©2022 International Transaction Journal of Engineering, Management, & Applied Sciences & Technologies

ISSN 2228-9860 eISSN 1906-9642 CODEN: ITJEA8

TuEngr Group

International Transaction Journal of Engineering, Management, & Applied Sciences & Technologies

http://TuEngr.com

Implementation and Evaluation of an Optimal Algorithm for Neural Networks Association in Machine Learning

S. Balapriya[1*], **N. Srinivasan**[2]

[1]*Sathyabama University, Chennai, INDIA.*
[2] *Department of CSE, Rajalakshmi Engineering College, Chennai, INDIA.*
**Corresponding Author (Email: s.balapriya@gmail.com).*

Paper ID: 13A6K
Volume 13 Issue 6
Received 25 January 2022
Received in revised form 29 April 2022
Accepted 05 May 2022
Available online 12 May 2022

Keywords:
A* algorithm; Pathfinding algorithm; Parallel computing; Modular programming Architecture; modularity.

Abstract

This study attempts to investigate the efficient and simple incorporation of a basic pathfinding algorithm that is suitable for students, graduates, freshmen, and scholars for exploring and upgrading research-related studies and theories. The aim of writing this paper is to present the fundamentals of AI path-finding algorithms most simply and effectively. The algorithm takes inspiration from the popular pathfinding A* algorithm used widely in modular programming architecture in association with parallel computing. This algorithm is well suitable for games or game levels created with the modular designing methodology or any application using modularity as its core foundation. The algorithm enables navigation of any game entity within the game world on a plane surface with obstacle detection.

Disciplinary: Artificial Intelligent

Cite This Article:

Balapriya, S., N., Srinivasan. (2022). Implementation and Evaluation of an Optimal Algorithm for Neural Networks Association in Machine Learning. *International Transaction Journal of Engineering, Management, & Applied Sciences & Technologies, 13*(6), 13A6K, 1-11. http://TUENGR.COM/V13/13A6K.pdf DOI: 10.14456/ITJEMAST.2022.116

1 Introduction

Traditional software were using programs that are written for serial computation. A serial computation is a whole problem divided into discrete chunks of sub-problems that are sequentially executed on a single processor. The main problem of serial computing is time-consuming due to the execution of one problem at a given time. Parallel computing emerged in order to replace those problems with serial computing where we use more than one processor to execute instructions faster. Parallel computing enables solving problems in a concurrent method. In order to execute the tasks concurrently, an overall control mechanism is built to control the system [1].

Parallel computers are not only distinguished from a hardware perspective but also from functional units. The various functional units are cache with different levels like L1, L2, etc, branch, prefetch, decoding, GPU and many more. Multiple threading can also be a part of software functionality to support parallel processing. To create bigger multiple processor clusters, networks connect several stand-alone computers, often known as nodes. Today the majority of the global total uses PCs with several processors known as supercomputers which are hardware clusters from well-known manufacturers. We chose distributed processing because it is better suited for modelling, simulating, and interpreting complex real-world phenomena such as galactic changes, structural geology, meteorology, transport, and many others. Parallel computing, in general, saves time and money in solving complex and difficult problems. Parallelism is the future of computing.

2 Literature Review

A pathfinding algorithm is used to solve the shortest path to travel from the given points of source and the destination. We try to find all the possible paths in a programmatic way and out of all the paths we select the best path based on some criteria. These criteria could be cost, time, distance, etc. In graph theory, the shortest path is to find the best and optimal path between two vertices. Pathfinding algorithms are useful in many applications like Google maps, satellite navigations, and routing packets over the internet [2]. This idea of finding the shortest path is not only applicable in real life but also in the gaming world.

The biggest challenge in game development is designing realistic Artificial Intelligence AI and movements for these AIs. Pathfinding strategies help us to solve this challenge by finding the next path within the game world [3]. This system takes the starting and endpoint called the source and destination and finds a series of points in between these points to take the optimal route. These pathfinder logics work based on a condition that requires accepting this path to be the best shortest path of all. The most popular path-finding algorithms are.

