CS 26110: 8 Puzzle Solver

Due on Tuesday, November 19, 2013

James Euesden - jee22

Contents

Introduction

This assignment tasked me to create a solution to solving an 8 Puzzle (`http://en.wikipedia.org/wiki/15_puzzle`), using Breadth First Search, Depth First Search and A* Search, using 2 heuristics. This document goes with my application in backing up my choices, testing carried out, the design of algorithms and explanation of specifics.
Particular things of note about the assignment: The search methods were predfined , I had been provided with 3 different sets of tests to start with and no GUI was necessary so all operations can be carried out on the command line. I wrote the application in Java using Eclipse IDE, and any results of the application shown in screenshot are from the Eclipse UI.

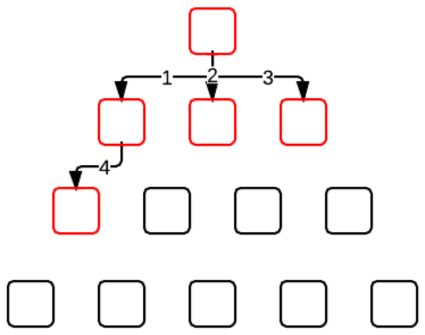
Analysis

There were two things I had in mind when thinking up what to build the data structure around: Time and Space. Whatever data structure I used had to be simple, maintainable, easy to access and not take up vast amounts of space. My first thought was a 2D array of integers/chars to represent a 3x3 Grid 8 Puzzle from 0-8. However, I would be accessing the 2D array often, involving the use of a double for loop, increasing the time of each search. Considering I knew files would be read in as Strings, I opted to keep the data as a String to represent each State.
As a String, the data can easily be accessed, won't be accidentally changed (Strings being immutable) and can be compared against other Strings/States easily using .equals(Object o). While I tried to avoid using double for loops, you will see later in my code that it became difficult to avoid when working with the Manhattan distance. As well as the Tiles (elements) of the grid in a String, other data needed to be kept, such as the depth of the search, any costs (Path and Heuristic) and the parent of a state. To contain each of these States of the Grid, I used a class 'GridState'. This will be described in more detail in my Class Description section.
For each different type of Search, there is a different data structure used:

Breadth First Search

BFS involves searching each top level State first (neighbours), before moving on lower in the 'tree' of States. The best way to represent this is the use of a Queue. A Queue is 'First In, First Out', meaning that the top level States that are put into the list will always be the first out of the list.

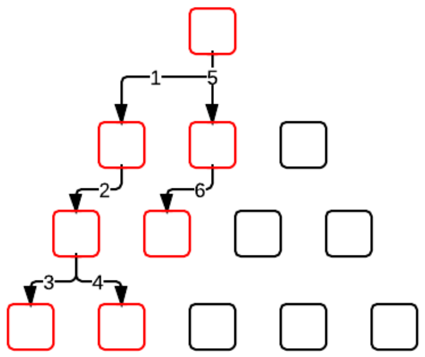
Figure 1: BFS - Searching top level states before their successors



Depth First Search

Unlike BFS, DFS searches through the successors of the first top level State found before moving onto neighbours, searching the deepest points. To represent this I needed a Stack, employing "Last In, First Out" methodology, ensuring that the deepest States are searched before the neighbours.

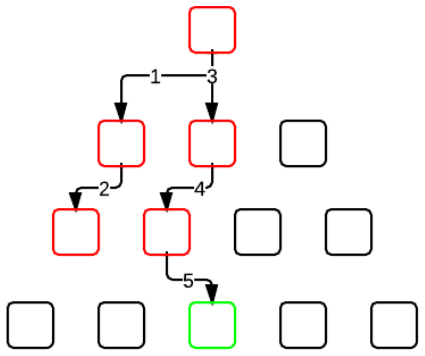
Figure 2: DFS - Searching successors before top level states



A* Search

Much like BFS, A* Search traverses the higher level states without plunging too deep into the state space unless necessary. A* prioritizes which states it should explore next via the use of Path Cost and Heuristics. To make this work in my application, I used a PriorityQueue, where each state is compared against those in the Queue (based on their overall cost) and put into the queue depending on their priority, in order to find the most effective path from the start State to the Goal State.

