CSM6120 - 8 puzzle solver

Stefan Klaus, stk4@aber.ac.uk

***Abstract*—The abstract goes here.**

I. Introduction

IN this report the created solution to the 8 puzzle solver will be discussed.

II. Structure

The overall structure can be seen in the UML diagram, found in appendix A though C. In this section the structure of the program will be explained and the different classes and their uses. Detailed description methods will not be given please refer to the JavaDoc in appendix **ADD JAVADOC AS APPENDIX**.
The structure of this section is as follows: for every package used in the program the use of the package as well as the classes inside it is given followed by a section about the justification of the design choices.

A. The csm6120 Package

This package holds the Main class, aswell as the *State* and *FileManager classes*.
The class diagram for this can be found in Appendix A on page 5.
This package acts as entry point to the program where the input files are read and where the *State* class is located.

1) Main: The Main class is the entrance to the program.
The main method takes command line arguments such as the path to the start state and goal state of the puzzle(in .txt files) and a string which specifies what algorithm to use.
It creates instances of the FileManager object and State objects and calls the algorithm chosen by the user.

2) State: The State class holds variables and methods to manipulate and represent each state of the puzzle, that is the tile placement of the puzzle.
The tiles are interpreted as a simple arrayList, this has been done in order to make manipulating the array easier.
The class holds methods to switch 2 given tiles(done based on their index in the arrayList) as well as other common methods such as getters and methods to compare a state object to another.
These methods are used in the other algorithms incorporated in the program.

3) FileManager: The FileManager class holds methods to read input from a file, converts the strings into integers and then save it to the integer ArrayList in a State object.
The main class instantiates a single FileManager object and than uses it to read the input files provided by the user.

4) Justification: The idea for the *csm6120 Package* was to have the methods which manipulate the input files all in one place. It was considered to move the State class to the *SearchTree* package, but considering it holds such a central role in the program the decision fell against it.
The FileManager class was created in order to have a separate class to handle input files, this was done in order to follow object orientated design structures.

B. The SearchTree Package

This package holds classes which represent and generate the search tree.
The UML diagram for this can be found in appendix B on page 6

1) TreeNode: Objects of the TreeNode class represent nodes of the search tree.
Every TreeNode holds a State object as well as 2 linkedLists. One linkedList holding the children of one particular node, the other for holding its siblings.
The class has methods to add, get and remove nodes from those lists, as well as common methods such as returning if a list is empty.

2) Graph: The graph is used to generate the next step of the graph, by expanding all possible children from a particular TreeNode object and than add them to the linkedlist of children of said TreeNode object.
The nextStep method of this class checks where the empty tile is in a given TreeNode object, based on that it calls one of three possible functions which generate the children of the object.

3) Justification: The idea was to have all classes which represent or manipulate the nodes of the search tree in a single package. It was also decided to only have a class for the nodes of the tree and representing the connections to its children using a internal linkedList rather than having a separate edge class which only links 2 nodes together. The reason this was done was for simplicity: using linkedList methods like adding and returning makes it easier considering every node has a very limited, maximum of 4, number of possible children.

C. SearchAlgorithm Package

This package holds all search algorithms implemented in the program, as well as all supporting classes needed for them.
The UML diagram for this package can be seen in appendix C on page 7.

In general all the search algorithms are build up much the same way, they take 2 parameters: a start and goal State object, these States are than used to create the first node of the search tree, the goal state gets saved in the current search algorithm object and is used to check if the goal state of the puzzle has been reached.
The algorithms use the classes from the SearchTree package(see section II-B) to generate the search tree.
At the beginning of each algorithm it is checked if the start state is equal the goal state, this might be a rare occurrence but a useful functionality non the less.
All algorithms also hold a list of all previous expanded nodes, which all new generated nodes are checked against in order to prevent infinity loops. That already visited nodes are created again is because the children to each node are generated without knowledge if such a node has been created before, so infinity loops would be created relatively early on in the search trees.

1) Breadth-First search: This class holds the breadth-first search algorithm.
This algorithm works using a first-come first served queue.

2) Depth-First search: This class holds the depth-first search algorithm.
It is in its structure very similar to the breadth-first search algorithm, however uses a last-in first-out stack.

