Maze Generation and Resolution Algorithms

Final Project Report

Authors:

Antonios Elie
Armagan Omer Ali
Chikhi Lounas
Jafari Zadeh Mehdi
Marteau Flavien
Tayache Rayane

December 17, 2024

Contents

1 Data Structure

1.1 Introduction

In this report, we present the data structure selected for representing a maze in our project, along with its formal specification. The objective was to design a lightweight and efficient structure that allows easy manipulation and rapid access to cell information, which is crucial for maze generation and pathfinding algorithms.

1.2 Maze Representation

We represent the maze as a two-dimensional matrix of integers. Each integer in the matrix corresponds to a cell in the maze and encodes multiple boolean properties using bitwise operations. This approach minimizes memory usage but stay simply usable and enables quick querying and updating of cell properties.

1.2.1 Bitwise Encoding of Cell Properties

Each cell uses bits to represent the presence of walls and special markers. Each bit was the response to a yes or no question. The significance of each bit (from least significant to most significant) is as follows:

- **Bit 0** (1): is a Wall on the **North** side.
- **Bit 1** (2): is a Wall on the **East** side.
- **Bit 2** (4): is a Wall on the **South** side.
- **Bit 3** (8): is a Wall on the **West** side.
- **Bit 4** (16): is the Cell an **Entrance**.
- **Bit 5** (32): is the Cell an **Exit**.
- **Bit 6** (64): Cell has been **Visited**.
- **Bit 7** (128): is the Cell a part of the **Path**.

1.3 Formal Specification

1.3.1 Data Types

- **Maze**: The maze data structure.
- **Height**, **Width**: Positive integers representing the maze dimensions.
- **X**, **Y**: Integers representing cell coordinates.
- **Orientation**: One of $\{\text{North}, \text{East}, \text{South}, \text{West}\}$.
- **Boolean**: A boolean value (`True` or `False`).

1.3.2 Function Specifications

1. init(h : Height, w : Width) $\rightarrow$ Maze

   Initializes a new maze with height h and width w.

2. open_wall(m : Maze, x : X, y : Y, o : Orientation) $\rightarrow$ Maze

   Removes the wall in cell (x, y) in the direction o and remove the corespondant wall in the next cell.

3. set_entrance(m : Maze, x : X, y : Y) $\rightarrow$ Maze

   Marks cell (x, y) as the entrance.

4. set_exit(m : Maze, x : X, y : Y) → Maze

   Marks cell (x, y) as the exit.

5. divide(m : Maze) → (Maze)

   Divides the maze m into two sub-mazes, open a wall between the two and fusion the tow with fusion function.

6. fusion(m_1 : Maze, m_2 : Maze, o : Orientation) → Maze

   Merges mazes m_1 and m_2 along orientation o.

7. destroy(m : Maze) → Void

   Frees memory associated with the maze m.

8. is_open(m : Maze, x : X, y : Y, o : Orientation) → Boolean

   Returns `True` if the wall in direction o at cell (x, y) is open.

9. is_entrance(m : Maze, x : X, y : Y) → Boolean

   Returns `True` if cell (x, y) is the entrance.

10. is_exit(m : Maze, x : X, y : Y) → Boolean

    Returns `True` if cell (x, y) is the exit.

11. size(m : Maze) → (Height, Width)

    Returns the dimensions of maze m.

12. tree gen(m : Maze) → (Tree, Matrix Nodes)

    generate the tree and the tree node matrix$m$.

1.3.3 Preconditions

- For each function coordinates x and y must be within maze bounds: $0 \leq x <$ Width, $0 \leq y <$ Height.
- For the three function *open_wall*, *open_wall*, *fusion*, *is_open*, the orientation o must be valid: $o \in \{\text{North}, \text{East}, \text{South}, \text{West}\}$.

1.4 Conclusion

The selected data structure efficiently represents the maze using a compact matrix of integers and bitwise encoding. This design permit direct access and modification of cell properties, which give us the possibility of making the best complexity algorithm for the maze generation and solving algorithms. The formal specification provides a clear interface and ensures the correct manipulation of the maze through well-defined operations.

