

PathFinding Visualizer

Project report submitted in partial fulfillment of the requirement for
the degree of Bachelor of Technology

in

Computer Science and Engineering/Information Technology

By

Meenakshi Sharma 181419

Under the supervision of

Dr. Himanshu Jindal

to

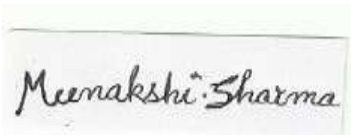


Department of Computer Science & Engineering and Information
Technology
**Jaypee University of Information Technology Waknaghat, Solan-
173234, Himachal Pradesh**

Candidate's Declaration

I hereby declare that the work presented in this report entitled "**PathFinding Visualizer**" in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology in Computer Science and Engineering/Information Technology** submitted in the department of Computer Science & Engineering and Information Technology, Jaypee University of Information Technology Waknaghat, is an authentic record of my own work carried out over a period from August 2021 to December 2021 under the supervision of **Dr. Himanshu Jindal**(Assistant Professor(SG) , Computer Science and Engineering and Information Technology).

The matter embodied in the report has not been submitted for the award of any other degree or diploma.

A rectangular box containing a handwritten signature in black ink that reads "Meenakshi Sharma".

Meenakshi Sharma,181419

This is to certify that the above statement made by the candidate is true to the best of my knowledge.

(Supervisor Signature)

Dr. Himanshu Jindal

Assistant Professor (SG)

Computer Science and Engineering and Information Technology

Dated: 14 May,2022

Acknowledgements

I would like to express my sincere gratitude to my project guide “**Dr. Himanshu Jindal**” for giving me the opportunity to work on this topic. It would never be possible for me to take this project to this level without his innovative ideas and his relentless support and encouragement.

I also thank our family, friends and all the faculties who gave their unfiltered suggestions and feedback and helped me face all the challenges and hurdles which came along during the project.

Meenakshi Sharma 181419

Table of Contents

Title Page.....	(i)
Certificate of Candidate’s Declaration.....	(iii)
Acknowledgment.....	(iv)
Table of Contents.....	(v)
List of Abbreviations.....	(vii)
List of Figures.....	(viii)
List of Graphs.....	(x)
List of Tables.....	(xi)
Abstract.....	(xii)
1. Chapter-1 Introduction	1-12
1.1 Introduction	1
1.1.1 Visualization of PF Algorithm.....	2
1.2 Problem Statement	3
1.3 Objectives	7
1.4 Methodology	8
1.5 Project Life Cycle	8
1.6 Representations of Maps	9-11
1.7 Types of algorithms	12
1.7.1 Informed Search	12
1.7.2 Uninformed Search	12
1.8 Project Organization	12
2. Chapter-2 Literature Survey	13-21
2.1 Literature Survey	13
2.2 Research Gaps	15
2.3 Research Papers and Thesis	15-18
2.4 Critical Review of Research Papers	20
2.5 Summary of PathFinding Research Papers	21
3. Chapter-3 System Development	22-66
3.1 Model System	22
3.1.1 Software and Hardware Requirements ..	23
3.1.2 Tools/Platforms used	24
3.2 Algorithms Implemented	28-66
3.2.1 BFS	29
3.2.2 DFS	32
3.2.3 Dijkstra	37
3.2.4 A* Search	41

3.2.5 D* Search	46
3.2.6 IDA* AND JPS	48
3.2.7 Maze Generation Algorithm	49
3.3 Animations and UI	58
4. Chapter-4 Performance Analysis	69-91
4.1 Sample Simulation	69
4.2 Analysis of Results	81
4.3 Summary of Comparative Study	84
5. Chapter-5 Conclusions	92-97
5.1 Conclusions	92
5.2 Future Scope	93
5.3 Applications Contributions	95
References	99
Appendix	100

List of Abbreviations

1. 2D/3D Two/Three Dimensional
2. AI Artificial Intelligence
3. BFS Breadth First Search
4. BGP Border Gateway Protocol
5. CSS Cascading Style Sheets
6. DAG Directed Acyclic Graph
7. DFS Depth First Search
8. FIFO First in First Out
9. GPS Global Positioning System
10. HPA* Hierarchical Path Finding Algorithm
11. IDA* Iterative Deepening A* Algorithm
12. IP Internet Protocol
13. JS JavaScript
14. JSON JavaScript Object Notation
15. LAN Local Area Network
16. LIFO Last in First Out
17. MIMO Multi Input Multi Output
18. Min. Minimum
19. OSPF Open Shortest Path First
20. PF Path Finding
21. PSO Particle Swarm Optimization
22. RIP Routing Information Protocol
23. UI User Interface
24. Viz. Visualization
25. VLSI Very Large-Scale Integration

List of Figures

FIGURE	TITLE	Page No.
Figure 1	3D Visualization of Bellman Ford Algorithm	2
Figure 2	Bellman Ford Algorithm	3
Figure 3	Bellman Ford Algorithm on Path finding Visualizer	3
Figure 4	3 Phases of Path Finding, Discretization	5
Figure 5	Different Grids	5
Figure 6	Navigation Mesh	5
Figure 7	Solve Convex Partitioning Problem	6
Figure 8	Path Finding in a Graph	6
Figure 9	Graph Algorithm's	6
Figure 10	Heuristical Improvements	7
Figure 11	Different Types of Grids	10
Figure 12	Navigation Mesh	11
Figure 13	Way points	11
Figure 14	Adjacency Matrix and Corresponding Graphs	14
Figure 15	Different Optimizations in PathFinding	17
Figure 16	PathFinding with Different Heuristics	18
Figure 17	Line vs Grid Movements	19
Figure 18	Particle Movement and Flowchart of PSO Algorithm	16
Figure 19	Optimal Paths on Different Grids	19
Figure 20	Shortest path using A* and Dijkstra Algorithm	22,23
Figure 21	Design of Path Finding Visualization	25
Figure 22	Structure of Path Finding Visualizer	26
Figure 23	Model of project	27
Figure 24	Different Types of Algorithms in Project	28
Figure 25	BFS Implementation as a Queue (FIFO)	30
Figure 26	BFS Visualization	31
Figure 27	BFS and DFS Code Snippet	32
Figure 28	Working of DFS	34
Figure 29	DFS Implementation as a Stack (LIFO)	35
Figure 30	DFS Visualization	37
Figure 31	Dijkstra's Implementation as a Priority Queue	38
Figure 32	Dijkstra Code Snippet	42
Figure 33	Dijkstra Visualization	42
Figure 34	Manhattan and Pythagorean Distance	44
Figure 35	Manhattan Distance Map	45
Figure 36	A* Visualization	47
Figure 37	Maze Generation Algorithms	51
Figure 39	Generating a 30x20 Maze using Kruskal's Algorithm	53

Figure 40	Generating a 30x20 Maze using Prim's Algorithm	54
Figure 41	Steps of Recursive Division	55
Figure 42	Recursive Maze Generation	56
Figure 49	Animations and their Types	61
Figure 62	Locust Swarm Algorithm for Path Finding	89
Figure 64	3D Simulation of Path Finding Visualizer	93

List of Graphs

Figure	Title	Page No.
Figure 38	Graph Theory Based Method	52
Figure 43	Recursive Division Maze	56
Figure 44	Recursive Division (Vertical Skew)	57
Figure 45	Recursive Division (Horizontal Skew)	57
Figure 46	Random Maze	58
Figure 47	Random Weights Maze	58
Figure 48	Simple Stairs Pattern	58
Figure 50	Testing Tool in Simulation 1, A*	70
Figure 51	Testing Tool in Simulation 1, Dijkstra	71
Figure 52	Testing Tool in Simulation 1, D*	72
Figure 53	Testing Tool in Simulation 2, A*	73
Figure 54	Testing Tool in Simulation 2, Dijkstra	74
Figure 55	Testing Tool in Simulation 2, D*	75
Figure 56	Testing Tool in Simulation 3, A*	76
Figure 57	Testing Tool in Simulation 3, Dijkstra	77
Figure 58	Testing Tool in Simulation 3, D*	78
Figure 59	Generating Single Source Shortest Path Algorithm	27
Figure 60,61	A* Search	87,88
Figure 63	A swarm intelligence graph-based Path Finding Algorithm	90

List of Tables

Table No.	Topic	Page No.
Table 1	Data from environment 1	79
Table 2	Data from environment 2	80
Table 3	Data from environment 3	81
Table 4	Efficiency Scores	82
Table 5	Comparison between BFS and Dijkstra	82
Table 6	Comparison between informed and uniformed algorithm	83
Table 7	Summary of Algorithm Performance	83

Abstract

Learning by Graphics and Visualization is the most effective pedagogy to understand any algorithm in the field of computer science. Theoretical concepts when learnt in a visually pleasing manner tend to interest us more and it is through these dynamic visualizations that we can see the underlying steps involved in any algorithm. This project intends to visualize all the shortest path algorithms that find the most optimal path from a source to destination node subject to constraints of time and cost. Experimentation with different sets and combinations of path finding algorithms help us get the most efficient applications of these algorithms, which can be useful in a variety of real-life applications like traffic congestion management, digital mapping services, social networking applications, robotics, game development and much more. The functionality consists of various algorithms like BFS, DFS, Dijkstra, A* Search, IDA* Search, Jump Point Search, and test simulations where all the above algorithms are compared by common parameters to get the most optimal path under each environment,

Chapter 1

Introduction

1.1 Introduction

Algorithm visualization / animation often uses dynamic graphics to visualize computation of any algorithm. Around 2000's , the advent of software tools in the form of Java and its libraries took place along with invention of complex hardware systems which changed the way algorithms were taught in computer science. Sorting , searching problems involving different data structures like trees, hash maps and graphs were being understood by visualization via animated algorithms. Animation tools and libraries were developed to allow development and research in the field of computer graphics.

Algorithm operates on a dataset, input data, variables and output data. Conventionally , graphs and trees are visualized by circles/nodes drawn by line segments , no. chains , vertical/horizontal bars via fundamental structures of matrices, vectors, real functions. An algorithm runs in steps to visualize the step-by-step flow of the algorithm. Visualizing the algorithm serves as a fundamental step to understand the internal working of it in small steps in order to develop a better sense of its utility and functioning.

1.1.1. VISUALIZATION OF PATH FINDING ALGORITHMS

Even though the example given in this segment can also be understood as an algorithm stable visualization, it is perhaps more convenient to speak about algorithmic idea visualization (AIV). Once more, there is no general method of AIV, because the underlying ideas of different algorithms in different fields have nothing in common, and each idea is unique and requires uncommon method of representation by dynamic graphic means. Well, in case, there is one general method of AIV. Even though very little is known about productive mental process that leads to discovery of new algorithms, we understand (based on our introspection) that a researcher visualization is perhaps often based, as the word recommend, on mental images - and AIV is just a straightforward projection of such mental images to a demonstration of a computer. Due to the space limitation, we give just one example - Fortune's algorithm (Fortune, 1987) for Voronoi diagram in the plane. There are several animations of the algorithm in the web, the reader is invited to look at them. The Voronoi diagram is eventually drawn, but the animations totally give no idea what the moving arcs mean and why and how they build the diagram. The algorithmic idea behind the method is following imagine the plane containing sites are enclosed as a horizontal plane into the 3-dimensional space. For each site, create a circular cone that has a vertical axis and uses the site as its apex. Observe the cone surfaces vertically from the limitless (to avoid effects of perspective). The junction of cones project to the site plane as the Voronoi diagram we are looking for. Moreover, if the "mountains" of the cones are swept by an inclined plane, the junction of the plane with the clear parts of the cones appear as the arcs that are visual in the planar animations.

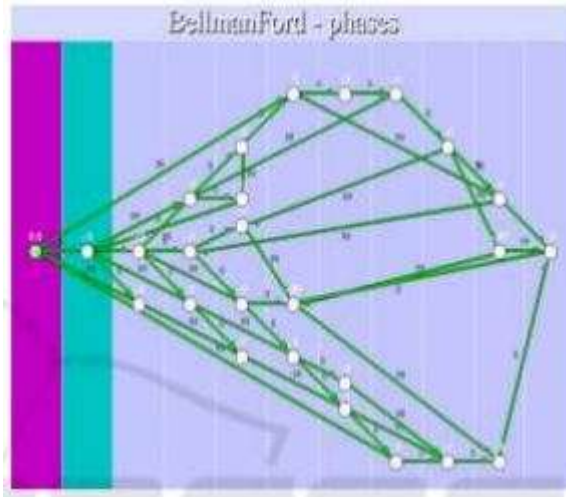


Figure 1: 3D Visualization of Bellman Ford Algorithm

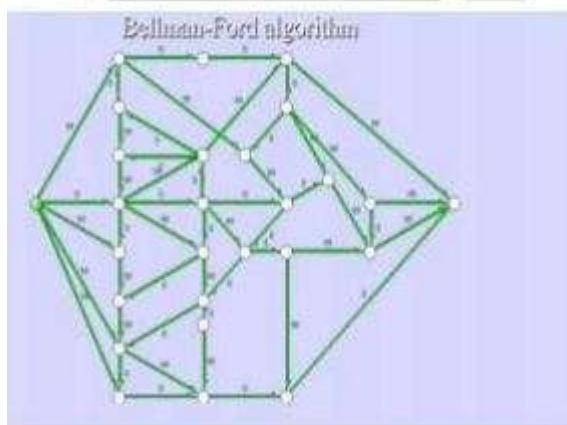


Figure 2: Bellman Ford Algorithm

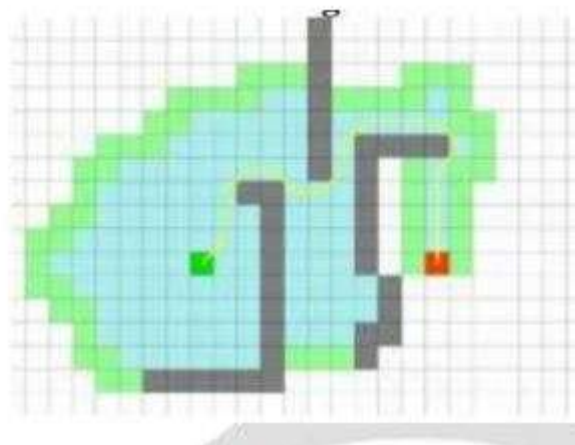


Figure 3: Bellman Ford Algorithm on Path finding Visualizer

1.2 Problem Statement

Pathfinding algorithms address the problem of finding a path from a source to a destination avoiding obstacles and minimizing the costs (time, distance, risks, fuel, price, etc.). This is a common programming challenge. Pathfinding is closely related to shortest path problem, thus the definition of pathfinding is finding the optimal path from a given start node(s) to goal node(g) in the given graph(G), where optimal refers to the shortest path, low-cost path, fastest path, or any other given criteria. Pathfinding can generally be divided into two categories: SAPF, that is Single Agent Pathfinding, to generate a path for one agent and MAPF, that is Multi-Agent Pathfinding, to generate the path for more than one agent. In this paper, we only consider the single-agent pathfinding problem in a static environment, which means the map does not change as the agent moves. Pathfinding has applications in different fields, and it is hard to consider all the application areas, so in this paper, only video game applications are used and in 2D environments.

Finding a Way is a major computer problem

- complex game worlds
- high number of businesses
- changing locations
- real-time response

The problem statement includes:

- if given first place if given first place and goal point and goal point r, get a, find a way from s to reducing a particular condition.
- search problem construction
- Find a way to reduce costs
- problem-solving and efficiency
- reduce costs depending on roadblocks

Three stages of finding a way:

1. Divide the game world

- select points and links select points and links

2. Find solution to problem of a path in given graph

- allow way points = vertices, connection = edges, allow way points = vertices, connection = edges, cost = weights cost = weights
- find the minimum route of the graph

3. Observe the movement in the game world

- aesthetic concerns
- Visual connector anxiety

The stages of path finding are mentioned as:



Figure 4: 3 Phases of Path Finding, Discretization

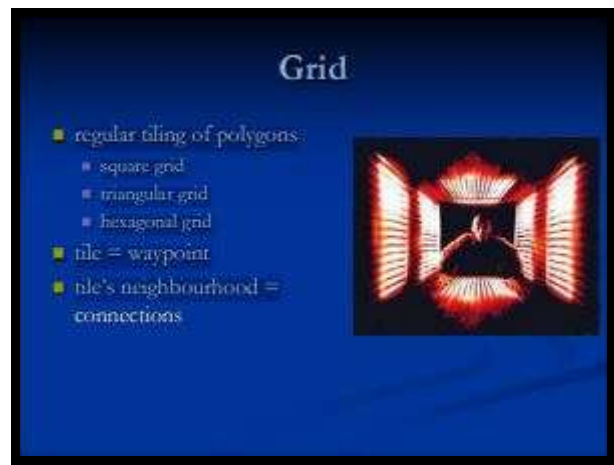


Figure 5: Different Grids

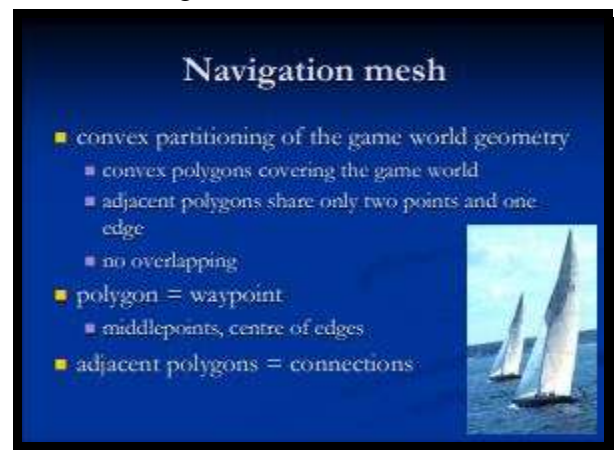


Figure 6: Navigation Mesh

Solving the convex partitioning problem

- minimize the number of polygons
 - points: n
 - points with concave interior angle (notches): $r \leq n - 3$
- optimal solution
 - dynamic programming: $O(r^2 n \log n)$
- Hertel-Mehlhorn heuristic
 - number of polygons $\leq 4 \times$ optimum
 - running time: $O(n + r \log r)$
 - requires triangulation
 - running time: $O(n)$ (at least in theory)
 - Seidel's algorithm: $O(n \log^2 n)$ (also in practice)

Figure 7: Solve Convex Partitioning Problem

Path finding in a graph

- after discretization form a graph $G = (V, E)$
 - waypoints = vertices (V)
 - connections = edges (E)
 - costs = weights of edges ($weight: E \rightarrow \mathbf{R}_+$)
- next, find a path in the graph




Figure 8: Path Finding in a Graph

Graph algorithms

- breadth-first search
 - running time: $O(|V| + |E|)$
- depth-first search
 - running time: $\Theta(|V| + |E|)$
- Dijkstra's algorithm
 - running time: $O(|V|^2)$
 - can be improved to $O(|V| \log |V| + |E|)$

Figure 9: Choosing graph algorithm

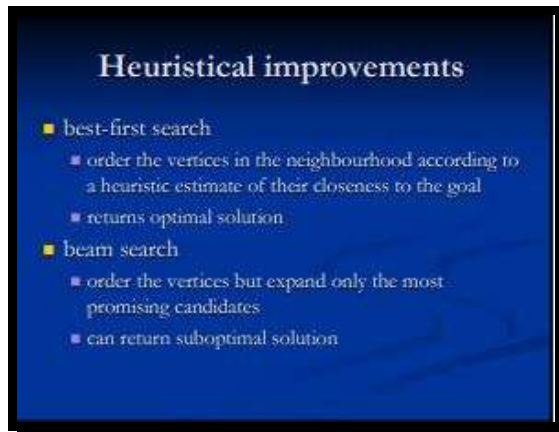


Figure 10: Heuristical Improvements

Path locating normally refers to locating the shortest direction among any two locations. Many current algorithms are designed exactly to solve the shortest path trouble consisting of Genetic, Floyd set of rules. This approach will visit unwanted nodes which might not be the nice path, and this additionally increases the time taken to discover the quality direction to our vacation spot. It might also advise us longer paths to our destination as these are uninformed algorithms.

Thus, given source and destination nodes we need to devise the minimal and optimal path connecting them, considering the obstacles in between in the most efficient manner possible.

1.3 Objectives

1. Studying a variety of weightless ways to find algorithms such as un-weighted BFS DFS because they are the basis for a graph (or tree) and often serve as benchmark for evaluating another algorithm. It covers highly developed algorithms that include heuristics such as Dijkstra, A * (A-Star), Swarm algorithm, Convergent Swarm, Bi-directional Swarm Algorithm etc.

2. Visualize the algorithms that work by finding the shortest / most direct path between the source and the target. Play with the algorithm using multiple sets of obstacles along the way between two nodes like Mazes, Obstacles such as Bomb Node and various self-designed patterns such as Simple Stair Pattern, Recursive Division etc.

3. Integrate UI Images with algorithm to visualize them working.

- Analyse their behaviour under various circumstances, obstacles and mazes
- Understand various weighty and weightless algorithms and their real-life programs such as grid-based games, distance maps, map analysis, GPS

1.4 Methodology

In essence, a path finding algorithm seeks to find the shortest path between two points. This project visualizes various path finding algorithms in action, and more. All 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.

Picking an Algorithm

Choose an algorithm from the "Algorithms" drop-down menu. Note that some algorithms are unweighted, while others are weighted. Unweighted algorithms do not take turns or weight nodes into account, whereas weighted ones do.

Additionally, not all algorithms guarantee the shortest path.

Meet the algorithms

Dijkstra's Algorithm: The father of path finding algorithms; guarantees the shortest path.

Greedy Best-first Search (weighted): A faster, more heuristic-heavy version of A*; does not guarantee the shortest path.

Breadth-first Search (unweighted): A great algorithm; guarantees the shortest path. Depth-first Search (unweighted): A very bad algorithm for path finding; does not guarantee the shortest path.

Adding walls

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.

We can clear the current path, clear walls, and weights, clear the entire board, and adjust the visualization speed, all from the navbar.

Setting up the tutorial, click on "Pathfinding Visualizer" in the top left corner of your screen.

- **Add Mazes and various patterns**

To test the algorithms with different sets of paths

- **Clear Board, Weights and Walls**

UI Options to clear the grid to clear the screen

- **Terrain settings**

Different distance metrics, diagonal options etc., Adjust Speed and Visualization Level

1.5 Project Life cycle

Project Design and Implementation is recorded in seven phases. These sections cover all project steps, from data collection and processing to user output.

The 7 categories are:

- 1. Graph Matrix Construction.
- 2. Added Walls, Obstacles like bomb nodes, weight nodes.
- 3. Design the JS Scripts and integrate the Graph Algorithms.
- 4. Measure and Improve Finder Performance.
- 5. Improved Animation themes, design, and UI.
- 6. Added Speed, Controls and associated visuals, Different Heuristics and Mazes, Patterns
- 7. Adding Different Location Features, Search Parameters and Combinations
- Post these we evaluated the project on different simulations and recorded the results.