Breadth First Search - BFS Breadth First Search is usually practiced in graphs and trees [4]. Mainly used in backtracking of any regular traversal, network analysis and finding places nearby in GPS. It explores equally in all directions but doesn't promise to find the best path.

Depth First Search - DFS Depth First Search is usually practiced in Graphs and Trees. Mainly used in backtracking of topological ordering, creating decision trees, and discovering a solution approaches with various leveled alternatives. Depth First Search is not suitable for finding the shortest path but when it finds a path it is considered as its goal. More suited for games and puzzles, for example, mazes.

Dijkstra - Instead of investigating all different possibilities equally, Dijkstra's Algorithm is primarily used to choose which directions to explore. It finds the best path involving the lowest cost of traveling. Dijkstra’s algorithm finds the shortest path from a starting node to every other node in the graph/tree. Dijkstra’s Algorithm can find paths to all locations from the start node.

There could be an ‘n’ number of destinations for this algorithm [5]. Here we optimise which approach to explore or the order of the paths to be explored.

A*(A-Star) - Dijkstra's Algorithm is efficient for finding the shortest path, however, it spends time examining directions that aren't necessary. A* is a single-destination of Dijkstra's Algorithm that has been updated.

A* finds the best path from one location to another on a map. It prioritizes paths that look closer to the goal. It is the most popular and widely used algorithm known to be the industry standard algorithm for finding the shortest path in both applications and games. In games, most of the AIs use this algorithm to determine the best path.

The A* search algorithm is one of the finest and almost always used path-finding strategies. It's a brilliant technique that many games and web-based navigation apply to locate the efficient way quickly (approximation).

To find the approximate shortest path or best optimal path, an A* algorithm is very effective in both real-time scenarios and in games [6]. Especially when you have a map to traverse. Consider the following image with source point, destination point and several obstacles:

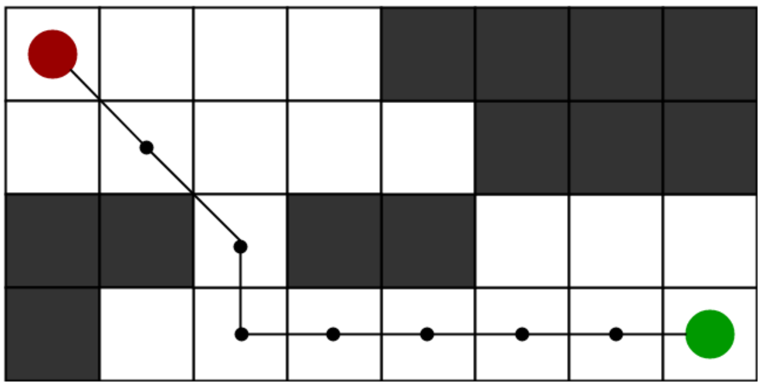


Figure 1: Obstacle.

In Figure 1, the path from source to destination is found with several obstacles. The algorithm takes the cost or considers other conditions to take the shortest path and takes every step depending upon these criteria.

The game is prepared before the algorithm is executed; meaning the source, destination and obstacles are created or known directly [7]. The entire map is also divided into nodes which are used to record the progression of the game element (usually AI) in the map during the search. In addition to all this, it also holds 3 different attributes called i, l, and n,

where

1. i - the cost of getting from the start node to the current node
2. l -heuristic cost from the current node to the goal node
3. n -the sum of g and h best estimate of the cost of the path going through the current node.
4. The purposes of i, l, and n are to quantitatively analyse how promising a path is up to

the present node.

5. Finally, A* maintains two lists, an Open and a Closed list. The Open list contains all the nodes in the map that have not been fully explored whereas the Closed list consists of all the nodes that have been fully explored.

