



Department of Artificial Intelligence

Date: 12/7/2023

PROJECT

Robot path planning in a maze

Authors:
Abubakar Waziri
Mohammed Sattar
Youssef El Nahas
Hamza AlKaf

Dept. of **Artificial Intelligence**
Faculty of Computer Science and Information Technology
University of Prince Mugrin, Madinah KSA

CONTACT INFORMATION

This project report is submitted to the Department of **Artificial Intelligence** at University of Prince Mugrin in partial fulfillment of the requirements for the course **Artificial Intelligence I**.

AUTHOR(S):

Mohammed Sattar

E-mail: mohammed.sattar1437@gmail.com

Abubakar Waziri

E-mail: abuwaziri@outlook.com

Youssef El Nahas

E-mail: elnahasyoussef123@gmail.com

Hamza AlKaf

E-mail: hthek19@gmail.com

UNIVERSITY SUPERVISOR(S):

Dr. Rami Gomaa

Department of Artificial Intelligence

Dept. of **Artificial Intelligence**
Faculty of Computer Science and Information Technology
University of Prince Mugrin
Kingdom of Saudi Arabia

Internet: <https://www.upm.edu.sa>

E-mail: info@upm.edu.sa

Phone: +966 014 831 8484

INTELLECTUAL PROPE RTY RIGHT DECLARATION

This is to declare that the work under the supervision of Dr. Rami Gomaa carried out in partial fulfillment of the requirements of the course Artificial Intelligence I, is the sole property of the University of Prince Mugrin and the respective supervisor and is protected under the intellectual property right laws and conventions. It can only be considered/ used for purposes like extension for further enhancement, product development, adoption for commercial/organizational usage, etc., with the permission of the University and respective supervisor.

This above statement applies to all students and faculty members.

DATE:

AUTHOR(S):

Name: Mohammed Sattar

Signature: 


Name: Abubakar Waziri

Signature: 

Name: Youssef El Nahas

Signature: 

Name: Hamza AlKaf

Signature: 

SUPERVISOR(S):

Name: Dr. Rami Gomaa

Signature: _____

ANTI-PLAGIARISM DECLARATION

This is to declare that the above publication produced under the supervision of Dr. Rami Gomaa is the sole contribution of the author(s) and to best of our knowledge no part hereof has been reproduced illegally (cut and paste) which can be considered as plagiarism. All referenced parts have been used to argue the idea and have been cited properly. I/We will be responsible and liable for any consequence if violation of this declaration is proven.

Date:

AUTHOR(S):

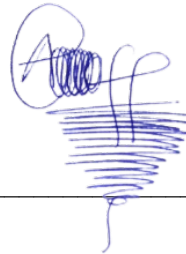
Name: Mohammed Sattar

Signature: _____



Name: Abubakar Waziri

Signature: _____




Name: Youssef El Nahas

Signature: _____



Name: Hamza AlKaf

Signature: _____



Code Tasks

Student ID	Image Processing	BFS	DFS	Heuristics	Greedy	A*	Weighted A*	UCS	Beam	Input	Output GUI
4220056	100%		100%								
4311779				100%	100%						
4311698		100%				100%	100%	100%	100%		
4310129										100%	100%

Report Tasks

Student ID	Pre-Intro	Intro	Problem Formulation	Design	Implementation	Results	Discussion	Conc
4220056				100%	25%		25%	
4311779	100%				25%			100%
4311698		100%	100%		50%		75%	
4310129						100%		

INTRODUCTION

In the field of Artificial Intelligence, a vast number of problems can be surprisingly solved by searching. These problems range from standardized ones that measure the performance of algorithms to real-world problems. Among these problems are maze problems, which are at first glance straightforward. However, such problems can become extremely complex and intractable for humans. In this project, we present a program capable of solving any maze problem, given an image. Additionally, this application will be able to employ several search techniques, which the user will choose from. These algorithms include breadth-first search, depth-first search, greedy, and A* searches. The problem's design and implementation of each algorithm will be discussed throughout this report. Moreover, test cases as well as screenshots of what the output will look like are considered at the end of this paper.

PROBLEM FORMULATION

State space:

A coordinate (x,y) corresponding to matrix where x is the horizontal position and y is the vertical position.

Successor function:

Generate the legal coordinates resulting from moving in one of eight directions: North, South, East, West, North-East, North-West, South-East, South-West.

Initial State:

Can be any state. Typically, it would be chosen by the user

Goal state:

Can be any state. Typically, it would be chosen by the user

Goal test:

Check if current state we are about to dequeue is equal to the goal's coordinates

Path cost:

Each step costs 1, so the path cost is the number of steps in the path

DESIGN:

DESCRIPTION FOR LESS TECHNICAL PEOPLE:

Our project aimed to create a system that helps navigate through a maze efficiently, just like solving a maze puzzle. We used different strategies for finding the best path from a starting point to an endpoint in the maze. Imagine a computer exploring various ways to reach the goal, considering different strategies like going straight, trying different paths, or even using a map to make decisions. We compared these strategies to see which one found the shortest or best path in the least amount of time. The project allowed us to understand how computers can find solutions in complex situations like mazes.

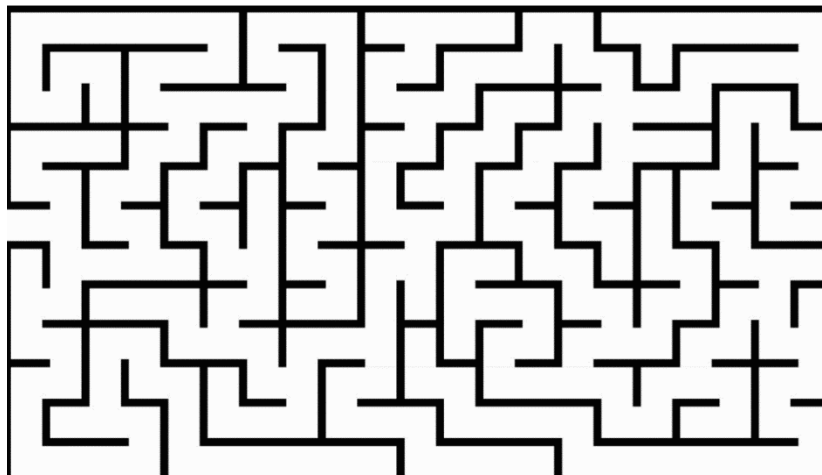
DESCRIPTION FOR MORE TECHNICAL PEOPLE:

The project involved designing a maze traversal system using pathfinding algorithms: Depth-First Search (DFS), Breadth-First Search (BFS), A* Search, and Greedy algorithms. The maze was represented as a 2D matrix. Each algorithm was implemented as functions with specific data structures like stacks, queues, or heuristics to explore and find paths through the maze. Metrics such as path length, nodes explored, memory usage, and computational complexity were tracked to compare algorithm efficiency. User inputs defined start and endpoint positions, and the system provided results indicating the path found, its length, and various performance metrics.

Design Components

1. **Maze Representation:** The maze was represented as a 2D matrix, where each cell indicated parts of the maze. Walls or barriers were represented by specific values, while open paths were denoted differently.

For example:



2. **Algorithm Implementation:** Each algorithm (DFS, BFS, A*, Greedy) was implemented as functions with specific strategies to explore the maze. Different algorithms used different data structures and techniques to find the optimal path.
3. **Metrics Measurement:** Performance metrics such as path length, nodes explored, memory usage, and computational complexity were measured and compared across algorithms to assess their efficiency.
4. **User Interaction:** The system allowed user inputs to define the starting and endpoint positions within the maze. Results were displayed to the user, indicating the path found, its length, and various performance metrics for evaluation.

ALGORITHMIC IMPLEMENTATION

FOR LESS TECHNICAL READERS:

The heart of the Maze Solver lies in its ability to solve mazes, which it accomplishes using different methods or 'algorithms'. Each algorithm has its unique way of exploring the maze - some might take a systematic approach (BFS), while others might try to guess the shortest path (A* and Greedy Best-First).

FOR MORE TECHNICAL READERS:

Four primary algorithms are implemented: A*, BFS, DFS, and Greedy Best-First. These algorithms are supported by appropriate data structures like queues and priority queues to optimize their pathfinding routines. The A* and Greedy Best-First algorithms employ heuristic functions (Manhattan and Euclidean distances) to enhance search efficiency.

IMAGE PROCESSING

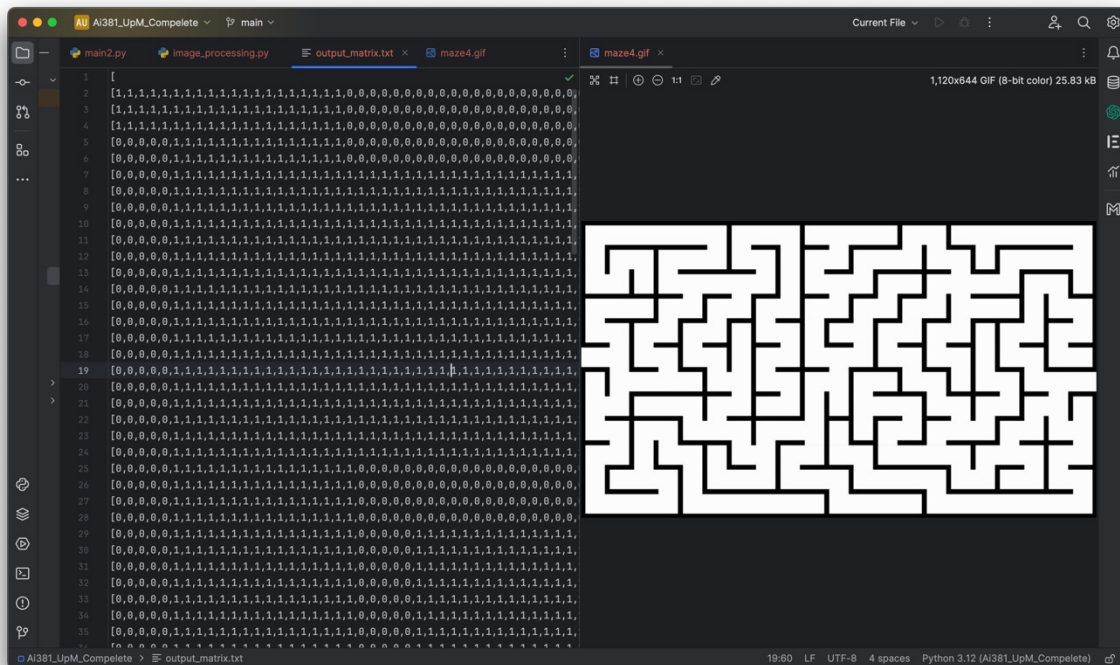
FOR LESS TECHNICAL READERS:

When you upload a maze image, the application doesn't see it the same way we do. It converts the image into a format it can understand, distinguishing between walls and open paths using colours.

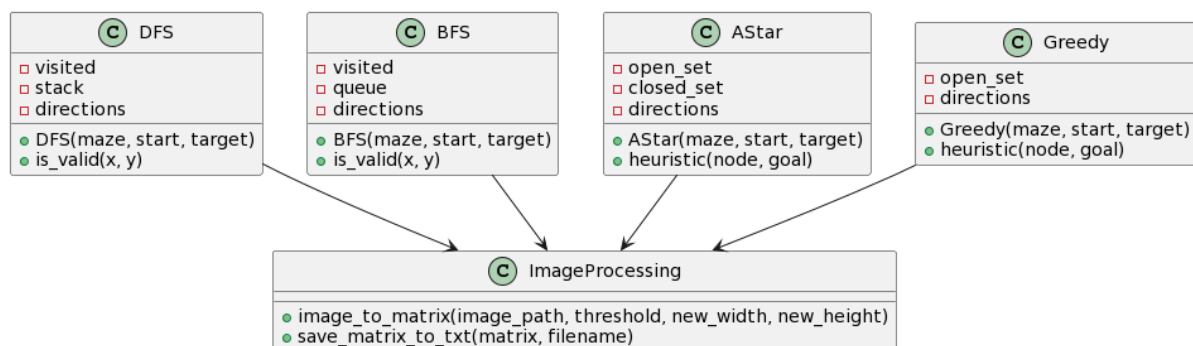
FOR MORE TECHNICAL READERS:

Upon image upload, PIL is used to convert the image into a grayscale binary matrix, where white pixels represent paths and black pixels represent walls. This matrix forms the basis for the pathfinding algorithms to process and determine a viable path.

Example of an original maze image and its binary matrix representation.



UML DIAGRAM (SIMPLIFIED):



TESTING AND OPTIMIZATION

FOR LESS TECHNICAL READERS:

Just like testing a new recipe before serving it, the application was rigorously tested with different mazes to ensure it finds the correct path efficiently and accurately.

FOR MORE TECHNICAL READERS:

The application underwent extensive unit testing, particularly focusing on algorithm efficiency and accuracy. Optimization was an iterative process, involving profiling for memory usage and computational time, especially for complex mazes.

PERFORMANCE METRICS

Performance metrics for different algorithms can vary based on maze complexities, including factors like maze size, complexity of paths, dead-ends, etc. Here's a general overview of our performance metrics for DFS, BFS, A*, and Greedy algorithms across different maze complexities:

1. MAZE COMPLEXITY FACTORS:

- a. **Size:** Small, Medium, Large (100x100)
- b. **Density:** Low density (few obstacles), High density (many obstacles)
- c. **Structure:** Few dead-ends (easy), Many dead-ends (complex)
- d. **Optimality:** Optimal path is available, No optimal path

2. PERFORMANCE METRICS:

- a. **Time Complexity:** The time taken by the algorithm to find the solution.

- b. **Space Complexity:** Amount of memory space used during the search.
- c. **Nodes Expanded:** Total nodes explored during the search.
- d. **Path Cost:** Cost (distance, weight, etc.) of the path found.
- e. **Optimality:** Whether the solution found is optimal (if applicable).

ALGORITHM PERFORMANCE ACROSS MAZE COMPLEXITIES

DFS:

- **Time Complexity:** Depends on the structure; generally lower in simpler mazes, may increase significantly in complex mazes.
- **Space Complexity:** Linear to the maximum depth of the search tree.
- **Nodes Expanded:** Can be minimal in simpler mazes but can explore many paths in complex mazes.
- **Path Cost:** May not find the shortest path; depends on search order.
- **Optimality:** Not optimal in finding the shortest path.

BFS:

- **Time Complexity:** Higher than DFS due to its breadth-first nature, can be affected by maze complexity.
- **Space Complexity:** Higher due to the need to store nodes at each level.
- **Nodes Expanded:** Expands all possible paths up to the target.
- **Path Cost:** Finds the shortest path in terms of number of edges or steps.
- **Optimality:** Optimal solution in terms of path length.

A*:

- **Time Complexity:** Generally efficient due to heuristic guidance, but complexity can vary with different heuristics and maze structures.
- **Space Complexity:** Can have higher memory usage due to maintaining priority queues.
- **Nodes Expanded:** Depends on the heuristic, can significantly reduce nodes expanded compared to uninformed searches.
- **Path Cost:** Finds optimal path considering the heuristic function.

- **Optimality:** Optimal with admissible heuristics.

Greedy:

- **Time Complexity:** Lower than A^* as it chooses the path that looks most promising without considering future steps.
- **Space Complexity:** Lower compared to A^* due to its simplicity.
- **Nodes Expanded:** Less than A^* but not necessarily minimal.
- **Path Cost:** Might not be optimal; depends on heuristic choice.
- **Optimality:** Not guaranteed to be optimal due to its greedy nature.

The performance metrics can significantly vary based on maze complexities, algorithm implementation, and specific characteristics of the maze being navigated.

