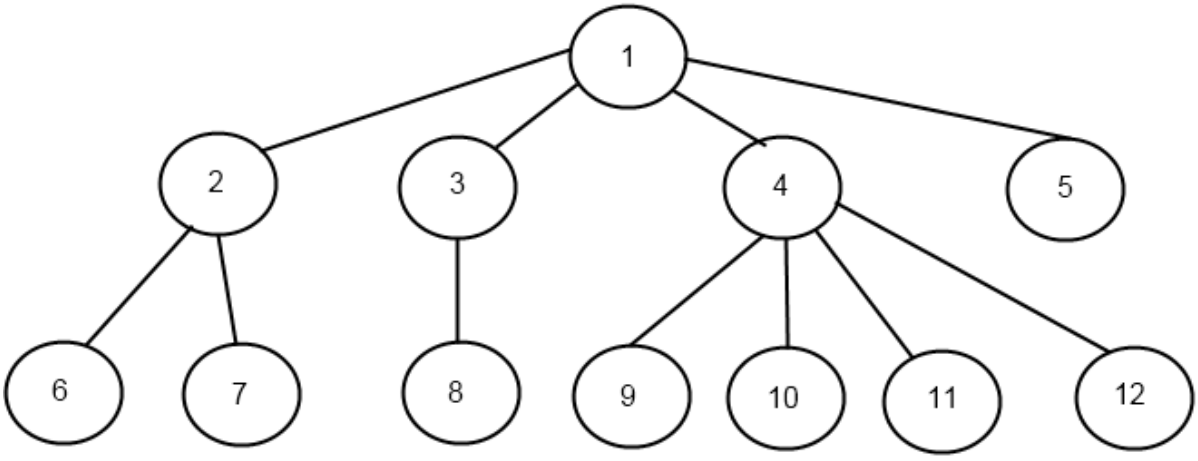

Figure 1 Node Search Order (BFS)

Assuming that every node in the search tree has a finite number of children, breadth-first search will always find a solution. However, it is an inefficient search method for any but the most simple

problems, as it is an uninformed search (i.e. does not use any data from within the node to determine a 'promising' path), thus searches each node without any focus or direction. Given the large number of children each node can have in the NxM puzzle (up to 4 new nodes for each blank tile in the puzzle grid), breadth-first search is only able to generate a solution for scrambled puzzle configurations which take very few moves to reach the goal state (i.e. when the goal node is at a low depth in the search tree). As the depth of the goal node increases, the time needed for the algorithm to find the solution also increases, but in a highly undesirable exponential pattern.

Breadth-first search can be made slightly more efficient by checking for repeated states. This requires the maintenance of an 'explored set' list. Similar to the fringe, the explored set contains a list of all nodes which are not the goal, and have been expanded. Before adding a newly generated node to the fringe, it can be compared against all nodes in the explored set. If there is already a node with the same state in the explored set, the new node does not need to be added to the fringe. However, despite this improvement, breadth-first search is still a highly inefficient algorithm, unsuitable for any but the most trivial instances of the NxM puzzle problem.

Because uninformed search algorithms are highly inefficient for this type of problem, they were primarily tested on scrambled configurations which took at most 2 – 3 moves to solve. As stated, breadth-first search expands a layer of nodes at a time. With the puzzle configuration set with only one blank square moved upwards, breadth-first search generates a total of four new nodes. However, with both blank squares (in a 3x5 puzzle grid) moved upwards, the search algorithm generates 24 nodes before finding the goal. This number increases greatly the more moves added, with a starting configuration needing six moves to solve generating between 1000 and 3000 nodes before finding the solution.

3.2 Depth-First Search (DFS)

Implementation By: Divyesh Prakash.

Depth-first search is a similar search algorithm to breadth-first. However, rather than search each layer of nodes as in breadth-first, depth-first follows one path of nodes until it gets to the 'bottom' of the search tree. Once the search process reaches a node with no children, it backtracks and takes the next deepest path. Essentially, where breadth-first search evaluates and expands the node in the fringe with the lowest depth, depth-first search evaluates and expands those with the highest depth. The order in which nodes would be searched in a depth-first search is shown in figure 2, below.

