Implementing Pathfinding Algorithms

Group Members:

19BCI0167 (Shiv Thaker)

Report submitted for the
Final Project Review

Submitted By
Name: Shiv Thaker
Reg.no.: 19BCI0167

Course Code: CSE3013 – Artificial Intelligence

Slot: F1 + TF1

Professor: AKILA VICTOR

Date: 3rd June 2021

Declaration by Author

This is to declare that this report has been written by us. No part of the report is plagiarized from other sources. All information included from other sources has been duly acknowledged. We aver that if any part of the report is found to be plagiarized, we shall take full responsibility for it.

Shiv Thaker 19BCI0167

Vellore Institute of Technology

Date: 3rd June 2021

Acknowledgement

I would like to express our special thanks of gratitude to my professor Akila Victor as well as my institute Vellore Institute of Technology that gave me the opportunity to do this wonderful project on the topic Implementing Pathfinding Algorithms. This helped me in gaining a tremendous amount of knowledge through Research and Implementation which helped me improve my skill set as well as deepen my knowledge in the subject.

Secondly, I would also like to thank my parents and friends who kept me motivated with their constant support and helped me complete the project within the given timeframe.

Abstract

Pathfinding algorithms are usually an attempt to solve the shortest path problem in graph theory, to try to find the best path given a starting point and ending point based on some predefined criteria. Path finding algorithms are important because they are used in applications like google maps, satellite navigation systems, routing packets over the internet. The usage of pathfinding algorithms isn't just limited to navigation systems. The overarching idea can be applied to other applications as well, maybe even somewhere in the future where we can discover another use for it. Pathfinding is a broad field including plenty of algorithms, used in different implementations in accordance to the requirements.

The project is aimed at implementing pathfinding algorithms (both informed and un-informed) and understanding why heuristic search algorithms are arguably the best path finding algorithms out there. A maze generation algorithm can also be implemented to make the project more dynamic.

GitHub link: https://github.com/freeeeez/AI-Pathfinding

"freeeeez" is my profile name

Keywords: *pathfinding, pathfinding algorithms, heuristic search, maze generation, dynamic*

List of Figures

List of Tables

Table of Contents

Introduction

Pathfinding had gained prominence in the early 1950's in the context of routing; that is, finding shortest directions between two or more nodes. In 1956, the Dutch software engineer Edsger Wybe Dijkstra created the best-known of these algorithms: Dijkstra.

As mentioned earlier pathfinding is a broad field including plenty of algorithms, used in different implementations in accordance to the requirements. Basic algorithms like BFS and DFS do perform the task but the goal here is to find the optimal path rather than just the path. At its core, a pathfinding method searches a graph by starting at one vertex and exploring adjacent nodes until the destination node is reached, generally with the intent of finding the cheapest route.

Pygame module can be used to create the UI (in general the graphical representation for the project). Different modules will contain the required algorithms and will be called upon backbone.py when selected. Variables like "time", "number of operations" and "length of the path" can be to be used to compare the functioning of the algorithms.

Literature Survey

Table 1 Literature Survey

Authors and Year (Reference)	***Title (Study)***	***Concept / Theoretical model / Framework***	***Methodology used / Implementation***	***Relevant Finding***	***Limitations / Future Research / Gaps identified***
Dian Rachmawati and Lysander Gustin - 2020	*Analysis of Dijkstra's Algorithm and A* Algorithm in Shortest Path*	*Dijkstra's and A* pseudo code and heuristic values*	*Manually gen. nodes on a large map as a platform*	*[1]*	-
Xiao Cui and Hao Shi - 2011	*RSA Algorithm*	*A*-based Pathfinding in Modern Computer Games*	–	*[2]*	*The paper is a decade old, can use modern research to further studies*
Silvester Dian Handy Permana, Ketut Bayu Yogha Bintoro, Budi Arifitama, Ade Syahputra - 2018	*Comparative Analysis of Pathfinding Algorithms A *, Dijkstra, BFS on Maze Runner Game*	*3 levels in a maze(same obstacle pos.), inc. with each level*	*Compare process time, length of path, blocks processed in computation*	*[3]*	-

Proposed Methodology

Pygame is a cross-platform set of Python modules which is used to create video games, consisting of computer graphics and sound libraries designed to be used with the Python programming language, and will be used here for the graphical representation for the project.

Class “Node” will include all the essentials

For informed search, we need a heuristic function f(n) = g(n) + h(n), which is a way to inform the search about the direction to a goal. It provides an informed way to guess which neighbor of a node will lead to a goal. First the nodes and the grid should be created to provide a platform for the search.

After that we have to create a mechanism to detect wherever the user clicks and generate the starting/ending node or the barrier/wall nodes there. Then comes the algorithm itself, which will run on the grid when user hits SPACEBAR. The user will have an option to choose from different algorithms (options will be displayed on the right).

GRID (*on a small scale*)

S – Start node E – Target/End node

Black nodes will be the barriers and the line highlighted will be the path, if there is one. The path will be different for different algorithms.

