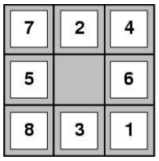

CS26110 - Artificial Intelligence Assignment

8-Puzzle

Author: Laura Collins (lac32)

Date last edited: November 19, 2013

Word Count: 2633

Contents

1 Section 1

1.1 Analysis

1.1.1 The Problem

The Problem presented in this Assignment is to make a Java Program that can solve 8-Puzzles using three different AI techniques, Breadth-First Search, Depth-First Search and A* Search. 3 sets of Testing Data files have been supplied and the program should be tested using these, and any extra files I may wish to run myself.

1.1.2 How I approached this problem

My first step after reading the Assignment Specification was to research into Queues and Stacks as they're not something I've had to implement before. Another one of my Modules, CS21120 covers both of these, so I thought that would be a good place to start, and on top of that some research online, especially on the Java 7 API. Next step I started implementing a GUI for the 8-puzzle, but upon reading the Specification again realised this was unnecessary, and stopped.

Each number is read in as an integer, as 0 is used instead of a blank space, meaning reading in was quick and simple. When loaded in, the integers are added into an array, the StartState, and the Solution (or Goal State) was also read in in the same way.

Once the Start State and Goal State have been read in, they can be used to implement the three different methods of search required to be used in this assignment as if they are equal, the puzzle is therefore solved.

Each AI Search Technique (BFS, DFS, Manhattan Distance & Hamming Distance) has a Class of it's own, which, upon reflection, uses a lot of the same methods to run the Searches, so could be refined further.

