MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
V.N. Karazin Kharkiv National University
School of Mathematics and Computer Science
Department of Theoretical and Applied Informatics

Master's Thesis

Maze solving algorithms

Author:
Final year Master's Program student,
group MCS-64
specialty - Computer Sciences and
Information Technologies,
educational program: "Informatics"
Jiang Hao
Supervisor: Anna Goncharuk
Reviewer: Andrii Chukhrai

Kharkiv, 2024

Table of Contents

1. INTRODUCTION

Maze games are a classic genre. With technological advancements, maze-solving techniques have found applications in various fields like robot navigation, path planning, graph theory, and even chip design. The most common application is in robot navigation.

Robotics has continuously driven the development of maze-solving algorithms. In the 1980s, researchers applied traditional maze-solving ideas to robot navigation. The simple "Hand-on-Wall" algorithm was among the first used ([7], p.38). This algorithm, as a search strategy, has many advantages and is widely used. However, it also has limitations, leading to the development of many other algorithms suitable for various scenarios. Some are based on improvements to "Hand-on-Wall," while others are based on computer versions of classic maze-solving algorithms like Trémaux and Pledge. There are also graph-based maze-solving algorithms like A* search ([6] chap. 3.5.2), depth-first search, and breadth-first search.

Besides robotics, maze-solving can be applied in seemingly unrelated fields like graph theory and network analysis. Here, mazes are abstracted into graph structures. Maze-solving algorithms analyze graph connectivity, identify shortest paths, and explore network topology. In chip design, these algorithms optimize the layout of internal circuits, minimizing signal delay and power consumption. For example, they identify the best connection paths between different modules on a chip, reducing signal transmission distance and improving chip performance.

This paper mainly explores solving mazes without prior knowledge of the maze structure. Therefore, algorithms relying on complete maze information, such as graph-based search algorithms, are not within the scope of this paper.

2. MAIN CONCEPTS

This paper aims to systematically analyze and summarize computer algorithms based on classic maze-solving methods, including "Hand-on-wall," Trémaux, and Pledge algorithms. We will analyze their basic principles, advantages, disadvantages, and suitable application scopes. Then, we compare and analyze the performance of different algorithms in efficiency, robustness, implementation difficulty, and resource consumption, examining their applicability to various maze types. Additionally, this paper explores the current research status and future development trends of maze-solving algorithms, providing insights into their potential applications in robot navigation, path planning, and other related fields.

The structure of this paper is as follows: First, we introduce the "Hand-on-Wall" algorithm, prove its effectiveness, and analyze its principles, advantages, disadvantages, and scope of application in detail. We then discuss some enhancements to this simple algorithm. Subsequently, we introduce the "Pledge algorithm" and "Trémaux algorithm," analyzing their principles, advantages, disadvantages, and application scope. Then, we conduct a theoretical comparison and evaluation of these three algorithms and use pyamaze (https://github.com/MAN1986/pyamaze) to randomly generate a large number of mazes to test the performance of these algorithms. Finally, this paper discusses the future directions of maze-solving algorithm research and prospects their application in various fields.

Hand-on-Wall Algorithm

Before formally introducing the Hand-on-Wall algorithm, let's discuss a naive method that can always find the correct path as long as the maze is connected. You simply wander randomly in the maze, and with enough time, you will eventually reach the end. This is the simplest maze-solving algorithm: the random walk algorithm.

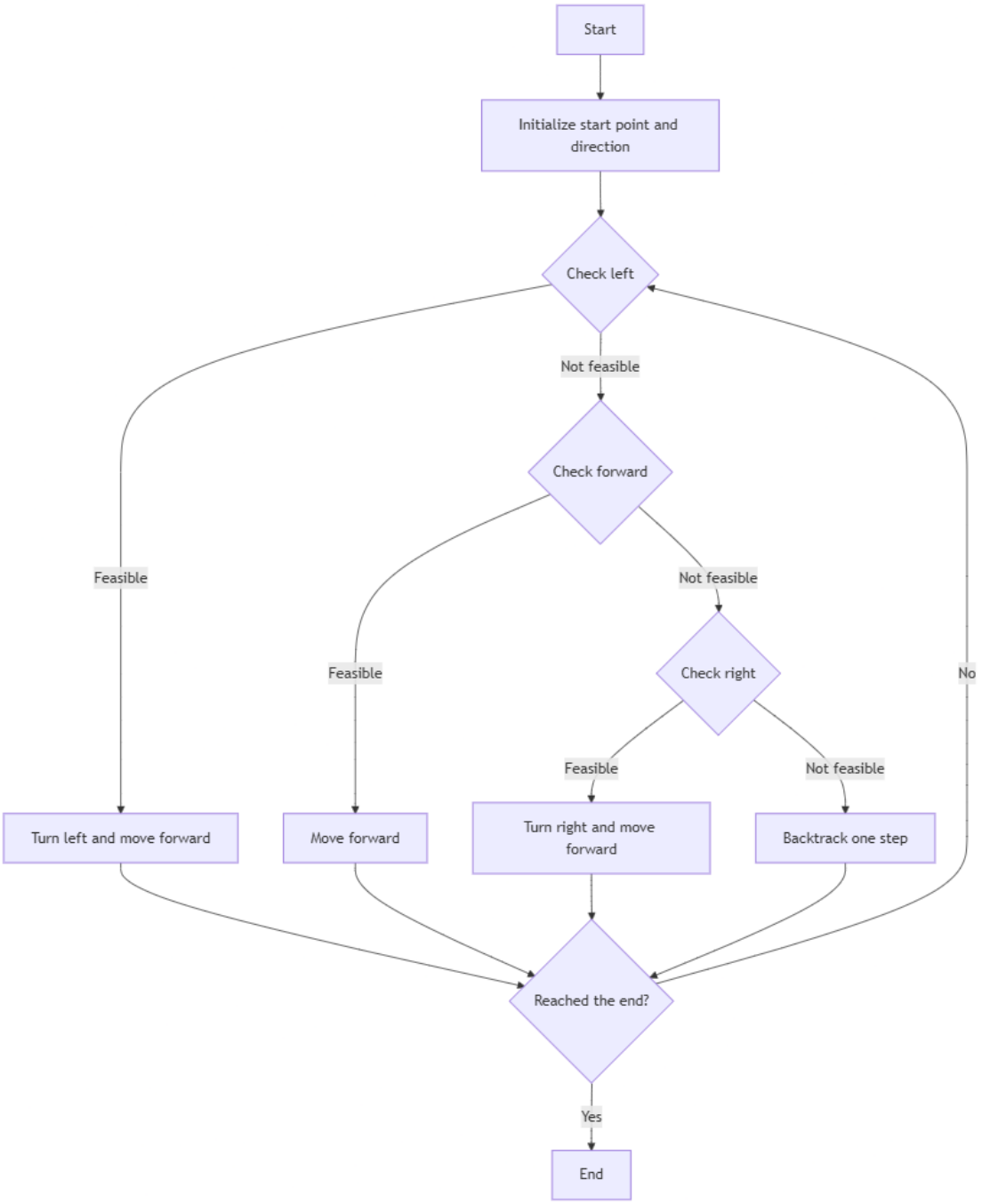
While the random walk algorithm can always find a path, the problem is also obvious: it's too slow, so it has no practical value.