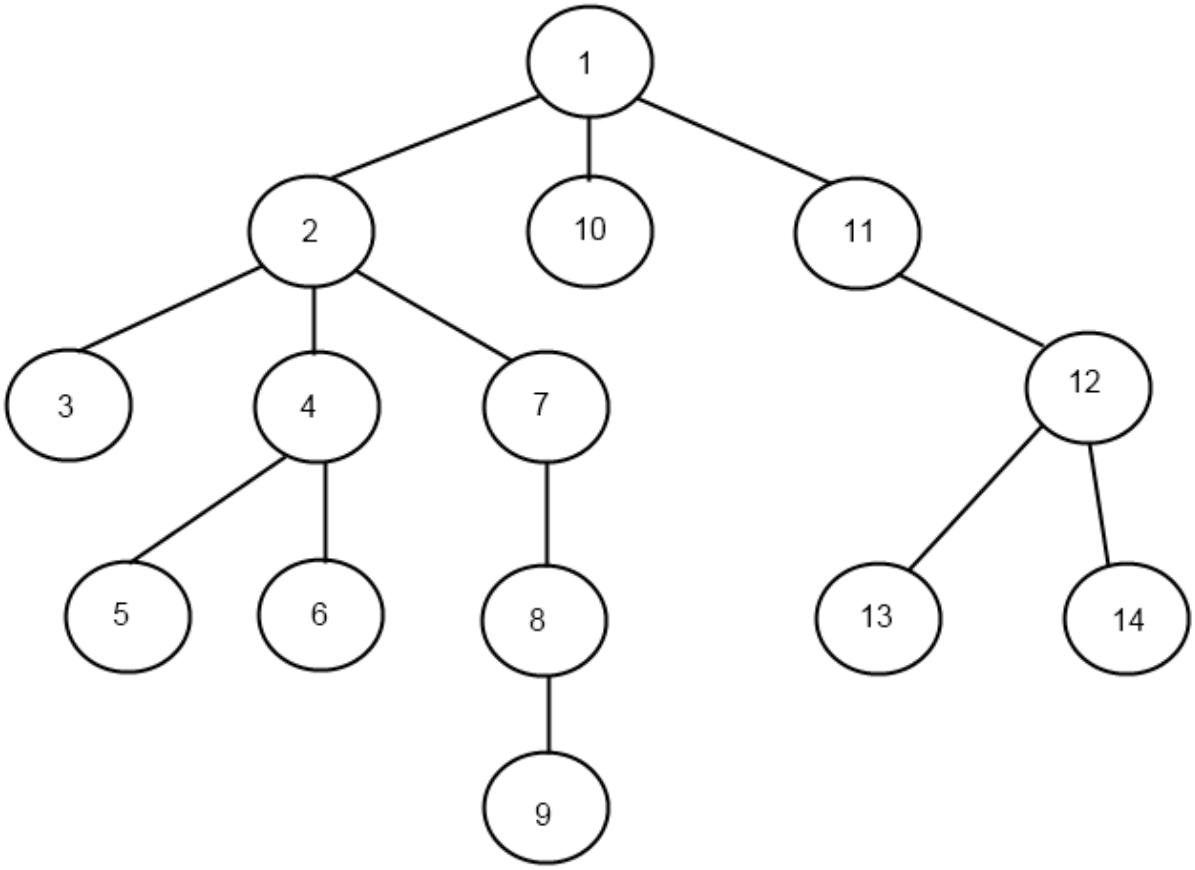

Figure 2 Node Search Order (DFS)

Assuming that the search tree has a finite depth, depth-first search will always find a solution. In the NxM puzzle problem, a finite search tree can only be achieved if an explored set is maintained and newly generated nodes are checked against it before being added to the fringe, as discussed above in breadth-first search. If the explored set list were not used, the algorithm would never find the solution, as it would continue generating nodes at greater and greater depths, because a node on the tree will always have at least one move that can be made. The explored set stops these infinite searches from occurring.

