UFR de mathématique et d'informatique

Université de Strasbourg

Licence d'informatique

Problem Solving With Algorithms

Casso Tobe
Hassan El Sahily
Jeronimo Herdoïza
Gwénaël Mayer
Ugo Berton

Final Report: Maze Resolution Algorithm

Wednesday 13th November, 2024

Course given by

Dr. Imogen Morris

imorris@unistra.fr

Contents

1 Data Structures and Formal Specification

We decided to use multiple Data Structures to try a different approach, other than the Breadth-First Search (BFS), to improve the complexity of the resolution. We initially looked for a logarithmic complexity.

1.1 Maze Data Structure

Fields:

- X: Integer (Width of the maze)
- Y: Integer (Height of the maze)
- nei: ListeC** (Neighbor matrix of cells)
- sections: Tree (Tree structure representing maze sections)
- exit: cell (Coordinates of the exit)
- entries: ListeC (List of maze entry points)

Operations:

- generate_maze_entries(Maze m, int nb_entries) → void
- generateMazeStruct(Tree* tree, ListeC** nei, int xmin, int ymin, int xmax, int ymax) → void
- create_maze(int x, int y, int nb_entries) → Maze
- print_maze(Maze m) → void
- free_maze(Maze m) → void

Axioms: The Maze structure has a single constructor and operations that do not return values, so no specific axioms are defined here.

1.2 Tree Data Structure

Fields:

- a: cell (First cell of the tree node)
- b: cell (Second cell of the tree node)
- rchild: Node* (Right child node)
- lchild: Node* (Left child node)

Operations:

- track_malloc(size_t size) $\rightarrow$ void
- track_recursive_call() $\rightarrow$ void
- track_recursive_return() $\rightarrow$ void
- root(cell c1, Tree lchild, cell c2, Tree rchild) $\rightarrow$ Tree
- vertical_door(Tree t) $\rightarrow$ bool
- find_youngest_comon_ancestor(Tree sections, cell start, cell end) $\rightarrow$ Tree
- find_path(Tree sections, ListeC** nei, cell start, cell end) $\rightarrow$ ListeC
- cell_in_visited(cell c, ListeC visited) $\rightarrow$ bool
- path_no_cycle(ListeC path) $\rightarrow$ bool
- free_tree(Tree tree) $\rightarrow$ void
- export_memory_counters_to_json(const char *filename, int width, int height) $\rightarrow$ void

Axioms: Let $c, c1, c2$: Cell, $Lvisited, L1, L2$: ListeC and t : Tree.

- `cell_in_visited`$(c, Lvisited) = c \in Lvisited$
- `path_no_cycle`$(L1) =$ if (all $c \in L1$ are unique) then true else false
- `vertical_door(t)` $=$ if (vertical_door $\in t$) then true else false

1.3 ListeC Data Structure

Fields:

- a: cell (Cell stored in the node)
- next: ListeC* (Pointer to the next node in the list)
- last: ListeC* (Pointer to the previous node in the list)

Operations:

- append_list_cell(ListeC l, cell c) $\rightarrow$ ListeC
- listeC_has_comon_element(ListeC l1, ListeC l2) $\rightarrow$ bool
- merge_listeC(ListeC l1, ListeC l2) $\rightarrow$ ListeC

- empty_nei_matrix(int x, int y) $\rightarrow$ ListeC**
- cell_in_list(cell c, ListeC l) $\rightarrow$ bool
- connected(ListeC** nei, cell c1, cell c2) $\rightarrow$ bool
- print_path(ListeC path) $\rightarrow$ void
- free_list(ListeC list) $\rightarrow$ void
- free_matrixe(ListeC** mat, int x, int y) $\rightarrow$ void

Axioms: Let c, d : Cell, and $L1, L2$: ListeC.

- `append_list_cell(L1, c)` $= L1 \cup \{c\}$
- `has_comon_element(append_list_cell(L1, c), append_list_cell(L2, d))` = if $(c == d)$ then true else has_comon_element$(L1, L2)$
- `has_comon_element(L1, NULL)` = `has_comon_element(NULL, L1)` = false
- `merge_listeC(L1, L2)` = if $(L2 == NULL)$ then $L1$ else if $(L1 == NULL)$ then $L2$ else $L1 \cup L2$
- `cell_in_list(c, append_list_cell(L1, d))` = if $(c == d)$ then true else cell_in_list$(c, L1)$
- `cell_in_list(c, NULL)` = false

Preconditions: Let c : Cell, and $L1, L2$: ListeC.

- `append_list_cell(L1, c)`: `c` must not already be in `L1`.

1.4 Cell Data Structure

Fields:

- x: Integer (The x-coordinate of the cell)
- y: Integer (The y-coordinate of the cell)

Operations:

- new_cell(int x, int y) $\rightarrow$ cell
- cellcp(cell c) $\rightarrow$ cell
- add_cst_to_cell(cell c, int x, int y) $\rightarrow$ cell
- cell_equal(cell c1, cell c2) $\rightarrow$ bool

- cell_nei(cell c1, cell c2) $\rightarrow$ bool
- onleft(cell c1, cell c2) $\rightarrow$ bool
- upper(cell c1, cell c2) $\rightarrow$ bool

Axioms: Let $x, y, z, a : \mathbb{N}$.

- `cellcp(new_cell(x, y))` = `new_cell(x, y)`
- `add_cst_to_cell(new_cell(x, y), z, a)` = `new_cell(x+z, y+a)`
- `cell_equal(new_cell(x, y), new_cell(z, a))` = if $(x == z \wedge y == a)$ then true else false
- `onleft(new_cell(x, y), new_cell(z, a))` = if $(x \leq z)$ then true else false
- `upper(new_cell(x, y), new_cell(z, a))` = if $(y \geq a)$ then true else false
- `cell_nei(new_cell(x, y), new_cell(z, a))` = if $(x == z)$ then if $(y == a - 1 \vee y == a + 1)$ then true else if $(y == a)$ then if $(x == z - 1 \vee x == z + 1)$ then true else false

Preconditions: Let $c1, c2$: Cell, and $x, y, z, a : \mathbb{N}$.

- `cell_nei(c1, c2)`: No precondition. This tests if the cells are the same, thus not neighbors.

1.5 Display Of The Maze

In our display process, each cell of the maze is represented by a 2x2 structure as follows:

Our cell and its status	Nothing or #
Nothing or #	#

In the diagonal, it's always a # because we can only move vertically or horizontally in the maze.

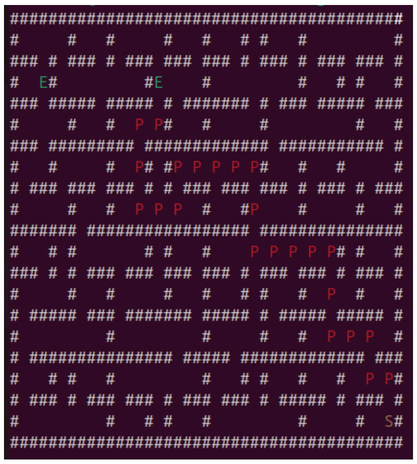
For cells on the right or at the bottom of the one we are interested in, we need to check if our current cell is connected to the next one in the line/column. If so, we print a space; otherwise, we print a #.

For the top-left cell, we print its status: entrance, exit, part of the resolution, or nothing (represented by a space).

1.6 Display Pseudo-code

```
for width j:
    print ##
endfor
print #
print \n
for height i:
    print #
    for width j:
        print_status_cell(i, j)
        if (connected_with_j+1):
            print white space
        else:
            print #
        endif
    endfor
    print \n
    print #
    for width j:
        if (connected_with_i+1):
            print white space
        else:
            print #
        endif
    print #
    endfor
    print \n
endfor
print \n
```

1.7 A picture of the maze



```
#########################################
#     #   #     #   #   # #   #         #
### # ### # ### ### ### # ### # ### ### #
#  E#         #E    #         #   # #   #
### ##### ##### # ####### # ### ##### ###
#     #   #  P P#   #     #         #   #
### ######### ############# ########### #
#   #     #  P# #P P P P P#   #   #     #
# ### ### ### # # ### ### ### # ### # ###
#     #   #  P P P  #   #P    #     #   #
####### ################# ###############
#   # #       # #   #    P P P P P# #   #
### # # ### ### ### ### # ### ### # ### #
#     #   #     #   #   # #   #  P  #   #
# ##### ### ####### ##### # ##### ##### #
#         #         #     #   #  P P P  #
# ############### ##### ############# ###
#   # #   #         #   # #   #   #  P P#
# ### # ### ### # ### ### # ##### # ### #
#         #   # #   #         #     #  S#
#########################################
```

Figure 1: Screenshot of the maze display in the terminal

2 Generation and resolution algorithms

2.1 A brief description of the Maze Generation Algorithm

The maze generation algorithm involves recursively dividing the maze into smaller sections, creating a tree structure that represents the maze's layout, and linking cells through a neighbor matrix. The process begins with defining the maze dimensions and initializing the structures needed to represent the maze, such as the neighbor matrix (`nei`), the sections tree (`sections`), and the entries list (`entries`).

