CSM6120 - 8 puzzle solver

Stefan Klaus, stk4@aber.ac.uk

I. 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 add the end of this document. 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 6.
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 7

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 8.

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 I-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.

II. 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 test cases is analysed.
The test cases used in this section are shown in table I. Test cases 1 through 3 are the ones supplied with the assignment description, test case 4 through 6 have been randomly generated.

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.

TABLE I: Test cases used to test the program

Test case number	**Test case representation**
1	1 2 0 3 4 5 6 7 8
2	1 2 5 3 0 4 6 7 8
3	3 2 0 6 1 5 7 4 8
4	1 7 6 3 6 2 4 0 8
5	2 0 7 4 5 3 6 1 8
6	7 4 1 2 8 0 5 6 3

1) Breadth-First search: Table II shows the performance of Breadth-First search.
As the test result show the number of explored nodes increases significantly when more tiles are out of place. The processing time also increases.
While during test cases 1 - 3 the number of expanded nodes and the actual path cost to the goal, as well as the processing time, increases marginally from the easiest test case(1) to the more advanced one(3), the path cost, number of expanded nodes as well as the running time increases drastically when solving the randomly generated puzzle, represented as test case 4 - 6.

TABLE II: Performance of Breadth-First Search

Test case	**Path Cost**	**Expanded Nodes**	**Running Time in nanosecond**
1	3	7	5
2	29	53	7
3	67	114	11
4	58375	80176	90107
5	74135	94765	154391
6	156672	169194	964196

2) Depth-First search: Table III 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, 3 and 4 actually have a lower path cost and lower computational time than the first one, where only 3 tiles are out of place. In fact test case 6 which is by far the most advanced puzzle is the fastest to be solved, while test case 5 which is the next most advanced has the longest processing time and highest path cost of all puzzles.

Even though Depth-First search has by far the most worst time and space complexity of all the implemented algorithms, however it is bound to find a solution to the problem since the search space is finite, however in the absolute worst case the performance of Depth-First search would be equal to a simple brute force approach.

TABLE III: Performance of Depth-First Search

Test case	Path Cost	Expanded Nodes	Running Time in nanosecond
1	169022	180688	1303747
2	139063	175713	1115838
3	111950	166823	895815
4	112545	167212	854431
5	176779	181273	1307069
6	77218	132209	441747

3) Greedy Best-First search: Table IV 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 Depth-First search, however shows shows, in some cases, worse performance than Breadth-First search. 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.
Also does the simple heuristic used in this algorithm not guarantee a better solution to the, as for test cases 2 and 3 the path cost and the number of expanded nodes is,though only minimal, larger than Breadth-First search.
It shows however that using a heuristic for more advanced search problems, represented here as test cases 4 - 6, leads to far better performance than using an uninformed search.

TABLE IV: Performance of Greedy Best-First search

Test case	Path Cost	Expanded Nodes	Running Time in nanosecond
1	2	5	6
2	8	16	7
3	72	117	19
4	946	1379	100
5	950	1385	167
6	5259	7345	1378

4) A star search: Table V shows the performance of the A^* Algorithm.
Different in this table compared to the previous once is that 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 more advanced puzzle(test case 3).

However test case 4 & 5 show that a good heuristic does not always leads to a goal state.
No solution could be found for these test cases, however the Manhattan Distance is estimated to be 11 in both.
As the other test case work fine there is no reason to believe that the algorithm itself it faulty, but rather that A^* simply is not able to find a solution to every puzzle.
Test case 6 is by far the most difficult puzzle to solve and here even A^* needs multiple thousand steps to find a solution.

TABLE V: Performance of A star search

Test case	Manhattan Distance	Patch cost	Expanded nodes	Running Time in nanosecond
1	2	2	4	6
2	4	4	7	7
3	6	21	28	11
4	11	/	/	5
5	11	/	/	6
6	17	4879	6789	1206

C. Problems

In this subsection the problems encountered in the development cycle are discussed, as well as how they were solved.

1) Infinite loop: Early during the development process, after implementing Breadth-First and Depth-First search, a problem with the graph generation was encountered. The problem was that states which were already encountered got generated again.
The reason for was that no method was implemented to check if nodes were already generated. This led to infinite long search trees.

Listing 1: Solution to the Infinite Search Tree Problem

```
while (node.childrenIsEmpty() != true) {
      /*
       Add the current node to a an ArrayList
             of expanded nodes
       */
      if (expanded.contains(node.getState().
```

```
                getStringtoString()) ==
                    false) {
            expanded.add(node.getState().
                getStringtoString());
        }
        String s = node.peekChild().getState().
            getStringtoString();
        if (expanded.contains(s) == false) {
            expanded.add(s);
            searchQueue.add(node.getFirstChild());
        } else {
            node.removeFirstChild();
        }
}
```

The solution to that is shown in listing 1.
With these few lines of code all expanded nodes are saved to a arrayList of expanded nodes, which than gets iterated through every iteration to check if a newly generated state already exists. If it does not the state will be added to the search queue/stack as well as to the list of expanded nodes.
This piece of code, although modified in case of the A^* algorithm, is included in all algorithms.