3) Greedy Best-First search: This class holds the greedy best-first search algorithm.
It is in its structure very like the breadth-first or depth-first algorithms, however uses a priority queue. In the priority queue the elements are sorted according to its *natural ordering*, meaning from lower to higher numbers.
That way the treeNode which is closer to the goal state will be checked next. The priority queue uses the stateComparator class to sort its elements.

4) A star algorithm: This class holds the A star search algorithm.
This algorithm is similar to the greedy best-first algorithm in that it uses a priority queue, but it also uses a more advanced heuristic to decide which nodes to add to the queue in the first place.
This class holds 2 methods, one is the search algorithm in itself, the other is a method used to decide which treeNode object to add to the search queue. This is done using the Manhattan Distance heuristic. In the beginning of the algorithm the Manhattan Distance to the goal state is calculated and saved to a variable.
The addNode method than uses the current treeNode object and the goal State to calculate the Manhattan distance for each of the current treeNode's children. If the calculated Manhattan Distance is equal or less than the one calculated at the beginning of the algorithm the node is added to the search queue. Should the calculated distance be larger the child is discarded.
To calculate the Manhattan Distance the a ManhattanDistance object is used.

5) StateComparator: This class holds the StateComparator method used to compare to states in a priority list. This class implements the *Comparator* interface.
It uses the String representation of a State objects state array to do the comparison.

6) ManhattanDistance: This class holds the methods used to calculate the Manhattan Distance.
During development of an early method to calculate the Manhattan Distance straight from a state array a number of problems have been found. The decision was made that a 2D representation of the array would be more fitting in order to calculate the Manhattan Distance.
However in order to avoid having to refactor all other algorithms the ManhattanDistance class holds methods to change an input arrayList to a simple array and after that to a 2 dimensional array.

The Manhattan Distance is calculated as such:
For each possible number of the puzzle find its index X and Y coordinates for each of the 2 input States. Then compare the X and Y coordinates of both indices and calculate the total difference in tiles.

Here 2 things must be nodded:
Depending on if the start State indices minus the goal State indices return a value less than zero or above, the formula gets flipped in order to prevent receiving a negative Manhattan Distance. The code for this is shown below:

```
if (startArray[i][j] == goalArray[i][j]) {
      continue;
} else if (startArray[i][j] - goalArray[i][j]
    < 0) {
      manhattanDistanceSum +=
          Math.abs(x_start - x_goal)
            + Math.abs(y_start - y_goal);
} else if (startArray[i][j] - goalArray[i][j]
    > 0) {
      manhattanDistanceSum += Math.abs(x_goal
          - x_start)
            + Math.abs(y_goal - y_start);
}
```

The other important thing to node is that empty/zero tile of the puzzle is ignored in the Manhattan Distance calculation, this is to accommodate the fact that at every iteration 2 tiles are switched.

7) Justification: It can be argued that the methods in the ManhattanDistance class are bad practice, as it would have been better to refactor all previously written algorithm to use 2D arrays rather than converting a State arrayList to a 2D array for every treeNode object.
However this decision was made as the use of a arrayList makes the use of the algorithms though out the program easier, thanks to Java's inbuilt arrayList methods.

Also since all arrayList have a limited size of 8, the computational cost for the conversion is constant and not large.

The list of expanded nodes which each class holds and all algorithms loop over at every iteration to check if the current nodes children already where created ones add a lot of computational overhead to each iteration.
While it gets apparent in breadth-first,however not as much in greedy best-first and A star search, this is not to much a problem because of the typical behaviour of these algorithms the overhead. But in depth-first search it causes massive computational overhead and causes long run times in order to find a solution.
The reason for that is the behaviour of the algorithm, unless in a few seldom instances the state space for depth-first search is much more massive than for the other algorithms.
So iterating over 10's of thousands of nodes in the list of expanded nodes takes a lot of time and computational power. Still the list needs to be maintained in order to avoid infinite loop problems.

III. Analysis

In this section the design choices will be analysed further. This includes the heuristic used as well as any problems encountered during the development cycle.

