**Subject: Computer science**

**Comparison of Breadth first search and Depth first search algorithms in mazes**

**Research Question: To what extent does the algorithms Depth first search is more efficient than Breadth first search to solve maze problems**

**Word Count: 2734**

Table of Contents

[1. Introduction 1](#_Toc119528108)

[2. Background information 3](#_Toc119528109)

[2.1 Graph theory 3](#_Toc119528110)

[2.2 Graph Traversal Algorithms 4](#_Toc119528111)

[*Depth first search (DFS)* 5](#_Toc119528112)

[*Breadth first search (BFS)* 7](#_Toc119528113)

[3. Methodology 12](#_Toc119528114)

[3.1 Results 15](#_Toc119528115)

[*3.2 Traversal patterns* 19](#_Toc119528116)

[Breadth first search 19](#_Toc119528117)

[Depth first search 20](#_Toc119528118)

[4. Conclusion 21](#_Toc119528119)

[5. References 22](#_Toc119528120)

[6. Appendix 24](#_Toc119528121)

# **1. Introduction**

Graph theory, the study of relationships. That is what it dials down to (Flovik, 2020). The study of relationship between one piece of information to another. Graph theory is used on a daily basis, from simple to understand structures such as the hierarchal systems (example in *appendix)* where the most powerful person stays at the top of the graph which he connects to people below him/her. To more complex algorithms such as topological graph (Example in *appendix*) which is what this paper will be using to explain the graph traversal algorithms. The study of graph theory revolutionized technology, transport and much more. One of the main tools used to develop and improve algorithms for graphs where mazes (Jonasson and Westerlind, 2016). Since they can be easily represented and interpreted as a graph, making mazes will be the focus of the experiment in this essay.

Mazes are one of the most well-known and effective ways that humans have developed and improved graph traversal algorithms. As stated in the previous paragraph they can act as graphs making mazes be used to develop features in applications such as navigations and social networking as they work similarly to an unweighted graph which is when values of directions are not given. Mazes are the foundations that created the most renown navigation applications since a maze can serve as a city just represented in a much simpler way where roads are the cells of the maze, and the building are represented by the walls of the maze. It is easier to develop and improve navigation due to the simplicity of a maze. All types of online connections we use today was also improved using mazes. (Beezer, 2008) Mazes also helped us humans to understand the concept of graph theory (Smith, 2020). It shows the path that each algorithm takes. By viewing the path each algorithms takes it will make it easier to have a further understanding of them. This makes the development of new traversal methods and improving o0n old methods much easier.

This essay will be exploring the fundamentals of two types of graph traversal algorithms, breadth first search and depth first search and some of their real-world applications. We will start by explaining what is graph theory and some of its terminology, explore and explain thoroughly two graph solving algorithms which are called depth first search and breadth first search and finally an experiment to analyze the research question. Our experiment aims to show how each algorithm works by exploring the path they take to solve the mazes and the time each algorithm took to complete each maze. With the experiment a greater understanding of depth first search and breadth first search will be met. The main focus of this paper will be testing the capabilities and speed of these algorithms by solving different mazes and then address the question “to what extent does the algorithms depth first search is more efficient than breadth first search to solve maze problems?”

# **2. Background information**

## **2.1 Graph theory**

To understand the different types of graph algorithms explored in this paper the concept of graph theory and its terminology is needed.Graph theory studies a mathematical structure that analyses the relationship between nodes, that are the component of the graph. A node could represent anything such as details from a website, people, numbers, blank spaces etc. The element that connects each node is called an edge. edges could be a hyperlink, family connections, wires etc.

A graph could be virtual or fiscal (In present days they are mostly virtual). In computer science graph theory is widely used in various areas. According to a student in the university of European of Tirana Graph theory is mostly seen in data mining, clustering networking, image capturing and image segmentation.

Background pattern

Description automatically generated with low confidence

Figure 1 Illustration on what is a node and an edge

Some other terminology worth noting is neighbors and degree. Neighbors is any node adjacent to another node just like a real life neighbor, and a degree is the number of neighbors a node has. In the example in *figure 1*, node-3 has neighbors’ node-2 and node-4 so it has a degree of 2.

## **2.2 Graph Traversal Algorithms**

One of the main focuses of graph theory is the creation of traversal algorithms, which consist of methods of exploring the nodes in a graph for a specific desired outcome. They can have different objectives such as exploring every node or find a desired node. These algorithms are created to solve graph theory problems due to the fact DFS and BFS have different objectives, DFS goes in random directions while BFS finds the most efficient path. They help complete many real-life computer science related issues such as maximum net flow problems which is used to find out the maximum volume a graph can hold (Jose, 2020). A real-life example of this is finding how much voltage each pole (would be represented as a node) in an electricity network would need to function (Jose, 2020). The two algorithms that will be discussed in this paper is Depth first search and Breadth first search.

### *Depth first search (DFS)*

This graph traversal algorithm uses an abstract data structure called stack. Which keeps track of which node has already been visited. A stack can only remove items in reverse order, so the last thing added will be the first thing to come out. By starting at the root node DFS will traverse the graph as much as possible until it reaches a dead end, by then the algorithm will go back to the last visited node and try to find any other passage. The algorithm knows it has explored the whole graph once it comes back to the root node. As DFS goes in random directions the time of completion for this algorithm is a lot more random than BFS due to the fact that it can reach many dead ends, or be lucky and go straight to the desired node with no errors.

**Stack –** *Stores elements inside the data structure. When items are removed the last item added will be the first one to leave the stack.*

**Diagram

Description automatically generated**

Figure 2 Pseudocode illustrating a DFS algorithm - Cormen, Leiserson Rivest. Introduction to Algorithms. MIT Press, 1990

*Visual representation:*

à Unexplored/unvisited node

à Current node being explored

à Explored node

Chart

Description automatically generated

Figure 3 Visual representation of a DFS traversal

The figure above shows the traversal path of DFS. In the first step the algorithm starts at A and has the objective to explore all nodes. As stated before DFS uses stack which is an abstract data structure where the last item which was added is the first item to be removed. The algorithm first sees if A has been explored or not, it finds out it was unexplored and now labels it as explored and puts it into the stack. Next the algorithm arbitrarily picks one of the neighbor nodes from A. In this example it chose B, DFS does the same process and labels node B as explored and puts it in the stack. The algorithm keeps repeating these same steps until it reaches a node where no neighbor node has been unexplored. In this case the algorithm reaches E. The stack has the nodes A,B,D,E in this order. Once the algorithm reaches E, it starts backtracking by removing each node, going from E to D. The algorithm then checks D’s neighbors and sees that C has not been unexplored and adds C to the queue. The stack now looks like this: A,B,D,C. The algorithm now removes C from the stack, then D and so on. Once the stack is empty again the algorithm knows it successfully explored every node.