The pseudo-code for the A* Algorithm is as follows:

```
1. Let V= starting point.
2. Assign i, l and n values to V.
3. AddVto the Open list. At this point,V is the only node on the Open list.
4. LetQ = the best node from the Open list (i.e. the node that has the lowest ivalue).
    a. If Qis the goal node, then quit – a path has been found.
    b. If the Open list is empty, then quit – a path cannot be found
5. Let S = a valid node connected to Q.
    a. Assign i, l, and n values toS.
    b. Check whether S is on the Open or Closed list.
        i. If so, check whether the new path is more efficient (i.e. has a lower fvalue).
            1. If so update the path.
        ii. Else, add S to the Open list.
    c. Repeat step 5 for all valid children of Q.
6. Repeat from step 4.
```

3 Creation of ObtuseDia Algorithm

As we all know, A* is the widely suitable algorithm for finding the shortest path in games and any application which requires finding the best optimal path. It works well in all cases to handle big data with maps and terrains [8]. In case we need a simple algorithm that works well in the same scenario without much complication in implementing and handling. This idea has led me to think of an algorithm without involving a cost element but simply considering the number of steps or nodes we travel to reach the shortest path.

Table 1: Terminologies

Terms	Meanings
Actual Map	The map of the game world
Trace Map	The map where we traverse
SB	Source Box
DB	Destination Box (always 0,0)
CB	Current Box where the box recently visited
NB	New/Next Box denotes the next possible box where we are likely to visit
r	row number to be checked for New/Next Box from Current Box
c	the column number to be checked for New/Next Box from Current Box
CB(r,c)	Current Box row & col number
NB(r,c)	New/Next Box row & col number
R	New/Next Box row value [up(-1), down(+1)]
C	New/Next Box column value [left(-1), right(+1)]
NB(R,C)	Value of New/Next Box row & col
SUM(R,C)	sum value of NB (R,C)
STEPS	no. of steps taken from Source Box to Current Box in order to reach the Destination Box

Naming the algorithm was really challenging since there could be a lot of algorithms in the market with usual technical names relating to pathfinding [9]. Observing the features and properties of this algorithm, the basic traversing criteria is to satisfy a diagonal path to reach from the source to the destination. Moving diagonally is considered to take less number of steps toward the destination. Hence the term 'diagonal' must be in the part of the name. The next step is to consider the direction of movement. Here also the algorithm always marks the destination as (0,0) which means the source will be some (x,y) and we keep moving from source to destination only in the upward direction (most of the time).

Considering the style and direction of movement this algorithm was given many names like Dia-Angle, Slant Tracing, Back Shadow, Obtuse-Angle and a few more. Finally, it is named "Obtuse-Dia" meaning moving throughout in an obtuse direction to find the best and optimal path of traversing.

3.1 Basic Logic

The fundamental of this algorithm is to diagonally move in the upward direction to reach from the source to the destination. According to the concept, below is the working logic:

There are 2 maps in this algorithm:

Actual Map - Actual Map is the original map where all the game entities are present and the game world exists.

Trace Map - After retrieving the Source and Destination from the Actual Map, a part of the Actual Map is taken to find the optimal path between them. This map is called the Trace Map. The Trace map is relatively less than the Actual Map.