This also led to improvement in the Breadth-First search algorithm. While the algorithm is complete, i.e. it is bound to find a solution in a finite search environment, but as already said the way states are generated in the algorithm leads to reoccurring states.
The implementation of this routine in the Breadth-First algorithm lead to improvements of path cost and expanded nodes. The path cost for test case 2 was reduces from 97 to 29 nodes and for test case 3 from 381 nodes to 67 nodes.

2) Implementation of the Manhattan Distance heuristics: The way the states of the puzzle was represented in this program caused problems when trying to calculated the Manhattan Distance used in the A^* algorithm.
The states were hold in a standard Java ArrayList, this caused however problems when calculating the distance.
This was, already mentioned in section I, by implementing a new class which converts the arrayList to a 2 dimensional array and then use the X and Y coordinates to calculate how much each tile had to move.
This was a problem because it took a while until it was realised that an arrayList makes it more difficult to calculate the Manhattan distance and after that how to get the algorithm for the Manhattan distance set up. A thread on stack overflow helped with finding a solution for that. The thread can be found using the URL in the footnote. [1]

III. Discussion

Section II shows the analysis of each algorithm and its performance on each of the test cases.
Inherently one could see it as a comparison of uninformed search VS informed search, with 2 examples for each type of search. It shows that for simple puzzles uninformed search methods perform equally well or better than some informed search algorithms, both in time and space complexity. It is however on the more advanced puzzles that the informedness of an heuristic shows an drastic improvement over uninformed strategies.
It also shows that an advanced heuristic can both perform better or worse than a simple heuristic. An example of that would be a direct comparison of Greedy Best-First vs A^* search. A^* uses a more advanced heuristic which leads to better results in the most test cases but it fails to find a solution for some cases.

The attributes of the algorithm play a big role as well. Depth-First search has very large time and space complexity, but is still guaranteed to find a solution as the search space is finite. Breadth-First search performs much better in every test case, however in the worst case scenario it would have the same time and space complexity as Depth-First search.
Greedy Best-First performs better for some test cases however has for the simplest puzzles worse time complexity. A^* has the best time and space complexity of all algorithms implemented in this program however, as already mentioned, fails to find a solution for some of the more advanced test cases. Whereas the other algorithms, even depth-first search, find a solution.

While it gets quite apparent that the uninformed search methods can find solutions, the informed search algorithms perform better and are generally a better choice for these kind of problems. The performance of uninformed search strategies could be tested further with implementation of algorithms such as iterative deepening, which would greatly improve on Depth-First search, which is currently not much more than a brute force approach.

IV. Conclusion

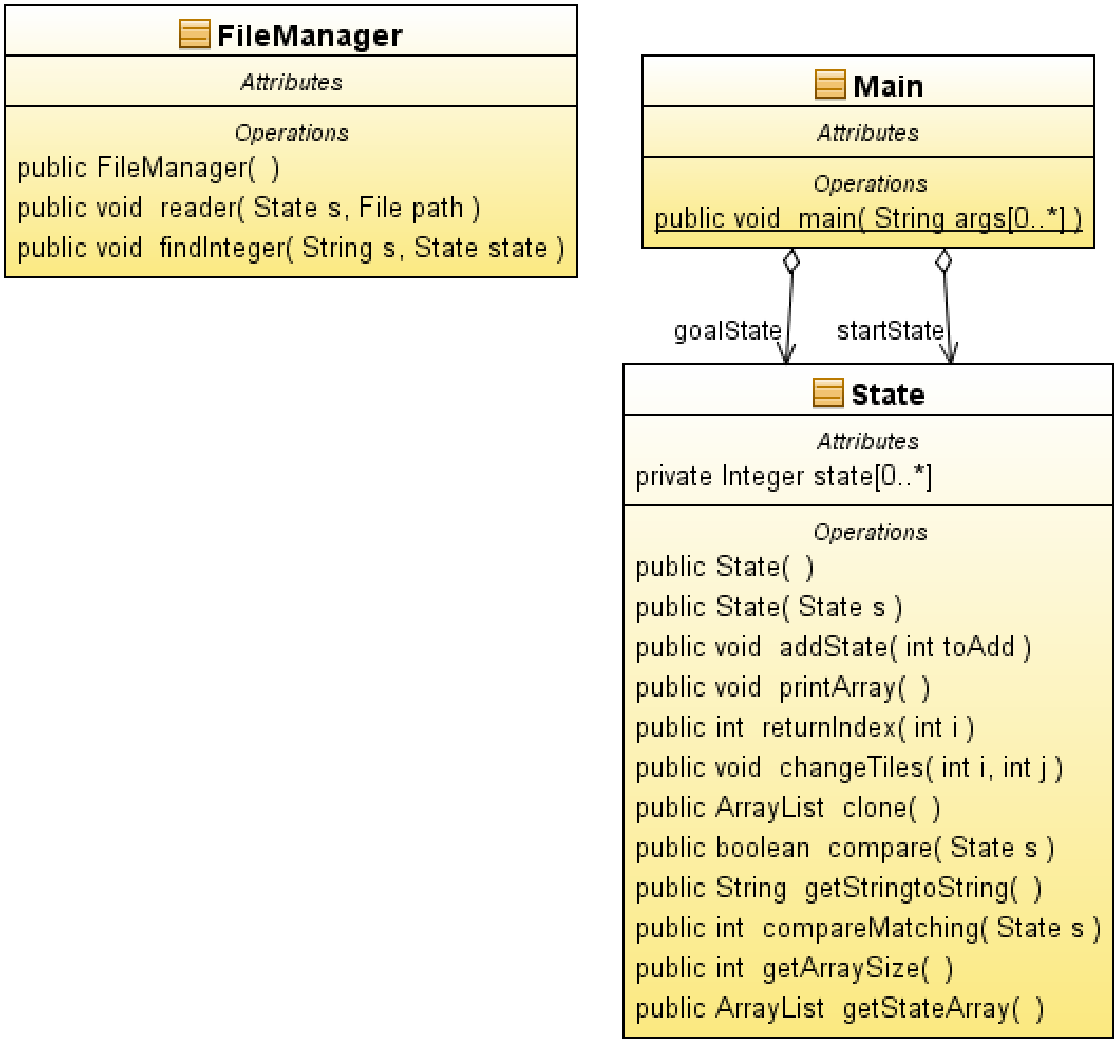
The test results delivered by the program show informed search strategies perform generally better than uninformed search strategies, even though uninformed search can find solutions faster for simple problems.
It would be interesting to implement further search search algorithms such as Iterative Deepening and Bidirectional search in the future and compare their results to that of the other algorithms.
Given more time it would also have been interesting to implement more advanced heuristics for A^* and see how different heuristics can lead to different solutions.