However, much like breadth-first search, depth-first is an uninformed search, thus traverses the search tree in a linear fashion without placing any focused effort on getting to the goal node. Due to this characteristic of the algorithm, depth-first search is just as inefficient as breadth-first search, and its use must therefore be confined to only the simplest instances of the NxM puzzle problem.

Depth-first often seems to be more inefficient than even breadth-first search when used on the NxM puzzle problem. The reason for this is because of the fact that it searches deeply, rather than widely. The depth of the search tree, even with checking for repeated states, is many orders of magnitude larger than the possible breadth of the tree, meaning breadth-first search sees a larger ‘variety’ of node states than depth-first search.

This is shown by the results gained from the program. A starting configuration where only one blank tile is moved upwards is solved quickly, generating only 4 new nodes (as in breadth-first search). However, move the same blank tile upwards once more and depth-first search does not find a

solution in any reasonable amount of time. The same applies for any form of complex starting configuration.

3.3 Greedy Best-First Search (GBFS)

Implementation By: Charlotte Pierce.

Greedy best first search is an informed search algorithm (i.e. it uses data from within the nodes to choose more 'promising' paths), which assigns a heuristic value to each node so that it can determine which paths are more likely to contain the goal node. Being an informed algorithm, it requires that each node contains data on its evaluation function (the evaluation function being the value representing how 'close' a node is to the goal, with a value of 0 denoting the goal node itself).

In greedy best-first search, the algorithm tries to get as close to the goal as possible with each step, without caring about the path cost – so chooses the node in the fringe which is closest to the goal according to the heuristic (the node 'closest' to the goal will have the lowest heuristic value). Given that greedy best-first search does not care about the path cost to a node, the evaluation function is equal to the heuristic value for each node. Figure 3 below shows the order in which nodes would be search when using a greedy best-first search algorithm (the heuristic for each node is fabricated in order to illustrate the concept).

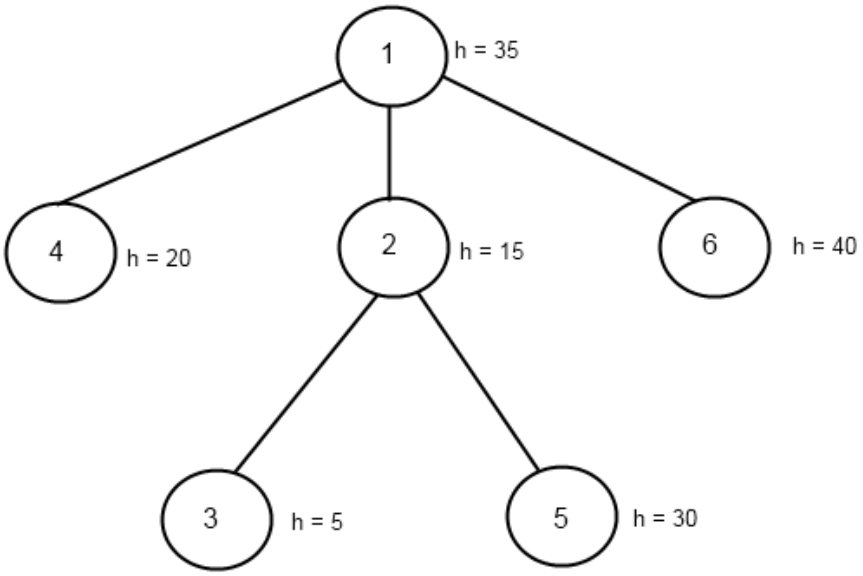

Figure 3 Node Search Order (GBFS) - 'h' refers to the heuristic value for each node

The effectiveness of greedy best-first search is highly dependent on the heuristic used. For example, in the NxM problem, two popular heuristics are:

1. The number of grid tiles that are not in their goal state.
2. The sum of manhattan distances for each grid tile (where the manhattan distance is the sum of the difference between the *x* and *y* co-ordinates of the grid tiles' current position, and its goal position).