1.2 Design

1.2.1 Class Diagram

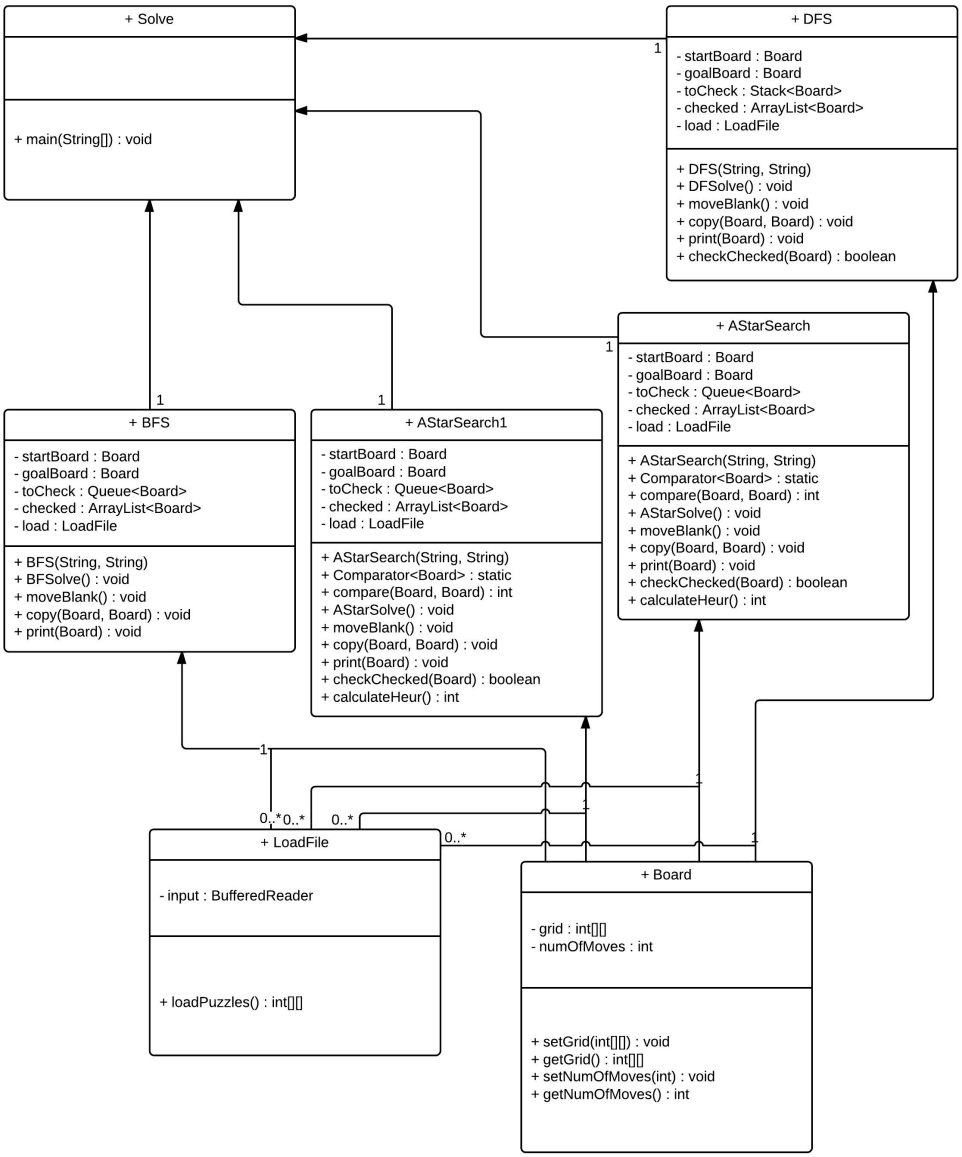

Figure 1: Version 2.1 Final

1.2.2 Class Descriptions

Solve
Solve is a simple Class running the entire Project by instantiating a new instance of whichever AI Technique has been specified by the user. This is the Class to be run when the user is using the Command Line/Terminal.

Board
The Board Class compromises mainly of methods to set the Values in the grid and return these said values afterwards. This is basically the content of the puzzle, setting out the layout, dealing with the Setting and Getting of the Heuristic Integer when using A*Search, and also for calculating the amount of moves needed to get from the Start State to the Goal State.

BFS
This Class is where the Breadth-First Search is run. This works by loading in the files at the start of the Class, and while the first-most node in the 'toCheck' queue is not equal to the Goal State, the Blank Space is moved around (up, down, left, right) with each node being added to the 'toCheck' queue and then being evaluated and added to the 'checked' ArrayList if still not equal to the goal. If no solution is found, the message 'No Solution Found' is displayed.

Breadth-first works by going along a height in a Tree, exploring each of these nodes first, before expanding the next height of nodes down and exploring them etc. This is an Complete Search technique, as if the solution is in the Tree, it *shall* be found. It has a Space Requirement and Time Complexity of $O(b^d)$ - *Where b is the maximum number of successors of any node and d is the depth of the shallowest goal.*

DFS
This Class is where the Depth-First Search is run. This works similarly to BFS, except instead of using Queues, it uses a Stack to store the nodes, so when it's being explored, it's done in a different way, by opening all Child nodes before opening other Parent nodes. If no solution is found, the message 'No Solution Found' is displayed.

Depth-First works by opening the first Parent node, and exploring all of it's children before moving along to the next Parent node on the same height. However DFS has some issues, it can get stuck down one path, never finding the solution, therefore is not Complete and if the Tree is extremely deep, it can take a very long time to actually get any solution. It has a Space Requirement of $O(bm)$ and a Time Complexity of $O(b^m)$ - *Where b is the maximum number of successors of any node and m is the maximum length of any path in the State Space.*

AStarSearch
This Class is where it calculates the Hamming Distance, which is how many tiles are out of place to help solve the 8-puzzle and runs the Search. This uses a PriorityQueue to store the nodes, and compares the current node with the Goal State with the aim for getting the difference to be smaller upon each step.

A*Search is an informed Search Method which will be covered more later on in the document.

AStarSearch1

This Class is where the Manhattan Distance heuristic of A* Search is implemented. Again, this uses the priority queue, but instead it calculates the distance each tile is from it's goal place.

A*Search is an informed Search Method which will be covered more later on in the document.

LoadFile

The LoadFile class sets up the structure for loading in the .txt files for Start State and Goal State. It does this by setting up a temporary value to take in the values from a file, and then returns these values, so when called, values can be added to a specific Array. This way the same load function can be used for both Start State and Goal State.

1.2.3 Data Structure

2D Array
I used 2 2D Arrays to store the Board values of 0-8 for the Start State and the Goal State because they're a fixed size, so there's no worry about changing the length of the array. It can also be used to represent more than one data item of the same type, using the same name, so the Start State and the Goal State can be called in using the Tiles[][] array.

ArrayList
An ArrayList was used to store the Expolored nodes, as ArrayLists don't need a fixed initial size, and there is no way of predicting how many nodes will need to be explored until the solution is found. The only downside to this is sometimes it can be slow with large amounts of data, which may happen, especially using DFS.

Queue
A queue is needed to implement Breadth-First Search. This is the best method as it enforces the *First In, First Out* order, so we process the parent nodes before any of the child nodes. A Queue is easy to implement, with simple operations such as '.add' and '.remove', and also has peek() and poll() to return data from the Collection.

