Data Structure Description and Formal Specification

Project Group

Introduction

This document describes the data structure chosen for representing a maze in our project, along with its formal specification. The maze is represented using a two-dimensional matrix of integers, where each integer encodes the presence of walls and other properties of a cell using bit manipulation.

Data Structure Description

Maze Representation

- **Maze Matrix**: The maze is represented as a matrix `maze`$[h][w]$, where h is the height and w is the width.
- **Cell Representation**: Each cell is an integer using bits to represent walls and properties.

Bit Encoding for Cell Properties

Each cell uses bits to represent the presence of walls and special properties:

- **Bit 0 (Value 1)**: Wall on the **North** side.
- **Bit 1 (Value 2)**: Wall on the **East** side.
- **Bit 2 (Value 4)**: Wall on the **South** side.
- **Bit 3 (Value 8)**: Wall on the **West** side.
- **Bit 4 (Value 16)**: **Entrance** flag.

Example

A cell with walls on the North and East sides and marked as an entrance has a value:

$$\text{Cell Value} = 1 \text{ (North wall)} + 2 \text{ (East wall)} + 16 \text{ (Entrance)} = 19$$

Formal Specification

Data Type

```
typedef struct {
    int **cells;   // 2D array of cell integers
    int height;    // Number of rows
    int width;     // Number of columns
} Maze;
```

Function Specifications

Function: `Maze* init(int h, int w)`

Description: Initializes a maze of height h and width w.
Preconditions:

- $h > 0$, $w > 0$.

Postconditions:

- Returns a pointer to a `Maze` with all walls set.
- No entrance is set.

Function: `void destroy(Maze* maze)`

Description: Deallocates the maze.
Preconditions:

- `maze` is not `NULL`.

Postconditions:

- Memory associated with `maze` is freed.

Function: `void open_wall(Maze* maze, int x, int y, int dir)`

Description: Removes the wall in direction `dir` at cell (x, y).
Preconditions:

- $0 \leq x <$ `maze->width`.
- $0 \leq y <$ `maze->height`.
- `dir` is one of `NORTH`, `EAST`, `SOUTH`, `WEST`.

Postconditions:

- The wall in direction `dir` at (x, y) is removed.
- The corresponding wall in the adjacent cell is also removed to maintain symmetry.

Function: `void set_entrance(Maze* maze, int x, int y)`

Description: Marks cell (x, y) as an entrance.
Preconditions:

- $0 \leq x <$ `maze->width`.
- $0 \leq y <$ `maze->height`.

Postconditions:

- Entrance flag is set for cell (x, y).

Function: `bool is_open(Maze* maze, int x, int y, int dir)`

Description: Checks if there is no wall in direction `dir` at cell (x, y).
Preconditions:

- Same as for `open_wall`.

Postconditions:

- Returns `true` if the wall is open; `false` otherwise.

Bit Manipulation Operations

- **Set a Bit** (e.g., mark wall present):

$$\texttt{cell} \mid= (1 \ll \texttt{bit\_pos});$$

- **Clear a Bit** (e.g., remove wall):

$$\texttt{cell} \; \&{=}{\sim} \; (1 \ll \texttt{bit\_pos});$$

- **Check a Bit**:

$$(\texttt{cell}\&(1 \ll \texttt{bit\_pos})) \neq 0$$

Direction Enumeration

```
typedef enum {
    NORTH = 0,   // Bit position 0
    EAST  = 1,   // Bit position 1
    SOUTH = 2,   // Bit position 2
    WEST  = 3    // Bit position 3
} Direction;
```

Invariants and Constraints

- **Symmetry**: If a wall is open between two cells, it must be open in both directions.
- **Boundary Conditions**: Edge cells have walls where there are no adjacent cells.
- **Entrance and Exit**:
  - At least one entrance must be set.
  - Exit is conventionally at cell $(\texttt{width} - 1, \texttt{height} - 1)$.

Conclusion

The chosen data structure efficiently represents the maze using bit manipulation within integers, allowing for quick access and modification of cell properties. The formal specification ensures clarity in implementation and consistency across operations.