The maze is constructed recursively by the `generateMazeStruct()` function, which divides the maze area either vertically or horizontally depending on its dimensions. If the width (`xmax - xmin`) is greater than the height (`ymax - ymin`), the algorithm divides the maze vertically; otherwise, it divides the maze horizontally. This division is done by creating a new wall between two sections and defining the neighboring cells in the `nei` matrix. The algorithm continues subdividing until the entire maze is broken down into small sections, represented as nodes in a binary tree.

Each division involves creating two child nodes (`lchild` and `rchild`), which represent the left and right sections of the maze. These sections are connected by a vertical or horizontal wall represented by two cells (`c1` and `c2`). The neighboring relationships between these two cells are established in the `nei` matrix. The tree structure (`sections`) is updated at each step, creating a new root node with the dividing wall and linking the left and right sections as its children.

Once the maze is divided into sections, entry points are generated by the `generate_maze_entries()` function. Random entry points are selected, ensuring that they do not overlap with the exit or any other entry points. The `entries` list stores these valid entry points. The exit is fixed at the bottom-right corner of the maze.

To generate the maze, we start with a maze made entirely of walls. At each step, the algorithm identifies the line of walls positioned at the midpoint of the longest side of the current section. Within this line, one wall is randomly chosen and replaced with a door, creating a passage between two sub-sections. The algorithm then recursively applies itself to each of these sub-sections, ensuring that the only way to move from one to the other is through this door. The recursion stops once each sub-section is reduced to a single cell.

The maze is then fully constructed and represented by the `Maze` structure, which holds the maze's dimensions, the `nei` matrix, the sections tree, the exit cell, and the list of entry points. This entire process is initiated by the `create_maze()` function, which calls `generateMazeStruct()` to build the maze structure and `generate_maze_entries()` to place the entry points.

During generation, doors are stored in both an adjacency matrix and a section tree. The adjacency matrix allows for constant-time determination of whether a cell is connected to another by a door. The section tree records each door as a pair of cells on either side of the door in each node. The left child represents the sub-section containing the first cell of the pair, while the right child represents the sub-section containing the second cell.

2.2 Maze generation pseudo-code

```
a- section_tree is an empty tree when this function is called and
    is initialized within it.
b- On the first call, nei is an empty matrix, which this function populates.
c- coord specifies which section of the maze is being processed
    in the current recursive call.

generate_maze_structure(tree section_tree, (pointer to a Matrix of listeC)
                        nei,int x1, int y1, int x2, int y2)
{
    if dim(x1,y1,x2,y2) == 1,1 //the maze is 1 cell
    {
        return;
    }
    else
    {
        if width > height
        {
            mid = (y1-y2)/2
            doorx = random(x1, x2-1)
            lchild = emptytree
            rchild = emptytree
            generate_maze_structure(lchild, nei, x1, y1, x2, mid)
            generate_maze_structure(rchild,nei, x1, mid, x2, y2)
            c1 = createCell(doorx, mid)
            c2 = createCell(doorx, mid+1)
            tree = root(c1,c2,rchild,lchild)
            addlistcell (matrix[doorx][mid], c2)
            addlistcell(matrix[doorx][mid+1], c1)
            return;
        }
        else
        {
            mid = (x1-x2)/2
            doory = random(y1, y2-1)
            lchild = emptytree
            rchild = emptytree
            generate_maze_structure(lchild, nei, x1, y1, mid, y2)
            generate_maze_structure(rchild,nei, mid, y1, x2, y2)
            c1 = createCell(mid, doory)
            c2 = createCell(mid+1, doory)
            tree = root(c1,c2,rchild,lchild)
            addlistcell (matrix[mid][doory], c2)
            addlistcell(matrix[mid+1][doory], c1)
            return;
        }
    }
}
```

2.3 A brief description of the maze resolution algorithm

The pathfinding algorithm is designed to find a path from an entry point to the exit within the maze. It operates by recursively traversing the maze, using the tree structure to guide the search and avoiding revisiting cells. The main objective of the algorithm is to identify the shortest or most efficient path from the start (entry) to the end (exit) while avoiding cycles.

The algorithm begins by initializing the starting and ending points. The starting point is retrieved from the `entries` list, and the exit point is set to the `exit` cell in the `Maze` structure. A visited list is used to keep track of the cells that have already been visited to prevent revisiting and creating cycles. The algorithm uses recursion to explore the maze, with each recursive call exploring one step further in the search for the exit.

At each step, the algorithm checks if the current cell (`start`) is equal to the exit (`end`). If the `start` and `end` are the same, the function returns a path consisting only of the start cell. If the `start` and `end` cells are directly connected (i.e., they are neighbors), the function returns a direct path between the two cells.

For the solution, we will use the section tree structure. When we want to move from one cell to another, we start by checking if they are in the same section. If they are, we recursively call the algorithm on that section until the entry and exit are in separate sub-sections. Once this step is reached, we then search for a path from the first cell to the door of the current section, and from that door to the cell in the second section.

If neither of the base cases is satisfied, the algorithm proceeds by identifying the youngest common ancestor (YCA) between the `start` and `end` cells using the `find_youngest_comon_ancestor()` function. The YCA is the most recent point in the tree where the paths from the `start` and `end` cells converge. The algorithm then recursively searches the left and right children of the YCA, depending on whether the division is vertical or horizontal.

After finding the paths from the `start` to the YCA and from the YCA to the `end`, the paths are merged using the `merge_listeC()` function. This merging combines the two subpaths into a single path from the `start` to the `end`. The recursive search continues until the exit is reached, at which point the full path is returned.

In our implementation, when the door is vertical, the left child is the section to the left of the door (and thus the right child is the section to the right of the door). When the door is horizontal, the left child is the section above the door (and thus the right child is the section below the door).

Therefore, when we want to know if two cells are in the same section, we check if the door is horizontal or vertical. If the door is horizontal, then if a cell is to the left of the door, it belongs to the left child; otherwise, it belongs to the right child. If the door is vertical, if the cell is higher than the door, it belongs to the left child; if the cell is lower, it belongs to the right child.

To prevent cycles, the algorithm uses the `path_no_cycle()` function, which ensures that no cell is revisited during the path construction. This function works by maintaining a list of visited cells and checking if each new cell in the path has already been visited.

2.4 Maze resolution pseudo-code

```
functions match_side find in witch section is the cell give in parameter and
return this section for match_side_section or the cell conect to the door
    in the good section for match_side_cell
List find_path(tree section_tree, matrix of list nei, cell start, cell end)
{
    if cell_equal(start, end)
    {
        return append_list(empty_list(), start);
    }
    else
    {
        //
        if(on_same_section(section_tree, start, end))
        {
            selected = common_section(section_tree, start, end);
            //common_section select left child or right child of the tree
            return find_path(selected, start, end);
        }
        else
        {
            new_end = match_side_cell(section_tree, start);
            new_start = match_side_cell(section_tree, end);
            section1 = match_side_section(section_tree, start);
            section2 = match_side_section(section_tree, end);
            //match side afffect the good cell/child to be in the same
            //side than the cell give in argument
            list1 = find_path(section1, start, new_end);
            list2 = find_path(section1, new_start, end);
            return concat(list1, list2);
        }
    }
}
```

3 Theoretical Complexity

3.1 Worst Case Complexity

In the worst case, the start and end cells are always in different sections at each step of the recursion, so the entire tree is explored. Thus, we have linear complexity, and due to the structure of the maze, the number of doors equals the number of cells minus one. Therefore, in the worst case, the complexity is $O(n)$, where n is the number of cells.

3.2 Best Case Complexity

In the best case, the start and end cells are the same cell. In this case, there are no recursive calls; the function simply returns a list containing the only cell that makes up the path. Thus, in the best case, the complexity is $O(1)$.