Stack
A Stack is used to implement Depth-First Search. This uses the *Last In,First Out* order, so all the child nodes created are explored before exploring the rest of the parent nodes, as needed in Depth-First Search.

Priority Queue
A Priority Queue is implemented for use within the A*Searches. This takes the one of the highest priority out of the Queue first, so in this case, it's the one with the lowest heuristic, therefore the one closest to the answer.

1.3 Algorithms

1.3.1 Breadth-First Search

```
while current node != goal state do

  Add Start State to checked ArrayList
  for int i:=0 to 3 do

    for int j:=0 to 3 do

      if Tile in Board = 0 then

        if i != 0 then
          Move 0 Up
          Add node to toCheck
        end if
        if i != 2 then
          Move 0 down
          Add nodes to toCheck
        end if
        if j != 0 then
          Move 0 left
          Add node to toCheck
        end if
        if j != 2 then
          Move 0 right
          Add node to toCheck
        end if
      end if
    end for
  end for
  Increase node count
  print  nodes
  if Queue is empty and Explored nodes >0 then

    print  No Solution Found
  end if
  print  Start State
  print  Number of Nodes explored
  print  Number of tile moves
  print  Last Node
  print  Goal State
end while
```

1.3.2 Depth-First Search

```
while current node != goal state do

  Add Start State to checked ArrayList
  for int i:=0 to 3 do

    for int j:=0 to 3 do

      if Tile in Board = 0 then

        if i != 0 then
          Move 0 Up
          if Node has not been explored then
            Add node to toCheck
          end if
        end if
        if i != 2 then
          Move 0 down
          if Node has not been explored then
            Add node to toCheck
          end if
        end if
        if j != 0 then
          Move 0 left
          if Node has not been explored then
            Add node to toCheck
          end if
        end if
        if j != 2 then
          Move 0 right
          if Node has not been explored then
            Add node to toCheck
          end if
        end if
      end if
    end for
  end for
  Increase node count
  print  nodes
  if Queue is empty and Explored nodes >0 then

    print  No Solution Found
  end if
```

```
    print Start State
    print Number of Nodes explored
    print Number of tile moves
    print Last Node
    print Goal State
  end while
```

1.3.3 A* Search - Hamming & Manhattan Distance

```
  while current node != goal state do

    Add Start State to checked ArrayList
    for int i:=0 to 3 do

      for int j:=0 to 3 do

        if Tile in Board = 0 then

          if i != 0 then
            Move 0 Up
            if Node has not been explored then
              Set Heuristic Number
              Add node to toCheck
            end if
          end if
          if i != 2 then
            Move 0 down
            if Node has not been explored then
              Set Heuristic Number
              Add node to toCheck
            end if
          end if
          if j != 0 then
            Move 0 left
            if Node has not been explored then
              Set Heuristic Number
              Add node to toCheck
            end if
          end if
          if j != 2 then
            Move 0 right
            if Node has not been explored then
              Set Heuristic Number
              Add node to toCheck
```

```
            end if
          end if
        end if
      end for
    end for
    Increase node count
    print nodes
    if Queue is empty and Explored nodes >0 then

      print No Solution Found
    end if
    print Start State
    print Number of Nodes explored
    print Number of tile moves
    print Last Node
    print Goal State
  end while
```

1.3.4 A* Search - Hamming Distance Calculating Heuristic

```
  for int i:=0 to 3 do

    for int j:=0 to 3 do

      if toCheck != 0 and the top value of toCheck = the value of goalBoard then
        Increment Heuristic Number
      end if
    end for
  end for
  return Heuristic Number
```

1.3.5 A* Search - Manhattan Distance Calculating Heuristic

```
for int i:=0 to 3 do

  for int j:=0 to 3 do

    for int x:=0 to 3 do

      for int y:=0 to 3 do

        if toCheck != 0 then
          integer distance = difference between x & i and difference between y & j
          in currentNode and goalBoard
        end if
      end for
      Integer total = Heuristic Number + Distance
    end for
  end for
end for
return Total
```

1.4 A* Heuristics

A* Heuristics are algorithms used for solving problems in an Informed way. It uses 'Best First Search' to consider the lowest cost path first, as this is most likely to lead to the Solution it believes.

1.4.1 A*Search 1 - The Hamming Distance

This A* Search heuristic was first presented to us in Lectures, although not named The Hamming Distance at the time. The Hamming Distance is the number of Tiles not in the correct place, so a comparison between the current node being checked and the Goal State needs to be made.