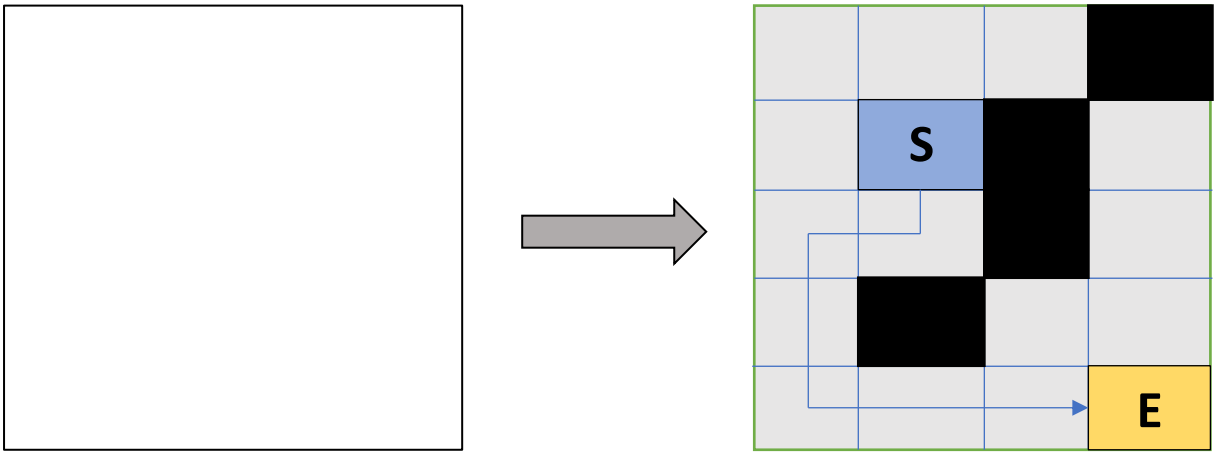


Figure 1 Small scale gird

A maze can also be generated on the grid, like using Recursive Back tracking algorithm, as it is fast, easy to understand, and straightforward to implement. The algo will follow a process to remove barriers from a pre-defined grid instead of waiting for the user to create them.

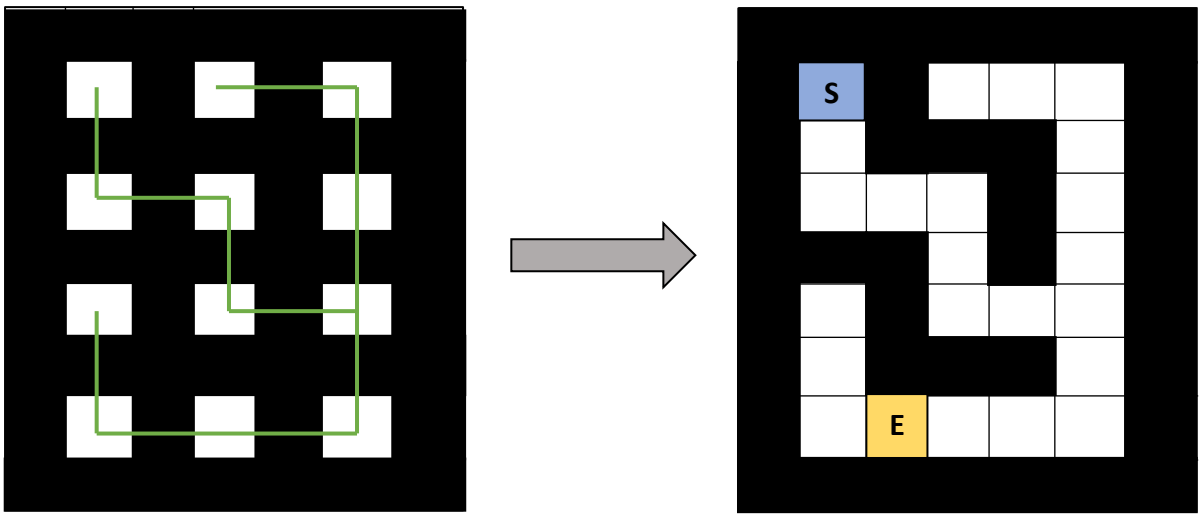


Figure 2 Small scale maze generation

The user can now create START and END node and run the algorithm like before.

Essential Pseudo-Code of this and all the pathfinding algorithms are discussed in subsequent sections.

The gird structure is shown below. All the child nodes are referred to as neighbors in the project.

R – Right node

D – Down node

L – Left node

U – Up node

The following structure will continue, barrier nodes (black nodes) will be excluded from the tree itself. All this information is stored in a 2-D array “grid”. Discussed further in the implementation section.

All the algorithms will follow simple steps until they reach the end node (if there is one). Then it will call a simple back tracer function to trace its path back to the starting node.

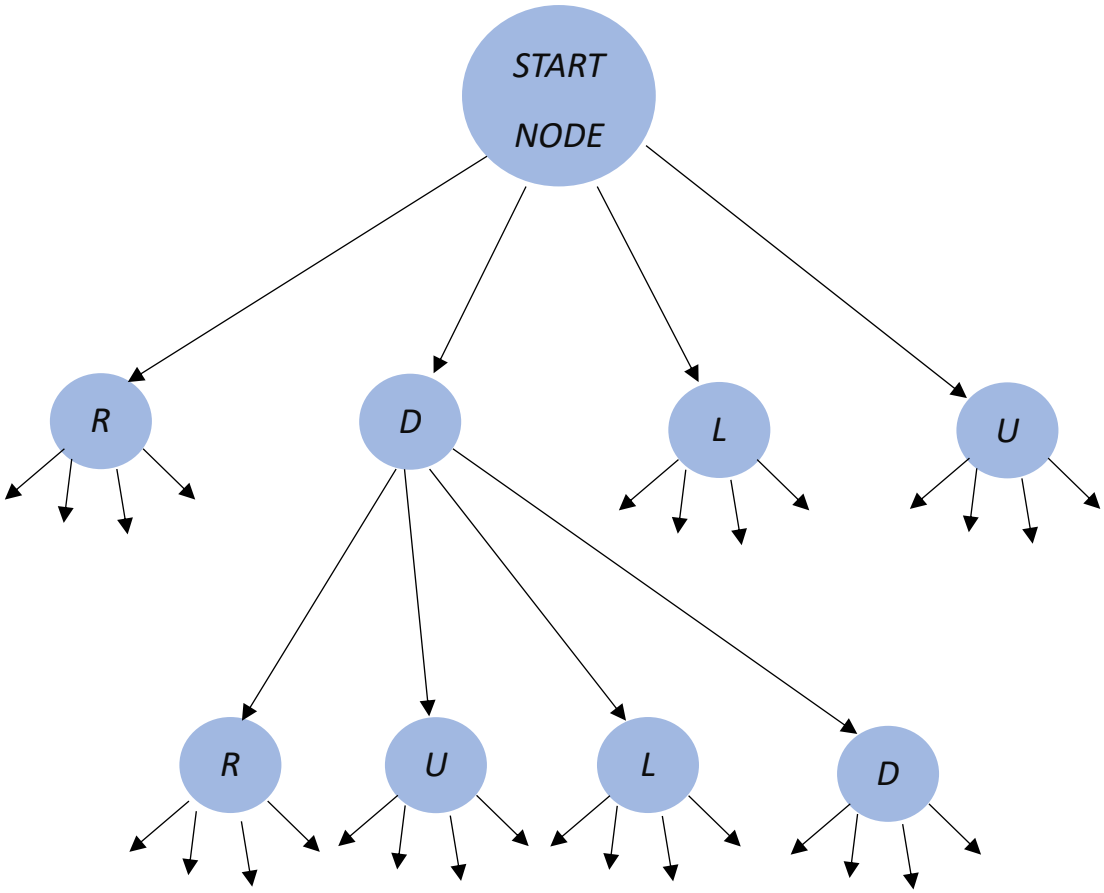


Figure 3 Grid structure

Work done and implementation

Methodology adapted:

1. START
2. The user will be displayed the window created by Pygame, can choose from three options there
3. Corresponding to the choice, a 2D array will be returned containing all the necessary information of the grid nodes, the start and end node once the user hits spacebar
4. All this information is passed on to the next stage of the algorithm
5. User will again choose an algorithm of his choice
6. Algorithm runs on the information returned in step 4
7. When the algo hits the target/end node, it will display the path
8. END
9. Backbone.py file will handle all the major function calls of the program
10. Measures taken to prevent user from advancing if not entered start and end nodes
11. The performance of the algorithm on a same grid is noted and is discussed in later sections

Hardware requirements of the system:

Processor: Intel Pentium IV processor or higher

RAM: Minimum 1 GB

Hard Disk: 10 GB or more

Keyboard: QWERTY layout/104 keys

Mouse: Optical Mouse

Monitor: Colour monitor

Software requirements:

OS: Windows 2000 professional/XP of higher

Python IDE: PyCharm Community Edition 2020.2.2 64 bits

Basic UI shown below:

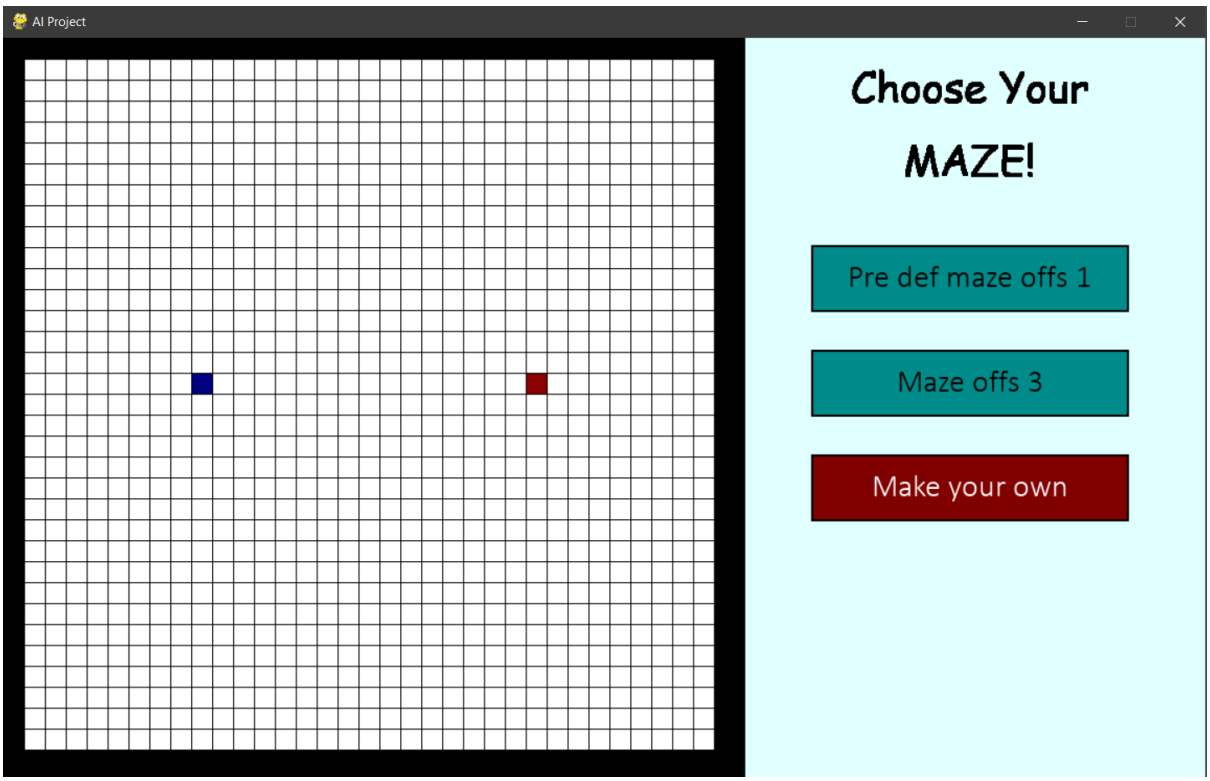


Figure 4 Display to choose maze

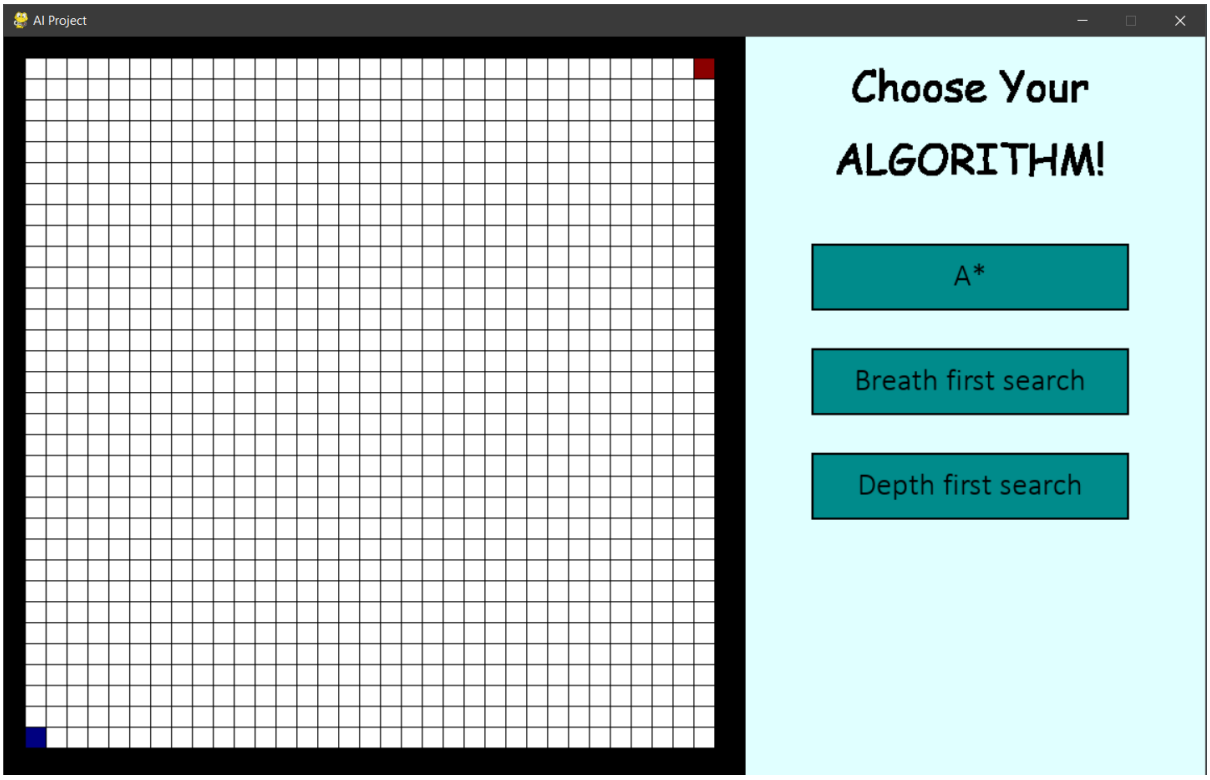


Figure 5 Display to choose algo

