Alex Savari (Student)

D3953514@live.tees.ac.uk

**Designing a New AI Solution**

Abstract

This report investigates the use of artificial intelligence algorithms to solve maze problems in Python.

Module Leader: **Dr Alessandro Di Stefano**

Module Code: CIS2048-N

**Abstract**



This report looks at how artificial intelligence algorithms can solve mazes using Python. A custom maze is created with the Pyamaze library and saved as a CSV file for consistent testing. If a saved maze exists, the program uses it; if not, it generates and saves a new one.

Three algorithms are tested: Breadth-First Search (BFS), Depth-First Search (DFS), and A\* (A-Star). Each one is applied to the same maze to ensure a fair comparison. Their performance is measured by the number of cells visited, the path length from start to goal, and overall efficiency.

BFS and DFS are uninformed search methods—they explore the maze without knowing where the goal is. A\* is an informed search method that uses heuristics to find the goal faster by focusing on better paths.

The goal of this report is to find the most efficient algorithm for solving mazes by comparing the results. The findings show important differences in performance and highlight how AI algorithms can help in real-world path-finding tasks like robotics, games, and navigation systems.

**Introduction**

Maze-solving is a common problem in computer science and AI, where the goal is to find the shortest path from a starting point to a goal. Although it seems simple, it models real-world problems like pathfinding in robotics, video games, and navigation systems.

For example, robots must find efficient paths around obstacles, video game characters need to move quickly through maps, and GPS systems find the best routes by distance, time, or traffic. Maze-solving gives a simple way to test how path-finding algorithms work.

AI provides different ways to solve mazes. These can be divided into two groups:

* **Uninformed search** methods, like Breadth-First Search (BFS) and Depth-First Search (DFS), which explore without knowing where the goal is.
* **Informed search** methods, like A\* (A-Star), which use heuristics to guide the search toward the goal.

This report compares BFS, DFS, and A\* on a maze created with Pyamaze. The comparison looks at the number of cells visited and the final path length to find out which algorithm is the most efficient and practical.

Breadth-First Search (BFS) – **Detailed Explanation**

A diagram of a diagram

AI-generated content may be incorrect.

Breadth-First Search (BFS) is a basic and widely used search algorithm that explores graphs or tree structures level by level. It was developed by Konrad Zuse in 1945 during his Ph.D. research but wasn’t published until 1972. Despite the delay, BFS has become one of the most important algorithms in computer science. It is used in various applications such as network analysis, game AI, puzzle solving, and pathfinding in maps.

BFS starts at the initial point and examines all neighboring nodes before moving deeper into the graph. It expands outward from the starting cell, visiting every node at a given depth before moving on to the next level of depth.

One of the main advantages of BFS is that it always finds the shortest path in a graph where all the steps have the same cost (unweighted graph). Since it explores all paths level by level, the first time it reaches the goal is guaranteed to be through the shortest route. This makes BFS ideal for applications like GPS systems or solving mazes where the shortest path is critical [2].

However, BFS has some disadvantages. The biggest issue is memory usage. In graphs with a high branching factor or large size, the queue can grow very large, using significant amounts of memory [1]. This may slow down the algorithm or cause it to run out of resources. Despite this limitation, BFS is still effective when the graph is relatively small or when finding the shortest path is more important than memory efficiency.

In summary, BFS is a reliable algorithm for finding the shortest path in simple graphs. It is easy to implement and understand and forms the foundation for more advanced algorithms like Dijkstra’s.

**Pseudocode of Breadth-First Search (BFS)**

For the BFS search, I created one queue and called frontier, and a list called Explored. Frontier contains the cells to be explored in the next steps, while **Explored** contains the cells that have already been searched.

Add start cell to both Frontier and Explored

Repeat until Goal is reached, or Frontier is Empty:

currCell = Frontier. Pop(0) // Remove the first cell from Frontier

for each direction (E, S, N, W):

childCell = Next possible cell in that direction

if childCell already in Explored:

Do nothing

else:

Add childCell to both Explored and Frontier

Direction Mapping:

E => childCell = (row, col + 1)

S => childCell = (row + 1, col)

N => childCell = (row - 1, col)

W => childCell = (row, col - 1)

To implement this BFS algorithm and solve the maze, I started by adding the start cell to two collections: one called **Frontier**, and one called **Explored**. The Frontier is a queue that stores the cells we plan to explore next. The Explored list keeps track of all cells that have already been visited, ensuring we don’t visit the same cell more than once. (Note: *The Explored list can also be created as a* ***set****, which automatically prevents duplicates and allows faster checking*.)