Our project design aimed to explore and compare different maze-solving strategies through computational algorithms. It helped in understanding how different approaches impact efficiency and effectiveness in finding optimal paths. The design facilitated a comparative analysis of algorithms in terms of time, memory, and overall performance, enabling insights into their practical applications in various scenarios. The design of the Maze Solver is a harmonious blend of intuitive user interaction and sophisticated computational logic. It's structured to demystify the complexities of pathfinding algorithms while providing a robust and interactive tool for maze solving.

IMPLEMENTATION

In this project, we used the Python programming language since it facilitates the implementation of searching algorithms and provides a variety of libraries for image processing. The core

implementation parts are the required searching algorithms along with the setup. Moreover, additional searching techniques were added to give a better understanding of how searching should be formalized. Now, we begin with the base classes that will lay the groundwork for the implementation.

1-Base classes:

A. Define a class “problem” containing the following:

Attributes:

Initial for initial state

goals for goal state(s)

Methods:

isGoal(state) for goal test

Actions(state) returns the applicable actions

Results(state,action) represents the transition model

ActionCost(state,action,next_state) the cost of applying some

action on some state

B. Define a class “Node” with the following attributes:

State

Parent

Action the action generating this node

Path_Cost the cumulative cost

Evaluation used only for best-first search

Depth

C. Define a class “Frontier” that can be set to “FIFO”, ”LIFO”, or “Priority” through its constructor. This class is composed of:

Attributes:

queue can be of any type

Methods:

pop()

push(node)

isEmpty()

size()

2-Searching algorithms:

A. Breadth-First Search (BFS):

This algorithm uses a “FIFO” queue as a frontier, and it has a feature called “early checking” where it applies the goal test to the state as it is generated rather than dequeued. The details of how this is done are as follows:

- Define the root *node* with *initial* as a state
- Define the *frontier* and the *reached* table with the root *node* as an element
- Repeat the following until *frontier* is empty:

- 1-pop a *node*
- 2-if goal then return *node*
- 3-get a *child* of node
- 4- if *child* is goal then return it
- 5-if *child* is not in *reached* then add it to both *frontier* and *reached*
- 6-repeat 1,5 for every child of *node*

BFS takes an object of *problem* as an argument. If the search fails, that means there is no solution in the given state space, as BFS examines every possible node. The implementation can be further illustrated in the pseudocode below:

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
node ← NODE(problem.INITIAL)
if problem.IS-GOAL(node.STATE) then return node
frontier ← a FIFO queue, with node as an element
reached ← {problem.INITIAL}
while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
        s ← child.STATE
        if problem.IS-GOAL(s) then return child
        if s is not in reached then add s to reached
            add child to frontier
return failure
```

B. Depth-First Search (DFS):

The DFS algorithm is a recursive or iterative process that explores as far as possible along each branch before backtracking. In the context of maze-solving, DFS starts at the given starting point and explores all possible paths until it reaches the target or exhausts all possibilities.

def run_dfs(maze, start_pos, target_pos):

Initialization and setup

while stack:

Explore paths using DFS

return path, path_cost, nodes_expanded, max_tree_depth, max_frontier_size

C. Best-First Search:

The most general technique of all searching algorithms that use a priority queue. This algorithm operates in the same way as BFS and DFS except that it inserts the nodes in the frontier according to a priority like the UCS, which is specified by a certain evaluation function. In fact, even BFS and DFS can be implemented by Best-First search with the depth and the minus depth as evaluation functions respectively. However, this algorithm does not feature an “early checking” for a goal, as this can affect the optimality of the solution path. It rather follows “late checking” which applies the goal test to a node as it is dequeued.

Some of the algorithms that can be constructed using Best-First Search are:

I- Uniform-Cost Search (UCS):

UCS uses the cumulative cost as an evaluation function. Basically, the path from start to finish taking the least each time.

II-Greedy Search (GS):

GS uses the heuristic **$h(n)$** as an evaluation function. Heuristic can be thought of as a hint, indicating how far the goal is.

III-A* Search:

It uses **$h(n)$** and the cumulative cost as an evaluation function. The formula is **$f(n)=h(n)+g(n)$** , where **$f(n)$** is the evaluation function, and **$g(n)$** is the cumulative cost.

IV-Beam Search:

It is similar to A* search with the addition of a limit k set on the frontier to save memory in large search spaces.

V-Weighted A* Search:

In this algorithm $f(n) = W \cdot h(n) + g(n)$, where W is a weight that usually lies between 1.2 and 1.6. This weight deliberately overestimates the heuristic to make A* greedier.

Since the implementation of Best-First Search is almost identical to BFS and DFS, the pseudocode will be sufficient:

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
node ← NODE(STATE=problem.INITIAL)
frontier ← a priority queue ordered by f, with node as an element
reached ← a lookup table, with one entry with key problem.INITIAL and value node
while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
        s ← child.STATE
        if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
            reached[s] ← child
            add child to frontier
return failure
```

3-Heuristics

In this program, a heuristic function is one of the most integral parts to implement most of the aforementioned Best-First Search Algorithms. In our program, heuristic function is simply a way for the program to predict the distance from the any position to the goal. It uses this heuristic function to transform the search in the maze problem from uninformed to informed search. Generally, there are two types of heuristic functions that we have implemented: Manhattan Distance Heuristic and Euclidean Distance Heuristic.

- A. Manhattan (Taxicab) Distance: this kind of heuristic is predicated on calculating the absolute difference between two points' Cartesian coordinates and using that result to calculate the

distance between them. For example, for points (1,3) and (5,9), the method of calculating would be as follows: absolute value of (5-1) + absolute value (9-3). So, in this case, 4+6=10 will be the Manhattan distance.

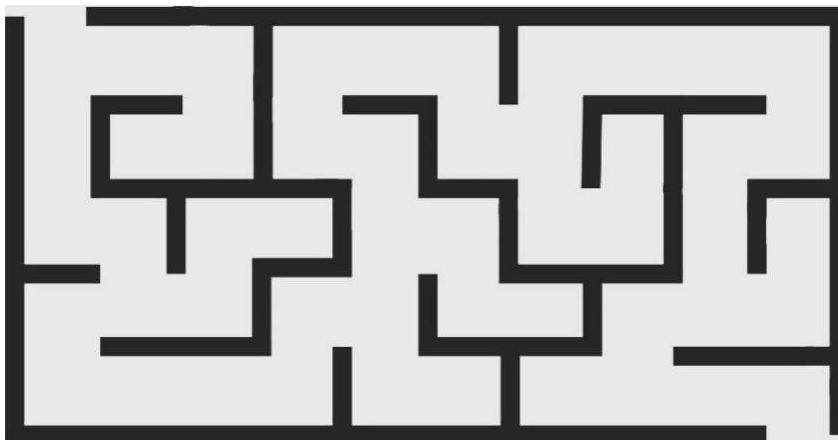
B. Euclidean (Straight-line) Distance: this type of heuristic is dependent on the straight-line distance between two points. Straight line distance is basically a rewritten version of the Pythagorean Theorem= $\sqrt{(y_1 - y_0)^2 + (x_1 - x_0)^2}$. For example, for points (1,3) and (5,9), the method of calculating would be as follows: $(9-3)^2 + (5-1)^2 = (36+16)^{1/2} = (52)^{1/2} \approx 7.21$ will be the Euclidean Distance for this case.

Both of these different methods of finding the heuristics were implemented successfully and effectively. Most of the aforementioned Best-First Search Algorithms use both types of heuristics when the search algorithm is run.

RESULTS

In the upcoming Results section, we present a comprehensive evaluation of our Python-based maze-solving program, showcasing the application of seven distinct artificial intelligence algorithms. To rigorously test the program's efficacy and versatility, three diverse mazes of different complexities were employed as benchmarks. Each algorithm, including breadth-first search (BFS), depth-first search (DFS), Greedy Best-First Search, A*, Weighted A*, Uniform Cost Search (UCS), and Beam Search, was systematically applied to navigate through these mazes. The subsequent discussion will delve into the specific results yielded by each algorithm for every maze, providing a nuanced understanding of their performance and suitability in various maze-solving scenarios.

Maze 1



Maze width after being converted
into a matrix: maze height: 110
maze width: 212 top-left
coordinate:(0,0)bottom-right
coordinate: (110, 212)

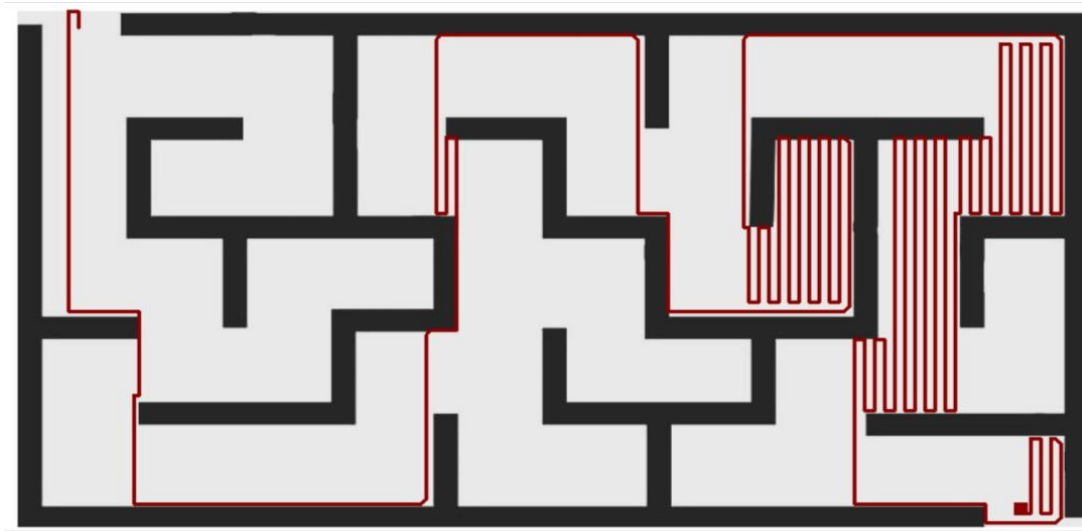
A square maze with a complex network of black walls. A red line traces a path from the top-left corner, moves right, then down, then right again, and continues through various turns and dead ends, eventually reaching the bottom-right corner.

Goal position: (107, 198)

19

that lead to a more direct trajectory. The final path reflects a series of strategic decisions made by BFS, where each chosen point contributes to the overall minimization of distance, culminating in the optimal solution from the start to the end point.

2 Depth First Search (DFS)



Start position: (3, 12)

Goal position: (106, 199)

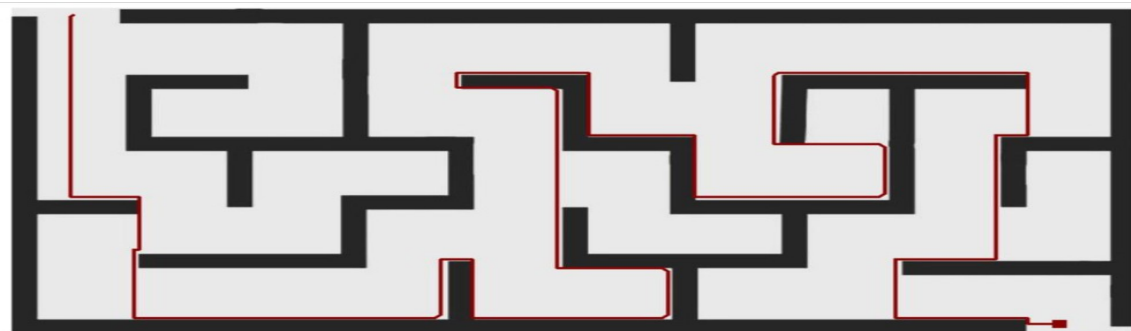
In employing Depth-First Search (DFS) to navigate from the start coordinate (3, 12) to the goal coordinate (106, 199), the algorithm explores the search space by prioritizing depth over breadth. The DFS strategy involves traversing as far as possible along each branch of the search tree before backtracking. In this particular scenario, the choice of specific points on the path is influenced by the order in which nodes are encountered during the traversal. Because of how the DFS algorithm delves deeply into one branch before exploring alternative paths, it can result in a longer path. The length of

the path, in this case, is reflective of the inherent nature of DFS, which may prioritize a specific route without immediate consideration of more efficient alternatives.

3 Greedy First Search

The Greedy Search algorithm selects the next step based on a heuristic $h(n)$ that minimizes the estimated cost to the goal, prioritizing immediate gains.

Manhattan Heuristic



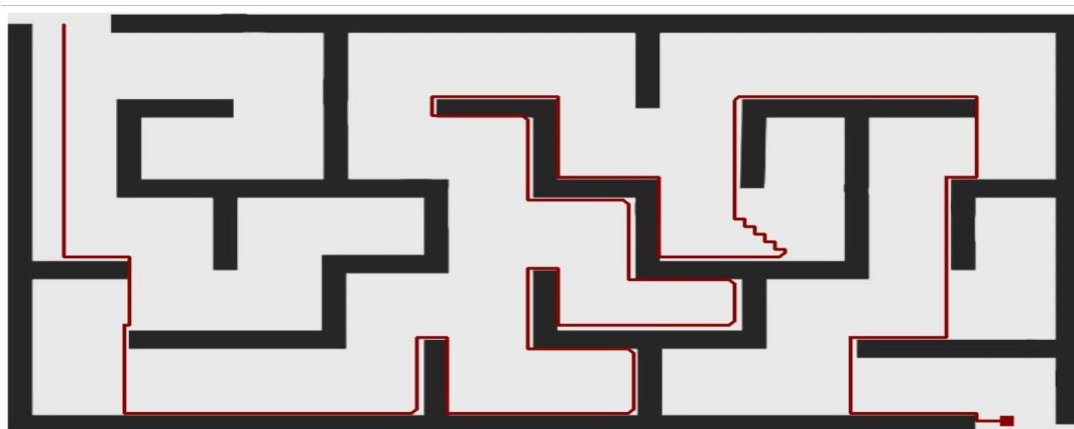
Start position: (2, 12)

Goal position: (107, 199)

When applying the Greedy Search algorithm with the Manhattan heuristic to navigate from the start coordinate (2, 12) to the goal coordinate (107, 199) within the binary matrix maze, the chosen path demonstrates a tendency to prioritize immediate proximity to the target based on the given heuristic. The Manhattan heuristic estimates the distance by summing the absolute differences in coordinates. The algorithm consistently selects points that minimize the Manhattan distance between the current position and the goal. The resulting path follows a vertical ascent from the starting point, highlighting a

preference for directions closer to the goal in a stepwise manner. Each selected point represents a local optimum, minimizing the heuristic cost and guiding the search toward the destination.