3.3 Average Complexity

Let $K(n)$ be the complexity in terms of the number of cells in the maze.

$$K(1) = 1 \quad \text{(the maze has only one cell, so there is only one solution)}$$

$$\begin{aligned} K(m) = {} & P(\text{start and end are the same cell}) \\ & + P(\text{the two cells are different and in the same section}) \times K\left(\left\lceil \frac{m}{2} \right\rceil\right) \\ & + P(\text{the two cells are different and in different sections}) \times 2 \times K\left(\left\lceil \frac{m}{2} \right\rceil\right) \\ & + 1 \end{aligned}$$

The terms $K\left(\left\lceil \frac{m}{2} \right\rceil\right)$ come from the fact that each recursive call increases the depth in the tree by 1, and the other half is either not processed or processed in another call.
The +1 term accounts for the operations performed at this stage of the recursive call.

Detailed Calculation:

$$\begin{aligned} K(m) &= \frac{1}{m} + \left(1 - \frac{1}{m}\right) \times \frac{1}{2} \times K\left(\left\lceil \frac{m}{2} \right\rceil\right) + \left(1 - \frac{1}{m}\right) \times \frac{1}{2} \times 2K\left(\left\lceil \frac{m}{2} \right\rceil\right) + 1 \\ &= \frac{1}{m} + 1 + \frac{m-1}{2m} \times K\left(\left\lceil \frac{m}{2} \right\rceil\right) + \frac{m-1}{m} \times K\left(\left\lceil \frac{m}{2} \right\rceil\right) \\ &= \frac{m+1}{m} + \frac{3(m-1)}{2m} \times K\left(\left\lceil \frac{m}{2} \right\rceil\right) \end{aligned}$$

$K(n)$ is bounded above by the worst-case complexity, which is $O(n)$.

Now, let's try to find a lower bound for K. For this, we introduce $K'(m) = K(2^m)$.

We will show that $K'(m)$ can be bounded below by $\left(\frac{9}{8}\right)^{m-1}$ by induction.

Base case:

$$K'(1) = \frac{3}{2} + \frac{3}{4} = \frac{9}{4} \geq 1 = \left(\frac{9}{8}\right)^{1-1}$$

Inductive step:

Suppose $m \geq 2$:

$$K'(m+1) = \frac{2^{m+1}+1}{2^{m+1}} + \frac{3(2^{m+1}-1)}{2 \times 2^{m+1}} K\left(\left\lceil \frac{2^{m+1}}{2} \right\rceil\right)$$

$$= \frac{2^{m+1}+1}{2^{m+1}} + \frac{3(2^{m+1}-1)}{2^{m+2}} K'(m)$$

$$\geq \frac{3(2^{m+1}-1)}{2^{m+2}} K'(m)$$

$$\geq \frac{3(2^{m+1}-1)}{2^{m+2}} \left(\frac{9}{8}\right)^{m-1}$$

We show that $\frac{3(2^{m+1}-1)}{2^{m+2}}$ is greater than $\frac{9}{8}$.

$$\frac{3(2^{m+1}-1)}{2^{m+2}} = \frac{3 \times 2^{m+1} - 3}{2^{m+2}}$$

$$= \frac{3 \times 2^{m+1}}{2^{m+2}} - \frac{3}{2^{m+2}}$$

$$= \frac{(2+1) \times 2^{m+1}}{2^{m+2}} - \frac{3}{2^{m+2}}$$

$$= \frac{2^{m+1}}{2^{m+2}} + \frac{2^{m+2}}{2^{m+2}} - \frac{3}{2^{m+2}}$$

$$= \frac{1}{2} + 1 - \frac{3}{2^{m+2}}$$

This function is increasing, and for $n = 2$:

$$\frac{1}{2} + 1 - \frac{3}{2^{2+2}} = \frac{3}{2} - \frac{3}{16}$$

$$= \frac{21}{16} \geq \frac{18}{16} \geq \frac{9}{8}$$

We can then return to:

$$K'(m+1) \geq \frac{3(2^{m+1}-1)}{2^{m+2}} \left(\frac{9}{8}\right)^{m-1}$$

$$\geq \left(\frac{9}{8}\right) \left(\frac{9}{8}\right)^{m-1}$$

$$\geq \left(\frac{9}{8}\right)^{m}$$

Thus, by induction, $K'(m) \geq \left(\frac{9}{8}\right)^{m-1}$.

We conclude that $K(2^m) \geq \left(\frac{9}{8}\right)^{m-1}$. Since K is increasing and positive, and so is $\left(\frac{9}{8}\right)^{m-1}$, we can apply $\log_2$:

$$K(m) \geq \frac{(m-1)\log\left(\frac{9}{8}\right)}{\log(2)}$$

$$K(m) \geq m\frac{\log\left(\frac{9}{8}\right)}{\log(2)} - \frac{\log\left(\frac{9}{8}\right)}{\log(2)}$$

Therefore, K is bounded below by a linear function in m. K is also bounded above by a linear function in m (worst-case complexity), so K is $\Theta(m)$, where m is the number of cells in the maze.

4 Performance analysis of our algorithm

The analysis was divided in two different parts: the memory and the time performances. For this we did two tests with functions in the code.

4.1 Memory performance

The memory performance was measured using 3 global counters: one that increases with each memory allocation, taking in consideration the size of the data type `total_bytes_allocated`, the other 2 count the recursion depth and the maximum recursion depth `recursive_depth_counter` and `max_recursive_depth`. We have 3 different functions to manipulate these counters and they are defined as so:

1.
```
void track_malloc (size_t size)
{
    total_bytes_allocated += size;
}
```

   This increases the total number of the bytes allocated by a `malloc()` by the size of the data type using the `sizeof` operator.

2.
```
void track_recursive_call ()
{
    recursive_depth_counter++;
    if (recursive_depth_counter > max_recursive_depth) {
        max_recursive_depth = recursive_depth_counter;
    }
}
```

This function increases the recursive depth counter after a recursive call (we call this function at the beginning of the pathfinding function) and if the current depth into the stack is greater than the previous maximum depth, this function sets it to the current depth.

3.
```
void track_recursive_return ()
{
    recursive_depth_counter--;
}
```

Finally, we decrease the current depth after a return (we call this function before merging the different tree parts

4.2 Time performance

The time performance was measured using functions and mechanisms designed to capture the execution time of specific parts of the code. For this, a global function `get_time_in_seconds()` was defined to accurately track the time taken by various sections of the code. This function leverages system-specific implementations to ensure cross-platform compatibility, using either the `QueryPerformanceCounter` for Windows or `gettimeofday` for Unix-based systems.

To evaluate the time performance, we used the following methodology:

1.
```
double get_time_in_seconds() {
    #ifdef _WIN32
    LARGE_INTEGER frequency;
    LARGE_INTEGER count;
    QueryPerformanceFrequency(&frequency);
    QueryPerformanceCounter(&count);
    return (double)count.QuadPart / frequency.QuadPart;
    #else
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec + tv.tv_usec / 1000000.0;
    #endif
}
```

This function returns the current time in seconds, allowing for precise timing measurement by calculating the difference between start and end times.

2. The time performance was also analyzed using more advanced functionality for recording and saving execution details. This is achieved through the function `chronometrer_et_enregistrer`, which captures the start and end times of function executions, calculates the duration, and stores the results in a JSON file for further analysis.

```
double start_time = get_time_in_seconds();
// Execute code block
double end_time = get_time_in_seconds();
chronometrer_et_enregistrer("function_name", start_time, end_time), width, height);
```

   This records the elapsed time, which is saved in JSON file for analysis.

4.3 Analysis Results

For both analysis, we ran the performance tests with mazes of different sizes 5 times each and calculated the mean of the results. After, we ran a linear regression in python using `numpy` and plotted the results using `matplotlib` as you can see in figures 2 and 3.