1.6 Representations of Map

Pathfinding is used in a wide variety of areas and usually implemented on different maps that are generated to test pathfinding algorithms. The widely popular maps are

implemented using a grid-based graph, set of nodes and edges, representations in the algorithm. Usually, a grid is superimposed over a map and then the graph is used to find the optimal path. Most widely used representations are square tile grid which can either be accessed as a 4-way path or 8-way path. Both have their own advantages and disadvantages. Grids are considered simple and easy to implement and are commonly used by researchers. Other representations are Hexagonal grid, Triangular grid, Navigation Mesh, and Waypoints. Various types of map representations are discussed below briefly:

1.6.1 **Tile Grids:** The composition of the grid includes vertices or points that are connected by edges. Basically, grids uniformly divide the map world into smaller groups of regular shapes called “tiles”. The movement in square tile grids can either be 4-way (no 4-diagonal movement) or 8-way (diagonal movement). The second most widely used grid representations are Hexagonal grids. Hexagonal grids are like square grids with the same properties and take less search time and reduced memory complexities. Triangular grids are not popular among game developers and researchers, but some methods are proposed to reduce the search effort and time consumption

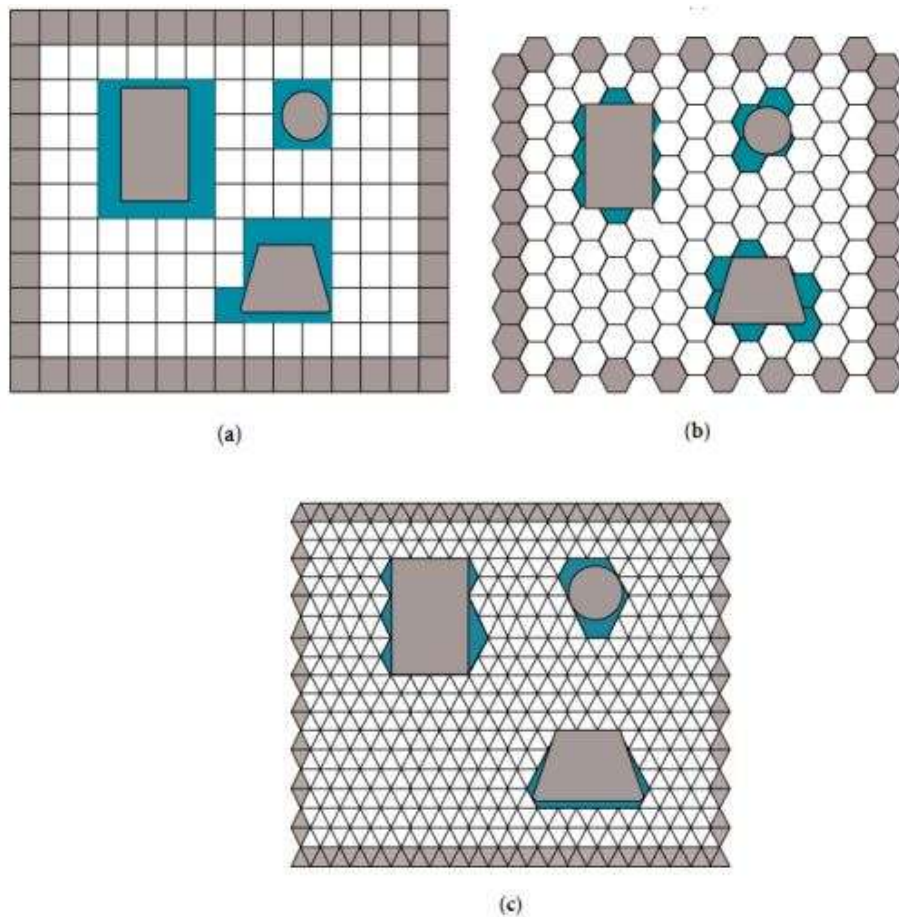


Figure 11: Different types of tile/square grids

1.6.2 Navigation Mesh: A navigation mesh is a connected graph of convex polygons, where the polygon is a node in a graph, also known as navmesh. Polygons represent a walkable area, thus movement in any direction is possible within the polygon. The map is pre-processed to generate nav-mesh and then the path can be found by traversing polygons (from polygon consisting of start point to polygon consisting of goal point). The benefits of using navigation mesh are that it reduces the number of nodes in the graph as the large walkable area can be represented as a single convex polygon, reduces the memory required to store pre-processed map, and increases the speed of pathfinding.

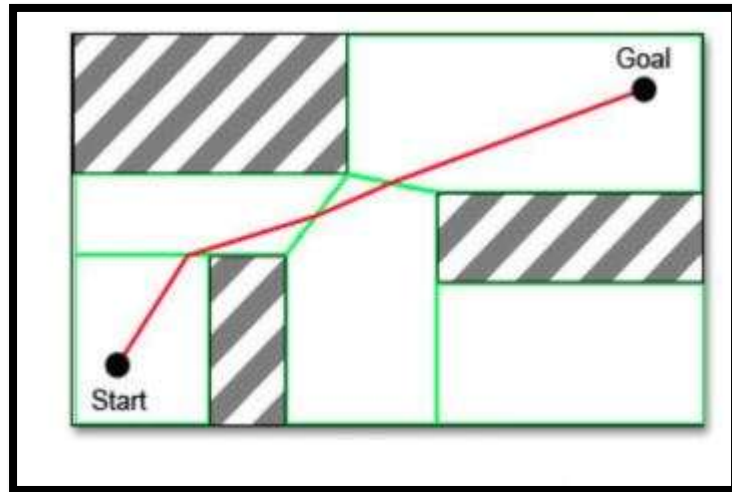


Figure 12: Navigation mesh

1.6.3 Waypoints: A waypoint can be defined as a point along the path which can be marked manually or can be automatically computed. The purpose of waypoints is to minimize the path representation as the shortest path can be pre-computed between any two points. Therefore, certain optimization techniques are developed to compute the path using waypoints. The main advantage of waypoints can be in a static world as the map does not change, so the shortest paths between two waypoints can be pre-computed and stored, reducing the time to calculate the final path after execution

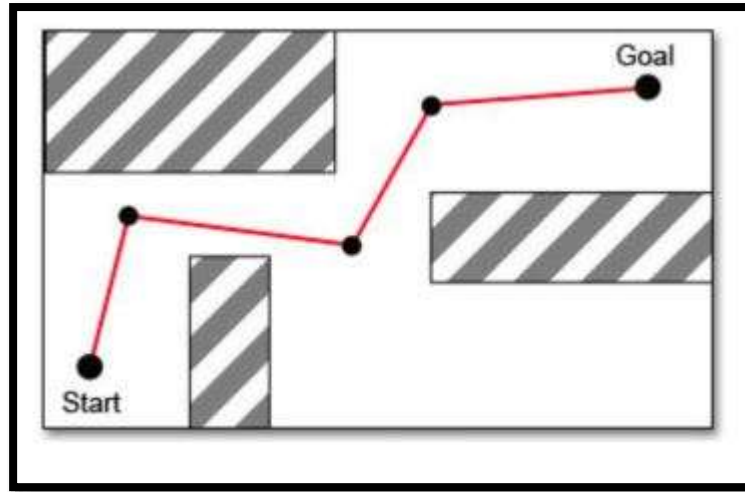


Figure 13: Way points

1.7 Algorithms

For finding a path between two nodes in each graph a search algorithm is required. Many search algorithms have been developed for graph-based pathfinding. Pathfinding algorithm generally finds the path by expanding nodes and neighboring nodes according to some given criteria. Pathfinding algorithms can be broadly divided into two categories: Informed and Uninformed pathfinding Algorithms.

1.7.1 **Informed Pathfinding Algorithms:** As the name suggests informed means having prior information about the problem space before searching it. Informed search refers to the use of knowledge about the search space like problem map, estimated costs, an estimate of goal location. Thus, the algorithm utilizes this information while searching a path and it makes pathfinding fast, optimal and reduces memory usage in node expansion . Various algorithms that fall under this category are A*, IDA*, D*, HPA*, and many more. These algorithms use different heuristic functions or uniform cost function to utilize the 7 information of the search problem. The following heuristic functions, used by these algorithms, are discussed briefly here:

Manhattan Distance: Manhattan distance is considered as a standard heuristic for the square grid, defined as the sum of the absolute difference between the start and goal position cartesian co-ordinates. In pathfinding, the Manhattan distance is the distance between start node to goal node when the movement is restricted to either vertical or horizontal axes in a square grid. The heuristic function is given below: $h(x) = |x1 - x2| + |y1 - y2|$

Octile Distance: Octile distance is the distance between two points when diagonal movement is possible along with horizontal and vertical. The Manhattan distance for going 3 up and then 3 right will be 6 units whereas only 3 units diagonally (octile distance). The function of octile distance is given below: $h(x) = \max((x1 - x2), (y1 - y2)) + (\sqrt{2} - 1) * \min((x1 - x2), (y1 - y2))$

Euclidean Distance: When any angle movement, not the grid directions (horizontal, vertical, diagonal), is allowed then the straight-line distance is the shortest distance between any two points which is also known as Euclidean

distance. The function is given below: $h(x) = \sqrt{(|x_1 - x_2| + |y_1 - y_2|)}$ In uniform cost search the next node is selected based on the cost so far, so the lowest cost node gets selected at each step. It is complete and optimal but not efficient as it takes lot of time to explore nodes.

- 1.7.2 **Uninformed Pathfinding Algorithms:** Uninformed pathfinding refers to finding the path without any knowledge of the destination in the search space with only information about start node and adjacent nodes, also known as blind search. Thus, the algorithm blindly searches the space by exploring adjacent nodes to the current node. Breadth-first search, depth-first search, Dijkstra are some algorithms that fall under this category. Uninformed search is slow and consumes lots of memory in storing nodes as it searches whole space until the destination node is found. The uninformed pathfinding is also known as an undirected search approach, which simply does not spend any time in planning. It just explores the nodes that relate to the current node and then explore their neighbor nodes and so on until finds the node marked as goal node.

1.8 Project Organization

This report consists of 6 units. The first unit defines the shortest path finding problem and its sub problems. It discusses different map representations and main types of algorithms. The second unit analyses the literature findings and highlights the gap in the research papers. The third unit describes analytical and computational model development and implementation details. The fourth unit is a comparative study of different algorithm under simulated environments to generate a set of tests to help determine efficiency of algorithms. The fifth unit concludes the findings and defines the future scope of the project in addition to the real-world applications of the algorithms hence studied under test.

Chapter 2

Literature Review

2.1 Literature Survey

An important area of mathematical theory is mathematics that studies the structure of abstract relationships between objects by means of diagrams (networks). Although the study of these structures can be purely theoretical, they can be used to model pairwise relationships in many real-world systems. One of the widely used applications is to determine the shortest path in many practical applications such as: maps; robotic navigation; texture map; typesetting in TeX; urban traffic planning; optimized pipeline of VLSI chips; subroutines in advanced algorithms; telephony programmer; telecommunications message routing; approximation of thin linear function; network routing protocols (OSPF, BGP, RIP); exploit arbitrage opportunities in currency exchange; optimal routing of trucks through a given traffic congestion pattern.

- DATA STRUCTURE

In practice, graphs are usually represented by one of two standard data structures: an adjacency list and an adjacency matrix. At a high level, both data structures are vertex-indexed arrays; this requires each vertex to have a unique integer identifier between 1 and V . In the formal sense, these integers are vertices.

- LIST OF PARTICIPANTS

By far the most common data structure for storing graphics is the adjacency list. An adjacency list is an array of lists, each of which contains the neighbors of one of the vertices (or neighbors if the graph is directed). For an undirected graph, each edge uv is stored twice, once in the neighborhood list of u and once in the neighborhood list of v ; for directed graphs, each edge uv is stored only once, in the list of neighbors ending in u . For both types of graphs, the overall space required for the adjacency list is $O(V + E)$. There are several different ways to represent these neighbor lists, but the standard implementation uses a simple linked list. The resulting data structure allows us to enumerate (outside) neighbors of a node v in $O(1 + \text{deg}(v))$ time; just scan the list of neighbors from v . Similarly, we can determine if uv is an edge in $O(1 + \text{deg}(u))$ time scanning the list of u 's neighbors. For an undirected graph, we can improve the arrival time to $O(1 + \min \{\text{deg}(u), \text{deg}(v)\})$ by traversing the neighbor lists of u and v simultaneously, stopping when we position the edge, i.e. when we fall from the end of the list

- STORING GRAPHS VIA ADJACENCY MATRICES

The other standard data structure for graphs is the adjacency matrix, first proposed by Georges Branel in. The adjacency matrix of a graph G is a $V \rightarrow V$ matrix of 0s and 1s, normally represented by a two-dimensional array $A [1 \dots V, 1 \dots V]$, where each entry indicates whether a particular edge is present in G . Specifically, for all vertices u and v : if the graph is undirected, then $A [u, v] := 1$ if and only if $uv \in E$, and if the graph is directed, then $A[u, v] := 1$ if and only if $uv \in E$.

For undirected graphs, the adjacency matrix is always symmetric, meaning $A[u, v] = A[v, u]$ for all vertices u and v , because uv and vu are just different names for the same edge, and the diagonal entries $A[u, u]$ are all zeros. For directed graphs, the adjacency matrix may or may not be symmetric, and the diagonal entries may or may not be zero. Given an adjacency matrix, we can decide in $\rightarrow (1)$ time whether two vertices are connected by an edge just by looking in the appropriate slot in the matrix. We can also list all the neighbors of a vertex in $\rightarrow (V)$ time by scanning the corresponding row (or column). This running time is optimal in the worst case, but even if a vertex has few neighbors, we still must scan the entire row to find them all. Similarly, adjacency matrices require $\rightarrow (V^2)$ space, regardless of how many edges the graph has, so they are only space-efficient for very dense graphs.

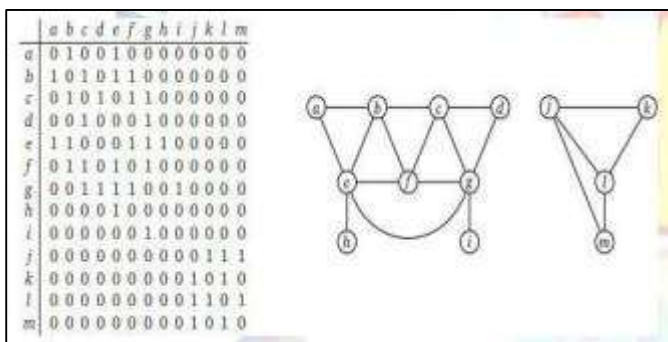


Figure 14: Adjacency Matrix and Corresponding Graph

2.1.2 Research Gaps

Intensive research is an important aspect of research in the field of pathfinding. When a new pathfinding algorithm is proposed in the literature, the performance is evaluated by empirical methods. Statistical methods are a combination of exploratory techniques and confirmatory procedures. Learning by exploring techniques are those techniques that provide visualization, summarization, and modeling of the data collected by confirmatory experiments. Empirical methods amplify our observations and help us understand the structure of our problem world. Empirical studies seek the explanation for the performance of the algorithm rather than finding the best performing algorithm. In pathfinding, many works of literatures published were not designed or evaluated empirically. There are no set guidelines for conducting experiments in pathfinding. It is crucial to have a standardized experimental setup to evaluate the performance of the algorithm empirically, to conclude reliable results because whatever we publish presently becomes the fundamentals of the future. Therefore, if a slight weakness or not reliable information gets into the mainstream it will lead to more chaos in the future. In the literature, we did not find any paper which could provide some standards or guidelines to conduct empirical studies in

pathfinding. Empirical research thoroughly examines the performance and provides experimental verification of the working of the method. The advancement in artificial intelligence in games and other fields is making the pathfinding problem more challenging as the resources are utilized in other AI 13 operations like graphics, player actions, leaving a very small space for running pathfinding search. Also, there is a pressure of developing a more advanced, fast, and optimal path planning search with limited resource utilization and minimum time frame

2.2 Research Papers and Thesis Consulted for Path Finding Visualizer

1. Robbi Rahim *et al* 2018 *J. Phys.: Conf. Ser.* 1019 012036: Shortest/Optimal path is determined using BFS algorithm on a Cartesian Field.
2. “A survey of shortest-path algorithms - ripublication.com.” [Online]. Available: https://www.ripublication.com/ijaer18/ijaerv13n9_43.pdf. [Accessed: 12-May-2022].

Shortest path was determined by a new technique called Particle Swarm Optimization for Wireless Sensor which needed less time for packets of data improving battery use.

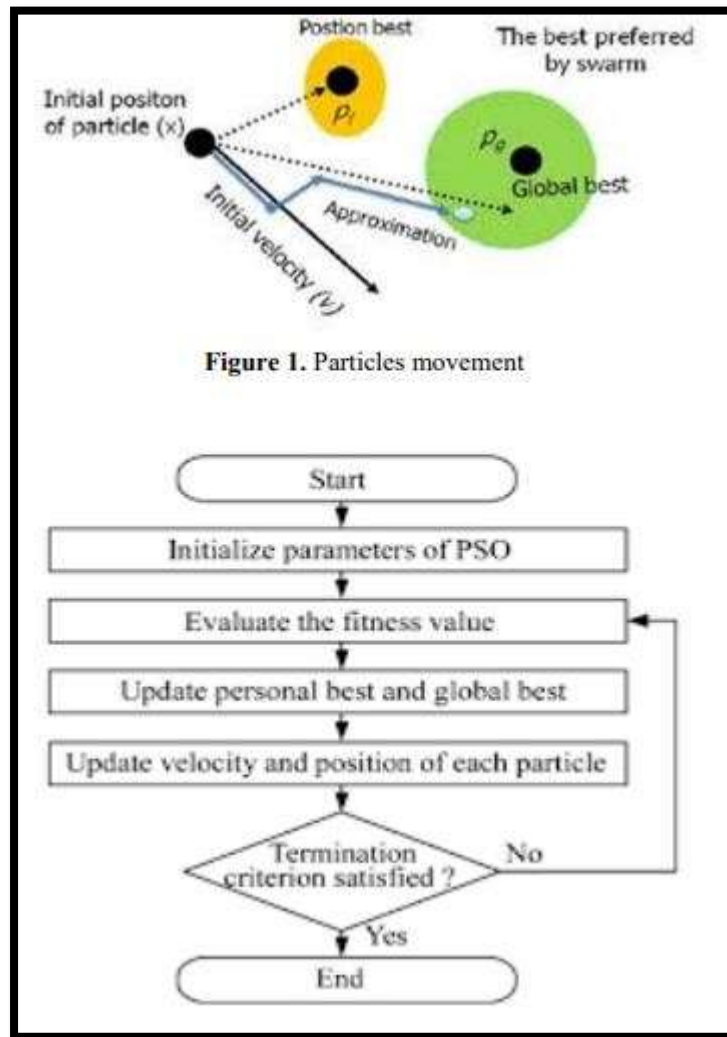


Figure 18: Particle Movement and Flowchart of PSO Algorithm

3. Sun Yigang, Fu Jie and Zhang Hongying “Research on the Shortest Path Algorithm of Vehicles Dispatch in Airport Emergency Rescue” 2011 Third International Conference on Intelligent Human-Machine Systems and Cybernetics : This paper proposed a hybrid algorithm combining DFS and BFS in the area of vehicle dispatch , saving rescue time and minimizing losses in case of accidents..
4. Kairanbay, Magzhan & Mat Jani, Hajar. (2013). A Review and Evaluations of Shortest Path Algorithms. International Journal of Scientific & Technology Research. 2. 99-104.: This paper’s main objective is to test the Dijkstra’s Algorithm, Floyd-Warshall Algorithm, Bellman-Ford Algorithm, and Genetic Algorithm (GA) in solving the shortest path problem in case of routing problem in computer networks.
5. Masilo Mapaila “EFFICIENT PATH FINDING FOR TILEBASED 2D GAMES”: Different educated guesses/heuristics used in an A* (A Star) algorithm were explored here to target efficient path finding within tile-based games. Various optimization techniques were listed which made real time path finding faster and less memory intensive.

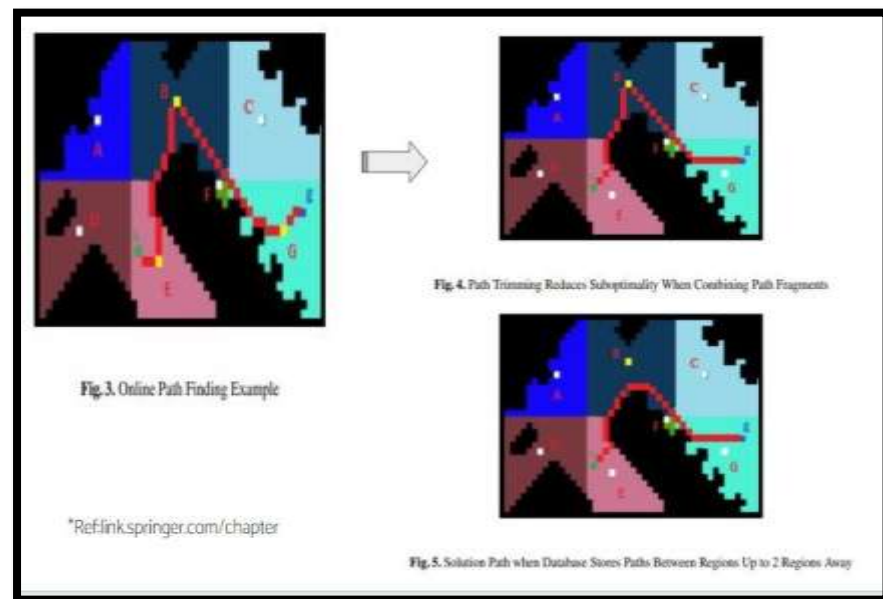


Figure 15: Different Optimizations in PathFinding

6. Yap, P. (2002). Grid-Based Pathfinding. In: Cohen, R., Spencer, B. (eds) Advances in Artificial Intelligence. Canadian AI 2002. Lecture Notes in Computer Science (), vol 2338. Springer, Berlin, Heidelberg. <https://doi.org/10.1007/3-540-47922-8> 4: Applications like network traffic, robot planning, military simulations, and computer games were explored in this paper to find the most optimal path in each case. Different grid visualizations and search algorithm were compared on A* and IDA* Search particularly.

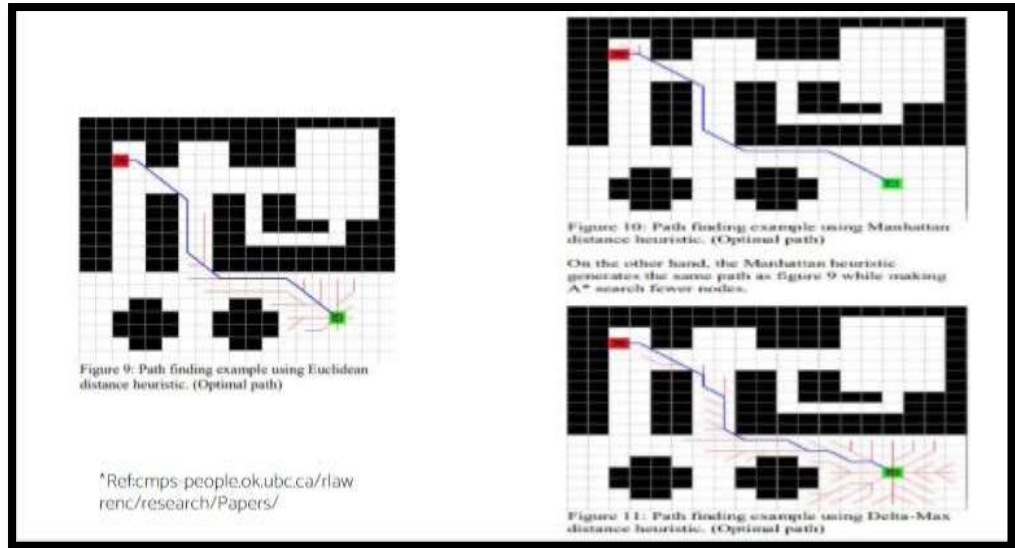


Figure 16: PathFinding with Different Heuristics

7. Lee W., Lawrence R. (2013) Fast Grid-Based Path Finding for Video Games. In: Zaïane O.R., Zilles S. (eds) Advances in Artificial Intelligence. Canadian AI 2013. Lecture Notes in Computer Science, vol 7884. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-38457-8_9: Video games and virtual agents interacted in grid-based path finding with map sizes and count of agents changing in real time. Evaluated against benchmarks maps from DBA* Search of Dragon Age game. On analysis 3 % more efficient than PRA* implementation of the benchmark.
8. Red Blob Games “Grid PathFinding Optimizations” <https://www.redblobgames.com/pathfinding/grids/algorithms.html>: Presents different optimizations in A* Search algorithm to improve its efficiency like Changing the map representation to a grid structure , using hexes, tiles , different traversal speed optimizations etc.