Admissibility
Hamming Distance is entirely admissible as the total number of moves is at least the total number of misplaced tiles at the beginning.

Informedness
For testStart2.txt & testGoal2.txt:

```
Nodes explored: 34
********************
Number of moves to Goal State: 4
********************
```

1.4.2 A*Search 2 - Manhattan Distance

This A* Search heuristic was also presented to us in our Lecture, and named as one we should use for solving the 8-Puzzle. This is calculated by the actual distance between where the Tile is in the node, to where it should be in the Goal State using it's Cartesian Coordinates.

Admissibility
Manhatten Distance is admissible also as every tile will have to be moved at least every movement between itself and it's desired destination.

Informedness
For testStart2.txt & testGoal2.txt:

```
Nodes explored: 34
********************
Number of moves to Goal State: 4
********************
```

2 Section 2

2.1 Testing

2.1.1 Breadth-First Search

```
TestStart1.txt testGoal1.txt
Breadth-First Search
1
2
3
4
5
6
7
******************
Start State:
120
345
678
******************
Nodes explored: 7
******************
Number of moves to Goal State: 2
******************
Last node:
012
345
678
******************
Goal node:
012
345
678
******************
```

```
TestStart2.txt testGoal2.txt
118
119
120
121
122
123
124
125
******************
Start State:
125
304
678
******************
Nodes explored: 125
******************
Number of moves to Goal State: 4
******************
Last node:
012
345
678
******************
Goal node:
012
345
678
******************
```

```
TestStart3.txt testGoal3.txt
422
423
424
425
426
427
428
429
******************
Start State:
320
615
748
******************
Nodes explored: 429
******************
Number of moves to Goal State: 6
******************
Last node:
012
345
678
******************
Goal node:
012
345
678
```

2.1.2 Depth-First Search

```
TestStart1.txt testGoal2.txt
Depth-First Search
1
2
******************
Start State:
120
345
678
******************
Nodes explored: 2
******************
Number of moves to Goal State: 2
******************
Last node:
012
345
678
******************
Goal node:
012
345
678
******************
```

```
TestStart2.txt testGoal2.txt
120852
120853
120854
120855
120856
120857
120858
120859
******************
Start State:
125
304
678
******************
Nodes explored: 120859
******************
Number of moves to Goal State: 109186
******************
Last node:
012
345
678
******************
Goal node:
012
345
678
******************
```

```
TestStart3.txt testGoal3.txt
49209
49210
49211
49212
49213
49214
49215
49216
******************
Start State:
320
615
748
******************
Nodes explored: 49216
******************
Number of moves to Goal State: 47668
******************
Last node:
012
345
678
******************
Goal node:
012
345
678
```

2.1.3 A*Search 1 - Tiles In Place

```
TestStart1.txt testGoal2.txt
Tiles out of Place
Iterations:
1
2
3
4
5
******************
Start State:
120
345
678
******************
Nodes explored: 5
******************
Number of moves to Goal State: 2
******************
Last node:
012
345
678
******************
Goal node:
012
345
678
******************
```

```
TestStart2.txt testGoal2.txt
27
28
29
30
31
32
33
34
******************
Start State:
125
304
678
******************
Nodes explored: 34
******************
Number of moves to Goal State: 4
******************
Last node:
012
345
678
******************
Goal node:
012
345
678
******************
```

```
TestStart3.txt testGoal3.txt
5
6
7
8
9
10
11
12
******************
Start State:
320
615
748
******************
Nodes explored: 12
******************
Number of moves to Goal State: 6
******************
Last node:
012
345
678
******************
Goal node:
012
345
678
```

2.1.4 A*Search 2 - Manhattan Distance

```
TestStart1.txt testGoal2.txt
Manhattan Distance
Iterations:
1
2
3
4
5
******************
Start State:
120
345
678
******************
Nodes explored: 5
******************
Number of moves to Goal State: 2
******************
Last node:
012
345
678
******************
Goal node:
012
345
678
******************
```

```
TestStart2.txt testGoal2.txt
27
28
29
30
31
32
33
34
******************
Start State:
125
304
678
******************
Nodes explored: 34
******************
Number of moves to Goal State: 4
******************
Last node:
012
345
678
******************
Goal node:
012
345
678
******************
```

```
TestStart3.txt testGoal3.txt
5
6
7
8
9
10
11
12
******************
Start State:
320
615
748
******************
Nodes explored: 12
******************
Number of moves to Goal State: 6
******************
Last node:
012
345
678
******************
Goal node:
012
345
678
```

2.2 Analysis of the Results

2.2.1 Breadth-First Search

BFS got to the Solution of each of the puzzles relatively quickly as the solution was not too deep within the Queue, especially in the first Test Cases. 429 iterations, on the third Test Case isn't much for the Program, and it's run almost instantly to the human-eye. The downside to BFS is it's Space Complexity as all nodes must be stored, as the Program needs to go back to them later on to explore the child nodes. However it is Optimal and Complete, as it always finds the answer (given the amount of data is finite) and it's Time Complexity of $O(b^d)$ is good, where d is the depth of the Shallowest Goal.