3.1.1 Properties of the Actual Map

Figure 2 shows original actual map with the properties

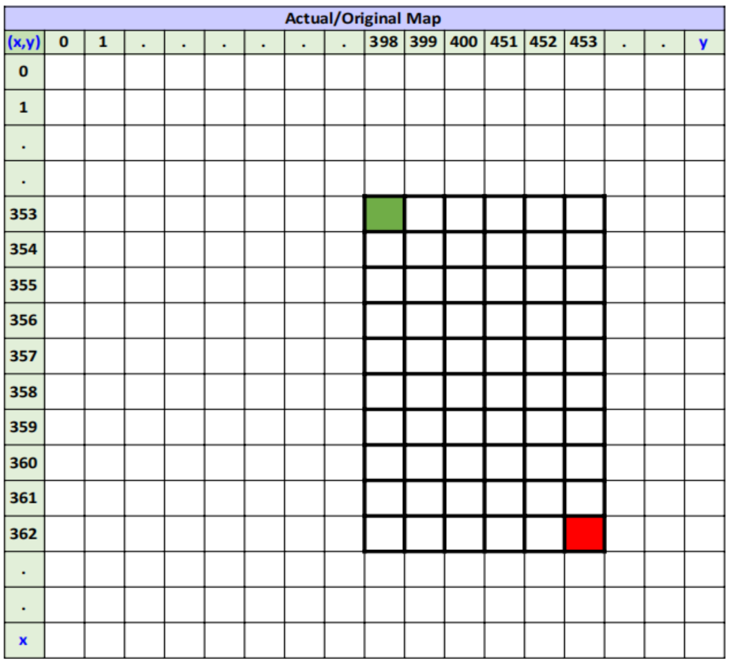


Figure 2: Actual/Original Map

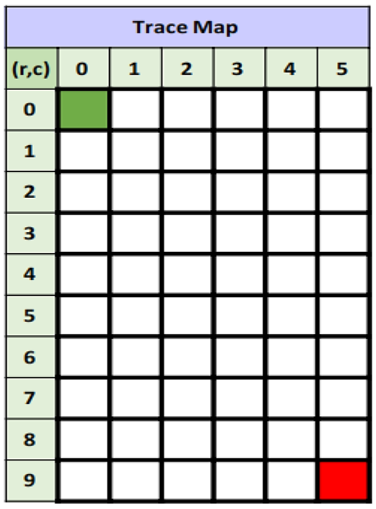
- The Actual Map space is converted as 2D TILES.
- Tiles are marked between TILE(0,0) to TILE(x,y), where x,y is the size of the map in rows and columns within the Actual Map.
- The Source Tile is the tile where the game entity is present, usually denoted in RED color. In this example Source Tile(362,453)
- The Destination Tile is the tile where the game entity would like to move, usually denoted in GREEN color. In this example Destination Tile(353,398)

3.1.2 Properties of the Trace Map

Figure 3 shows the trace map with the properties

- The Trace map space is converted as BOXES.
- Boxes are marked between BOX(0,0) to BOX(r,c) where r,c is the size of the map in rows and columns within the Trace map.
- The Source Box is the place where the game entity is present, usually denoted in RED color. In this example Source Box(9,5)
- The Destination Box is the place where the game entity wants to move. In the Trace Map, the Destination Box will always be Box(0,0) denoted in GREEN color.

1. We aim at finding the optimal path from the Source Box to the Destination Box in the Trace Map. We assume that the cost of travelling from one box to another is zero and therefore we will not consider the element called cost. Instead of the cost we take a path that can reach the Destination Box by counting the number of steps.

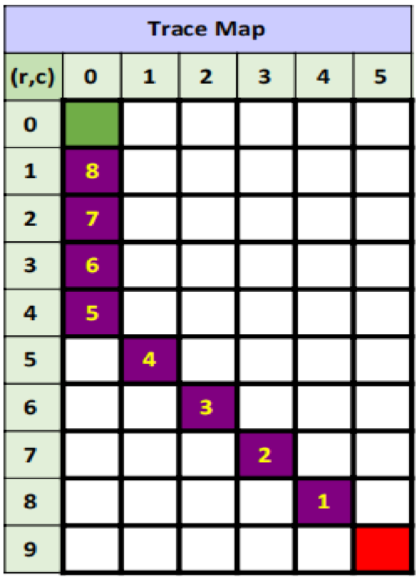


Trace Map						
(r,c)	0	1	2	3	4	5
0	🟩					
1						
2						
3						
4						
5						
6						
7						
8						
9						🟥

Figure 3: Trace Map

2. The path that takes a minimal number of steps is the best and optimal path. In that scenario, in each iteration, we will be moving 1 step by row and column to get close to the Destination Box. This movement would mostly be the diagonal movement within the Trace Map.