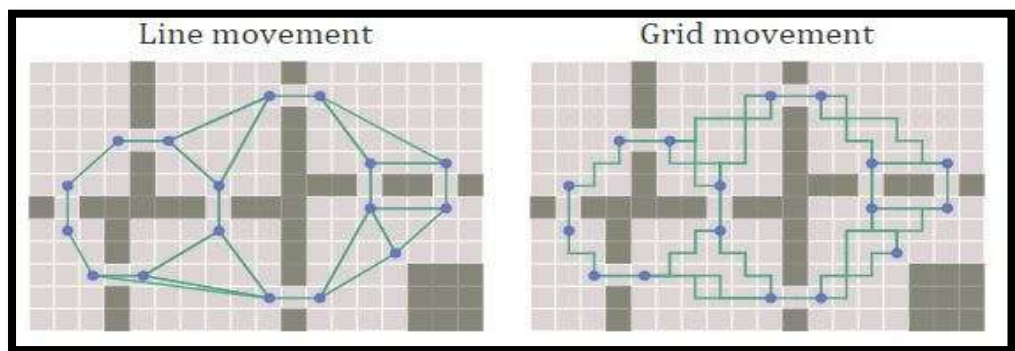


Figure 17: Line vs Grid Movement

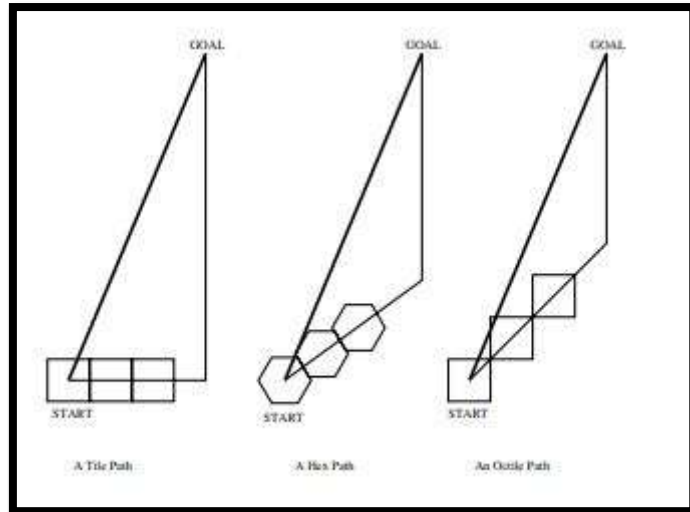


Figure 19: Optimal Paths in Different Grids

2.3 A Critical Analysis of Path finding Research Papers

Research papers are an important part of current developments in pathfinding area, as it provides insight into the work done in past and future expectations. It is necessary to critically review the research papers published in past to make future research more valuable and reliable by learning from their mistakes and weaknesses. In this thesis nearly 150 research papers are critically reviewed during literature study and throughout our thesis work to find out the pitfalls and problem areas while conducting research. Critical review of some papers is presented in this section. Kai Li et al. proposed a boundary iterative-deepening depth-first search (BIDDFS) algorithm which repeats its search from the saved boundary location, minimizing the search redundancy in most of the iterative search algorithms (repeats its search from starting point each time). The experimental results only show the time taken and threshold, in which Dijkstra beats the time of BIDDFS, but it does not provide clear evidence like the memory usage. Therefore, the paper fails to provide causal explanation of their algorithms performance which might result in mislead 16 interpretations of data. Also, the paper gave no evidence that their algorithm had been tried on more than one set of maps (randomly generated square maps), no real-world maps were used to test the algorithm's performance. Another problem was that BIDDFS had the same threshold as of IDDFS in all the cases and takes more runtime for lower obstacle density. Threshold was described as memory efficiency, but the paper did not provide any memory comparisons, which was the claim, that proposed algorithm consumes less memory. The research paper written by Yngvi et al. proposed two new effective heuristics for A*. The first one is, the dead – end heuristic, that reduces the search area from the map which is irrelevant to the current path query and thus claimed to be more effective than general octile heuristic. The second heuristic, called gateways heuristic, used the decomposed map from the previous heuristic, then consider the boundaries of the omitted areas as gateways to pre-process the path from one gateway to all other gateways and thus, claimed to better estimate the path cost. Now, the way this paper was presented has three main areas where it lacks empirically or did not provide enough information to the readers. Firstly, it uses one demo map and nine game maps, but did not provide any information regarding the range of map size, obstacle density and distribution of obstacles. Secondly, the author did admit that the proposed heuristics use extra memory for pre – calculations but did not give any range

or number for the memory usage. Thirdly, the author claims that heuristics take less time for the final pathfinding but did not provide any data for the time taken by these heuristics for pre-processing of the map decomposition and distance between gateways. Whereas, in octile heuristic neither pre-calculation nor decomposing the map is required. So, from reader's perspective this paper did not answer all the questions arising in reader's mind.

The Breadth First Search algorithm applied to determine the path from the original location to the destination location in the Cartesian field yields several alternative solutions based on the tests performed. Thus, the Breadth First Search algorithm applied to the Cartesian field can produce some optimal solution (shortest path) and solution which is non-optimal.

The tool for path finding for 2D environments. At present, has limitations and we see the work presented in the papers as a step towards the development of a complete tool. The current work focused on static and dynamic environments, and we have worked with both fixed and dynamic cost environments. Most of the research in academic AI has been focused on robotics and very little has been done towards their application in tile-based games. This study bridges that gap and with further research, it would be possible to develop a complete tool that would be useful in academia and would certainly benefit the game industry

Hex grids provide a better topological representation than tile or octile grids. Moreover, for memory constrained domains that necessitate IDA*, the hex grid is the optimal grid choice. Finally, the implementation of hex grids is made easier with tex grids.

Grid-based path finding must minimize response time and memory usage. DBA* has lower suboptimality than other abstraction-based approaches with a faster response time. It represents a quality balance and integration of the best features of previous algorithms. Future work includes defining techniques for efficiently updating pre-computed information to reflect grid changes.

2.4 Path finding Research Papers: A Summary

Thus, on comparing different research papers I concluded that most papers had poor experimental setup as the simulation environments used to test various algorithms were almost always simple and could not incorporate complex path scenarios. There was insufficient data collection and analysis and poor data visualization in these setups. Only one type of representation ex. Maps/Grids were used which were either prebuilt game maps or random maps. They were used due to easy availability and easy generation by random maze generation algorithm; thus, I intend to use recursive and diagonal maps/grids in my project to solve this issue. It was difficult to compare results cross papers due to the same reason and thus not enough evidence could be generated to ensure that the performance that the authors claimed in their setup was completely true to its use. In papers where the experiment setup was well planned, they were mostly penned by experienced researchers. The statistical methods used were mean, median and standard deviation. Other papers were completely theory based which particularly used heuristics derived from mathematic proofs to supposedly improve the algorithm but had no experimental evidence as such.

Chapter 3

System Development

3.1 Model/Proposed System

The approach proposed is A* algorithm with heuristic search, will likely locate the shortest path solution in a totally quick amount of time and minimal distance. A* set of rules, a type of knowledgeable search, is widely used for finding the shortest route, because the place of beginning and finishing point is considered in advance. The A* algorithm is a refinement of the shortest course set of rules that directs the quest toward the preferred aim. The preferred purpose of heuristic set of rules is to discover a most desirable answer where the time or resources are constrained. As illustrated in fig20[C]. This is later as compared with Dijkstra set of rules which is easy and wonderful technique for route planning. Dijkstra's set of rules chooses one with the minimal value until located the purpose, however the search isn't always over because it calculates all possible paths from beginning node to the purpose, then choose the excellent answer by means of comparing which way had the minimum distance.



Figure 20.1: Shortest Path using A* Search

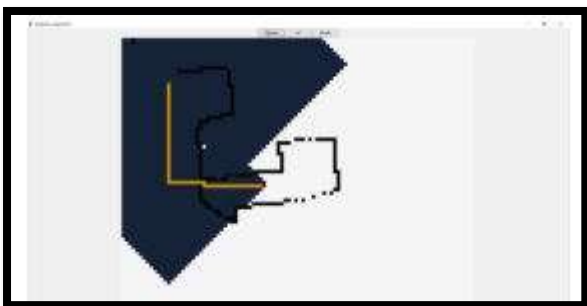


Figure 20.2: Shortest Path using Dijkstra Algorithm

3.1.1 SOFTWARE DEPLOYED IN VISUALIZATION

A dynamic visualization system for algorithm animation should satisfy the following

conditions:

- Animation speed - the system should be able to present a good dynamic visualization.
- Programming effort - it should be easy to write a visualization code.
- Widespread access - it must be easy to run a visualization code without (much of) downloading and installing software.

Of course, the first condition is the principal one: in many cases, a visualization involves a continuous transformation of the displayed picture, and it might be computationally very demanding to deliver at least 20-25 frames per second to guarantee a smooth animation. A failure in this point would make the system useless.

Programming languages can be divided into three classes:

- Compiled languages - a code written by a programmer is compiled into the machine language and runs at the maximum possible speed. Examples are the languages C and C++ that also offer libraries of graphical functions (e.g., graphics.h).
- Semi-compiled languages - The code written by programmer is transformed into a simpler code that is then interpreted by a special software. An example is Java - the intermediate code is interpreted by JVM program (Java Virtual Machine)
- Interpreted languages - the runtime system reads human written program instructions in runtime and interpret them. An example is JavaScript, see below. There are big differences in the speed among the above classes. While one simple instruction is often executed in just several machine clock tacts, if the same instruction is interpreted, the software must first read and parse the corresponding code, use tables to find the equivalent machine instruction, and only after that the instruction is executed. Interpreted languages are often several order of magnitude slower than compiled languages.
- Typical animations that can be found in the web are quite simple and computationally almost insignificant. Consequently, practically any system that allows dynamic animation can be used, preference is given to simple scripting languages

3.1.2 Tools and Platforms Deployed

- React.js: React is a open source JavaScript library that is mostly used to build single pages user interfaces application. it is used to handle the view for the web and mobile apps. It was important as it gives us the re usability of the UI component. This tool helped me have that base interface for the tool.
- CSS:CSS stands for cascading Style Sheets which describes how HTML elements are to be displayed on the screen, pages or other media. It is very important as it helps with the layout control. CSS helped US make the middle grid where our starting and end node are displayed and with display of the wall between them.
- Visual Studio Code: Visual studio code is a source code editor developed by Microsoft for windows, Linux, and macOS
- Node.js Library for JS
- JS Meter to benchmark visualization parameters

Design:

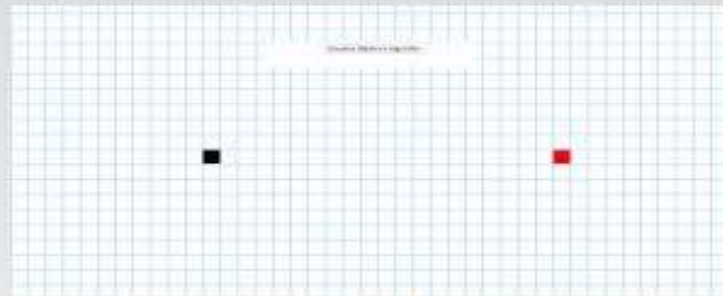


Image1: The image shows the starting page of the app the black dot represent the starting node and the red dot is the end node each grid represent a node. In the center up is the button that start the application.

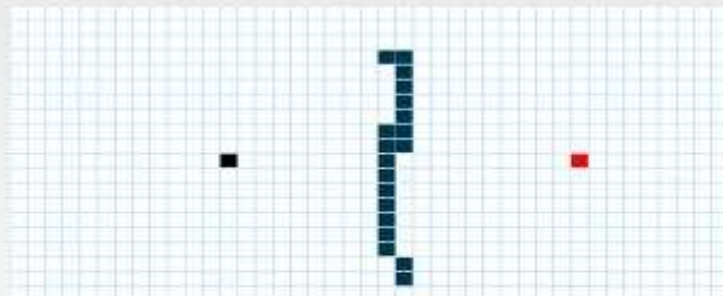


Image2: The second image shows the in addition to the starting node and the end node the wall that the user can add in between. The application has to find the shortest path avoiding the wall.

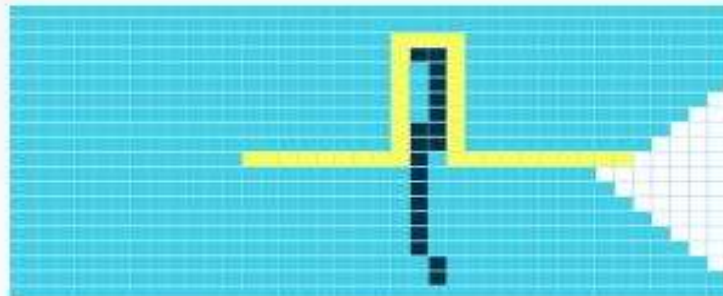


Image3: The following image shows the yellow ligne that represent that path taken by the path from the starting to end node avoiding the wall in between.

Figure 21: Design of PathFinding Visualizer

The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with files like `manifest.json`, `index.html`, `logo192.png`, `logo512.png`, `robots.txt`, and a folder named `Algorithms` containing `dijkstra.js`. The code editor shows the implementation of a Dijkstra's algorithm in JavaScript, including functions for `getUnvisitedNeighbors`, `sortNodesByDistance`, and `updateUnvisitedNeighbors`.

```

my-app > src > Algorithms > js dijkstra.js > getUnvisitedNeighbors
export function dijkstra(grid, startNode, finishNode) {
  const visitedNodesInOrder = [];
  startNode.distance = 0;
  const unvisitedNodes = getAllNodes(grid);
  while (!unvisitedNodes.length) {
    sortNodesByDistance(unvisitedNodes);
    const closestNode = unvisitedNodes.shift();
    // If we encounter a wall, we skip it.
    if (closestNode.isWall) continue;
    // If the closest node is at a distance of infinity,
    // we must be trapped and should therefore stop.
    if (closestNode.distance === Infinity) return visitedNodesInOrder;
    closestNode.isVisited = true;
    visitedNodesInOrder.push(closestNode);
    if (closestNode === finishNode) return visitedNodesInOrder;
    updateUnvisitedNeighbors(closestNode, grid);
  }
}

function sortNodesByDistance(unvisitedNodes) {
  unvisitedNodes.sort((nodeA, nodeB) => nodeA.distance - nodeB.distance);
}

function updateUnvisitedNeighbors(node, grid) {
  const unvisitedNeighbors = getUnvisitedNeighbors(node, grid);
  for (const neighbor of unvisitedNeighbors) {
    neighbor.distance = node.distance + 1;
    neighbor.previousNode = node;
  }
}

function getUnvisitedNeighbors(node, grid) {
  const neighbors = [];
  const {col, row} = node;
  if (row > 0) neighbors.push(grid[row - 1][col]);
  if (row < grid.length - 1) neighbors.push(grid[row + 1][col]);
  if (col > 0) neighbors.push(grid[row][col - 1]);
  if (col < grid[0].length - 1) neighbors.push(grid[row][col + 1]);
  return neighbors.filter(neighbor => !neighbor.isVisited);
}

```

Figure 22: Basic File Structure of the App

The model and prototype of the project are as follows:

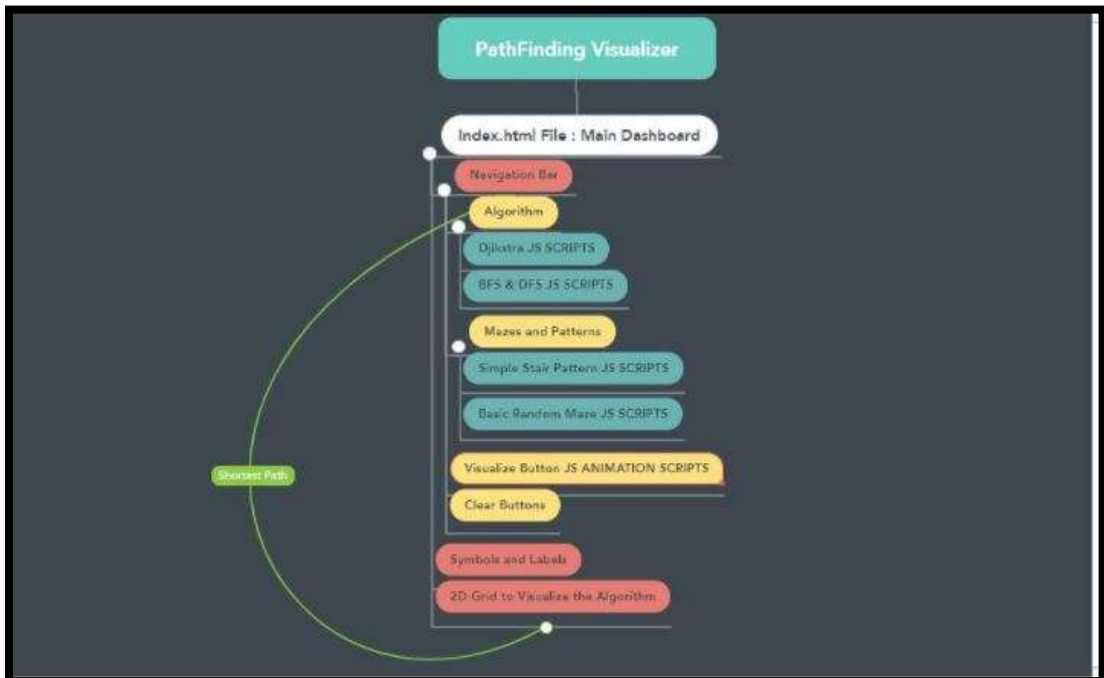


Figure 23: Workflow of the Project

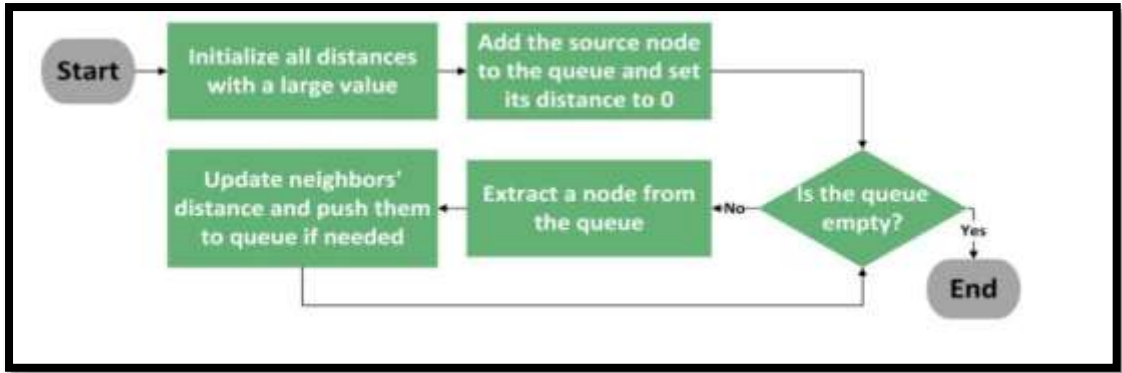


Figure 59: GENERAL SINGLE SOURCE SHORTEST PATH ALGORITHMS

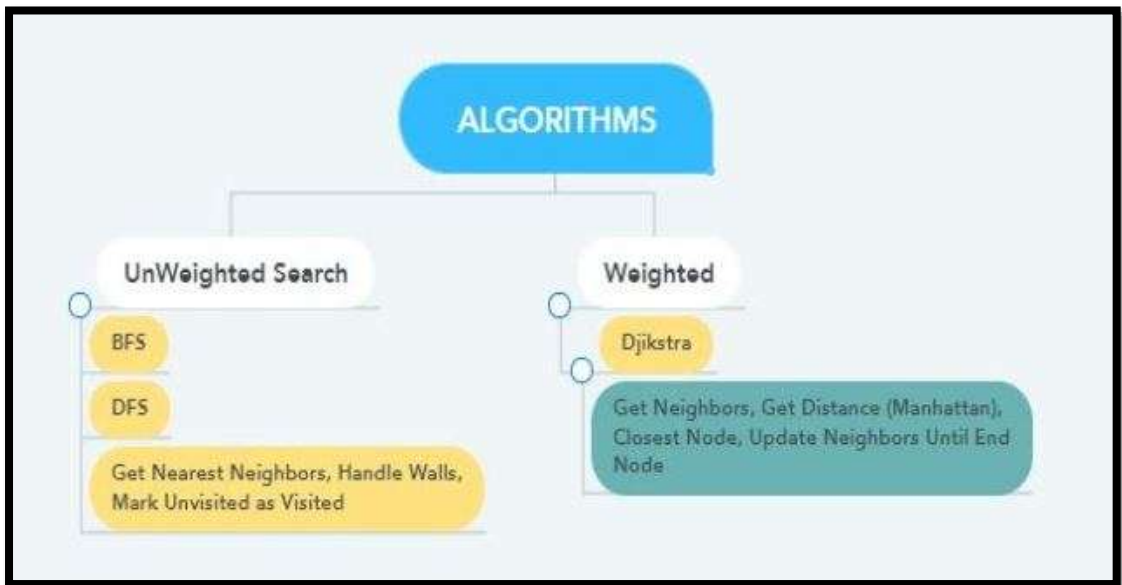


Figure 24: Different types of Algorithms used (Unweighted and Weighted)

3.2 Types of Algorithms

In all the general single source shortest path algorithms, a common methodology is followed which consists of initializing all distances with a large value (preferably, ∞), adding the source node to the data structure in use. Traversing while it is not empty, to extract the front node of the data structure and visit all its un-visited neighbours one by one until the end node is encountered. I have utilized an adjacency lists data structure, consisting of an array consisting of nodes with node storing the list of their immediate/potential neighbours. This is chosen as it was simple to formulate, implement and test effectively. Representation in JSON works as:

```
[
  {
    "x":0, "y":0,
    "neighbors":[1,3]
  }, {
    "x":0, "y":1,
    "neighbors":[0]
  }
  ...
]
```

Array[i] i.e., is set as unique node id, which used to refer to the neighbours.
Result Obtained:

We got the series of nodes of path between two nodes, and we denoted it by an ordered (by step order) array which we further passed to visualization functions.

```
[0, 3, 19, 9, ...]
```

Starting from the node 0, we go to node 3, then to node 19 etc. until the destination node.

1. **Breadth First Search:**BFS explores the node breadth wise to find all possible combinations.

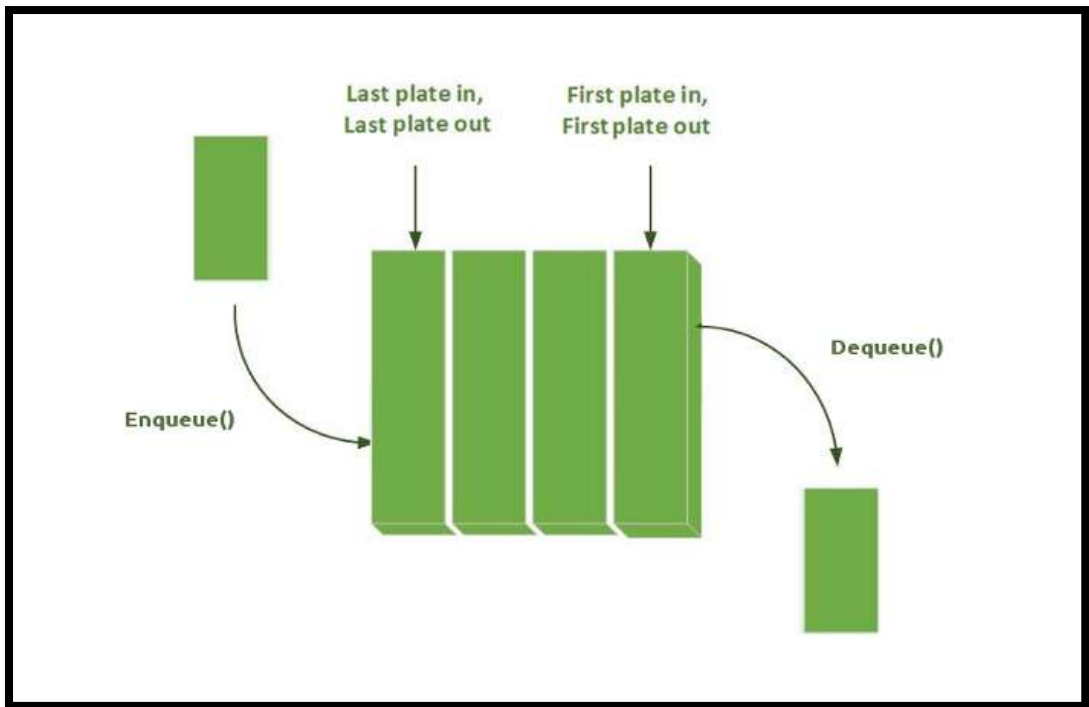


Figure 25: BFS implementation as a Queue (FIFO)

Pseudo Code

```

BFS(Maze m , Node start_node, Node end_node)
{
//initialize the queue with start_node
start_node.visit=True
Queue q(start_node)
//while there is a node entry in the queue
While(!q.empty())
{
//Front node operation
Node current_node=q.pop();
//If the end target is reached, we terminate
If(current_node==end)
Break
//Else we traverse the neighbors breadth wise
Auto Neighbors=current_node.get_unvisited_neighbours ()
for( auto i=0;i<Neighbors.size();++i)
{
Neighbors[i].visited=True
Neighbors[i].parent=current_node
q.push(Neighbors[i])
}
}

```

```

}
//We get a path traversing backwards from destination node to source node
}
//If end has no parent , continue traversing for end

```



Figure 26: BFS Visualization

```

function unweightedSearchAlgorithm(nodes, start, target, nodesToAnimate, boardArray, name) {
  if (!start || !target || start === target) {
    return false;
  }
  let structure = [nodes[start]];
  let exploredNodes = {start: true};
  while (structure.length) {
    let currentNode = name === "bfs" ? structure.shift() : structure.pop();
    nodesToAnimate.push(currentNode);
    if (name === "dfs") exploredNodes[currentNode.id] = true;
    currentNode.status = "visited";
    if (currentNode.id === target) {
      return "success";
    }
    let currentNeighbors = getNeighbors(currentNode.id, nodes, boardArray, name);
    currentNeighbors.forEach(neighbor => {
      if (!exploredNodes[neighbor]) {
        if (name === "bfs") exploredNodes[neighbor] = true;
        nodes[neighbor].previousNode = currentNode.id;
        structure.push(nodes[neighbor]);
      }
    });
  }
  return false;
}

```

Figure 27: BFS and DFS Code Snippet

2. Depth First Search (DFS)

- **Depth First Search (DFS) is a fundamental graph traversal algorithm.** The DFS can be used to generate a topological ordering, to generate mazes (cf. DFS maze generators), to traverse trees in specific order, to build decision trees, to discover a solution path with hierarchical choices, to detect a cycle in a graph, however the DFS does not guarantee the shortest path.