Essential Pseudo – Code

Breath-First Search *(BFS)*

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

Depth-First Search *(DFS)*

1. Mark the current node as visited (initially current node is the root node)
2. Check if current node is the goal, If so, then return it.
3. Iterate over children's nodes of current node, and do the following:
   3.1 Check if a child node is not visited.
   3.2 If so, then, mark it as visited.
   3.3 Go to its sub tree recursively until you find the goal node (In other words, do the same steps here passing the child node as the current node in the next recursive call).
   3.4 If the child node has the goal node in this sub tree, then, return it.
4. If goal node is not found, then goal node is not in the tree!

A* *(A star)*

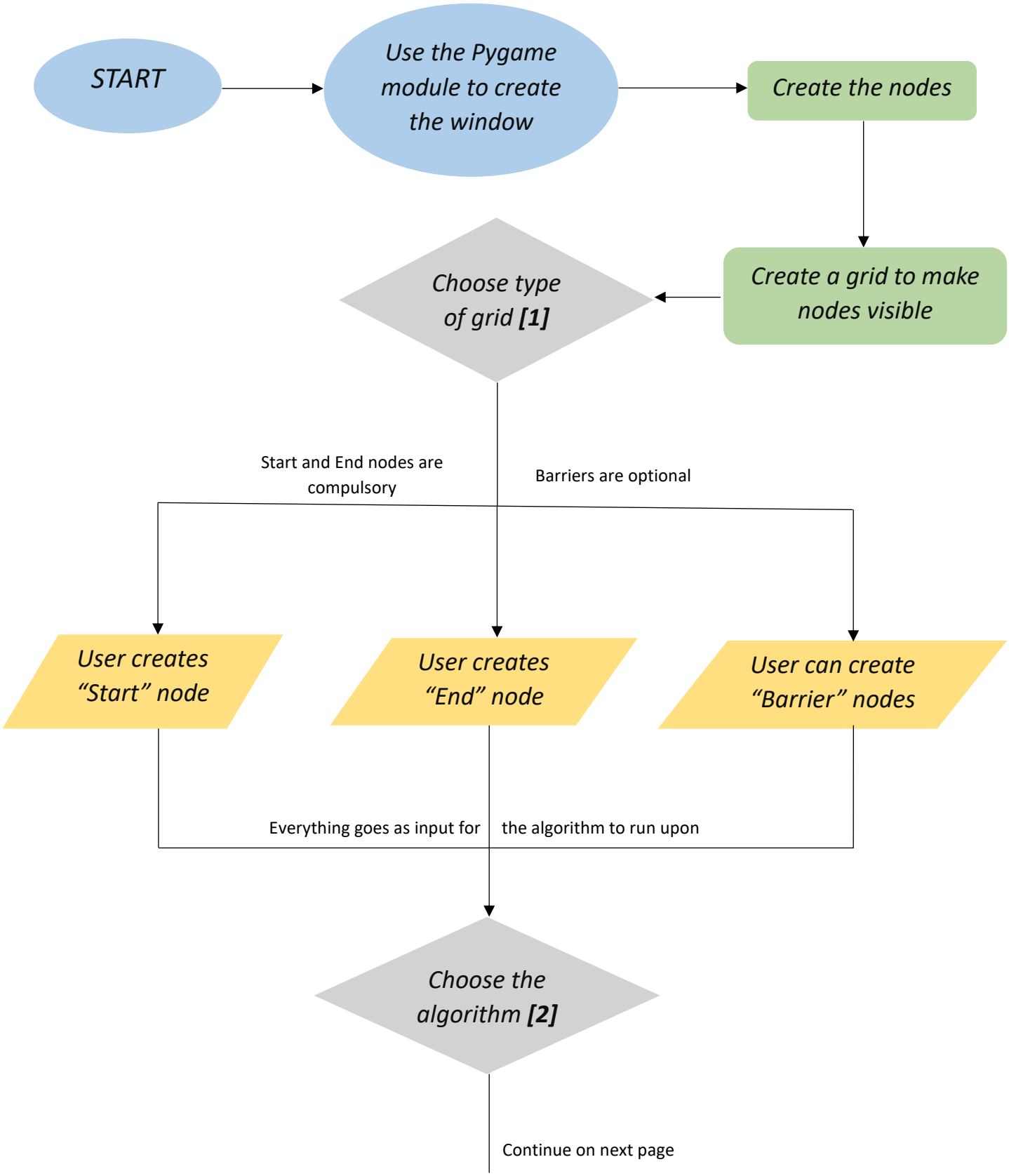
1. Assign dis[v] for all nodes = INT_MAX (distance from root node + heuristics of every node).
2. Assign dis[root] = 0 + heuristic (root, goal) (distance from root node to itself + heuristics). (Manhattan distance for the heuristic)
3. Add root node to priority queue.
4. Loop on the queue as long as it's not empty.
   4.1 In every loop, choose the node with the minimum distance from the root node in the queue + heuristic (root node will be selected first).