The algorithm runs inside a loop. This loop continues until we reach the goal cell or there are no more cells left to explore (i.e., the Frontier is empty). In each iteration of the loop, we remove the first cell from the Frontier—this is the **currCell**, which we are currently exploring.

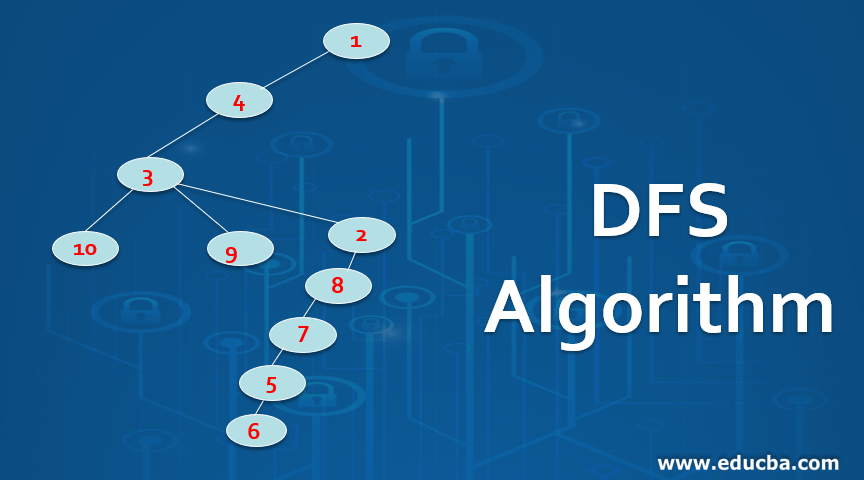
From this cell, we check each of the four directions:

* **East (E):** move to (row, col + 1)
* **South (S):** move to (row + 1, col)
* **North (N):** move to (row - 1, col)
* **West (W):** move to (row, col - 1)

For each direction, we first check if it is possible to move there (i.e., there is no wall blocking the path). If the new cell (called **childCell**) has not been visited before, we add it to both the Frontier and the Explored collection. This ensures it will be checked in a future loop and not revisited again.

By using a queue (FIFO), BFS explores all nearby cells first. This helps us find the **shortest path** from the start to the goal cell. It avoids cycles and redundant paths by keeping track of explored cells.

Depth First Search



**Depth-First Search (DFS)** is a way to go through a graph or tree by following one path as far as it goes, then going back and trying a different path. It was first written about in the 1800s in math work on graphs [5]. Today, it’s used for things like solving mazes, making puzzles, sorting tasks in order, finding loops in graphs, and helping computers make decisions [1].

**DFS** is different from Breadth-First Search (BFS). While BFS checks all nearby points first (going wide), DFS goes in one direction as far as it can (going deep), then comes back and tries the next way. It starts from a starting point (called the root or source) and goes down a path until it can't go further. Then it goes back and tries the next path.

**DFS** uses a stack, which is like a pile where the last thing added is the first taken out. It can also be done using recursion, where a function keeps calling itself. DFS keeps track of which places it has already visited so it doesn’t get stuck in a loop [3].

One good thing about **DFS** is that it uses less memory than **BFS** because it only remembers the path it’s currently on. It’s also useful for big tasks like checking for loops, finding connected parts in a graph, and sorting jobs that depend on each other, especially in graphs without loops [5].

However, **DFS** doesn’t always find the shortest way to something. It can also get stuck if there are loops and it doesn’t remember where it’s been.

**DFS** is used in many real-world problems like solving Sudoku, exploring mazes, checking websites through links (web crawling), and making choices in computer games.

In short, **DFS** is a simple and helpful way to search through graphs or trees. It’s good when memory is limited or when you need to explore all paths, even if it doesn’t always find the quickest way.

**Pseudocode of Depth-First Search (DFS)**

In the DFS algorithm, I used a stack named *Frontier* to hold the cells that need to be explored, and a list called *Explored* to record the cells that have already been visited.

Order = W N => S => E

Push the start cell in Frontier and Explored

Repeat until Goal is reached, or Frontier is Empty:

currCell = Frontier. Pop // remove Last Item

for each direction (ENSW):

childCell = Next Possible Cell

if childCell already in Explored list Do nothing

otherwise Append childCell to both Explored and frontier

E 🡪 childCell = possible childCell

N 🡪 childCell = possible childCell

S 🡪 childCell = possible childCell

W 🡪 childCell = possible childCel

I used an empty list called path to store the sequence of cells from the goal back to the start.