### *Breadth first search (BFS)*

Breadth first search purpose is the find the shortest path to a desired node. Unlike D.F.S that uses stack to store its nodes, BFS uses another abstract data structure called queue to keep track of the already explored nodes. It starts at one node, then it explores all of its neighbour’s node, from that node it explores its neighbours and so on. Once it has explored all the nodes, the algorithm calculates the shortest path. BFS has to explore every node to uncover the shortest path which affects the time it completes a graph traversal.

**Queue –** *A queue stores elements just like the stack. However, it works just like a queue in real life, when you remove an item the first thing added will be the first thing to be removed.*

Diagram

Description automatically generated

Figure 4 Pseudocode illustrating a BFS algorithm - Cormen, Leiserson Rivest. Introduction to Algorithms. Mit Press, 1990.

*Visual representation*

à Unvisited/unexplored node

à Current node being explored

à Nodes in queue

à Explored node

*Chart, scatter chart

Description automatically generated*

Figure 5 Visual representation of a BFS traversal

The breadth first search algorithm starts by exploring all neighbor nodes, so in the example above the algorithm starts at node “A” and selects all neighbor nodes from it. As said before this type if algorithm uses an abstract data structure called queue which works exactly like a real-world queue. So, the first thing that enters the queue is the first thing to be removed from the queue. The algorithm will first add “A” to the queue and subsequently all its neighbors accordingly. BFS will then remove “A” form the queue and keep going in the order of the queue so the next node BFS will explore is “B”. The algorithm adds B’s neighbors to the queue. This process repeats itself until the full queue is empty. After the algorithm has scanned through the whole algorithm it will do a few calculations (explained in pseudocode) to determine the shortest path to from one node to another.

*Applications for DFS and BFS*

Applications for DFS include topological sorting, which is reordering a graph to a desired sequence, the vertex always has to be of degree 0 (Jose, 2020). Solving puzzles with only one solution, path finding. There are many more applications for depth first search. However, they will not be discussed.

Breadth first search is used more noticeably compared to breadth first search. Firstly, it is used in navigation systems such as Waze. As it finds the shortest fastest path to arrive somewhere. In social networking sites they are widely used as well, to find the shortest path from person “x” to person “y”.

Both algorithms are widely used. It only comes down to preference and what desired outcome the person would like. Some applications are path finding, finding all notes in a graph. As both graphs are used for these applications. However, DFS comes with its advantages and disadvantages, and vice versa. These advantages and disadvantages are going to be explored further in this essay.

In mazes the cell can be interpreted as a node and the path to each cell is represented by an edge. Which makes them very versatile for working with graphs, especially in computer science as computers can easily be represented as a graph. The famous mathematician Euler has used mazes to discover the branch of mathematics called topology (Zelenski, 2021) . Similar experiments have been done in relation to the graph algorithms explored in this essay. Where they tested the efficiency of the different types of graph algorithms.

# **3. Methodology**

The experiment this paper will be doing is testing the two graph traversal algorithms (Breadth first search and Depth first search) to see which one will be faster at solving a maze. The program (*Appendix)* will be using three different sized mazes with increasing complexity and timing each algorithm to see which one will be the fastest.

*Hypothesis –* As the mazes get larger the time taken for the graph traversal algorithms will take to complete the graphs will increase linearly. Also, depth first search will be faster overall than breadth first search. Although, DFS will sometimes be slower than BFS.

*Independent Variable*

An independent variable relies on what will be changed throughout my experiment. In this experiment the size of the mazes will be changed, from a 13 x 12 to a 22 x 14 and finally a 44 x 27. By doing this we will have a good idea on which algorithm will be faster and to see if one is better at solving a bigger maze.

*Dependent Variable*

The variable that will be measured in this experiment will the time taken by the algorithms. This will be done by the program for more accurate results.

*Control Variables*

* Same algorithms used
* Same data type
* Same program
* Same processor

*Method*

1 – Select the smallest maze in the data base

2 – Run program and record the times each algorithm has completed the maze (given by code)

3 – Repeat five times

4 – Re-do the experiment with the two other mazes created.

5 – Take averages of each set of mazes

*Maze 1 (13 x 12):*

Diagram

Description automatically generated



*Maze 2 (22 x 14)*

A picture containing diagram

Description automatically generated



*Maze #3 (44 x 27)*

A picture containing text, crossword puzzle

Description automatically generated



## **3.1 Results**

The tables below show all the times each algorithm has completed the mazes.

*Maze #1*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Time (Seconds) | | | | |  |
| Algorithm | Trial #1 | Trial #2 | Trial #3 | Trial #4 | Trial #5 | Average |
| Depth first search | 0.002 | 0.002 | 0.002 | 0.002 | 0.001 | 0.0018 |
| Breadth first search | 0.004 | 0.006 | 0.003 | 0.014 | 0.003 | 0.006 |

*Maze #2*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Time (Seconds) | | | | |  |
| Algorithm | Trial #1 | Trial #2 | Trial #3 | Trial #4 | Trial #5 | Average |
| Depth first search | 0.004 | 0.0 | 0.006 | 0.005 | 0.003 | 0.0036 |
| Breadth first search | 0.009 | 0.006 | 0.005 | 0.029 | 0.008 | 0.0114 |

*Maze #3*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Time (Seconds) | | | | |  |
| Algorithm | Trial #1 | Trial #2 | Trial #3 | Trial #4 | Trial #5 | Average |
| Depth first search | 0.022 | 0.025 | 0.028 | 0.025 | 0.021 | 0.0242 |
| Breadth first search | 1.344 | 1.27 | 0.926 | 1.077 | 1.165 | 1.1564 |

Figure Bar chart illustrating results collected for Depth first search

Figure Bar chart illustrating results collected for Breadth first search

*Analysis*

The hypothesis was partially correct, as the mazes got bigger and more difficult the time it took for both algorithms to complete the maze were longer. Also, D.F.S was faster and more consistent in completing the mazes. Whilst B.F.S has a much larger time to complete the mazes and being inconsistent at the same time. B.F.S is slower at solving mazes for multiple reasons. However, the main cause which makes B.F.S slower is that it must explore all nodes in the maze and then calculates which path is the fastest to the end node. D.F.S just goes to the closest node and keeps going in random directions (never back to an already explored node). Surprisingly going in random directions with no specific strategies is significantly faster at solving mazes for a computer. Even though breadth first search finds the shortest path it takes significantly longer to complete any given maze.

