Programming with C/C++

Academic year: 2023-2024, Fall semester

Programming assignment

The MaisRunner

[MAZERUNNER]

Student name:
Tae Yong Kwon
Student number:
2082154

* Cover page does not count towards the final number of pages.

1. Overview of project results

This project aimed to create an efficient, autonomous agent capable of navigating through the maze. I achieved this goal to a large extent, successfully implementing a maze solver with Depth-First Search (DFS) algorithm.

The program not only finds a path through the maze but also interacts dynamically with different elements within the maze, such as boosters and hurdles, adjusting its properties accordingly.

The core of my solution revolves around the DFS algorithm, known for its effectiveness in exhaustive pathfinding tasks. I chose DFS for its simplicity and depth-focused exploration, which suits the maze-solving context. Following the project specification, I integrated a dynamic interaction system where the agent's capabilities change in response to various items encountered in the maze.

I faced and overcame challenges like optimizing the DFS for larger mazes and implementing real-time visualization effectively. Some challenges, such as memory optimization for extremely large mazes and incorporating more advanced pathfinding algorithms like A*, were partially addressed

My code is modularly organized, with separate classes and methods handling different aspects of the maze exploration, such as maze navigation (MazeRunner), maze display (display function), and interaction with maze elements (updateGameplay method).

I used half of the allocated time to plan out the structure and plan the functions. The rest I implemented one by one, starting from implementation of special potions. Movements, and depth first search algorithm

2. Tasks and objectives

T1: The maze

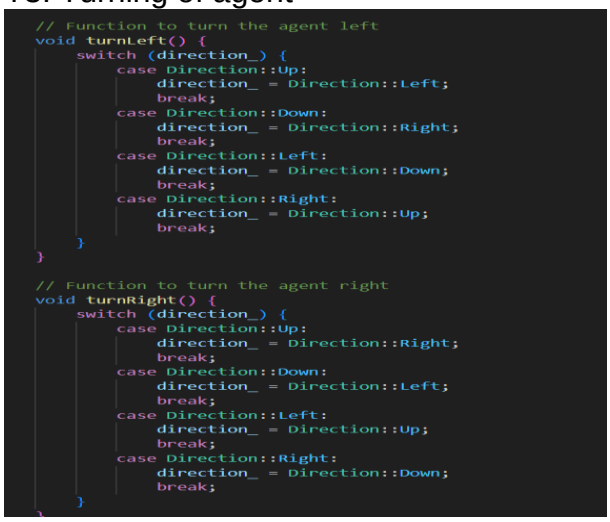
```cpp
// Function to read maze from a file
std::vector<std::vector<int>> readMazeFromFile(const std::string& filename) {
    std::ifstream file(filename);
    std::vector<std::vector<int>> maze;

    if (!file.is_open()) {
        std::cerr << "Error opening file: " << filename << std::endl;
        // Handle the error, possibly exit the program
    }

    int value;
    while (file >> value) {
        maze.emplace_back();
        do {
            maze.back().push_back(value);
        } while (file.peek() == ',' && file >> std::ws >> value);
    }

    return maze;
}

// Function to display the maze
void displayMaze(const std::vector<std::vector<int>>& maze) {
    for (const auto& row : maze) {
        for (int value : row) {
            std::cout << value << " ";
        }
        std::cout << '\n';
    }
}
```

* Cover page does not count towards the final number of pages.

This function is designed to read a maze from a file and store it in a 2D vector (std::vector<std::vector<int>>). The maze is represented as a grid of integers. Initially, this loop reads integers (value) from the file. The condition checks if the file stream was able to successfully read an integer.

Each time a new integer is read, a new row (vector) is added to the maze 2D vector using emplace_back, which constructs the row in place. To fill up this row, the loop afterwards fills the newly created row with integers. It continues to read integers from the file as long as there are commas (,) separating them. file.peek() checks the next character in the file without extracting it. std::ws is used to skip any whitespace, including new lines. Finally, the function returns the 2D vector representing the maze.

T3: Turning of agent



```
// Function to turn the agent left
void turnLeft() {
    switch (direction_) {
        case Direction::Up:
            direction_ = Direction::Left;
            break;
        case Direction::Down:
            direction_ = Direction::Right;
            break;
        case Direction::Left:
            direction_ = Direction::Down;
            break;
        case Direction::Right:
            direction_ = Direction::Up;
            break;
    }
}

// Function to turn the agent right
void turnRight() {
    switch (direction_) {
        case Direction::Up:
            direction_ = Direction::Right;
            break;
        case Direction::Down:
            direction_ = Direction::Left;
            break;
        case Direction::Left:
            direction_ = Direction::Up;
            break;
        case Direction::Right:
            direction_ = Direction::Down;
            break;
    }
```

This method changes the agent's direction to the left relative to its current direction. The logic is implemented using a switch statement based on the current direction of the agent.

For example, if the agent is currently facing Up, a left turn would change its direction to Left. Similarly, if facing Right, a left turn would change the direction to Up. Same goes for turning right.