3. If there is no diagonal box available or the diagonal box is filled with obstacles, we choose to either move 1 step to a row or column. If moving 1 step closer by row or column or both is not possible we will try to re-route and take a long path.



Trace Map						
(r,c)	0	1	2	3	4	5
0	🟩					
1	8					
2	7					
3	6					
4	5					
5		4				
6			3			
7				2		
8					1	
9						🟥

Figure 4: Trace Map for optimized path

4. For example, in the above example the optimal path found by the algorithm ObtuseDia() is given below :
5. In this example after step 5, we do not have any diagonal box so we move 1 step by row to reach the Destination Box. Here we reach the Destination Box in 8 steps which are considered the best and optimal path.
6. If we take any other route also, you may reach the Destination Box with 8 or greater than 8 steps only. But not less than 8 steps.
7. Once we get this path, we apply the same path in the Actual Map to move the game entities to reach the Destination Tile from the Source Tile.

4 Case Studies

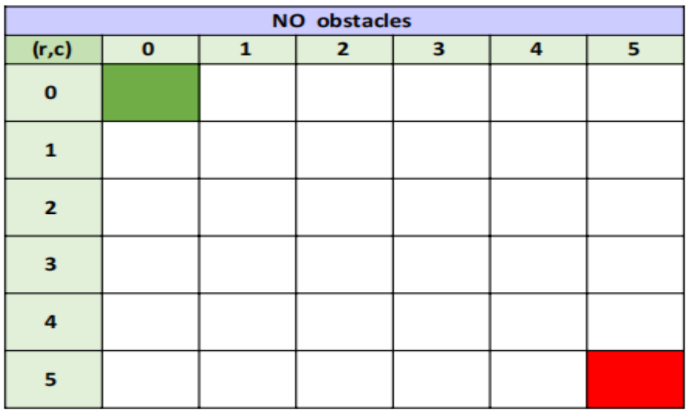
Case Study 1: Assume there are no obstacles in the Trace Map (Figure 5). Source Box(5,5) is marked as RED and Destination Box(0,0) is marked as GREEN.

Solution: In this scenario, in each iteration we try to find the next best box to move so that we get close to the Destination Box. Source Box is SB, Destination Box is DB, Current Box CB is the box where the game entity is present, NB New Box is the box we are considering moving in this iteration. The R and C store the direction value from the CB. R and C may have the values -1, 0, +1. This algorithm calculates the Sum(R,C) before deciding the NB.

if Sum(R,C) is -2 or -1, it is considered the best box to move.

If Sum(R,C) is 0, then this box is considered to be the medium optimal choice.

If Sum(R,C) is greater than 0, then this box is considered to be the worst optimal choice taken since there is no other option.

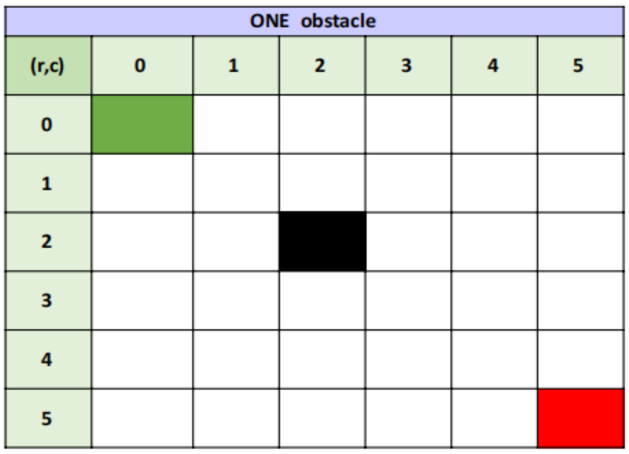


NO obstacles						
(r,c)	0	1	2	3	4	5
0	🟩					
1						
2						
3						
4						
5						🟥

Figure 5: No obstacles

Look at the below calculations for this scenario. Hence the best or optimal path is found to be the following :

Case Study 2: Assume there is one obstacle in the Trace Map (Figure 6). Source Box(5,5) is marked as RED, Destination Box(0,0) is marked as GREEN, Box(2,2) is marked as Obstacle in BLACK.