It can also be seen that as the mazes got bigger the time for completion for the breadth first search starts to grow exponentially. whilst depth first search also grew exponentially it still remains almost the same for every maze only increasing slightly. This counter argues with the hypothesis which stated that both algorithms will have a positive linear time of completion. It can be seen in the graph and from the results in the table, that as the mazes got bigger the completion time grew exponentially. Comparing both algorithms BFS has significantly larger increase in time as the mazes got bigger. Moreover, for every trial DFS was much faster again counter arguing the hypothesis that stated that in some trial BFS will be faster. If more trials were conducted there might have been one or two trials where BFS was faster. But, its truly random if DFS will complete in a fast or slow time.

## *3.2 Traversal patterns*

### Breadth first search

A picture containing text, crossword puzzle

Description automatically generated

The picture above shows the traversal that B.F.S has done in the largest maze. As you can see the algorithm has found the shortest path to the end of the maze. But to do this it had to scan through the whole maze (even though not show in path) to find and calculate the fastest way. This has harmed its completion time. Although this algorithm is slower it could serve as more useful if your goal is to find the shortest path to completion of a maze, graph etc.

### Depth first search

A picture containing text, crossword puzzle

Description automatically generated

It is possible to see that the path that this algorithm took is much longer, random, and less efficient compared to the breadth first search path. This is due to D.F.S’s feature which just goes and scans to its neighbor’s node with no goal of finding the shortest path, only finding the desired node. Although it is much more random this algorithm has completed the maze in a much shorter time. This may be due to less steps needed to run this type of algorithms in how no calculations are needed. In addition, occasionally the algorithm might go completely off path and can make it, so the time of completion is slower, this was not the case for the trials in this paper.

# 

# **4. Conclusion**

This essay has explored what is meant by graph theory. From that, it explained two graph traversal algorithms called, depth first search and breadth first search. It tested and compared their speeds in solving mazes of different sizes. The explanation of why one was faster than the other was also provided.

As the algorithms attempted to solve all mazes, they took more time in the bigger mazes to complete them. Which was expected as stated in the hypothesis. Furthermore, depth first search completed all mazes noticeably faster and more constant compared to breadth first search. Due to its random nature it managed to complete the maze faster as the algorithm is not required to find the fastest route. While breadth first search must find the fastest route, so it has to scan the whole maze first before completing it making this algorithm much slower.

For better results in the experiment the use of a stronger computer and larger mazes could have been. However, this was not possible due to the code restraint and the strength of the computer used to run the trials. The access to a stronger computer was not possible. Furthermore, more trials could be done to get more accurate results but due to time constraints this was not possible, not giving all possible outcomes.

# 

# **5. References**

Anderson, Ian. *The Mathematical Gazette*, vol. 70, no. 453, 1986, pp. 245–245. *JSTOR*, https://doi.org/10.2307/3615713. Accessed 15 Nov. 2022.

Beezer, Robert A. *SIAM Review*, vol. 50, no. 3, 2008, pp. 594–96. *JSTOR*, http://www.jstor.org/stable/20454149. Accessed 15 Nov. 2022.

Cormen, Leiserson Rivest. *Introduction to Algorithms*. Mit Press, 1990.

Darrien , Gabriel. “Gabrieldrn/Java-Maze-Solver: Maze Solving in Java with BFS, DFS, A\*.” *GitHub*, 5 Oct. 2018, https://github.com/gabrieldrn/Java-Maze-Solver. Accessed 13 Oct. 2022

“Depth First Search (DFS) Explained: Algorithm, Examples, and Code.” *Depth First Search (DFS) Explained : Algorithm, Examples and Code* , Reducible, 5 June 2020, https://www.youtube.com/watch?v=PMMc4VsIacU&ab\_channel=Reducible. Accessed 21 Sept. 2022.

Hayes, Brian. “Computing Science: Graph Theory in Practice: Part I.” *American Scientist*, vol. 88, no. 1, 2000, pp. 9–13. *JSTOR*, http://www.jstor.org/stable/27857955. Accessed 21 Sep. 2022.

Hayes, Brian. “Computing Science: Graph Theory in Practice: Part II.” *American Scientist*, vol. 88, no. 2, 2000, pp. 104–09. *JSTOR*, http://www.jstor.org/stable/27857986.

Accessed 21 Sep. 2022.

Jonasson, Anton, and Simon Westerlind. “Generic Algorithms in Mazes .” *Diva - Portal* , 2016, https://www.diva-portal.org/smash/get/diva2:1156189/FULLTEXT01.pdf.

Jose, Kelvin. “Common Graph Theory Problems.” *Medium*, Towards Data Science, 27 May 2020, https://towardsdatascience.com/common-graph-theory-problems-ca990c6865f1. Accessed 12 Sep. 2022.

Tosuni, Besjana. “Graph Theory in Computer Science - an Overview.” *Graph Theory in Computer Science - An Overview*, University of Europian of Tirana , 2015, https://www.idpublications.org/wp-content/uploads/2015/05/GRAPH-THEORY-IN-COMPUTER-SCIENCE-AN-OVERVIEW.pdf. Accessed 12 Sep. 2022.

“Topological Sorting.” *GeeksforGeeks*, Geeks for Geeks , 25 Sept. 2022, https://www.geeksforgeeks.org/topological-sorting/. Accessed 9 Sep. 2022.

Tosuni, Besjana. “Graph Theory in Computer Science - an Overview.” *Graph Theory in Computer Science - An Overview*, University of Europian of Tirana , 2015 , https://www.idpublications.org/wp-content/uploads/2015/05/GRAPH-THEORY-IN-COMPUTER-SCIENCE-AN-OVERVIEW.pdf. Accessed 9 Sep .2022.

“What Is Graph Theory, and Why Should You Care?” *Towardsdatascience*, Vegard Flovik, 12 Aug. 2020, https://towardsdatascience.com/what-is-graph-theory-and-why-should-you-care-28d6a715a5c2. Accessed 14 Sep. 2022.