Greedy best-first search still requires an explored set to be maintained, not to avoid infinite loops as in depth-first search, but simply to increase the efficiency of the algorithm, as in breadth-first search. If an appropriate heuristic is chosen, greedy best-first search will invariably outperform both depth-first and breadth-first search algorithms, simply because it actually 'looks' at each node and uses the information within to its benefit, rather than blindly search all nodes available in a linear fashion. As such, with a good heuristic, greedy best-first search can be used effectively on large and complex instances of the NxM puzzle problem to provide a solution in a reasonable amount of time.

On the surface, greedy best-first search seems inefficient compared to breadth-first search. On the starting grid configuration of one blank tile moved upwards (where breadth-first search generated 4 new nodes), greedy best-first search generates a total of 7 new nodes before finding the solution. However, this can be attributed to the fact that breadth-first search checks whether a node contains the goal configuration before adding the node to the fringe, whereas greedy best-first search, being worried about the path cost, only checks whether a node is the goal when that node is selected for expansion.

Further benefits of greedy best-first search can be seen when a complex start configuration is used. A problem requiring 60 moves to solve cannot be solved using breadth-first or depth-first search in anything resembling a reasonable amount of time (tests in the lengths of 10+ hours still did not garner a solution from either algorithm). However, the same problem was solved by greedy best-first search using the manhattan distance heuristic in a matter of seconds, generating 3558 new nodes (breadth-first search is estimated to generate approximately 8^60 nodes in the worst case for the same problem).

3.4 A-Star (AS)

Implementation By: Divyesh Prakash.

A star (A*) is an informed search algorithm which is very similar to greedy best-first search. The only difference between greedy best-first search and A* is that A* takes into account the path to get to a node. Thus, nodes which are 'easier' to get to are given a higher search priority than 'harder-to-get-to' nodes with the same heuristic value. Figure 4 below illustrates this concept.

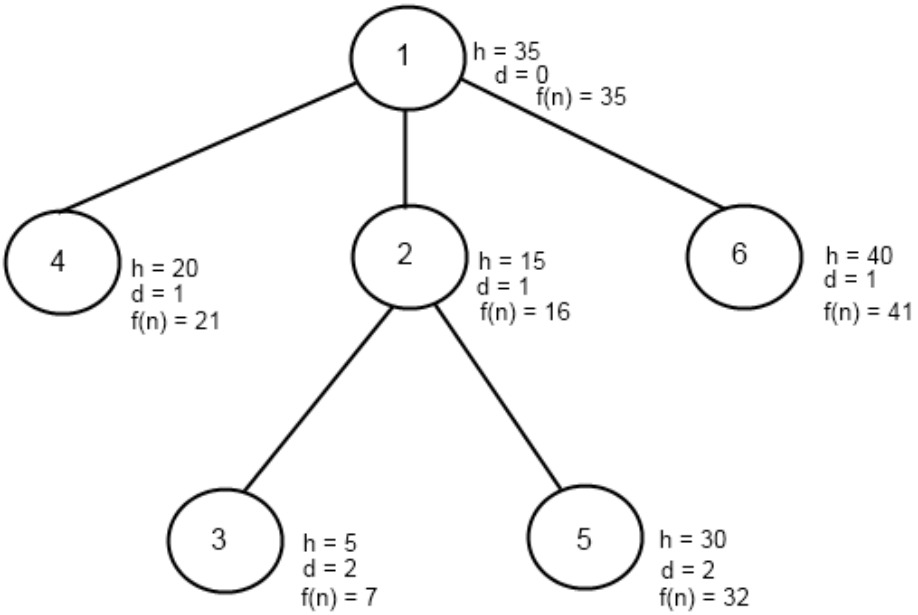

Figure 4 Node Search Order (A*) - 'h' is the heuristic value; 'd' is the path cost and 'f(n)' is the total cost (i.e. the evaluation function) for each node

Much like greedy best-first search, A* is a feasible solution for large, complex configurations of the NxM puzzle as long as a good heuristic is chosen. An explored set is not essential when using A*, because identical nodes created will have a higher depth than their counterparts, thus are likely to have a higher path cost and given a lower search priority regardless. In some search problems (such as searching for a route from point A to point B), searching for these repeated states can negatively affect the search process. However, in the NxM puzzle problem, this is not so, and nodes can be checked against the explored set for a slight (but negligible) efficiency increase.