Euclidean Heuristic



Start position: (4, 11)

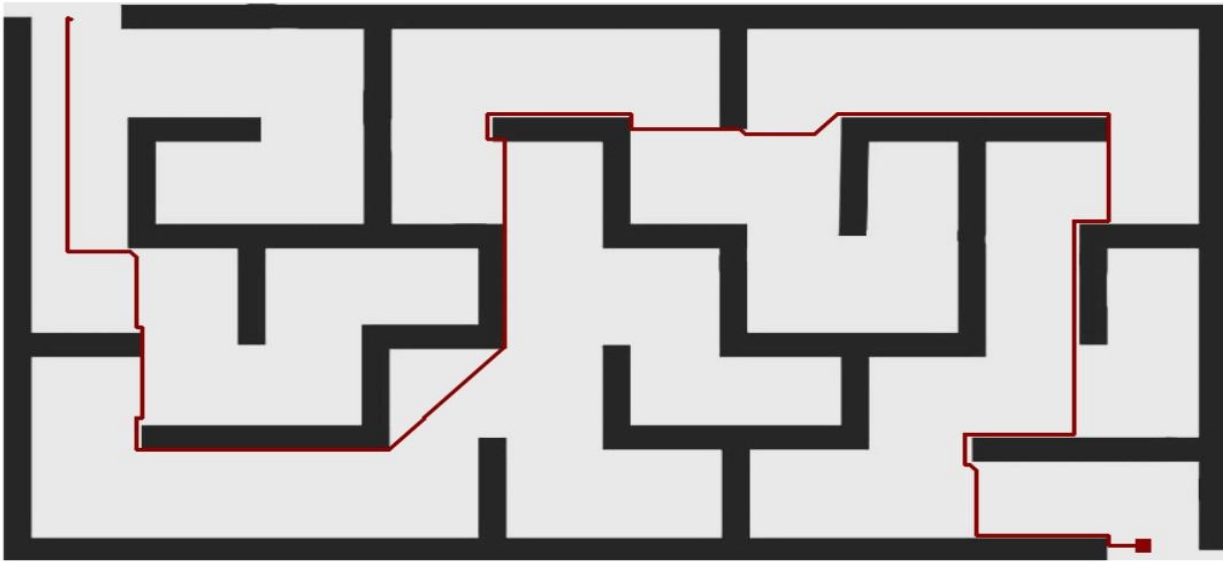
Goal position: (108, 199)

In navigating the binary matrix maze from the start position at coordinates (4, 11) to the goal position at (108, 199) using the Greedy Search algorithm with the Euclidean heuristic, the resulting path displays a distinct pattern. The algorithm systematically chooses points that minimize the estimated Euclidean distance to the goal, creating a stepwise trajectory toward the target. The distinct selection of points in the final path, reflects the algorithm's inclination to prioritize immediate gains and move closer to the goal in a sequential manner. As the algorithm progresses, it consistently opts for points that lead it along a vertical ascent, aligning with the heuristic's guidance to minimize the straight-line distance to the destination.

4 A* Search

The A* search algorithm explores possible paths step by step, evaluating each based on a combination of actual cost and estimated future cost, ultimately selecting the path that seems most promising.

Manhattan Heuristic

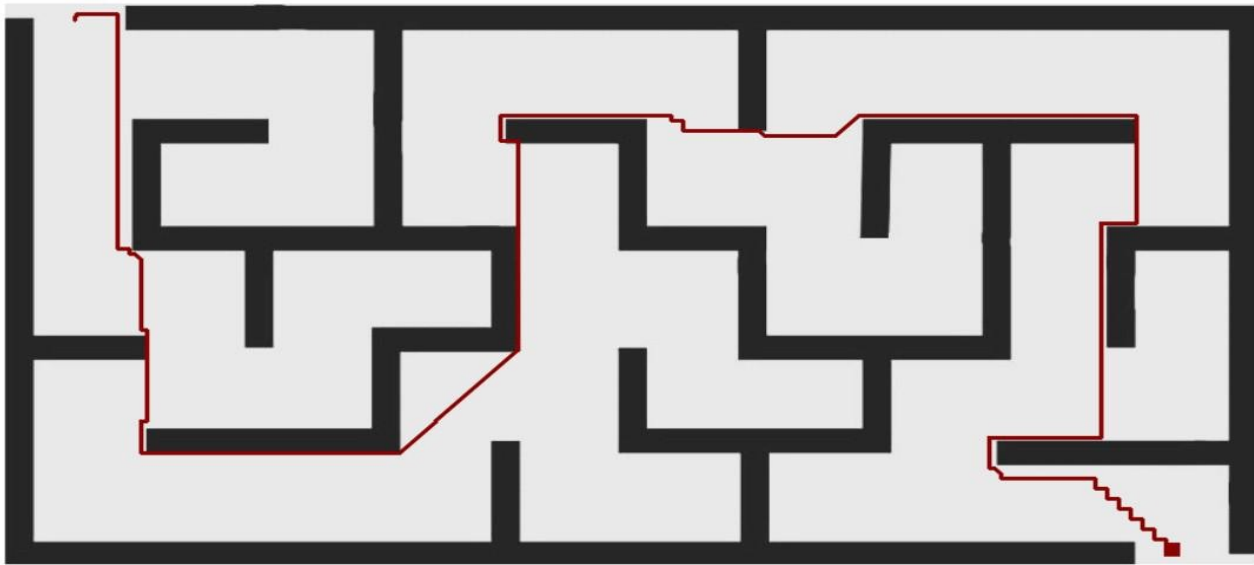


Start position: (3, 12)

Goal position: (108, 204)

The A* Search algorithm with the Manhattan heuristic successfully found a path from the starting point at (3, 12) to the destination at (108, 204) in the maze. The algorithm made decisions by considering both the distance covered so far and an estimate of the remaining distance to the goal. It tended to choose paths that minimized the total distance to the goal. For instance, when the algorithm reached the coordinates (105, 24), it continued vertically towards the goal rather than horizontally, as this choice minimized the Manhattan distance. Overall, A* Search effectively navigated the maze by smartly selecting paths, demonstrating its ability to make good decisions and reach the goal efficiently.

Euclidean Heuristic



Start position: (3, 12)

Goal position: (107, 199)

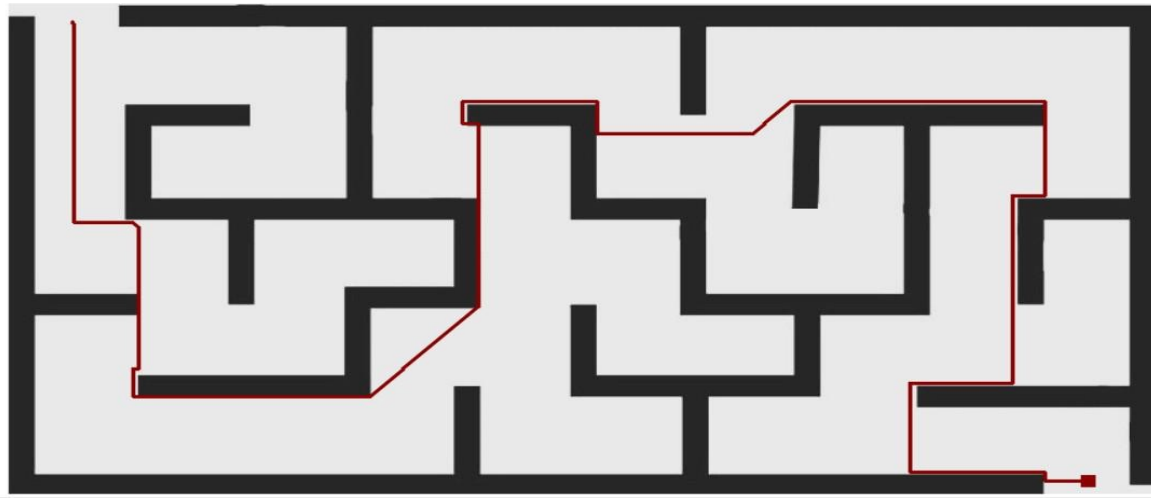
The A* Search algorithm, when applied to the binary matrix maze with a start position at (3, 12) and a goal position at (107, 199), successfully navigated through the maze, producing a path comprised of specific coordinates. The algorithm, utilizing the Euclidean heuristic, made decisions based on both the cost of the path traveled so far and an estimate of the remaining distance to the goal. Notably, it strategically selected points that minimized the total distance, considering the straight-line distance to the goal. For instance, when the algorithm reached coordinates like (48, 19) and subsequently moved

vertically and horizontally, it aimed to minimize the overall Euclidean distance to the destination. The path demonstrates the algorithm's ability to intelligently choose waypoints, optimizing the trade-off between the cost incurred and the estimated remaining distance, ultimately resulting in an efficient route from the start to the goal.

5 Weighted A* Search

Weighted A* is a pathfinding algorithm that extends A* by introducing a "weight" parameter, influencing the balance between the actual cost and a heuristic estimate when searching for the shortest path. A higher weight emphasizes the heuristic more, potentially favoring more direct routes to the goal.

Manhattan Heuristic



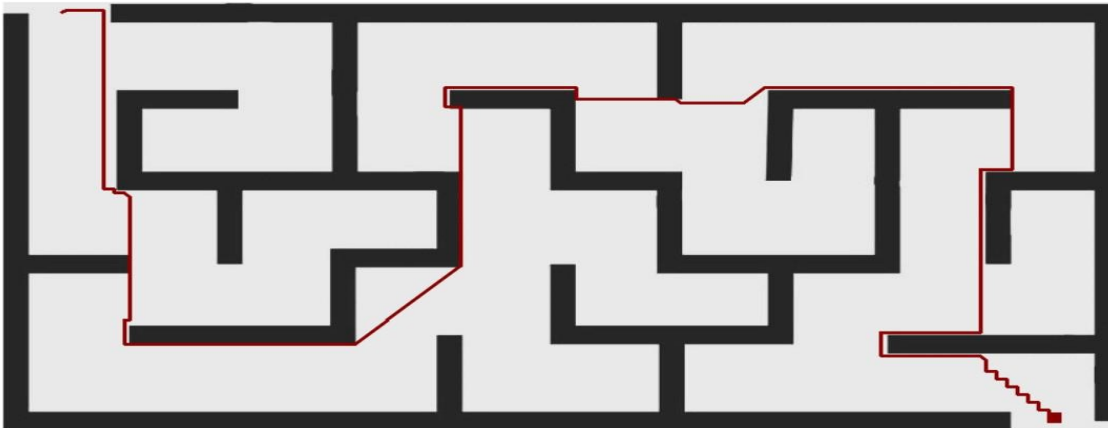
Weight: 1.5

Start position: (4, 12)

Goal position: (107, 200)

The Weighted A* Search algorithm with a Manhattan heuristic and a weight of 1.5 was employed to navigate the provided binary matrix maze from the start position at (4, 12) to the goal position at (107, 200). The Manhattan heuristic, which calculates the distance between two points on a grid as the sum of the absolute differences in their coordinates, was utilized to estimate the cost of reaching the goal from each explored point. By incorporating a weight of 1.5, the algorithm prioritizes paths that may be longer but have a lower estimated cost. The points chosen on the path reflect the algorithm's attempt to strike a balance between reaching the goal in fewer moves and minimizing the estimated cost.

Euclidean Heuristic



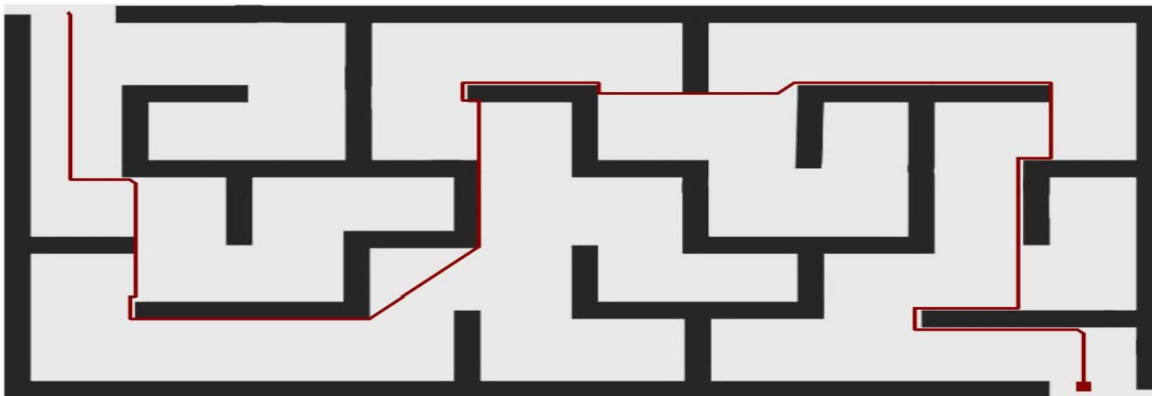
Weight: 1.5

Start position: (2, 11)

Goal position: (107, 200)

The Weighted A* Search algorithm with the Euclidean heuristic and a weight of 1.5 efficiently navigates the provided maze from the start position at (2, 11) to the goal at (107, 200). The chosen path is a series of coordinates that make strategic use of the Euclidean distance metric to estimate the remaining distance to the goal. The algorithm favors nodes that minimize the sum of the actual cost to reach the current node and the estimated cost from the current node to the goal. Additionally, the influence of the weight parameter emphasizes the balance between the actual cost and the heuristic estimate in determining the optimal trajectory from start to goal.

6 Uniform Cost Search (UCS)



Start position: (2, 12)

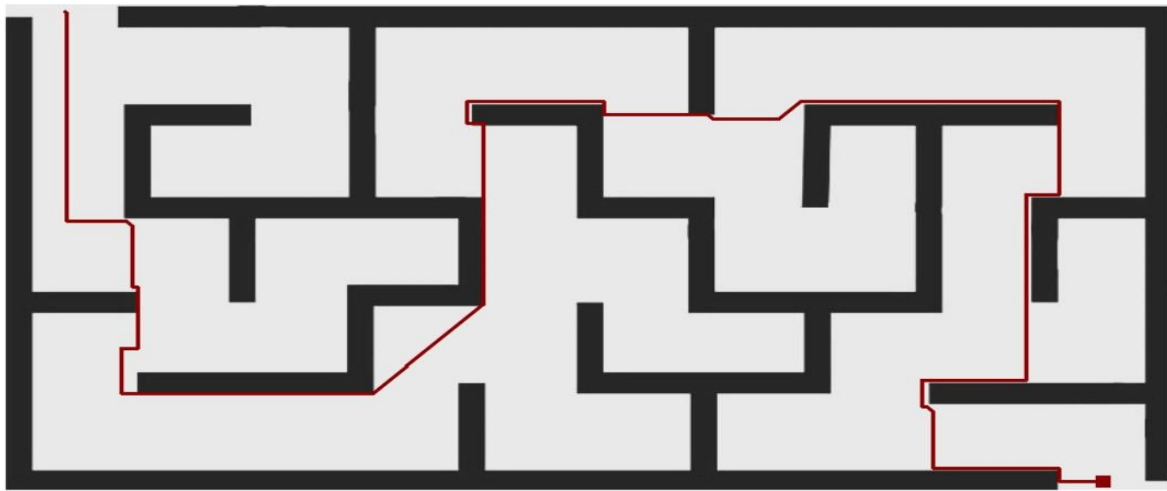
Goal position: (107, 198)

The Uniform Cost Search (UCS) algorithm efficiently navigates the provided maze from the start position at (2, 12) to the goal at (107, 198). This algorithm explores the maze by considering the cumulative cost of reaching each explored node from the start. The final path reveals a strategic selection of coordinates, highlighting the algorithm's emphasis on minimizing the overall cost. The algorithm opts for paths that involve traversing through open spaces, prioritizing nodes with lower accumulated costs. In this specific scenario, the chosen points showcase the algorithm's systematic exploration, favoring a route that incrementally moves towards the goal while carefully considering the cost associated with each step.

7 Beam Search

Beam search is a heuristic search algorithm that explores multiple paths simultaneously, maintaining a fixed number of the most promising paths, known as the "beam width." At each step, potential paths are expanded, and the algorithm selects the top candidates based on a heuristic evaluation. This process helps to efficiently navigate large search spaces, pruning less promising paths and focusing on those with higher likelihoods of success.