References

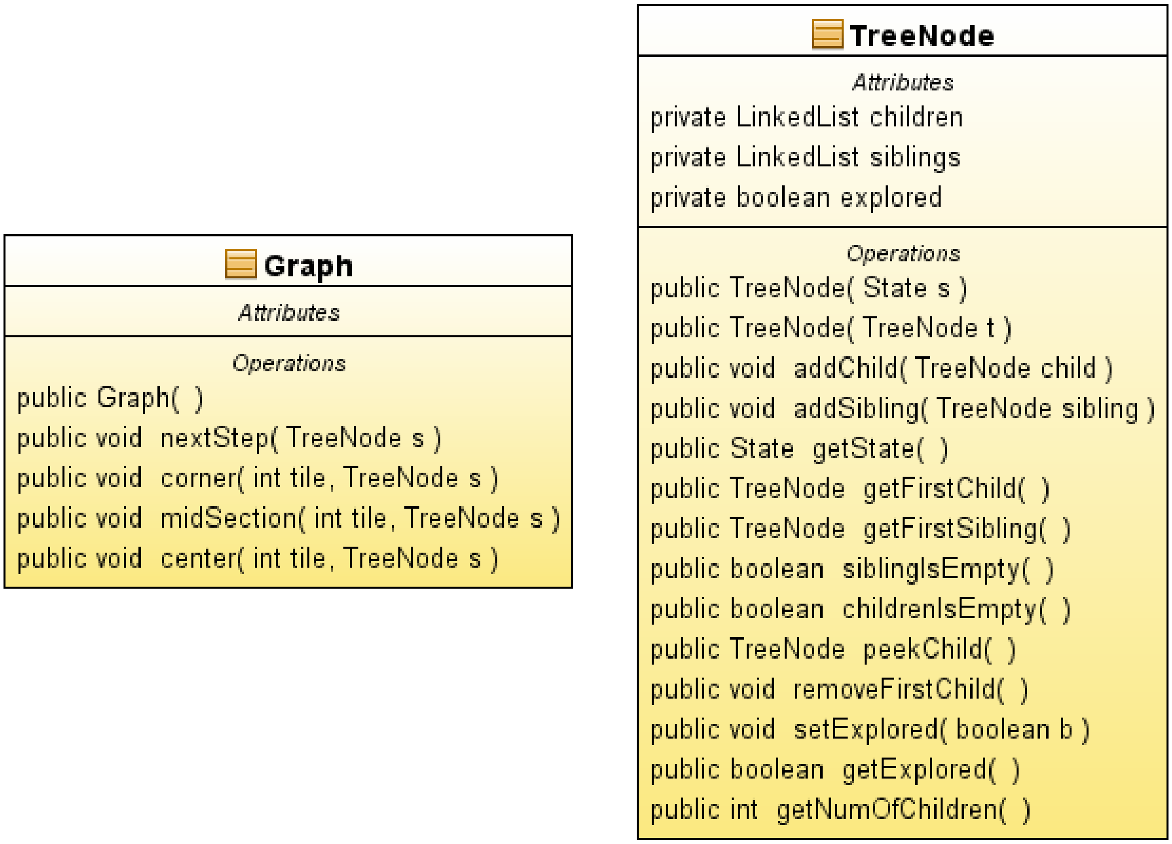
[1] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach (3rd edition)*. Prentice Hall, 2009.