Zelenski, Julie. “Mazes.” *CS106B Mazes*, 2021, https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1222/assignments/2-adt/maze. Accessed 16 Sep 2022

# **6. Appendix**

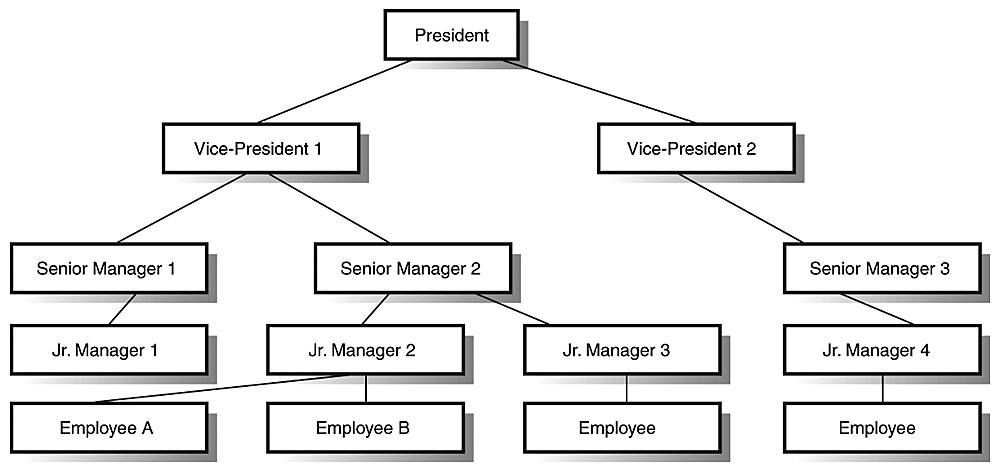
Hierarchal System

Figure 8 - “Informit.” InformIT, https://www.informit.com/articles/article.aspx?p=410742&amp;seqNum=2.

Topological graph

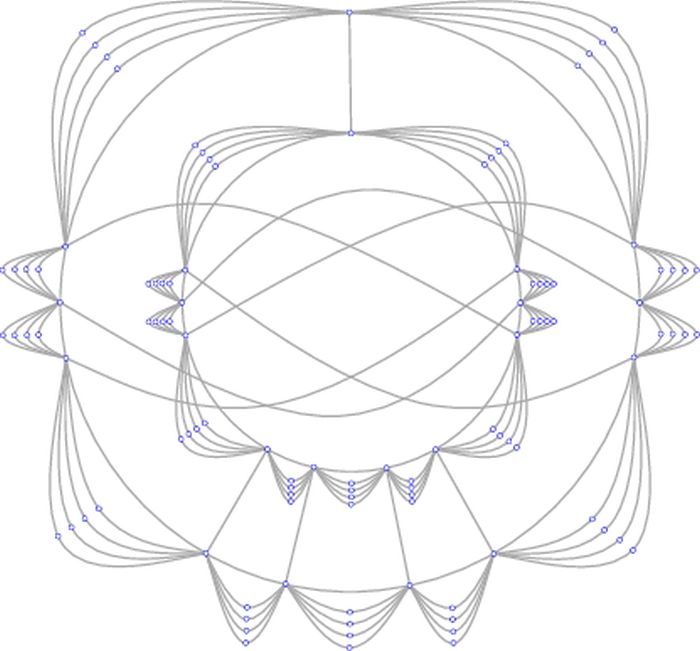


Figure - “Topological Graph.” Wikipedia, Wikimedia Foundation, 20 June 2022, https://en.wikipedia.org/wiki/Topological\_graph.

Code to test solving speed of DFS and BFS (provided by Gabriel Derrien)

**import** java.io.IOException;

**public** **class** Main

{

**public** **static** **void** main(String[] args) **throws** IOException

{

Square start1 = **new** Square(1, 0, "S");

Square end1 = **new** Square(3, 4, "E");

Maze lab1 = **new** Maze(6, 5, start1, end1);

lab1.setMazeWall(2, 0);

lab1.setMazeWall(3, 0);

lab1.setMazeWall(1, 2);

lab1.setMazeWall(2, 2);

lab1.setMazeWall(3, 2);

lab1.setMazeWall(4, 2);

lab1.setMazeWall(4, 3);

lab1.setMazeWall(1, 4);

lab1.setMazeWall(2, 4);

//====================

//====================

Maze lab2 = **new** Maze("./data/test.txt");

System.***out***.println(lab2.toString());

BFS\_Solver b2 = **new** BFS\_Solver(lab2);

DFS\_Solver d2 = **new** DFS\_Solver(lab2);

System.***out***.println(b2.solve());

System.***out***.println(d2.solve());

System.***out***.println("./data/lab\_big.txt");

}

}

**import** java.util.AbstractCollection;

**import** java.util.Iterator;

**import** java.util.LinkedList;

**public** **class** BFS\_Solver **extends** Solver

{

/\*

\* Constructor

\* m: Maze to solve

\*/

**public** BFS\_Solver(Maze m)

{

**this**.maze = m;

**this**.result = "";

**this**.frontier = **new** LinkedList<Node<Maze>>();

**this**.closedSquares = **new** LinkedList<Square>();

}

**public** String solve()

{

Boolean endfound = **false**;

**this**.nodesCounter = 0;

**this**.pathLength = 0;

//Init maze

**this**.closedSquares.clear();

**this**.maze.initMaze();

//Init frontier

**this**.frontier.clear();

**this**.frontier.add(**new** Node<Maze>(**this**.maze)); //Add initial state\

//Measure run time

**long** startTime = System.*currentTimeMillis*();

//Search

**while**(!endfound)

{

**if**(**this**.frontier.isEmpty()) //Check if the frontier is empty

**break**;

**else**

{

Node<Maze> current = ((LinkedList<Node<Maze>>) **this**.frontier).removeFirst(); //Get first node from the frontier

**this**.maze = (Maze) current.getContent(); //Get maze from the node

Square currState = **this**.maze.getCurrState(); //Get current state from the maze

//System.out.println(this.maze.printMaze());

**if**(currState.getLine() == **this**.maze.getEnd().getLine() && currState.getCol() == **this**.maze.getEnd().getCol())

{

Node<Maze> temp = **new** Node<Maze>(**this**.maze);

temp.setFather(current); //Set current as father for all next states

**this**.frontier.add(temp);

endfound = **true**;

}

**else**

{

LinkedList<Node<Maze>> nexts = **this**.getNextSquares(); //Get next possible states

**if**(!**this**.closedSquares.contains(currState))

{

**this**.closedSquares.add(currState);

currState.setAttribute("\*");

}

Iterator<Node<Maze>> x = nexts.iterator();

//Set fathers

**while**(x.hasNext())

{

Node<Maze> temp = x.next();

temp.setFather(current); //Set current as father for all next states

**this**.frontier.add(temp);

**this**.nodesCounter++;

}

}

//System.out.println(this.frontier.toString());

}

}

**long** endTime = System.*currentTimeMillis*();

**long** time = endTime - startTime;

**if**(endfound)

{

**this**.maze.resetGrid();

Node<Maze> revertedTree = ((LinkedList<Node<Maze>>) **this**.frontier).removeLast();

revertedTree = revertedTree.getFather();

**this**.result += "Path: " + **this**.maze.getEnd().toString() + "(End) <- ";

**this**.pathLength++;

**while**(revertedTree.hasFather())

{

Maze temp = revertedTree.getContent();

Square state = temp.getCurrState();

**if**(!state.equals(**this**.maze.getEnd()))

{

**this**.result += state.toString() + " <- ";

**this**.maze.getGrid()[state.getLine()][state.getCol()].setAttribute("\*");

**this**.pathLength++;

}

revertedTree = revertedTree.getFather();