Given that the major benefit of A* is that it takes into account the path cost of a node, it proves to be no more efficient than GBFS on the NxM puzzle problem. This is due to the fact that each 'move' in the puzzle will always have the same cost, thus calculations performed by the A* algorithm to determine the path cost of a node is, if anything, a hindrance of its performance given that the resulting value will have no impact on the order in which nodes are searched when compared to greedy best-first search.

The A* algorithm, for all intents and purposes, is identical to greedy best-first search when they both use the same heuristic, because path cost is irrelevant in the context of the puzzle (for reasons outlined above). As such, the A* algorithm performs identically to greedy best-first search, solving problems in the same amount of time and generating exactly the same number of new nodes.

3.5 Custom 1 (CUS1)

Implementation By: Divyesh Prakash.

The first custom search is similar to the breadth-first and depth-first algorithms in that it is uninformed. However, in its implementation it is unlike any other algorithm discussed. Rather than trying to create some form of order, as even breadth-first and depth-first do, this search chooses randomly from the fringe. This is illustrated in figure 5, below.

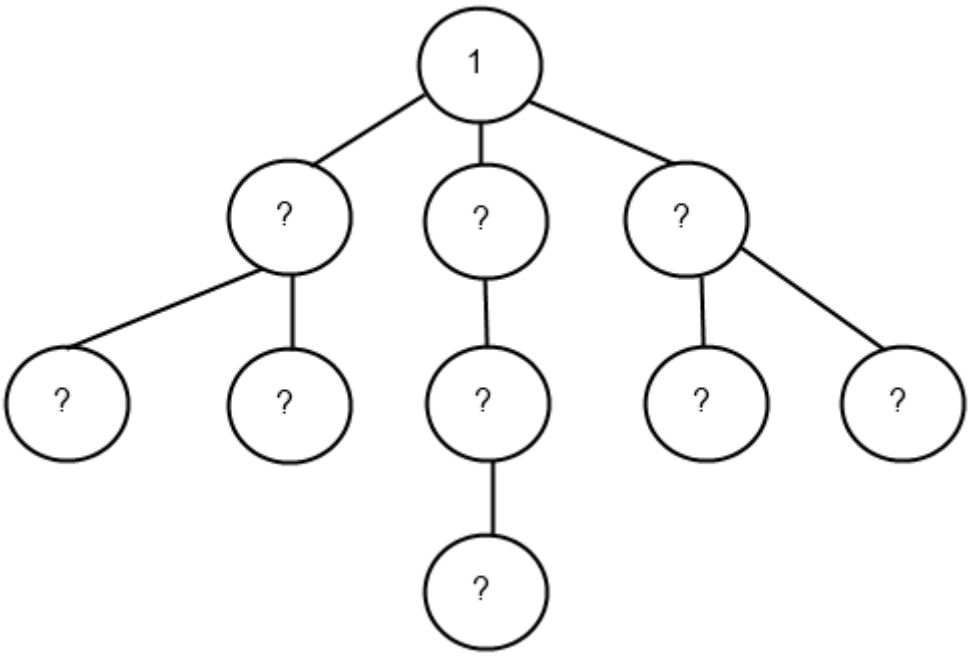

Figure 5 Node Search Order (CUS1)

This search algorithm could be horribly inefficient or highly efficient, depending on the 'luck of the draw' in which node from the fringe is chosen for expansion. The chances of the algorithm can be slightly improved by using the explored set, but it is still a horrible choice for all instances of the NxM problem, as there is no consistency and no guarantee that the goal node will be found in a reasonable amount of time for even the simplest scrambled configurations.