Then, this is not a very good algorithm if the unique purpose is to do a simple pathfinding. Depth First Search Traverses by exploring as far as possible down each path before going back. It is the reason why you may also find this algorithm under the name of **Backtracking**. Furthermore, this property allows the algorithm to be implemented succinctly in both iterative and recursive forms. If we consider a tree (which is a simplified graph), the DFS will proceed as follows:

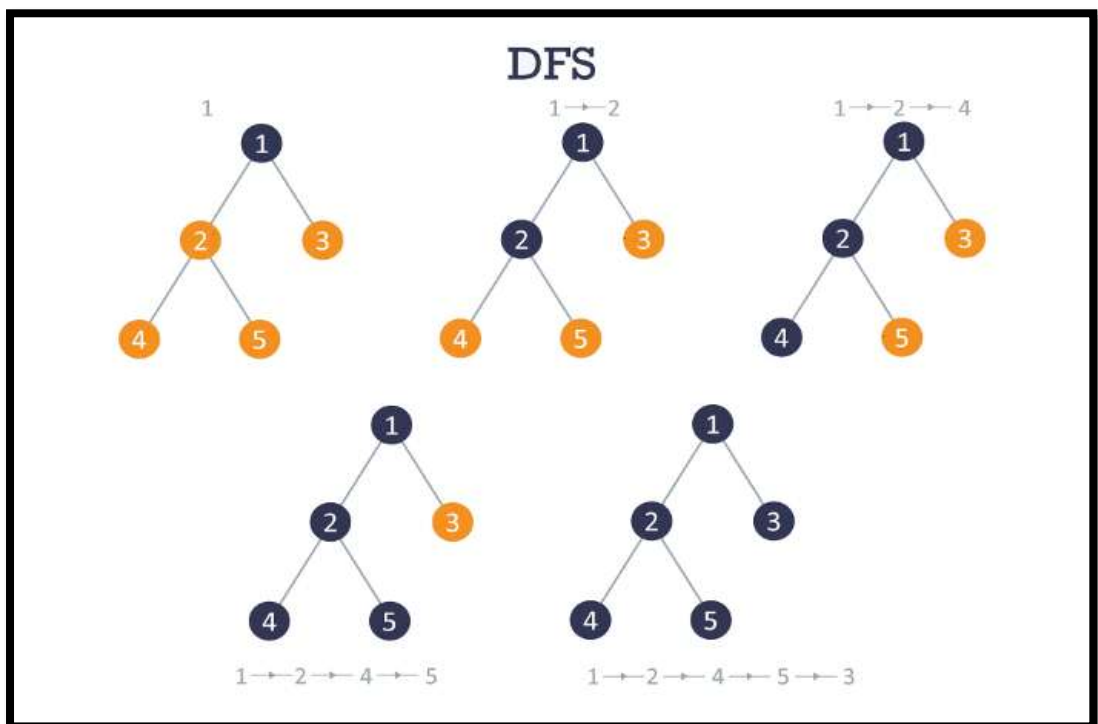


Figure 28: Basic flow of DFS

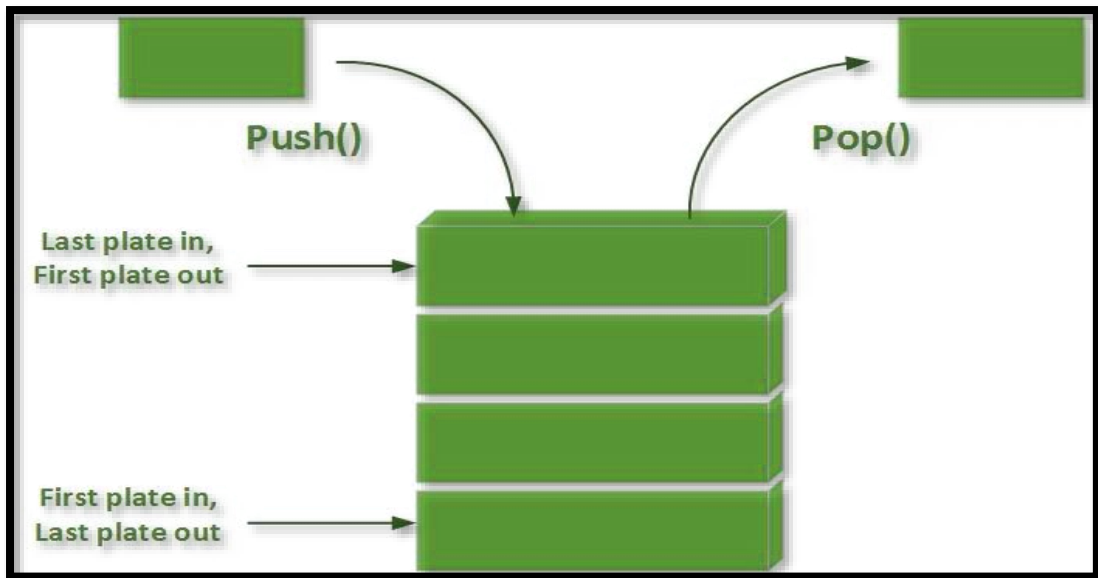


Figure 29: DFS implementation as a Stack (LIFO)

Pseudo Code

```

DFS(Maze m, Node start_node, Node end_node)
{
//Inserting starting node in a stack
Start_node.visit=True
Stack s(start_node)
//while the stack is not empty
While(!s.empty())
{
//operating on the topmost node
Node current_node=s.pop()
//if target is reached stop
If(current_node==end)
Break
//else take unvisited neighbors in order, set parents , mark as visited and add to stack
Auto Neighbors=current_node.GetUnvisitedNeighbors()
for(auto i=0;i<Neighbors.size(),++i)
{
Neighbors[i].visited=True
Neighbors[i].parent=current_node
s.push(Neighbors[i])
}
}
//Reach the end traversing neighbor by neighbor , backtracking to previous neighbor
to a different path until end target is reached

```

```
//If end has no parent , keep looking for the end
}
```

Recursive Variant of DFS

The recursion of the DFS algorithm stems from the fact that we don't finish checking a "parent" node until we reach a dead end, and inevitably pop off one of the "parent" node's children from the top of the call stack.

Algorithm 2 Pseudo-code for (Recursive) DFS

```
function DFS-REC( $N, C, u, \text{Discovered}$ )
    Discovered.add( $u$ )
    if  $C(u)$  then return  $u$ 
    for  $v$  in  $N(u)$  do
        if not  $v \in \text{Discovered}$  then
            DFS( $N, C, v, \text{Discovered}$ )
    Discovered  $\leftarrow$  emptySet
    DFS-REC( $N, C, \text{start-node}$ )
```

```
Bool Recursive_DFS(Maze m, Node start_node, Node end_node)
{
//If end target is reached terminate
If(start_node==end_node)
Return True
//traverse unvisited neighbors in order
Start_node.visit=True
Auto Neighbors=start_node.GetUnvisitedNeighbors()
for(auto i=0;i<Neighbors.size();++i)
{
Neighbors[i].parent=start_node
If(Recursive_DFS(m, Neighbors[i], end_node) return True
}
Return False
}}
```




Figure 30: DFS Visualization

3. Dijkstra Algorithm

- Dijkstra's algorithm finds the shortest path from a root node to every other node (until the target is reached). Dijkstra is one of the most useful graph algorithms; furthermore, it can easily be modified to solve many different problems. Dijkstra's algorithm eliminates useless traversal using heuristics to find the optimal paths.
- Dijkstra's algorithm uses priority queue (or heap) as the required operations (extract minimum and decrease key) match with specialty this data structure. Here the priority is defined by D (the distance between a node and the root). Higher priorities are given for nodes with lower distance D .

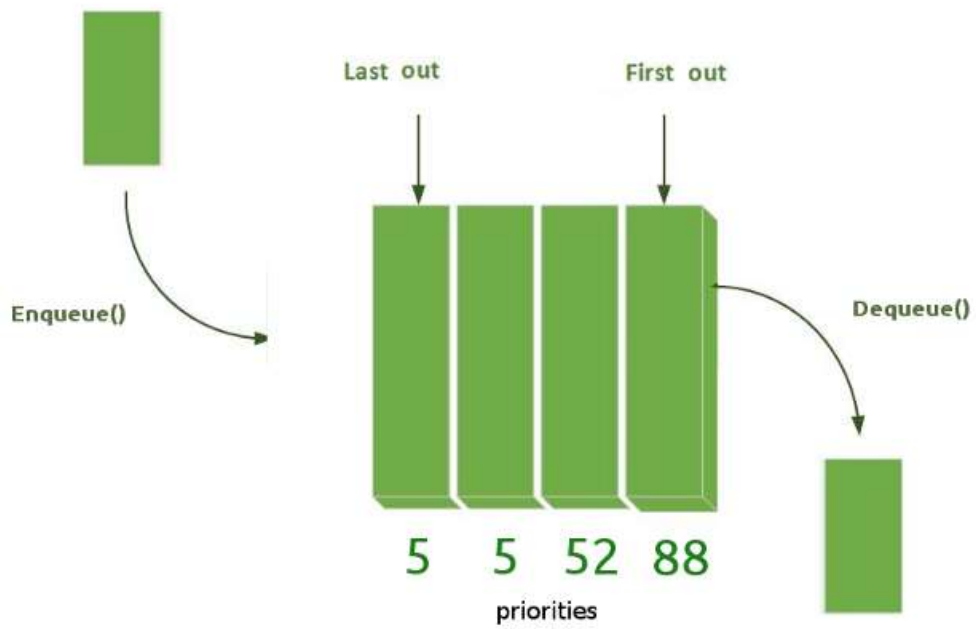


Figure 31: Dijkstra Implementation as a Priority Queue

We can visualize it as regular attraction queue in a stall /fair with some people having access to shortcut heading to different checkpoints like VIP Access.



Figure 32.1: Visualizing Dijkstra Algorithm

A priority queue should have the following methods of Enqueue and Dequeue Operations:

```
// Add new element at the end of the queue
void enqueue (const T& data, const int priority) // Remove element from the beginning of the
queue
void dequeue () // Change the priority of the element in the queue.
void changePriority(const T& item, const int priority);
```

Pseudo Code

```

Dijkstra(Maze maze, Node start, Node end) {
    priority_queue queue();

    // Init all distances with infinity
    for (auto&& node : maze.nodes) node.distance = Infinity;

    // Distance to the root itself is zero
    start.distance = 0;

    // Init queue with the root node
    queue.add(start, 0);

    // Iterate over the priority queue until it is empty.
    while (!queue.isEmpty()) {
        curNode = queue.dequeue(); // Fetch next closest node
        curNode.discovered = true; // Mark as discovered

        // Iterate over unvisited neighbors
        for (auto&& neighbor : curNode.GetUnvisitedNeighbors())
        {
            // Update minimal distance to neighbor
            // Note: distance between to adjacent node is constant
            const minDistance =
                Math.min(neighbor.distance, curNode.distance + 1);
            if (minDistance != neighbor.distance) {
                neighbor.distance = minDistance;
                neighbor.parent = curNode;

                // Change queue priority of the neighbor
                if (queue.has(neighbor))
                    queue.setPriority(neighbor, minDistance);
            }

            // Add neighbor to the queue for further visiting.
            if (!queue.has(neighbor))
                queue.add(neighbor, neighbor.distance);
        }
    }

    // from here we walk back from the end using the parent
    // If end does not have a parent, it means that path has not been found.
}

```

The Dijkstra algorithm does not support -ve weights. The algorithm assumes that adding relationships to the path will never make the path shorter, more stable, and less likely to be broken, however, to manage negative weights we use bellman ford algorithm.

```
function weightedSearchAlgorithm(nodes, start, target, nodesToAnimate, boardArray, name, heuristic) {
  if (name === "astar") return astar(nodes, start, target, nodesToAnimate, boardArray, name)
  if (!start || !target || start === target) {
    return false;
  }
  nodes[start].distance = 0;
  nodes[start].direction = "right";
  let unvisitedNodes = Object.keys(nodes);
  while (unvisitedNodes.length) {
    let currentNode = closestNode(nodes, unvisitedNodes);
    while (currentNode.status === "wall" && unvisitedNodes.length) {
      currentNode = closestNode(nodes, unvisitedNodes);
    }
    if (currentNode.distance === Infinity) {
      return false;
    }
    nodesToAnimate.push(currentNode);
    currentNode.status = "visited";
    if (currentNode.id === target) return "success!";
    if (name === "CLA" || name === "greedy") {
      updateNeighbors(nodes, currentNode, boardArray, target, name, start, heuristic);
    } else if (name === "dijkstra") {
      updateNeighbors(nodes, currentNode, boardArray);
    }
  }
}
```

Figure 32.2: Dijkstra Code Snippet

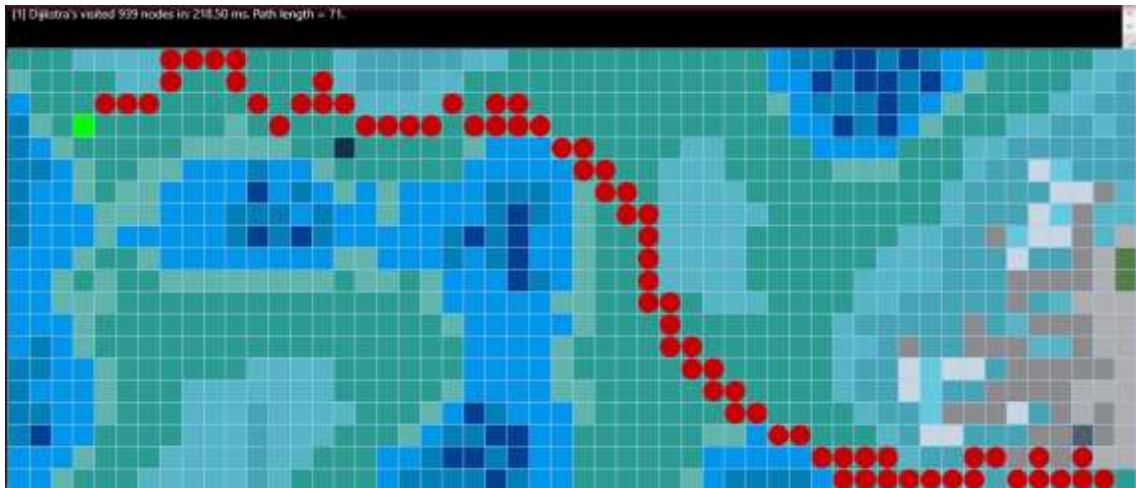


Figure 33: Dijkstra Visualization

4. * (A-Star)

The A* (A-Star) algorithm improves on Dijkstra's by finding the shortest path more quickly. It was invented by Peter Hart, Nils Nilsson, and Bertram Raphael (three American computer scientists) and described in their paper "A Formal Basis for the Heuristic Determination of Minimum Cost Paths" in 1968.

This distance is ideal for our mazes that allow 4-way movement (up, down, left, right).
$$h(n)=|goal.x-root.x|+|goal.y-root.y|$$

Here are the distance maps computed (Manhattan, Dijkstra and A*):

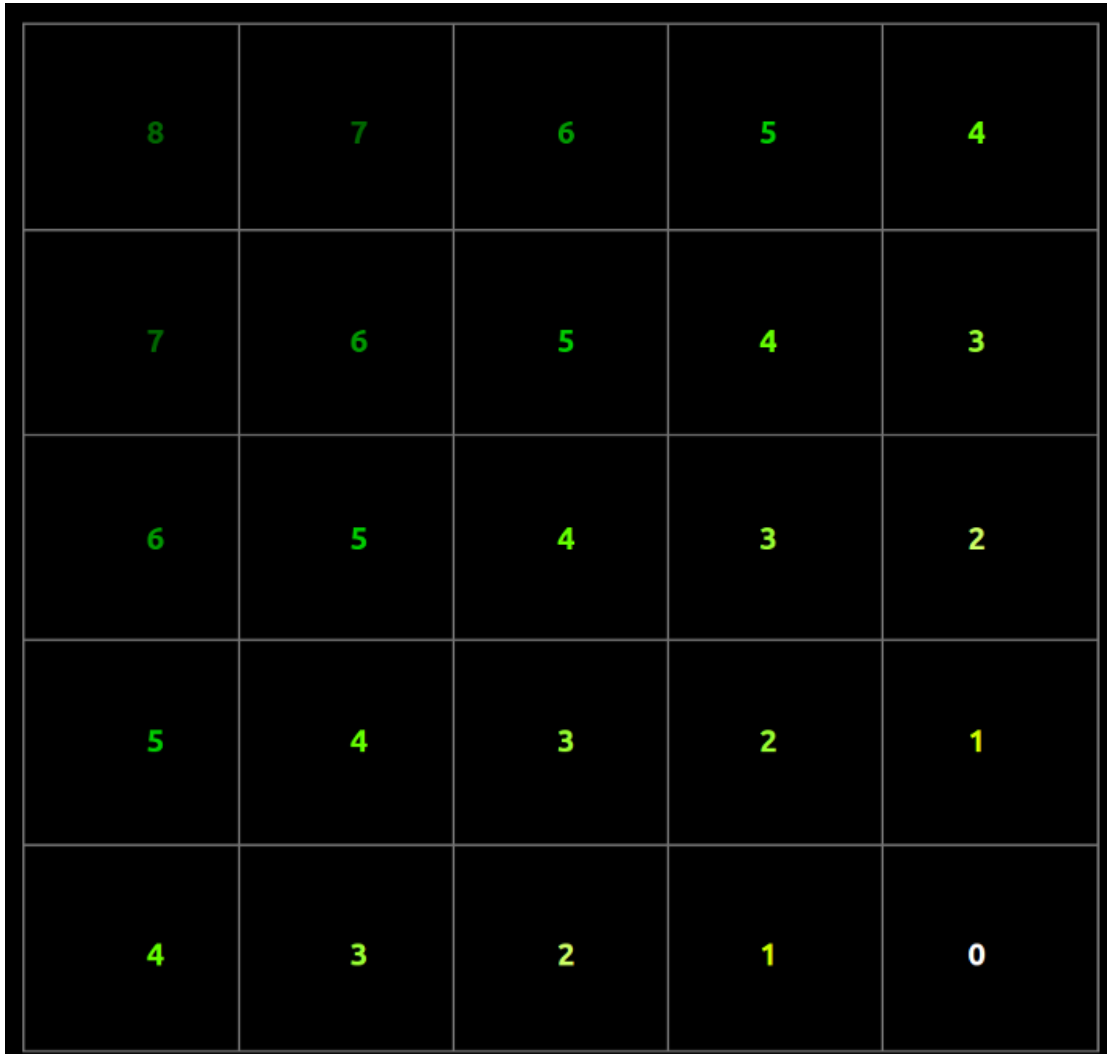


Figure 35: Manhattan distance map (from right bottom destination): $h(n)$

It is possible to reinforce Manhattan heuristic by using a factor function. For example: $h(n)$

Pseudo Code

```
A* Search( Maze m, Node start_node, Node end_node)
{
Priority_queue pq()
//initialize all distances with inf
```

```

For( auto && Node: m Nodes){
Node.distance=inf
Node.root_distance=inf
//Using Manhattan heuristic
Node.manhattan_distance=2*(Math.abs(end_node.x-node.x)+(Math.abs(node_end.y-
node.y))
}
//setting distance to root as zero
Start_node.root_distance=0
Pq.add(start_node,0)
//Traverse until priority queue is empty
While(!pq.empty())
{
Current_node=pq.dequeue()
Current_node.discovered=True
//fetch next closet node and mark it as found
//visiting the unvisited neighbors
For( auto&& Neighbor : current_node.GetUnvisitedNeighbors())
{
//Update the minimal root distance
Neighbor.root_distance=Math.min(Neighbor.root_distance,current_node.root_distanc
e+1)
Const minimum_distance=Math.min(Neighbor.distance,
Neighbor.root_distance+Neighbor.manhattan_distance)
If(minimum_distance !=Neighbor_distance)
{
Neighbor_distance=minimum_distance
Neighbor.parent=current_node
//change the priority in queue of the neighbor
If(pq.has(Neighbor))
Pq.setPriority(Neighbor,minimum_distance)
}
If(!pq.has(Neighbor))
{
Pq.add(Neighbor,Neighbor.distance)}
}
}
//Trace path beginning from destination to root node
}
// If end doesn't have a parent continue traversing for the end

```




Figure 36: A*Algorithm Visualization

5) D* Search: A variant of A* Search

D* is a dynamic variant of the A* search algorithm optimized for partially obscured environments. It was first described by Anthony Stentz in 1994. D* maintains the same three values as A*: the cost-to-current-node value, the heuristic- 10 to-end estimate, and the cost-plus-heuristic value (Stentz refers to these as c , h , and k , respectively). The value for c is calculated the same way as the value for $g(x)$ in the A* algorithm, and h is calculated the same way as A*'s $h(x)$ value. k is functionally equivalent to A*'s $f(x)$. D* also tags nodes with a place on an open and closed list, like A*, but uses an additional tag of “new” for a node that has not yet been added to the open list. Unlike A* and Dijkstra’s Algorithm, however, the algorithm starts at the node representing the goal, and works toward the node at which the agent starts. The goal node is first tagged as “open,” and values for c , h , and k are calculated for each neighbor node, and the node with the lowest k value is chosen as the next node. The process is repeated until the starting node is tagged as “closed” or no next node can be found. D* then uses the path found as a starting path and runs it until an error is found (i.e., an obstacle has moved into the agent’s path, or the next node turns out to be a dead end). In the event, the last node the agent traversed is set as a starting node and D* recalculated the h , c , and k values for all nodes examined between the new start and the goal. These values are compared to the previously computed values. If the k value for a node is lower than its old k value, then the node is placed in a “lower” state. If it is higher, the node is placed in a “raise” state. Nodes in the “lower” state are given higher priority in subsequent calculation loops. The entire process repeats until the starting node is tagged as “closed” or no next node can be

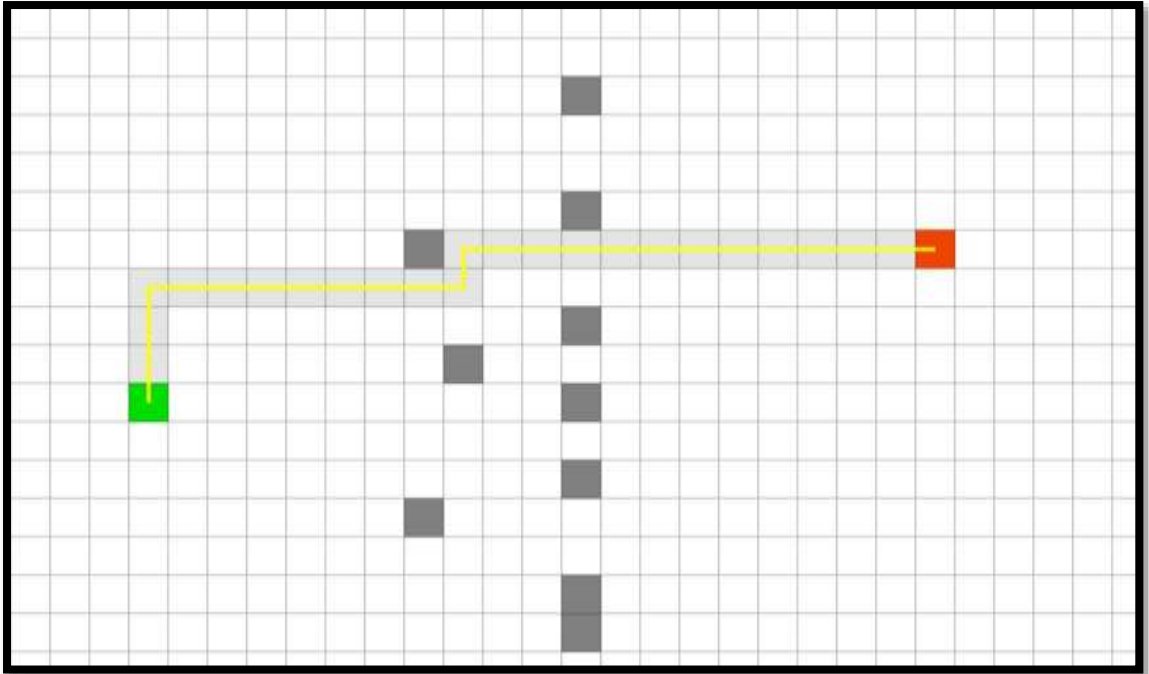
```

Pseudocode:
procedure D*(graph, start, target)
/* Path finding phase */
num_visits = 0
for each vertex v in graph
    c[v] = infinity
    h[v] = abs(start_x - v_x) + abs(start_z - v_z)
    k[v] = c[v] + h[v]
    tag[v] = open
    next[v] = undefined
end for
c[target] = 0
h[target] = abs(start_x - target_x) + abs(start_z - target_z)
k[target] = c[start] + h[start]
Q = the set of all nodes in graph
while Q is not empty
    x = vertex in Q with smallest k []
    if k[x] == infinity
        break
    remove x from Q
    tag[x] = closed
end while
if x is not target
    for each neighbor v of x
        if tag[v] is not closed
            alt = dist_between(x, v) + c[v] + h[v]
            if alt < k[v]
                k[v] = alt
                next[v] = x
            end if
        end if
    end for
end if
/* Path reconstruction phase */
s = empty sequence
x = start
while next[x] is defined
    insert x at the end of s
    x = next[x]
end while

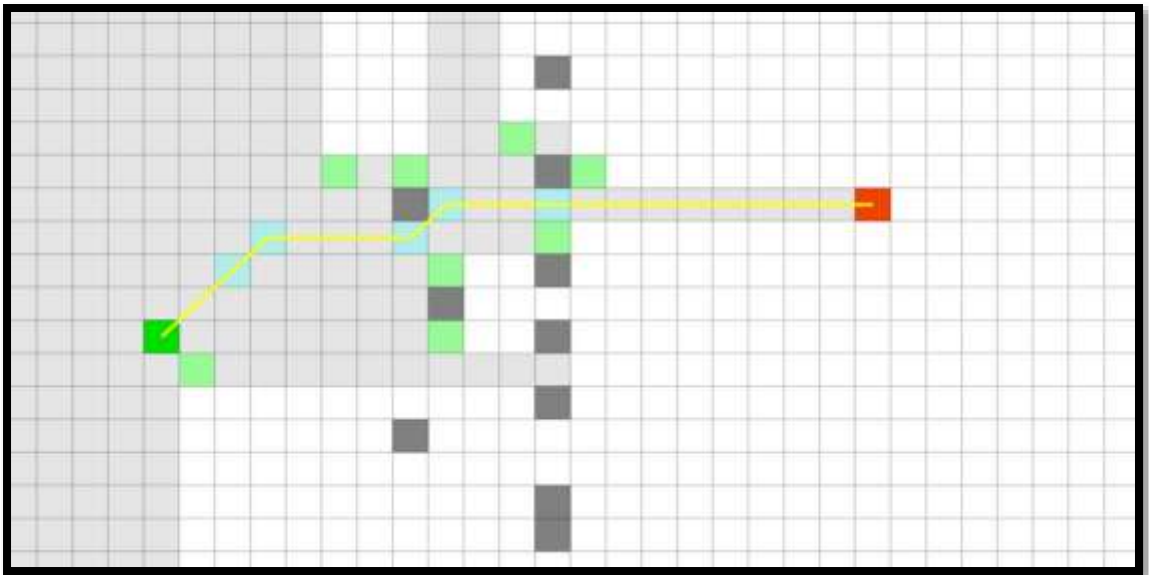
```

starting node is tagged as “closed” or no next node can be found.