[1] http://tinyurl.com/nullh9d

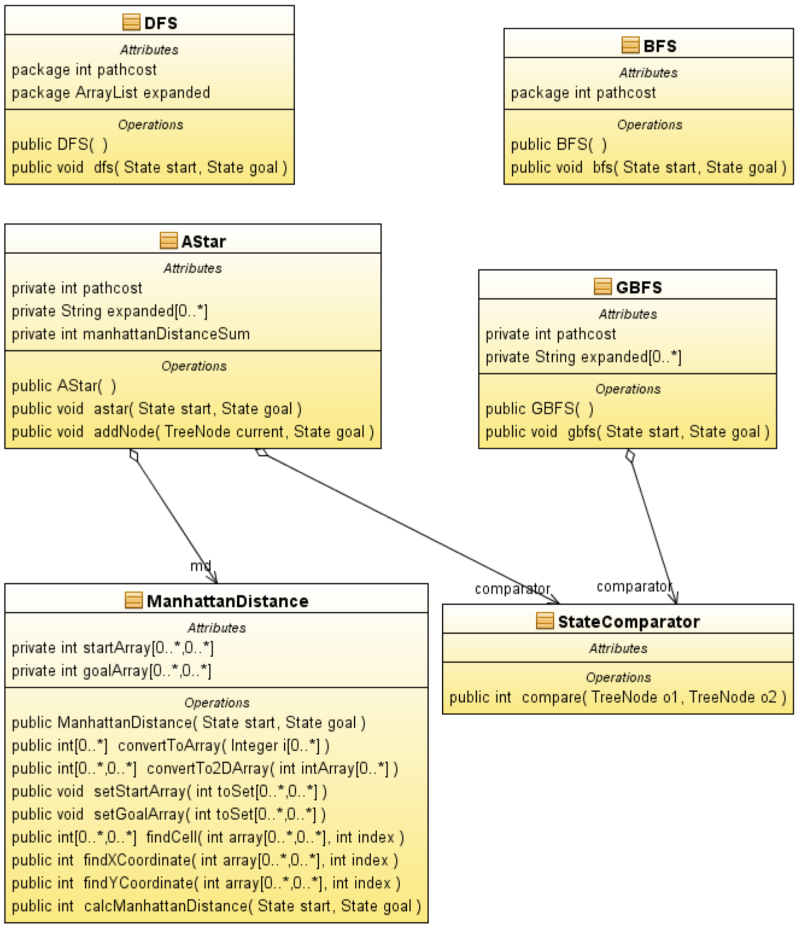
APPENDIX A
CSM1620 PACKAGE UML



Appendix B
SearchTree Package UML



APPENDIX C
SEARCHALGORITHM PACKAGE UML



8 Puzzle Solver

Generated by Doxygen 1.8.8

Mon Dec 8 2014 20:38:07

Contents

Chapter 1

Namespace Index

1.1 Packages

Here are the packages with brief descriptions (if available):

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 Package csm6120

Classes

- class **FileManager**
- class **Main**
- class **State**

5.2 Package SearchAlgortihms

Classes

- class **AStar**
- class **BFS**
- class **DFS**
- class **GBFS**
- class **ManhattanDistance**
- class **StateComparator**

5.3 Package SearchTree

Classes

- class **Graph**
- class **TreeNode**

Chapter 6

Class Documentation

6.1 SearchAlgortihms.AStar Class Reference

Collaboration diagram for SearchAlgortihms.AStar:

Public Member Functions

- **AStar** ()
- void **astar** (**State** start, **State** goal)
- void **addNode** (**TreeNode** current, **State** goal)
- void **printPath** ()

6.1.1 Detailed Description

A∗ algorithm class

Author

Stefan

6.1.2 Constructor & Destructor Documentation

6.1.2.1 SearchAlgortihms.AStar.AStar ( )

A∗ class constructor

6.1.3 Member Function Documentation

6.1.3.1 void SearchAlgortihms.AStar.addNode (TreeNode *current*, State *goal*)

Method to add a new node the the search queue. This method calculates the Manhattan Distance for every child in the current node and only add them to the search queue if the Manhattan Distance is less than the original calculated one.