As expected, this algorithm performs sporadically on any complex problem. For simple problems it performs similarly to breadth-first search. However, get more complex, and it could find a solution in a matter of seconds, or seemingly never. The greater the number of moves required to find a solution, the less likely the algorithm is to find it quickly, as the number of nodes in the fringe to choose from will be larger.

3.6 Custom 2 (CUS2)

Implementation By: Charlotte Pierce.

The second custom search algorithm is informed, so uses a heuristic value like greedy best-first search and A*. This algorithm is an extension on the A* algorithm already implemented, used to prove the difference in performance caused by using a different heuristic. As such, the second custom search uses a heuristic known to be less effective than that used in the implementation of A*. The algorithm also uses the path cost of a node, as A* does, so that the substantial decrease in performance can be attributed to nothing but the change in heuristic.

This algorithm performs badly for an informed algorithm, especially considering the success of the greedy best-first search and A* algorithms on the same problems. On simple problems it performs identically to the other informed algorithms, generating 7 new nodes when one blank tile has been moved. However, beyond this configuration it becomes steadily less efficient, generating 20 new

nodes when both blank tiles have been moved upwards once, where greedy best-first search and A* both generated only 14. Its inefficiency only increases as the complexity becomes greater, unable to generate a solution for the 60-move configuration mentioned for A* and greedy best-first search in any reasonable amount of time.

4.0 Implementation

Every algorithm implemented was done so as an extension of an abstract class *Generic Solver*. During development it was noted that many methods being created were applicable to every algorithm that was to be implemented. Thus, the relevant methods forming the core of the solution were abstracted into the *Generic Solver* class, to facilitate code reuse.

Generic Solver includes functions to create a solver, complete with the root node. It also gives the programmer a complete method for expanding a particular node – a method which searches through the given node's state and creates a new node for every possible move that could be made (give the height and width of the puzzle grid, and the number of blank tiles). Given that not all the search algorithms use path cost or heuristics, these features were not included in the *Generic Solver*, but left to the specific solver class. The *Generic Solver*, however, does include functions for testing whether a node contains the goal state, and testing against previously generated nodes in the fringe or explored set.

4.1 Breadth-First Search (BFS)

Breadth-first search is one of the most simple out of the six implemented. It is an uncomplicated extension on the *General Solver*, using almost only methods from the solver in order to function. The central method to the solver, *run()*, is implemented simply, as shown by the pseudo-code below.

```
Repeat while still nodes in fringe or until goal is found
     get first node from fringe
     expand node
     add node to explored set
     remove node from fringe
End repeat
```

As shown, the method simply loops whilst there are still items in the fringe, removing the first item for expansion each time. Given that new nodes generated on expansion are automatically added to the end of the fringe, the breadth-first solver does not need to sort the fringe, because shallower nodes will already be placed at the front as the method loops.

4.2 Depth-First Search (DFS)

Depth-first search is implemented similarly to breadth-first search, as a simple extension on the *General Solver*. The only difference is that the loop that processes the nodes in the fringe, rather

than removing the first item in the fringe, removes the last. As described above, when a node is expanded it is added to the end of the fringe by the *General Solver* class. In breadth-first search this worked well, as it automatically places deeper nodes at the end. For depth-first search, it is just as effective. The pseudo-code for this method is shown below

```
Repeat while still nodes in fringe or until goal is found
     get last node from fringe
     expand node
     add node to explored set
     remove node from fringe
End repeat
```

4.3 Greedy Best-First Search (GBFS)

Greedy best-first search is the first informed search algorithm discussed. Because it is informed, it requires a slightly more complex extension on the *General Solver*, with additional calculations being required to calculate the heuristic value. However, this added complexity only requires a seemingly small addition compared to the breadth-first and depth-first searches, as shown below.

```
Repeat while still nodes in fringe or until goal is found
     get first node from fringe
     expand the node
     add node to explored set
     remove node from fringe
     sort the fringe according to total cost
End repeat
```

As seen, only a single line has been added to the main loop, which sorts the fringe. The functionality allowing nodes to be compared to one-another is implemented in the *Node* class, where it specifies that nodes with a higher total cost should be placed at the end of the fringe when compared to another node with a lower total cost. The main loop in greedy best-first search simply calls on this functionality.