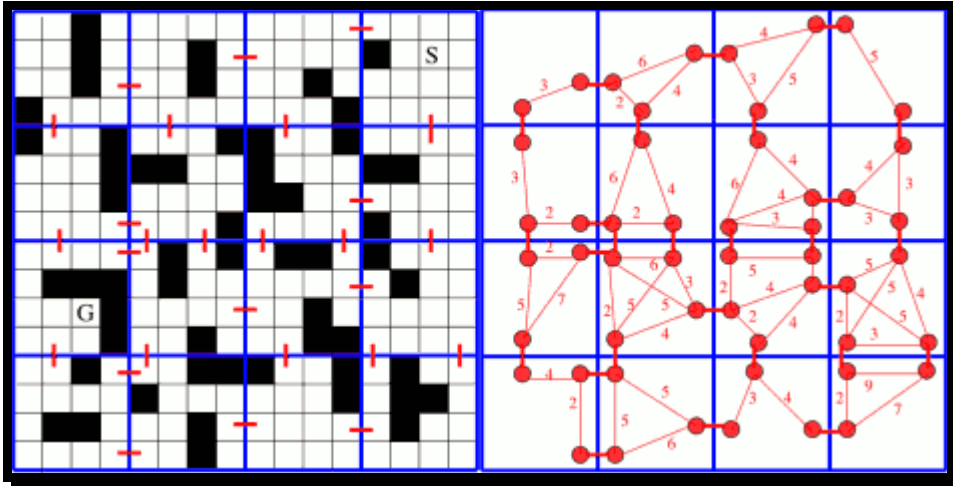
6) IDA* Algorithm (Refer Appendix)



7) Jump Point Search Algorithm (Refer Appendix)



HPA* (Refer Appendix)



8) Maze Generation Algorithms



Figure 37: Types of Maze Patterns in Pathfinding Visualizer

8.1 Graph theory-based mazes

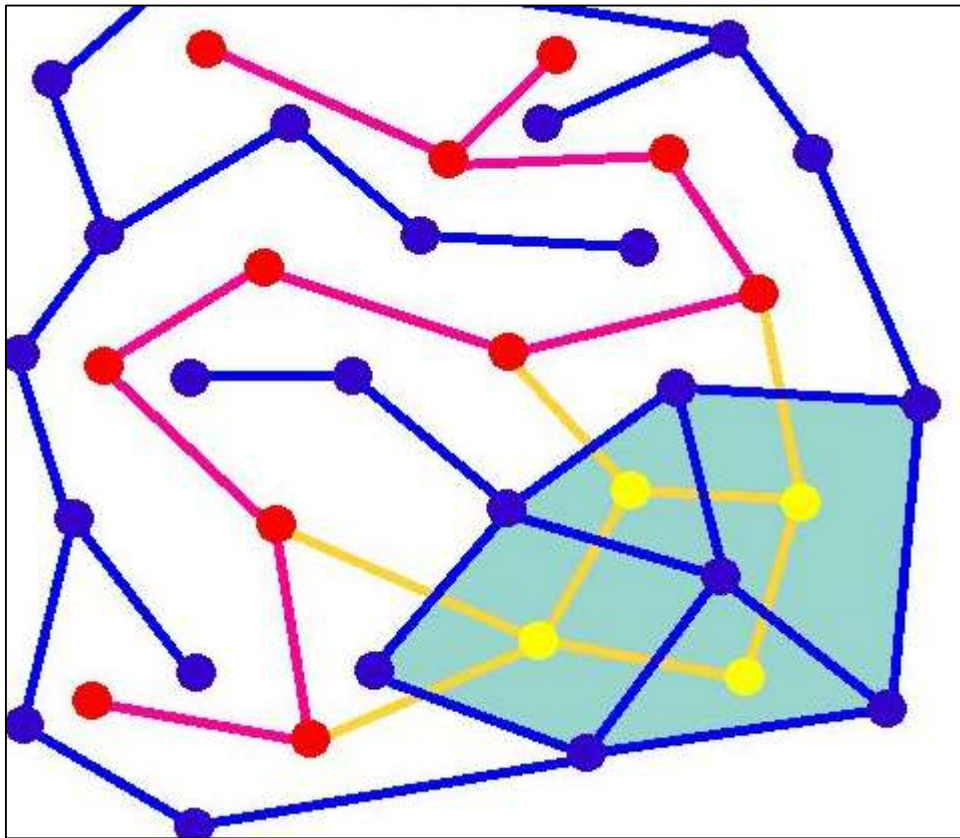


Figure 38: Graph theory-based Mazes Using Randomized depth-first search/DFS)

A maze can be generated by starting with a predetermined arrangement of cells (most commonly a rectangular grid but other arrangements are possible) with wall sites between them. This predetermined arrangement can be considered as a connected graph with the edges representing possible wall sites and the nodes representing cells. The purpose of the maze generation algorithm can then be making a subgraph in which it is challenging to find a route between two nodes.

If the subgraph is not connected, then there are regions of the graph that are wasted because they do not contribute to the search space. If the graph contains loops, then there may be multiple paths between the chosen nodes. Because of this, maze generation is often approached as generating a random spanning tree. Loops, which can confound naive maze solvers, may be introduced by adding random edges to the result during the algorithm.

The animation shows the maze generation steps for a graph that is not on a rectangular grid. First, the computer creates a random planar graph G shown in blue, and its dual F shown in yellow. Second, computer traverses F using a chosen algorithm, such as a depth-first search, colouring the path red. During the traversal, whenever a red edge crosses over a blue edge, the blue edge is removed. Finally, when all vertices of F have been visited, F is erased and two edges from G , one for the entrance and one for the exit, are removed.

8.2 Randomized depth-first search

The depth-first search algorithm of maze generation is frequently implemented using backtracking. This can be described with a following recursive routine:

1. Given a current cell as a parameter,
2. Mark the current cell as visited
3. While the current cell has any unvisited neighbour cells
 1. Choose one of the unvisited neighbours
 2. Remove the wall between the current cell and the chosen cell
 3. Invoke the routine recursively for a chosen cell

which is invoked once for any initial cell in the area.

8.3 Random Kruskal's algorithm



Figure 39: Creating a 30 x 20 maze using Randomized Kruskal's algorithm.

This algorithm is a randomized version of Kruskal's algorithm.

1. Create a list of all walls, and create a set for each cell, each containing just that one cell.
2. For each wall, in some random order:
 1. If the cells divided by this wall belong to distinct sets:
 1. Remove the current wall.
 2. Join the sets of the formerly divided cells.

There are several data structures that can be used to model the sets of cells. An efficient implementation using a disjoint-set data structure can perform each union and find operation on two sets in nearly constant amortized time so the running time of this algorithm is essentially proportional to the number of walls available to the maze.

8.4 Random Prim's algorithm

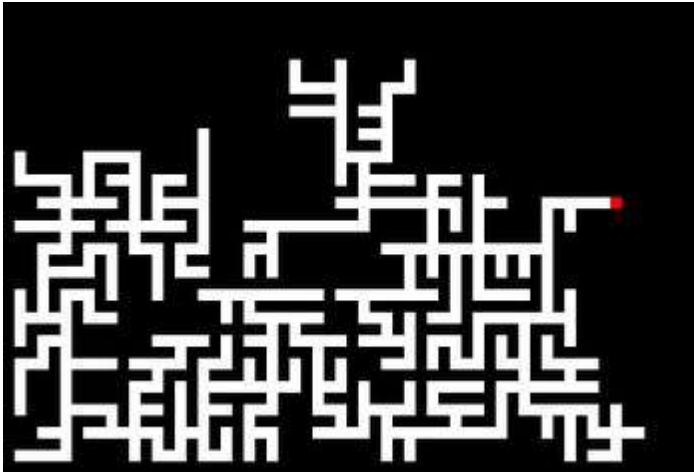


Figure 40: Creating a 30 x 20 maze using Prim's algorithm.

This maze creation algorithm is a randomized version of Prim's algorithm.

1. Start with a grid full of walls.
2. Pick a cell, mark it as part of the maze. Add the walls of the cell to the wall list.
3. While there are walls in the list:
4. Pick a random wall from the list. If only one of the cells that the wall divides is visited, then:
 - 4.1. Make the wall a passage and mark the unvisited cell as part of the maze.
 - 4.2 Add the neighboring walls of the cell to the wall list.
 - 4.3 Remove the wall from the list.


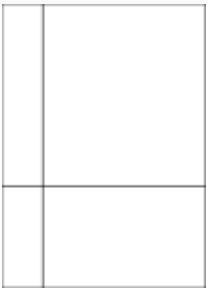

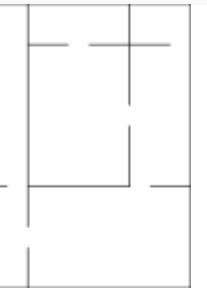
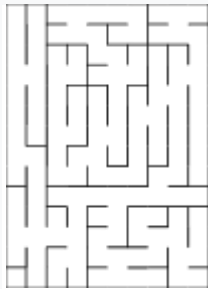
8.5 Aldous-Broder algorithm for Maze Generation

The Aldous-Broder algorithm also produces uniform spanning trees.

1. Pick a random cell as the current cell and mark it as visited.
2. While there are unvisited cells:
 1. Pick a random neighbor.
 2. If the chosen neighbor has not been visited:
 1. Remove the wall between the current cell and the chosen neighbor.
 2. Mark the chosen neighbor as visited.
 3. Make the chosen neighbor the current cell.

8.6 Maze via Recursive division method

Figure 41: Recursive Division Pattern

original chamber	division by two walls	holes in walls	continue subdividing...	completed
				
step 1	step 2	step 3	step 4	step 5

Mazes can be created with recursive division, an algorithm which works as follows: Begin with the maze's space with no walls. Call this a chamber. Divide the chamber with a randomly positioned wall (or multiple walls) where each wall contains a randomly positioned passage opening within it. Then recursively repeat the process on the sub chambers until all chambers are minimum sized. This method results in mazes with long straight walls crossing their space, making it easier to see which areas to avoid.



Figure 42: Recursive Maze Generation

For example, in a rectangular maze, build at random points two walls that are perpendicular to each other. These two walls divide the large chamber into four smaller chambers separated by four walls. Choose three of the four walls at random and open a one cell-wide hole at a random point in each of the three. Continue in this manner recursively, until every chamber has a width of one cell in either of the two directions.

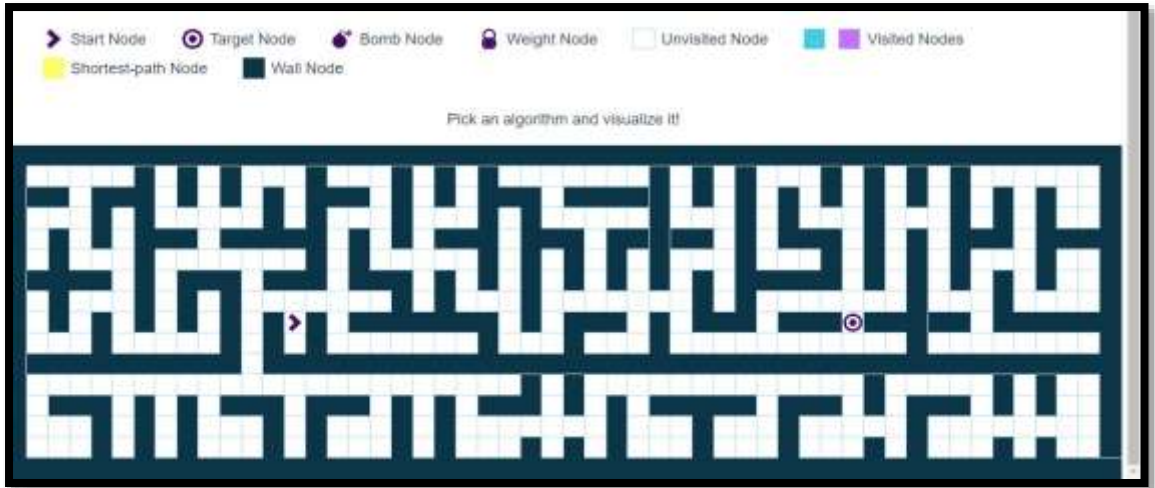


Figure 43: Recursive Division Maze



Figure 44: Recursive Division (Vertical skew)

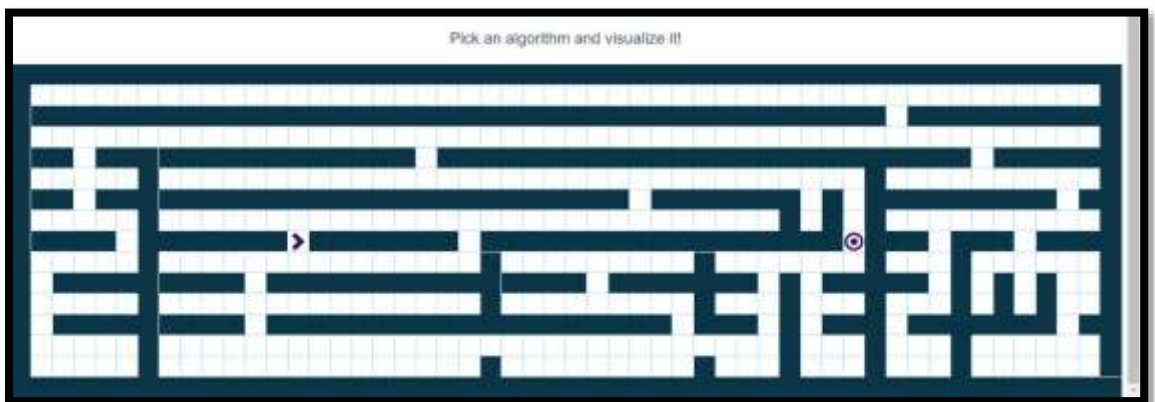


Figure 45: Recursive Division (horizontal skew)

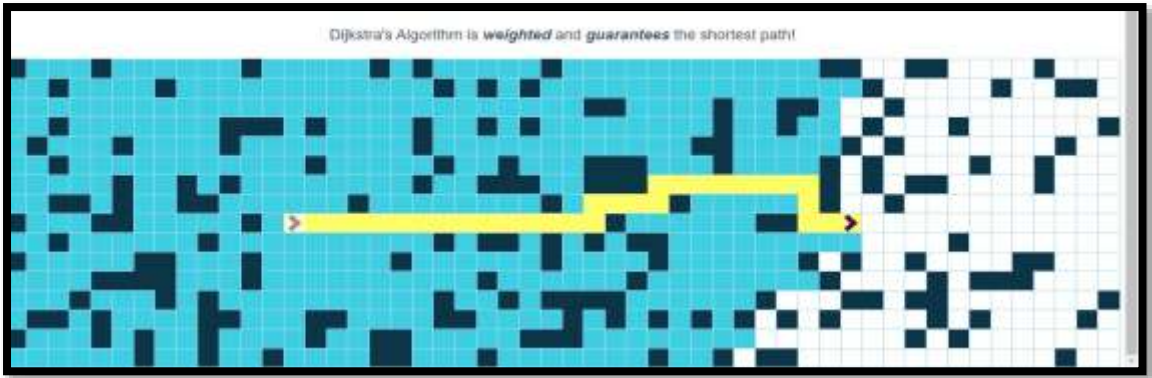


Figure 46: Random Maze

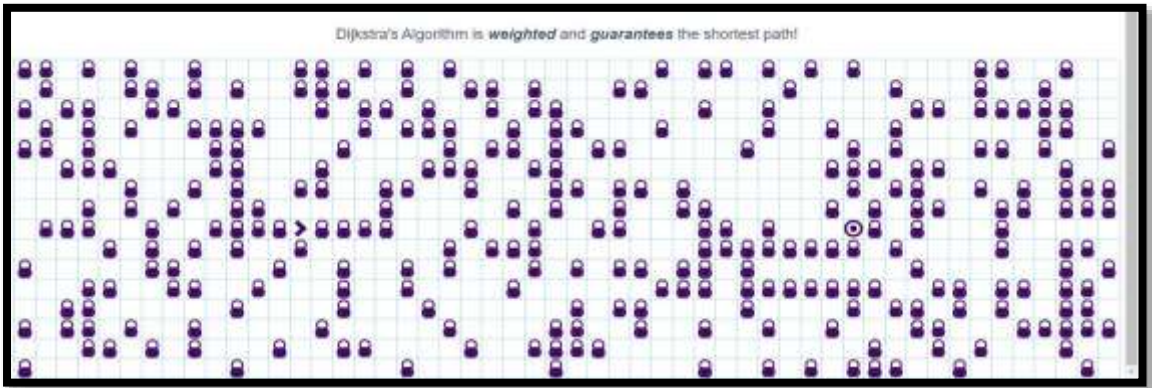


Figure 47: Random Weights Maze

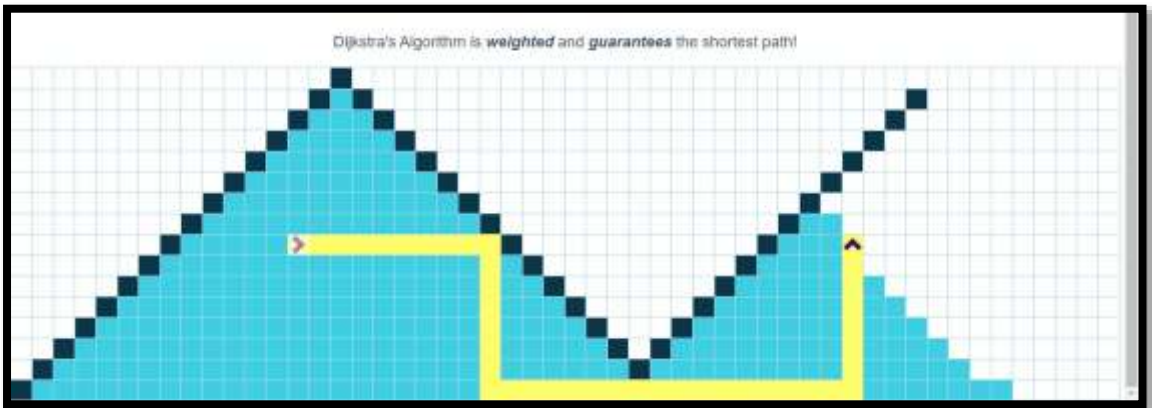


Figure 48: Simple Stairs Pattern

```

if (rowEnd - rowStart > currentCol - 2 - colStart) {
  recursiveDivisionMaze(board, rowStart, rowEnd, colStart, currentCol - 2, "horizontal", surroundingWalls)
} else {
  recursiveDivisionMaze(board, rowStart, rowEnd, colStart, currentCol - 2, "horizontal", surroundingWalls)
}
if (rowEnd - rowStart > colEnd - (currentCol + 2)) {
  recursiveDivisionMaze(board, rowStart, rowEnd, currentCol + 2, colEnd, "horizontal", surroundingWalls);
} else {
  recursiveDivisionMaze(board, rowStart, rowEnd, currentCol + 2, colEnd, orientation, surroundingWalls);
}
}
}
};

module.exports = recursiveDivisionMaze;

```

Recursive Division Code Snippet

```

function weightsDemonstration(board) {
  let currentIdx = board.height - 1;
  let currentIdY = 35;
  while (currentIdx > 5) {
    let currentId = `${currentIdx}-${currentIdY}`;
    let currentElement = document.getElementById(currentId);
    board.wallsToAnimate.push(currentElement);
    let currentNode = board.nodes[currentId];
    currentNode.status = "wall";
    currentNode.weight = 0;
    currentIdx--;
  }
  for (let currentIdx = board.height - 2; currentIdx > board.height - 11; currentIdx--) {
    for (let currentIdY = 1; currentIdY < 35; currentIdY++) {
      let currentId = `${currentIdx}-${currentIdY}`;
      let currentElement = document.getElementById(currentId);
      board.wallsToAnimate.push(currentElement);
      let currentNode = board.nodes[currentId];
      if (currentIdx === board.height - 2 && currentIdY < 35 && currentIdY > 26) {
        currentNode.status = "wall";
        currentNode.weight = 0;
      } else {
        currentNode.status = "unvisited";
        currentNode.weight = 15;
      }
    }
  }
}

```

Random Weights Maze Snippet

```

while (currentIdx < board.height - 2 && currentIdY < board.width) {
  let currentId = `${currentIdx}-${currentIdY}`;
  let currentNode = board.nodes[currentId];
  let currentHTMLNode = document.getElementById(currentId);
  if (!relevantStatuses.includes(currentNode.status)) {
    currentNode.status = "wall";
    board.wallsToAnimate.push(currentHTMLNode);
  }
  currentIdx++;
  currentIdY++;
}
while (currentIdx > 0 && currentIdY < board.width - 1) {
  let currentId = `${currentIdx}-${currentIdY}`;
  let currentNode = board.nodes[currentId];
  let currentHTMLNode = document.getElementById(currentId);
  if (!relevantStatuses.includes(currentNode.status)) {
    currentNode.status = "wall";
    board.wallsToAnimate.push(currentHTMLNode);
  }
  currentIdx--;
  currentIdY++;
}
}
}

```

Simple Stairs Pattern Code Snippet

3.3 Animations and UI



Figure 49: Animations and its Types

Animations to visualize the algorithm in action. They consist of:

- 1) On launch Animations: Which display animations on the starting page
- 2) Maze and Pattern Generating Animations: Modify weights and nodes on the 2D grid
- 3) Instant Animations: Display shortest path algorithms in action by changing color gradient using underlying CSS using JavaScript's internal timeout function.