Figure 3: A* Search - Prioritizing states most likely to reach the goal state

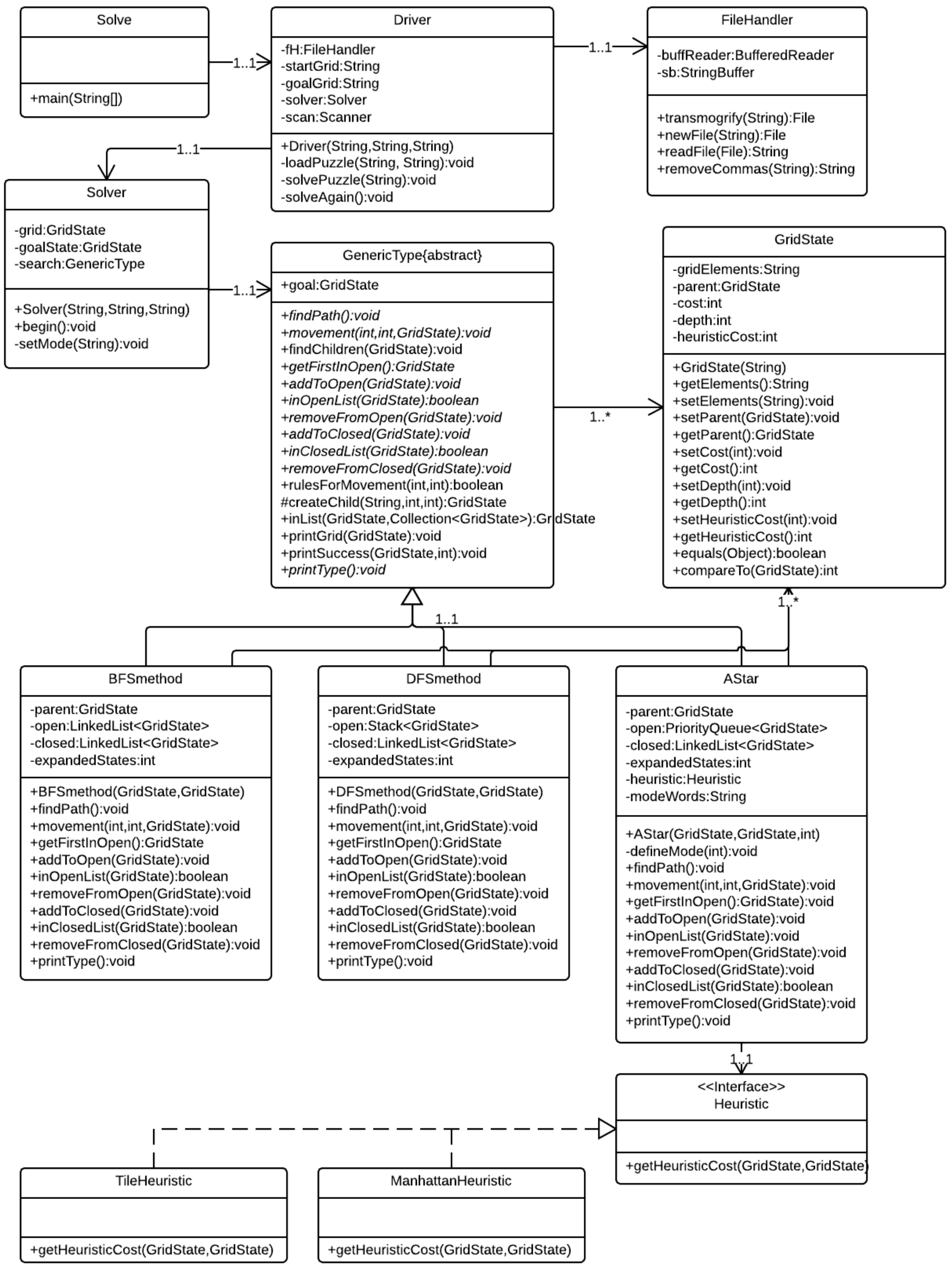


Design & Data Structure

UML Diagram

This UML diagram shows the classes and methods used within my application. Below is a brief description of each class, how it fits into the application and some significant methods. For more in-depth information on each particular method, consult the JavaDoc comments within the source code.

Figure 4: 8 Puzzle UML Class Diagram


Solve
+main(String[])
1..1
Driver
-fH:FileHandler
-startGrid:String
-goalGrid:String
-solver:Solver
-scan:Scanner
+Driver(String,String,String)
-loadPuzzle(String, String):void
-solvePuzzle(String):void
-solveAgain():void
1..1
FileHandler
-buffReader:BufferedReader
-sb:StringBuffer
+transmogrify(String):File
+newFile(String):File
+readFile(File):String
+removeCommas(String):String
1..1
Solver
-grid:GridState
-goalState:GridState
-search:GenericType
+Solver(String,String,String)
+begin():void
-setMode(String):void
1..1
GenericType{abstract}
+goal:GridState
+findPath():void
+movement(int,int,GridState):void
+findChildren(GridState):void
+getFirstInOpen():GridState
+addToOpen(GridState):void
+inOpenList(GridState):boolean
+removeFromOpen(GridState):void
+addToClosed(GridState):void
+inClosedList(GridState):boolean
+removeFromClosed(GridState):void
+rulesForMovement(int,int):boolean
#createChild(String,int,int):GridState
+inList(GridState,Collection<GridState>):GridState
+printGrid(GridState):void
+printSuccess(GridState,int):void
+printType():void
1..*
GridState
-gridElements:String
-parent:GridState
-cost:int
-depth:int
-heuristicCost:int
+GridState(String)
+getElements():String
+setElements(String):void