}

**this**.result += **this**.maze.getStart().toString() + "(Start) \n" + "Path length: " + **this**.pathLength + "\nNumber of nodes created: " + **this**.nodesCounter + "\nExecution time: " + time/1000d + " seconds\n";

**this**.result += **this**.maze.printMaze();

}

**else**

{

**this**.result += "Failed : Unable to go further and/or end is unreachable.";

}

**return** **this**.result;

}

**public** LinkedList<Node<Maze>> getNextSquares()

{

LinkedList<Node<Maze>> res = **new** LinkedList<Node<Maze>>();

//Get 4 next squares

LinkedList<Maze> nexts = **this**.maze.getCurrState().getNexts();

**for**(**int** i = 0; i < nexts.size(); i++)

{

Square tempSq = nexts.get(i).getCurrState();

**if**(!**this**.closedSquares.contains(tempSq))

{

//this.closedSquares.add(tempSq);

//this.maze.getGrid()[tempSq.getLine()][tempSq.getCol()].setAttribute("\*");

Node<Maze> tempNode = **new** Node<Maze>(nexts.get(i));

res.add(tempNode); //Add the state

}

}

**return** res;

}

**public** String getResult()

{

**if**(result == "")

**return** "No resolution computed, please use BFS\_Solver.solve() first";

**else**

**return** **this**.result;

}

**public** AbstractCollection<Node<Maze>> getFrontier()

{

**return** **this**.frontier;

}

}

**import** java.util.AbstractCollection;

**import** java.util.Iterator;

**import** java.util.LinkedList;

**import** java.util.Stack;

**public** **class** DFS\_Solver **extends** Solver

{

/\*

\* Constructor

\* m: The maze to solve

\*/

**public** DFS\_Solver(Maze m)

{

**this**.maze = m;

**this**.result = "";

**this**.frontier = **new** Stack<Node<Maze>>();

**this**.closedSquares = **new** Stack<Square>();

}

**public** String solve()

{

Boolean endfound = **false**;

**this**.nodesCounter = 0;

**this**.pathLength = 0;

//Init maze

**this**.closedSquares.clear();

**this**.maze.initMaze();

//Init frontier

**this**.frontier.clear();

((Stack<Node<Maze>>) **this**.frontier).push(**new** Node<Maze>(**this**.maze)); //Add first state

//Measure run time

**long** startTime = System.*currentTimeMillis*();

//Search

**while**(!endfound)

{

**if**(**this**.frontier.isEmpty())

**break**;

**else**

{

Node<Maze> current = ((Stack<Node<Maze>>) **this**.frontier).pop(); //Get first node from the frontier

**this**.maze = (Maze) current.getContent();

Square currState = **this**.maze.getCurrState();

**if**(currState.getLine() == **this**.maze.getEnd().getLine() && currState.getCol() == **this**.maze.getEnd().getCol())

{

Node<Maze> temp = **new** Node<Maze>(**this**.maze);

temp.setFather(current);

((Stack<Node<Maze>>) **this**.frontier).push(temp);

endfound = **true**;

}

**else**

{

LinkedList<Node<Maze>> nexts = **this**.getNextSquares(); //Get next possible states

**if**(!**this**.closedSquares.contains(currState))

{

((Stack<Square>) **this**.closedSquares).push(currState);

currState.setAttribute("\*");

}

Iterator<Node<Maze>> x = nexts.descendingIterator();

**while**(x.hasNext())

{

Node<Maze> temp = x.next();

temp.setFather(current);

((Stack<Node<Maze>>) **this**.frontier).push(temp);

**this**.nodesCounter++;

}

}

}

}

**long** endTime = System.*currentTimeMillis*();

**long** time = endTime - startTime;

**if**(endfound)

{

**this**.maze.resetGrid();

Node<Maze> revertedTree = ((Stack<Node<Maze>>) **this**.frontier).pop();

revertedTree = revertedTree.getFather().getFather();

**this**.result += "Path: " + **this**.maze.getEnd().toString() + "(End) <- ";

**this**.pathLength++;

**while**(revertedTree.hasFather())

{

Maze temp = revertedTree.getContent();

Square state = temp.getCurrState();

**if**(!state.equals(**this**.maze.getEnd()))

{

**this**.result += state.toString() + " <- ";

**this**.maze.getGrid()[state.getLine()][state.getCol()].setAttribute("\*");

**this**.pathLength++;

}

revertedTree = revertedTree.getFather();

}

**this**.result += **this**.maze.getStart().toString() + "(Start) \n" + "Path length: " + **this**.pathLength + "\nNumber of nodes created: " + **this**.nodesCounter + "\nExecution time: " + time/1000d + " seconds\n";

**this**.result += **this**.maze.printMaze();

}

**else**

{

**this**.result += "Failed : Unable to go further and/or end is unreachable.";

}

**return** **this**.result;

}

**public** LinkedList<Node<Maze>> getNextSquares()

{

LinkedList<Node<Maze>> res = **new** LinkedList<Node<Maze>>();

//Get 4 next squares

LinkedList<Maze> nexts = **this**.maze.getCurrState().getNexts();

**for**(**int** i = 0; i < nexts.size(); i++)

{

Square tempSq = nexts.get(i).getCurrState();

**if**(!**this**.closedSquares.contains(tempSq))

{

Node<Maze> tempNode = **new** Node<Maze>(nexts.get(i));

res.add(tempNode);

}

}

**return** res;

}

**public** String getResult()

{

**if**(result == "")

**return** "No resolution computed, please use DFS\_Solver.solve() first";

**else**

**return** **this**.result;

}

**public** AbstractCollection<Node<Maze>> getFrontier()

{

**return** **this**.frontier;

}

}

**import** java.io.BufferedReader;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.IOException;

**public** **class** Maze

{

**private** Square[][] grid;

**private** Square start;

**private** Square end;

**private** Square currState;

**public** **int** lMax;

**public** **int** cMax;

**public** **char**[] order = {'N', 'E', 'S', 'W'}; //Shift order in the grid during solving in cardinals

/\*

\* Constructor with no file

\* lRange: Number of lines in the maze

\* cRange: Number of columns in the maze

\* start: The starting square of the maze

\* end: the ending square of the maze

\*/

**public** Maze(**int** lRange, **int** cRange, Square start, Square end)

{

//Init grid

**this**.grid = **new** Square[lRange][cRange];

//Set max values

**this**.lMax = lRange;

**this**.cMax = cRange;

**for**(**int** i = 0; i < lRange; i++)

{

**for**(**int** j = 0; j < cRange; j++)

{

**this**.grid[i][j] = **new** Square(i,j," ");

}

}

**this**.assignMazeToGridSquares();

//Create start and end MazeState objects contaning the start and end squares (stated)

**this**.end = end;

**this**.start = start;

**this**.currState = **this**.getStart();

//At this point, the grid is and stay the inital Maze unsolved.