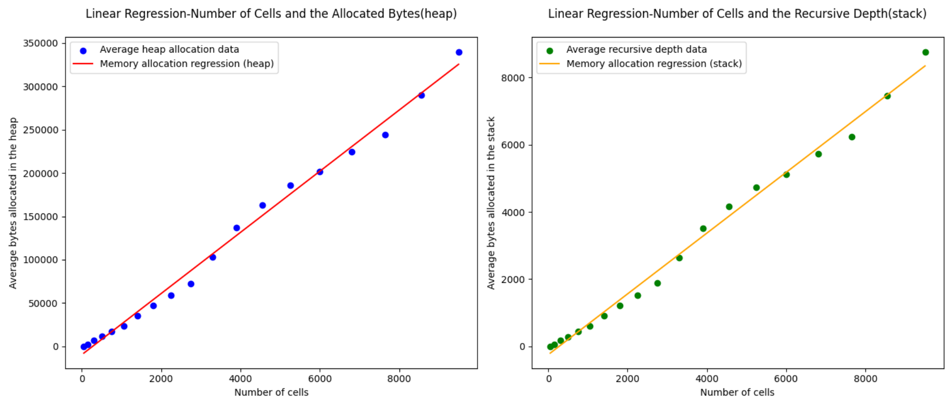


Figure 2: Analysis of the memory performance tests

4.4 Comparison to the BFS algorithm

We decided to compare our algorithm to the logic solution for this problem which is the BFS, as mentioned in section 1. After the theoretical complexity made in section 3, we concluded our algorithm have the same complexity as the BFS. We implemented the code for the BFS algorithm and tested the time it takes for the resolution and we concluded that our algorithm is 2% more efficient than the BFS.

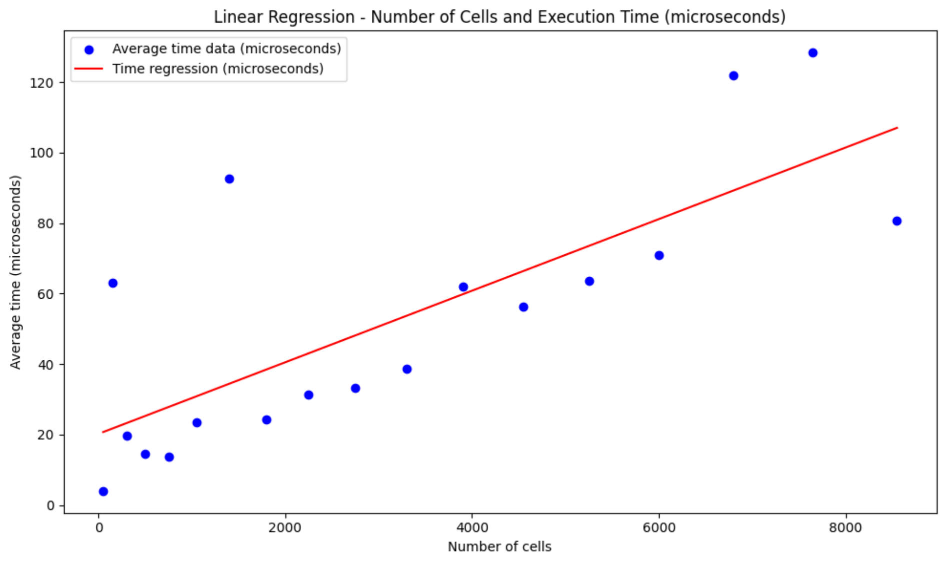


Figure 3: Analysis of the time performance tests

5 Conclusion

5.1 Conclusion about the Data Structure and algorithms chosen

We selected an array of Data Structures tailored for efficient maze generation and traversal, opting for a unique approach that goes beyond standard Breadth-First Search (BFS), to have a direct access to the neighboring cells. The decision to incorporate structures such as Tree for maze sections (the doors), `ListeC` for managing neighbor lists, and cell for representing coordinates aimed to improve computational efficiency and memory management. This design proved beneficial in managing complexity, yet it brought forth challenges related to memory allocation conclusion about the Data Structure and algorithms chosen and recursion handling.

The use of the Tree Data Structure to represent sections of the maze allowed for hierarchical division, aiding in efficient pathfinding by limiting the search area. However, the depth of recursive calls within the tree structure introduced significant memory overhead and recursion depth constraints, especially in larger mazes. The `ListeC` list provided flexibility in managing dynamic lists of cells, which was useful for path tracking and neighbor connectivity. Although this offered ease of list operations, the list's linear traversal occasionally affected time complexity, particularly in functions such as cell in list and merge `listeC`.

5.2 Conclusion about the performance analysis results and the theoretical complexity

5.2.1 Heap Memory Allocation (Left Graph)

The left graph (figure 2) shows a linear regression for the relationship between the number of cells and the allocated bytes in the heap memory. The data points follow a linear trend, as indicated by the red regression line

Conclusion: This linear relationship aligns with the theoretical complexity of memory allocation for arrays or dynamically allocated memory, which grows proportionally with the number of elements. This suggests that the memory allocation process for each additional cell has a constant cost.

5.2.2 Stack Memory Allocation (Middle Graph)

The right graph (figure 2) analyzes recursive depth and stack memory allocation, showing a similar linear trend.

Conclusion: This implies that the stack memory usage also grows linearly with the number of recursive calls. This is expected since each recursive call requires a fixed amount of stack memory for its local variables and return address. Thus, the theoretical complexity of stack memory allocation in this context is O(n) where n is the recursion depth or number of cells.

5.2.3 Execution Time (Bottom Graph)

The bottom graph (figure 3) represents the execution time in microseconds as a function of the number of cells. The scatter plot shows some variability, but the regression line suggests an upward trend.

Conclusion: While execution time increases with the number of cells, there is some dispersion, likely due to variability in system performance or other factors affecting processing time. However, the general trend supports a linear time complexity O(n) for the algorithm, consistent with the expectation for iterative or single-recursive algorithms, where each cell incurs a constant processing time.
One notable strength of our code is its near-instantaneous performance, even when processing mazes with extremely large numbers of cells (e.g., 1 million, 2 million, and beyond). This efficiency highlights that, despite our choice to deviate from a standard Breadth-First Search (BFS) implementation, our solution scales well with maze size, remaining highly effective for very large labyrinths.

5.3 What we could have done differently

Our approach ultimately proved to be only marginally faster than a standard Breadth-First Search (BFS), with performance tests showing around a 2% improvement over BFS on average. This minimal gain did not justify the added complexity and development time involved in implementing a new structure, given the well-established efficiency of BFS for grid-based traversal problems. In retrospect, directly using BFS would have been a more straightforward choice, as it would have provided reliable linear complexity with simpler implementation and fewer resource overheads.

If we had chosen to work with BFS from the outset, we would have saved significant time and effort, which could have been redirected toward optimizing other parts of the project or enhancing additional functionalities. Our initial decision to pursue a novel approach was based on the hope of finding a significantly faster alternative, but as our results showed, a 2% gain does not justify the extra hard work introduced.

5.4 Lessons Learned from Solving This Problem

1- Practical Overhead vs. Theoretical Gains: Marginal theoretical improvements do not always translate into significant practical gains. Given the minimal performance boost observed, BFS's simplicity and well-understood behavior would have made it the preferable choice.

2- Importance of Early Benchmarking: Running performance benchmarks early on would have given us clearer feedback about the efficiency of our approach compared to BFS. This could have helped us realize sooner that our method offered only slight advantages, allowing us to pivot back to BFS if necessary.

3- Balancing Innovation and Reliability: While exploring innovative approaches can be rewarding, it is important to balance this with practical considerations. Reliable, well-established algorithms like BFS can often be the optimal choice, especially when the task requires predictable performance.

5.5 Questions

1 - Optimizing Known Algorithms for Edge Cases: BFS is often cited as the most efficient for grid traversal, but are there optimizations within BFS that could further improve its performance for specific cases? For example, are there any memory-efficient BFS variations that could be useful for large grids?
2- Estimating Complexity in Hybrid Structures: In situations where hybrid structures, like our tree-linked list approach, are used, how can we best estimate their complexity in a way that accounts for both time and memory overhead? Is there a systematic way to assess the impact of combining Data Structures in terms of performance?

6 Workload distribution

During this project, our team initially consisted of six members, though one member dropped out at the beginning of the semester. Despite this, the remaining team members collaborated actively, working both independently and in smaller groups to cover the different aspects of the project.

At the outset, we held discussions to evaluate possible algorithms for the maze generation. By the following session, we had finalized our Data Structure and selected a maze generation algorithm that integrated ideas from all contributors. This collaborative process allowed us to establish a strong foundation for the algorithm, with the pseudo-code created collectively and the final C implementation led by Gwénaël, with additional input and support from Hassan and Jeronimo to ensure functionality and smooth maintenance.

After the initial development, we conducted multiple tests on the generation algorithm to verify its robustness. These tests included validating various entry points within the maze and ensuring a consistent exit point, confirming that the maze structure was both functional and met the requirements we had set out.

When it came to the path resolution algorithm, we faced a new challenge. Given our team's diverse perspectives, we had several ideas to evaluate. After careful review and discussion, we arrived at an algorithm that merged the best aspects of everyones suggestions. Rigorous testing followed, aimed at validating the algorithm's accuracy and efficiency. We designed specific tests, such as the Path Connectivity Test, Path and List Merging Test, Obstacle Avoidance Test, Cycle and Shortest Path Test, and Start and Exit Validation, to ensure the resolution algorithm would perform correctly and efficiently within our maze.