ONE obstacle						
(r,c)	0	1	2	3	4	5
0	🟩					
1						
2			⬛			
3						
4						
5						🟥

Figure 6: One obstacle.

Solution: In this scenario, in the first iteration, we take the New Box(4,4) and in the second iteration New Box(3,3). In the third iteration, we find an obstacle in Box(2,2) hence, we have to take a different path. Let us see what are the different paths we can take after identifying an obstacle:

Now, we have 6 options or ways to travel on the Trace map to reach the Destination Box (Figure 7). The different possible boxes that we can move from the current box(3,3) are highlighted as A, B, C, D, E, and F.

Box A(3,2) - This box moves forward from the Current Box by 1 column

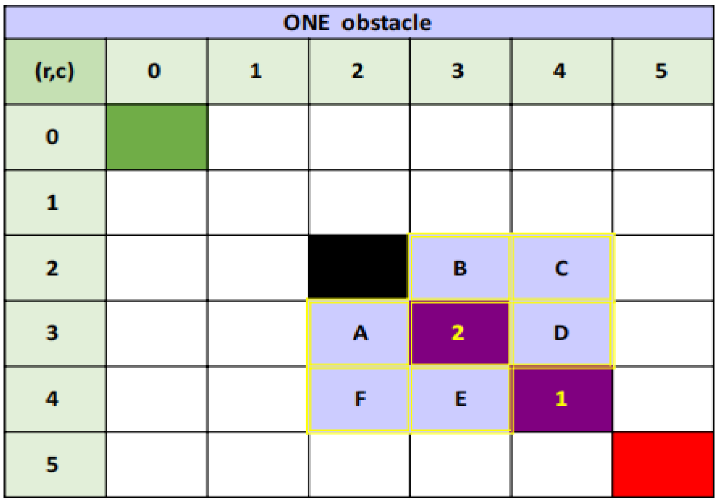
Box B(2,3) - This box moves forward from the Current Box by 1 row

Box C(2,4) - This box moves forward by 1 row but backward by 1 column

Box D(3,4) - This box moves backward by 1 column

Box E(4,3) - This box moves backward by 1 row

Box F(4,2) - This box moves backward by 1 row but forward by 1 column.

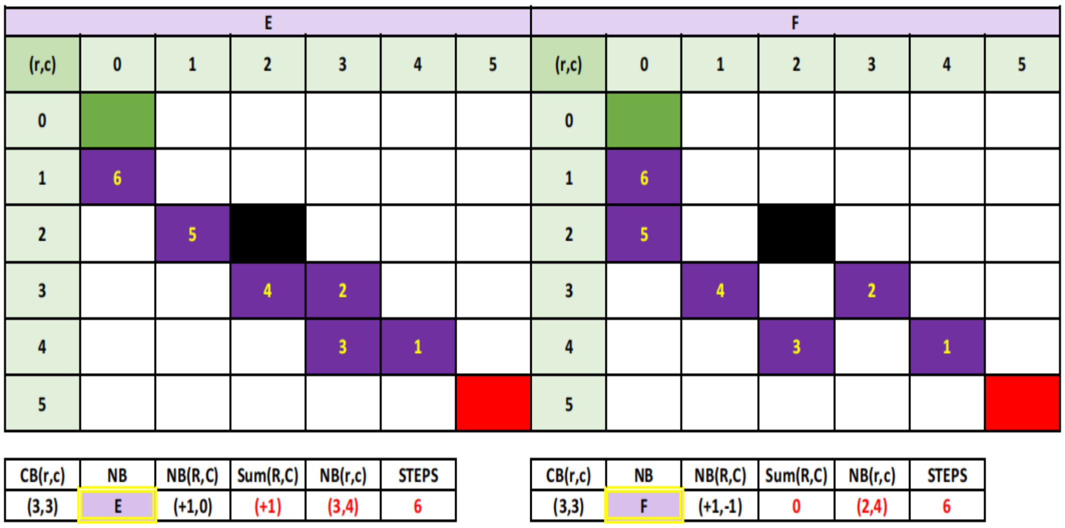


ONE obstacle						
(r,c)	0	1	2	3	4	5
0	🟩					
1						
2			⬛	B	C	
3			A	2	D	
4			F	E	1	
5						🟥

Figure 7: One obstacle

Let us consider Boxes A and B. In both cases, after evading the obstacle box, we can reach the destination in 5 steps.

