

A project report on

PATH FINDING VISUALIZER

Submitted in partial fulfillment of the requirements for the Degree of

B. Tech in Information Technology

by

Aditya Mishra (1806453)
Suman Kumar Bhunia (1806171)
Swastik Supernesh(1706175)

under the guidance of

Prof. Dr. Dipak Kumar Mohanty

School of Computer Engineering
Kalinga Institute of Industrial Technology
Deemed to be University
Bhubaneswar

November 2019

KALINGA INSTITUTE
OF INDUSTRIAL TECHNOLOGY
Deemed to be University U/S 3 of the UGC Act, 1956

CERTIFICATE

This is to certify that the project report entitled "**PATH FINDING VISUALIZER"** submitted by

Aditya Mishra	**1806453**
Suman Kumar Bhunia	**1806171**
Swastik Supernesh	**1706175**

in partial fulfillment of the requirements for the award of the **Degree of Bachelor of Technology** in **Discipline of Engineering** is a bonafide record of the work carried out under my(our) guidance and supervision at School of Computer Engineering, Kalinga Institute of Industrial Technology, Deemed to be University.

Signature of Supervisor 2 (if applicable)
Prof. Dr. Dipak Kumar Mohanty
Academic affiliation
Organization
Kalinga Institute of Industrial Technology

The Project was evaluated by us on 21/11/2021

EXAMINER 1 EXAMINER 2

EXAMINER 3 EXAMINER 4

ACKNOWLEDGEMENTS

We are profoundly grateful to Dr. Dipak Kumar Mohanty Sir for his expert guidance and encouragement throughout to see that this project meets its target since its commencement to its completion. The work is a team effort minus which the completion of the project was not possible.

Aditya Mishra
Suman Kumar Bhunia
Swastik Supernesh

ABSTRACT

Pathfinding algorithms are very important in respect of theoretical computer science and in industrial scenarios. It is very important to not only learn the algorithms and how they vary in context of time complexity but to understand how they are implemented and how they actually work. It is important to understand that algorithms behave differently in different scenarios.

Humans understand visual representations more quickly than a simple equation. So a visualization platform with fun game-like structure with support of some of the most complex algorithms will be a great tool to make students understand the algorithms in a better way.

Platform needs a UI panel for start and end points to visualize. It is done by HTML, CSS and javascript because the platform will need to trace the path dynamically and to trace the final shortest path and the algorithms are implemented in javascript.

This project mainly focuses on the algorithms visualization and to make learning algorithms fun and interesting and follow-through on this project will help them deal with it effectively.

TABLE OF CONTENTS

Abstract : 1
Table of Contents : 2

CHAPTER 1: INTRODUCTION : 8
1.1 Purpose : 8
1.2 Intended Audience and Reading Suggestion :

CHAPTER 2: BACKGROUND/BASIC CONCEPTS : 9
2.1 Objective

2.2 Background : 9

CHAPTER3: PROJECT ANALYSIS/ PROJECT IMPLEMENTATION : 22

CHAPTER 4: RESULTS AND DISCUSSION : 23

CHAPTER 5: CONCLUSION & FURTHER WORK : 24

CHAPTER 1
INTRODUCTION

Purpose

Pathfinding Algorithms play a very important role in various applications related to routing , delivery management, etc. Pathfinding algorithms address the challenges of finding a path from source to destination avoiding obstacles and minimizing the costs(time, distance, riskts, fuel, price, etc).

Humans quickly understand pictorial representation in comparison to non pictorial data. So to understand algorithms it is very important to make them work with a good pictorial representation of the flow of the algorithm.

Intended Audience and Reading Suggestion

The document is made purely for the sake of ideation and gives an overall information about the activities both in frontend and backend that shall take place. This basically represents the model and the platform required and is not considered to be the final report of business making or any sort of operation in life. The reader shall consider it a prototype, which is currently in the process of development and does not hold any particular day or date of release. However, this is a real time project and may come into existence.

CHAPTER 2
BACKGROUND

Objective

The aim/Objective of this project is to make the process of understanding pathfinding algorithms more efficient and easy to grasp. This is done by making an animated simulation of algorithms which is controlled by the user itself.

Background

To begin with we need some good background understanding in some areas related to this project.