Following these functionality tests, we shifted our focus to performance analysis. Each team member contributed to evaluating key metrics, including comparisons with a Breadth-First Search (BFS) algorithm for benchmarking, as well as time and memory consumption assessments. To gain further insights, we developed a Python script for a more in-depth statistical analysis, incorporating data visualizations and linear regression to verify theoretical complexity and provide clear insights into our algorithm's efficiency.

In addition to the technical tasks, writing the report became an integral part of our collaborative effort. Documenting the project required a structured approach, where each team member contributed to different sections. Throughout the report-writing phase, we worked to create a cohesive and comprehensive document that accurately reflected our methodology, findings, and conclusions. Hassan and Jeronimo took the lead in organizing and editing the final draft, ensuring clarity and consistency across sections.

Each phase of the project-from brainstorming algorithms to testing, performance analysis, and report writing-benefited from the team's combined effort. This collaborative approach allowed us to deliver a thorough and well-documented project that demonstrated the strength of our teamwork and individual contributions.

If we had to give out grades based of n points everyone would get the same amount of points, so $n/5$.

7 Appendix

Below, you can find our code. We used a Git repository to organize our work, which includes a Makefile, a .gitignore file, and .h header files.

7.1 C Code of the maze structure

```c
#include "maze.h"

//precondition : nei is a matrix of listeC initialize empty and tree will be filled in this function
void generateMazeStruct(Tree* tree,ListeC** nei, int xmin, int ymin, int xmax, int ymax){
    if(xmax-xmin <= 0 && ymax-ymin <= 0){
        *tree = NULL;
        return;
    }
    if(xmax-xmin > ymax-ymin){ //section with more width than height
        int xmid = (xmax+xmin)/2;
        int yrand = ymin + (rand() % (ymax-ymin+1));
        Tree lchild = NULL;
        Tree rchild = NULL;
        generateMazeStruct(&lchild, nei, xmin, ymin, xmid, ymax);
        generateMazeStruct(&rchild, nei, xmid+1, ymin, xmax, ymax);
        cell c1 = {xmid, yrand};
        cell c2 = {xmid+1, yrand};
        nei[xmid][yrand] = append_list_cell(nei[xmid][yrand], c2);
        nei[xmid+1][yrand] = append_list_cell(nei[xmid+1][yrand], c1);
        *tree = root(c1, lchild, c2, rchild);
    }
    else{
        int ymid = (ymax+ymin)/2;
        int xrand = xmin + (rand() % (xmax-xmin+1));
        Tree lchild = NULL;
        Tree rchild = NULL;
        generateMazeStruct(&lchild, nei, xmin, ymin, xmax, ymid);
        generateMazeStruct(&rchild, nei, xmin, ymid+1, xmax, ymax);
        cell c1 = {xrand, ymid};
        cell c2 = {xrand, ymid+1};
        nei[xrand][ymid] = append_list_cell(nei[xrand][ymid], c2);
        nei[xrand][ymid+1] = append_list_cell(nei[xrand][ymid+1], c1);
        *tree = root(c1, lchild, c2, rchild);
    }
}


void generate_maze_entries(Maze m, int nb_entries) {
    if (nb_entries <= 0) {
        return;
    }
    int x = rand() % m->X;
    int y = rand() % m->Y;
    cell c = new_cell(x, y);
    if (!cell_equal(c, m->exit) && !(cell_in_list(c, m->entries))) {
        m->entries = append_list_cell(m->entries, c);
        generate_maze_entries(m, nb_entries - 1);
    }
    else {
        generate_maze_entries(m, nb_entries);
    }
}

// the number of entries is a really naive algorithm (it is slow when the
   number of entries is really high)
```

```c
// nb_entries must < x * y
Maze create_maze(int x, int y, int nb_entries) {
    Maze ret = malloc(sizeof(maze));
    ret->X = x;
    ret->Y = y;
    ret->nei = empty_nei_matrix(x, y);
    ret->sections = NULL;
    generateMazeStruct(&ret->sections, ret->nei, 0, 0, x - 1, y - 1);
    ret->exit = new_cell(x - 1, y - 1);
    ret->entries = NULL;
    generate_maze_entries(ret, nb_entries);
    return ret;
}


void print_maze(Maze m, ListeC solution_path) {

    // First row of #
    for (int i = 0; i < m->X; i++) {
        printf(WHITE "##" RESET);
    }
        printf("#"); //Final column #
        printf("\n");


    for (int j = 0; j < m->Y; j++) { //for height j
        printf(WHITE "#" RESET); // first # of the row

    //First width loop
    for (int i = 0; i < m->X; i++) { //for width i
        cell current_cell = {i, j};
        //Integers to know the status of my cell : entree, exit, part of
           the path or nothing
        int is_solution = cell_in_list(current_cell, solution_path);
        int is_entry = cell_in_list(current_cell, m->entries);
        int is_exit = cell_equal(current_cell, m->exit);

        // Choose the symbol to print for our current cell
        if (is_entry) {
            printf(GREEN "E" RESET); // Entree in green
        } else if (is_exit) {
            printf(YELLOW "S" RESET); // Exit in yellow
        } else if (is_solution) {
        printf(RED "P" RESET); // Path in red
        } else {
            printf(" "); // White space for nothing
        }

        // Print the wall or the white space on the right of our cell <=>
           If the two cells are connected
        if (cell_in_list(new_cell(i + 1, j), m->nei[i][j])) {
            cell right_cell_of_the_current_one = {i+1, j};
            int right_is_solution_too = cell_in_list (
               right_cell_of_the_current_one, solution_path);
        if (is_solution && !is_entry && !is_exit) {

            if(right_is_solution_too){
                printf(RED "P" RESET); // Print red P between the two
                   cells which are in the path
            }
            else {
                printf(" "); //White space
            }

        }
        else if(is_entry && right_is_solution_too){
            printf(RED "P" RESET);
        }

        else {
            printf(" "); // White space
        }
        } else {
            printf(WHITE "#" RESET); // Right wall of the cell
        }
    }
    //End of first loop

    // Go to the next line
    printf("\n");

    // Start the second line with the # of the first column
    printf(WHITE "#" RESET);

    //Second width loop
    for (int i = 0; i < m->X; i++) { //For width i
        cell current_cell = {i, j};
        int is_solution = cell_in_list(current_cell, solution_path);
        int is_entry = cell_in_list(current_cell, m->entries);
        int is_exit = cell_equal(current_cell, m->exit);

    // Print the wall or white space at the bottom <=> our two cells are
       connected
    if (cell_in_list(new_cell(i, j+1), m->nei[i][j])) {
        cell bottom_cell_of_the_current_one = {i, j+1};
        int bottom_is_solution_too = cell_in_list (
           bottom_cell_of_the_current_one , solution_path);

        if (is_solution && !is_entry && !is_exit) {
            if(bottom_is_solution_too){
                printf(RED "P" RESET); // Print red P between the two
                   cells which are in the path
            }
            else{
                printf(" "); // White space
            }

        }
        else if(is_entry && bottom_is_solution_too){
            printf(RED "P" RESET);
        }

        else {
            printf(" "); // White Space
        }
    }
        else {
            printf(WHITE "#" RESET); // Wall below our current cell cause
               not connected
        }

    printf(WHITE "#" RESET); //Diagonal wall => always # cause we can't
       move diagonally

    }
    //End of the second loop

    printf("\n"); //Go to the next pair of rows
    }

    printf("\n"); //for a clean printing

}


void free_maze(Maze test) {
    free_matrixe(test->nei, X_maze, Y_maze);
    free_tree(test->sections);
    free_list(test->entries);
    free(test);}
```