Manhattan Heuristic

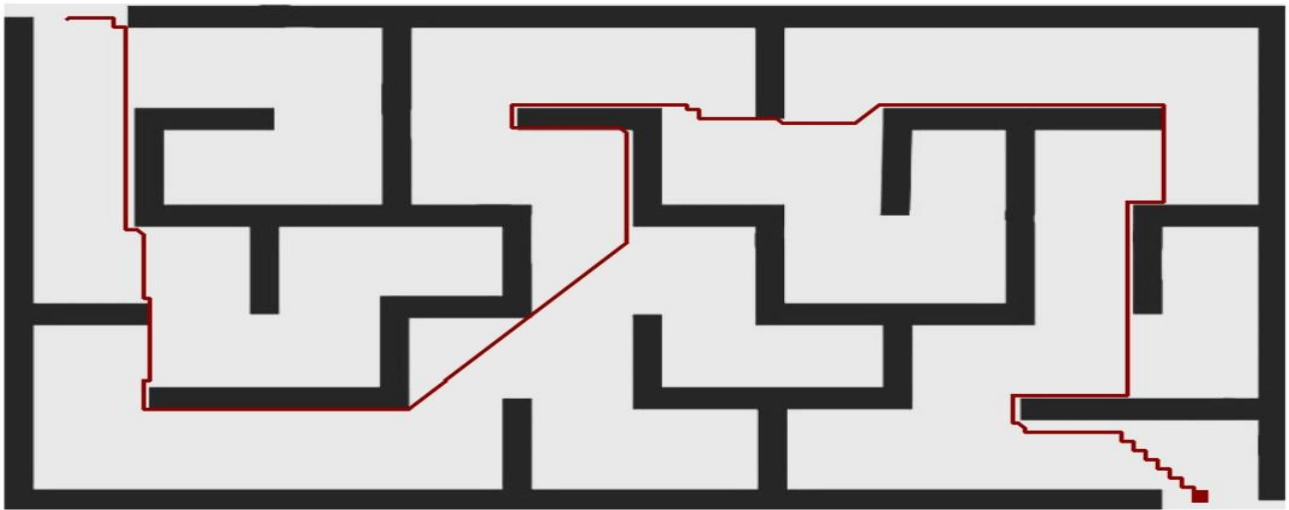


Start position: (1, 11)

Goal position: (108, 200)

The Beam Search algorithm, employing the Manhattan heuristic with a limit of 145, navigated the maze from the start at coordinates (1, 11) to the goal at (108, 200) by iteratively selecting the most promising paths. The chosen path, represented by coordinates such as (2, 11), (3, 11), and so on, was determined based on the heuristic's estimation of the remaining cost to reach the goal. With a beam width limit of 145, the algorithm pruned less promising paths, focusing on those with lower estimated costs. Consequently, the chosen coordinates on the final path reflect the algorithm's strategic exploration, favoring paths that seemed more likely to lead to the goal within the specified heuristic limit. This approach balances exploration and computational efficiency.

Euclidean Heuristic



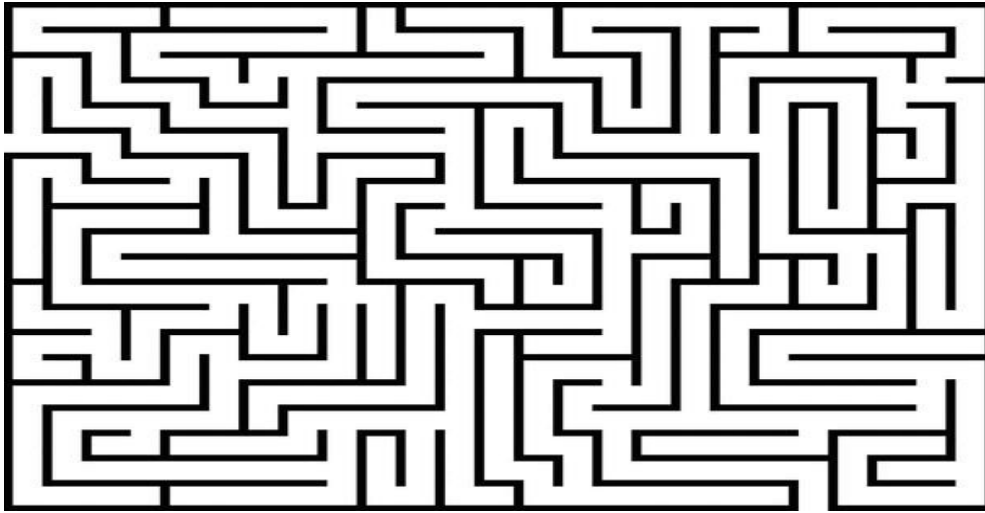
Start position: (3, 10)

Goal position: (107, 198)

The Beam Search algorithm, employed with the Euclidean heuristic and a limit of 145, successfully navigated the provided maze from the start position at (3, 10) to the goal at (107, 198). The resulting path exhibits a series of coordinates, each strategically chosen based on the heuristic evaluation. As the algorithm progresses, it consistently selects points that contribute to minimizing the Euclidean distance to the goal while adhering to the imposed limit of 145. The algorithm intelligently explores various possible routes simultaneously, but due to the limit, it ultimately prioritizes points that efficiently converge toward the goal within the specified constraints. The chosen coordinates in the final

path showcase the algorithm's ability to balance exploration and exploitation, opting for a route that is both promising and within the heuristic limit.

Maze 2



maze height: 122

maze width: 152

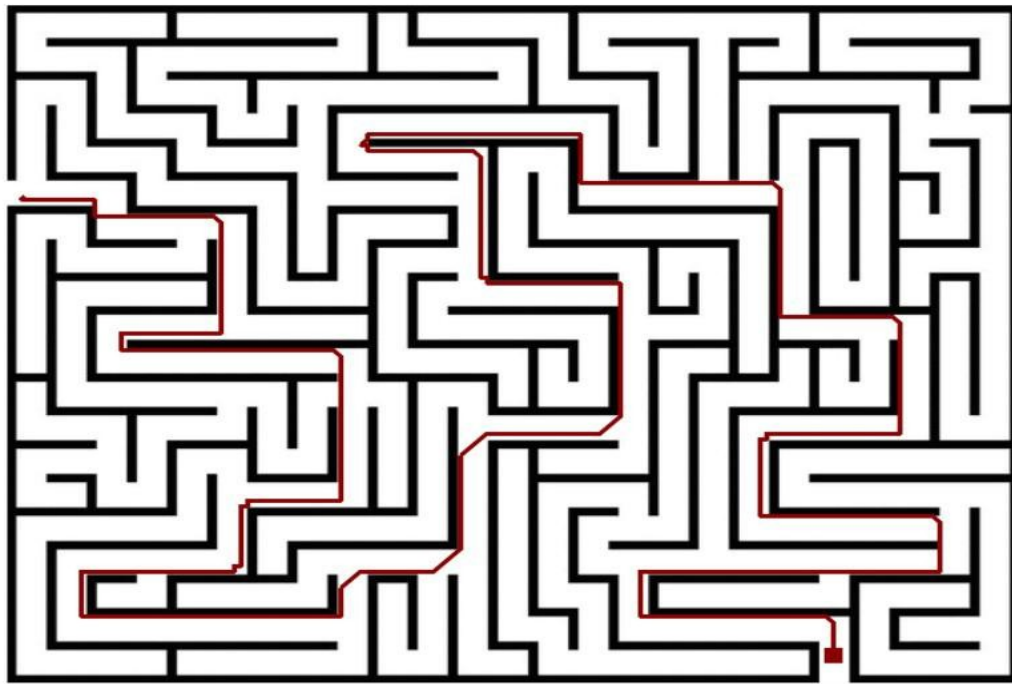
top-left coordinate:

(0, 0) bottom-right

coordinate: (122,

152)

1 Breadth First Search (BFS)



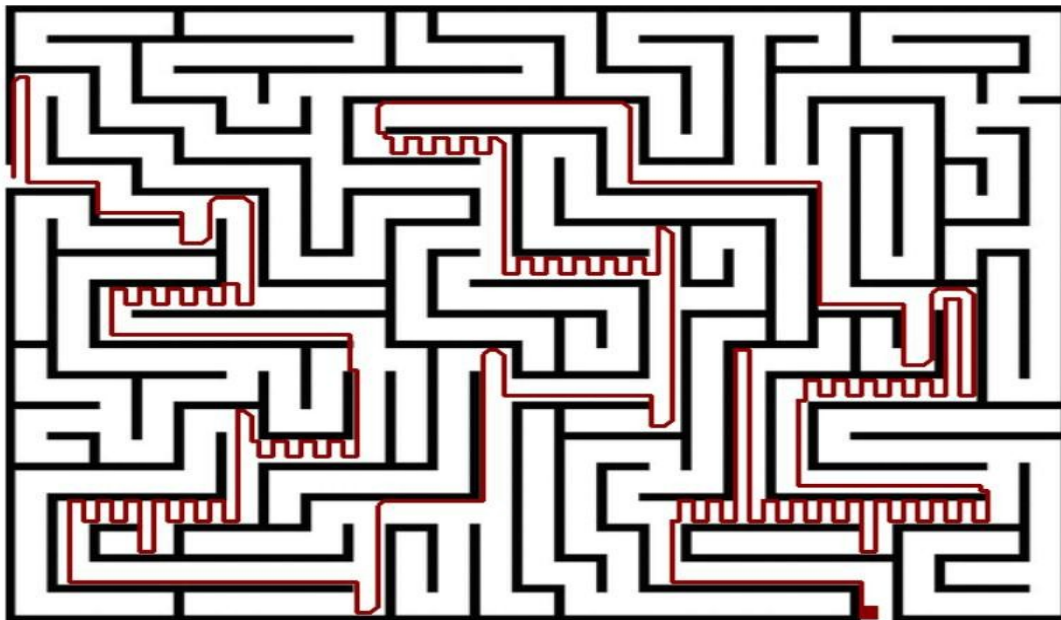
Start position: (34, 2)

Goal position: (117, 124)

The Breadth-First Search (BFS) algorithm, when applied to the binary matrix maze with a start position at (34, 2) and a goal position at (117, 124), successfully navigated through the maze's intricate pathways to discover the shortest route from the designated start to the end point. The algorithm's systematic

exploration method ensures that the chosen path optimally progresses from one point to the next, strategically evaluating adjacent positions based on their proximity to the current node. In the case of the provided coordinates, the BFS algorithm consistently moved towards the goal. This methodical approach is rooted in BFS's core principle of prioritizing the exploration of nodes in a layer-wise manner, guaranteeing that the algorithm efficiently explores and identifies the shortest path while avoiding unnecessary detours.

2 Depth First Search (DFS)



Start position: (34, 1)

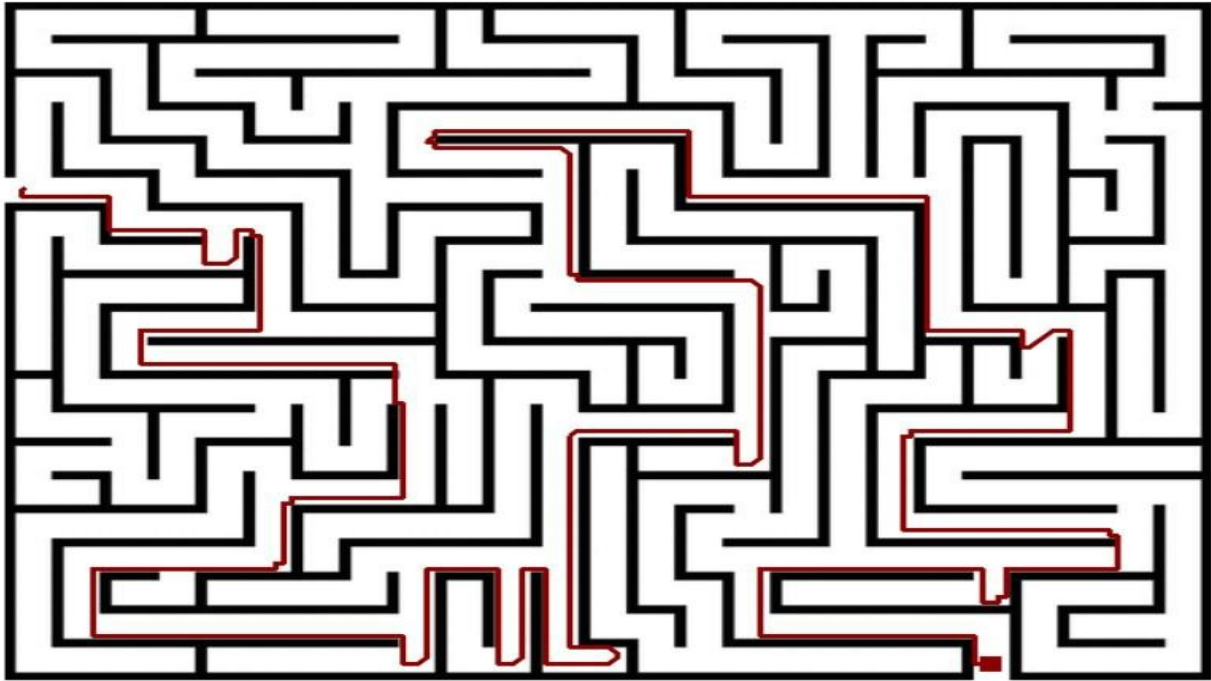
Goal position: (120, 123)

The Depth-First Search (DFS) algorithm is employed to navigate through the provided binary matrix maze, with the start position at (34, 1) and the goal at (120, 123). DFS explores as far as possible along each branch before backtracking, making it a suitable choice for maze traversal. The algorithm's selection of points is guided by the stack-based nature of DFS, favoring exploration along a single path until a dead end is reached. In this context, the points on the final path were chosen because they represented critical decision points where the algorithm opted for unexplored routes. The length of the

path is notably extensive due to DFS's inclination to delve deeply into a branch before considering alternative paths.

3 Greedy First Search

Manhattan Heuristic

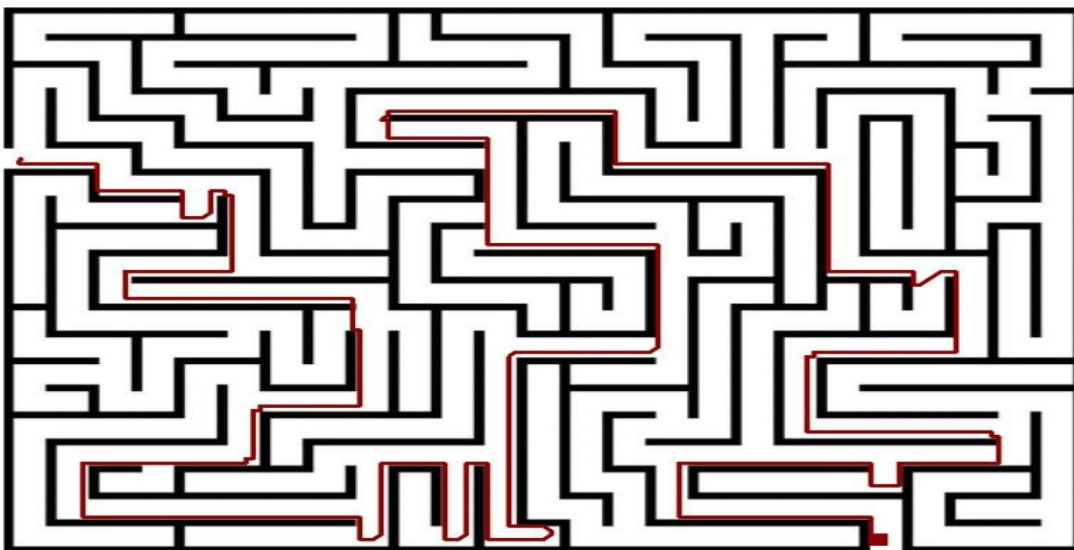


Start position: (33, 2)

Goal position: (119, 124)

The Greedy search algorithm with the Manhattan heuristic efficiently navigated through the binary matrix maze from the specified start point at coordinates (33, 2) to the goal point at (119, 124). The chosen path, consisting of coordinates such as (34, 2), (35, 2), (35, 3), and so on, demonstrates the algorithm's tendency to prioritize nodes with lower heuristic values. The Manhattan heuristic as mentioned previously, calculates the distance between two points by summing the absolute differences of their respective depth and width coordinates. As a result, the algorithm favorably selected neighboring nodes that progressively approached the goal while minimizing the heuristic function.