1. HTML
2. CSS
3. Javascript
4. Path finding Algorithms

1. HTML: Since the whole platform is web based, we need HTML as a structuring language. For the whole algorithm animation we need an interface with start and end points. We also need to implement the user defined barriers. For this a grid is required because it provides cells within which we can fix our start and end points.

2. CSS: CSS stands for Cascading Style Sheets with an emphasis placed on Style. While HTML is used to structure a web document (defining things like headlines and paragraphs, and allowing you to embed images, video, and other media), CSS comes through and specifies your document's style—page layouts, colors, and fonts are all determined with CSS.

3. Javascript: JavaScript is a scripting or programming language that allows you to implement complex features on web pages — every time a web page does more than just sit there and display static information for you to look at — displaying timely content updates, interactive maps, animated 2D/3D graphics, scrolling video jukeboxes, etc.

The core client-side JavaScript language consists of some common programming features that allow you to do things like:

- Store useful values inside variables. In the above example for instance, we ask for a new name to be entered then store that name in a variable called name.
- Operations on pieces of text (known as strings in programming). In the above example we take the string Player 1: and join it to the name variable to create the complete text label, e.g. Player 1: Chris .
- Running code in response to certain events occurring on a web page. We used a click event in our example above to detect when the button is clicked and then run the code that updates the text label.

4. Pathfinding Algorithms: Pathfinding algorithms are usually an attempt to solve the shortest path problem in graph theory. They try to find the best path given a starting point and ending point based on some predefined criteria.

In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

Algorithms

DFS(Depth First Search)

The Depth-First Search (also DFS) algorithm is an algorithm used to find a node in a tree. This means that given a tree data structure, the algorithm will return the first node in this tree that matches the specified condition (i.e. being equal to a value). Nodes are sometimes referred to as vertices (plural of vertex) - here, we'll call them nodes . The edges have to be unweighted. This algorithm can also work with unweighted graphs if a mechanism to keep track of already visited nodes is added.

Description of Algorithm

The basic principle of the algorithm is to start with a start node, and then look at the first child of this node. It then looks at the first child of that node (grandchild of the start node) and so on, until a node has no more children (we've reached a leaf node). It then goes up one level, and looks at the next child. If there are no more children, it goes up one more level, and so on, until it finds more children or reaches the start node. If it hasn't found the goal node after returning from the last

child of the start node, the goal node cannot be found, since by then all nodes have been traversed.

Specifically, these are the steps:

1. For each child of the current node
2. If it is the target node, return. The node has been found.
3. Set the current node to this node and go back to 1.
4. If there are no more child nodes to visit, return to the parent.
5. If the node has no parent (i.e. it is the root), return. The node has not been found.

Runtime of the Algorithm:

The runtime of regular Depth-First Search (DFS) is $O(|N|)$ ($|N|$ = number of Nodes in the tree), since every node is traversed at most once. The number of nodes is equal to b^d, where b is the branching factor and d is the depth, so the runtime can be rewritten as $O(b^d)$.

Space of the Algorithm

The space complexity of Depth-First Search (DFS) is, if we exclude the tree itself, $O(d)$, with d being the depth, which is also the size of the call stack at maximum depth. If we include the tree, the space complexity is the same as the runtime complexity, as each node needs to be saved.

Algorithm

1. Add a root node to the stack.

2. Loop on the stack as long as it's not empty.

1. Get the node at the top of the stack(*current*), mark it as visited, and remove it.
2. For every non-visited *child* of the *current* node, do the following:
    1. Check if it's the *goal* node, If so, then return this *child* node.
    2. Otherwise, push it to the stack.
3. If the stack is empty, then the goal node was not found!

BFS(Breadth First Search)

The Breadth-first search algorithm is an algorithm used to solve the shortest path problem in a graph without edge weights (i.e. a graph where all nodes are the same "distance" from each other, and they are either connected or not). This means that given a number of nodes and the edges between them, the Breadth-first search algorithm finds the shortest path from the specified start node to all other nodes. Nodes are sometimes referred to as vertices (plural of vertex) - here, we'll call them nodes .

Description of Algorithm

The basic principle behind the Breadth-first search algorithm is to take the current node (the start node in the beginning) and then add all of its neighbors that we haven't visited yet to a queue. Continue this with the next node in the queue (in a queue that is the "oldest" node). Before we add a node to the queue, we set its distance to the distance of the current node plus 1 (since all edges are weighted equally), with the distance to the start node being 0. This is repeated until there are no more nodes in the queue (all nodes are visited).