2 Maze Generation and Resolution Algorithms

2.1 Introduction

This report provides a brief description of the algorithms used for maze generation and resolution in our project. We explain our choices and present the algorithms in both English and pseudocode for clarity.

2.2 Maze Generation Algorithm

2.2.1 Algorithm Description

We implemented a recursive division algorithm to generate the maze. We divide the maze in two half mazes (not necessarily in the middle, this is randomly chosen) and create a passage between the two (who is also randomly chosen). At the end of the recursive call, we go back and fusion the two small mazes into the bigger one. This way, we finish the function with a fully randomly create maze with all cells connected.

2.2.2 Pseudo-Code

```
function divide(maze):
    width=m.width
    height=m.height

    if width <= 1 or height <= 1:
        return maze

    int x=0
    int y=0
    if width>1:
        x=random(0,width-1)
    if height>1:
        y=random(0,height-1)

    if width=height:
        orient = random([VERTICAL,HORIZONTAL])
    else if width>height:
        orient = VERTICAL
    else:
        orient = HORIZONTAL

    open_wall(m,x,y,orient)
    if orient == VERTICAL:
        m1 = init_maze(height, x + 1)
        m2 = init_maze(height, width - (x + 1))
    else:
        m1 = init_maze(y + 1, width)
        m2 = init_maze(height - (y + 1), width)

    // feel m1:
    for i in range(0,height):
      for j in range(0,width):
        m1->matrix[i][j] = m->matrix[i][j]

    // feel m2:
    for i in range(0,height):
      for j int range(0,width):
        m2->matrix[i][j] = m->matrix[i + (y + 1) * (1 - orient)][j + (x + 1) * orient]

    //recursive call
    divide(m1)
    divide(m2)

    //fusion call
```

```
    fusion(m, m1, m2, orient)
    maze_free(m1)
    maze_free(m2)

    return maze
```

2.3 Maze Resolution Algorithm

2.3.1 Explanation of Choice

For the maze resolution, we based our logic on the fact that we have only one exit at the south-east corner of the maze. And we can start in the maze from any cells. In this way, we can make only one BFS from the exit to build a tree of all cells and a matrix who pointed to the tree node (each cell in the maze corresponds to a node on the tree). With this data structure, we only make one BFS after generating the maze. And the resolution is resumed to go up in the tree to find the exit. The consequence is the path to the exit is always the same, and we have a longer algorithm for the generation but the shorter possible for finding the path to the exit.

2.3.2 Algorithm Description

For the maze resolution, we start by a BFS who build a tree from the exit. Each cell corresponds to a node in the tree and a pointer to this node is put in a matrix, this corresponds to the function tree gen. When we will find the path to the exit from a cell, we take in the matrix the corresponding node in the tree and go up as far as we arrived to the exit. This corresponds to the function path to exit. The function is not leaf, to take the coordinate of a cell and return the reachable cells (and not already borrowed).