Launch Animation Function Code Snippet

```
function launchAnimations(board, success, type, object, algorithm, heuristic) {
  let nodes = object ? board.objectNodesToAnimate.slice(0) : board.nodesToAnimate.slice(0);
  let speed = board.speed === "fast" ?
    0 : board.speed === "average" ?
    100 : 500;
  let shortestNodes;
  function timeout(index) {
    setTimeout(function () {
      if (index === nodes.length) {
        if (object) {
          board.objectNodesToAnimate = [];
          if (success) {
            board.addShortestPath(board.object, board.start, "object");
            board.clearNodeStatuses();
            let newSuccess;
            if (board.currentAlgorithm === "bidirectional") {
              // ...
            } else {
              if (type === "weighted") {
                newSuccess = weightedSearchAlgorithm(board.nodes, board.object, board.target, board.nodesToAnimate, board.boardArray, algorithm, heuristic);
              } else {
                newSuccess = unweightedSearchAlgorithm(board.nodes, board.object, board.target, board.nodesToAnimate, board.boardArray, algorithm);
              }
            }
            document.getElementById(board.object).className = "visitedObjectNode";
            launchAnimations(board, newSuccess, type);
            return;
          } else {
            console.log("Failure.");
            board.reset();
            board.toggleButtons();
          }
        }
      }
    }, speed);
  }
}
```

```
function shortestPathTimeout(index) {
  setTimeout(function () {
    if (index === shortestNodes.length) {
      board.reset();
      if (object) {
        shortestPathChange(board.nodes[board.target], shortestNodes[index - 1]);
        board.objectShortestPathNodesToAnimate = [];
        board.shortestPathNodesToAnimate = [];
        board.clearNodeStatuses();
        let newSuccess;
        if (type === "weighted") {
          newSuccess = weightedSearchAlgorithm(board.nodes, board.object, board.target, board.nodesToAnimate, board.boardArray, algorithm);
        } else {
          newSuccess = unweightedSearchAlgorithm(board.nodes, board.object, board.target, board.nodesToAnimate, board.boardArray, algorithm);
        }
        launchAnimations(board, newSuccess, type);
        return;
      } else {
        shortestPathChange(board.nodes[board.target], shortestNodes[index - 1]);
        board.objectShortestPathNodesToAnimate = [];
        board.shortestPathNodesToAnimate = [];
        return;
      }
    } else if (index === 0) {
      shortestPathChange(shortestNodes[index]);
    } else {
      shortestPathChange(shortestNodes[index], shortestNodes[index - 1]);
    }
    shortestPathTimeout(index + 1);
  }, 40);
}
```

Shortest Path Timeout Function

```

function shortestPathChange(currentNode, previousNode) {
  let currentHTMLNode = document.getElementById(currentNode.id);
  if (type === "unweighted") {
    currentHTMLNode.className = "shortest-path-unweighted";
  } else {
    if (currentNode.direction === "up") {
      currentHTMLNode.className = "shortest-path-up";
    } else if (currentNode.direction === "down") {
      currentHTMLNode.className = "shortest-path-down";
    } else if (currentNode.direction === "right") {
      currentHTMLNode.className = "shortest-path-right";
    } else if (currentNode.direction === "left") {
      currentHTMLNode.className = "shortest-path-left";
    } else if (currentNode.direction === "down-right") {
      currentHTMLNode.className = "wall"
    }
  }
}
if (previousNode) {
  let previousHTMLNode = document.getElementById(previousNode.id);
  previousHTMLNode.className = "shortest-path";
} else {
  let element = document.getElementById(board.start);
  element.className = "shortest-path";
  element.removeAttribute("style");
}
}

```

Shortest Path Change Function

```

function getDistance(nodeOne, nodeTwo) {
  let currentCoordinates = nodeOne.id.split("-");
  let targetCoordinates = nodeTwo.id.split("-");
  let x1 = parseInt(currentCoordinates[0]);
  let y1 = parseInt(currentCoordinates[1]);
  let x2 = parseInt(targetCoordinates[0]);
  let y2 = parseInt(targetCoordinates[1]);
  if (x2 < x1) {
    if (nodeOne.direction === "up") {
      return [1, ["f"], "up"];
    } else if (nodeOne.direction === "right") {
      return [2, ["l", "f"], "up"];
    } else if (nodeOne.direction === "left") {
      return [2, ["r", "f"], "up"];
    } else if (nodeOne.direction === "down") {
      return [3, ["r", "r", "f"], "up"];
    }
  } else if (x2 > x1) {
    if (nodeOne.direction === "up") {
      return [3, ["r", "r", "f"], "down"];
    } else if (nodeOne.direction === "right") {
      return [2, ["r", "f"], "down"];
    } else if (nodeOne.direction === "left") {
      return [2, ["l", "f"], "down"];
    } else if (nodeOne.direction === "down") {
      return [1, ["f"], "down"];
    }
  }
}

```

Get Distance Function: To measure distance between coordinates(x1,y1) and (x2,y2)

```

function getNeighbors(id, nodes, boardArray) {
  let coordinates = id.split("-");
  let x = parseInt(coordinates[0]);
  let y = parseInt(coordinates[1]);
  let neighbors = [];
  let potentialNeighbor;
  if (boardArray[x - 1] && boardArray[x - 1][y]) {
    potentialNeighbor = `${(x - 1).toString()}-${y.toString()}`
    if (nodes[potentialNeighbor].status !== "wall") neighbors.push(potentialNeighbor);
  }
  if (boardArray[x + 1] && boardArray[x + 1][y]) {
    potentialNeighbor = `${(x + 1).toString()}-${y.toString()}`
    if (nodes[potentialNeighbor].status !== "wall") neighbors.push(potentialNeighbor);
  }
  if (boardArray[x][y - 1]) {
    potentialNeighbor = `${x.toString()}-${(y - 1).toString()}`
    if (nodes[potentialNeighbor].status !== "wall") neighbors.push(potentialNeighbor);
  }
  if (boardArray[x][y + 1]) {
    potentialNeighbor = `${x.toString()}-${(y + 1).toString()}`
    if (nodes[potentialNeighbor].status !== "wall") neighbors.push(potentialNeighbor);
  }
  return neighbors;
}

```

Function to get closest Potential Neighbor Nodes

```

function updateNode(currentNode, targetNode, actualTargetNode, name, nodes, actualStartNode, heuristic, boardArray) {
  let distance = getDistance(currentNode, targetNode);
  let distanceToCompare;
  if (actualTargetNode && name === "aster") {
    if (heuristic === "manhattanDistance") {
      distanceToCompare = currentNode.distance + targetNode.weight + distance[0] + manhattanDistance(targetNode, actualTargetNode);
    } else if (heuristic === "poweredManhattanDistance") {
      distanceToCompare = currentNode.distance + targetNode.weight + distance[0] + Math.pow(manhattanDistance(targetNode, actualTargetNode), 3);
    } else if (heuristic === "extraPoweredManhattanDistance") {
      distanceToCompare = currentNode.distance + targetNode.weight + distance[0] + Math.pow(manhattanDistance(targetNode, actualTargetNode), 5);
    }
  }
  let startNodeManhattanDistance = manhattanDistance(actualStartNode, actualTargetNode);
  } else if (actualTargetNode && name === "greedy") {
    distanceToCompare = targetNode.weight + distance[0] + manhattanDistance(targetNode, actualTargetNode);
  } else {
    distanceToCompare = currentNode.distance + targetNode.weight + distance[0];
  }
  if (distanceToCompare < targetNode.distance) {
    targetNode.distance = distanceToCompare;
    targetNode.previousNode = currentNode.id;
    targetNode.path = distance[1];
    targetNode.direction = distance[2];
  }
}

```

Function to Update Nodes if Comparative distance < Target node distance


```

function test(nodes, start, target, nodesToAnimate, boardArray, name, heuristic) {
  if (!start || !target || start === target) {
    return false;
  }
  nodes[start].distance = 0;
  nodes[start].direction = "up";
  let unvisitedNodes = Object.keys(nodes);
  while (unvisitedNodes.length) {
    let currentNode = closestNode(nodes, unvisitedNodes);
    while (currentNode.status === "wall" && unvisitedNodes.length) {
      currentNode = closestNode(nodes, unvisitedNodes);
    }
    if (currentNode.distance === Infinity) return false;
    currentNode.status = "visited";
    if (currentNode.id === target) {
      while (currentNode.id !== start) {
        nodesToAnimate.unshift(currentNode);
        currentNode = nodes[currentNode.previousNode];
      }
      return "success!";
    }
    if (name === "astar" || name === "greedy") {
      updateNeighbors(nodes, currentNode, boardArray, target, name, start, heuristic);
    } else if (name === "dijkstra") {
      updateNeighbors(nodes, currentNode, boardArray);
    }
  }
}

```

Function to test Different Algorithms (Both weighted and un-weighted) with given obstacles and animate nodes simultaneously

Chapter 4

Performance Analysis

4.1 Sample simulations

In these stills from the simulator, small red boxes (1) signify nodes, and blue lines show the links between the nodes, signifying the entire graph tree. Yellow boxes (2), slightly larger, signify nodes that have been examined by the search algorithm, and green boxes (3), slightly larger than the yellow boxes, signify nodes that were determined to be part of the best path chosen by the algorithm. Black boxes (not shown) are nodes that were determined to be dead ends. The read out in the bottom left corner shows how many simulations (of all specified algorithms) out of the total have been run, followed by the name of the algorithm being used in the current test. The agent is the penguin model, and the target is the bull's-eye texture. In each case, the path calculated by the algorithm has been shown using a thicker line.

Simulation 1 : Nodes placed at the boundary near the edges and destination node at the cross section between two paths, The path grid is in form of a uniform square mesh

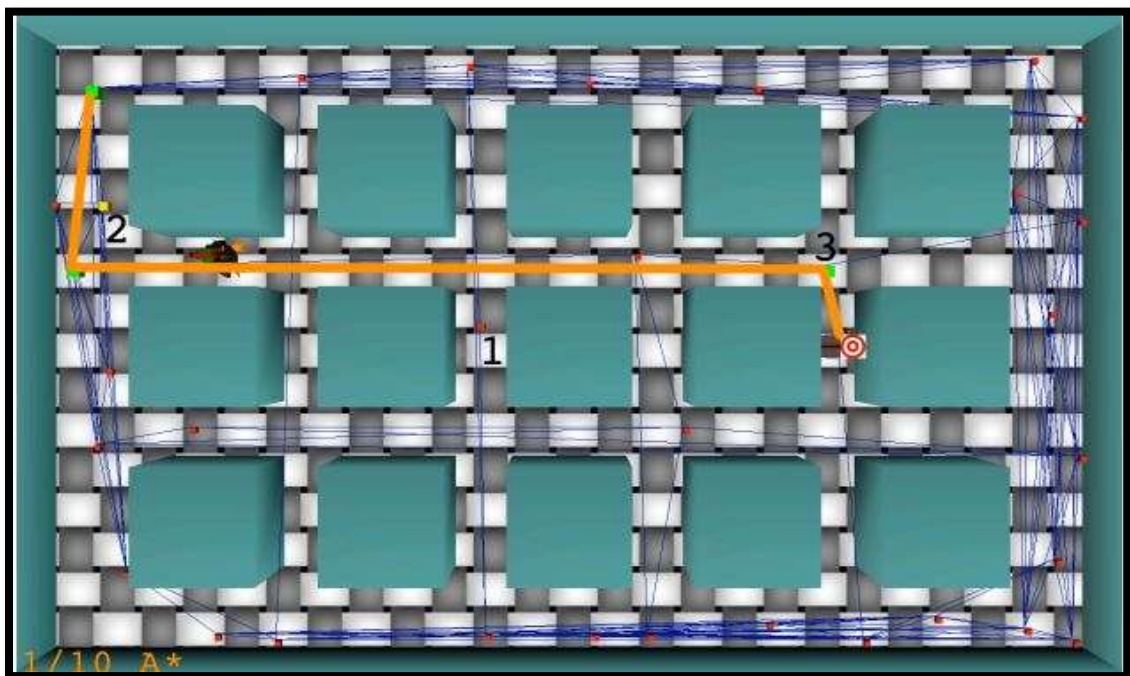


Figure 50. Testing Tool in Simulation 1, A*



Figure 51. Testing Tool in Simulation 1, Dijkstra's algorithm

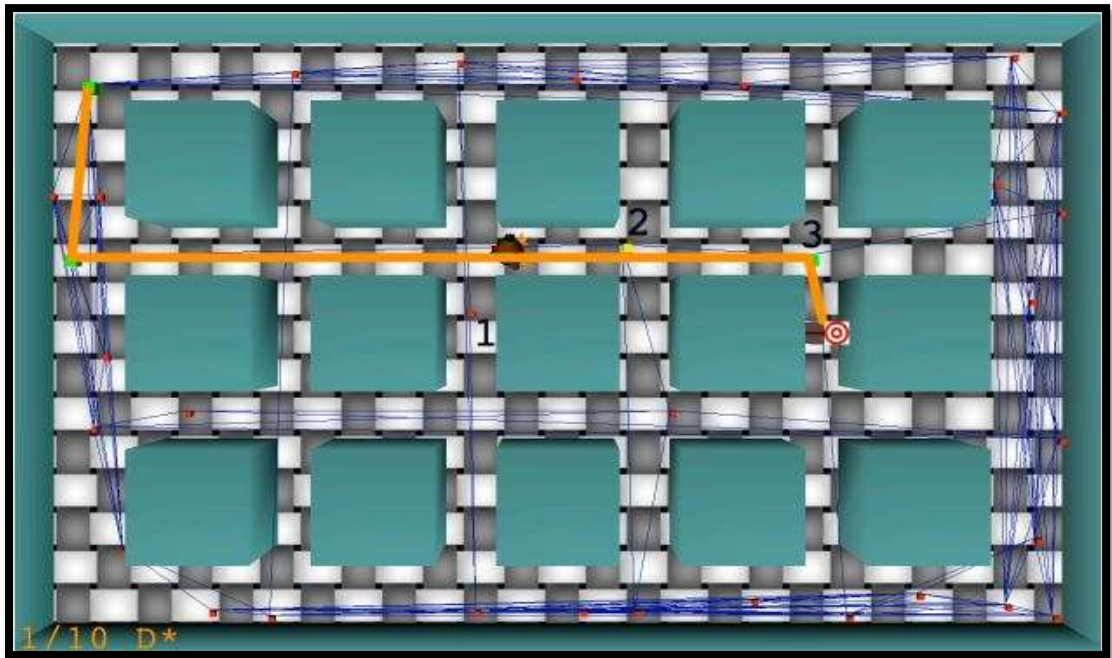


Figure 52. Testing Tool in Simulation 1, D*

In these examples from Environment 1, all algorithms found the same path. However, in the screenshot from the run using Dijkstra's algorithm, 10 nodes are marked as examined, including the nodes that are part of the path, as opposed to 5 each in the runs using A* and D*. Dijkstra's algorithm examined more nodes than either of the other algorithms owing to its greedy nature.

Simulation 2: The path grid is non-uniform mesh with broken linkages to make the path more complex, Nodes placed randomly in the mesh and Destination node at the center of the grid

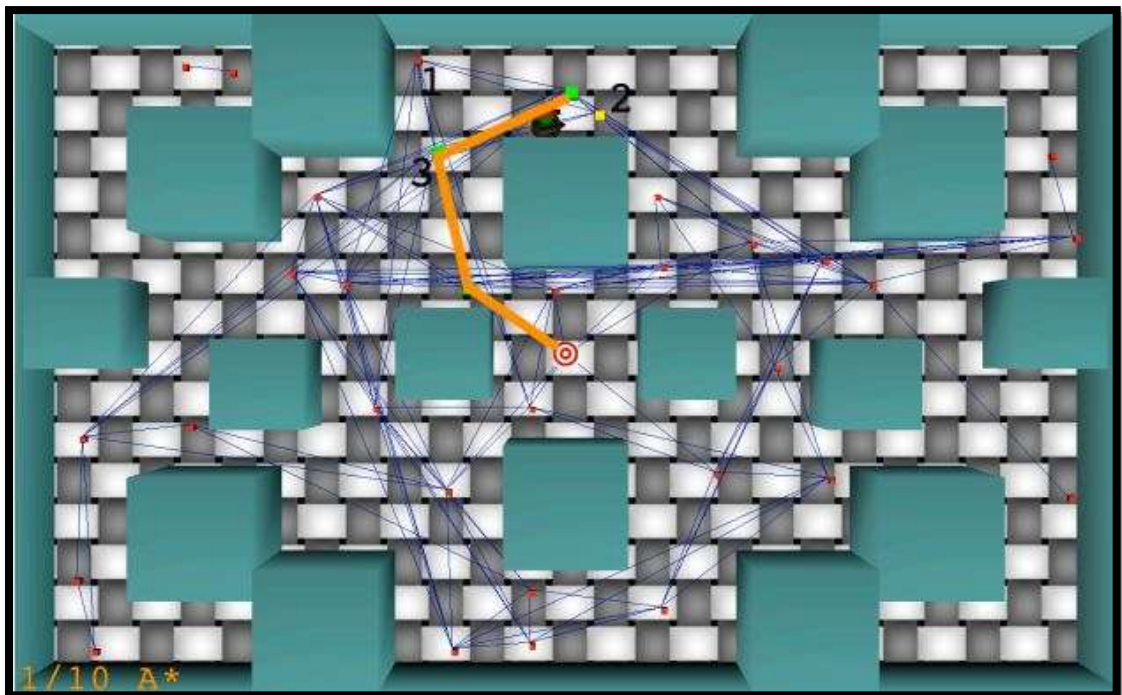


Figure 53 Testing Tool in Simulation 2, A*



Figure 54. Testing Tool in Simulation 2, Dijkstra's algorithm

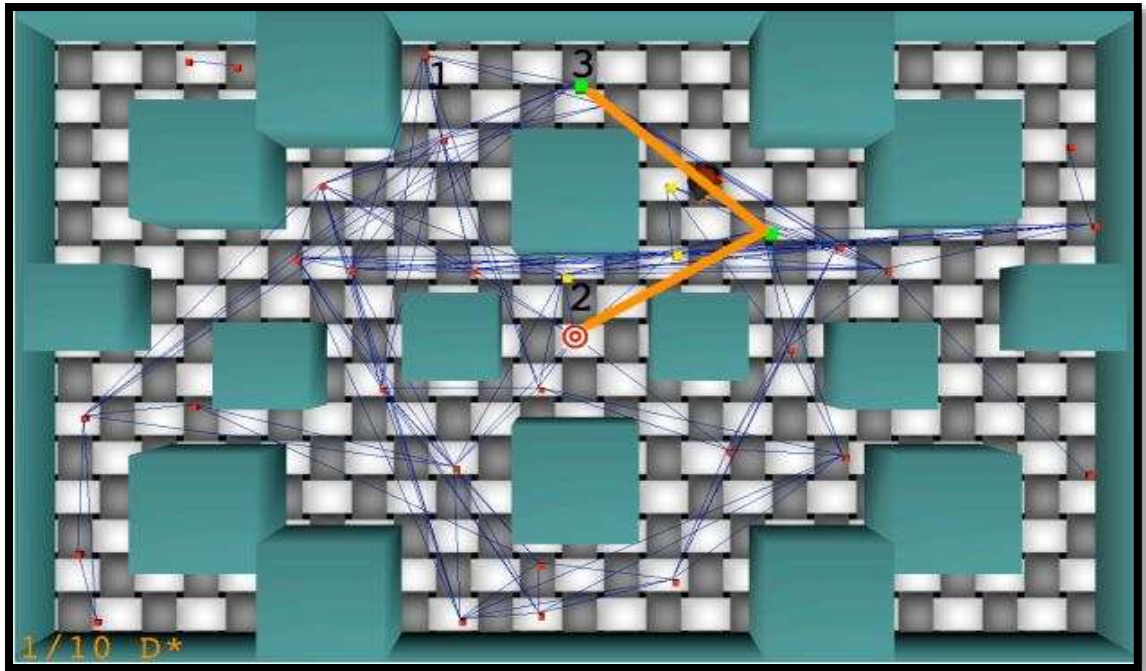
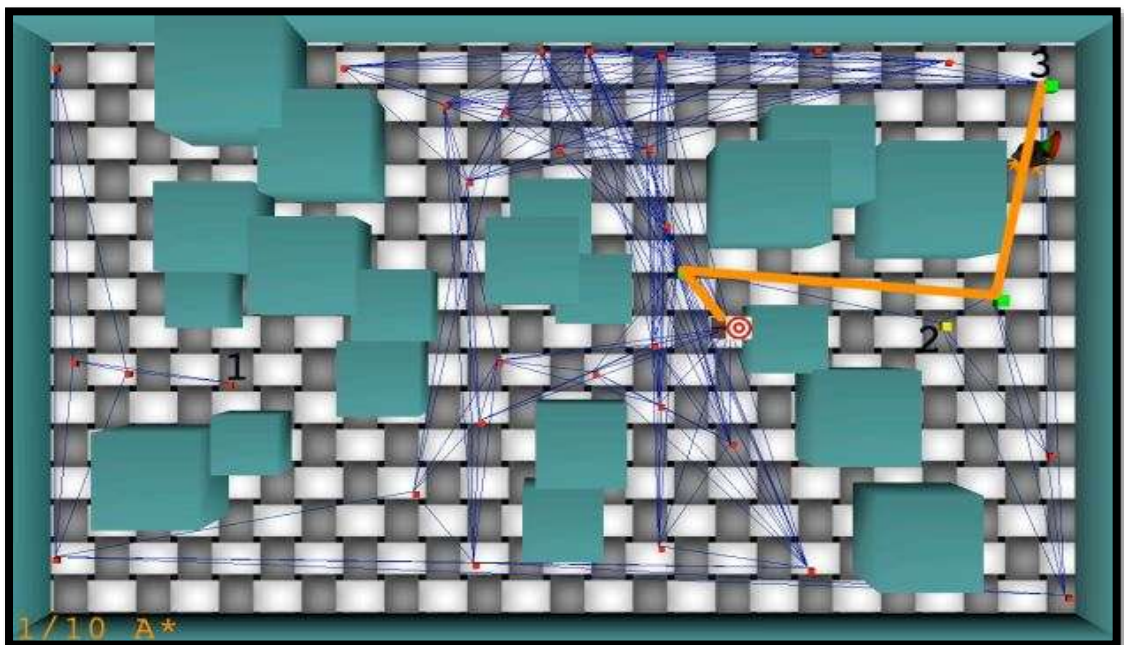


Figure 55. Testing Tool in Simulation 2, D*

We can devise from simulation 2, A* and Dijkstra's algorithm found the same path, but Dijkstra's algorithm examined 16 nodes to find it, while A* examined 5. D* found a path that traversed fewer nodes than the other path. This path is 53.081 units long, versus the other path, which is 56.835 units long. D* calculated this path by working backwards from the target to the start, rather than the other way around.

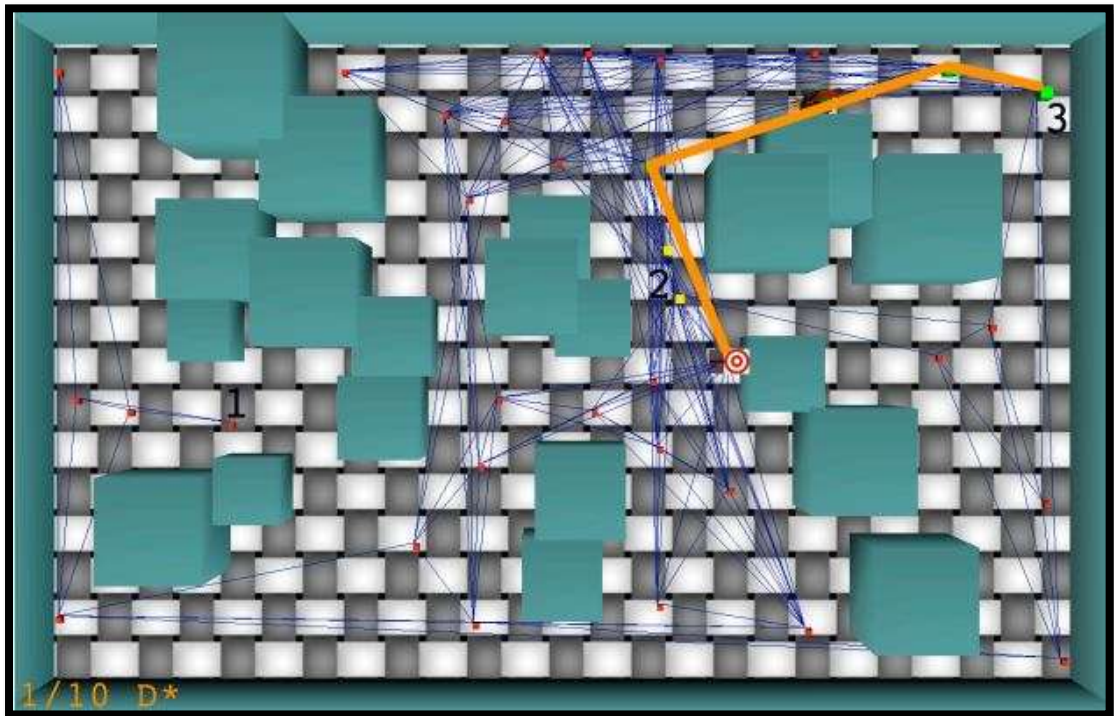
Simulation 3: Another variant of the broken non-uniform mesh grid with nodes spread over a large area , Destination node at the center of the grid



Figures 56. Testing Tool in Simulation 3, A*



Figures 57. Testing Tool in Simulation 3, Dijkstra's algorithm



Figures 58. Testing Tool in Simulation 3, D*

Testing simulation 3, A* found a path that was different from the path found by Dijkstra's game world, of the calculated method over all given algorithm tests and location. This parameter is selected by L. 4. The average time, in seconds, spent following the path from start to finish over all given algorithm tests and location. This parameter is designated by Es. 5. Estimated value of the main application loop completed during route execution for all given algorithm tests and location. This parameter is selected by El. 6. The total number of times the method was successfully detected by all three algorithms 7. The total number of times all algorithms found the same method in all the tests in each area.