A. Manhattan Distance

For this program 2 heuristics where considered.
The first being simply the number of misplaced tiles, this heuristic is admissible as all misplaced tiles must be moved atleast once.
However it was decided to implement something more "advanced" in form of the Manhattan Distance heuristic, which is another common heuristic for these kind of problems [1].
The Manhattan Distance is admissible for this kind of problem as it either estimates the exact number of moves needed or underestimates the total path costs needed to find a solution.
The Manhattan Distance is calculated by the sum of the vertical and horizontal distances of between to tiles, because tiles can not move along diagonals.

B. Performance

In this section the performance of each algorithm with the 3 provided test cases is analysed.
This will be done algorithm for algorithm, with a large table showing all the results which can be found in appendix **ADD APPENDIX WITH RESULT TABLE**
It is interesting to node that the processing time varies between runs. While on the first run it is often a lot higher, the processing time reduces in the following couple of runs. until it either gets stable or varies between 2 values. All running times shown in this section are taken after 5 runs as it then appears to be stable.

1) Breadth-First search: Table I shows the performance of Breadth-First search.
As the test result show the number of explored nodes increases

TABLE I
Performance of Breadth-First Search

Test case	Path Cost	Expanded Nodes	Running Time in nanosecond
1	3	5	
2	97	7	
3	381	11	

significantly when more tiles are out of place. The processing time also increases.

2) Depth-First search: Table II shows the performance of Depth-First search.
As can be seen the path cost is massive, a lot more than the in any other search algorithm implemented in the system. As well is the processing time. The reason for the immense path cost and processing time is the way children are added to nodes in the program, in combination with the working of Depth-First search.
It is likely that if nodes where added in a different way the path cost and processing time decrease. It could also be improved with implementing an abbreviation of Depth-First search, such as Depth-Limited search.
It is worth noting that the arguably more complex systems of Test Case 2 and 3 actually have a lower path cost and lower computational time than the first one, where only 3 tiles are out of place.

TABLE II
Performance of Depth-First Search

Test case	Path Cost	Expanded Nodes in nanosecond	Running Time
1	169022		1303747
2	139063		1115838
3	111950		895815

3) Greedy Best-First search: Table III shows the performance of the Greedy Best-First search algorithm.
As the table shows, in the first and easiest test case the algorithm performs with perfect accuracy, choosing the direct path to the goal state.
Test case 2 and 3 as well show a large improvement in path cost over Breadth-First and Depth-First search. However it can be seen that the extra computational overhead of sorting the priority queue causes the algorithm to perform marginal slower than Breadth-First search. This small computational overhead is however made up by the far smaller number of explored search tree nodes.

4) A star search: Table IV shows the performance of the A^* Algorithm.
Different in this table compared to the previous once is that