2.3.3 Pseudo-Code

```
function is_not_leaf(int x, int y, maze m):
    height=m.height
    width=m.width
    mat=m.matrix
    int res = 0
    if mat[y][x] & NORTH && y < height && (mat[y - 1][x] & VISITE) == 0
        res += NORTH
    if mat[y][x] & EAST && x < width && (mat[y][x + 1] & VISITE) == 0
        res += EAST
    if mat[y][x] & SOUTH && y >= 0 && (mat[y + 1][x] & VISITE) == 0
        res += SOUTH
    if mat[y][x] & WEST && x >= 0 && (mat[y][x - 1] & VISITE) == 0
        res += WEST
    mat[y][x] += VISITE
    return res


function tree_gen(maze m):
    height=m.height
    width=m.width
    tree t = tree_init(NULL, width - 1, height - 1)
    m->mat_expl[height - 1][width - 1] = t   //mat_expl is the matrix of the tree node
    queue q = {0}
    enqueue(q, t)
    while !is_empty_queue(q):
        tree father = dequeue(q)
        int x = father.x
```

```
        int y = father.y
        int n = is_not_leaf(x, y, m)
        if n & NORTH:
            m->mat_expl[y - 1][x] = tree_init(father, x, y - 1)
            enqueue(q, m->mat_expl[y - 1][x])

        if n & EAST:
            m->mat_expl[y][x + 1] = tree_init(father, x + 1, y)
            enqueue(q, m->mat_expl[y][x + 1])

        if n & SOUTH:
            m->mat_expl[y + 1][x] = tree_init(father, x, y + 1)
            enqueue(q, m->mat_expl[y + 1][x])
        if n & WEST:
            m->mat_expl[y][x - 1] = tree_init(father, x - 1, y);
            enqueue(q, m->mat_expl[y][x - 1]);

function path_to_exit(maze m, int x,int y):
    m.matrix[y][x]+=ENTRANCE
    tree_t t = m->mat_expl[y][x]
    while (t != NULL):
        m->matrix[t->y][t->x] += PATH;
        t = t->father;
```

2.4 Conclusion

In conclusion, we have an algorithm which recursively creates a maze with all the cells connected with only one exit at the south-east corner, like Minos ask we. the entry can be anywhere, while in the palace there are many trap doors. But we are careful, so we decide to generate a tree of all the maze. Like if we are trapped in the maze, we can directly find the path to the exit.

3 Proof of the Theoretical Complexity of the Maze Resolution Algorithm

3.1 Introduction

In this part, we provide a proof of the theoretical complexity of our maze resolution algorithm. We have implemented the Breadth-First Search (BFS) algorithm to find the shortest path from the entrance to the exit in a maze. We will demonstrate that the time complexity of this algorithm is linear with respect to the number of cells in the maze.

3.2 Maze as a Graph

A maze can be represented as an undirected graph G with N cells. All cells are like a Vertex of a graph with maximum degree of 4, because the maximum number of neighbors is 4 (North, East, South, West) we also know that in this graph all the cells are connected. the demonstration is simple, it is demonstrated that our graph is related:

- Initialize: one cell is a related graph
- Recursion: when we concatenate two part of the maze, we add an opening between the two. so we create a link between the two related graphs. The result is a connected graph.

- Conclusion: the propriety is initialized and recursive, so all the maze generate by this algorithm were related.

3.3 Tree generation (BFS) Algorithm complexity

if we start to the simplest element in our function:

- enqueue an element is in constant complexity,
- the cases for the 4 directions are also constant, so is not leaf too.

Consequently, all inside the loop was in constant complexity, the complexity of our algorithm is the number of loop iteration.

Our algorithm start at the exit, for each cell he has in the queue he adds all his not visited neighbor in the queue and stopped when the queue is empty. Or, all the maze cells are connected to each other, so our algorithm make only one loop iteration for each cell in the maze. Finally, our tree generation have a linear complexity in the number of cells O(N).

3.4 Path finding complexity

When we fall in the maze we can just search in the matrix in our position, the Vertex of the tree. This is in constant complexity. After that, we just need to go up in the tree to reach the exit. On the worst case the path from our cell to the exit pass by each cell, so the complexity is linear (O(N)). But when we take a closer look every time we go directly by the shortest path so it's the best that we can do.

3.4.1 Conclusion

In conclusion, we have a tree generation algorithm that prepares the tree during the generation phase. Its complexity is linear that less than the maze generation, so we can ignore it. And the complexity of the path finding was the shortest possible, just reach the exit by the shortest path, but this is at worst in O(N). Consequently, the path finding is the best that is possible to do.

4 Performance Analysis of the Maze Resolution Algorithm

4.1 Introduction

This report provides an explanation of the tests conducted to analyze the performance of our maze resolution algorithm, along with the results obtained. The primary objective was to evaluate the time and space complexities of the algorithm in practical scenarios and to verify its scalability and efficiency. We conducted experiments on mazes of varying sizes to observe how the algorithm behaves as the maze dimensions increase.