The Hand-on-Wall algorithm, derived from an ancient and classic maze-solving method, is also one of the earliest maze-solving algorithms used. Due to its simplicity, ease of implementation, low performance requirements for devices, and no reliance on additional sensors, it still has many practical applications today.

The implementation of the Hand-on-Wall algorithm is very simple, the core of which is only choosing one side of the wall, which is also the only decision-making step of this algorithm.

You just need to repeat the following steps until you reach the maze exit:

1. Choose to stick to the left or right wall.
2. Enter the maze, keeping the robot's left or right side (as chosen in step 1) in contact with the wall.
3. Keep sticking to the wall and move forward.
4. When you need to turn, turn along the wall and continue moving forward.



The Hand-on-Wall algorithm has many advantages. It is simple and very easy to understand and implement. It does not require storing a map, so it has no memory requirements in this regard. It can even be applied to dynamic mazes to a certain extent, because it will not change its rules due to external interference. After the interference disappears, it will continue to follow its original path.

Let's prove that the Hand-on-Wall algorithm can solve any connected and loopless two-dimensional maze.

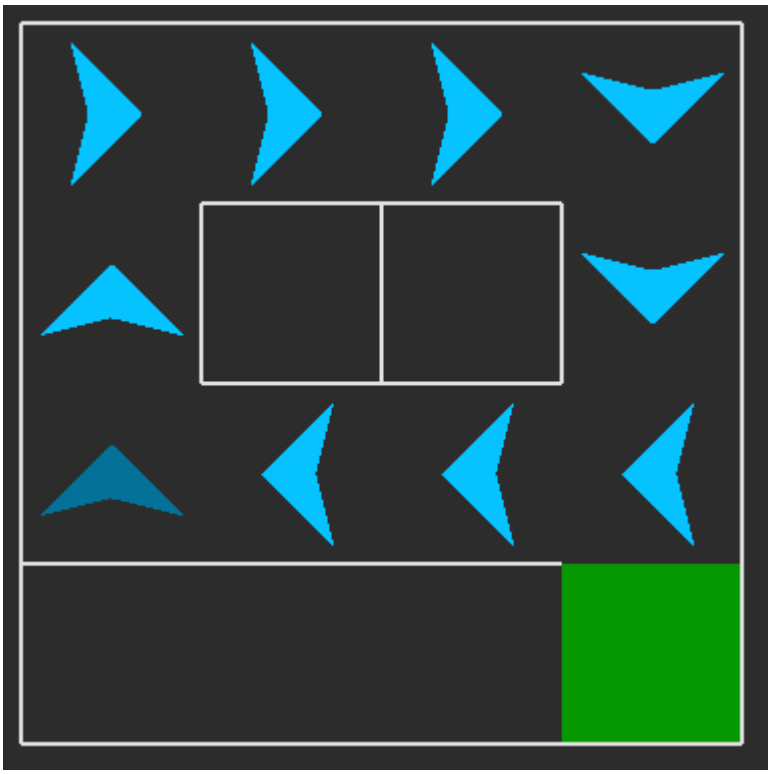
To prove its effectiveness, consider a connected and loopless maze where all walls are interconnected or connected to the outer boundary of the maze. The wall-following algorithm requires always maintaining contact with the maze walls. Due to the simple connectivity of the maze, the walls can be considered a continuous loop. Traversing along the walls is equivalent to moving within this loop. Due to the finiteness of the loop, the algorithm will eventually return to the starting point or find another exit (if it exists).

In a maze with only one exit, the algorithm will follow the walls until it reaches the exit. In a maze with multiple exits, the algorithm will follow the walls until it encounters one of the exits. Therefore, regardless of the number of exits, the wall-following algorithm guarantees finding an exit.

Even in mazes with loops, the wall-following algorithm guarantees returning to the starting point. This is because the algorithm treats the maze as a graph, using the wall-following strategy to revisit every intersection and path. In the absence of an exit or a feasible path to an exit (for example, in a disconnected part of the maze), the algorithm will traverse all reachable paths and eventually return to the starting point after thoroughly exploring the boundaries.

The limitation of the Hand-on-Wall algorithm is that it cannot solve mazes with loops or "islands" because it may fall into an infinite loop and fail to find the exit.

Here's a simple example of the Hand-on-Wall algorithm getting stuck in an infinite loop within a maze:



Start at the (1,2) and move to the right, using the right-hand rule, you will fall into an infinite loop and never reach the end point in the lower right corner. In fact, if you use the left-hand rule, you can successfully solve the maze.

To address these shortcomings, some improvements can be made. For example, you can randomly choose the wall-sticking direction instead of always choosing the left or right side, thereby reducing the possibility of entering a loop. Additionally, recording the previously traversed path can prevent revisiting the same path, improving solving efficiency and avoiding falling into loops.

Here's an example of an modified Hand-on-Wall algorithm that records the traversed path and backtracks when it reaches a repeated point and alternates between left-hand and right-hand rules to avoid loops:

1. Choose the maze entrance as the starting point, specify the initial direction, initialize the path record and turn counter.
2. Dynamically switch between the left-hand rule and the right-hand rule based on the parity of the turn counter. Use the left-hand rule for even turns and the right-hand rule for odd turns.