Table 1: Simulation 1 Output

ALGORITHM	Avg. # node visitations (V)	Avg. # nodes in a path (N)	Avg. length of a path (units) (L)	Avg. path execution time (seconds) (Es)	Avg. path execution time (app. loops) (El)
A* Search Algorithm	23.915	2.9	137.65	3.564	108.97
Dijkstra's Algorithm	145.665	2.855	134.014	3.507	102.33
D* Search Algorithm	3.64	2.77	132.412	3.472	106.05
BFS	112.57	2.83	135.43	3.402	110.56
DFS	119.55	2.75	130.33	3.55	111.22

Total number of successful path calculations: 94/100 (94.00%)

Total number of times all algorithms found the same path: 75/98 (76.53%)

The lengths of all paths, as well as the execution times, are similar. However, the average number of node visitation made by the D* algorithm is far less than the number of visitations made by either of the other algorithms, indicating the shorter search time for D* in this environment. In second place is A*, which made, on average, a higher number of node visitations than D*, but a lower number than Dijkstra's algorithm. All three algorithms calculated the same path over 76% of the time.

Table 2: Simulation 2 Output

ALGORITHM	Avg. # node visitations (V)	Avg. # nodes in a path (N)	Avg. length of a path (units) (L)	Avg. path execution time (seconds) (Es)	Avg. path execution time (app. loops) (El)
A* Search Algorithm	5.062	3.353	108.004	3.046	94.682
Dijkstra's Algorithm	24.494	3.271	105.206	2.93	86.833
D* Search Algorithm	12.3	3.271	106.417	3.006	93.282
BFS	17.32	3.2	106.32	3.052	97.238
DFS	15.22	3.201	108.34	3.15	97.554

Total number of successful path calculations: 88/100 (88.00%) Total number of times all algorithms found the same path: 60/94 (63.82%)

In this case, the average length of a calculated path, as well as the execution times, are again very close. However, this time, A* makes the fewest node visitations, with D* in second place, making, on average, about half the number of node visitations as Dijkstra's algorithm. The algorithms had the lowest rate of successful path calculation in this environment. The algorithms also produced the most variations in paths in this environment. All three algorithms calculated the same path 64% of the time.

Table 3 : Simulation 3 Output

ALGORITHM	Avg. # node visitations (V)	Avg. # nodes in a path (N)	Avg. length of a path (units) (L)	Avg. path execution time (seconds) (Es)	Avg. path execution time (app. loops) (El)
A* Search Algorithm	4.96	2.38	113.254	2.997	91.04
Dijkstra's Algorithm	23.36	2.38	111.552	2.956	85.96
D* Search Algorithm	7.18	2.3	109.805	2.899	87.98
BFS	11.22	2.4	112.55	2.91	90.02
DFS	12.34	2.5	112.76	3.01	90.99

Total number of successful path calculations: 96/100(9600%)

Total number of times all algorithms found the same path: 72/98 (73.46%)

The paths found by all algorithms were very close to each other in average length and execution time. A* made the fewest node visitations, on average. D* is a close second, and its average number of node visitations are closer to the average number of node visitations made by A* than to the number made by Dijkstra's algorithm, BFS DFS had visitations lower than it. All algorithms found the same path 73% of the time, which is a higher rate of consistency among the simulations.

4.2 Analysis of Different Algorithms

An efficient algorithm is one that calculates the shortest path with the fewest number of node visitations. A basic "efficiency score", S, can be calculated for each algorithm in each environment by multiplying the average number of node visitations by the average length of a path. Multiply the reciprocal of this number by a constant k to increase understandability (in this case, k=10000). Therefore, $S = k/(V*L)$, where S is the efficiency score, V is the average number of node visitation, and L is the average length of a path in units. A larger value indicates a more efficient algorithm. These values are specific to the environment and can only be used to compare the efficiency of different algorithms in the same environment.

Table 4: Computing Efficiency scores

ALGORITHM	GRID1	GRID2	GRID3
A* Search Algorithm	3.037	18.291	17.802
Dijkstra's Algorithm	0.512	3.881	3.838
D* Search Algorithm	20.748	7.357	12.684
BFS	0.442	1.223	2.445
DFS	0.344	1.003	2.303

In all 6 cases , BFS,DFS were the least efficient, followed by Dijkstra as these algorithms have no method of cutting down on the search space. As seen in the data tables above, the average length of a path found by all algorithms was similar, but in all environments, Dijkstra's algorithm made far more node visitations during the path calculation phase than either A* or D*. BFS,DFS performed slightly better than Dijkstra on this parameter. This led to low performance scores for Dijkstra,BFS,DFS. D* was by far the most efficient algorithm, with a fewer average node visitations per

path. It had the lowest average number of nodes in a path, and shortest average path length. In case of more obstructed grids, however A* performed better than D*. D* could determine that no path existed more quickly than A* only in unobstructed space where it expanded less rapidly, fewer node visitations.

In open/ far spaced grids A* could either determine more quickly that no path existed. D* took more time and had to check a greater number of nodes, expanding the search space more quickly.

Dijkstra, BFS and DFS were the slowest performers on a general basis.

Table 5: Comparison of BFS and Dijkstra on Optimality, Queue type and Time Complexity

	BFS	Dijkstra
Main Concept	Visit nodes level by level based on the closest to the source	In each step, visit the node with the lowest cost
Optimality	Gives an optimal solution for unweighted graphs or weighted ones with equal weights	Gives an optimal solution for both weighted and unweighted graphs
Queue Type	Simple queue	Priority queue
Time Complexity	$O(V + E)$	$O(V + E(\log V))$

Table 6: Comparison between Uninformed and Informed Search Algorithms in terms of execution time, Nodes Traversed and Path length

Type	Algorithm	Execution time(ms)	Traversed Nodes	Length
Uninformed Search	Dijkstra	1.89	496	23.36
	IDDFS	9.64	423	23.36
	BIDDFS	3.67	231	23.36
	BFS(Breadth)	7.33	993	23.36
Informed Search	Greedy Best First Search	2.2	53	29.31
	Ida*	5.232	312	28.54
	A*	1.96	46	23.36
	Jump point search	1.54	312	23.36
	HPA*	1.11	36	23.36

Table 7: Comparison of Algorithm Performance

	Time	Expansion	Path Length	Path Nodes
A*	Good	Good	Average	Average
Theta*	Average	Good	Good	Great
HPA* online	Bad	Bad	Bad	Bad
HPA* offline	Great	Great	Bad	Bad
IDA*	Very Bad	Very Bad	Average	Average
Dijkstra	Average	Average	Average	Average

4.4 Summary of Comparative study of Different Algorithms

Based on the three simulations I calculated values of V, the average no. of node visitation and L, the average length of path to get the performance score P which depicted the efficiency of each algorithm under different environments. In case of uniform grid with no broken paths i.e., in grid 1 , D* Search had the maximum efficiency score of 20.748 followed by A* Search of 3.037, approx. seven times faster. In non-uniform grid 2, A* Search was better than D* Search by approx. (18.291/7.357) 2.5 times. In grid 3, A* Search again outperformed D*Search by (17.802/12.684) 1.4 times but this time the difference was much lower. Thus, D* Search performed best overall but slowed in case the grids were much more obstructed/broken as it expanded much more rapidly and had more node visitations, D* Search iterates over all the possible paths between source and destination nodes to find the most optimal route requiring least amount of time and distance to be travelled.

On testing above algorithms on metrics of path length, node visitations, node expansion and computation time , different algorithms excelled on different parameters, Theta * (A* and D* combination) gave the best path, A* Search had the best computation time and HPA* although had a high initial computational cost but evolved to get more effective in the subsequent searches. IDA* performed bad in case of small search spaces when denoted by octile maps as it expanded too rapidly in time and distance.

Summarizing the algorithms, time complexity , graphs used and applications:

- **Depth-first search**

Depth-first search (DFS) is an algorithm for traversing or searching graph or tree. It uses a stack, and it delays checking whether a vertex has been discovered until the vertex is popped from the stack rather than making this check before adding the vertex.

Time Complexity: $O(E+V)$

Graph type:

- Undirected Graph
- Directed Acyclic Graphs (DAG) without weight

Applications:

- Topological sorting
- Maze
- Finding strongly connected components
- Detecting cycle in a graph

- **Breadth-first search**

Primarily uses a queue data structure to traverse a tree/ graph.

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It uses a queue.

Time Complexity: $O(E+V)$

Graph type:

- Undirected Graph
- Directed Acyclic Graphs (DAG) without weight

Applications:

- Shortest Path
- Social Media Network
- Peer to Peer Networks (BitTorrent)
- Crawlers in Search Engines
- Copying garbage collection

- **Dijkstra's algorithm**

A greedy algorithm, it first chooses the unvisited vertices having least distance to the source nodes and subsequently computes distance via it to each and every unvisited neighbour node and if the distance is smaller than the one being considered it updates

it, using a priority queue in the process.

Time Complexity: $O(E+V \log V)$ (with min priority queue)

Graph type:

- non-negative weighted DAG

Applications:

- Shortest path
- Digital Mapping Services in Google Maps (G-Maps)
- IP routing to find Open shortest Path First: Open Shortest Path First (OSPF)
- Designate file server in LAN
- Social Networking Applications (Smaller Graphs)

A* algorithm

A* Search advances Dijkstra and behaves as a Greedy Best First Search Algorithm via using educated guesses or heuristics to help itself reach the most optimal path in a faster time.

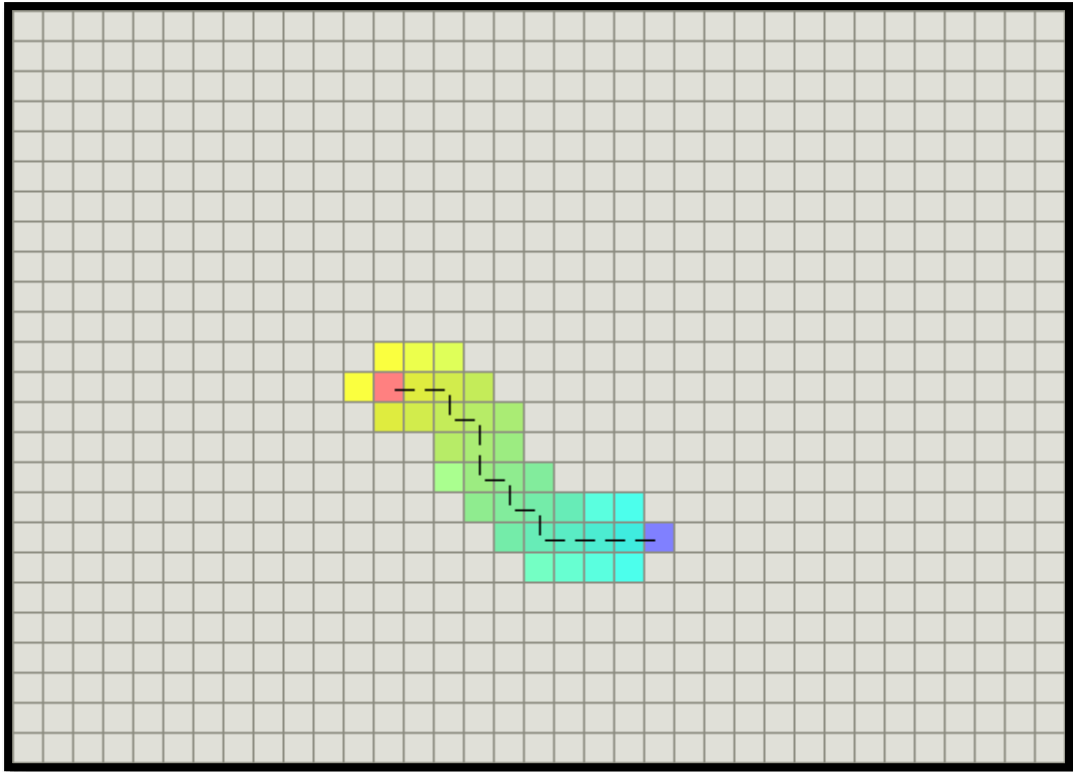


Figure 60: A* Search Algorithm

With a concave obstacle, A* and Dijkstra's Algorithm both find almost similar paths:

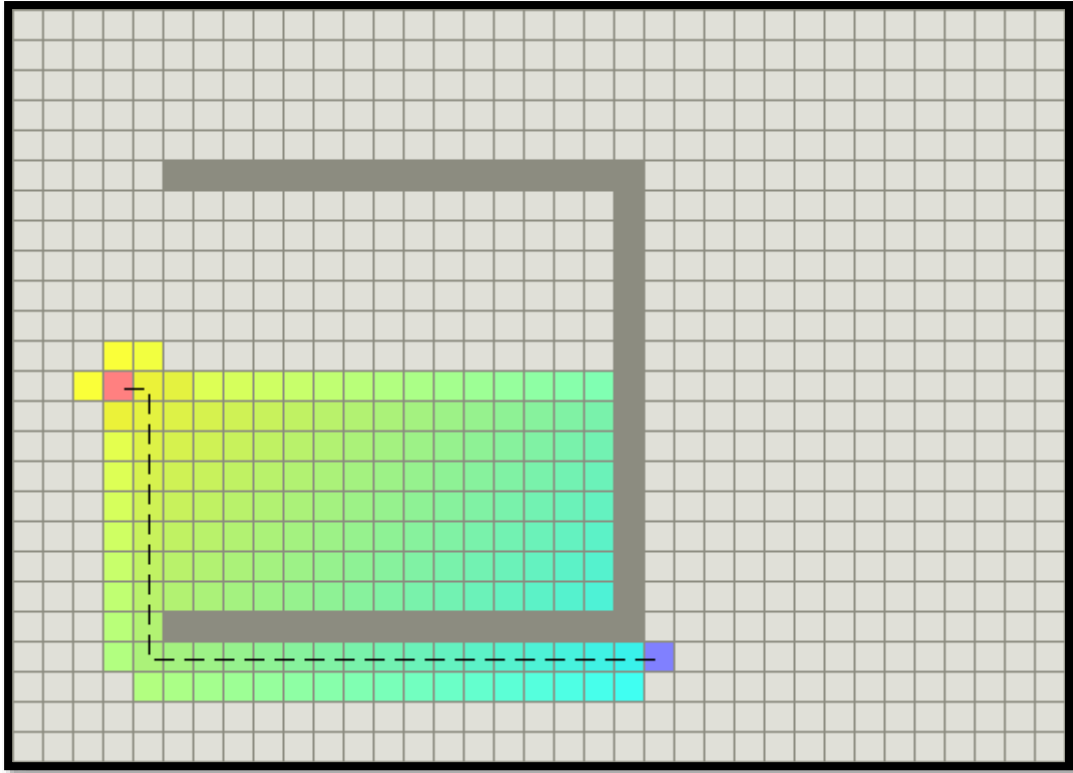


Figure 61: A* Search

The secret to its success is that it combines the pieces of information that Dijkstra's Algorithm uses (favouring vertices that are close to the starting point) and information that Greedy Best-First Search uses (favouring vertices that are close to the goal). In the standard terminology used when talking about A*, $g(n)$ represents the exact cost of the path from the starting point to any vertex n , and $h(n)$ represents the heuristic estimated cost from vertex n to the goal. In the above diagrams, the yellow (h) represents vertices far from the goal and teal (g) represents vertices far from the starting point. A* balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the vertex n that has the lowest $f(n) = g(n) + h(n)$.

- **Swarm Algorithm**

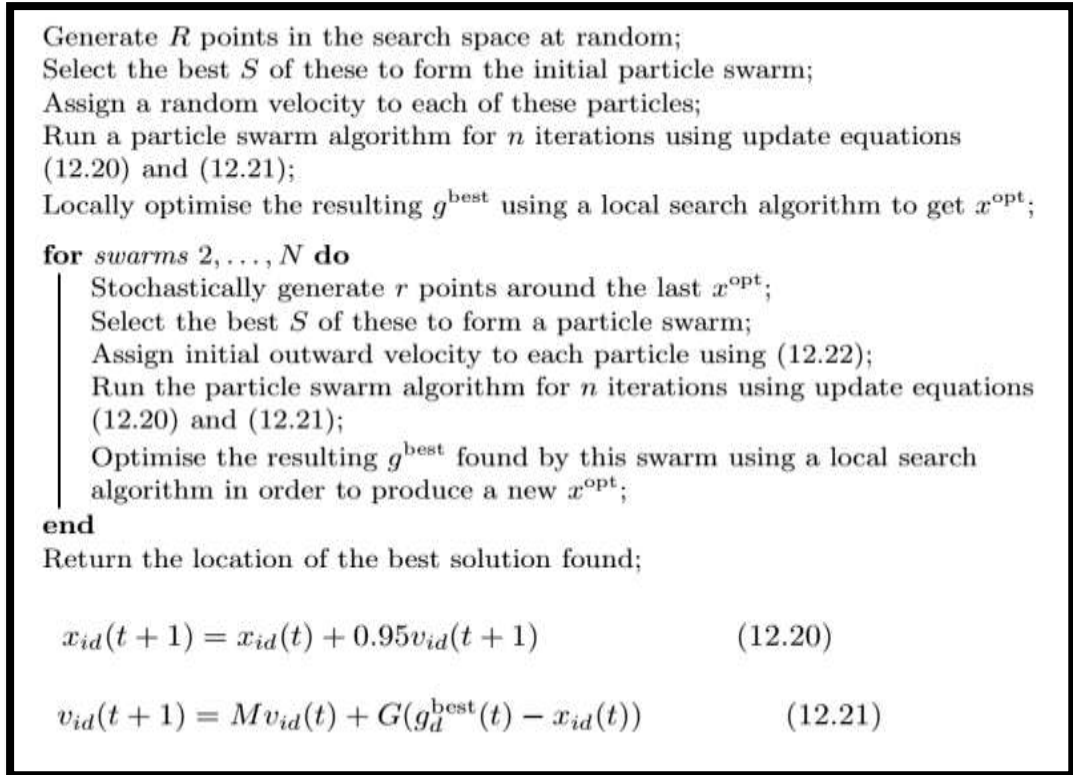


Figure 62: Locust Swarm Algorithm for Path Finding

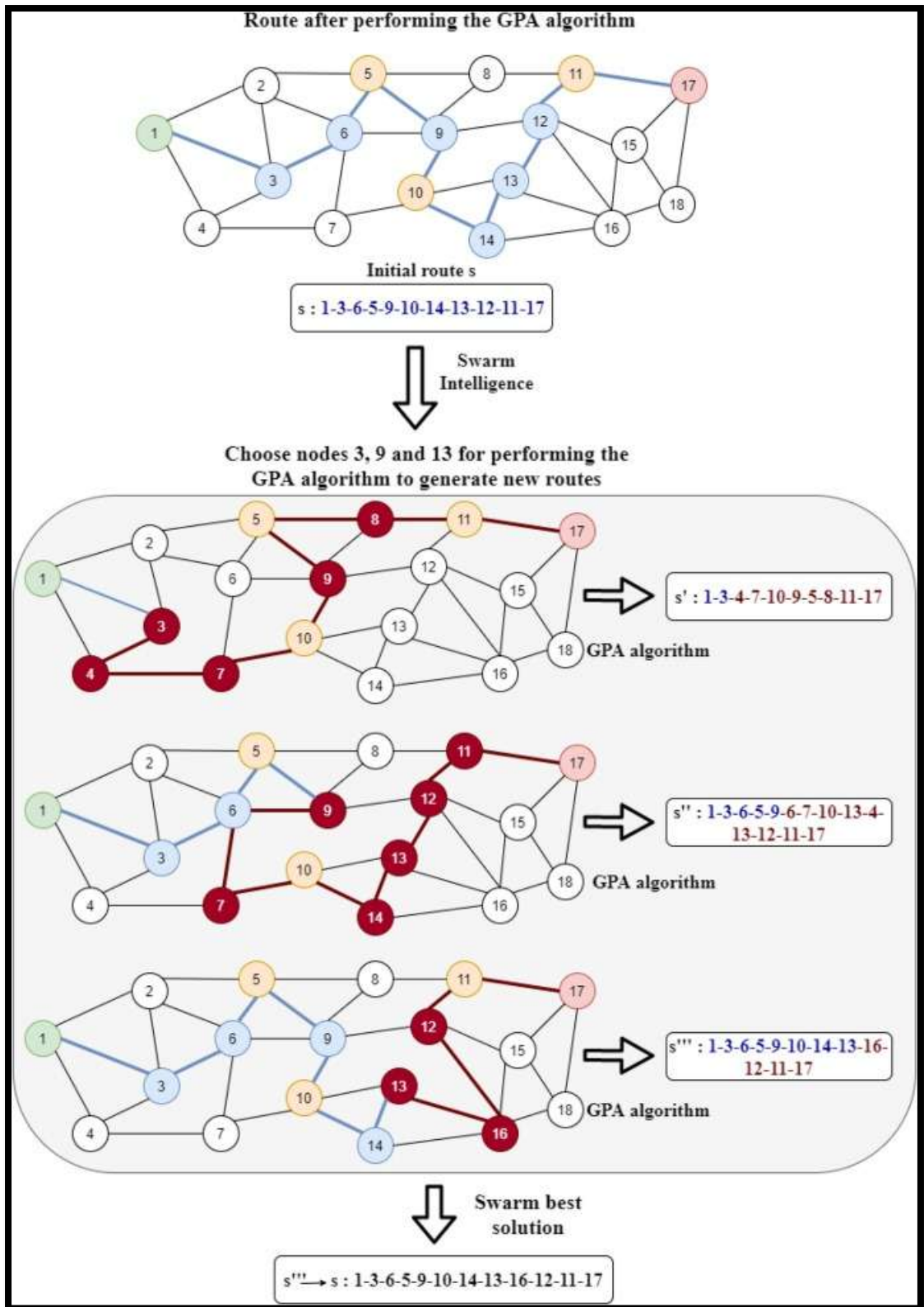


Figure 63: A swarm intelligence graph-based path finding algorithm

Chapter 5

Conclusions

5.1 Conclusions

On successfully concluding the project we have achieved our project goal of visualizing Path Finding Algorithm in action and comparing their performance. As we were aware that there was a huge ridge between thesis and practical understanding of the implementation of algorithms, the main objective of the project is to implement these algorithms and hence understand the integrated graph problems available. In conclusion, we have gained a lot by revisiting the implementation of these algorithm and the basic web concepts. Following remarks can be made after analyzing different algorithms:

- Dijkstra's algorithm is BFS but with a priority queue data structure and failed in case of negative weights, for which Bellman Ford algorithm is used but many more algorithms were built on it and it served as a base for many others.
- DFS keeps traversing nodes along their depth and backtracking until either it finds a path or reaches a dead end, While Dijkstra is more like a BFS except it keeps track of weights (not all paths have equal cost) and will keep checking the shortest path not already visited until it finds the destination node.
- Generally, DFS is (usually) the fastest way to find a path but it is not necessarily shortest and can be implemented very easily with recursion, but Dijkstra's algorithm is the general way to find the shortest possible path faster. DFS traverses to the depth and then backtracks to find other possible paths which may be shorter than the initial paths.
- Particularly, A*, which is a modified version of Dijkstra's algorithm with some extra heuristics/ educated guesses to make better decisions of choosing the correct path nearest to the end node first. Selecting the correct set of heuristics forms an integral part of the algorithm,
- A* and D* both could almost equally determine that no path existed. Dijkstra on the other hand uses more node visitations to give the same result as it lacks heuristics and thus needed more computation time expanding search space unnecessarily, wasting computation power on useless nodes which ultimately did not lead to the end node.
- D* Search was the best algorithm overall in terms of performance but could not perform well in case of obstructed spaces, where A* Search was better. BFS, DFS and Dijkstra were slowest performers with DFS not even always guaranteeing the shortest path. DFS was much more efficient in finding all the possible paths from source to destination rather than the most optimal route.

5.2 Future Scope

Having worked on a 2D grid to visualize all the path finding algorithms in this project, i plan to extend the scope of this project to a 3D grid, wherein one can find shortest route between two actual locations in a 3D simulation kind of game. This would allow us to test these algorithms in a real time environment and help us devise the most efficient combination considering different terrains, altitudes, and locations. A blueprint for this concept as follows.