Parameters

current	The current node
goal	The goal node, use for the Manhattan Distance calculation

6.1.3.2 void SearchAlgortihms.AStar.astar (State *start*, State *goal*)

A∗ algorithm

Parameters

start	The start State
goal	The goal State

6.1.3.3 void SearchAlgortihms.AStar.printPath ( )

Prints the path from the start to the goal state of the puzzle

The documentation for this class was generated from the following file:

- C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/SearchAlgortihms/**AStar.java**

6.2 SearchAlgortihms.BFS Class Reference

Collaboration diagram for SearchAlgortihms.BFS:

Public Member Functions

- **BFS** ()
- void **bfs** (**State** start, **State** goal)
- void **printPath** ()

6.2.1 Detailed Description

Class file for the Breadth first search algorithm

Author

stefan

6.2.2 Constructor & Destructor Documentation

6.2.2.1 SearchAlgortihms.BFS.BFS ( )

Constructor of the **BFS** (p. 12) object

6.2.3 Member Function Documentation

6.2.3.1 void SearchAlgortihms.BFS.bfs (State *start*, State *goal*)

Breath-First search method

Parameters

start	The start State of the graph
goal	The goal State of the graph

6.2.3.2 void SearchAlgortihms.BFS.printPath ( )

Prints the path from the start to the goal state of the puzzle

The documentation for this class was generated from the following file:

- C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/SearchAlgortihms/**BFS.java**

6.3 SearchAlgortihms.DFS Class Reference

Collaboration diagram for SearchAlgortihms.DFS:

Public Member Functions

- **DFS** ()
- void **dfs** (**State** start, **State** goal)
- void **printPath** ()

6.3.1 Detailed Description

Depth-First search algorithm class

Author

stefan

6.3.2 Constructor & Destructor Documentation

6.3.2.1 SearchAlgortihms.DFS.DFS ( )

Constructor of the **DFS** (p. 13) object

6.3.3 Member Function Documentation

6.3.3.1 void SearchAlgortihms.DFS.dfs (State *start,* State *goal*)

Depth-First Search algorithm

Parameters

start	The start state of the graph
goal	The goal state of the graph

6.3.3.2 void SearchAlgortihms.DFS.printPath ( )

Prints the path from the start to the goal state of the puzzle

The documentation for this class was generated from the following file:

- C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/SearchAlgortihms/**DFS.java**

6.4 csm6120.FileManager Class Reference

Public Member Functions

- **FileManager** ()
- void **reader** (**State** s, File path)
- void **findInteger** (String s, **State** state)

6.4.1 Detailed Description

This class holds methods to manipulate the input files.

Author

stefan

6.4.2 Constructor & Destructor Documentation

6.4.2.1 csm6120.FileManager.FileManager ( )

Constructor of the **FileManager** (p. 14) class

6.4.3 Member Function Documentation

6.4.3.1 void csm6120.FileManager.findInteger (String *s*, State *state*)

Changes the input line from being Strings to single Integers.

Parameters

s	The String to analyse and change
state	The **State** (p. 21) object to save too

6.4.3.2 void csm6120.FileManager.reader (State *s*, File *path*)

Read a given file path and calls the **findInteger()** (p. 14) method. This is used to read the input files and read them line for line.

Parameters

s	The **State** (p. 21) object to save too
path	The path of the input file

The documentation for this class was generated from the following file:

- C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/csm6120/**FileManager.java**

6.5 SearchAlgortihms.GBFS Class Reference

Collaboration diagram for SearchAlgortihms.GBFS:

Public Member Functions

- **GBFS** ()
- void **gbfs** (**State** start, **State** goal)
- void **printPath** ()

6.5.1 Detailed Description

Greedy Best-First Search class

Author

stefan

6.5.2 Constructor & Destructor Documentation

6.5.2.1 SearchAlgortihms.GBFS.GBFS ( )

Constructor of the **GBFS** (p. 14) class

6.5.3 Member Function Documentation

6.5.3.1 void SearchAlgortihms.GBFS.gbfs (State *start*, State *goal*)

Greedy Best-First Search algorithm

Parameters

start	The start State of the graph
goal	The goal State of the graph

6.5.3.2 void SearchAlgortihms.GBFS.printPath ( )

Prints the path from the start to the goal state of the puzzle

The documentation for this class was generated from the following file:

- C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/SearchAlgortihms/**GBFS.java**

6.6 SearchTree.Graph Class Reference

Public Member Functions

- **Graph** ()
- void **nextStep** (**TreeNode** s)
- void **corner** (int tile, **TreeNode** s)
- void **midSection** (int tile, **TreeNode** s)
- void **center** (int tile, **TreeNode** s)

6.6.1 Detailed Description

This class is used to generate the next step in the graph.

Author

Stefan

6.6.2 Constructor & Destructor Documentation

6.6.2.1 SearchTree.Graph.Graph ( )

Constructor for the graph object

6.6.3 Member Function Documentation

6.6.3.1 void SearchTree.Graph.center (int *tile*, TreeNode *s*)

This method generates the next level of the graph if the empty tile(0) is in the center of the puzzle. (Tile 4 in the representation below Saves all possible states to an arrayList.