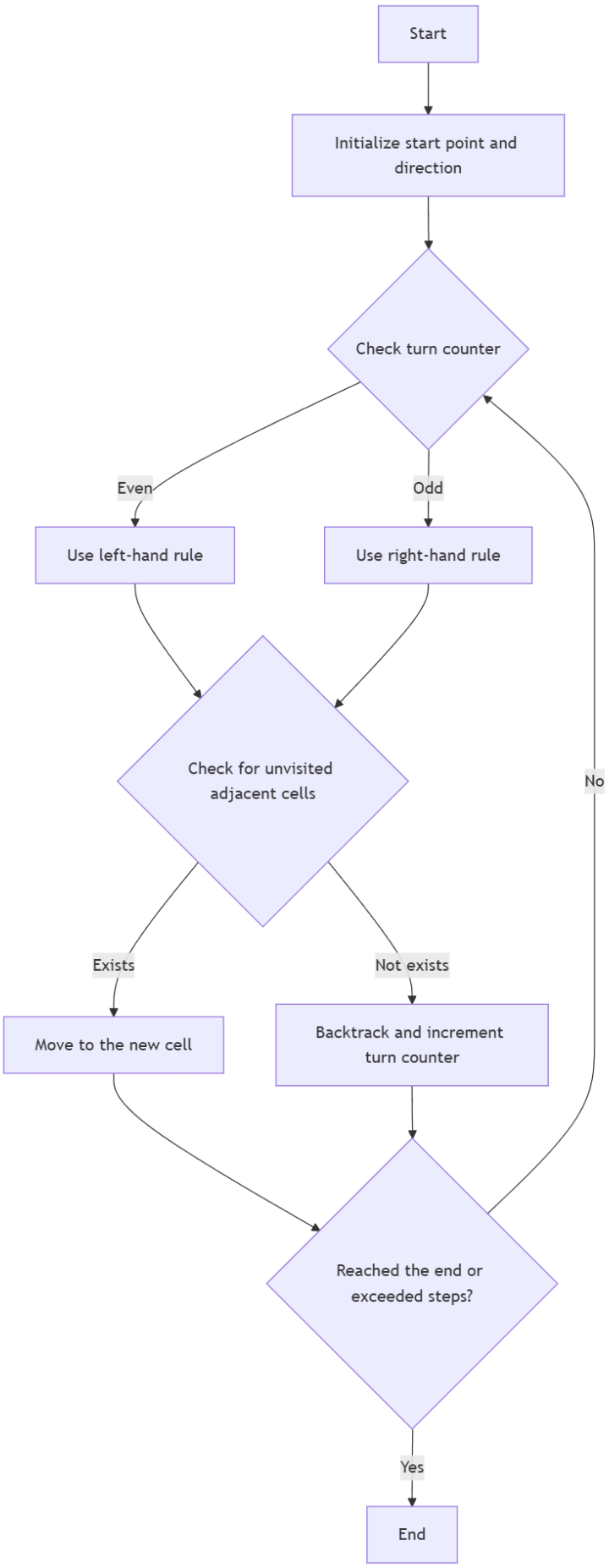
3. When an unvisited location is discovered, add the current location to the path record. If no feasible direction is found, remove the current location from the path dictionary, backtrack, and increment the turn counter.
4. Repeat until the maze exit is reached.

The counter used here can alternate between the left-hand rule and the right-hand rule. This counter is optional and is only used to approximate the "random direction selection" mentioned above. Even without using a counter, just recording the traversed path can avoid falling into a loop.

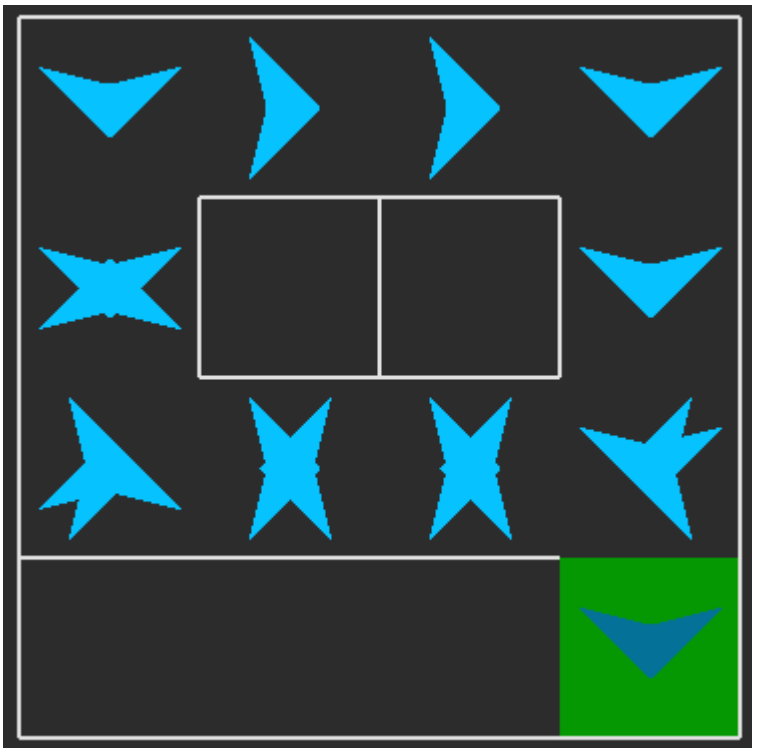
The direction selection is not truly random because I want to design a deterministic algorithm. This way, every time it encounters the same maze, it will follow the same path and take the same number of steps, which will be helpful for comparing the performance of different algorithms in later sections.

By remembering the visited locations, this algorithm can solve any connected maze without entering an infinite loop. It can explore all reachable areas. While it solves the looping problem, it requires O(n) space, and its efficiency is not as good as the Trémaux algorithm, which also uses O(n) space (mentioned in later sections).

The example of the Modified Hand-on-Wall algorithm here is only to illustrate "random direction selection" and "recording visited locations" and has no practical application value.