+setParent(GridState):void
+getParent():GridState
+setCost(int):void
+getCost():int
+setDepth(int):void
+getDepth():int
+setHeuristicCost(int):void
+getHeuristicCost():int
+equals(Object):boolean
+compareTo(GridState):int
1..1
1..*
BFSmethod
-parent:GridState
-open:LinkedList<GridState>
-closed:LinkedList<GridState>
-expandedStates:int
+BFSmethod(GridState,GridState)
+findPath():void
+movement(int,int,GridState):void
+getFirstInOpen():GridState
+addToOpen(GridState):void
+inOpenList(GridState):boolean
+removeFromOpen(GridState):void
+addToClosed(GridState):void
+inClosedList(GridState):boolean
+removeFromClosed(GridState):void
+printType():void
DFSmethod
-parent:GridState
-open:Stack<GridState>
-closed:LinkedList<GridState>
-expandedStates:int
+DFSmethod(GridState,GridState)
+findPath():void
+movement(int,int,GridState):void
+getFirstInOpen():GridState
+addToOpen(GridState):void
+inOpenList(GridState):boolean
+removeFromOpen(GridState):void
+addToClosed(GridState):void
+inClosedList(GridState):boolean
+removeFromClosed(GridState):void
+printType():void
AStar
-parent:GridState
-open:PriorityQueue<GridState>
-closed:LinkedList<GridState>
-expandedStates:int
-heuristic:Heuristic
-modeWords:String
+AStar(GridState,GridState,int)
-defineMode(int):void
+findPath():void
+movement(int,int,GridState):void
+getFirstInOpen():GridState):void
+addToOpen(GridState):void
+inOpenList(GridState):void
+removeFromOpen(GridState):void
+addToClosed(GridState):void
+inClosedList(GridState):boolean
+removeFromClosed(GridState):void
+printType():void
1..1
<<Interface>>
Heuristic
+getHeuristicCost(GridState,GridState)
TileHeuristic
+getHeuristicCost(GridState,GridState)
ManhattanHeuristic
+getHeuristicCost(GridState,GridState)