4.2 Remove the current chosen node from the queue (vis[current] = true).
4.3 If the current node is the goal node, then return it.
4.4 For every child of the current node, do the following:
4.4.1 Assign temp = distance (root, current) + distance (current, child) + heuristic (child, goal).
4.4.2 If temp < dis[child], then, assign dist[child] = temp. This denotes a shorter path to child node has been found.
4.4.3 And, add child node to the queue if not already in the queue (thus, it's now marked as not visited again).
5. If queue is empty, then goal node was not found!

Recursive backtracker implementation *(maze generation)*

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
3.1 Choose one of the unvisited neighbours
3.2 Remove the wall between the current cell and the chosen cell
3.3 Invoke the routine recursively for a chosen cell

Block diagram / Flow chart



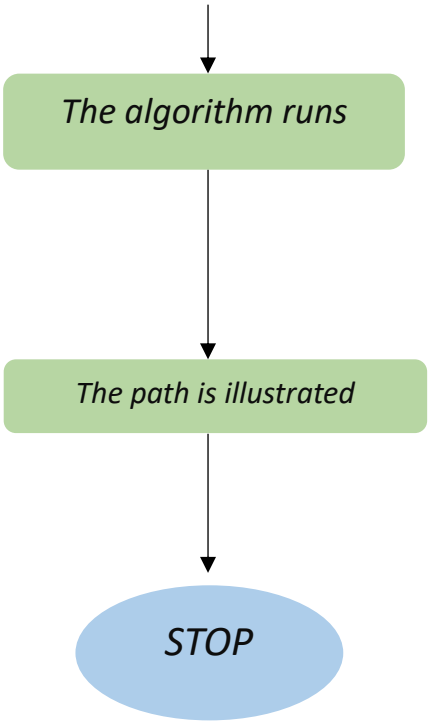


Figure 5 Block diagram / Flow chart

[1] *The user will three options for the maze, use a predefined grid, generate one using maze generation algorithm (different maze at every run) or generate his own*