Start
Initialize start point and direction
Check turn counter
Even
Odd
Use left-hand rule
Use right-hand rule
Check for unvisited adjacent cells
Exists
Not exists
Move to the new cell
Backtrack and increment turn counter
Reached the end or exceeded steps?
No
Yes
End

Using the modified Hand-on-Wall algorithm to solve the previous island maze:



Same as the previous example, but after encountering a visited path, it will backtrack and try unvisited paths, successfully solving the maze

The Modified Hand-on-Wall algorithm can successfully solve mazes with isolated islands and reach the endpoint.

Pledge Algorithm

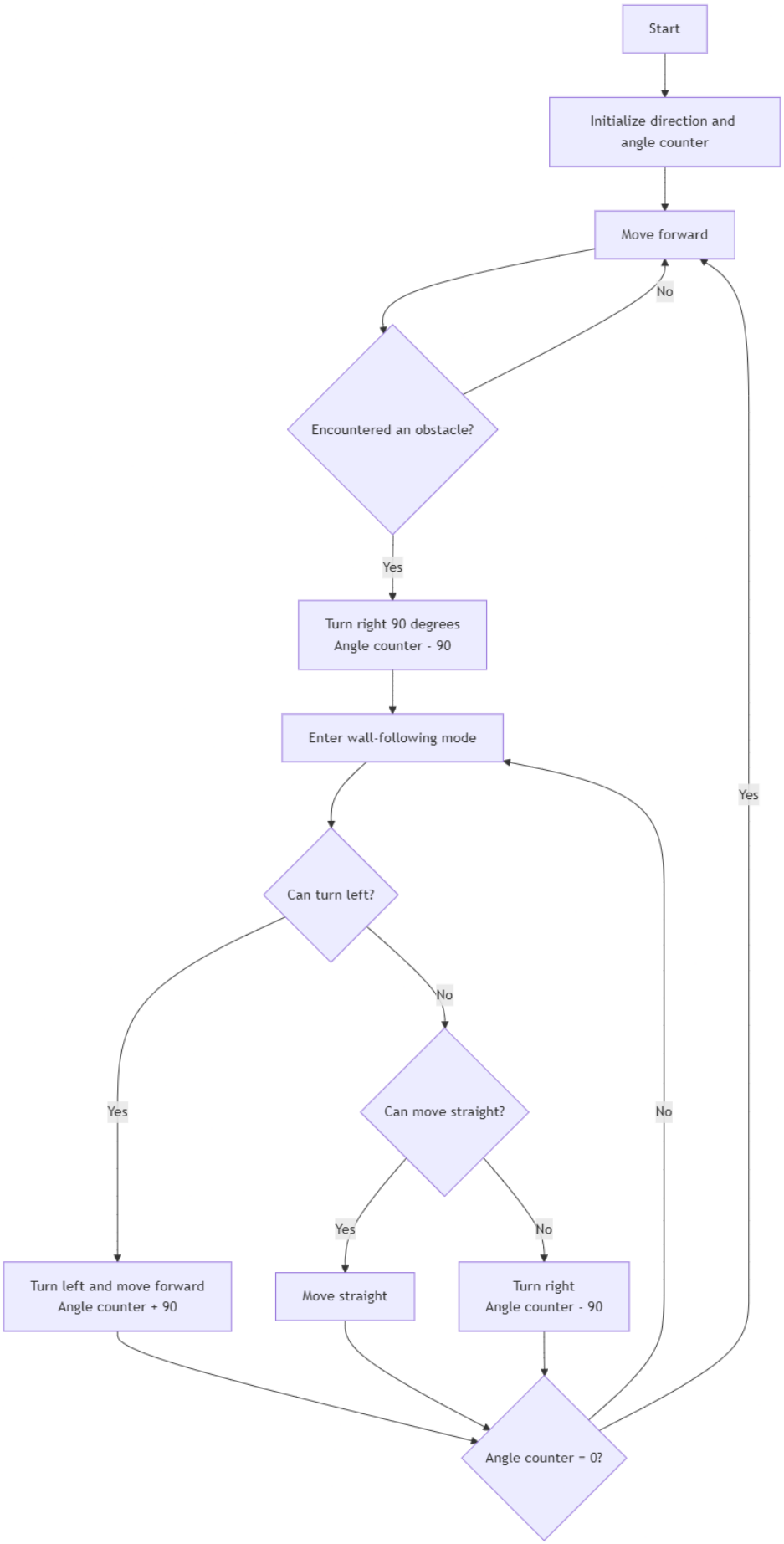
Before introducing the Pledge algorithm, imagine an algorithm where we let the robot always try to move in the same direction until it encounters an obstacle. Stick to the wall and walk around the obstacle until it can move in the predetermined direction again. How effective would such an algorithm be?

This is the core idea of the Pledge algorithm, but the algorithm described above cannot be applied to all mazes. But with a slight change to the part that counts turns, it becomes the Pledge algorithm, which can be applied to any maze. More detailed discussion can be found in ([1], p. 177).

The Pledge algorithm was developed by John Pledge of Exeter, England, at the age of 12 ([1], p. 177). It addresses the limitations of the standard wall-following algorithm, which can get stuck in loops in complex mazes. This algorithm guarantees not falling into loops in mazes with loops, and it does not need to record the maze structure and the path it has traveled. Reference ([1] chap 4.4) provides a detailed proof of the effectiveness of the algorithm, demonstrating its ability to avoid infinite loops like the wall-following algorithm.

Reference [1] describes how it works:

> *1. Select an arbitrary initial direction, call it "north," and face that*
> *way.*
> *2. Walk straight "northward" until you hit an obstacle.*
> *3. Turn left until that obstacle is on your right.*
> *4. Follow the obstacle around, keeping it on your right, until the total*
> *turning (including the initial turn in step 3) is equal to zero.*
> *5. Go back to step 2.*
>
> *([1], pp. 177-178)*


Start
Initialize direction and angle counter
Move forward
No
Encountered an obstacle?
Yes
Turn right 90 degrees
Angle counter - 90
Enter wall-following mode
Can turn left?
No
Yes
Can move straight?
No
Yes
No
Turn left and move forward
Angle counter + 90
Move straight
Turn right
Angle counter - 90
Angle counter = 0?
Yes

The advantage of the Pledge algorithm is that, like the Hand-on-Wall algorithm, it does not need to record maze information and the path it has traveled. It maintains O(1) space complexity while ensuring that it can solve mazes with loops.