Class Descriptions

In writing my application, I found inspiration for the design of the search methods from the worksheets given for the module. These were based on the code from Coke and Code - Path Finding [1]. While there are some similar methods, the implementation is of my own writing and specific to my application.
For writing the A* algorithm, I did some research on how to implement A* and found some useful pseudo code which can be found at justinhj [2]. My resulting code is not the same as the pseudo code, and does not specifically match how my application works. The pseudo code was used as merely a base to help my understanding of how to best implement an A* search algorithm to solve the problem presented.

Solve

The main method for the application, taking in arguments from the command line and making a new Driver to begin the solver using them.

Driver

Receives the command line input and sends the Strings to the FileHandler, receiving back Strings to represent the Tiles of the Start state and the Goal state of the Puzzle. These are passed to the Solver class as GridStates. After a puzzle is solved, the user has a choice of running the application again with different inputs. This class also keeps track of how many milliseconds the application takes to solve each puzzle.

FileHandler

Takes a .txt file and extracts the contents as a String.

Solver

The link between the user input of the puzzle and the different types of Search methodst. Sets the mode and starts the puzzle solving.

GridState

A single state of the puzzle, keeping reference to the position of the elements/tiles in the Grid, it's depth, any costs and has methods for comparing it against other GridStates for use in the Priority Queue and discovering if two states are the same.

GenericType

An Abstract class, holding shared methods and holds prototypes to methods that must be implemented but might differ between each Search Type.
Notable Methods:

- `rulesForMovement(int zeroIndex, int direction)`

  Since I used a String to contain the state of the grid, the 'movement' of the empty Tile (0) around the grid is not as simple as it might be in a double for loop, where the boundaries of the grid are known by the boundaries of the arrays. The rules for movement set here state that if the 0 is on an 'edge' (0, 3, 6 / 2, 5, 8 / 0 - 3 / 6 - 8 indexes), the 0 cannot 'wrap' to the other side or go 'lower/'higher' than the bounds of the grid. This method checks that the 'direction' would not result in an illegal move.

- `createChild(String parentElements, int zeroIndex, int direction)`

  A new String must be made each time a new successor is required (child/state). In order to do this, the parent State is broken into chars, then the blank tile is swapped with the tile at the index to move to, and the tile at said index is swapped to the old empty tile slot. This array is then concatenated into a new String and made into a new GridState.

BFSmethod

Searches the Frontier by using the top level nodes first before expanding their children, continuing until the Goal evaluation finds the Goal State in the State Space. I implemented this using a LinkedList, although a Queue works just the same.

This method is complete and optimal, although may not find the best path and can be intensive on memory and time. If there is a solution to be found and a finitie State Space, then BFS will eventually find it. The method is not informed and can search without knowing anything except the start and the Goal State

This method avoids infinite loops by using an Explored list with the Frontier list. By keeping track of any States already in the Frontier and Explored list, the method can search through every single possible state in the State Space until it reaches a solution. Due to this, if the goal state is deep into the state space then time and memory become an issue. The more states it has to explore, the further into the tree it must get, taking more time, and the more states it must save to memory! This method is good for small State Spaces but slow and bad for memory on larger State Spaces.
Worst-case Time and Space Complexity: $O(b^d)$ (d is the shallowest step of the solution)

Notable Methods:

- `findPath()`

```
ADD Start State to Frontier
WHILE there are States to Explore
   GET head of Frontier
   GOAL Evaluation - If Found - Halt!
REMOVE expanded State from Frontier
Add expanded state to Explored list
FIND Children of explored State and add to Frontier
```

  The core pathfinding algorithm, searching the Frontier by looking a what is at the head of the collection, as set by the DataStructure (Queue/List).