4.2 Testing Methodology

4.2.1 Test Setup

To systematically assess the performance, we developed an automated testing framework comprising:

1. A shell script to generate mazes of different sizes.
2. Execution of the maze resolution algorithm on each generated maze.
3. Measurement of execution time and memory usage for each test case.
4. Computing the average time of execution and memory usage of multiple same dimension maze.
5. Collection and analysis of the performance data.

4.2.2 Maze Generation

We generated mazes with dimensions ranging from $2^1 \times 2^1$ to $2^{12} \times 2^{12}$, increasing in increments of 2^{n+1}. Each maze was generated using our recursive division algorithm to ensure randomness and variability in the maze structure.

4.2.3 Performance Metrics

The key performance metrics measured were:

- **Execution Time:** Total time taken by the algorithm to solve the maze.
- **Memory Usage:** Peak memory consumption during the algorithm's execution.

We utilized the `time` command for execution time measurements and `valgrind` with the `massif` tool for memory profiling.

NB: These results were gotten on a M1 Macbook, results may differ on different machines, observations will remains applicable.

4.3 Results

4.3.1 Execution Time Analysis

Table 1 presents the execution times for different maze sizes. Figure 1 illustrates the relationship between maze size and execution time.

Maze Size	Execution Time (seconds)
2×2	0.00006
4×4	0.00006
8×8	0.00006
16×16	0.0008
32×32	0.002
64×64	0.006
128×128	0.01
256×256	0.06
512×512	0.27
1024×1024	1.600
2048×2048	4.610
4096×4096	18.950

Table 1: Execution Time for Different Maze Sizes

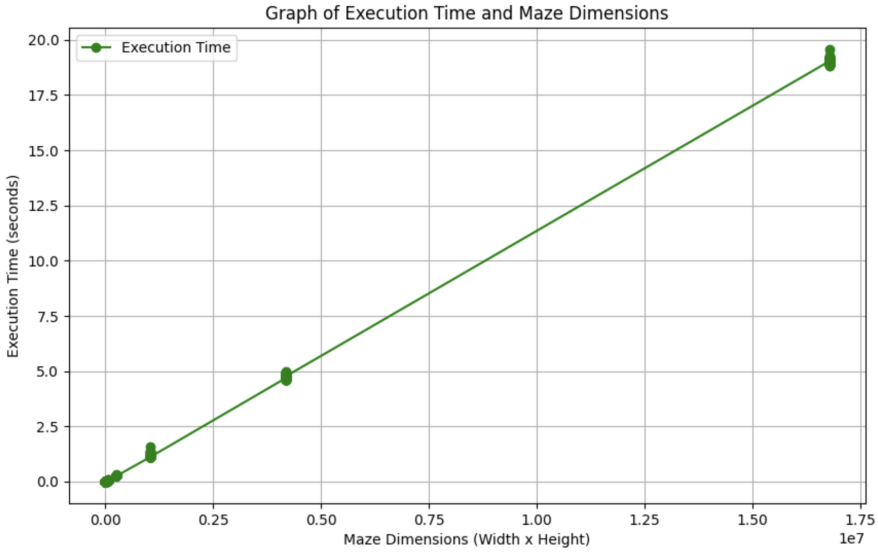

Figure 1: Execution Time vs. Maze Size

4.3.2 Memory Usage Analysis

Memory usage statistics are shown in Table 2 and plotted in Figure 2.