If the current cell is the goal cell:

Set a variable ‘cell’ to the goal point.

While ‘cell’ is not equal to the start point:

Add ‘cell’ to the path list

Update ‘cell’ to be its parent

Reverse the path list to get the correct order from start to goal

Reconstruct path from start to goal.

The implementation of DFS algorithm is similar to BFS algorithm, but the key difference is that in DFS, the currentCell is taken from the last item in the frontier (which is a stack), whereas in BFS, the currentCell is taken from the first item in the frontier ( a queue).

To implement the DFS algorithm and solve the maze, Similar to BFS, I started by adding the start cell to two collections: one called **Frontier**, and one called **Explored**. The **Frontier** is a stack that stores the cells we plan to explore next, while the **Explored** collection keeps track of all cells that have already been visited. This ensures we don’t visit the same cell more than once.

The algorithm runs inside a loop. This loop continues until we reach the goal cell or there are no more cells left to explore (i.e., the **Frontier** is empty). In each iteration of the loop, we remove the last cell from the **Frontier**—this is the **currCell**, which we are currently exploring.

From this cell, we check each of the four directions:

* **East (E)**: move to (row, col + 1)
* **South (S)**: move to (row + 1, col)
* **North (N)**: move to (row - 1, col)
* **West (W)**: move to (row, col - 1)

For each direction, we first check if it is possible to move there (i.e., there is no wall blocking the path). If the new cell (called **childCell**) has not been visited before, we add it to both the **Frontier** and the **Explored** collection. This ensures it will be checked in a future loop and not revisited again.

By using a stack (LIFO), DFS explores as deeply as possible along each branch before backtracking. This means that it will explore one path all the way to the end (or until it’s blocked) before checking other paths. This process continues until the goal cell is reached or there are no more cells to explore.

In short, this pseudocode explains how DFS explores one path as deeply as possible before backtracking and checking other paths. It avoids revisiting cells by keeping track of the **Explored** cells and ensures that if a path exists, it will eventually be found, although it may not necessarily be the shortest path.

A\* Search



A\* Search is a clever and popular way to find the best path from one place to another. It was first made in 1968 by Hart, Nilsson, and Raphael [6], and is now used in GPS, games, robots, and puzzle-solving.

Unlike other methods like DFS and BFS, which don’t know where the goal is, A\* is an informed search. This means it uses extra information (called a heuristic) to guess the best direction to go. The heuristic helps A\* focus on paths that are more likely to reach the goal quickly.

A\* uses a formula: f(n) = g(n) + h(n) [8].

* g(n) is the real cost from the start to the current point.
* h(n) is the estimated cost to the goal (the heuristic).

It picks the path with the lowest f(n), which means it’s both low-cost and likely to reach the goal quickly.

Heuristics are smart guesses that help A\* work quicker and still find the shortest path. Two common types are the ***Manhattan*** distance and the ***Euclidean*** distance [8]. A good heuristic must not guess too high.

A\* is great because it often finds the shortest path and is faster than checking everything. But it can use a lot of memory, especially in big problems or with poor heuristic.

In real life, A\* is used in GPS systems, games for moving characters, robots for safe travel, and puzzles like mazes or sliding tiles.

In short, A\* is a smart and useful way to find the best path. It combines real steps with smart guesses to reach the goal quickly and correctly.

**Pseudocode of A\* Search**

open = Priority Queue

g\_score = {cell: infinity for all cells, and 0 for start cell}

f\_score = {cell: infinity for all cells, and R(start) for start cell}

open.put 🡪 (f\_score(start), h(start), start)

while open is Not empty or Goal reached:

currCell 🡪 open.get cell value # get the cell with the lowest f\_score

if currCell == goal:

break #goal reached

for each direction (ESNW):

childCell = Next Possible Cell

temp\_g\_score=g\_score(currCell)+1

temp\_f\_score=temp\_g\_score+h(childCell)

if temp\_f\_score < f\_score(childCell):

g\_score(childCell) = temp\_g\_score

f\_score(childCell) = temp\_f\_score

open.put 🡪 (f\_score(childCell), h(childCell), childCell)

To implement the A\* algorithm, I used PriorityQueue from the queue module, which is a data structure used to order nodes based on their priority (in this case, the estimated cost to reach the goal).