TABLE III
PERFORMANCE OF GREEDY BEST-FIRST SEARCH

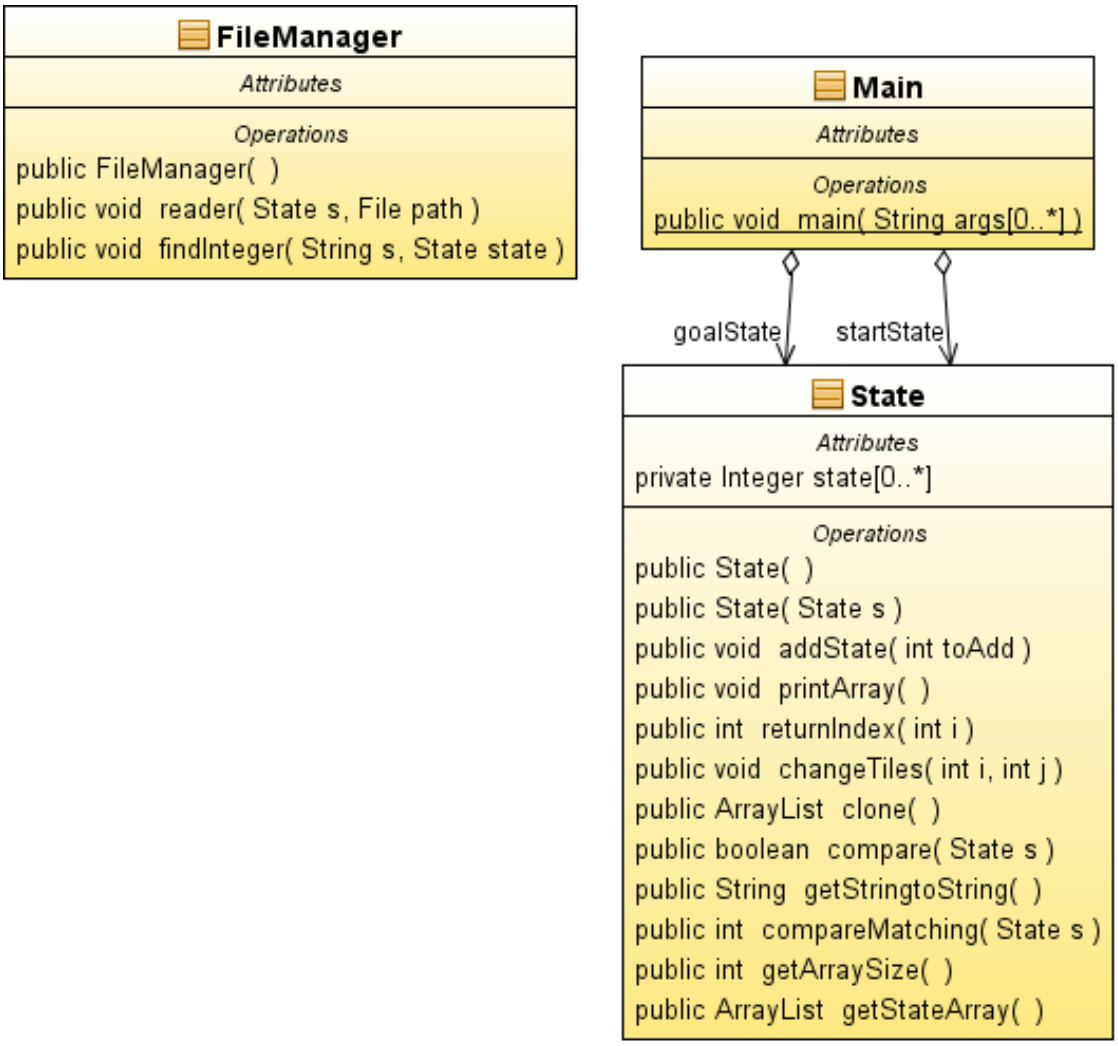
Test case	Path Cost	Expanded Nodes in nanosecond	Running Time
1	2		6
2	8		7
3	72		19

an extra column has been added in to show the Manhattan Distance value which was calculated at the beginning of the algorithm.
As the table shows for test cases 1 and 2 the estimated Manhattan Distance was reached precisely. On test case 3 the algorithm had to explore 21 nodes in order to find the goal state, while the estimated Manhattan Distance was 6 notes. This shows that the heuristic is admissible as it does not overestimate the path cost.
The processing time is similar to Greedy Best-First and Breadth-First search. Even for the most advanced puzzle(test case 3).