0 1 2 3 4 5 6 7 8

Parameters

tile	The index of the empty tile
s	The state to base algorithm on

6.6.3.2 void SearchTree.Graph.corner (int *tile*, TreeNode *s*)

This method is used to generate the next level of the graph when the empty tile is at a corner. Saves all changes to a arrayList of possible states. The tiles where this method is used corresponds with the fields 0, 2, 6 ,and 8 as shown below

0 1 2 3 4 5 6 7 8

Parameters

tile	The index of the empty tile(0)
s	The state to base algorithm on

6.6.3.3 void SearchTree.Graph.midSection (int *tile*, TreeNode *s*)

This method to generate the next level of the graph when the empty tile(0) is on the midsection of the sides. Saves all possible states to an arrayList. The tiles where this method will be used correspond to the fields 1, 3, 5, and 7 as shown below

0 1 2 3 4 5 6 7 8

Parameters

tile	The index of the empty tile(0)
s	The state to base algorithm on

6.6.3.4 void SearchTree.Graph.nextStep (TreeNode *s*)

Algorithm to generate the next state in the graph Based on the fact that empty space can only move horizontally and vertically. To make the process easier simple numbers identifiers are assigned to the possible tiles in the puzzle. These numbers represent indices in the arrayList and are : 0 1 2 3 4 5 6 7 8 This method checks where the empty tile is and calls other methods to switch the tiles.

Parameters

s	The state on which the next step will be based

The documentation for this class was generated from the following file:

- C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/SearchTree/**Graph.java**

6.7 csm6120.Main Class Reference

Static Public Member Functions

- static void **main** (String[ ] args)

6.7.1 Detailed Description

This class is the entrance to the program.

Author

stefan

6.7.2 Member Function Documentation

6.7.2.1 static void csm6120.Main.main (String[ ] *args*) [static]

Main (p. 17) method of the program. This method can be called from the command line with a set of arguments.

javac main theStartFile theGoalFile theAlgorithmToUse

where theStartFile is a text file holding the start **State** (p. 21), theGoalFile holds the goal **State** (p. 21) of the puzzle. TheAlgorithmToUse specifies which algorithm, possibilities are:

bfs - Breadth-First search dfs - Depth-First search gbfs - Greedy Best-First search astar - A∗ search

Parameters

args	the command line arguments

The documentation for this class was generated from the following file:

- C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/csm6120/**Main.java**

6.8 SearchAlgortihms.ManhattanDistance Class Reference

Public Member Functions

- **ManhattanDistance** (**State** start, **State** goal)
- int[ ] **convertToArray** (ArrayList< Integer > i)
- int[ ][ ] **convertTo2DArray** (int[ ] intArray)
- void **setStartArray** (int[ ][ ] toSet)
- void **setGoalArray** (int[ ][ ] toSet)
- int[ ][ ] **findCell** (int[ ][ ] array, int index)
- int **findXCoordinate** (int[ ][ ] array, int index)
- int **findYCoordinate** (int[ ][ ] array, int index)
- int **calcManhattanDistance** (**State** start, **State** goal)

6.8.1 Detailed Description

This class is used to calculate the Manhattan distance for the A$*$ algorithm

Author

Stefan

6.8.2 Constructor & Destructor Documentation

6.8.2.1 SearchAlgortihms.ManhattanDistance.ManhattanDistance (State *start*, State *goal*)

Constructor of the **ManhattanDistance** (p. 17) class

Parameters

start	The State to compare to the goal
goal	The goal State to compare too

6.8.3 Member Function Documentation

6.8.3.1 int SearchAlgortihms.ManhattanDistance.calcManhattanDistance (State *start*, State *goal*)

Calculate the Manhattan distance for 2 input states

Parameters

start	The start State for the calculation
goal	The goal State to calculate the distance to

Returns

An integer representing the Manhattan Distance;

6.8.3.2 int [ ][ ] SearchAlgortihms.ManhattanDistance.convertTo2DArray (int[ ] *intArray*)

Method to convert an 1D integer array to a 2D integer array

Parameters

intArray	The integer array to convert

Returns

An 2D integer array

6.8.3.3 int [ ] SearchAlgortihms.ManhattanDistance.convertToArray (ArrayList$<$ Integer $>$ *i*)

Method to convert an ArrayList to an array

Parameters

i	The arrayList to convert

Returns

An integer array

6.8.3.4 int [ ][ ] SearchAlgortihms.ManhattanDistance.findCell (int *array[ ][ ]*, int *index*)

Method to find the X and Y coordinates of a given tile in a 2D array

Parameters

array	The 2D array to search in
index	The number/tile to search for

Returns

A 2D array holding the X and Y coordinates

6.8.3.5 int SearchAlgortihms.ManhattanDistance.findXCoordinate (int *array[ ][ ]*, int *index*)