The Pledge algorithm also has some disadvantages. Its implementation is more complex than the Hand-on-Wall algorithm, requiring additional maintenance of a turn counter and implementation of turn judgment. In real-world scenarios, introducing turn angle judgment requires additional sensors.

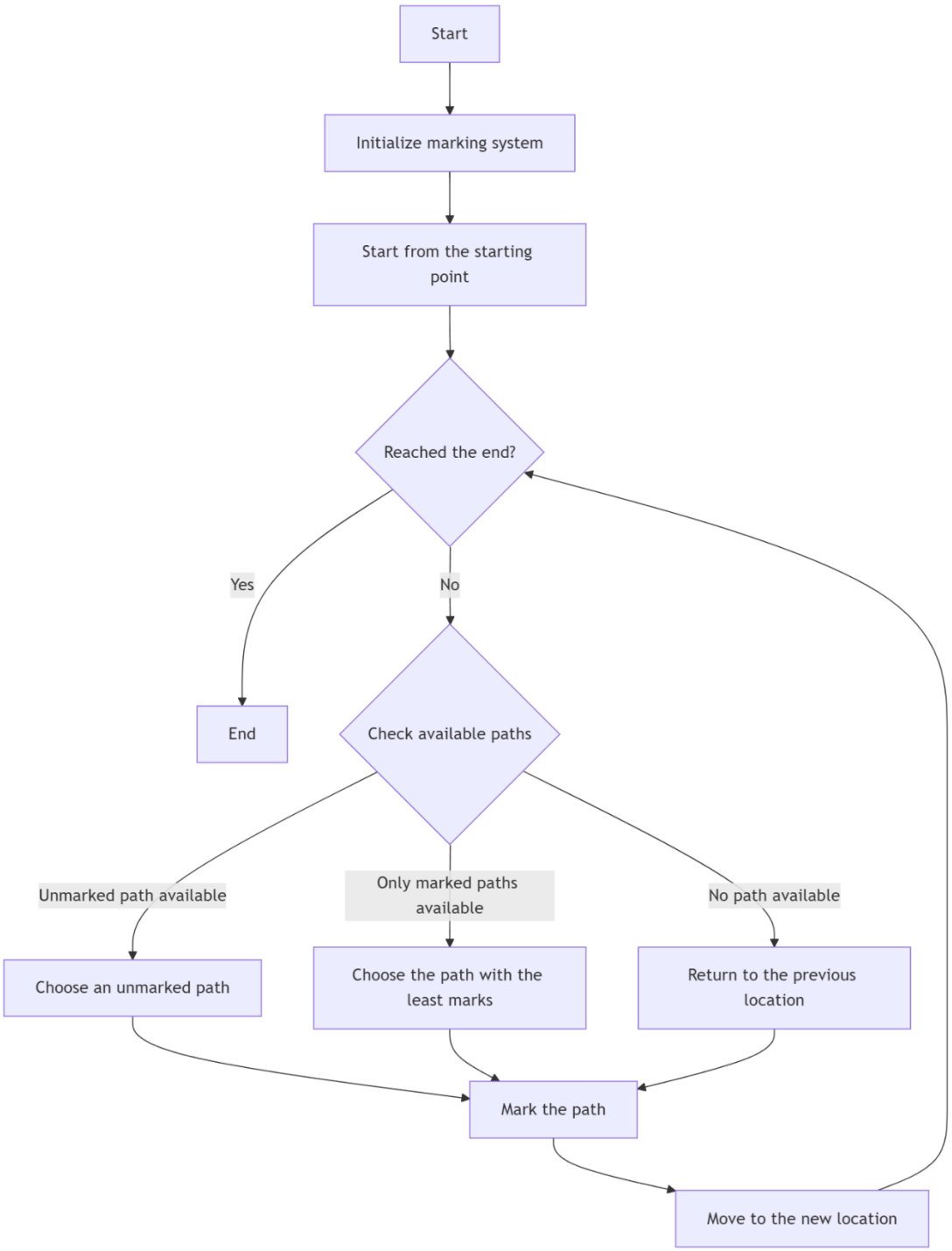
Since sensors always have errors, many papers have proposed various improvement schemes for the Pledge algorithm to make it more reliably solve mazes, such as [8].

Trémaux Algorithm

The Trémaux algorithm, invented by French telegraph engineer Charles Pierre Trémaux in the 19th century ([3], p. 47), marks paths in the maze to prevent repeated traversal, ensuring that the maze exit is found or all reachable paths are traversed within a finite number of steps. Incidentally, this algorithm is a very old form of depth-first search. The Trémaux algorithm is not only suitable for two-dimensional mazes but can even be extended to three-dimensional or even higher-dimensional mazes, making it very versatile.

The core principle of the Trémaux algorithm is to distinguish different paths and use these markings to guide the robot's movement ([3], pp. 47-49). Here's a brief overview of how the algorithm works:

1. Start from the starting point and mark the path you took away from the starting point.
2. Follow any path until you reach a dead end or a fork in the road.
If you reach a dead end: Go back the same way and treat the mark on this path as deleted (because it was walked twice).
If you reach a fork in the road: Choose a path at random and add marks where you enter and leave the fork.
3. When you reach a fork in the road that has already been explored, apply different rules depending on how you arrived:
Arrived via a new path: Back off and mark two lines where you enter and leave the fork.
Arrived via a path that has been walked: Give priority to a path that has not been walked.
If there is no path that has not been walked, choose a path that has only been walked once.
4. Return to step 2 until you reach the end.



Proof of the effectiveness and termination of the Trémaux algorithm can be found in reference [3], pp. 49-51.

The disadvantage of the Trémaux algorithm is that it needs to mark paths in the maze, which requires additional storage space and may cause some difficulties in

practical applications. The implementation of the Trémaux algorithm is relatively complex, involving path marking maintenance and path backtracking.

Similarly, the Trémaux algorithm is suitable for solving mazes with loops. In fact, the scope of application of this algorithm is far more than just mazes. It can be used to solve a wide variety of problems.

The Trémaux algorithm has some variants, such as the well-known Tarry algorithm. It uses three markers, unlike Trémaux's two markers [9].

Algorithm Comparison and Evaluation

The previous sections introduced three classic maze-solving algorithms: wall-following, Pledge, and Trémaux. This section will compare and evaluate these algorithms from various aspects to explore their characteristics and suitability for different scenarios.