Maze Size	**Memory Usage (KB)**
2×2	976.00
4×4	976.00
8×8	976.00
16×16	992.00
32×32	1024.00
64×64	1232.00
128×128	2576.00
256×256	7728.00
512×512	27136.00
1024×1024	84880.00
2048×2048	243536.00
4096×4096	1081040.00

Table 2: Memory Usage for Different Maze Sizes

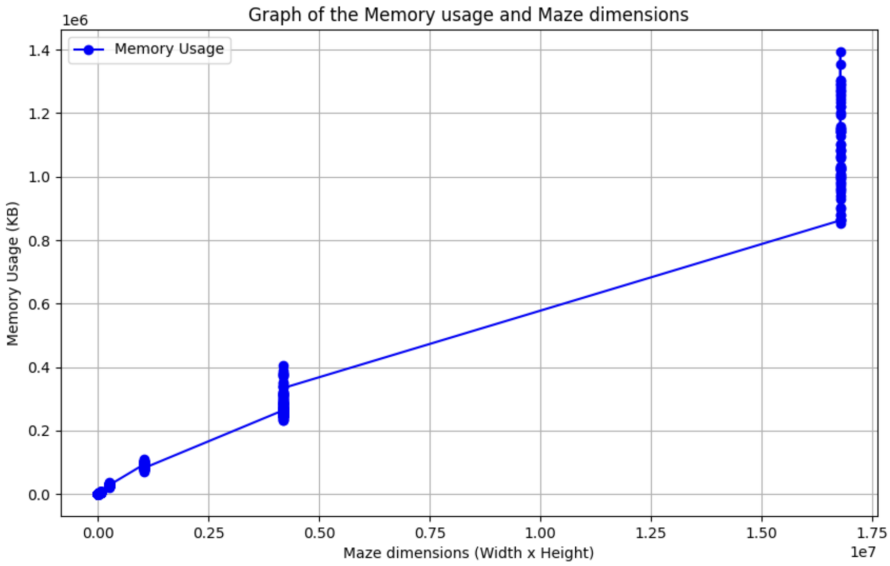

Figure 2: Memory Usage vs. Maze Size

4.4 Analysis of Results

4.4.1 Execution Time

The execution time increases approximately linearly with the maze size. This trend is consistent with the theoretical time complexity of the BFS algorithm when applied to a grid-based maze.

Quadratic Growth Explanation

In a maze of size $n \times n$:

- The number of cells (vertices) $V = n^2$.
- The BFS algorithm's time complexity is $O(V + E)$, but since the number of edges E is proportional to V in a grid, the overall complexity simplifies to $O(V)$.

However, because V itself is proportional to n^2, the time complexity in terms of n becomes $O(n^2)$. This explains the observed quadratic growth in execution time with respect to the maze dimension n.

Empirical Observations

The plotted execution times closely follow a linear curve. Small deviations can be attributed to system performance variability and the overhead of additional operations in the code.

4.4.2 Memory Usage

Memory usage also exhibits a linear growth pattern. This is expected because:

- The algorithm maintains data structures proportional to the number of cells, such as the maze matrix and visited status.
- Each cell requires a fixed amount of memory, so total memory usage scales with n^2.
- We observe that for a fixed maze size n, as n increases, the memory usage difference between multiple samples can be quite huge, due to the tree generation, some maze are easier to solve than other, implying a shorter tree.

4.4.3 Algorithm Efficiency

The BFS algorithm performs well for small to medium-sized mazes. However, for very large mazes, the quadratic time and space complexities pose challenges.

An implementation of the A* Algorithm was used as comparison, however the results were not always great due to the geographic constraint, the heuristic used was the manhattan distance from the entry point to the exit.

It was not always the shortest path, and is taking longer on average to find a path, in comparison to a simple BFS.

Potential Optimizations

To improve performance on larger mazes:

- **Heuristic Search Algorithms:** Implementing A* search with an admissible heuristic can reduce the search space by prioritizing promising paths.
- **Bidirectional Search:** Running simultaneous searches from the entrance and exit could potentially halve the search space.
- **Parallelization:** Utilizing multi-threading to explore multiple paths concurrently.
- **Memory Management:** Releasing memory of cells no longer needed or using iterative methods to reduce stack usage.