I began by defining the start and goal points as global variables. These points are represented as tuples in the form (row, column) and can be easily changed depending on where the search should begin and end.

Next, I created two separate functions to calculate the heuristic values between two given cells:

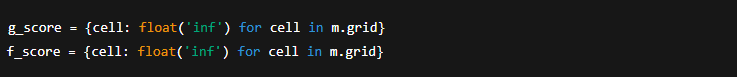
* One function (M) uses Manhattan Distance, which is the sum of the absolute differences between the x and y coordinates.
* The other function (E) uses Euclidean Distance, which calculates the straight-line distance between two points.

These heuristic functions take two cells (both as tuples) as input and return the estimated cost between them. A global reference R is used to allow easy switching between the two heuristic methods by simply changing the value of R.

The A\* search algorithm is implemented in the aStar function, which takes the maze object m as input. Within this function, the start point is defined, and two dictionaries g\_score and f\_score are initialized for all the cells in the maze. These dictionaries store:

* g\_score: the cost of the shortest path found so far from the start to the current cell (initialized to infinity for all cells except the start).
* f\_score: the total estimated cost from start to goal through a given cell, which is the sum of g\_score and the heuristic estimate.

I used dictionary comprehension to initialize these scores efficiently:



A priority queue (open\_set) is then used to keep track of which cells to visit next, ordered by the lowest f\_score. As the algorithm runs:

* It explores neighboring cells (East, South, North, West) based on the maze's structure.
* It updates the g\_score and f\_score if a shorter path is found to a neighboring cell.
* It also keeps track of the path taken using the aPath dictionary, mapping each cell to the cell it came from.

An agent is placed at the start cell to visually trace the search in real time, and a delay is added for clearer visualization.

Once the goal cell is reached, the path is reconstructed by backtracking from the goal to the start using the aPath dictionary. The reconstructed path is then traced separately using another agent to highlight the final optimal path found by the A\* algorithm.

**Experiments and Results**

I implemented and evaluated three different search algorithms to solve the same maze: Breadth-First Search (BFS), Depth-First Search (DFS), and A\* (A-star) algorithm.

The **A\*** algorithm with the Manhattan heuristic worked the best because it visited the fewest number of cells **99** while still finding the shortest path of **27** steps. This shows that A\* with the Manhattan heuristic is very good at finding the fastest route without checking too many unnecessary places.  
The A\* algorithm with the Euclidean heuristic also found the shortest path with **27** steps but visited more cells **179** compared to the Manhattan version.

**BFS** also found the shortest path of **27** steps, but it visited a lot more cells **296** to do it. This makes BFS slower and uses more computer memory, because it checks every possible path before finding the right one.

**DFS** visited fewer cells than BFS, with 237 cells visited, but its path was longer at 31 steps. This happened because DFS goes deep into one path first without thinking if it is the best, which can lead to longer and less direct routes.

**Explanation of Results**

The results look like this because each search algorithm works in a different way and follows its own rules to find the goal.

**A\*** (A-star) is a smart pathfinding method because it doesn’t just explore randomly. It uses a heuristic, which is a clever guess that helps the algorithm decide which path looks best to try next. This makes it faster and more efficient than other methods like BFS and DFS, especially in large or complex mazes [9].

**A\*** uses this formula:  
f(n) = g(n) + h(n)

* g(n) is the real cost (or distance) it has taken to get from the start point to where it is now.
* h(n) is the heuristic – an estimate of how far it is from the current point to the goal.

The algorithm adds these two values to decide which path looks the best overall. It always picks the path with the lowest total cost (f(n)). This makes A\* good at finding the shortest route while skipping unhelpful directions.

There are two popular types of heuristics used in A\* [10]:

1. **Manhattan** distance – These counts how many steps it takes to move left, right, up, or down to reach the goal. The formula is:  
   |x2 - x1| + |y2 - y1|  
   It’s called "Manhattan" because it works like walking around blocks in a city grid. You can’t move diagonally, only in straight lines.
2. **Euclidean** distance – This measures the straight-line distance between two points, like drawing a direct line with a ruler. The formula is:  
   √((x2 - x1)² + (y2 - y1)²)  
   It works well when diagonal movement is allowed, such as in open spaces or maps with more flexible movement [11].