Euclidean Heuristic



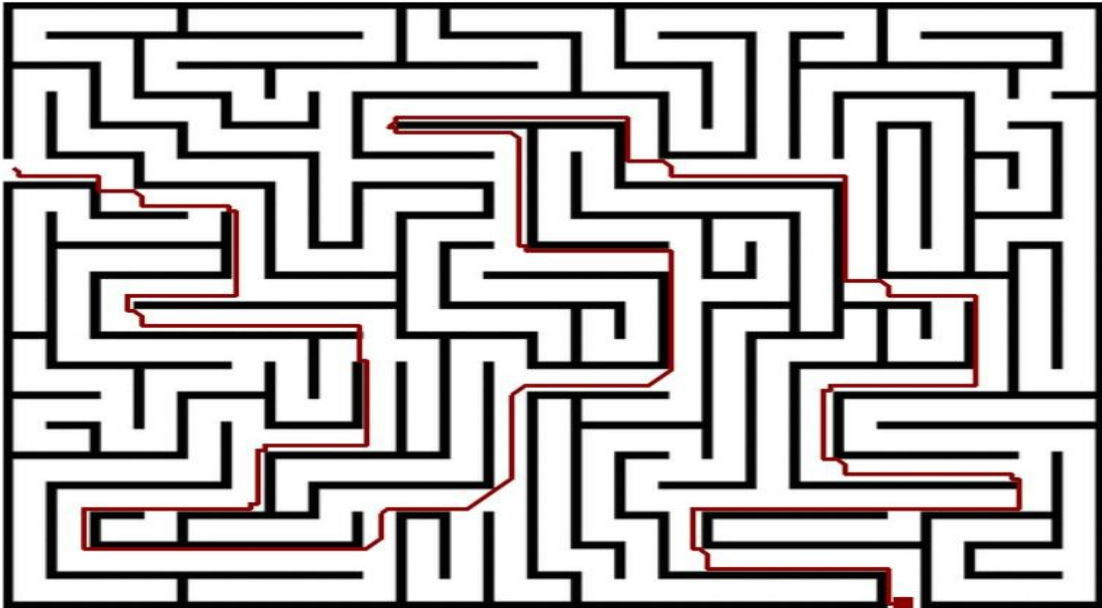
Start position: (34, 2)

Goal position: (119, 123)

The Greedy search algorithm with the Euclidean heuristic efficiently navigated through the binary matrix maze from the specified start point at coordinates (34, 2) to the goal point at (119, 123). This algorithm prioritizes nodes closer to the target, resulting in a path that predominantly moves upward, following the heuristic's inclination towards decreasing Euclidean distances. The chosen trajectory, including points like (35, 2), (35, 3), (35, 4), and so forth, highlights the algorithm's local responsiveness. Emphasizing immediate gains, it adapts to the local terrain, basing choices solely on Euclidean distances without considering obstacles or potential dead ends.

4 A* Search

Manhattan Heuristic



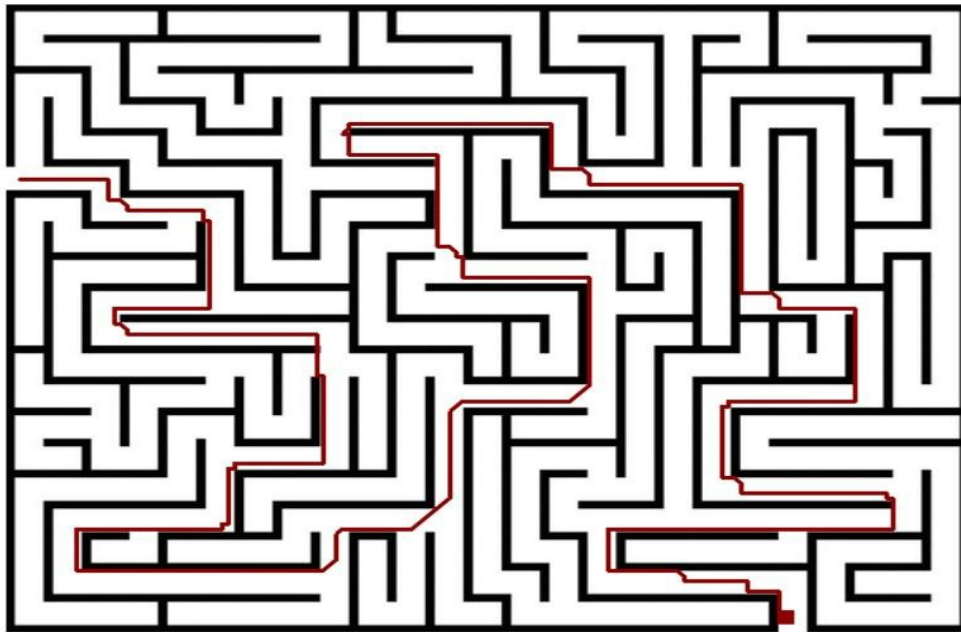
Start position: (33, 2)

Goal position: (121, 124)

The A* search algorithm, employed with the Manhattan heuristic, adeptly navigated the binary matrix maze from the designated start position at coordinates (33, 2) to the goal point at (121, 124). The resulting path showcases a systematic exploration of nodes, favoring those with lower Manhattan

distances and successfully circumventing obstacles. As it traverses through points like (35, 2), (35, 3), and (35, 4), the algorithm strategically selects nodes based on the heuristic's evaluation of the cost, considering the Manhattan distance from each point to the goal.

Euclidean Heuristic



Start position: (34, 2)

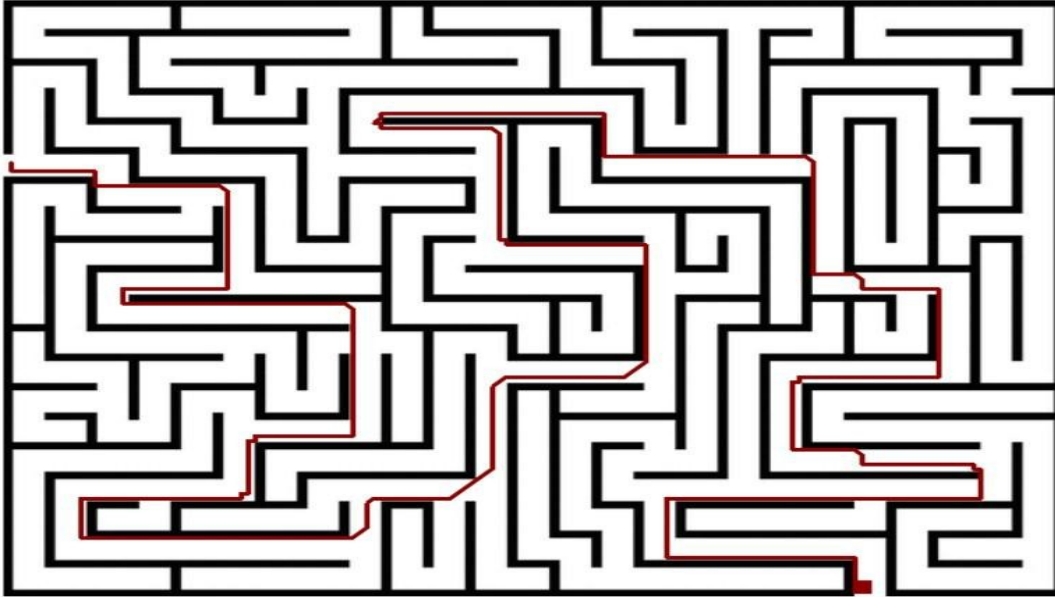
Goal position: (119, 123)

The A* search algorithm, augmented with the Euclidean heuristic, effectively navigated the provided maze from the start position at coordinates (34, 2) to the goal position at (119, 123). The resulting path, comprising coordinates such as (34, 3), (34, 4), ..., (119, 123), demonstrates the algorithm's ability to find the optimal route by intelligently balancing the cost of reaching the goal and the estimated remaining

distance. The Euclidean heuristic, which measures the straight-line distance between two points in a continuous space, guides A* towards the goal efficiently. The algorithm consistently favored positions that minimized the combined cost of the actual distance traveled and the estimated distance to the goal. This preference for optimal points allowed the algorithm to navigate around obstacles and adapt to the maze's complex structure. Notably, the path avoids unnecessary detours and selects positions that contribute most significantly to reaching the goal efficiently.

5 Weighted A* Search

Manhattan Heuristic

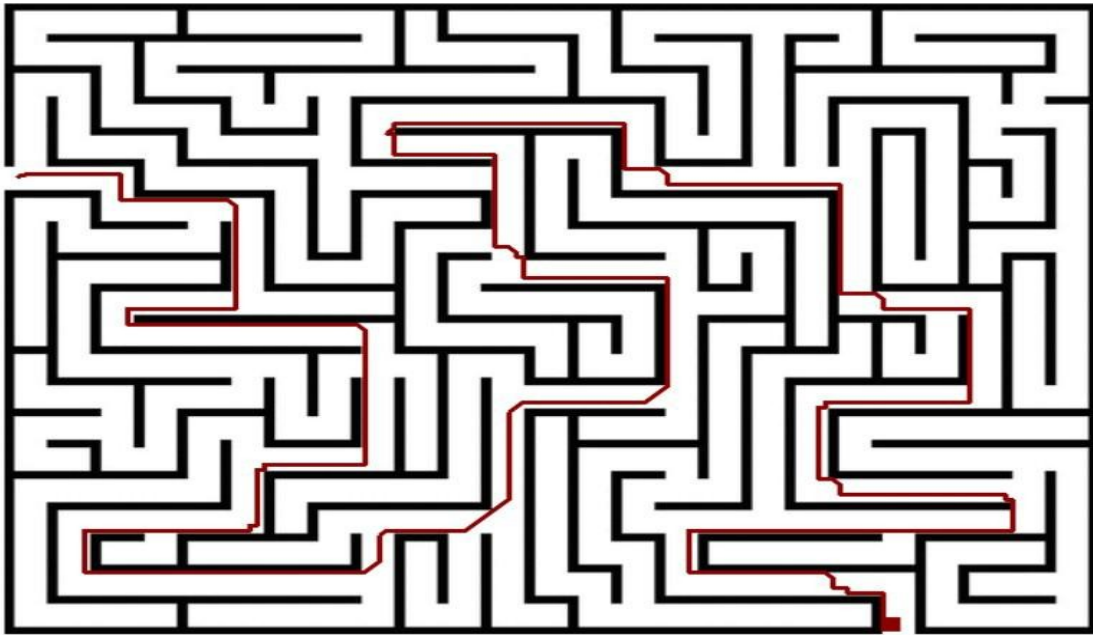


Weight: 1.5

Start position: (33, 1)

Goal position: (120, 123)

Euclidean Heuristic

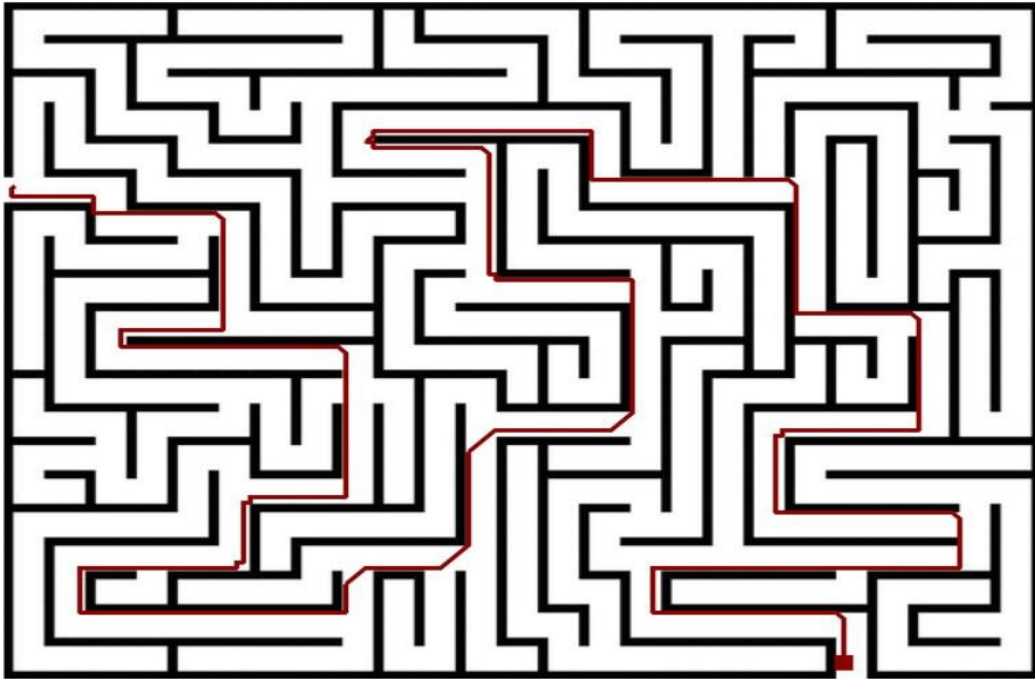


Weight: 1.5

Start position: (33, 2)

Goal position: (120, 123)

6 Uniform Cost Search (UCS)

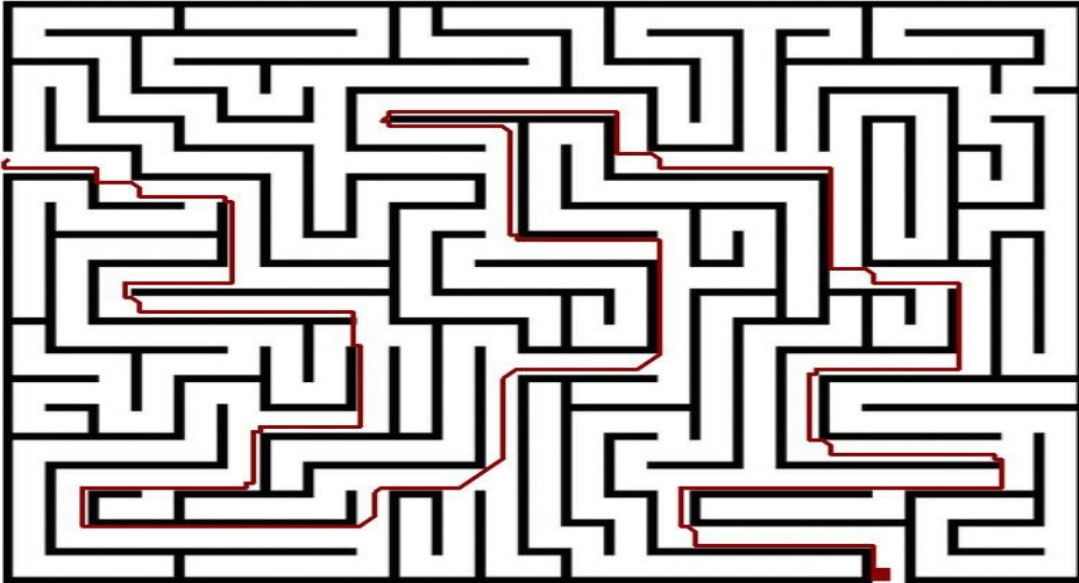


Start position: (33, 1)