[2] *The user can choose from three algorithms namely BFS, DFS and A**

Screenshots

BFS with the dynamic maze made by user

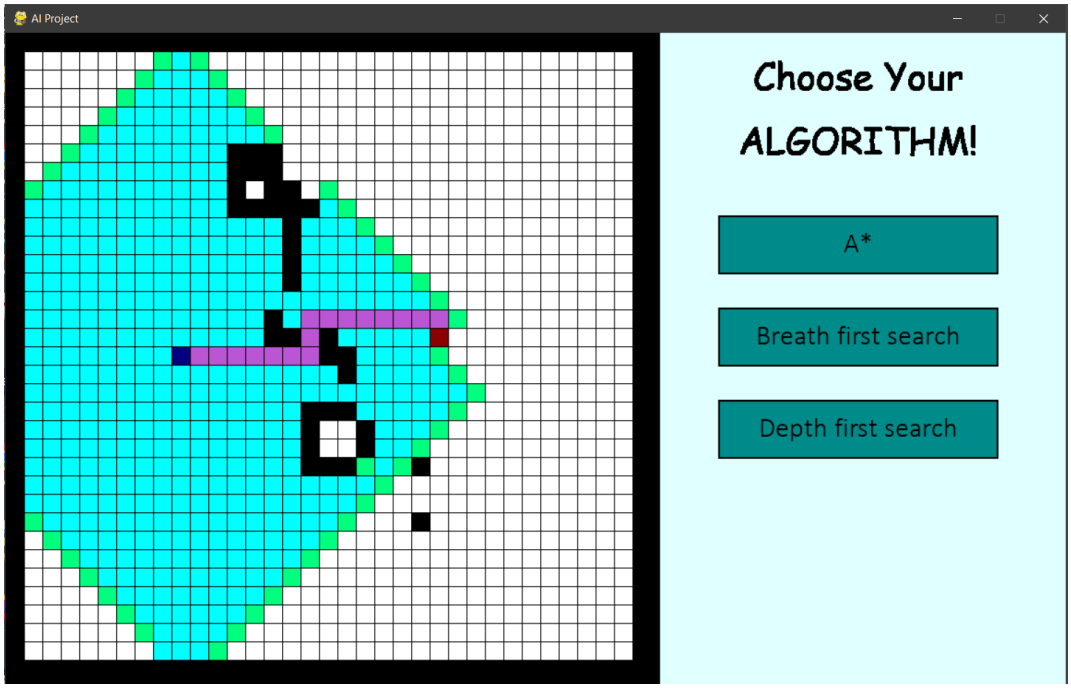


Figure 7 BFS screenshot

DFS in the recursive maze generation algorithm

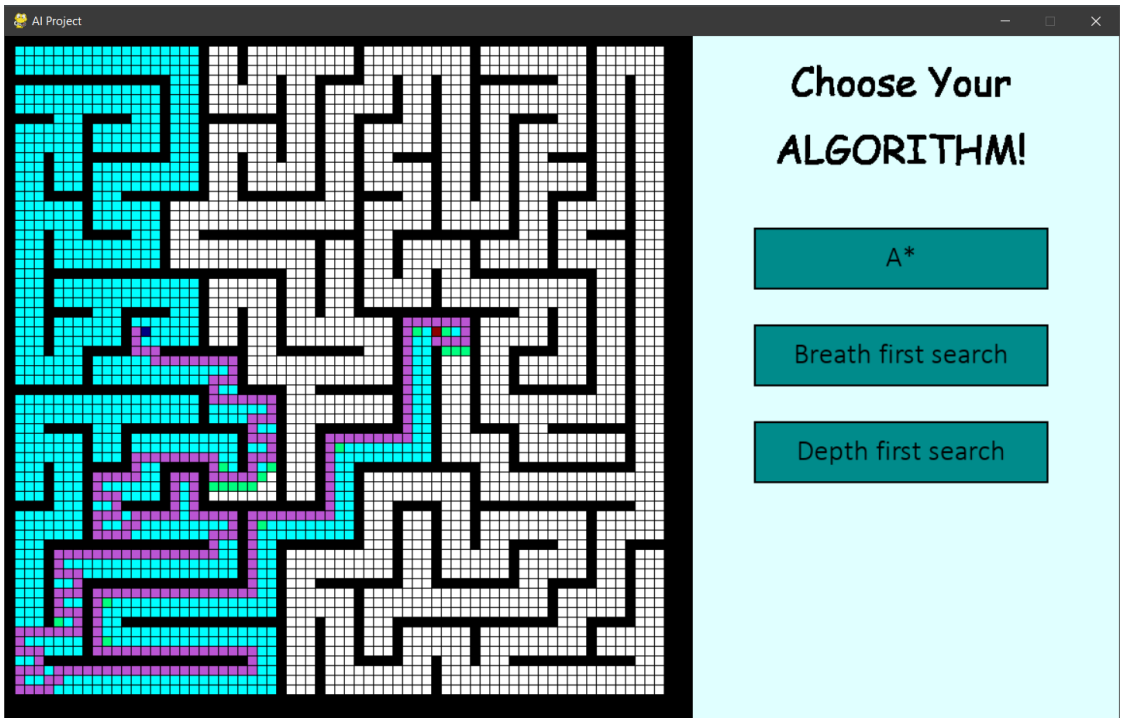


Figure 8 DFS screenshot

A in the predefined maze*

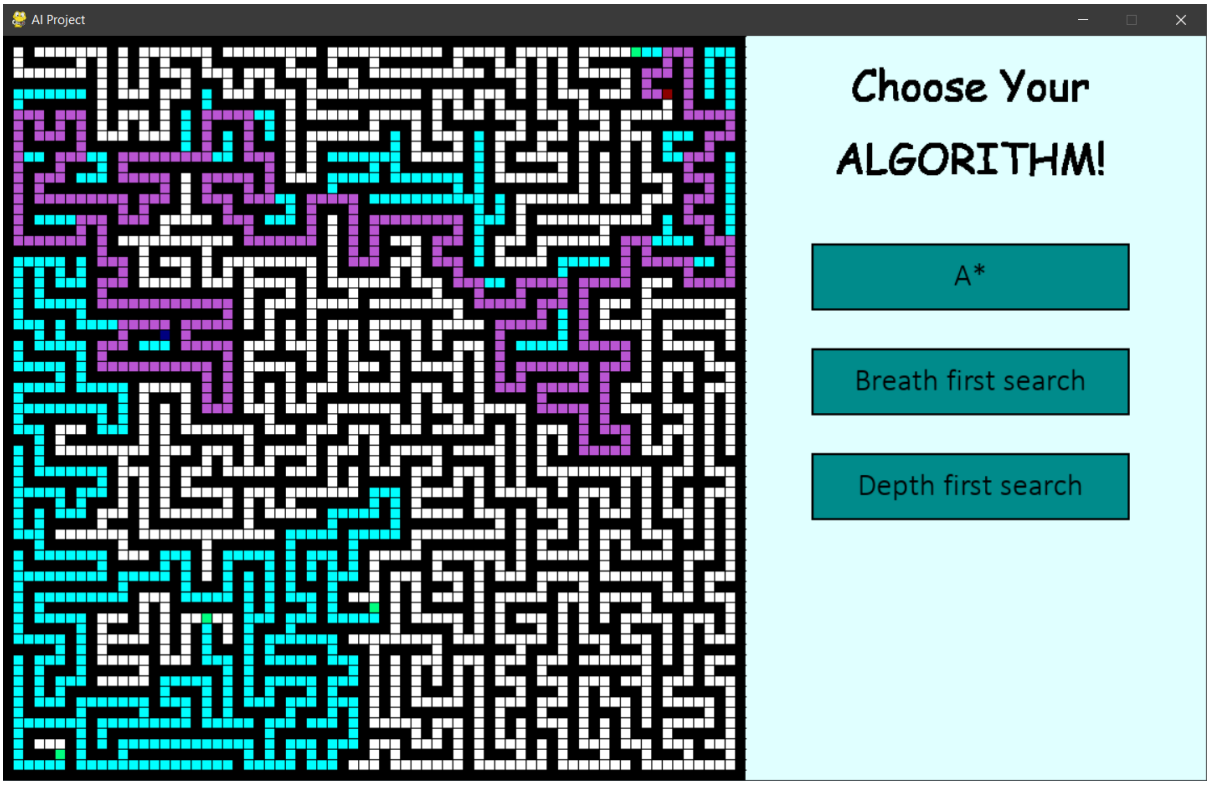


Figure 9 A screenshot*

Many variants of the implementation are possible, only some of them are shown in the output screenshots above.

Results and observations

All the algorithms are run in all three different situations. Given that the maze generation will give out a different maze each time its run, we can't use it to compare the performance. So, using predefined maze and making our own, the observations were made.

DFS will miss the node even if it is adjacent to the start node. Given the nature of the algorithm, it will find longer path if it doesn't hit the end node on time.