- `movement(int zeroIndex, int direction, GridState current)`

  Creates new legal States (children) of the last explored state and adds them to the Frontier list if they have not already been explored or added to the Frontier.

DFSmethod

Searches the Frontier by using the deepest found States before any neighbours. DFS uses a Stack to implement this, where the last added State will always be the deepest found State and so the first to be explored.

DFS is not complete, nor is it optimal or will necessarily find the best path. For problems will a small State Space but deep answer, it is a good method to use as is can quickly plunge the depths of a state space to find the answer. With problems that have a large state space (such as this 8 puzzle), it should generally be avoided.
DFS can also get caught in an infinite loop should it find states that connect back to one another, getting stuck down a particular path of depth and repeating itself. This is due to DFS only keeping track of States on its current path, not all Explored States. However, by not saving all explored States, this does make DFS

better than BFS on memory.
Worst-case Space Complexity: $O(bm)$ (m = maximum depth of the goal)
Worst-case Time Complexity: $O(b^m)$ (BAD! If m is much larger than depth)

The Movement and Find Path methods in this class are the same as in BFS, only the data structure is different.

A* Search

A* functions like BFS, except it attempts to find the lowest cost path from the start to the goal State by using an 'informed' heuristic of what is 'closer' to the goal in the Frontier than what is not. For my implementation, I use two Heuristics: the Manhattan Distance and the Hamming distance (how many Tiles are not in the correct position). These will be covered in their own classes.
Similarly to BFS, A* Search uses a List, or a Queue, with the addition of prioritizing the ordering of the States based on their cost values, attempting to put the States which are most likely to lead to the goal at the head of the Frontier. With this in mind, I used a PriorityQueue to hold the Frontier, and have each GridState comparable based on their total cost.

A* is complete and optimal, assuming the heuristic doesn't overestimate the actual cost of the goal through the state space. The priorities of the States are weighted by $f = h + g$, where F is the function, H is the heuristic cost and G is the path cost. There are an infinite amount of different heuristics that can be used, where each has it's own admissibility. The Path cost isthe cost of how many states have been explored to reach the current state: $g(n)$.

The effectiveness of A* really depends on the heuristic provided (admissibility). Too weak of a heuristic and the path might not be the best, too strong of a heuristic and the actual goal can be missed and take longer to find. Issues can also arise with space, as A* keeps all States, in memory, just like BFS.
Worst-case Space Complexity: $O(b^d)$ or worse, depending on state space size.
Worst-case Time Complexity (weak heuristic): $O(b^d)$ - Same as BFS
Best-case Time Complexity (good heuristic): $O(d)$

Noteable Methods:

- `findpath()`
  ```
  SET starting PATH COST to 0
  SET starting DEPTH to 0
  ADD start state to Frontier
  WHILE there are states in Frontier
     GET head of Frontier
     GOAL Evaluation - If Found - Halt!
  REMOVE current State from Frontier
  ADD current State to Explored
  FIND successors of current State and ADD to Frontier
  ```
  Similar to the BFS method.

- `movement(int zeroIndex, int direction, GridState current)`
  ```
  MAKE new Successor to current State
  SET Path Cost of Successor to that of current State +1
  SET Parent of Successor to current State
  CALCULATE Heuristic Cost of Successor
  ```

```
    IF Successor is NOT in the Frontier or Explored lists
       ADD Successor to Frontier (PriorityQueue)
    ELSE
       Successor MUST be in one/both lists, CHECK which
       IF Successor is in Frontier AND has a lower Cost Now
          REMOVE old State from Frontier
          ADD Successor to Frontier
             ELSE IF Successor is in Explored and has a lower Cost Now
                REMOVE old State from Explored
                ADD Successor to Explored
```

Through this method, we can be sure that any Successor State added to the Frontier will always be the lowest costing version of that State and that any State added to the Frontier will always be placed in order of lowest cost at the head.