In more detail, this leads to the following Steps:

1. Initialize the distance to the starting node as 0. The distances to all other nodes do not need to be initialized since every node is visited exactly once.
2. Set all nodes to “unvisited”
3. Add the first node to the queue and label it visited.
4. While there are nodes in the queue:

    1. Take a node out of the queue
    2. For all nodes next to it that we haven’t visited yet, add them to the queue, set their distance to the distance to the current node plus 1, and set them as “visited”

In the end, the distances to all nodes will be correct.

Runtime of the Algorithm

The runtime complexity of Breadth-first search is $O(|E| + |V|)$ ($|V|$ = number of Nodes, $|E|$ = number of Edges) if adjacency-lists are used. If a we simply search all nodes to find connected nodes in each step, and use a matrix to look up whether two nodes are adjacent, the runtime complexity increases to $O(|V|^2)$.

Depending on the graph this might not matter, since the number of edges can be as big as $|V|^2$ if all nodes are connected with each other.

Space of the Algorithm

The space complexity of Breadth-first search depends on how it is implemented as well and is equal to the runtime complexity.

Algorithm

1. Add the root node to the queue, and mark it as visited(already explored).
2. Loop on the queue as long as it's not empty.
    1. Get and remove the node at the top of the queue(*current*).
    2. For every non-visited *child* of the *current* node, do the following:
        1. Mark it as visited.
        2. Check if it's the *goal* node, If so, then return it.
        3. Otherwise, push it to the queue.
3. If the queue is empty, then the goal node was not found!

Dijkstra's Algorithm

The Dijkstra algorithm is an algorithm used to solve the shortest path problem in a graph. This means that given a number of nodes and the edges between them as well as the “length” of the edges (referred to as “weight”), the Dijkstra algorithm finds the shortest path from the specified start node to all other nodes. Nodes are sometimes referred to as vertices (plural of vertex) - here, we’ll call them nodes.

Description of the Algorithm

The basic principle behind the Dijkstra algorithm is to iteratively look at the node with the currently smallest distance to the source and update all not yet visited

neighbors if the path to it *via* the current node is shorter. In more detail, this leads to the following Steps:

1. Initialize the distance to the starting node as 0 and the distances to all other nodes as infinite
2. Set all nodes to “unvisited”
3. While we haven’t visited all nodes:

    1. Find the node with currently shortest distance from the source (for the first pass, this will be the source node itself)
    2. For all nodes next to it that we haven’t visited yet, check if the currently smallest distance to that neighbor is bigger than if we were to go via the current node
    3. If it is, update the smallest distance of that neighbor to be the distance from the source to the current node plus the distance from the current node to that neighbor

In the end, the array we used to keep track of the currently shortest distance from the source to all other nodes will contain the (final) shortest distances.

Algorithm

1. Assign dis[v] for all nodes = INT_MAX (distance from *root* node to every other node).

2. Assign dis[root] = 0(distance from *root* node to itself).

3. Add all nodes to a priority queue.

4. Loop on the queue as long as it's not empty.

1. In every loop, choose the node with the minimum distance from the *root* node in the queue(*root* node will be selected first).

2. Remove the *current* chosen node from the queue (vis[*current*] = true).

3. If the *current* chosen node is the *goal* node, then return it.

4. For every *child* of the *current* node, do the following:

    1. If the child node is not already in the queue (already visited), then skip this iteration.

    2. Assign temp = dist[*current*] + distance from *current to child node.*

    3. If temp < dist[*child*], then, assign dist[*child*] = temp. This denotes a shorter path to the child node has been found.

5. If the queue is empty, then the goal node was not found!

A* (A Star) Algorithm

The A star (A*) algorithm is an algorithm used to solve the shortest path problem in a graph. This means that given a number of nodes and the edges between them as well as the length of the edges (referred to as 'weight') and a heuristic (more on that later), the A* algorithm finds the shortest path from the specified start node to all other nodes. Nodes are sometimes referred to as vertices (plural of vertex) - here, we'll call them nodes.

Description of Algorithm

The basic principle behind the A star (A*) algorithm is to iteratively look at the node with the currently smallest priority (which is the shortest distance from the start plus the heuristic to the goal) and update all not yet visited neighbors if the

path to it *via* the current node is shorter . This is very similar to the Dijkstra algorithm, with the difference being that the lowest priority node is visited next, rather than the shortest distance node. In essence, Dijkstra uses the distance as the priority, whereas A* uses the distance *plus* the heuristic.