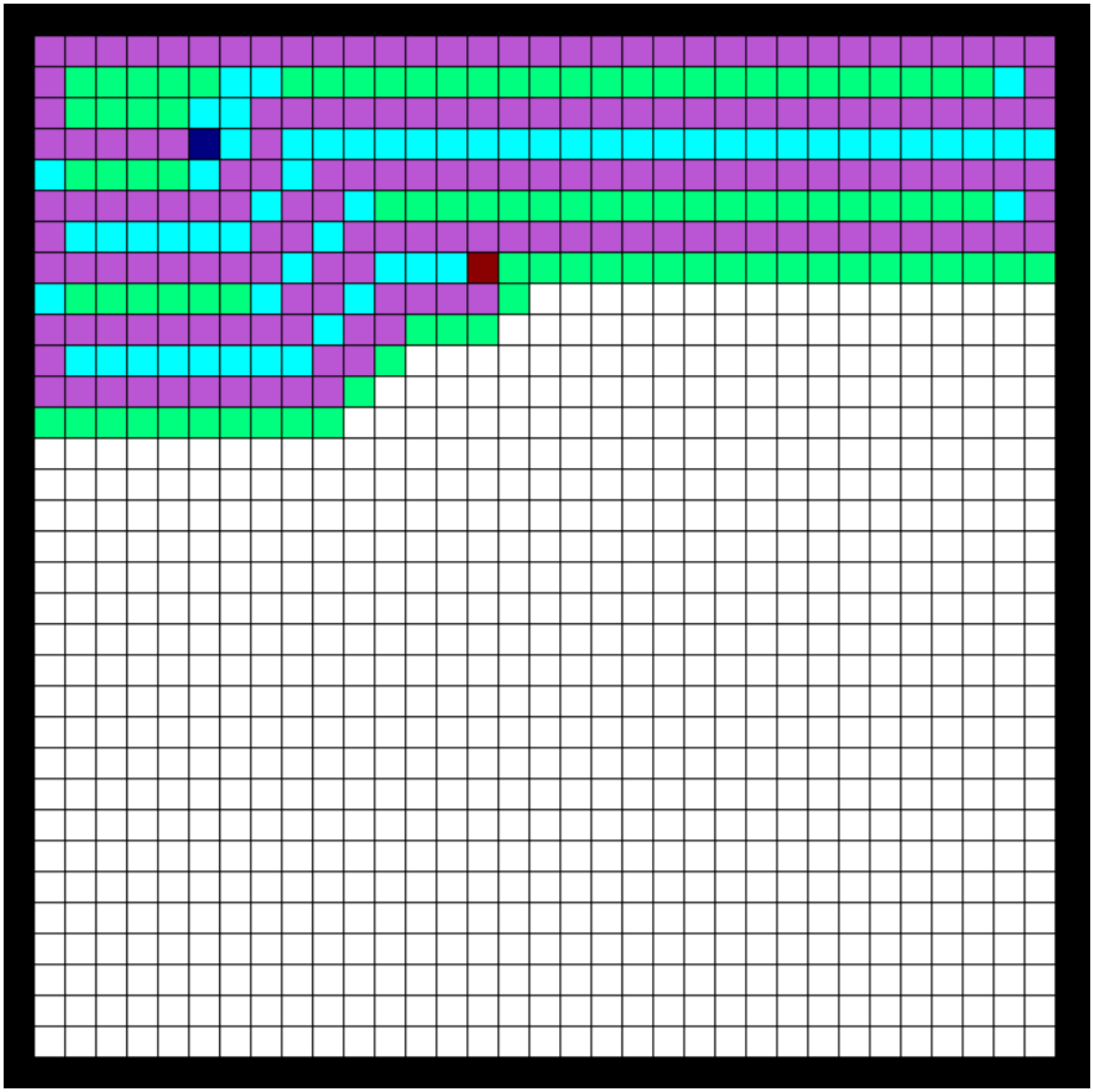


Figure 10 DFS result

BFS is in general a better algorithm as it will traverse in level order, unlike DFS, it won't miss the node even if its adjacent to the Start node. To find the path but the algo will have to traverse through a large number of nodes to get to the end node, which is a disadvantage.