2.2.2 Depth-First Search

DFS ran slower than BFS as for the second and third Test Case it had to go through more iterations to get to the answer, therefore showing the solution was down a branch further to the right. 120,859 iterations is run in about 4-5mins on my Computer (Mac), so it's slower than the BFS, but it does far more iterations. The Space Complexity for DFS is much better than BFS, as it only needs to store the current branch it's working on at a time, also, if the solution is deep, it may be a lot quicker to the solution than BFS. However, it's time complexity of $O(b^m)$, where m is the maximum length of path, may have downfalls as if m is much greater than 'd' in BFS's Time Complexity, it takes a lot longer to get to the solution.

2.2.3 A*Search

A* Search is both Complete and Optimal if it does not overestimate the true cost of the path to the solution. It's Time Complexity is good, in worst case scenario it's the same as BFS, so for these Test Cases, this is not bad as the solutions aren't too deep. The Space Complexity keeps all nodes in it's memory, in case of repetition, so it can occur that it runs out of Memory before it runs out of time. I did find when testing with special Test Cases I had created myself, that if a solution cannot be found within a certain amount of iterations, then it runs out of memory.

2.3 Appropriateness of the Search Methods

2.3.1 Breadth-First Search

For this Project, BFS was suitable, as the State Space wasn't too large, so it finds the solution quickly, although taking up a fair bit of Memory, storing all the nodes as it goes. The fact it is Complete and Optimal is a bonus, as we know we'd get the answer eventually, and the best answer at that.

2.3.2 Depth-First Search

For this Project, DFS wasn't so good. It did well on Test Case 1, but this was purely coincidence that it happened to be just down the left-most branch, but on the other Test Cases it is significantly slower than the other searches, and needed to check a lot more nodes, even though the solution wasn't too deep. This shows the Time Complexity has let it down, even though it's Space Complexity may be better than BFS.

2.3.3 A* Search

I feel both heuristics I chose for this Project were the Optimal ones for the 8-Puzzle. Their Informedness combined with Time Complexity means the solutions are found extremely quickly, although Space Complexity does let it down slightly, however if the solution is found quickly (as it usually is) this shouldn't be an issue.

2.3.4 A*Search 1 - Hamming Distance

For this Project, Hamming Distance is extremely useful, as it reaches the answer at an optimal rate, with the Time Complexity of $O(n)$ where n is the amount of Tiles out of Place. It got to the answer in the Optimal amount of moves as it's well-informed.

2.3.5 A*Search 2 - Manhattan Distance

For this Project, Manhattan Distance may have had a downfall in the fact that Tiles cannot go through each other, but must go around each other. With the same Time Complexity as Hamming Distance it's extremely quick to finding the answer.

2.4 Future Improvements

1. As a future improvement I would tweak my A*Searches, as they're currently running the same, which is incorrect. They both have the same nodes and amount of tiles moved.
2. Also I would test it on more Test Cases to ensure it worked sufficiently on more than just the 3 Test Cases.
3. I would ensure the 'No Solution Found' works always, as currently it's glitchy and doesn't always work.
4. I would refine my code to stop duplication, as currently the Move Blank method is very similar in all the classes.
5. Given more time I could implement a bigger/smaller board for different-sized puzzles.
6. Could implement a GUI for the 8-Puzzle.

3 References

3.1 BFS

- Date Accessed: 18/11/13
- URL: http://www.cs.bu.edu/teaching/c/tree/breadth-first/
- Used for: Research into Breadth-First's use of Queues

3.2 Admissible Heuristics

- Date Accessed: 18/11/13
- URL: http://en.wikipedia.org/wiki/Admissible_heuristic
- Used for: Research into the Admissibility of A* Searches

3.3 Hamming Distance

- Date Accessed: 19/11/13
- URL: http://www.ritambhara.in/hamming-distance/
- Used for: For researching Time Complexity of Hamming Distance