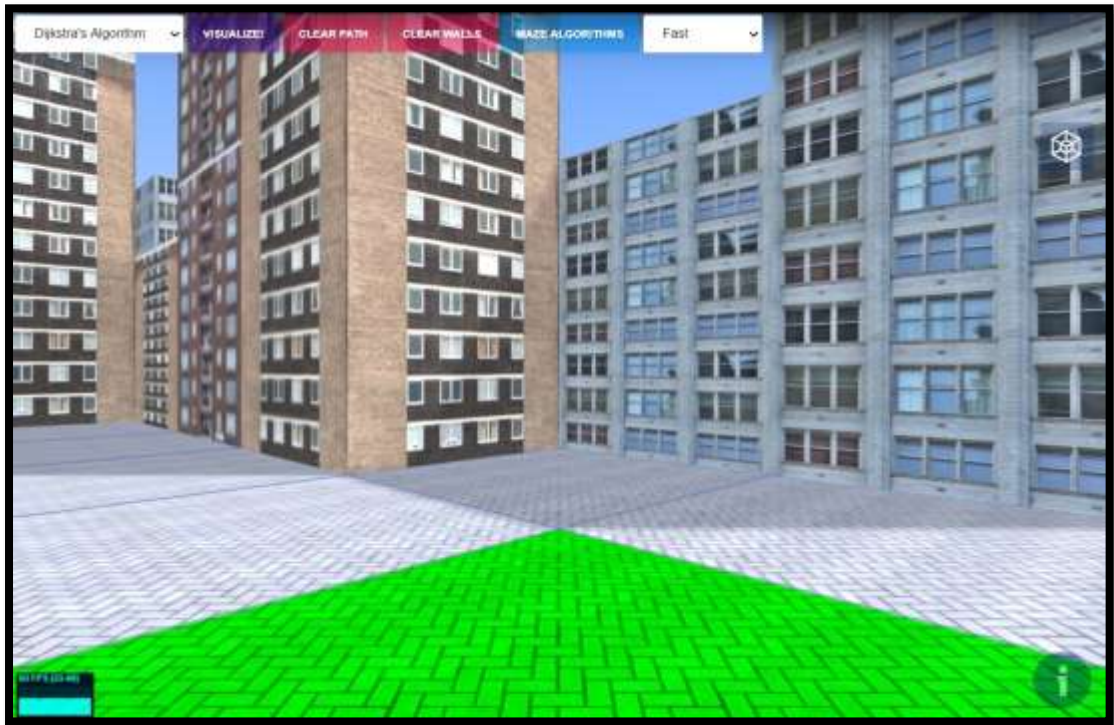


Figure 64: 3D Simulation of PathFinding Visualizer

I also plan to test more hybrid combinations of algorithms particularly Swarm algorithm's, Jump-Point Search, Orthogonal Jump Point Search and Trace Search to test these out in different set of environments. We have used Manhattan and Euclidean Heuristics particularly in this project and would like to extent that to cover Octile and Chebyshev in the future.

Addition of more grid features to manipulate the nodes like allow/disallow diagonals, cross corners etc. to devise more interesting patterns of these algorithms can enable better understanding of the algorithms.

A possible future task is to conduct a comprehensive experiment with additional algorithms. The algorithms tested in this project are just a sample of the most common algorithms. In the standard ones, the selected algorithms differ in the expected results.

Another related future task could be to compare one-dimensional algorithms such as comparing advanced A* focusing on improved memory performance and Theta* for improved memory efficiency. The only heuristic tested in this Euclidean thesis of all algorithms except IDA* as the calculation time has increased significantly with this heuristic. Instead, IDA* used octile to calculate its path.

Future work could be to magnify this and explore how different algorithms are

affected by changing heuristics. In the case of HPA *, only one batch size (16 by 16) was used. If the size of the collection is large, the estimated results will likely vary, especially if the HPA * pre-calculated is offline.

Also, we did not work in dynamic environments with moving targets or moving obstacles. It was initially discussed to be included but due to time was withdrawn.

Another area to consider is the representation of a three-dimensional map instead of the two examined in this thesis. The grid used in the maps in this thesis uses only the octile grid. With more representation of the map the calculated results may be different.

In this project, multi-threading was not used in computer algorithms. There are algorithms that can be very useful in string calculation. HPA * can benefit from calculating its acquisition in collections. Algorithms can be used both on the CPU and GPU. Due to previous CPU programming knowledge and lack of GPU programming experience, this project only focuses on CPU usage. One thing that can be further investigated is the use of algorithms on both the CPU and GPU to compare how they move.

5.3 Application Contributions

The major uses of algorithms and methods of finding methods today include:

World of Video Games and Game Theory

Now, the most common use of navigation is in video games, where the computer must direct the opponents to the map which updates dynamically. This is a very complex application of finding methods again we must consider that some methods are less attractive than others, in spite of the length of time it takes (some are highly dangerous) and dealing with many variables. In some cases, these variables are also under the control of the computer, so an improved system may need to consider 10 different businesses and move them all in their direction in a way that minimizes total delays. (This becomes even more difficult when you consider the key components of the unit and other features.)

Simulating hard to reach environments

The growing, but exciting, use of findings is in exploring hard-to-reach / dangerous areas. These were places that people could not easily navigate, but the robots could have explored their advantages if they did not get lost or hit a rock. This is particularly evident in the exploration of other planets, as NASA travels sending robots rather than humans into space.

Interestingly, there is also the impetus for robots to explore the world's most inaccessible places, such as deserts and valleys, seeking exciting and confusing life. Getting algorithms play a role in the development of these robots.

Commercial Use

There is a small market for robots carrying industrial units, office buildings, and other workplaces, which saves labor costs otherwise is spent on paying people to carry goods and these can save people from the monotonous and simple repetitive tasks. There is also emerging market like Ola, Uber Services which require accurate real time visualization.

Logistic and Transport Activities

The problem of finding a way encompasses the condition of multidisciplinary networks, which aims to improve traffic efficiency, accuracy, and cost of the entire transport network. There are still many challenges to find algorithmic improvements to meet real-world needs that include multiple goals or multiple goals, multiple approaches involving very different networks, multiple destinations, time-based planning, traffic speed, real-time planning, and so on. In real-world use, the complexity of the economic, social, and environmental environment requires search results for more than one factor, other than distance. Tasks dealing with single-purpose problems, defined by shortcut algorithms in previous sections, do not meet the requirement. Finding multiple conditional methods thus turns into easily accessible and feasible. Generally, most shortcut algorithms can only deal with a single network with one condition, far from reality. Even multiple conditioning solutions are automatically integrated and treated as a single condition in each network. To date there is no solution for calculating a complete route that connects two different networks, each with a set of different conditions. For example, the travel network needs to consider the level of difficulty posed by the slope, the roughness of the road, and the level of closed lanes to avoid rain, while the traffic network is more focused on time, distance, and transportation costs. Currently, many commercial and government applications such as Google, Baidu, Bing tend to offer only one network-based solution.

With improved transportation and more travel options like rail, road, air etc., there is a growing need for more information to meet the diverse needs of today's commuters. To have a complete solution that combines two or more special networks, then the algorithm needs to consider the dependability of all networks before a better solution to tackle all related problems can be devised.

Applications of the path finding visualizer:

- It can be used as a E learning tool to understand Algorithms.
- It is used in finding Shortest Path.
- It is used in the telephone network.
- It is used in IP routing to find Open shortest Path First.
- It is used in geographical Maps to find locations of Map which refers to vertices of graph.
- We can make a GPS system which will guide you to the locations.
- Search engine crawlers are used BFS to build index. Starting from source page, it finds all links in it to get new pages.
- In peer-to-peer network like bit-torrent, BFS is used to find all neighbor nodes.
- As users of wireless technology, people demand high data rates beyond Gigabytes per second for Voice, Video, and other applications. There are many standards to achieve data rates beyond GB/s. One of the standards is MIMO (Multi input Multi output). MIMO employs K-best Algorithm (which is a Breadth-First Search algorithm) to find the shortest partial Euclidean distances

REFERENCES

- [1]. Bassat Levy, R. B. and Ben-Ari, M. , ITiCSE '07, Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education, Dundee, Scotland. ACM Press., 2007, pp. 120–150.
- [2]. Brown, M. and Sedgewick, R. , “Techniques for algorithm animation”, IEEE Software, Algorithm Animation. MIT Press., 1985, pp. 2:28-39.
- [3]. Yngvi., Halldorsson, K., “Improved heuristics for optimal Pathfinding on Game maps”, in AIIDE., 2006, pp. 9-14.
- [4]. Kai Li Lim, KahPhooi Seng, Lee Seng Yeong., “Uninformed pathfinding: A new approach, Expert systems with applications”., 2015, pp. 2722-2730.
- [5]. SilverDavid., “Cooperative Pathfinding and Artificial Intelligence”, in Interactive Digital Entertainment Conference, AIIDE., 2005, pp. 117-122.
- [6]. Krishnaswamy Nikhil., “Comparison of Efficiency in Pathfinding Algorithms in Game Development”, Retrieved from <https://via.library.depaul.edu/tr/10>
- [7]. Michael Moran Walden University., “On Comparative Algorithmic Pathfinding in Complex Networks for Resource-Constrained Software Agents”. Retrieved from https://scholarworks.waldenu.edu/dissertations?utm_source=scholarworks.waldenu.edu%2Fdissertations%2F3951&utm_medium=PDF&utm_campaign=PDFCoverPages, 2017, pp. 200-300.
- [8]. Victor Martell Aron Sandberg., “Performance Evaluation of A* Algorithms” Thesis no: BCS-2016-07, 2016.
- [9]. SidhuHarinder Kaur., “Performance Evaluation of Pathfinding Algorithms”. Electronic Theses and Dissertations. 8178. Retrieved from <https://scholar.uwindsor.ca/etd/8178> , 2020.
- [10]. Chan, Simon Yew Meng, et al. , “An experiment on the performance of shortest path algorithm.”, 2016, pp. 7-12.
- [11]. Madhumita Panda Abinash Mishra. , “A Survey of Shortest-Path Algorithms.”, in International Journal of Applied Engineering Research ISSN 0973-4562 Volume 13, Number 9, pp. 6817-6820, 2018.
- [12]. Robbi Rahim et al 2018 J. Phys.: Conf. Ser. 1019 012036
- [13]. Masilo Mapaila. University of Cape Town., “EFFICIENT PATH FINDING FOR TILEBASED 2D GAMES.” , 2012.
- [14]. Yap, P. , “Grid-Based Path-Finding.”, In: Cohen, R., Spencer, B. (eds) Advances in Artificial Intelligence. Canadian AI 2002. Lecture Notes in



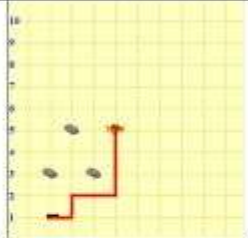
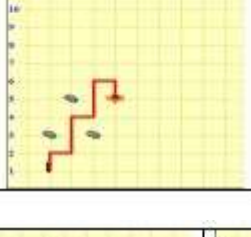
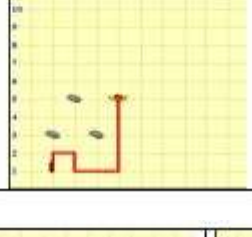
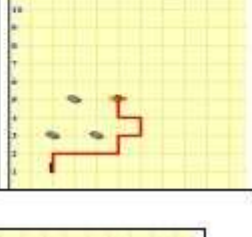
Computer Science(), vol 2338. Springer, Berlin, Heidelberg. Retrieved from https://doi.org/10.1007/3-540-47922-8_4

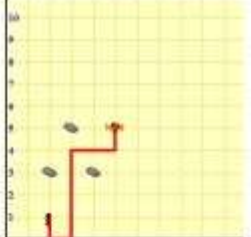
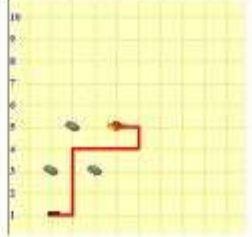
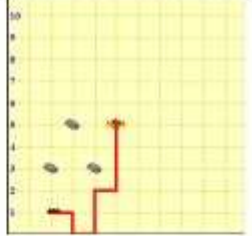
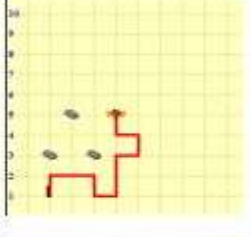
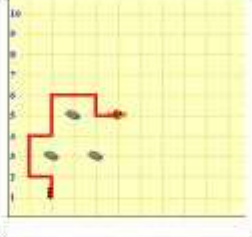
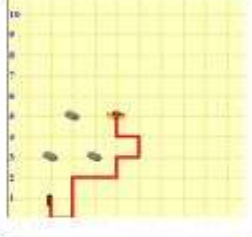
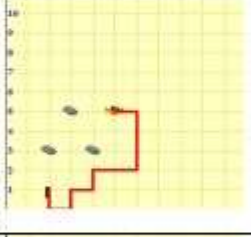
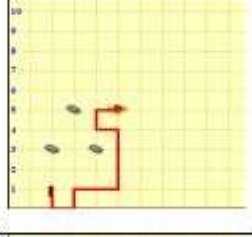
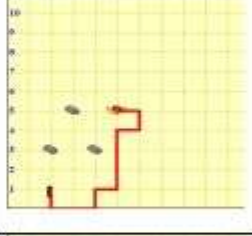
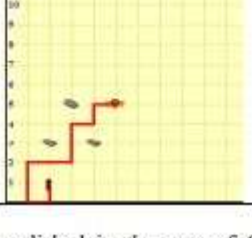
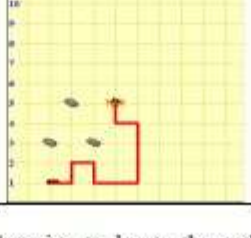
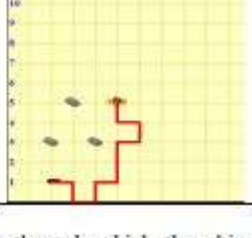
- [15]. Lee, W., Lawrence, R. , “Fast Grid-Based Path Finding for Video Games.”, In: Zaïane, O.R., Zilles, S. (eds) Advances in Artificial Intelligence. Canadian AI 2013. Lecture Notes in Computer Science(), vol 7884. Springer, Berlin, Heidelberg. Retrieved from https://doi.org/10.1007/978-3-642-38457-8_9
- [16]. Pan, T.; Pun-Cheng, S.C. A Discussion on the Evolution of the Pathfinding Algorithms. Preprints 2020, 2020080627 (doi: 10.20944/preprints202008.0627.v1).
- [17]. Saif Ulla Shariff, M Ganeshan. , “A Path Finding Visualization Using A Star Algorithm and Dijkstra’s Algorithm”, in International Journal of Trend in Scientific Research and Development (IJTSRD) Volume 5 Issue 1, 2020.
- [18]. Nishant Kumar and Nidhi Sengar., “Pathfinder Visualizer of Shortest Paths Algorithms.”, in International Journal for Modern Trends in Science and Technology., 2020, pp. 6(12): 479-483.
- [19]. Aakansha N. TabhaneNikhil Likhar Kalyani Mohod Manali Kadbe, KDK College of Engineering. , “PATH FINDING VISUALIZER” Vol-7 Issue-3 2021 IJARIE-ISSN(O)-2395-4396, 2021, pp. 2395-4396.
- [20]. Thierry Thierry Okie Department, Minnesota State University Moorhead. “PATHFINDING VISUALIZER.”, 2017.
- [21]. Alex T Mathew et al 2021 J. Phys.: Conf. Ser. 1831 012008
- [22]. Pathfinding Grids. Retrieved from <https://www.redblobgames.com/pathfinding/grids/algorithms.html>

Appendix

1. BFS: The process of searching the solution search using the Breadth First Search algorithm are performed in the following Cartesian case:

Table 1. Solutions obtained

No	Steps	Few Solution Step		
1	7 Step Found 8 Solution			
2	9 Step Found 61 Solution			

				
3	11 Step Found 236 Solution			
				
				

Testing was accomplished in the area of Cartesian to locate the path through which the object produces some options that could be passed, with the most optimal solution consists of 7 steps.

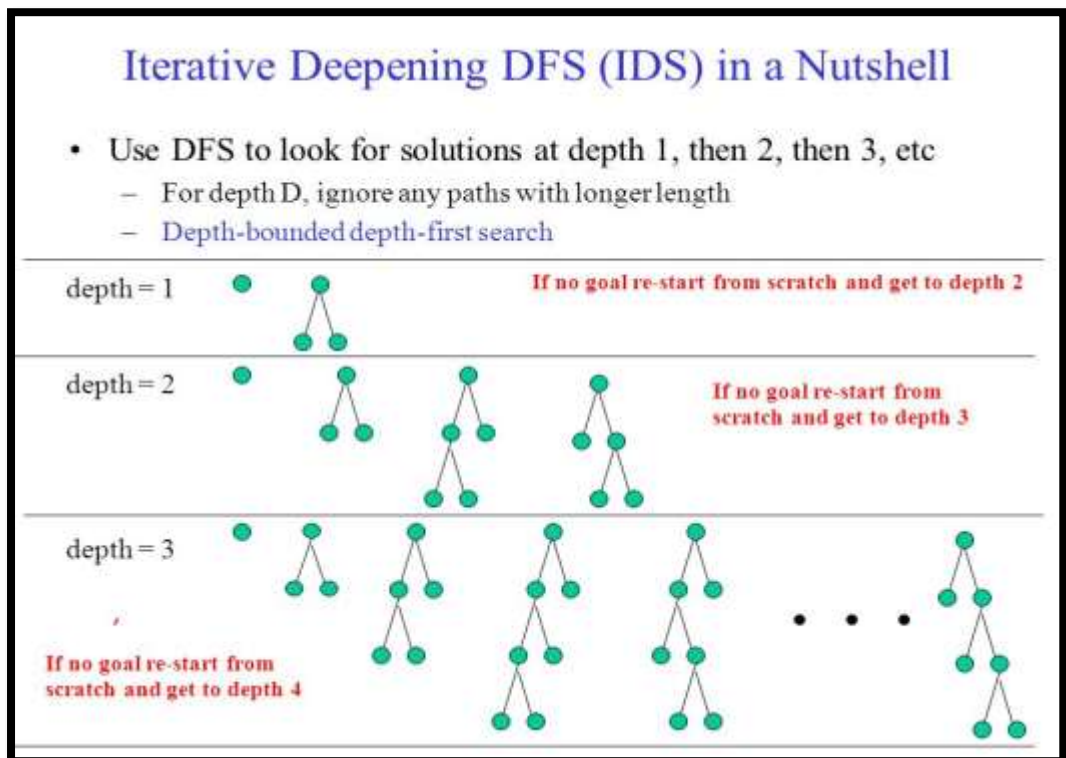
	List of pathfinding papers critically reviewed for thesis
1	A heuristic search algorithm with modifiable estimate
2	A combined tactical and strategic hierarchical learning framework in multi-agent games
3	A comparison between A* pathfinding and waypoint navigator algorithm
4	A comparison of high-level approaches for speeding up pathfinding
5	A Navigation meshes and real time dynamic planning for Virtual worlds
6	A formal basis for the heuristic determination of minimum cost paths
7	A game map complexity measure based on hamming distance
8	A heuristic for domain independent planning, and its use in an enforced hill-climbing algorithm
9	A hierarchical data structure for picture processing
10	A hierarchical data structure for representing the spatial decomposition of 3-D objects
11	A Comparative analysis of the algorithms for pathfinding in GPS systems
12	A hierarchical space indexing method
13	A note on two problems in connexion with graphs
14	A partial pathfinding using map abstraction and refinement
15	A polynomial-time algorithm for non-optimal multi-agent pathfinding.
16	An efficient memory bounded search method
17	A path planning algorithm for low-cost autonomous robot navigation in indoor environments
18	A*-based pathfinding in modern computer games
19	Accelerated A* Trajectory Planning: Grid- based Path Planning comparison
20	Adaptive A*

2.

Pathfinding algorithm efficiency analysis in 2D grid
Pathfinding and collision avoidance in crowd simulation
Pathfinding Design Architecture
Pathfinding in computer games
Pathfinding in partially explored games environments

3. Issues in experimental design: It cover the papers with experiments using only one or two type of maps, three or less map size variations, two or less obstacle density variation and no obstacle distribution. 2. Issues in data collection: It cover papers which collected data from 3 or less types of map and variations or data collected does not provide direct evidence supporting their claims like data of time consumption indirectly pointing to less memory consumption, no data for memory consumption. 3. Issues in data analysis: The analysis only provides average mean, median results and did not provide standard deviation, variance of the data.

4.IDA*: Working of IDA*



Pseudo-Code of IDA*

function IDA*(*problem*) **returns** a solution sequence

inputs: *problem*, a problem

static: *f-limit*, the current *f*- COST limit

root, a node

root \leftarrow MAKE-NODE(INITIAL-STATE[*problem*])

f-limit \leftarrow *f*- COST(*root*)

loop do

solution, *f-limit* \leftarrow DFS-CONTOUR(*root*, *f-limit*)

if *solution* is non-null **then return** *solution*

if *f-limit* = ∞ **then return** failure; **end**

function DFS-CONTOUR(*node*, *f-limit*) **returns** a solution sequence and a new *f*- COST limit

inputs: *node*, a node

f-limit, the current *f*- COST limit

static: *next-f*, the *f*- COST limit for the next contour, initially ∞

if *f*- COST[*node*] > *f-limit* **then return** null, *f*- COST[*node*]

if GOAL-TEST[*problem*](STATE[*node*]) **then return** *node*, *f-limit*

for each node *s* **in** SUCCESSORS(*node*) **do**

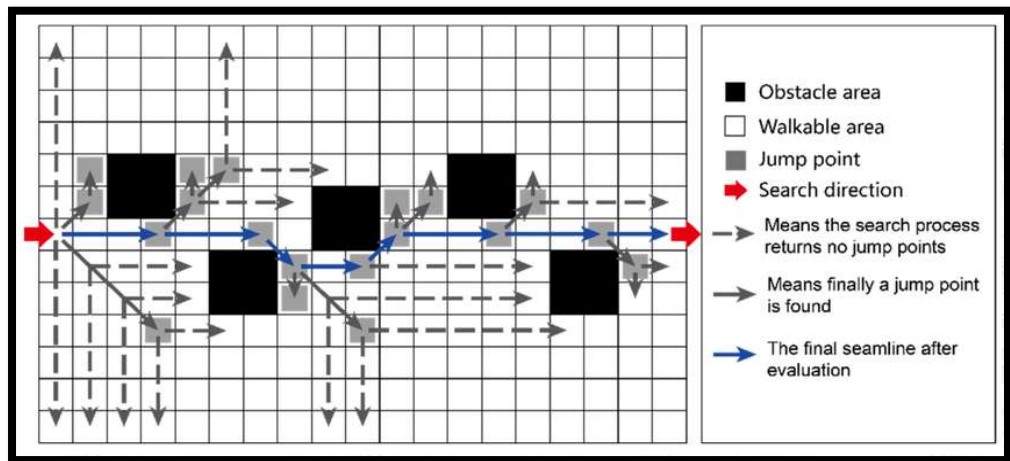
solution, *new-f* \leftarrow DFS-CONTOUR(*s*, *f-limit*)

if *solution* is non-null **then return** *solution*, *f-limit*

next-f \leftarrow MIN(*next-f*, *new-f*); **end**

return null, *next-f*

5. Jump Point Search: Working of JPS



```

function identifySuccessors(current:Node, start:Node, end:Node):Vector.<Node> {
    var successors:Vector.<Node> = new Vector.<Node>();
    var neighbours:Vector.<Node> = nodeNeighbours(current);

    for each (var neighbour:Node in neighbours) {
        // Direction from current node to neighbor:
        var dx:int = clamp(neighbour.x - current.x, -1, 1);
        var dy:int = clamp(neighbour.y - current.y, -1, 1);

        // Try to find a node to jump to:
        var jumpPoint:Node = jump(current.x, current.y, dx, dy, start, end);

        // If found add it to the list:
        if (jumpPoint) successors.push(jumpPoint);
    }

    return successors;
}

```

Pseudo Code of JPS

```

function jump(cX:int, cY:int, dx:int, dy:int, start:Node, end:Node):Node {
    // cX, cY - Current Node Position, dx, dy - Direction

    // Position of new node we are going to consider:
    var nextX:int = cX + dx;
    var nextY:int = cY + dy;

    // If it's blocked we can't jump here
    if (_world.isBlocked(nextX, nextY)) return null;

    // If the node is the goal return it
    if (nextX == end.x && nextY == end.y) return new Node(nextX, nextY);

    // Diagonal Case
    if (dx != 0 && dy != 0) {
        if (/*... Diagonal Forced Neighbor Check ...*/) {
            return Node.pooledNode(nextX, nextY);
        }

        // Check in horizontal and vertical directions for forced neighbors
        // This is a special case for diagonal direction
        if (jump(nextX, nextY, dx, 0, start, end) != null ||
            jump(nextX, nextY, 0, dy, start, end) != null)
        {
            return Node.pooledNode(nextX, nextY);
        }
    } else {
        // Horizontal case
        if (dx != 0) {
            if (/*... Horizontal Forced Neighbor Check ...*/) {
                return Node.pooledNode(nextX, nextY);
            }
        }
        // Vertical case
    } else {
        if (/*... Vertical Forced Neighbor Check ...*/) {
            return Node.pooledNode(nextX, nextY);
        }
    }
}

// If forced neighbor was not found try next jump point
return jump(nextX, nextY, dx, dy, start, end);
}

```

6.

HPA* (Hierarchical Path-Finding A*), a hierarchical approach for reducing problem complexity in pathfinding on grid-based maps. This technique abstracts a map into linked local clusters. At the local level, the optimal distances for crossing each cluster are pre-computed and cached. At the global level, clusters are traversed in a single big step. A hierarchy can be extended to more than two levels. Small clusters are grouped together to form larger clusters. Computing crossing distances for a large cluster uses distances computed for the smaller contained clusters