Why does adding the heuristic make sense? Without it, the algorithm has no idea if it's going in the right direction. When manually searching for the shortest path in this example, you probably prioritised paths going to the right over paths going up or down. This is because the goal node is to the right of the start node, so going right is at least generally the correct direction. The heuristic gives the algorithm this *spatial* information.

So if a node has the currently shortest distance but is generally going in the wrong direction, whereas Dijkstra would have visited that node next, A Star will not. For this to work, the heuristic needs to be *admissible*, meaning it has to never overestimate the actual cost (i.e. distance) - which is the case for straight line distance in street networks, for example. Intuitively, that way the algorithm never overlooks a shorter path because the priority will always be lower than the real distance (if the current shortest path is A, then if there is any way path B could be shorter it will be explored). One simple heuristic that fulfils this property is straight line distance (e.g. in a street network)

In more detail, this leads to the following Steps:

1. Initialize the distance to the starting node as 0 and the distances to all other nodes as infinite
2. Initialize the priority to the starting node as the straight-line distance to the goal and the priorities of all other nodes as infinite
3. Set all nodes to unvisited

4. While we haven't visited all nodes and haven't found the goal node:

    1. Find the node with currently lowest priority (for the first pass, this will be the source node itself)
    2. If it's the goal node, return its distance
    3. For all nodes next to it that we haven't visited yet, check if the currently smallest distance to that neighbor is bigger than if we were to go via the current node
    4. If it is, update the smallest distance of that neighbor to be the distance from the source to the current node plus the distance from the current node to that neighbor, and update its priority to be the distance plus its straight-line distance to the goal node

Algorithm

1. Assign dis[v] for all nodes = INT_MAX (distance from *root* node + heuristics of every node).

2. Assign dis[root] = 0 + heuristic(root, goal) (distance from *root* node to itself + heuristics).

2. Add the root node to the priority queue.

3. Loop on the queue as long as it's not empty.

1. In every loop, choose the node with the minimum distance from the *root* node in the queue + heuristic (*root* node will be selected first).