Greedy best-first search also requires that each node, before being added to the fringe, has its heuristic calculated. This is implemented in a method *updateHeuristic(Node aNode)*, which automatically calculates the chosen heuristic for a node. The use of a separate method to implement this allows the heuristic calculations to be changed or modified without requiring changes to other aspects of the program.

4.4 A-Star (AS)

A* is implemented almost identically to greedy best-first search. As discussed, the only difference between the two is that A* requires the path cost of a node also be calculated and added to a nodes total cost value. Where greedy best-first search uses the method *updateHeuristic(Node aNode)*, A* extends this slightly, also updating a nodes path cost to be one move deeper than its parents (in the NxM puzzle each move only ‘costs’ 1). The calculation of the heuristic for each node is shown below.

```
While there are still grid tiles to process
     if grid tile not in its goal state
          get manhattan distance for tile
          add manhattan distance to heuristic cost
     end if
     get path cost for node
     update path cost for node
end while
```

4.5 Custom 1 (CUS1)

The first custom search (uninformed – randomly choose a node from the fringe) contains the most unique implementation from the other search algorithms. However, the main loop is still very similar, as can be seen by the pseudo-code below.

```
Repeat while still nodes in fringe or until goal is found
     generate random number
     get node at random number index
     expand node
     add node to explored set
     remove node from fringe
End repeat
```

Rather than sorting the fringe, or taking a node directly from one end of the fringe, as the other search methods do, the first custom search generates a random number from the range of indexes currently available in the fringe (i.e. from 0 to the fringe size less 1). It then uses this random number as an index to select a node from the fringe. The node is processed as in breadth-first and depth-first search, and removed from the fringe accordingly.

4.6 Custom 2 (CUS2)

The second custom search (informed), is almost identical to A*. The only difference between the two is the heuristic used. Where A* uses the sum of manhattan distances for each node, this custom method uses the number of nodes which are not in their goal state. As outlined, the use of a separate method to calculate and update the heuristic value of a node allows the calculation of the heuristic to be changed easily. In this case, it was changed from manhattan distance to the simple sum already mentioned. The calculation of the heuristic is shown in the pseudo-code below.

```
While there are still grid tiles to process
     if grid tile not in its goal state
          add one to total
     end if
     get path cost for node
     update path cost for node
end while
```

5.0 Conclusion

5.1 Improvements

One possible improvement for both the A* and greedy best-first search algorithms would be to improve the sorting process. Currently, the sorting process requires the program to sort through all nodes in the fringe, and place them in ascending order according to their total cost values. If the node(s) recently end up close to the beginning of the fringe, the algorithm has processed all nodes afterwards for no reason.

To remove this inefficiency, the program could be changed to loop through the fringe when a new node is going to be added, and insert the node after one with the same or lower total cost value, but before a node with a higher total cost. This would remove the need to process all nodes in the fringe (an expensive operation if there are a large number in the fringe), and only process the nodes up to the point at which the new node would be inserted. However, in most instances of the NxM puzzle problem, the performance benefit gained by making this improvement would be negligible, and unlikely to be noticeable by the user.

5.2 The Best

It becomes clear when using the program that informed search algorithms are the most effective at solving the NxM puzzle problem. However, it has also been shown that even an informed algorithm can fail on complex problems if the heuristic is not appropriate. Given the difference in performance between the A*, greedy best-first search and custom 2 algorithms, it is clear that manhattan distance is the greater heuristic of the two used.

Both A* and greedy best-first search can be categorised as the best algorithms for the NxM puzzle problem. This is because, as discussed, they are essentially the same. The benefit of A* over greedy best-first search in general, is that A* takes into account the path cost to get to a node. However, because the path cost is always the same, regardless of the move(s) made, A* holds no benefit over greedy in this regard. Both algorithms perform identically, and are equally effective at solving the NxM Puzzle Problem.