Goal position: (119, 123)

7 Beam Search

Manhattan Heuristic

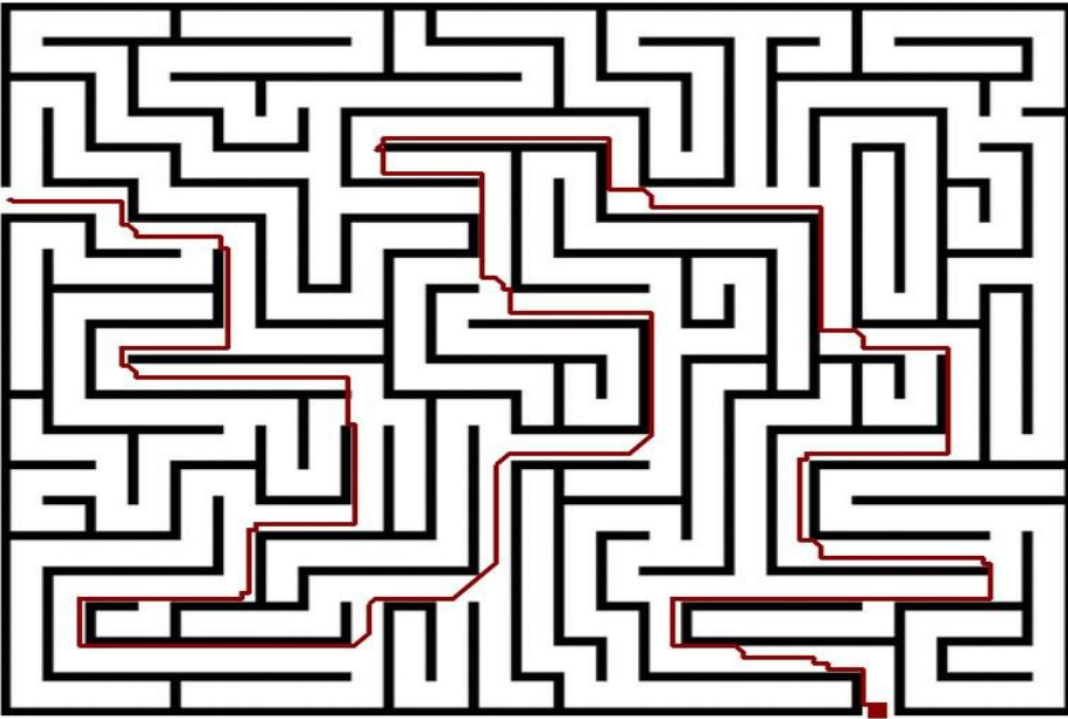


Limit: 145

Start position: (33, 0)

Goal position: (120, 123)

Euclidean Heuristic

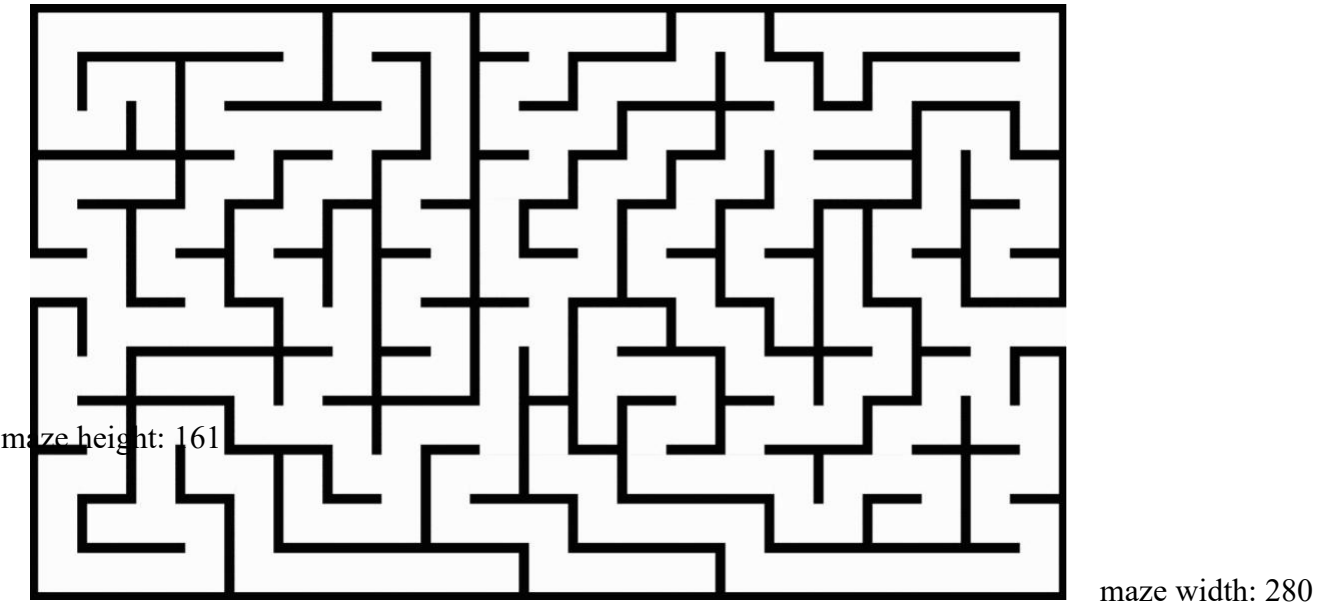


Weight: 145

Start position: (34, 1)

Goal position: (121, 124)

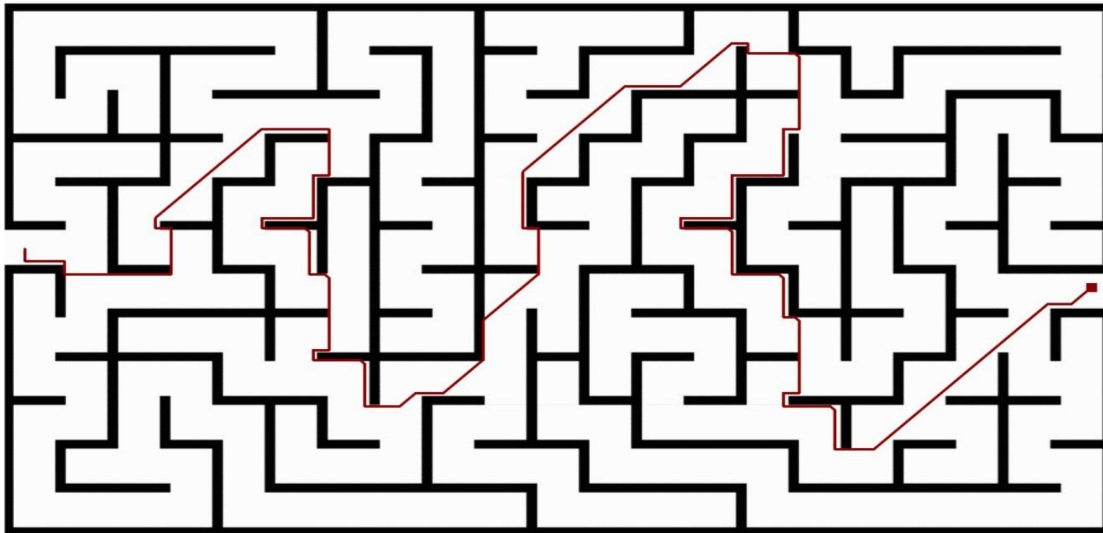
Maze 3



top-left coordinate:
(0, 0)

bottom-right
coordinate:(161,
280)

1 Breadth First Search (BFS)

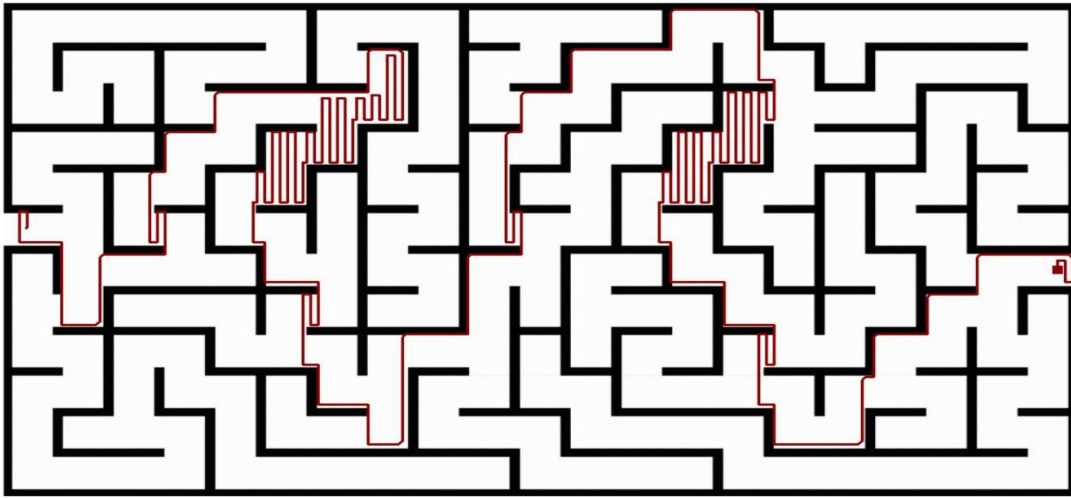


Start position: (74, 5)

Goal position: (86, 275)

The Breadth-First Search (BFS) algorithm is a powerful and systematic approach to finding the shortest path from a start to a goal in a maze. In the provided maze scenario, the algorithm begins at the start position (74, 5) and explores neighboring points systematically, layer by layer, gradually expanding outward. The goal position (86, 275) serves as the destination, and BFS ensures that the path discovered is the shortest possible. The algorithm navigates through the maze by considering all possible moves from the current position, prioritizing exploration of adjacent points before moving to farther ones. This ensures that the earliest viable path to the goal is identified, leading to the optimal solution.

2 Depth First Search (DFS)



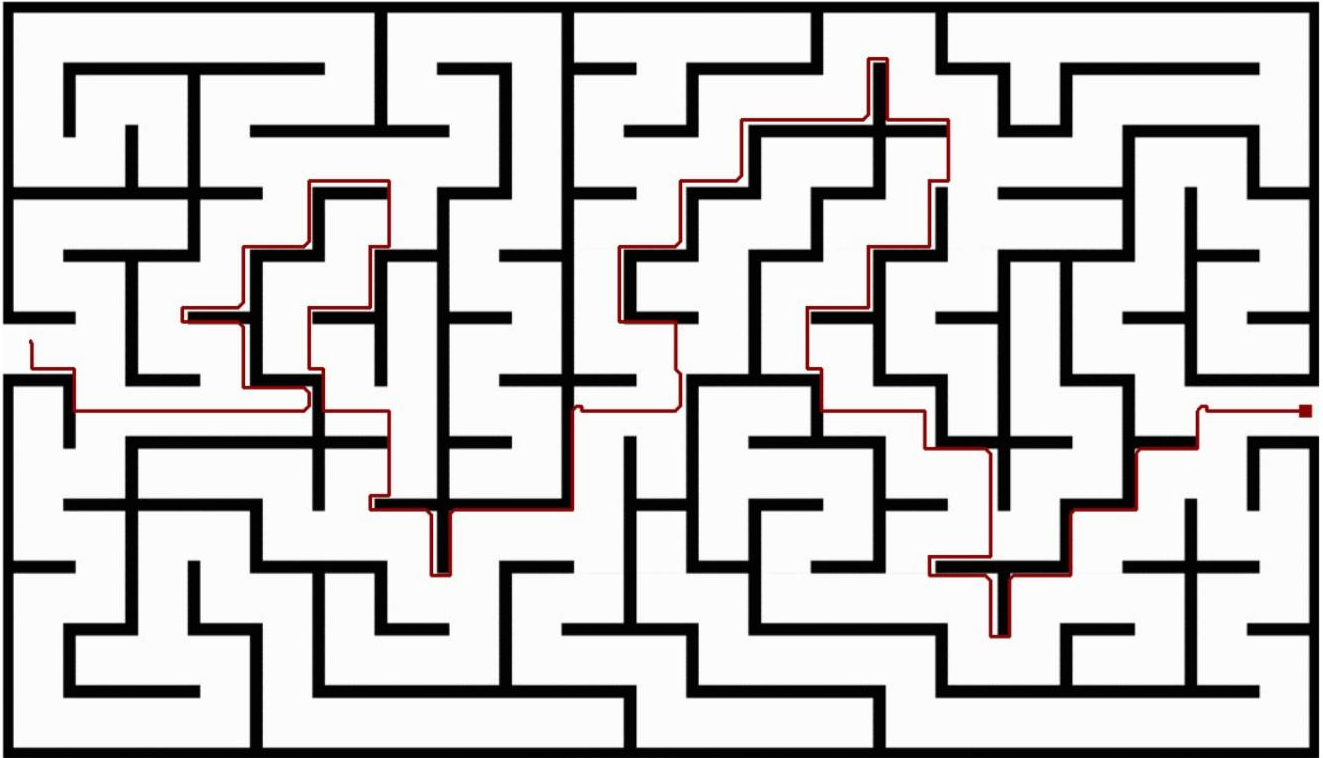
Start position: (74, 6)

Goal position: (87, 275)

The Depth First Search (DFS) algorithm was employed to find a path from the start position at coordinates (74, 6) to the goal position at coordinates (87, 275). DFS, as an uninformed search algorithm, systematically explores each branch of the search tree before backtracking. In this context, the algorithm starts from the initial point and explores adjacent points, prioritizing the deepest unexplored node along each branch. The resulting path captures the decision process of DFS, with the algorithm navigating through the maze by consistently choosing the path with the deepest unexplored options. The selected points in the partial path are influenced by the algorithm's strategy of moving as far as possible along a branch before backtracking.