}

/\*

\* Constructor with file

\* file: .txt file from where to import the maze

\*/

**public** Maze(String filepath) **throws** IOException

{

**try**

{

BufferedReader file = **new** BufferedReader(**new** FileReader(filepath));

String line = file.readLine();

String buffer = line.toUpperCase();

**int** count = 0;

//Get columns range

**this**.cMax = line.length();

**this**.lMax = 1;

//Get lines range

**while**((line = file.readLine()) != **null**)

{

**this**.lMax++;

buffer += line.toUpperCase();

}

//Init grid

**this**.grid = **new** Square[**this**.lMax][**this**.cMax];

**for**(**int** i = 0; i < **this**.lMax; i++)

{

**for**(**int** j = 0; j < **this**.cMax; j++)

{

**switch**(buffer.charAt(count++))

{

**case** 'E': **this**.setEnd(**new** Square(i, j, "E"));

**break**;

**case** 'S': **this**.setStart(**new** Square(i, j, "S"));

**break**;

**case** 'X': **this**.grid[i][j] = **new** Square(i, j, **true**);

**break**;

**default** : **this**.grid[i][j] = **new** Square(i, j, " ");

**break**;

}

}

}

file.close();

**this**.assignMazeToGridSquares();

**this**.currState = **this**.getStart();

}

**catch** (FileNotFoundException e)

{

e.printStackTrace();

}

}

**public** Maze(Square[][] grid, Square start, Square end, Square currState, **int** lMax, **int** cMax)

{

**this**.grid = grid;

**this**.start = start;

**this**.end = end;

**this**.currState = currState;

**this**.lMax = lMax;

**this**.cMax = cMax;

}

/\*

\* Retuns the starting Square

\*/

**public** Square getStart()

{

**return** start;

}

/\*

\* Sets the starting Square for the attribute and the grid

\* start: The square to set as starting square

\*/

**public** **void** setStart(Square start)

{

**this**.start = start;

**this**.grid[start.getLine()][start.getCol()] = start;

}

/\*

\* Returns the ending Square

\*/

**public** Square getEnd()

{

**return** end;

}

/\*

\* Sets the ending Square for the attribute and the grid

\* end: The square to set as ending square

\*/

**public** **void** setEnd(Square end)

{

**this**.end = end;

**this**.grid[end.getLine()][end.getCol()] = end;

}

/\*

\* Sets a wall in the maze

\* l: Line position of the wall

\* c: Column position of the wall

\*/

**public** **void** setMazeWall(**int** l, **int** c)

{

**this**.grid[l][c].setWall();

}

**public** Square getCurrState()

{

**return** **this**.currState;

}

**public** **void** setCurrState(Square c)

{

**this**.currState = c;

}

**public** **void** setNextState(Square c)

{

**this**.grid[**this**.currState.getLine()][**this**.currState.getCol()].setAttribute("\*");

**this**.currState = c;

}

**public** **void** assignMazeToGridSquares()

{

**for**(**int** i = 0; i < **this**.lMax; i++)

{

**for**(**int** j = 0; j < **this**.cMax; j++)

{

**this**.grid[i][j].assignMaze(**this**);

}

}

}

/\*

\* Initiates the maze

\*/

**public** **void** initMaze()

{

//Init grid

**this**.resetGrid();

**this**.currState = **this**.getStart();

}

**public** **void** resetGrid()

{

**for**(**int** i = 0; i < **this**.lMax; i++)

{

**for**(**int** j = 0; j < **this**.cMax; j++)

{

**if**(**this**.grid[i][j].getAttribute() == "\*")

**this**.grid[i][j].setAttribute(" ");

}

}

}

/\*

\* Sets a new shift order

\* newOrder: 4 entries char array, each char (in caps) means a shift direction, for example : ['N', 'S', 'W', 'E'] will shifts North -> South -> West -> East.

\*/

**public** **void** setOrder(**char**[] newOrder)

{

**if**(newOrder.length == 4)

{

**this**.order = newOrder;

}

}

/\*

\* Resets the default order North -> East -> South -> West (Clockwise)

\*/

**public** **void** resetOrder()

{

**char**[] o = {'N', 'E', 'S', 'W'};

**this**.order = o;

}

/\*

\* Returns the maze grid

\*/

**public** Square[][] getGrid()

{

**return** grid;

}

**public** Maze clone()

{

**return** **new** Maze(**this**.grid, **this**.start, **this**.end, **this**.currState, **this**.lMax, **this**.cMax);

}

/\*

\* Get the maze in a unicode box draw format

\*/

**public** String printMaze()

{

String res = " ";

String res\_under = "";

Square temp = **null**;

Square templineunder = **null**;

Square tempnextcol = **null**;

Square tempdiag = **null**;

//Columns numbers

**for**(**int** i = 0; i < cMax; i++)

{

**if**(i < 10)

res += " " + i + " ";

**else**

res += " " + i;

}

res += "\n ╔";

//First row : Maze top edge

**for**(**int** i = 1; i < **this**.cMax; i++)

{

temp = **this**.grid[0][i - 1];

tempnextcol = **this**.grid[0][i];

**if**(temp.isWall())

res += "═══╤";

**else**

**if**(tempnextcol.isWall())

res += "═══╤";

**else**

res += "════";

}

res += "═══╗\n";

//Browse all squares

// res = the line containing the square states

// res\_under = the graphics under the squares line with the corner unicode characters

// contatenation of res + res\_under at each line

//Example :

// │ │ │ │ <- res

// └───┼───┼───┘ <- res\_under

// │ │ <- res

// └───┘ <- res\_under

// etc...