In case we take the Boxes C, D, E, and F, after evading the obstacle box we will be taking a total of 6 steps to reach the Destination Box.



E						
(r,c)	0	1	2	3	4	5
0	🟩					
1	6					
2		5	⬛			
3			4	2		
4				3	1	
5						🟥

F						
(r,c)	0	1	2	3	4	5
0	🟩					
1	6					
2	5		⬛			
3		4		2		
4			3		1	
5						🟥

CB(r,c)	NB	NB(R,C)	Sum(R,C)	NB(r,c)	STEPS
(3,3)	E	(+1,0)	(+1)	(3,4)	6

CB(r,c)	NB	NB(R,C)	Sum(R,C)	NB(r,c)	STEPS
(3,3)	F	(+1,-1)	0	(2,4)	6

Figure 8: ObtuseDia algorithm description

Since the motive of our algorithm is to reach a short distance by taking less number of steps to traverse the Trace map, our algorithm will take either the route of A or B according to the order of execution.

The ObtuseDia algorithm is implemented as a prototype to check the functionality with sample data (Figure 8). This prototype is working fine with any valid data given to it. The output of the execution of the prototype is given below:

Consider the case without obstacles: In Figure 9, the trace map is highlighted in WHITE color. And the shortest path between the source and destination is marked in PURPLE color.

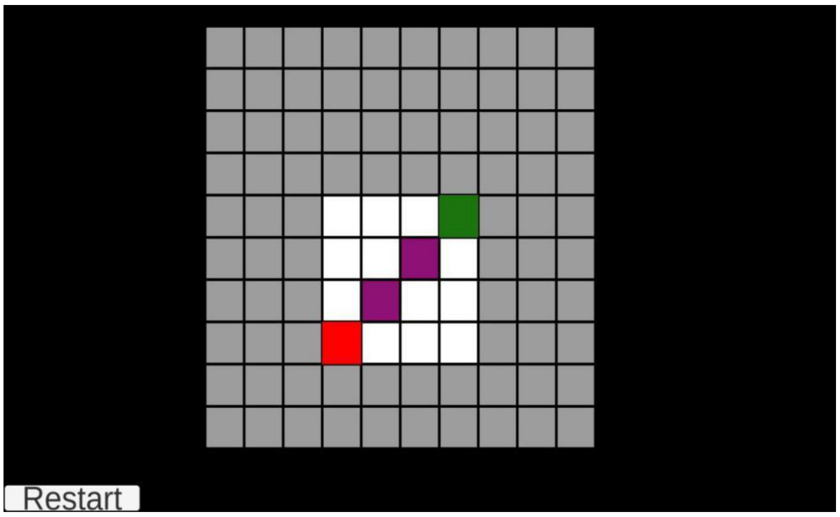


Figure 9: Trace Map (without obstacles)

Consider the case with obstacles: In Figure 10, the trace map is highlighted in WHITE color. And the shortest path between the source and destination is marked in PURPLE color.