In this code example, I have two functions to calculate the heuristic in two different ways: M for Manhattan and E for Euclidean. A global value called R decides which type of heuristic will be used in the algorithm. If R is set to “M”, it uses Manhattan; if R is “E”, it uses Euclidean. This gives flexibility based on the type of map or movement allowed.

In this maze example, Manhattan distance worked better because the maze only allows moving in four directions — up, down, left, and right. Since diagonal moves are not allowed, Manhattan gives a more accurate guess of how far the goal is. This makes it easier for A\* to focus on useful paths and avoid wasting time exploring unnecessary areas (LaValle, 2006).

BFS (Breadth-First Search) does not use any guessing. It explores all possible paths step by step, starting from the beginning and working outwards. It checks every node at the same level before going deeper. The good thing about BFS is that it always finds the shortest path in an unweighted graph [1]. But it has a downside: it needs to store and check a lot of nodes, which can take up a lot of memory and slow things down in bigger mazes.

DFS (Depth-First Search) goes all the way down one path before it goes back and tries another. This can be fast and uses less memory than BFS, but it does not always find the shortest path. In this case, DFS found a longer path in the maze because it went deep in the wrong direction first [1].

**Conclusion**

In this project, three different algorithms BFS, DFS, and A\* were tested for solving mazes.  
The results showed that A\* was the most efficient algorithm. It found the shortest path **27** steps while visiting the fewest number of cells **179**, making it fast and memory-efficient.  
BFS also found the shortest path **27** steps but had to visit many more cells **296**, which made it slower and heavier on memory.  
DFS visited fewer cells than BFS **237**, but it produced a longer path **31** steps because it explores deep into one direction without checking if it is the best way.

To make the test fair, I tested these algorithms on different maze sizes, with different complexity levels, and with different start and goal points. I found similar results each time: A\* consistently gave the best balance between finding the shortest path and keeping the number of visited cells low.

There were some limitations in this experiment. I did not measure the exact time each algorithm took.

**Self-Reflection**

This was my first project using AI algorithms, and it helped me understand how they work in real problems. Before this, I only knew about AI in theory.

I learned how to use BFS, DFS, and A\* algorithms to solve a maze. At first, it was difficult to understand how they search and manage memory but coding them in Python made it clearer. I also learned how important it is to test fairly and compare results properly.

I saw how A\* can find the best path faster by using smart guesses, while BFS and DFS explore differently. This project showed me how important AI is in real-world tasks like navigation and games.

If I do another project, I will also measure the time it takes for each algorithm to solve the maze and test on more types of mazes. Overall, I feel more confident about using AI in the future.

**GitHub Repository**

The GitHub repository has files that are written in Python and made using Visual Studio Code.

[**https://github.com/Al135663/MazeFirst.git**](https://github.com/Al135663/MazeFirst.git)

[**GitHub repository for Jupyter notebook files**](https://github.com/Al135663/Maze.ipynb.git)

**References**

[1], **Cormen, T.H., Leiserson, C.E., Rivest, R.L. & Stein, C., 2009**. *Introduction to Algorithms*. 3rd ed. MIT Press.

[2], **Russell, S. & Norvig, P., 2021. Artificial Intelligence***: A Modern Approach*. 4th ed. Pearson.

[3], **Sedgewick, R., & Wayne, K. (2011).** *Algorithms* (4th ed.). Addison-Wesley.

[5], **Knuth, D. E. (1997). The Art of Computer Programming***, Volume 1: Fundamental Algorithms* (3rd ed.). Addison-Wesley.

[6], **Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2008).** *Algorithms*. McGraw-Hill Education.

[7], **Hart, P. E., Nilsson, N. J., & Raphael, B. (1968).** *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.*  
[8], **GeeksforGeeks** – A\* Search Algorithm: https://www.geeksforgeeks.org/a-search-algorithm/  
[9], **Red Blob Games** – Pathfinding Visual Guide: https://www.redblobgames.com/pathfinding/a-star/introduction.html

[10], **Hart, P.E., Nilsson, N.J. and Raphael, B.** (1968) ‘A formal basis for the heuristic determination of minimum cost paths’, *IEEE Transactions on Systems Science and Cybernetics*, 4(2), pp. 100–107.

[11], Russell**, S.J. and Norvig, P.** (2021) *Artificial Intelligence: A Modern Approach*. 4th edn. Harlow: Pearson.

[12], LaValle**, S.M.** (2006) *Planning Algorithms*. Cambridge: Cambridge University Press.