7.2 C Code of the tree structure

```c
#include "tree.h"

size_t total_bytes_allocated = 0;
int recursive_depth_counter = 0;
int max_recursive_depth = 0;

void track_malloc (size_t size)
{
    total_bytes_allocated += size;
}

void track_recursive_call ()
{
    recursive_depth_counter++;
    if (recursive_depth_counter > max_recursive_depth) {
        max_recursive_depth = recursive_depth_counter;
    }
}

void track_recursive_return ()
{
    recursive_depth_counter--;
}

Tree root(cell c1, Tree lchild, cell c2, Tree rchild)
{
    Tree ret = malloc(sizeof(node));
    ret->a = cellcp(c1);
    ret->b = cellcp(c2);
    ret->lchild = lchild;
    ret->rchild = rchild;
    return ret;
}

bool vertical_door(Tree t)
{
    return t->a.y == t->b.y;
}

Tree find_youngest_comon_ancestor(Tree sections, cell start, cell end)
{
    if (vertical_door(sections) && onleft(start, sections->a) && onleft(
       end, sections->a))
        return find_youngest_comon_ancestor(sections->lchild, start, end);
    if (vertical_door(sections) && onleft(start, sections->a) && onleft(
       sections->b, end))
        return sections;
    if (vertical_door(sections) && onleft(sections->b, start) && onleft(end, sections->a))
        return sections;
    if (vertical_door(sections) && onleft(sections->b, start) && onleft(sections->b, end))
        return find_youngest_comon_ancestor(sections->rchild, start, end);
    if (!vertical_door(sections) && upper(start, sections->a) && upper(end, sections->a))
        return find_youngest_comon_ancestor(sections->lchild, start, end);
    if (!vertical_door(sections) && upper(start, sections->a) && upper(sections->b, end))
        return sections;
    if (!vertical_door(sections) && upper(sections->b, start) && upper(end, sections->a))
        return sections;
    if (!vertical_door(sections) && upper(sections->b, start) && upper(sections->b, end))
        return find_youngest_comon_ancestor(sections->rchild, start, end);
    return NULL;
}

ListeC find_path(Tree sections, ListeC** nei, cell start, cell end)
{
    track_recursive_call();

    if (cell_equal(start, end)) {
        track_malloc(sizeof(listeC));
        return append_list_cell(NULL, start);
    }
    if (connected(nei, start, end)) {
        track_malloc(sizeof(listeC));
        track_malloc(sizeof(listeC));
        return append_list_cell(append_list_cell(NULL, end), start);
    }
    Tree yca = find_youngest_comon_ancestor(sections, start, end);
    ListeC part1;
    ListeC part2;
    if ((vertical_door(yca) && start.x <= yca->a.x) || (!vertical_door(yca) && start.y <= yca->a.y)) {
        part1 = find_path(yca->lchild, nei, start, yca->a);
        part2 = find_path(yca->rchild, nei, yca->b, end);
    } else {
        part1 = find_path(yca->rchild, nei, start, yca->b);
        part2 = find_path(yca->lchild, nei, yca->a, end);
    }

    track_recursive_return();
    return merge_listeC(part1, part2);
```

```c
}

void free_tree(Tree tree)
{
    if (tree != NULL) {
        free_tree(tree->lchild);
        free_tree(tree->rchild);
        free(tree);
    }
}

bool cell_in_visited(cell c, ListeC visited)
{
    while (visited != NULL) {
        if (cell_equal(visited->a, c)) {
            return true;
        }
        visited = visited->next;
    }
    return false;
}

bool path_no_cycle(ListeC path)
{
    ListeC visited = NULL;
    ListeC current = path;

    while (current != NULL) {
        if (cell_in_visited(current->a, visited)) {
            free_list(visited);
            return false;
        }
        visited = append_list_cell(visited, current->a);

        current = current->next;
    }

    free_list(visited);
    return true;
}

void export_memory_counters_to_json(const char *filename, int width, int
   height)
{
    FILE *file = fopen(filename, "r");
    char *existing_content = NULL;
    size_t length = 0;

    if (file != NULL) {
        fseek(file, 0, SEEK_END);
        length = ftell(file);
        fseek(file, 0, SEEK_SET);

        existing_content = (char *)malloc(length + 1);
        if (existing_content == NULL) {
            perror("Error allocating memory for existing content");
            fclose(file);
            return;
        }

        fread(existing_content, 1, length, file);
        existing_content[length] = '\0';
        fclose(file);
    }

    file = fopen(filename, "w");
    if (file == NULL) {
        perror("Error opening file");
        if (existing_content) free(existing_content);
        return;
    }

    if (existing_content == NULL || length == 0) {
        fprintf(file, "[\n");
    } else {
        if (length > 2) {
            existing_content[length - 2] = '\0';
            fprintf(file, "%s,\n", existing_content);
        }
    }

    fprintf(file, "    {\n");
    fprintf(file, "        \"total_bytes_allocated (heap)\": %zu,\n",
       total_bytes_allocated);
    fprintf(file, "        \"peak_recursive_depth (stack)\": %d,\n",
       max_recursive_depth);
    fprintf(file, "        \"maze_width\": %d,\n", width);
    fprintf(file, "        \"maze_height\": %d\n", height);
    fprintf(file, "    }\n");

    fprintf(file, "]\n");

    if (existing_content) free(existing_content);
    fclose(file);
}
```

7.3 C code of the list structure

```c
#include <stdlib.h>
#include "list.h"

ListeC append_list_cell(ListeC l, cell c) {
    ListeC ret = malloc(sizeof(ListeC));
    ret->a = cellcp(c);
    ret->next = l;
    if (l == NULL) {
        ret->last = ret;
    } else {
        ret->last = l->last;
    }
    return ret;
}

bool listeC_has_comon_element(ListeC l1, ListeC l2) {
    for (ListeC i = l1; i != NULL; i = i->next) {
        for (ListeC j = l2; j != NULL; j = j->next) {
            if (cell_equal(i->a, j->a)) {
                return true;
            }
        }
    }
    return false;
}

bool cell_in_list(cell c, ListeC l) {
    if (l == NULL) {
        return false;
    }
    if (cell_equal(c, l->a)) {
        return true;
    }
    return cell_in_list(c, l->next);
}

//warning merge list dont update link with last element in l1, only first
   and last element
ListeC merge_listeC(ListeC l1, ListeC l2) {
    if (l1 == NULL) {
        return l2;
    }
    if (l2 == NULL) {
        return l1;
    }
    l1->last->next = l2;
    l1->last = l2->last;
    return l1;
}

bool connected(ListeC** nei, cell c1, cell c2) {
    return cell_in_list(c2, nei[c1.x][c1.y]);
}

ListeC** empty_nei_matrix(int x, int y) {
    ListeC** ret = malloc(sizeof(ListeC*) * x);
    for (int i = 0; i < x; i++) {
        ret[i] = malloc(sizeof(ListeC) * y);
        for (int j = 0; j < y; j++) {
            ret[i][j] = NULL;
        }
    }
    return ret;
}

bool test_connexe_path(ListeC path) {
    if (path == NULL) {
        return true;
    }
    while (path->next != NULL) {
        if (!cell_nei(path->a, path->next->a)) {
            return false;
        }
        path = path->next;
    }
    return true;
}

void print_path(ListeC path) {
    if (path == NULL) {
        printf("done\n");
    } else {
        printf("(%d,%d) -> ", path->a.x, path->a.y);
        print_path(path->next);
    }
}

void free_list(ListeC list) {
    while (list != NULL) {
        ListeC temp = list;
        list = list->next;
        free(temp);
    }
}

```

```c
void free_matrixe(ListeC** mat, int x, int y) {
    for (int i = 0; i < x; i++) {
        for (int j = 0; j < y; j++) {
            free_list(mat[i][j]);
        }
        free(mat[i]);
    }
    free(mat);
}

void test_merge_listeC_normale() {
    ListeC list1 = NULL;
    list1 = append_list_cell(list1, new_cell(1, 1));
    list1 = append_list_cell(list1, new_cell(2, 2));

    ListeC list2 = NULL;
    list2 = append_list_cell(list2, new_cell(3, 3));
    list2 = append_list_cell(list2, new_cell(4, 4));

    ListeC merged_list = merge_listeC(list1, list2);

    ListeC courant = merged_list;
    printf("\n");
    print_path(courant);
    assert(courant != NULL);
    assert(cell_equal(courant->a, new_cell(2, 2)));
    courant = courant->next;
    assert(cell_equal(courant->a, new_cell(1, 1)));
    courant = courant->next;
    assert(cell_equal(courant->a, new_cell(4, 4)));
    courant = courant->next;
    assert(cell_equal(courant->a, new_cell(3, 3)));
    courant = courant->next;
    assert(courant == NULL);
    free_list(merged_list);
    printf("Test de merge_listeC r\\uc{e}ussi.\n");
}
```