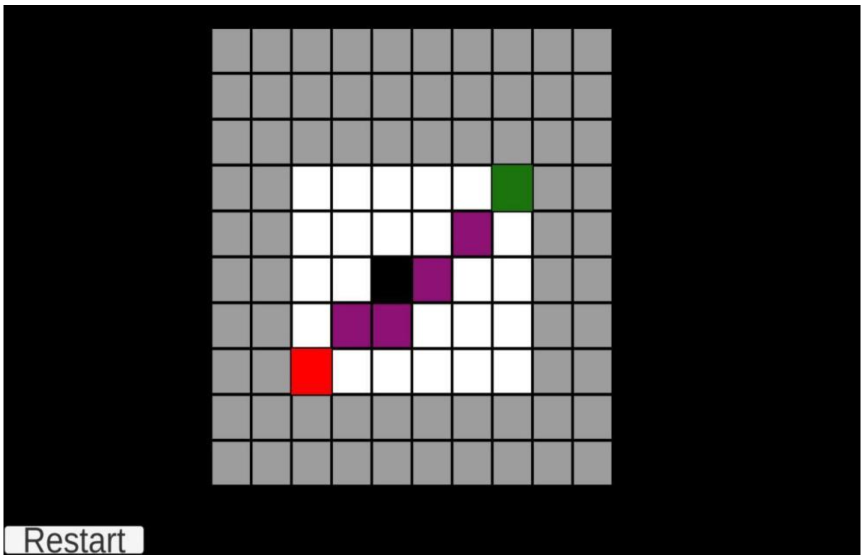


Figure 10: Trace Map (with obstacles)

5 Conclusion

This study investigated the efficient and simple incorporation of a basic path finding algorithm that is suitable for students, graduates, freshmen and scholars for exploring and upgrading research-related studies and theories. The algorithm takes inspiration from the popular pathfinding A* algorithm used widely in modular programming architecture in association with parallel computing. This algorithm is well suitable for games or game levels created with the modular designing methodology or any application using modularity as its core foundation. The algorithm enables navigation of any game entity within the game world on a plane surface with obstacle detection.

6 Availability of Data and Material

Data can be made available by contacting the corresponding author.

7 References

[1] Beamer, S., Asanovic, K and Patterson, D. (2012). Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (pp. 1-10). IEEE.

[2] Luo, L., Wong, M and Hwu, W. M. (2010). An eective GPU implementation of breadthfirst search. In *Design Automation Conference* (pp. 52-55). IEEE.

[3] Awerbuch, B. (1985). A new distributed depth-first-search algorithm. *Information Processing Letters*, 20(3), 147-150.

[4] Holzmann, G. J., Peled, D. A and Yannakakis, M. (1996). On nested depth first search. *The Spin Verication System*, 32, 81-89.

[5] Duchon, F., Babinec, A., Kajan, M., Beno, P., Florek, M., Fico, T and Jurisica, L. (2014). Path planning with modied a star algorithm for a mobile robot. *Procedia Engineering*, 96, 59-69.

[6] AlShawi, I. S., Yan, L., Pan, W and Luo, B. (2012). Lifetime enhancement in wireless sensor networks using fuzzy approach and A-star algorithm.

[7] Yao, Junfeng, Chao Lin, Xiaobiao Xie, Andy JuAn Wang, and Chih-Cheng Hung. Path planning for virtual human motion using improved A* star algorithm. In 2010 Seventh international conference on information technology: new generations, pp. 1154-1158. IEEE, 2010.

[8] Noto, M abd Sato, H. (2000, October). A method for the shortest path search by extended Dijkstra algorithm. In Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics.'cybernetics evolving to systems, humans, organizations, and their complex interactions'(cat. no. 0 (Vol. 3, pp. 2316-2320). IEEE.

[9] Deng, Y., Chen, Y., Zhang, Y and Mahadevan, S. (2012). Fuzzy Dijkstra algorithm for shortest path problem under uncertain environment. *Applied Soft Computing*, 12(3), 1231-1237.

S. Balapriya is a student at the Faculty of Computing, Sathyabama University, Chennai, TamilNadu. She got a Master's degree in Computer Application. Her researches are on Computer Game Technologies.

Dr. Srinivasan N is a Professor at the Department of Computer Science, Rajalakshmi Engineering College, Chennai, TamilNadu. He got his Master’s and Ph.D. degrees in Computer Science. His research focuses on Cloud Technology, Data Mining and Testing.