Method to find the X coordinates of a given tile in a 2D array

Parameters

array	The 2D array to search through
index	The number/tile to search for

Returns

An integer value representing the X coordinate in a 2d Array

6.8.3.6 int SearchAlgortihms.ManhattanDistance.findYCoordinate (int *array[ ][ ]*, int *index*)

Method to find the Y coordinates of a given tile in a 2D array

Parameters

array	The 2D array to search through
index	The number/tile to search for

Returns

An integer value representing the Y coordinate in a 2d Array

6.8.3.7 void SearchAlgortihms.ManhattanDistance.setGoalArray (int *toSet[ ][ ]*)

Method to set the goalArray

Parameters

toSet	2D array to set too

6.8.3.8 void SearchAlgortihms.ManhattanDistance.setStartArray (int *toSet[ ][ ]*)

Method to set the startArray

Parameters

toSet	2D array to set too

The documentation for this class was generated from the following file:

- C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/SearchAlgortihms/**ManhattanDistance.↩ java**

6.9 csm6120.State Class Reference

Public Member Functions

- **State** ()
- **State** (**State** s)
- void **addState** (int toAdd)
- void **printArray** ()
- int **returnIndex** (int i)
- void **changeTiles** (int i, int j)
- ArrayList **clone** ()
- boolean **compare** (**State** s)
- String **getStringtoString** ()
- int **compareMatching** (**State** s)
- int **getArraySize** ()
- ArrayList **getStateArray** ()

6.9.1 Detailed Description

This class has methods and variables to hold an input state. This will be used to hold the start and goal state object.

Author

stefan

6.9.2 Constructor & Destructor Documentation

6.9.2.1 csm6120.State.State ( )

Constructor for the **State** (p. 21) object creates an empty arrayList in which the state data will be saved

6.9.2.2 csm6120.State.State (State *s*)

Constructor for the **State** (p. 21) object creates a deep clone of the state object which is specified in the parameter field

Parameters

s	The state to clone

6.9.3 Member Function Documentation

6.9.3.1 void csm6120.State.addState (int *toAdd*)

Method to add an integer to the arrayList

Parameters

toAdd	The integer to add

6.9.3.2 void csm6120.State.changeTiles (int *i*, int *j*)

Method to exchange to tiles

Parameters

i	Index of the tile to change
j	Index of the Empty tile to change

6.9.3.3 ArrayList csm6120.State.clone ( )

This method clones the arrayList and returns it

Returns

The cloned arrayList

6.9.3.4 boolean csm6120.State.compare (State *s*)

Method to compare this object to another state object

Parameters

s	The state to compare too

Returns

True if the states are the same, false if not

6.9.3.5 int csm6120.State.compareMatching (State *s*)

Method to return how many integers in this object compared to another object match

Parameters

s	The state to compare too

Returns

The number of matching ints

6.9.3.6 int csm6120.State.getArraySize ( )

Method to return the size of the state array

Returns

int value of the state array size

6.9.3.7 ArrayList csm6120.State.getStateArray ( )

Method to return the state ArrayList

Returns

The state ArrayList

6.9.3.8 String csm6120.State.getStringtoString ( )

Method to return the string representation of the "state" ArrayList

Returns

The toString representation of the "state" ArrayList

6.9.3.9 void csm6120.State.printArray ( )

Print the ArrayList

6.9.3.10 int csm6120.State.returnIndex (int *i*)

Method to return the index of a specific item in the ArrayList

Parameters

i	The item to search for

Returns

The position of the item in the ArrayList

The documentation for this class was generated from the following file:

- C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/csm6120/**State.java**

6.10 SearchAlgortihms.StateComparator Class Reference

Inheritance diagram for SearchAlgortihms.StateComparator:

Collaboration diagram for SearchAlgortihms.StateComparator:

Public Member Functions

- int **compare** (**TreeNode** o1, **TreeNode** o2)

6.10.1 Detailed Description

This class is used to compare to states together. Implements the Comparator interface

Author

Stefan

6.10.2 Member Function Documentation

6.10.2.1 int SearchAlgortihms.StateComparator.compare (TreeNode *o1*, TreeNode *o2*)

Method to compare 2 TreeNode objects for order. Compares 2 objects state string representation and orders them based on their natural ordering i.e. 0 1 2 3 4 5 6 7 8

Parameters

o1	TreeNode object 1 to compare
o2	TreeNode object 2 to compare

Returns

a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second

The documentation for this class was generated from the following file:

- C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/SearchAlgortihms/**StateComparator.java**

6.11 SearchTree.TreeNode Class Reference

Public Member Functions

- **TreeNode** (**State** s)
- **TreeNode** (**TreeNode** t)
- void **addChild** (**TreeNode** child)
- void **addSibling** (**TreeNode** sibling)
- **State getState** ()
- **TreeNode getFirstChild** ()
- **TreeNode getFirstSibling** ()
- boolean **siblingIsEmpty** ()
- boolean **childrenIsEmpty** ()
- **TreeNode peekChild** ()
- void **removeFirstChild** ()
- void **setExplored** (boolean b)
- boolean **getExplored** ()
- int **getNumOfChildren** ()

6.11.1 Detailed Description

This class represents a node in the search tree/graph

Author

stefan

6.11.2 Constructor & Destructor Documentation

6.11.2.1 SearchTree.TreeNode.TreeNode (State *s*)

Constructor of the **TreeNode** (p. 24) class. Creates a deep copy of the state which is passed as parameter Initialises the linkedLists.