Evaluation criteria include time complexity, space complexity, implementation complexity, and robustness. Both time complexity and space complexity are expressed using Big O notation ([2], chap. 3.2). Implementation complexity refers to the difficulty of implementing the algorithm code, measured by code logic complexity. Robustness represents the stability and reliability of the algorithm in handling various maze types, especially mazes with "islands" or complex loops. For example, it considers whether the algorithm is prone to getting stuck in loops and whether it can successfully find the exit. The Hand-on-Wall algorithm may fall into an infinite loop in complex mazes and fail to find the exit, so its robustness is low. Although the Pledge algorithm can solve mazes with loops, as mentioned earlier, sensors always have errors in real-world scenarios, so I rate its robustness as medium.

Algorithm	Time Complexity	Space Complexity	Implementation Complexity	Robustness
Hand-on-wall	O(n)	O(1)	Low	Low
Pledge	O(n)	O(1)	Medium	Medium
Trémaux	O(n)	O(n)	High	High

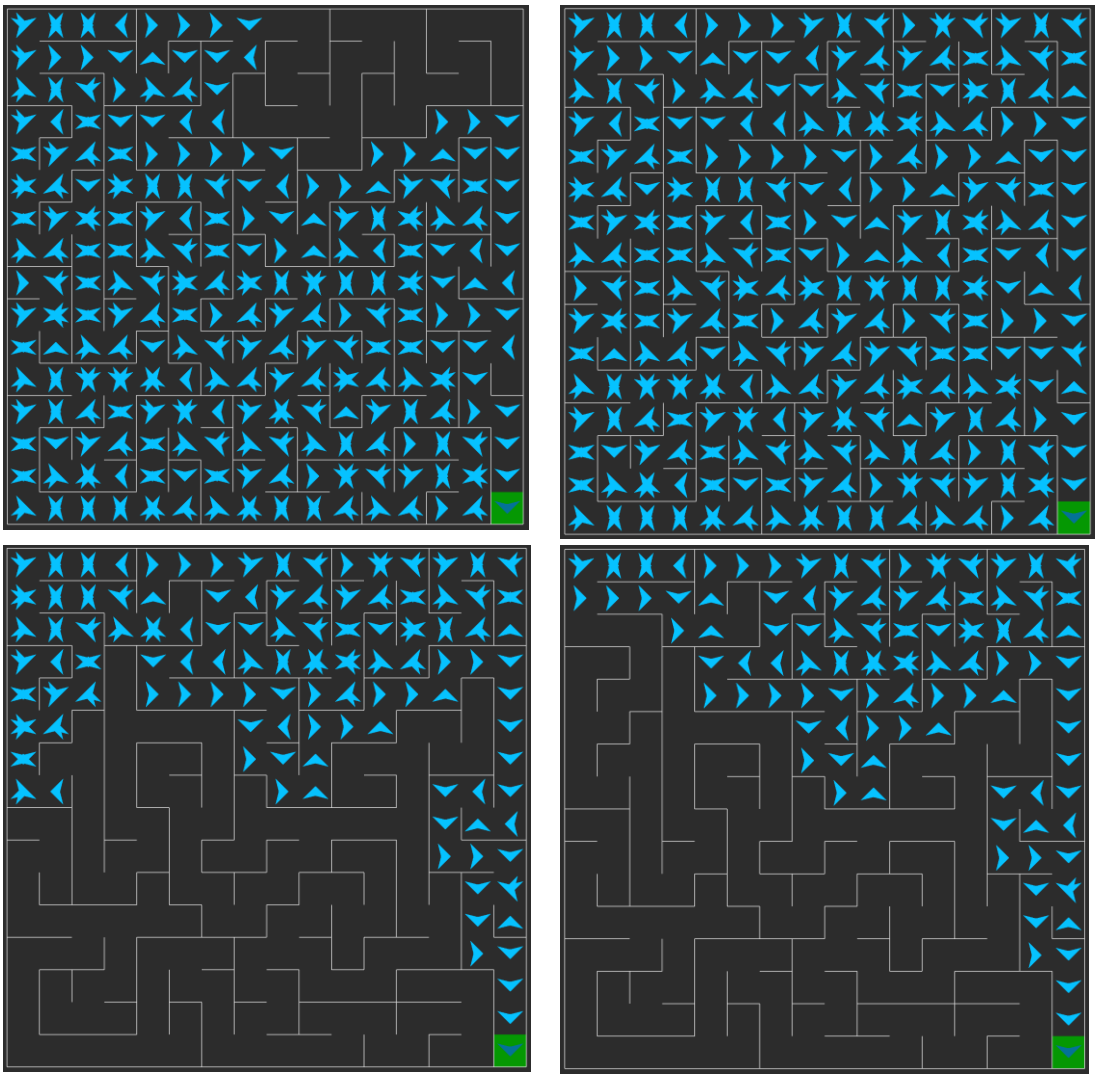
* n represents the number of nodes in the maze.

To more directly compare the maze-solving capabilities of the three algorithms, we conducted empirical experiments.

Using the pyamaze library, 1000 two-dimensional mazes of size 16x16 were randomly generated. Then, four different maze-solving algorithms were used to solve these mazes:

Hand-on-Wall algorithm
Modified Hand-on-Wall algorithm: mentioned earlier
Pledge algorithm
Trémaux algorithm
The mazes generated by pyamaze are loopless, ensuring that all parts of the maze are reachable. Therefore, all algorithms can solve these mazes with a 100% probability.

Next, we record the number of steps required for each algorithm to find the exit in each maze.



Processes of different maze-solving algorithms solving the same maze. Top left: Hand on wall, Top right: Modified hand on wall, Bottom left: Pledge, Bottom right: Trémaux

The following table lists the average steps, median steps, best ratio, worst ratio, and standard deviation of the four algorithms in 1000 mazes:

Algorithm	Mean Steps	Median Steps	Best Rate%	Worst Rate%	Std Dev
Hand-on-wall	342.6	341.0	34.1%	41.6%	135.7
Trémaux	256.5	257.0	63.2%	0.2%	78.3
Modified Hand-on-wall	373.6	391.0	0.6%	35.3%	74.1
Pledge	315.5	314.0	2.4%	23.2%	90.8

Let's analyze the data.