5 Conclusion about our choices

5.1 algorithm and data structure conclusion

5.1.1 Data structure

For the data structure we choose to use a matrix of integer, after making the project we think that's a good idea, the big problem of this technic was the coherence of the walls: Our data structure allows us to open a wall but only in one direction, that a problem, but we deal with that by creating a function open wall, that opens the two walls at the same time. And like that we only use this function to open a wall, the coherence of our maze is guarantee. However, we see an upgrade that we can make: for saved multiple boolean value, we use an integer but if we create a struct of boolean it can be a little bit better why we use exactly the good number of bite. And that can simply the usage of the data if someone was not comfortable with the notion of mask and bite operation.

5.1.2 Generation algorithm

For the generation algorithm there is a lot of think that we can improve. first and the most important, when we divide the maze for open a wall we really create two smaller maze, allocate memory and copy value from the big to the smaller. That's too heavy we should have to work for all step with the big matrix but add cursor too know on what we work. like it we don't need to allocate more memory and we don't need to copy the matrix. Like it we can have a best complexity with only the essential operation, that permit to work on bigger maze or have a quick algorithm.

5.1.3 Resolution algorithm

For the resolution algorithm, we choose to prepare all the work during the maze generation algorithm. Like that, we don't see how to improve the part of pathfinding. But for the tree generation part, we imagine that is possible to create the tree during the maze generation, by connecting node when we open a wall. But we don't see how to manage the distance to the exit to connect the Vertex only if it's the shortest path to the exit. It's why we choose an algorithm who is simple and quick. In addition, it's look impossible to create a

tree with V cells in a complexity lower than O(V). So if it's to redo we can take time to think about this idea, but if it's harder or impossible we can redo exactly the same.

An idea for improving all of our algorithm is to try to make parallelize algorithm, but it is difficult why we're just starting to learn how to make it during this semester. For this reason, that's not a solution in the start of the semester, but that can be one in the future.

5.2 learned lessons

- **Importance of Data Structure and Optimization.**
  We learned that the choice of data structure directly impacts the algorithm's performance. By selecting an optimal structure we were able to reduce computational time and improve efficiency.

- **Collaboration and Managing Differing Opinions.**
  At the start, reaching consensus was difficult because each team member had strong opinions and often very different approaches. We each had to learn to express and defend our viewpoints while also listening actively to others. This experience highlighted the importance of open-mindedness and compromise. We needed to assess each proposal against objective criteria (efficiency, simplicity, feasibility) and adopt the most suitable solution, even if it wasn't the one we initially preferred.

- **Simplicity First: Avoiding Over-Complexity.**
  An important lesson was realizing that the most complex ideas are not always the best. Sometimes, our initial ideas were too complicated and required unnecessary resources. This realization led us to simplify and streamline our approach, choosing solutions that were simpler and more direct, yet just as effective.

- **Importance of Testing for a Reliable Solution.**
  The unit testing phase reminded us that implementation alone is not enough. By testing our program with various types of mazes, we were able to identify and fix errors, improving our code to cover more edge cases. This underscored the importance of robust code and the need to validate solutions in an environment as close as possible to real conditions.

- **Learning Pathfinding Algorithms (BFS and A*).**
  We learned that the choice of algorithm depends on the problem's nature and resource constraints. BFS, although simple, can be costly for large structures, whereas A* uses a heuristic that can speed up the search but requires precise heuristic evaluation. Working with these methods deepened our understanding of pathfinding techniques and helped us choose the algorithm best suited to the project's needs.

- **Communication in English: A Challenge and an Asset.**
  In the beginning, communicating in English was challenging because it slowed our exchanges and limited our ability to express ourselves as fluently as we could in our native language. However, with practice, it became more natural and even improved our work, as we began to integrate common technical terms more confidently.

5.3 Questions and Future Considerations.

- Moving forward, we have a few key questions regarding the choices we made during the project. First, we wonder if our choices for maze generation, path searching, and testing were indeed optimal, or if there are ways to further refine them. Are the data structures and algorithms we selected the best options for performance and accuracy? And second, is there a way to make our implementation even more optimized?

6 Appendix

The repository of C code, 2 branches, for the tests perf, bash scripts differs, on ubuntu and macos machines. The Algorithm is working on every machines :

https://git.unistra.fr/fmarteau/marteauflavien_problem_solving