3 Greedy First Search

Manhattan Heuristic

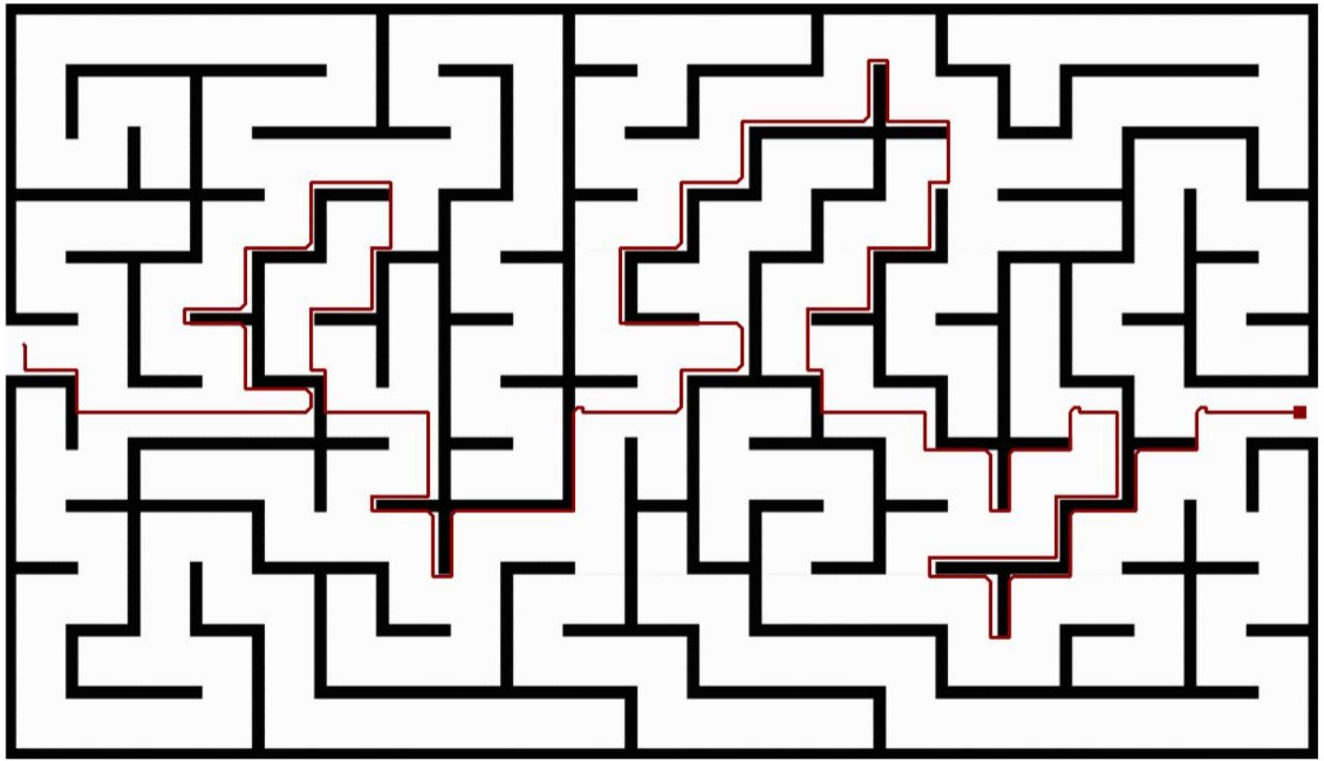


Start position: (72, 6)

Goal position: (87, 277)

The Greedy Search algorithm with the Manhattan heuristic effectively navigated the provided maze, determined by the binary matrix. Starting from the coordinates (72, 6) and aiming to reach the goal at (87, 277), the algorithm produced a path with coordinates [(73, 6), (74, 6), ..., (87, 277)]. The Manhattan heuristic measures the distance between two points as the sum of the absolute differences in their vertical and horizontal positions. In the context of this algorithm, the heuristic guides the search by favoring paths that are closer to the goal based on this distance metric.

Euclidean Heuristic



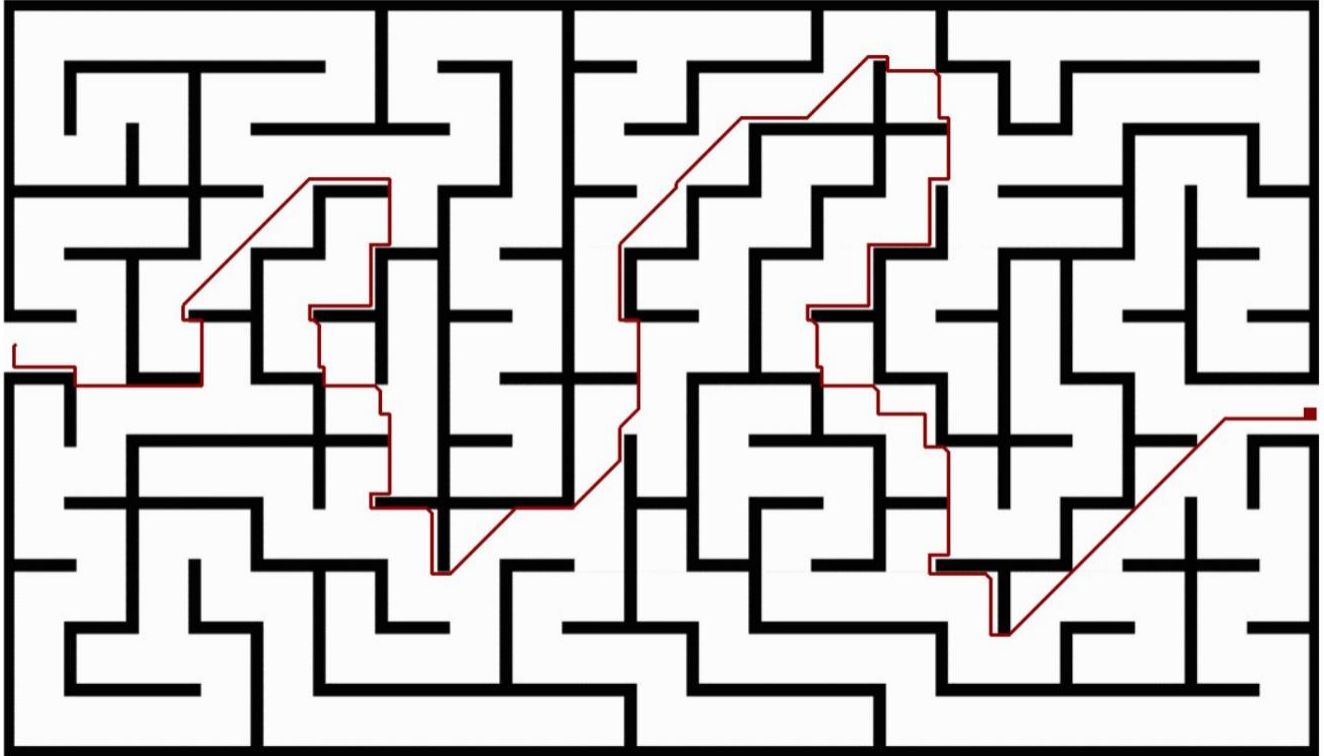
Start position: (72, 4)

Goal position: (87, 276)

The Greedy Search algorithm, employing the Euclidean heuristic, efficiently navigated the provided maze from the specified start point at (72, 4) to the goal point at (87, 276). The algorithm prioritizes exploration based on the Euclidean distance from the current position to the goal, favoring paths that seemingly lead directly toward the target. This is evident in the selected coordinates forming the final path, such as (73, 4), (78, 14), (87, 15), and so forth. The Euclidean heuristic calculates the straight-line distance between two points, essentially allowing the algorithm to make locally optimal choices by moving toward the goal in a straight-line fashion. Notably, the algorithm exhibits a greedy nature, consistently choosing the most immediately appealing option at each step without considering the global context.

4 A* Search

Manhattan Heuristic

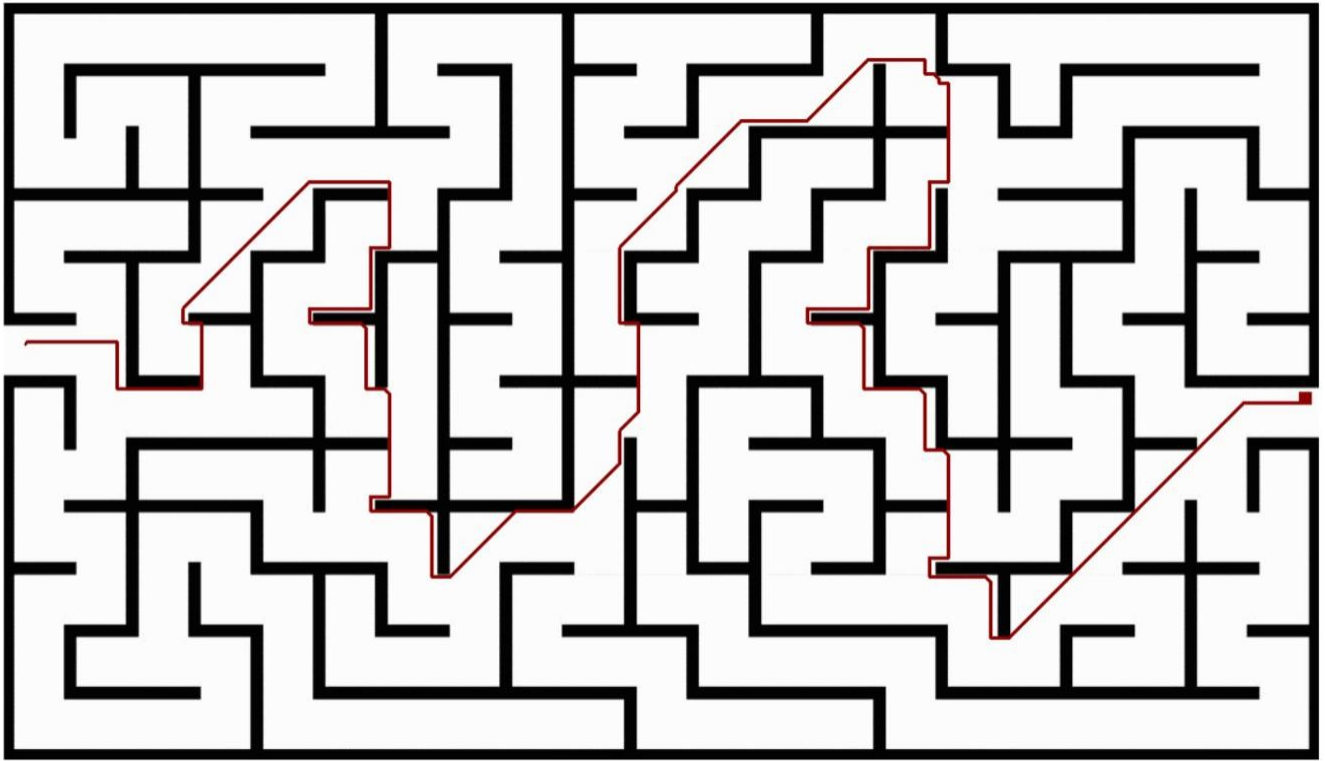


Start position: (73, 2)

Goal position: (88, 278)

In the A* Search algorithm with the Manhattan heuristic, the chosen path from the provided start position at coordinates (73, 2) to the goal at (88, 278) reveals an optimal route through the maze. The heuristic, which estimates the cost from a given point to the goal based on the Manhattan distance (sum of horizontal and vertical distances), guides the algorithm in selecting the most promising paths. As the algorithm explores different options, it intelligently prioritizes nodes that are likely to lead to the goal efficiently. In the specific path outlined, the algorithm follows a continuous and direct trajectory, making use of the maze's structure to minimize the number of steps taken. The chosen coordinates in the final path reflect the algorithm's decision-making process, favoring nodes that consistently reduce the cumulative cost and bring it closer to the destination. The path avoids unnecessary detours and demonstrates the algorithm's ability to navigate efficiently through the maze's intricate layout

Euclidean Heuristic



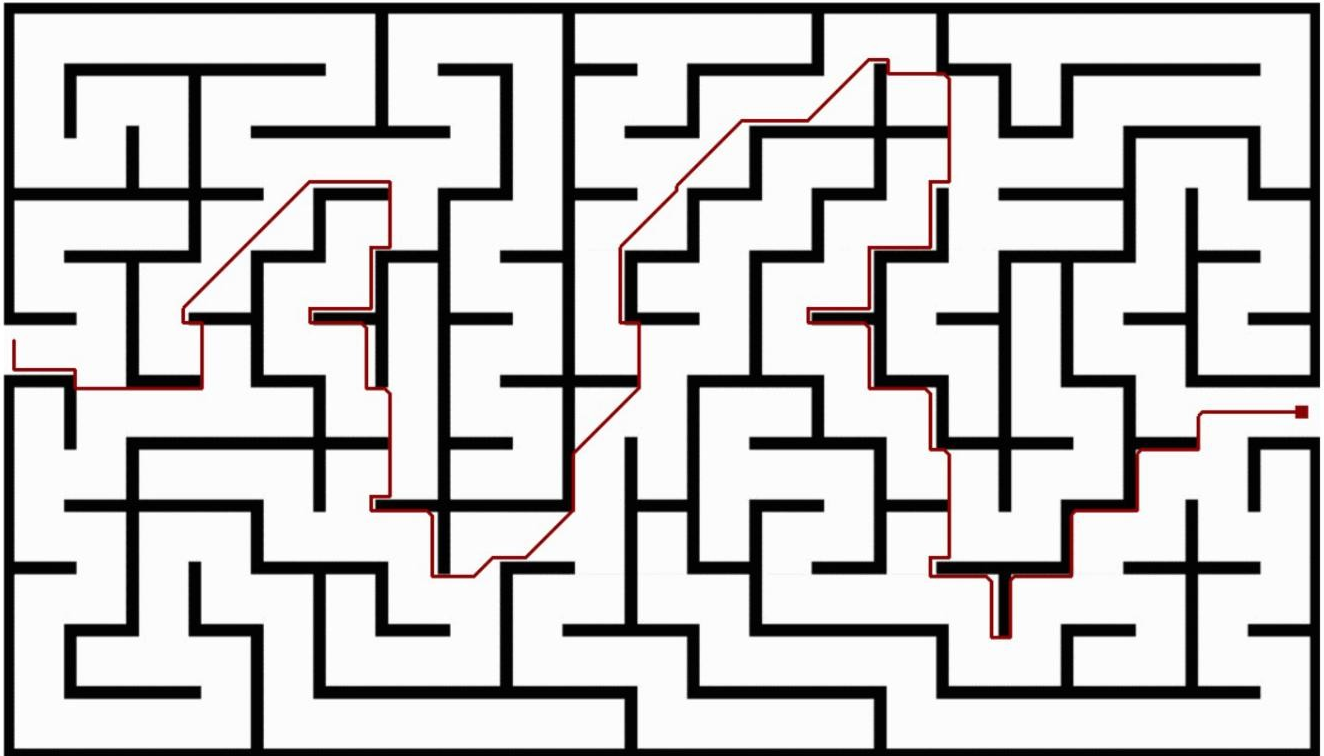
Start position: (72, 4)

Goal position: (84, 277)

In the A* Search algorithm with the Euclidean heuristic applied to the provided maze, the path from the start position at (72, 4) to the goal position at (84, 277) exhibits a clear trajectory through specific coordinates. Notably, the algorithm tends to favor paths that minimize both the cost to reach the current point and the estimated cost from that point to the goal. The Euclidean heuristic, which calculates the straight-line distance between two points, guides the algorithm in selecting points that offer a more direct route to the goal. Consequently, the path takes a series of continuous and gradually changing coordinates as shown.