In terms of overall performance, the Trémaux algorithm performed best, with an average of 256.5 steps and a median of 257.0 steps, significantly lower than other algorithms. It achieved the best performance in 63.2% of the tests, and the standard deviation was also the lowest, indicating its stable performance and fast maze-solving speed.

The simplest Hand-on-Wall algorithm performed poorly. The average number of steps was 342.6, the median was 341.0, the best ratio was 34.1%, and the standard deviation was high, indicating unstable performance.

The Pledge algorithm performed better than the Hand-on-Wall algorithm. The average number of steps was 315.5, the median was 314.0, which was between Trémaux and Hand-on-Wall. Its standard deviation was 90.8, indicating relatively stable performance.

The Modified Hand-on-Wall algorithm, as I said before, has no practical application value. Its average number of steps was 373.6, the median was 391.0, and the standard deviation was 74.1, achieving the best performance in only 0.6% of the tests. Although the modified wall-following algorithm avoids infinite

loops, its nearly random direction selection leads to slower solving time. It was designed only to demonstrate the modification direction of the Hand-on-Wall algorithm. If it is necessary to guarantee the solution of loop mazes, the other two algorithms are more suitable.

Considering space complexity, although the Trémaux algorithm requires O(n) space to store visit history, its performance is significantly faster than other algorithms. The other two algorithms only require O(1) space, but their performance is not as strong as the Trémaux algorithm. This trade-off between space complexity and performance should be considered based on specific application needs.

In practical applications, each algorithm has its advantages. When computing resources are abundant and there are no obstacles to map marking, the speed and robustness of the Trémaux algorithm make it the preferred choice. When storage space is limited, the Pledge algorithm can also provide efficient performance without requiring additional storage. In simple applications, the Hand-on-Wall algorithm may be sufficient.

3. CONCLUSIONS

This paper conducted an in-depth analysis and comparison of classic maze-solving algorithms, focusing on the basic principles, advantages, disadvantages, and applicable scopes of the "Hand-on-Wall," "Pledge," and "Trémaux" algorithms. We performed theoretical and experimental analyses to compare the performance of these algorithms.

The Hand-on-Wall algorithm, due to its simplicity and low implementation cost, can be considered as the first choice in robot navigation. Its robustness in complex mazes is poor and may lead to infinite loops. If it is found that this algorithm cannot meet practical needs, other algorithms need to be considered.

The Pledge algorithm can effectively avoid falling into loops when dealing with mazes with loops while maintaining O(1) space complexity, making it a worthwhile option when memory resources are limited. In practical applications, it may be necessary to pay attention to the handling of sensor errors.

The Trémaux algorithm performs best and has the widest range of applications. It requires additional storage space to record the path. If conditions permit, its advantages in speed and robustness make it the preferred choice for solving maze problems.

In practical applications, there are more algorithms to choose from, and trade-offs need to be made based on the needs of various application scenarios.

In addition, there are many aspects of maze-solving algorithm research that are not discussed in this paper, such as:

Dynamic Maze Solving: In practical applications, the environment is not necessarily static. For example, a sweeping robot may encounter various obstacles, causing the maze to change, which requires the use of algorithms suitable for dynamic mazes.
Multi-Robot Collaboration: Using multiple robots to solve a maze simultaneously will effectively improve the efficiency of maze-solving. However, this also requires the use of new algorithms or improvements to

existing algorithms to design efficient communication and coordination mechanisms.
Hybrid Use of Multiple Algorithms: For example, combining the algorithms mentioned in the paper with graph-based search algorithms such as the A* algorithm. Use an exploration algorithm to map the maze structure and then use a graph search algorithm to find the shortest path. This can combine the advantages of multiple algorithms.
Application in Other Fields: Such as networks, logistics, chip design, etc.

Through continuous exploration and research, maze-solving algorithms can play a role in many practical applications, providing more ideas and methods for solving complex problems.

4. REFERENCES

1. Abelson, H., & diSessa, A. A. (1981). Turtle Geometry: The Computer as a Medium for Exploring Mathematics. MIT press.
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms. (Fourth Edition) MIT press.
3. Lucas, É. (1892). Récréations Mathématiques Volume I. Paris : Gauthier-Villars et fils
4. Even, S. (1979). Graph algorithms. Computer Science Press.
5. Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. (1968). Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics , vol. 4, no. 2
6. Russell, Stuart J.; Norvig, Peter (2018). Artificial intelligence a modern approach (4th ed.). Boston: Pearson.
7. D. C. Lee (1996). The Map-Building and Exploration Strategies of a Simple Sonar-Equipped Mobile Robot. Cambridge University Press
8. Tom Kamphans & Elmar Langetepe (2003). The Pledge Algorithm Reconsidered under Errors in Sensors and Motion. University of Bonn
9. Tarry, Gaston. (1895). “Le problème des Labyrinthes.” Nouvelles Annales de Mathématiques 3

5. APPENDIX

Hand-on-Wall

```
def move(cell, direction):
    if direction == 'N':
        return cell[0] - 1, cell[1]
    elif direction == 'S':
        return cell[0] + 1, cell[1]
    elif direction == 'E':
        return cell[0], cell[1] + 1
    elif direction == 'W':
        return cell[0], cell[1] - 1


class HandOnWallSolver:
    def __init__(self, maze):
        self.maze = maze
        self.direction = 'N'

    def solve(self, start, end):
        current = start
        path = [current]

        while current != end:
            right_direction = self.right_direction()
            if self.maze.maze_map[current][right_direction] == 1:
                self.direction = right_direction
                current = move(current, self.direction)
            elif self.maze.maze_map[current][self.direction] == 1:
                current = move(current, self.direction)
            else:
                self.direction = self.left_direction()

            path.append(current)

        return path

    def left_direction(self):
        return {'N': 'W', 'E': 'N', 'S': 'E', 'W':
'S'}[self.direction]

    def right_direction(self):
        return {'N': 'E', 'E': 'S', 'S': 'W', 'W':
'N'}[self.direction]
```