Heuristic

An Interface, declaring a prototype method that every other Heuristic class must have: getHeuristicCost. Using an interface, I can use the same A* Search class and 'plug-in' any different type of Heuristic as long as it has this method.

Mahattan Heuristic

The Manhattan Heuristic (Taxicab), is a well known and documented function. It works on the principle that to get from one state to another, there are a certain amount of 'moves' (streets, tiles, etc) one must make and so is informed of how the puzzle must be solved and how tiles 'move' between States. The Manhattan Distance Heuristic is the sum total of all these moves from one State to the next. Generally, the lower the amount of moves (cost) of the next move, the better. The Manhattan Distance is represented by $h(n)$, where n is the amount of overall moves needed for each tile to be in place for the Goal State.

To find the Manhattan Distance, I resorted to putting the data into a 2D array, to represent the grid as it would be (3x3) physically. Doing this meant that I could calculate the difference between each Tile's current location to the goal location in each State using the X and Y co-ordinates, though losing some performance in time due to the double for loop needed to iterate the array.

```
getHeuristicCost(GridState child, GridState goal)
CREATE 2D Array representations of the State to be evaluated and Goal State
FOR every Tile in the State
   FOR every Tile in the Goal State
      COMPARE the current Tile State to the Goal State Tile
      IF they match
         CALCULATE the amount of 'moves' it would take to
          get from the current State index to the Goal State index
RETURN Heuristic Cost of the sum of every State vs the Goal State Tiles
```

The Manhattan Distance is a good heuristic to use in this 8 puzzle that it neither over nor underestimates the 'distance' from the goal and is admissible. It attempts to find the approximate distance a State is from the Goal State, giving an indication of which State is closer to the Goal state than others. The returned value in this case has enough variance between those States closer to the Goal than those further away, where a Tile can be anywhere from 0 to 4 moves away from it's goal position.

Tile/Hamming Heuristic

The Tile Heuristic (Hamming) is counting the amount of Tiles that are 'out of place' in the State being evaluated against where they should be in the Goal State. This allows the Heuristic to determine if the State is close to the Goal by if it 'looks' close to the end of the path to the Goal.

While not a terrible heuristic and also admissible, it is not as reliable as the Manhattan Distance. With a potentially complex Puzzle such as this, a number of tiles can be in the correct location, but if one tile is out of place, it may be in such a way that in order to get it to the Goal would change all other Tiles into the 'wrong' locations before the solution. This makes finding the path to the solution a little longer than the Manhattan Distance. It can be represented as $h(n)$, where N is the number of tiles out of place.

Testing

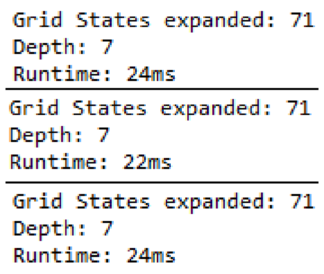
My Approach

The first thing I did when writing my application was test the GridState. I used JUnit and wrote the tests before the methods in the class itself. This ensured that every method I wrote would be correct, and if I changed methods, they would still be correct if the tests stayed successful. In many of my tests I cross-tested, where one test would not only test the GridState, but other tests. I wanted to make sure that my tests were correct before the methods were.

I continued this method of testing for the rest of my classes that dealt with data manipulation, to be sure that my methods were working correctly as a single unit and individually. Once my application was written and I knew that the methods worked correctly, I moved on to black box testing, running the application with different inputs and observing the outputs. I paid particular attention to the amount of states expanded, the run time of each method and which Search was the best, in terms of speed and accuracy.