7.4 C code of the cell structure

```c
#include "cell.h"

cell new_cell(int x, int y){
    return (cell){x,y};
}

cell cellcp(cell c){
    return (cell){c.x, c.y};
}

cell add_cst_to_cell(cell c, int x, int y){
    return (cell){c.x+x, c.y+y};
}

bool cell_equal(cell c1, cell c2){
    return c1.x == c2.x && c1.y == c2.y;
}

bool cell_nei(cell c1, cell c2){
    if(c1.x == c2.x){
        return c1.y == c2.y-1 || c1.y == c2.y+1;
    }
    if(c1.y == c2.y){
        return c1.x == c2.x-1 || c1.x == c2.x+1;
    }
    return false;
}

bool onleft(cell c1, cell c2){
    return c1.x <= c2.x;
}

bool upper(cell c1, cell c2){
    return c1.y <= c2.y;
}
```

7.5 C code of our main

```c
#include <stdlib.h>
#include <stdbool.h>
#include <stdio.h>
#include <time.h>
#include <assert.h>
#include <string.h>
#include "maze.h"

int main(){
    srand( time( NULL ) );
    Maze test = create_maze(X_maze,Y_maze, NB_entries);

    ListeC chemin = find_path(test->sections, test->nei,test->entries->a,
       test->exit);

    print_maze(test, chemin);
    print_path(chemin);

    export_memory_counters_to_json("memory.json", X_maze, Y_maze);
    //printf("Does the path has no cycles? %s\n", path_no_cycle(chemin) ? "true" : "false");
    free_list(chemin);

    //test_merge_listeC_normale();
    free_maze(test);

    // int maze_sizes[] = {5, 10, 20, 50, 100};
    // int num_sizes = sizeof maze_sizes / sizeof maze_sizes[0];

}
```

7.6 Python code of the memory analysis

```
import json
from collections import defaultdict
import numpy as np
import matplotlib.pyplot as plt

# Charger les donnÃ©es du fichier JSON
with open('memory.json', 'r') as file:
    data = json.load(file)

grouped_data = defaultdict(list)

for entry in data:
    \# CrÃ©ation de la clÃ© pour le regroupement
    key = (entry['maze_width'], entry['maze_height'])
    filtered_entry = {
        'total_bytes_allocated (heap)': entry['total_bytes_allocated (heap)'],
        'peak_recursive_depth (stack)': entry['peak_recursive_depth (stack)']
    }
    grouped_data[key].append(filtered_entry)

# Moyenne des donnÃ©es groupÃ©es
averaged_data = []
for (maze_width, maze_height), entries in grouped_data.items():
    avg_total_bytes_allocated = sum(e['total_bytes_allocated (heap)'] for e in entries) / len(entries)
    avg_peak_recursive_depth = sum(e['peak_recursive_depth (stack)'] for e in entries) / len(entries)

    averaged_data.append({
        'maze_width': maze_width,
        'maze_height': maze_height,
        'cell_count': maze_width * maze_height,
        'average_total_bytes_allocated (heap)': avg_total_bytes_allocated,
        'average_peak_recursive_depth (stack)': avg_peak_recursive_depth
    })

# Tracer les donnÃ©es et les rÃ©gressions linÃ©aires
cell_counts = [entry['cell_count'] for entry in averaged_data]
avg_bytes_allocated = [entry['average_total_bytes_allocated (heap)'] for entry in averaged_data]
avg_recursive_depth= [entry['average_peak_recursive_depth (stack)'] for entry in averaged_data]

poly = np.polyfit(cell_counts, avg_bytes_allocated, 1)  # Polyfit de degrÃ© 1 (rÃ©gression linÃ©aire)
poly2 = np.polyfit(cell_counts, avg_recursive_depth, 1)
regression = np.poly1d(poly)
regression2 = np.poly1d(poly2)

# CrÃ©er une figure avec deux sous-graphes cÃ´te Ã  cÃ´te
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))  # 1 ligne, 2 colonnes

# Tracer les donnÃ©es pour l'allocation mÃ©moire et la rÃ©gression linÃ©aire (Graphique 1)
ax1.scatter(cell_counts, avg_bytes_allocated, label='Average heap allocation data', color='blue')
ax1.plot(cell_counts, regression(cell_counts), label='Memory allocation regression (heap)', color='red')
ax1.set_xlabel('Number of cells')
ax1.set_ylabel('Average bytes allocated in the heap')
ax1.set_title('Linear Regression-Number of Cells and the Allocated Bytes(heap)', pad=20)
ax1.legend(loc='upper left')

# Tracer les donnÃ©es pour la profondeur rÃ©cursive et la rÃ©gression linÃ©aire (Graphique 2)
ax2.scatter(cell_counts, avg_recursive_depth, label='Average recursive depth data', color='green')
ax2.plot(cell_counts, regression2(cell_counts), label='Memory allocation regression (stack)', color='orange')
ax2.set_xlabel('Number of cells')
ax2.set_ylabel('Average bytes allocated in the stack')
ax2.set_title('Linear Regression-Number of Cells and the Recursive Depth(stack)', pad=20)
ax2.legend(loc='upper left')
```

```

# Ajuster les espaces entre les deux graphiques
plt.tight_layout()

plt.savefig('regression_linÃ©aire_memory.png')
# Afficher le graphique avec les deux rÃ©gressions cÃ´te Ã  cÃ´te
plt.show()

# Afficher les coefficients de rÃ©gression
print(f"Coefficients de rÃ©gression pour l'allocation mÃ©moire : {poly}")
print(f"Coefficients de rÃ©gression pour la profondeur rÃ©cursive : {poly2
    }")
```

7.7 Python code of the time analysis

```
import json
from collections import defaultdict
import numpy as np
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

# Charger les donnÃ©es du fichier JSON
with open('resultats_temps.json', 'r') as file:
    data = json.load(file)

grouped_data = defaultdict(list)

for entry in data:
    # CrÃ©ation de la clÃ© pour le regroupement
    key = (entry['width'], entry['height'])
    filtered_entry = {
        'duree_seconds': entry['duree_seconds'],
    }
    grouped_data[key].append(filtered_entry)

# Afficher les groupes pour vÃ©rifier
# print("DonnÃ©es groupÃ©es :", grouped_data)

averaged_data = []
for (maze_width, maze_height), entries in grouped_data.items():
    avg_time = sum(e['duree_seconds'] for e in entries) / len(entries)
    avg_time_microseconds = avg_time * 1_000_000
    averaged_data.append({
        'width': maze_width,
        'height': maze_height,
        'cell_count': maze_width * maze_height,
        'avg_duree_microseconds': avg_time_microseconds,
    })

print("DonnÃ©es moyennÃ©es :", averaged_data)

cell_counts = [entry['cell_count'] for entry in averaged_data]
avg_time_taken = [entry['avg_duree_microseconds'] for entry in averaged_data]

# Calcul de la rÃ©gression linÃ©aire
poly = np.polyfit(cell_counts, avg_time_taken , 1)
regression = np.poly1d(poly)

# CrÃ©ation d'une seule figure
plt.figure(figsize=(10, 6))
plt.scatter(cell_counts, avg_time_taken, label='Average time data (microseconds)', color='blue')
plt.plot(cell_counts, regression(cell_counts), label='Time regression (microseconds)', color='red')
plt.xlabel('Number of cells')
plt.ylabel('Average time (microseconds)')
plt.title('Linear Regression - Number of Cells and Execution Time (microseconds)')
plt.legend(loc='upper left')

# Sauvegarder et afficher la figure
plt.tight_layout()
plt.savefig('regression_linÃ©aire_time_microseconds.png')
plt.show()

# Afficher les coefficients de rÃ©gression
print(f"Coefficients de rÃ©gression pour le temps d'exÃ©cution (microsecondes) : {poly}")
```