Modified Hand-on-Wall

```
class ModifiedHandOnWallSolver:
    def __init__(self, maze):
        self.maze = maze
        self.path = {}
        self.direction = 'E'
        self.turn_count = 0
        self.visited = set()

    def solve(self, start, end):
        current = start
        self.path[current] = current
        self.visited.add(current)
        path = [current]

        while current != end:
            if self.turn_count % 2 == 0:
                next_cell = self.follow_right_hand_rule(current)
            else:
                next_cell = self.follow_left_hand_rule(current)

            if next_cell:
                self.path[next_cell] = current
                current = next_cell
                self.visited.add(current)
                path.append(current)
            else:
                prev = self.path[current]
                del self.path[current]
                current = prev
                self.turn_count += 1
                path.append(current)

        return path
```

```
    def follow_left_hand_rule(self, cell):
        directions = {
            'N': ['W', 'N', 'E', 'S'],
            'E': ['N', 'E', 'S', 'W'],
            'S': ['E', 'S', 'W', 'N'],
            'W': ['S', 'W', 'N', 'E']
        }

        for check_direction in directions[self.direction]:
            if self.maze.maze_map[cell][check_direction] == 1:
                next_cell = self.move_forward(cell,
check_direction)
                if next_cell not in self.visited:
                    self.direction = check_direction
                    return next_cell
        return None

    def follow_right_hand_rule(self, cell):
        directions = {
            'N': ['E', 'N', 'W', 'S'],
            'E': ['S', 'E', 'N', 'W'],
            'S': ['W', 'S', 'E', 'N'],
            'W': ['N', 'W', 'S', 'E']
        }

        for check_direction in directions[self.direction]:
            if self.maze.maze_map[cell][check_direction] == 1:
                next_cell = self.move_forward(cell,
check_direction)
                if next_cell not in self.visited:
                    self.direction = check_direction
                    return next_cell
        return None

    @staticmethod
    def move_forward(cell, direction):
        return {
            'N': (cell[0]-1, cell[1]),
            'E': (cell[0], cell[1]+1),
            'S': (cell[0]+1, cell[1]),
            'W': (cell[0], cell[1]-1)
        }[direction]
```

Pledge

```
class PledgeSolver:
    def __init__(self, maze):
        self.maze = maze
        self.path = []
        self.direction = 'E'
        self.angle_sum = 0

    def solve(self, start, end):
        current = start

        while current != end:
            while self.can_move_forward(current):
                current = self.move_forward(current)
                self.path.append(current)
                if len(self.path) > 10000:
                    return self.path

            self.turn_right()
            self.angle_sum -= 90

            while self.angle_sum != 0 and current != end:
                if self.can_turn_left(current):
                    self.turn_left()
                    self.angle_sum += 90
                    current = self.move_forward(current)
                    self.path.append(current)
                elif self.can_move_forward(current):
                    current = self.move_forward(current)
                    self.path.append(current)
                else:
                    self.turn_right()
                    self.angle_sum -= 90

        return self.path

    def can_move_forward(self, cell):
        next_cell = self.move_forward(cell)
        return self.is_valid_move(cell, next_cell)
```

```
    def can_turn_left(self, cell):
        left_dir = self.get_left_direction()
        next_cell = self.move_forward(cell, left_dir)
        return self.is_valid_move(cell, next_cell)

    def is_valid_move(self, current, next_cell):
        if not (1 <= next_cell[0] <= self.maze.rows and 1 <=
next_cell[1] <= self.maze.cols):
            return False
        direction = self.get_direction(current, next_cell)
        return self.maze.maze_map[current][direction] == 1

    def move_forward(self, cell, direction=None):
        if direction is None:
            direction = self.direction
        return {
            'N': (cell[0]-1, cell[1]),
            'E': (cell[0], cell[1]+1),
            'S': (cell[0]+1, cell[1]),
            'W': (cell[0], cell[1]-1)
        }[direction]

    def get_direction(self, current, next_cell):
        if next_cell[0] < current[0]: return 'N'
        if next_cell[0] > current[0]: return 'S'
        if next_cell[1] < current[1]: return 'W'
        return 'E'

    def get_left_direction(self):
        return {'N': 'W', 'E': 'N', 'S': 'E', 'W':
'S'}[self.direction]

    def turn_right(self):
        self.direction = {'N': 'E', 'E': 'S', 'S': 'W', 'W':
'N'}[self.direction]

    def turn_left(self):
        self.direction = {'N': 'W', 'E': 'N', 'S': 'E', 'W':
'S'}[self.direction]
```

Tremaux

```
from collections import defaultdict

class TremauxSolver:
    def __init__(self, maze):
        self.maze = maze
        self.marks = defaultdict(int)
        self.exploration_path = []
        self.solution_path = {}

    def solve(self, start, end):
        current = start
        previous = None
        self.solution_path[current] = current
        path = [current]

        while current != end:
            next_cell = self.choose_next_cell(current, previous)

            if next_cell:
                self.mark_path(current, next_cell)
                self.solution_path[next_cell] = current
                self.exploration_path.append(next_cell)
                path.append(next_cell)
                previous = current
                current = next_cell
            else:
                next_cell = self.solution_path[current]
                self.mark_path(current, next_cell)
                self.exploration_path.append(next_cell)
                path.append(next_cell)
                previous = current
                current = next_cell

        return path
```

```
    def choose_next_cell(self, cell, previous):
        options = []
        for direction in 'NESW':
            if self.maze.maze_map[cell][direction] == 1:
                next_cell = self.move(cell, direction)
                if next_cell != previous:
                    marks = self.get_marks(cell, next_cell)
                    if marks < 2:
                        options.append((next_cell, marks))

        if not options:
            return None

        options.sort(key=lambda x: x[1])

        if previous and self.get_marks(cell, previous) == 1:
            unmarked = [opt for opt in options if opt[1] == 0]
            if unmarked:
                return unmarked[0][0]

        return options[0][0]

    def move(self, cell, direction):
        return {
            'N': (cell[0]-1, cell[1]),
            'E': (cell[0], cell[1]+1),
            'S': (cell[0]+1, cell[1]),
            'W': (cell[0], cell[1]-1)
        }[direction]

    def mark_path(self, cell1, cell2):
        key = tuple(sorted([cell1, cell2]))
        if self.marks[key] < 2:
            self.marks[key] += 1

    def get_marks(self, cell1, cell2):
        key = tuple(sorted([cell1, cell2]))
        return self.marks[key]
```