2. Remove the *current* chosen node from the queue (vis[*current*] = true).

3. If the *current* node is the *goal* node, then return it.

4. For every *child* of the *current* node, do the following:

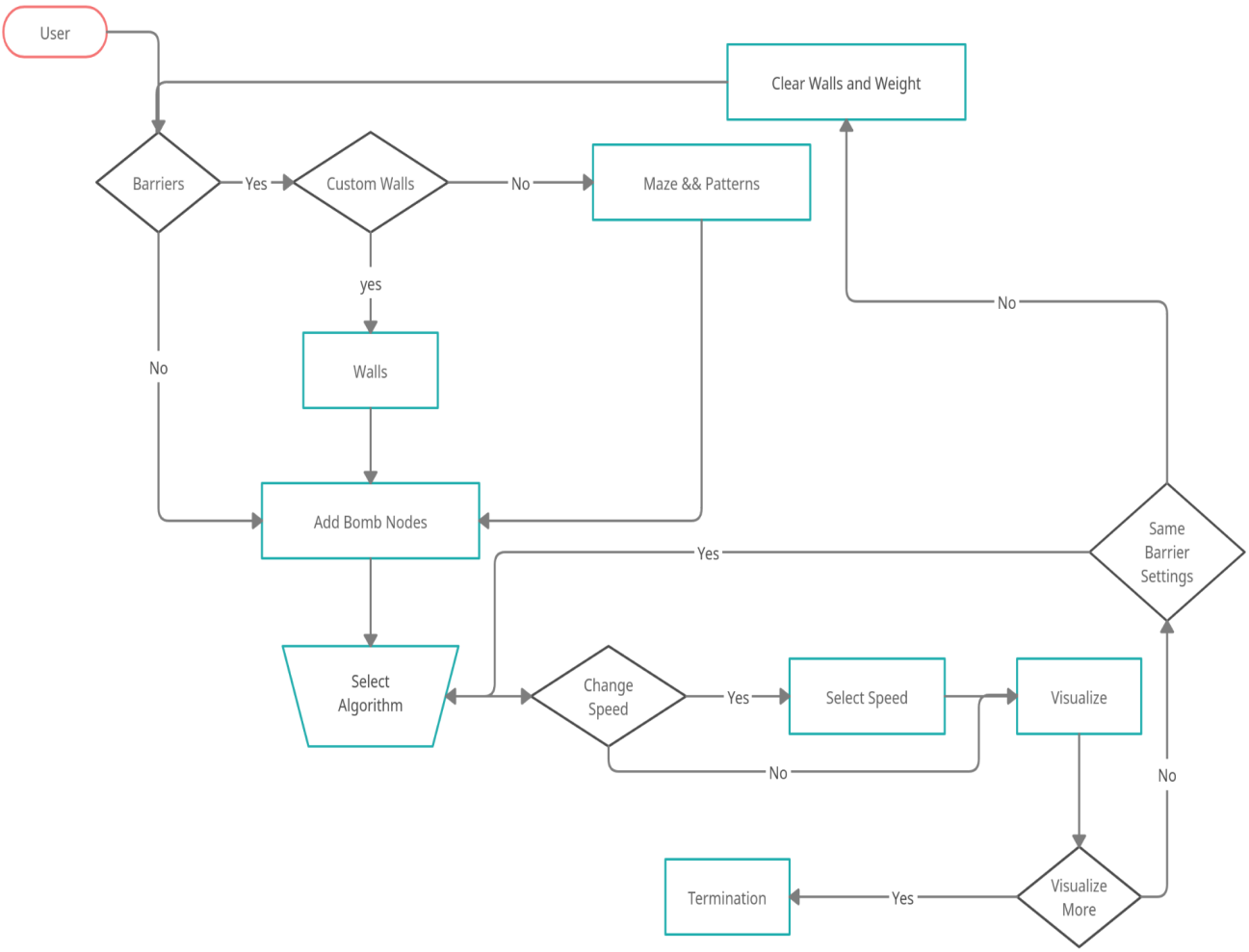
1. Assign temp = distance(root, *current)* + distance(current, child) + heuristic(child, goal).

2. If temp < dis[child], then, assign dist[*child*] = temp. This denotes a shorter path to the child node has been found.

3. And, add *child* nodes to the queue if not already in the queue (thus, it's now marked as not visited again).

4. If the queue is empty, then the goal node was not found!

Flow Diagram

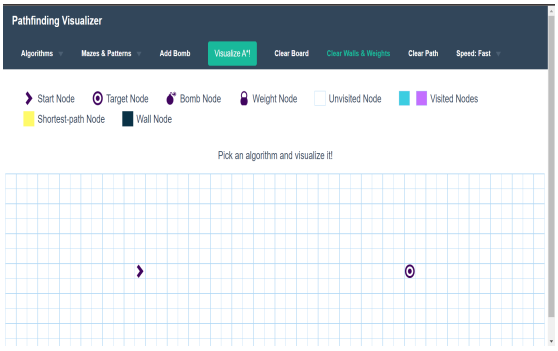


CHAPTER 3

PROJECT ANALYSIS/ PROJECT IMPLEMENTATION

The development of the project has been carved out in SixPhases. These phases include all the steps of the project, beginning from data collection and processing to the output for the user.

The Six phases are:



1. Building of Graph Matrix.

2. Added Walls and Event listeners.

3. Embed the Graph Algorithms.

4. Integrated the Path finding Functionality.

5. Improved the Design and UI.

6. Added the Timer Functionality After all these phases the project is completely ready for the user to use. Each Phase has been discussed in detail from here on onwards, letting to a complete understanding of the project

At its core, a pathfinding algorithm seeks to find the shortest path between two points. This project visualizes various pathfinding algorithms in action, and more!

All of the algorithms on this project are adapted for a 2D grid, where 90 degree turns have a "cost" of 1 and movements from a node to another have a ‘cost’ of 1.

CHAPTER 4

RESULTS & DISCUSSIONS

- Firstly select the algorithm and then select mazes and patterns .Then visualize the pattern.
- In the example we've selected A* search algorithm and then we decided to go with a basic random maze.