7.8 chronoTest.c

```
// chronoTest.c

#include "chronoTest.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#ifdef _WIN32
#include <windows.h>
double get_time_in_seconds() {
    LARGE_INTEGER frequency;
    LARGE_INTEGER count;
    QueryPerformanceFrequency(&frequency);
    QueryPerformanceCounter(&count);
    return (double)count.QuadPart / frequency.QuadPart;
}
#else
#include <sys/time.h>
double get_time_in_seconds() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec + tv.tv_usec / 1000000.0;
}
#endif

// Fonction pour Ã©chapper les caractÃ¨res spÃ©ciaux dans une chaÃ®ne JSON
void echapper_json(const char* src, char* dest, size_t dest_size) {
    size_t j = 0;
    for (size_t i = 0; src[i] != '\0' && j < dest_size - 1; i++) {
        if (src[i] == '\"' || src[i] == '\\') {
            if (j + 2 >= dest_size) break; // Assurer la sÃ©curitÃ©
            dest[j++] = '\\';
            dest[j++] = src[i];
        } else {
            dest[j++] = src[i];
        }
    }
    dest[j] = '\0';
}

// Fonction pour lire tout le contenu d'un fichier
char* lire_fichier(const char* nom_fichier) {
    FILE *fichier = fopen(nom_fichier, "rb"); // Ouvrir en mode binaire
    if (fichier == NULL) {
        return NULL; // Fichier n'existe pas
    }

    fseek(fichier, 0, SEEK_END);
    long taille = ftell(fichier);
    fseek(fichier, 0, SEEK_SET);

    char *contenu = (char*)malloc(taille + 1);
    if (contenu == NULL) {
        fclose(fichier);
        return NULL;
    }

    size_t lu = fread(contenu, 1, taille, fichier);
    if (lu != taille) {
```

```c
        // Erreur de lecture
        free(contenu);
        fclose(fichier);
        return NULL;
    }

    contenu[taille] = '\0';
    fclose(fichier);
    return contenu;
}

// Fonction pour Ã©crire une chaÃ®ne dans un fichier
int ecrire_fichier(const char* nom_fichier, const char* contenu) {
    FILE *fichier = fopen(nom_fichier, "wb"); // Ouvrir en mode binaire
    if (fichier == NULL) {
        return -1; // Erreur d'ouverture
    }

    size_t ecrit = fwrite(contenu, 1, strlen(contenu), fichier);
    if (ecrit != strlen(contenu)) {
        // Erreur d'Ã©criture
        fclose(fichier);
        return -1;
    }

    fclose(fichier);
    return 0;
}

// Fonction de chronomÃ©trage et enregistrement
void chronometrer_et_enregistrer(const char* nom_fonction, double debut, double fin, int width, int height) {
    double duree = fin - debut; // DurÃ©e en secondes

    // Echapper les caracteres speciaux dans le nom de la fonction
    char nom_fonction_echappe[256];
    echapper_json(nom_fonction, nom_fonction_echappe, sizeof(nom_fonction_echappe));

    // CrÃ©er le nouvel objet JSON en tant que chaÃ®ne avec width et height
    char nouvel_objet[512];
    snprintf(nouvel_objet, sizeof(nouvel_objet),
             "    {\n        \"fonction\": \"%s\",\n        \"duree_seconds\": %.6f,\n        \"width\": %d,\n        \"height\": %d\n    }",
             nom_fonction_echappe, duree, width, height);

    // Lire le fichier JSON existant
    char* contenu = lire_fichier("resultats.json");
    if (contenu == NULL) {
        // Le fichier n'existe pas, crÃ©er un nouveau tableau JSON avec le premier objet
        char premier_tableau[600];
        snprintf(premier_tableau, sizeof(premier_tableau), "[\n%s\n]", nouvel_objet);
        if (ecrire_fichier("resultats.json", premier_tableau) != 0) {
            fprintf(stderr, "Erreur lors de l'Ã©criture du fichier JSON.\n");
            return;
        }
        // Pas besoin de free(contenu) ici car contenu est NULL
    }
    else {
        // Traiter les espaces blancs Ã  la fin du contenu
        size_t contenu_length = strlen(contenu);
        while (contenu_length > 0 &&
               (contenu[contenu_length - 1] == '\n' ||
                contenu[contenu_length - 1] == '\r' ||
                contenu[contenu_length - 1] == ' ' ||
                contenu[contenu_length - 1] == '\t')) {
            contenu_length--;
        }

        if (contenu_length == 0 || contenu[contenu_length - 1] != ']') {
            fprintf(stderr, "Format JSON incorrect dans resultats.json.\n");
            free(contenu);
            return;
        }

        contenu[contenu_length - 1] = '\0'; // Supprimer le dernier ']'

        // VÃ©rifier si le tableau est vide (c'est-Ã -dire, commence par '[' et se termine immÃ©diatement)
        int is_empty_array = 1;
        for (size_t i = 0; i < contenu_length - 1; i++) {
            if (contenu[i] != '[' && contenu[i] != ' ' && contenu[i] != '\n' && contenu[i] != '\r' && contenu[i] != '\t') {
                is_empty_array = 0;
                break;
            }
        }

        // PrÃ©parer le nouveau contenu
        char* nouveau_contenu = NULL;
        if (is_empty_array) {
            // Si le tableau est vide, ajouter simplement le nouvel objet
            size_t nouvelle_taille = contenu_length + strlen(nouvel_objet) + 3; // +3 pour '\n', ']' et '\0'
            nouveau_contenu = (char*)malloc(nouvelle_taille);
            if (nouveau_contenu == NULL) {
                fprintf(stderr, "Erreur d'allocation mÃ©moire.\n");
                free(contenu);
                return;
            }
            snprintf(nouveau_contenu, nouvelle_taille, "[\n%s\n]", nouvel_objet);
        } else {
            // Si le tableau n'est pas vide, ajouter une virgule et le nouvel objet
            size_t nouvelle_taille = contenu_length + strlen(nouvel_objet) + 5; // +4 pour ',', '\n', ']' et '\0'
            nouveau_contenu = (char*)malloc(nouvelle_taille);
            if (nouveau_contenu == NULL) {
                fprintf(stderr, "Erreur d'allocation mÃ©moire.\n");
                free(contenu);
                return;
            }
            snprintf(nouveau_contenu, nouvelle_taille, "%s,\n%s\n]", contenu, nouvel_objet);
        }

        // Ecrire le nouveau contenu dans le fichier
        if (ecrire_fichier("resultats.json", nouveau_contenu) != 0) {
            fprintf(stderr, "Erreur lors de l'Ã©criture du fichier JSON.\n");
        }

        free(contenu);
        free(nouveau_contenu);
    }
}

void free_Chrono_test_maze(Maze test, int taille_X, int taille_Y) {
    free_matrixe(test->nei, taille_X, taille_Y);
    free_tree(test->sections);
    free_list(test->entries);
    free(test);
}

void chronoTest(int nb_maze, int start_width, int start_height){
    for(int i = 0; i < nb_maze; i++){
        Maze test = create_maze((start_width + (5 * i)), (start_height +
           (5 * i)), NB_entries);
        double debut = get_time_in_seconds();
        ListeC chemin = find_path(test->sections, test->nei, test->entries
           ->a, test->exit);
        double fin = get_time_in_seconds();
        chronometrer_et_enregistrer("find_path", debut, fin, test->X, test
           ->Y);
        free_list(chemin);
        free_Chrono_test_maze(test, test->X, test->Y);
    }
}

void chronoTestBis(int nb_maze){
    for(int i = 5; i < nb_maze; i=i+5){
        for(int x = 0; x < 5; x++){
            Maze test = create_maze((i), (i+5), NB_entries);
            double debut = get_time_in_seconds();
            ListeC chemin = find_path(test->sections, test->nei, test->
               entries->a, test->exit);
            double fin = get_time_in_seconds();
            chronometrer_et_enregistrer("find_path", debut, fin, test->X,
               test->Y);
            free_list(chemin);
            free_Chrono_test_maze(test, test->X, test->Y);
        }}}
```

7.9 testIO.c

```
#include "testIO.h"

int test_entries_exit(Maze maze){
    if(maze->entries == NULL){
        return 1;
    }
    ListeC lEntries = maze->entries;
    while(lEntries->next != NULL){
        ListeC chemin = find_path(maze->sections, maze->nei,lEntries->a,
           maze->exit);
        if(chemin->a.x != lEntries->a.x || chemin->a.y != lEntries->a.y){
            return 1;
        }
        while(chemin->next != NULL){
            chemin = chemin->next;
        }
        if(chemin->a.x != maze->exit.x || chemin->a.y != maze->exit.y){
            return 1;
        }
        lEntries = lEntries->next;
    }
    return 0;
```

$X(f) = \int_{\_} x(t) exp(-i * 2 * \pi * f * t)\ dt$

$x(t) = \int_{\_} X(f) exp(i * 2 * \pi * f * t)\ df$
$x^{@}(t) = \int [X(f) exp(i * 2 * \pi * f * t)\ ]^{@}\ df$

$[X(f) exp(i * 2\pi * f * t)]^{*} = X^{(}f) * exp(-i * 2\pi * f * t)$

$x^{@}(t) = \int_{\_} [X^{@}(-f') exp(-i * 2\pi * f'^{@}\ t)\ ]^{*}\ df'$

$x^{@}(t) = \int_{\_} [X^{@}(-f') exp(i * 2\pi * f'^{@}\ t)\ ]\ df'$