**for**(**int** l = 0; l < **this**.lMax; l++)

{

res\_under = "";

**for**(**int** c = 0; c < **this**.cMax; c++)

{

//Get Squares

temp = **this**.grid[l][c]; // = A -> Current square

tempnextcol = temp.getEast(); // = B -> Square at the right of temp

templineunder = temp.getSouth(); // = C -> Square below temp

**if**(l < **this**.lMax - 1 && c < **this**.cMax - 1)

tempdiag = templineunder.getEast(); // = D -> Square in the temp lower right-hand diagonal

**if**(c == 0) //First colomn of current line l

{

**if**(l < 10)

res += l + " ║";

**else**

res += l + " ║";

**if**(templineunder != **null**)

{

**if**(temp.isWall() || templineunder.isWall())

res\_under += " ╟";

**else**

res\_under += " ║";

}

}

**if**(temp.isWall())

{

res += " ";

res\_under += "───";

}

**else**

{

**if**(temp.getLine() == **this**.currState.getLine() && temp.getCol() == **this**.currState.getCol())

res += " o ";

**else** **if** (temp.getLine() == **this**.start.getLine() && temp.getCol() == **this**.start.getCol())

res += " S ";

**else** **if** (temp.getLine() == **this**.end.getLine() && temp.getCol() == **this**.end.getCol())

res += " E ";

**else**

res += " " + temp.getAttribute() + " ";

**if**(l < **this**.lMax - 1)

{

**if**(templineunder.isWall())

res\_under += "───";

**else**

res\_under += " ";

}

}

//Maze right edge

**if**(c == **this**.cMax - 1)

{

res += "║";

**if**(temp != **null** && templineunder != **null**)

{

**if**(temp.isWall() || templineunder.isWall())

res\_under += "╢";

**else**

res\_under += "║";

}

}

**else**

{

//Squares corners.

// two cases : wall square or not

**if**(temp.isWall())

{

res += "│";

**if**(templineunder != **null** && tempdiag != **null** && tempnextcol != **null**)

{

//"┼" = (B + D).(C + D) -> The most reccurent corner to write

**if**((tempnextcol.isWall() || tempdiag.isWall()) && (templineunder.isWall() || tempdiag.isWall()))

res\_under += "┼";

**else**

{

**if**(!templineunder.isWall() && !tempdiag.isWall() && !tempnextcol.isWall()) //Wall on top left only

res\_under += "┘";

**else** **if**(!templineunder.isWall() && !tempdiag.isWall() && tempnextcol.isWall()) // Walls on top

res\_under += "┴";

**else** **if**(templineunder.isWall() && !tempdiag.isWall() && !tempnextcol.isWall()) // Walls on left

res\_under += "┤";

}

}

}

**else**

{

**if**(tempnextcol != **null**)

{

**if**(tempnextcol.isWall())

res += "│";

**else**

res += " ";

**if**(templineunder != **null** && tempdiag != **null**)

{

//"┼" = (C).(D) -> The most reccurent corner to write

**if**(templineunder.isWall() && tempnextcol.isWall())

res\_under += "┼";

**else**

{

**if**(!templineunder.isWall() && !tempdiag.isWall() && !tempnextcol.isWall()) //No wall

res\_under += " ";

**else** **if**(!templineunder.isWall() && tempdiag.isWall() && !tempnextcol.isWall()) //Wall on right below

res\_under += "┌";

**else** **if**(templineunder.isWall() && !tempdiag.isWall() && !tempnextcol.isWall()) //Wall on left below

res\_under += "┐";

**else** **if**(!templineunder.isWall() && !tempdiag.isWall() && tempnextcol.isWall()) //Wall on top right

res\_under += "└";

**else** **if**(!templineunder.isWall() && tempdiag.isWall() && tempnextcol.isWall()) //Walls on right

res\_under += "├";

**else** **if**(templineunder.isWall() && tempdiag.isWall() && !tempnextcol.isWall()) //Walls below

res\_under += "┬";

}

}

}

}

}

}//<- for each column

**if**(l < **this**.lMax - 1)

res += "\n" + res\_under + "\n"; //Concatenate res + res\_under

**else**

{

//Maze bottom edge

res += "\n ╚";

**for**(**int** i = 1; i < **this**.cMax; i++)

{

temp = **this**.getGrid()[l][i - 1];

**if**(temp.getEast().isWall() || temp.isWall())

res += "═══╧";

**else**

res += "════";

}

res += "═══╝\n";

}

}

/\*//Affichage simple

String res = " ";

for(int i = 0; i < cMax; i++)

{

if(i >= 9)

res += " " + i;

else

res += " " + i + " ";

}

res += "\n";

for(int i = 0; i < this.lMax; i++)

{

if(i > 9)

res += i + " ";

else

res += i + " ";

for(int j = 0; j < this.cMax; j++)

{

if(this.getGrid()[i][j].getAttribute() != "" && !this.getGrid()[i][j].isWall())

res += "[" + this.getGrid()[i][j].getAttribute() + "]";

else

res += "[■]";

}

res += "\n";

}\*/

**return** res;

}

**public** String toString()

{

**return** **this**.currState.toString();

}

}

**public** **class** Node<T>

{

**private** T value;

**private** Node<T> father;

/\*

\* Constructor

\* c: Content to bind to the Node

\*/

**public** Node(T c)

{

**this**.value = c;

**this**.father = **null**;

}

/\*

\* Returns the content binded to the Node

\*/

**public** T getContent()

{

**return** value;

}

/\*

\* Sets the content given to the Node

\* value: The content to bind

\*/

**public** **void** setContent(T value)

{

**this**.value = value;

}

/\*

\* Returns the father Node from this Node

\*/

**public** Node<T> getFather()

{

**return** father;

}

/\*

\* Sets the the father of this Node

\* father: The father to set

\*/

**public** **void** setFather(Node<T> father)

{

**this**.father = father;

}

/\*

\* Returns true if this Node have a father

\*/

**public** **boolean** hasFather()

{

**return** **this**.father != **null**;

}

/\*

\* Returns the link father -> son in a String

\*/

**public** String printFather()

{

**return** **this**.value.toString() + " <- " + **this**.father.value.toString();

}

/\*

\* Returns the toString() method of the content binded to this Node

\*/

**public** String toString()

{

**return** **this**.getContent().toString();

}

}

**import** java.util.AbstractCollection;

**import** java.util.LinkedList;

**public** **abstract** **class** Solver

{

**protected** Maze maze;

**protected** String result;

**protected** AbstractCollection<Node<Maze>> frontier;

**protected** AbstractCollection<Square> closedSquares;

**protected** **int** nodesCounter;

**protected** **int** pathLength;

**protected** Boolean manhattan;

/\*

\* Solves the maze with the related (BFS, DFS, A\*) algorithm and

\* sets the result in this format :

\* - Path trace

\* - Path length

\* - Number of nodes created

\* - The maze with the path written

\*/

**public** **abstract** String solve();

/\*

\* Get the nexts ("walkables") squares from the given square

\*/

**public** **abstract** LinkedList<Node<Maze>> getNextSquares();

/\*

\* Returns the result from the last solving

\*/

**public** **abstract** String getResult();

/\*

\* Returns the frontier from the last solving

\*/

**public** **abstract** AbstractCollection<Node<Maze>> getFrontier();

}

**import** java.lang.Math;

**import** java.util.LinkedList;

**public** **class** Square

{

**private** Maze maze;

**private** **int** l;

**private** **int** c;

**private** String attribute;

**private** **boolean** wall;

**private** **int** g;

**private** **double** h;

**private** **double** f;

/\*

\* Constructor for a "walkable" Square in the maze

\* l: Line position

\* c: Column position

\* a: Square state

\* -> S = Start position

\* -> E = End position

\* -> [Space] = Open

\* -> \* = Closed (when the solving algorithms run through the maze)

\* -> o = Current position

\*

\* -- Note: The wall attribute will be set as false because it can't be a wall.