- Click on the grid to add a wall. Walls are impenetrable, meaning that a path cannot cross through them.
- Visualizing and more Use the navbar buttons to visualize algorithms and to do other stuff! You can clear the current path, clear walls and weights, clear the entire board, and adjust the visualization speed, all from the navbar. If you want to access this tutorial again, click on "Pathfinding Visualizer" in the top left corner of your screen.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion :

With the completion of this project, we have successfully achieved our objective of our project is to embed Graph Path Finding with Visualization and Comparing their performance. As is the case with most other teaching areas, there has been a significant gap between the theory and practical understanding of algorithms realization. This is true also for shortest paths algorithms and in particular for Dijkstra algorithm.

The main goal of the project is to use it from operations research educators and students for teaching and studying the existing known combinatorial graph algorithms. The main idea of the system is to provide an integrated educational environment for both instructors and students to facilitate the learning process in an efficient way.

This paper concludes that visualization can be achievable in the near future in each and every algorithm learning. It is part of our project; the new pathfinding has been designed and implemented. Pathfinding is plotting by computer application of the shortest route between two points; it is a more particular variant on solving maze . A part of the project describes developing a pathfinding algorithm and implementation of still behaviour.

Future Work :

The motivation behind this project was to understand how map applications work to find the destination from one point to another; With that wonder in mind I decided to visualize the path finding algorithm which I thought was close to the way maps works. Starting this project i was expecting to have it be more user friendly for example making the starting and the end node to be able to move around or the user to choose wherever he want it to start and end or even move the end node as the application is executing and make the algorithm figure out where the new position of the end node is located. From now forward that's what I will be working on to make this application more user friendly.

INDIVIDUAL CONTRIBUTION REPORT

PATH FINDING VISUALIZER

Aditya Mishra

1806453

Abstract: The aim/Objective of this project is to make the process of understanding pathfinding algorithms more efficient and easy to grasp. This is done by making an animated simulation of algorithms which is controlled by the user itself.

Contribution

1. **Contribution to the project report:** Contributed in the Introduction and background of the project.
2. **Contribution during Implementation:** Designed and implemented all the algorithms and designed the flow of UI.
3. **Contribution for the project demonstration/presentation:** Algorithms and its implementation.

Full Signature of Supervisor: Full signature of the student:

…….. ……..

INDIVIDUAL CONTRIBUTION REPORT

PATH FINDING VISUALIZER

Suman Bhunia

180171

Abstract: The aim/Objective of this project is to make the process of understanding pathfinding algorithms more efficient and easy to grasp. This is done by making an animated simulation of algorithms which is controlled by the user itself.

Contribution

1. **Contribution to the project report:** Contributed in the Analysis and Implementation of the project .
2. **Contribution during Implementation:** Designed and developed the frontend UI and animations.
3. **Contribution for the project demonstration/presentation: working of** frontend UI and animations.

Full Signature of Supervisor: Full signature of the student:

…….. ……..

INDIVIDUAL CONTRIBUTION REPORT

PATH FINDING VISUALIZER

Swastik Supernesh

1706175

Abstract: The aim/Objective of this project is to make the process of understanding pathfinding algorithms more efficient and easy to grasp. This is done by making an animated simulation of algorithms which is controlled by the user itself.

Contribution

4. **Contribution to the project report:** Contributed in the conclusion and future work.
5. **Contribution during Implementation:** Designed and developed the frontend UI and animations.
6. **Contribution for the project demonstration/presentation: working of** frontend UI and animations.

Full Signature of Supervisor: Full signature of the student:

…… ……..

Project 2

ORIGINALITY REPORT

25%	18%	1%	14%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to University of Sydney Student Paper	5%
2	riggs-devais-csinalsz.biz Internet Source	5%
3	ijmtst.com Internet Source	5%
4	medium.com Internet Source	2%
5	developer.mozilla.org Internet Source	2%
6	suyog2255.github.io Internet Source	1%
7	harshthakare70.medium.com Internet Source	1%
8	Submitted to Indian Institute of Technology, Bombay Student Paper	1%
9	dokumen.pub Internet Source	1%

10	www.coursehero.com Internet Source	1 %
11	Submitted to City University Student Paper	<1 %
12	Crina Grosan. "Problem Solving by Search", Intelligent Systems Reference Library, 2011 Publication	<1 %
13	Submitted to Queen Mary and Westfield College Student Paper	<1 %
14	Tarik Terzimehic. "Path finding simulator for mobile robot navigation", 2011 XXIII International Symposium on Information Communication and Automation Technologies, 10/2011 Publication	<1 %

Exclude quotes	On	Exclude matches	Off
Exclude bibliography	On		