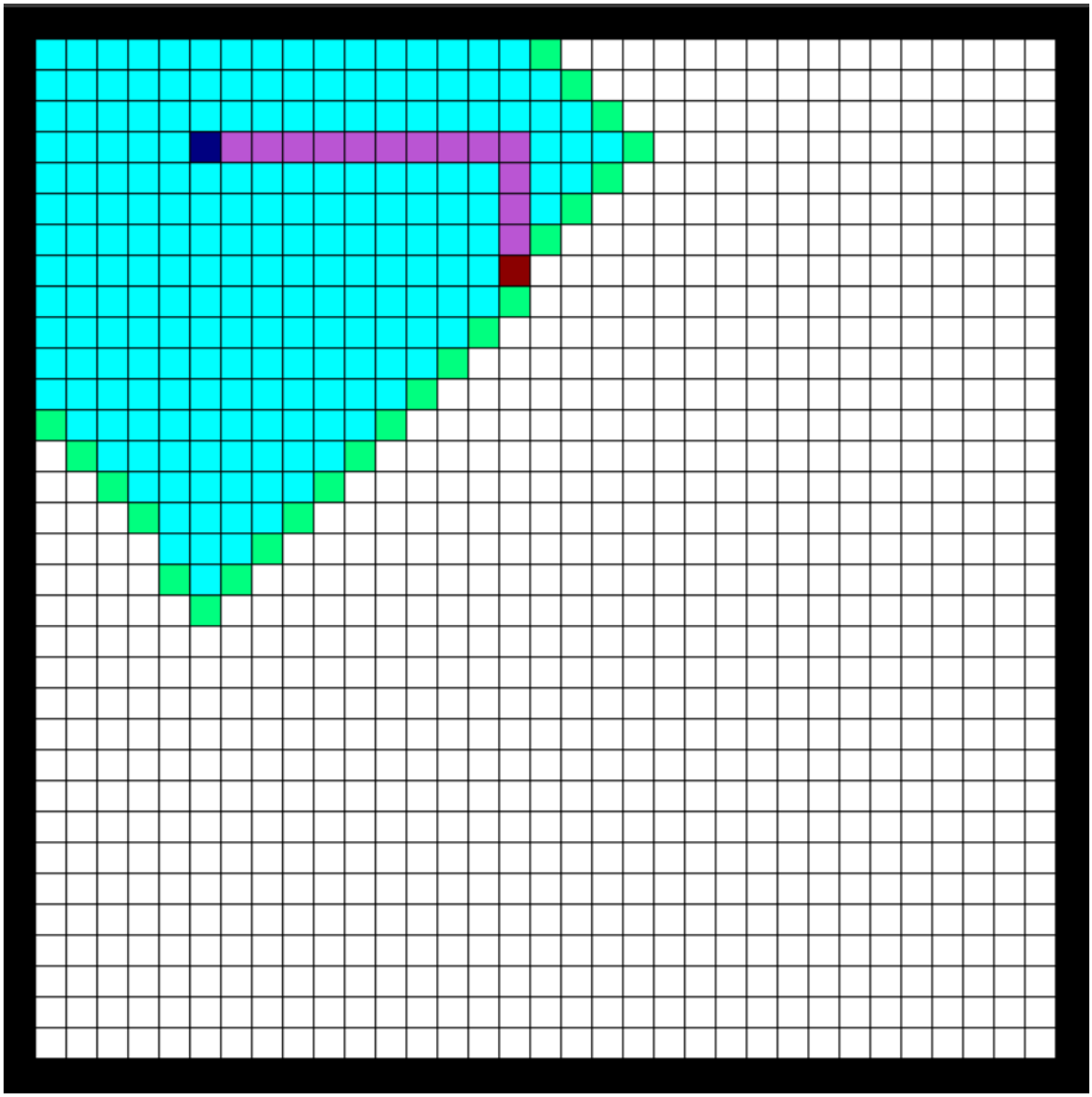


Figure 11 BFS result

To counter this, we use A* algorithm. From the below figure, it is obvious that the number of nodes it needs to traverse through is a lot less compared to other BFS and DFS algorithms.

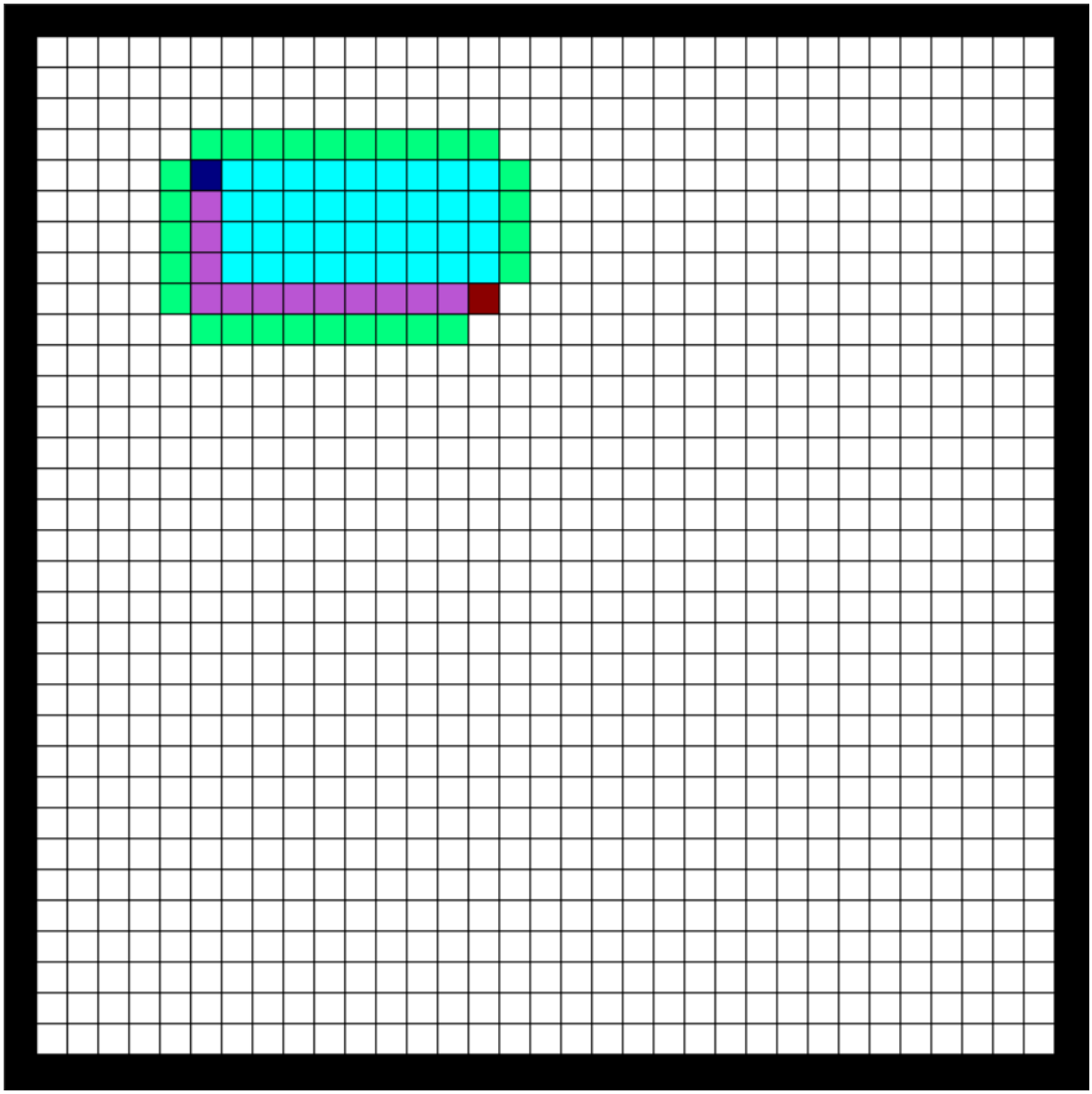


Figure 12 A result*

As A* uses a heuristic approach, it uses on an average fewer resource than BFS and DFS. Admissible heuristic guaranteed it will find the most optimal path.

References

https://www.javatpoint.com/ai-informed-search-algorithms

https://stackoverflow.com/questions/38502/whats-a-good-algorithm-to-generate-a-maze

https://weblog.jamisbuck.org/2011/1/3/maze-generation-kruskal-s-algorithm

https://flaviocopes.com/rgb-color-codes/

https://www.youtube.com/watch?v=elMXlO28Q1U&list=LL&index=2

http://www.astrolog.org/labyrnth/algrithm.htm

https://medium.com/omarelgabrys-blog/path-finding-algorithms-f65a8902eb40

https://www.redblobgames.com/pathfinding/a-star/implementation.html