\*/

**public** Square(**int** l, **int** c, String a)

{

**this**.l = l;

**this**.c = c;

**this**.attribute = a; //S = Start; E = End; ' ' = Playable; \* = Closed

**this**.wall = **false**;

**this**.g = 0;

**this**.h = 0;

**this**.f = 0;

}

/\*

\* Constructor for a wall square in the maze

\* l: Line poisition

\* c: Column position

\* w: Wall state

\* -> Set as true to define the square as a wall

\*/

**public** Square(**int** l, **int** c, **boolean** w)

{

**this**.l = l;

**this**.c = c;

**this**.attribute = **null**;

**this**.wall = w;

**this**.g = 0;

**this**.h = 0;

**this**.f = 0;

}

**public** **void** assignMaze(Maze m)

{

**this**.maze = m;

}

/\*

\* Returns the line position of the square in the maze

\*/

**public** **int** getLine()

{

**return** l;

}

/\*

\* Sets the line position of the square in the maze

\* l: Line position

\*/

**public** **void** setLine(**int** l)

{

**this**.l = l;

}

/\*

\* Returns the column position of the square in the maze

\*/

**public** **int** getCol()

{

**return** c;

}

/\*

\* Sets the column position of the square in the maze

\* c: Column position

\*/

**public** **void** setCol(**int** c)

{

**this**.c = c;

}

/\*

\*

\* c: The origin Square from where to get the next squares

\*/

**public** LinkedList<Maze> getNexts()

{

LinkedList<Maze> nexts = **new** LinkedList<Maze>();

**for**(**int** i = 0; i < 4; i++)

{

Maze tempMaze = **this**.maze.clone();

**if**(**this**.maze.order[i] == 'N')

{

**if**(**this**.getNorth() != **null** && !**this**.getNorth().isWall())

{

tempMaze.setNextState(**this**.getNorth());

nexts.push(tempMaze);

}

}

**else** **if** (**this**.maze.order[i] == 'E')

{

**if**(**this**.getEast() != **null** && !**this**.getEast().isWall())

{

tempMaze.setNextState(**this**.getEast());

nexts.push(tempMaze);

}

}

**else** **if** (**this**.maze.order[i] == 'S')

{

**if**(**this**.getSouth() != **null** && !**this**.getSouth().isWall())

{

tempMaze.setNextState(**this**.getSouth());

nexts.push(tempMaze);

}

}

**else** **if** (**this**.maze.order[i] == 'W')

{

**if**(**this**.getWest() != **null** && !**this**.getWest().isWall())

{

tempMaze.setNextState(**this**.getWest());

nexts.push(tempMaze);

}

}

}

**return** nexts;

}

/\*

\* Returns the Square at North from the given Square

\* c: The origin Square from where to get the North Square

\*/

**public** Square getNorth()

{

**if**(**this**.l - 1 < 0)

**return** **null**;

**else**

**return** **this**.maze.getGrid()[**this**.l - 1][**this**.c];

}

/\*

\* Returns the Square at West from the given Square

\* c: The origin Square from where to get the West Square

\*/

**public** Square getWest()

{

**if**(**this**.c - 1 < 0)

**return** **null**;

**else**

**return** **this**.maze.getGrid()[**this**.l][**this**.c - 1];

}

/\*

\* Returns the Square at South from the given Square

\* c: The origin Square from where to get the South Square

\*/

**public** Square getSouth()

{

**if**(**this**.l + 1 == **this**.maze.lMax)

**return** **null**;

**else**

**return** **this**.maze.getGrid()[**this**.l + 1][**this**.c];

}

/\*

\* Returns the Square at East from the given Square

\* c: The origin Square from where to get the East Square

\*/

**public** Square getEast()

{

**if**(**this**.c + 1 == **this**.maze.cMax)

**return** **null**;

**else**

**return** **this**.maze.getGrid()[**this**.l][**this**.c + 1];

}

/\*

\* Returns the square attribute

\* If the square is a wall, it returns null

\*/

**public** String getAttribute()

{

**return** attribute;

}

/\*

\* Sets the square attribute

\* If the attribute given is not correct, it changes nothing

\*/

**public** **void** setAttribute(String attribute)

{

**if**(attribute == " " || attribute == "S" || attribute == "E" || attribute == "\*")

{

**this**.attribute = attribute;

**this**.wall = **false**;

}

}

/\*

\* Returns wall attribute

\*/

**public** **boolean** isWall()

{

**return** **this**.wall;

}

/\*

\* Sets the square as a wall

\* Also sets the square attribute as null

\*/

**public** **void** setWall()

{

**this**.wall = **true**;

**this**.attribute = **null**;

}

//----------------------

// H Value

//----------------------

/\*

\* Returns H value

\*/

**public** **double** getH()

{

**return** **this**.h;

}

/\*

\* Computes the H value with the Manhattan distance

\* end: The ending Square in the maze

\*/

**public** **void** calcManhattanH()

{

**this**.h = Math.*abs*( **this**.getLine() - **this**.maze.getEnd().getLine() )

+ Math.*abs*( **this**.getCol() - **this**.maze.getEnd().getCol() );

}

/\*

\* Computes the H value with the Euclidean distance

\* end: The ending Square in the maze

\*/

**public** **void** calcEuclidH()

{

**this**.h = Math.*sqrt*(

Math.*pow*( **this**.getLine() - **this**.maze.getEnd().getLine(), 2 )

+ Math.*pow*( **this**.getCol() - **this**.maze.getEnd().getCol(), 2 )

);

}

//----------------------

// G Value

//----------------------

/\*

\* Returns G value

\*/

**public** **int** getG()

{

**return** g;

}

/\*

\* Increases the G value

\*/

**public** **void** incG(**int** prev)

{

**this**.g = 1 + prev;

}

//----------------------

// F Value

//----------------------

/\*

\* Computes F value by addition of H and G

\*/

**public** **double** calcF()

{

**this**.f = **this**.g + **this**.h;

**return** **this**.f;

}

/\*

\* Returns F value

\*/

**public** **double** getF()

{

**return** **this**.f;

}

/\*

\* Returns the Square in a string with the format "[LINE, COLUMN](F)"

\*/

**public** String toString()

{

**return** "[" + **this**.l + ", " + **this**.c + "](" + **this**.f + ")";

}

}