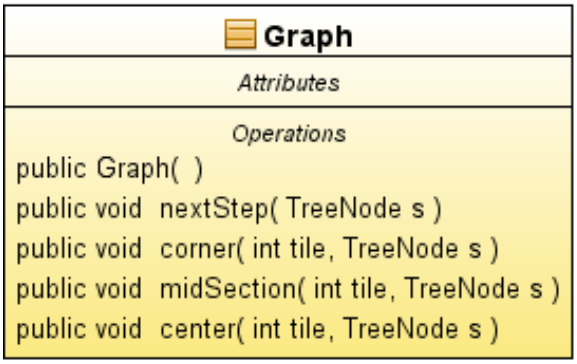
TABLE IV
PERFORMANCE OF A STAR SEARCH

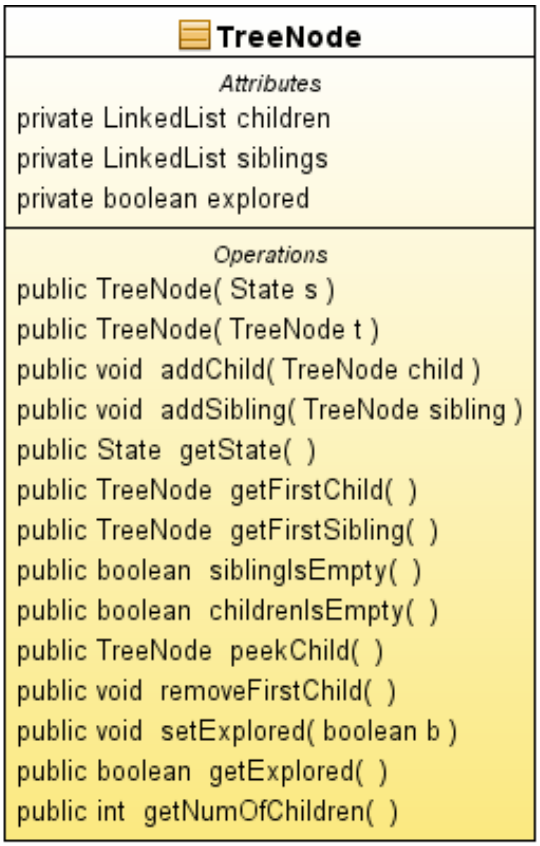
Test case	Manhattan Distance	Patch cost	Expanded nodes in nanosecond	Running Time
1	2	2		6
2	4	4		7
3	6	21		11

APPENDIX A
CSM1620 PACKAGE UML

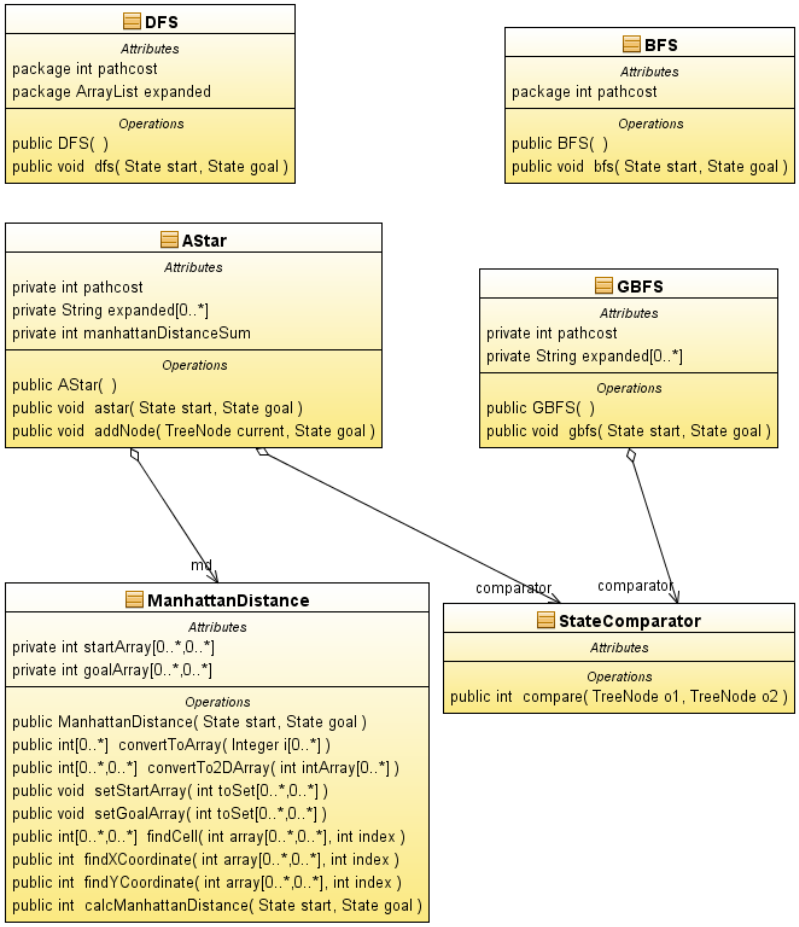


APPENDIX B
SEARCHTREE PACKAGE UML





APPENDIX C
SEARCHALGORITHM PACKAGE UML



REFERENCES

[1] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach (3rd edition)*. Prentice Hall, 2009.