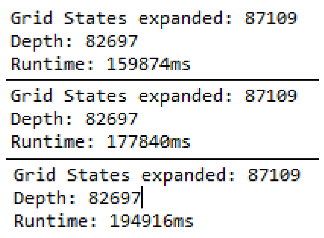
Test Cases

To test the application as a whole in solving a given puzzle, I used the three test cases provided and a number of test cases of my own. My aim was to test the theory of each type of Search, their complexities and the admissibility of my heuristics. To demonstrate this here, I will be displaying 'testStart3.txt', using each Search type 3 times to compare to one another.



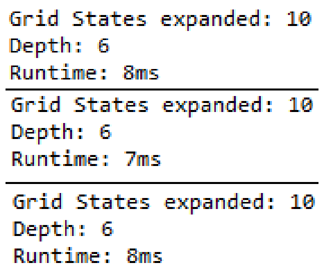
```
Grid States expanded: 71
Depth: 7
Runtime: 24ms
Grid States expanded: 71
Depth: 7
Runtime: 22ms
Grid States expanded: 71
Depth: 7
Runtime: 24ms
```

(a) BFS results



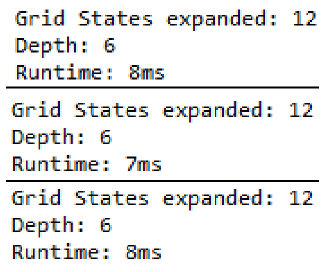
```
Grid States expanded: 87109
Depth: 82697
Runtime: 159874ms
Grid States expanded: 87109
Depth: 82697
Runtime: 177840ms
Grid States expanded: 87109
Depth: 82697
Runtime: 194916ms
```

(b) DFS results



```
Grid States expanded: 10
Depth: 6
Runtime: 8ms
Grid States expanded: 10
Depth: 6
Runtime: 7ms
Grid States expanded: 10
Depth: 6
Runtime: 8ms
```

(c) A* results using Manhattan heuristic



```
Grid States expanded: 12
Depth: 6
Runtime: 8ms
Grid States expanded: 12
Depth: 6
Runtime: 7ms
Grid States expanded: 12
Depth: 6
Runtime: 8ms
```

(d) A* results using Hamming heuristic

Figure 5: Tests carried out on each Search method, using testStart3.txt

As can be seen from the results, BFS finds the solution relatively easily (71 expansions), while DFS takes much more time and must expand more states (many more!), indicating that the solution is not too deep into the state space, while being quite far through 'neighbouring' states and so not a good technique here. We can see that A* finds a solution effectively and through a small amount of steps.
When comparing heuristics, the Manhattan Distance (10 expansions) is better than Hamming (12 expansions). As stated previously, the Manhattan distances gives a very good approximation of how close a state is to being correct by calculating the amount of 'moves' needing to be taken, better than just by how the grid 'looks' like with Hamming. Both do very well though, are complete and have a very short runtime, proving they are admissible, complete and optimal.

Conclusion

To conclude, from the test data it shows A* is best algorithm of the three to use when solving an 8 puzzle (although when compared to other methods, such as Greedy Best First Search, may not be the fastest for some puzzles). As we know what the puzzle looks like, how the tiles move, what the goal state should look like and can keep track of the puzzle states, A* is the best search method, especially using the Manhattan Distance. When it comes between DFS and BFS, it is dependant on the goal. BFS will always find a result, and often quite quickly for smaller puzzles. However, if the goal is deep into the tree and close to the state State, DFS should be used.

If given more time, I would provide the user more options, try out different configurations of puzzles and even attempt to expand on the 8 Puzzle and create a 15 Puzzle solver to really put my search algorithms to the test. On top of this, in order to help with testing and inform the user of more, I would create a GUI displaying the solving process as it happens, any children created and give the user a step-by-step walk through of the solving method after the solution has been found.

References

[1] Path Finding — Coke And Code, *Coke And Code*,[online], http://www.cokeandcode.com/main/tutorials/path-finding/ (Accessed: 15/11/2013)

[2] justinhj page, *justinhj*,[online], http://heyes-jones.com/pseudocode.php (Accessed:15/11/2013)