5 Weighted A* Search

Manhattan Heuristic

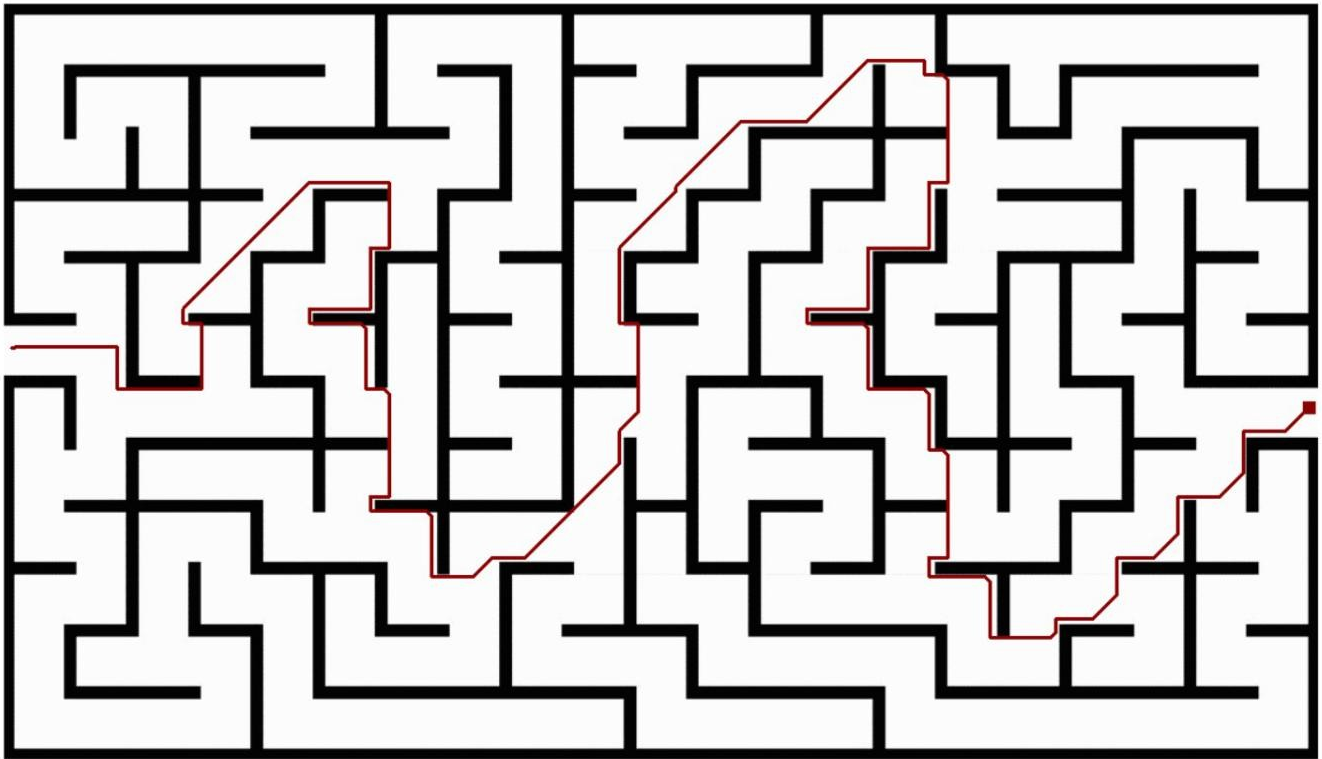


Weight: 1.4

Start position: (72, 2)

Goal position: (87, 276)

Euclidean Heuristic

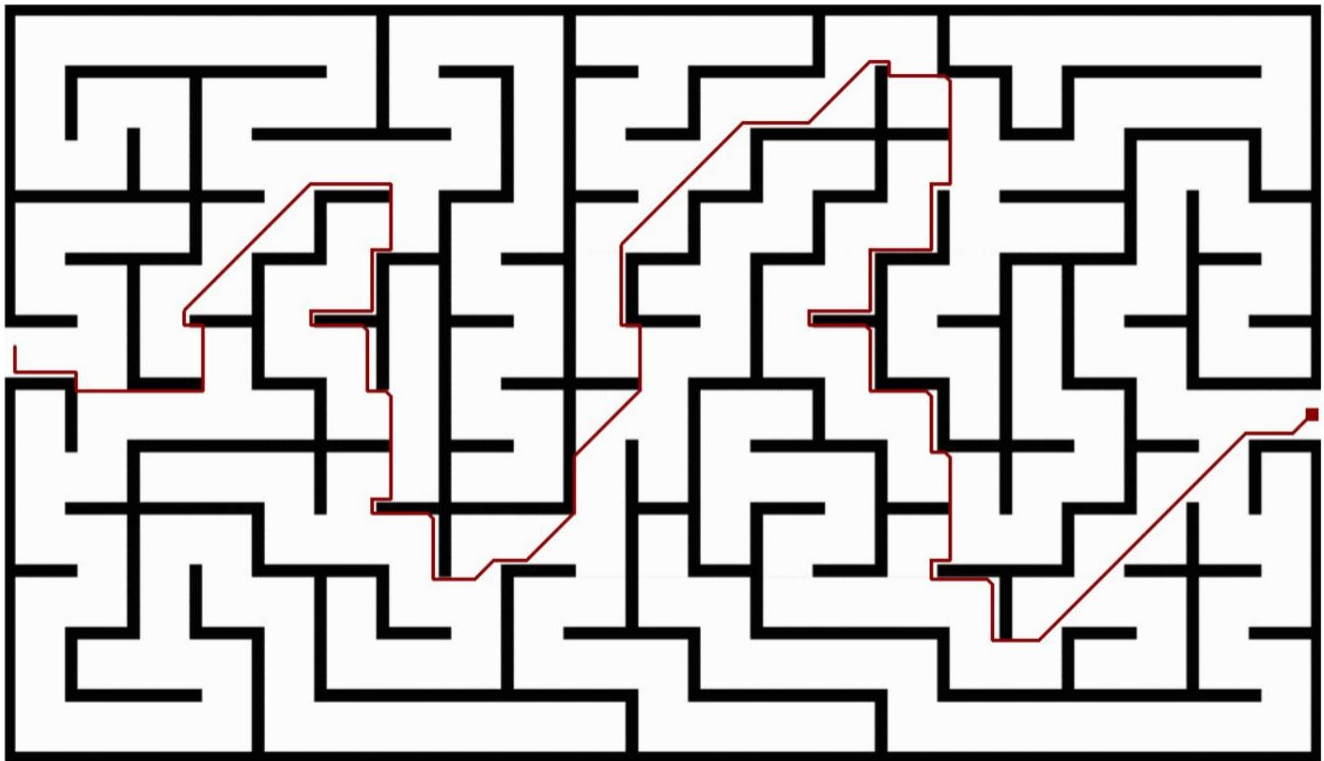


Weight: 1.4

Start position: (73, 2)

Goal position: (86, 278)

6 Uniform Cost Search (UCS)

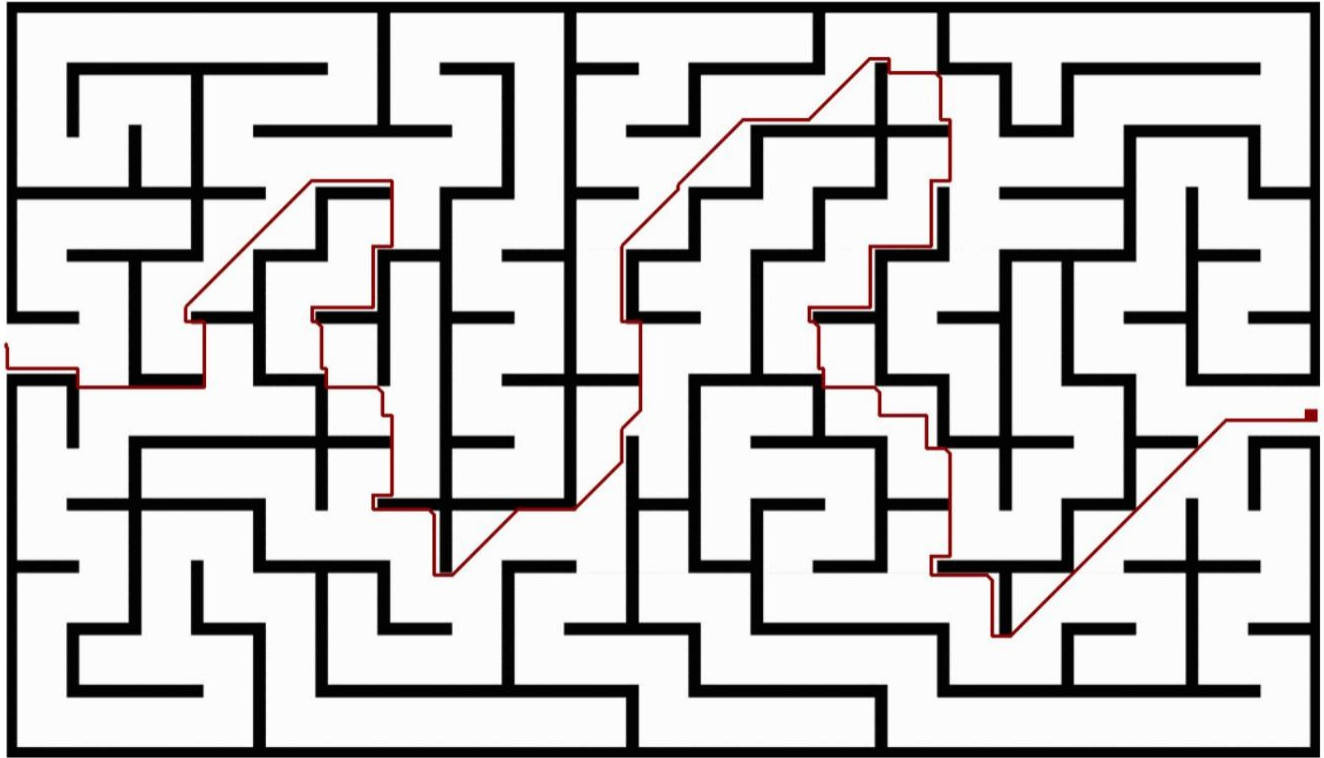


Start position: (72, 2)

Goal position: (87, 278)

7 Beam Search

Manhattan Heuristic

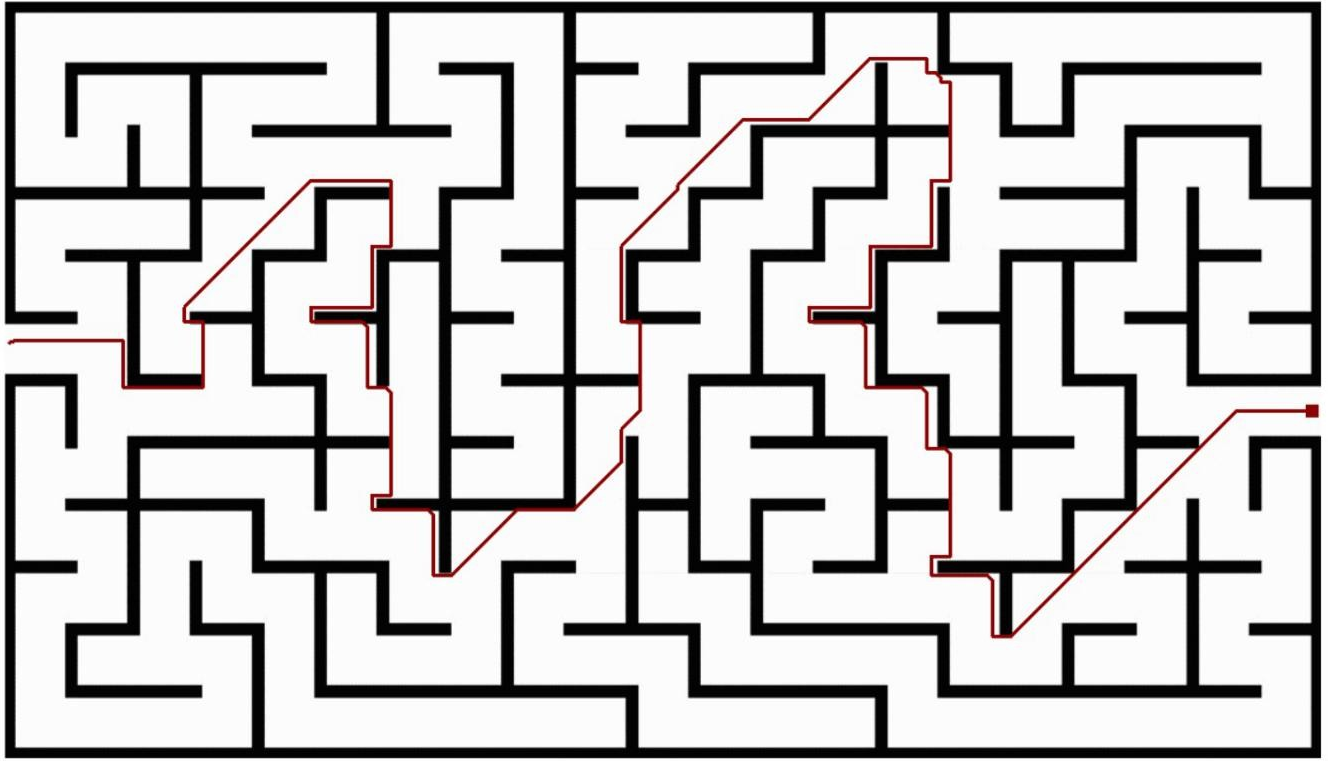


Limit: 170

Start position: (73, 0)

Goal position: (88, 278)

Euclidean Heuristic



Limit: 170

Start position: (72, 1)

Goal position: (87, 278)

DISCUSSION

In this project, we mainly used seven different searching strategies, which is a small proportion of the real number of strategies. In order to deliver a clear idea of why such a huge number of searching algorithm exists, we are going to examine the advantages and the disadvantages of each of the seven algorithms that we adapted.

Before diving into the details, we will clarify some notations and terminologies for the unexperienced reader:

- An algorithm is called informed, it uses a heuristic $h(n)$ to look for the goal, otherwise it is considered uninformed or blind

- Since the execution of search is done using a tree, we will denote the depth of that tree by m , the degree by b and the depth of the shallowest goal by d

- Big O notation, which focuses on the worst-case scenario will be used for space and time complexity.

- 1 Breadth-First Search (BFS):

A blind search algorithm that examines the surface then dig one level deeper. It is complete for all kinds of search spaces, optimal if the costs for all actions are identical, which is the case in our project, but both time and spaces complexities are $O(b^d)$, which is too much. For this reason, this algorithm is only useful for small problems and where no heuristic can be established.

- 2-DFS:

It is blind and begins with the depth of the tree. It saves a lot of memory unlike BFS as it expands less node for each level. However, it is only complete in finite state spaces if it at least checks for

cycles to avoid getting caught, while in an infinite state space, it can never be complete. Moreover, it is not optimal as it does not consider any kind of cost. The time complexity is $O(b^d)$, like BFS wherever the space complexity is $O(bd)$.

3-Uniform-Cost Search (UCS):

An uninformed search algorithm that prioritizes the nodes according to their path cost. This technique is useful each action has a different cost. It is complete if and only if there is ϵ such that no action costs less than ϵ . Obviously, this limit is set to prevent the search from going down an infinite path of decreasing costs. Optimality is guaranteed by this algorithm since it always considers the minimum cost path. Moreover, the time and space complexities are $O(bC^*)$, where C^* is the optimal cost.

4-Greedy Search:

A type of informed search that relays completely on the heuristic $h(n)$. This technique is only complete when checking for redundant path is included in the implementation (which is the case for our program). However, it is not optimal as it rushes towards the goal without taking the consequence into account. The time and space complexity range from $O(bd)$ to $O(b^d)$ based on the quality of $h(n)$.

5-A* Search:

Searches using $f(n)$ as an evaluation function where $f(n) = h(n) + g(n)$. It is always complete, but optimal only when $h(n)$ is not overestimated (admissible). The time and space complexities in this case depend on several factors here, but in the worst-case it is

$O(b^d)$. However, in most cases it performs way more efficiently.

6-Weighted A* Search:

A* expands almost every node looking for an optimal path. This consumes time and memory. For this purpose, weighted version of A* ruins the optimality to reach a suboptimal but a satisfactory solution. This is done by adding a weight **W** in order to overestimate **h(n)**, so the evaluation function will be **f(n)= W*h(n)+g(n)**.

7-Beam Search:

An implementation of A* that limits the frontier in order to save memory. It keeps the k best nodes in the frontier and remove any extra ones. Of course, this modification sacrifices the optimality of A*. Moreover, a poor choice of k will cause the search to fail.

CONCLUSION

The main algorithms we have discussed were breadth-first search, depth-first search, uniform-cost search, greedy search, beam search, A* search and even weighted A* search. The first three are uninformed while the others are informed (using two different types of heuristics). Then, the results of each algorithm were provided and discussed exhaustively. Finally, they were compared with each other with a valid main criterion. In conclusion, after exploring the extensive and distinctive search algorithms, we can all agree that each of these algorithms solve various search problems. Our program only proved to work on rectangular mazes and solves them according to the algorithm chosen; however, the same concept can be applied to any search problem, even complex real-world ones.