Parameters

s	The State the node referes too

6.11.2.2 SearchTree.TreeNode.TreeNode (TreeNode *t*)

Constructor of the **TreeNode** (p. 24) class. Creates a deep copy of another **TreeNode** (p. 24) object.

Parameters

t	The **TreeNode** (p. 24) object this instance is a copy off

6.11.3 Member Function Documentation

6.11.3.1 void SearchTree.TreeNode.addChild (TreeNode *child*)

Method to add a child to the linkedList

Parameters

child	The **TreeNode** (p. 24) object to add to the children list

6.11.3.2 void SearchTree.TreeNode.addSibling (TreeNode *sibling*)

Method to add a sibling to the linkedList of siblings

Parameters

sibling	The **TreeNode** (p. 24) object to add to the siblings list

6.11.3.3 boolean SearchTree.TreeNode.childrenIsEmpty ( )

Method to check if the **TreeNode** (p. 24) object has children. Returns true if the linkedList is empty

Returns

Boolean "True" if the list is empty

6.11.3.4 boolean SearchTree.TreeNode.getExplored ( )

Method to get the "explored" variable of the object

Returns

The boolean value of "explored"

6.11.3.5 TreeNode SearchTree.TreeNode.getFirstChild ( )

Method to return(poll) and remove the first element of the "Children" linkedList

Returns

The head of the "children" linkedList

6.11.3.6 TreeNode SearchTree.TreeNode.getFirstSibling ( )

Method to return(poll) and remove the first element of the "siblings" linkedList

Returns

The head of the "siblings" linkedList

6.11.3.7 int SearchTree.TreeNode.getNumOfChildren ( )

Method to return the size of the "children" linkedList

Returns

The size of the List

6.11.3.8 State SearchTree.TreeNode.getState ( )

Method to return the state of the node object

Returns

the State object of the node

6.11.3.9 TreeNode SearchTree.TreeNode.peekChild ( )

Method to peek(return but not remove) the head of the "children" linkedList

Returns

The head of the "children" LinkedList

6.11.3.10 void SearchTree.TreeNode.removeFirstChild ( )

Method to remove the head of the children linkedList

6.11.3.11 void SearchTree.TreeNode.setExplored (boolean *b*)

Method to set the "explored" variable of the object

Parameters

b	The boolean value to set

6.11.3.12 boolean SearchTree.TreeNode.siblingIsEmpty ( )

Method to check if the **TreeNode** (p. 24) object has siblings. Returns true if the linkedList is empty

Returns

Boolean "True" if the list is empty

The documentation for this class was generated from the following file:

- C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/SearchTree/**TreeNode.java**

Chapter 7

File Documentation

7.1 C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/csm6120/FileManager.java File Reference

Classes

- class **csm6120.FileManager**

Packages

- package **csm6120**

7.2 C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/csm6120/Main.java File Reference

Classes

- class **csm6120.Main**

Packages

- package **csm6120**

7.3 C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/csm6120/State.java File Reference

Classes

- class **csm6120.State**

Packages

- package **csm6120**

7.4 C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/SearchAlgortihms/↩ AStar.java File Reference

Classes

- class **SearchAlgortihms.AStar**

Packages

- package **SearchAlgortihms**

7.5 C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/SearchAlgortihms/↩ BFS.java File Reference

Classes

- class **SearchAlgortihms.BFS**

Packages

- package **SearchAlgortihms**

7.6 C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/SearchAlgortihms/↩ DFS.java File Reference

Classes

- class **SearchAlgortihms.DFS**

Packages

- package **SearchAlgortihms**

7.7 C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/SearchAlgortihms/↩ GBFS.java File Reference

Classes

- class **SearchAlgortihms.GBFS**

Packages

- package **SearchAlgortihms**

7.8 C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/SearchAlgortihms/↩ ManhattanDistance.java File Reference

Classes

- class **SearchAlgortihms.ManhattanDistance**

Packages

- package **SearchAlgortihms**

7.9 C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/SearchAlgortihms/↩ StateComparator.java File Reference

Classes

- class **SearchAlgortihms.StateComparator**

Packages

- package **SearchAlgortihms**

7.10 C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/SearchTree/Graph.java File Reference

Classes

- class **SearchTree.Graph**

Packages

- package **SearchTree**

7.11 C:/Users/Stefan/Documents/GitHub/CSM6120_Assignment2/src/SearchTree/Tree↩ Node.java File Reference

Classes

- class **SearchTree.TreeNode**

Packages

- package **SearchTree**