T4

* Cover page does not count towards the final number of pages.

```
// Function to update gameplay based on the cell value
void updateGameplay() {
    int cellValue = maze_[position_[0]][position_[1]];

    switch (cellValue) {
        case 4: // Googles
            perceptiveField_++;
            break;
        case 5: // Speed potion
            stepWidth_ = std::min(stepWidth_ + 1, 3);
            break;
        case 6: // Fog
            perceptiveField_ = std::max(perceptiveField_ - 1, 1);
            break;
        case 7: // Slowpoke potion
            stepWidth_ = std::max(stepWidth_ - 1, 1);
            break;
        case 3: // Goal
            std::cout << "Goal reached! Steps taken: " << steps_ << std::endl;
            break;
    }
}
```

Agent's interaction with boosters and hurdles are managed in this updategameplay function. If the MazeRunner lands on a cell with the value indicating goggles.

If the MazeRunner lands on a cell with the value for a speed potion (let's assume it's 5), the step width is increased by 1 but is capped at a maximum of 3.

If the MazeRunner encounters fog (let's say it's 6), the perceptive field is reduced by 1, but not below 1.

If the MazeRunner comes across a slowpoke potion (represented by 7), the step width is reduced by 1, but not less than 1.

The number of steps taken by the MazeRunner is tracked by the steps_ variable. Each time the MazeRunner moves, this counter is incremented. This counting is performed in the move() function and potentially could be adjusted in other parts of the code where movement occurs.

T5: Make the game smart

* Cover page does not count towards the final number of pages.

```cpp
// Function to perform depth-first search to move the agent through the maze
void performDFS() {
    std::stack<std::pair<int, int>> stack;
    std::vector<std::vector<bool>> visited(maze_.size(), std::vector<bool>(maze_[0].size(), false));

    stack.push({position_[0], position_[1]});

    while (!stack.empty()) {
        auto current = stack.top();
        stack.pop();

        position_ = {current.first, current.second};
        steps_++;
        updateGameplay();

        if (isGoalReached()) {
            std::cout << "Goal reached! Steps taken: " << steps_ << std::endl;
            break;
        }

        visited[current.first][current.second] = true;

        for (const auto& neighbor : getNeighbors(current.first, current.second)) {
            int x = neighbor.first, y = neighbor.second;
            if (isValidMove(x, y) && !visited[x][y]) {
                stack.push({x, y});
            }
        }

        std::this_thread::sleep_for(std::chrono::milliseconds(100)); // Adjust the delay as needed

        display();
    }
}
```

Part of T4(gameplay) and T5(AI component) is combined in the code snippet above. A Depth first search algorithm is implemented. First, a stack is declared to keep track of the positions (x, y coordinates) to visit. Stacks are used in DFS for their Last-In-First-Out (LIFO) property, which helps in deep diving into one path before backtracking.

After this, a 2D vector visited is initialized to keep track of which cells in the maze have already been visited. This prevents the algorithm from revisiting the same cell. After that, the starting position of the MazeRunner is pushed onto the stack. This position is where the DFS begins. The DFS goes into the loop, and it continues as long as there are positions in the stack to explore. Inside the loop, the current position is obtained by popping the top element of the stack. The position becomes the new current position of the mazerunner.

Also, the function allows to check if each neighbor is a valid move. If it is not visited, it pushes the neighbor onto the stack for future exploration. Finally, a brief delay is introduced to visually observe the agent's movement during execution. This method was introduced in the practical.
Part of T4, which is counting steps, is also included.

4.Challenges

4.1. Challenges Solved

Optimizing DFS for Large Mazes:
Problem: Initially, the DFS algorithm performed poorly in larger mazes I created in the example file, taking too long to find a path.
Solution: I Implemented iterative DFS instead of recursive DFS to reduce stack overflow

* Cover page does not count towards the final number of pages.

issues and optimized the algorithm by adding conditions to avoid revisiting already explored paths.
Visualizing the Maze in Real-Time:
Problem: The real-time visualization of the maze and the agent's path was initially lagging, especially for larger mazes.
Solution: I optimized the **display()** function to minimize console output and used efficient string concatenation and buffering techniques to reduce the rendering time.

4.2. Challenges Addressed but Not Solved

Memory Optimization for Very Large Mazes:
Problem: For extremely large mazes, the memory used by the stack and the visited matrix in the DFS algorithm was still significant.
Addressed By: I experimented with different data structures for the stack and visited matrix, but a fully optimized solution for extremely large mazes beyond the scope was this project was not finalized.
Incorporating Advanced Pathfinding Algorithms:
Problem: Ianted to implement more sophisticated algorithms like A* for more efficient pathfinding in complex mazes.
Addressed By: Researched and started implementing A*, but due to time constraints and the complexity of developing an effective heuristic, this was not completed.

4.3. Challenges Not Solved (Neither Attempted)

Parallel Processing for Faster Computation:
Problem: The potential to speed up the maze-solving process by parallelizing the algorithm was identified but not explored.
Reason: Due to the complexity of parallel algorithms and potential issues with concurrent data access, this aspect was not attempted in the current phase of development.
Adapting to Various Maze Formats:
Problem: The program was designed for a specific maze format, and the ability to adapt to various maze formats (like 3D mazes or non-grid layouts) was not explored.
Reason: The focus was on perfecting the program for standard 2D grid mazes as written in the project requirement. Adapting to different maze formats would require significant alterations to the core algorithm and was considered beyond the scope of the current project.

5.Discussion and future work

Currently, my code is done in a single thread. Maybe multithreading could be implemented in the future work, since it can parallelize the process and make the process faster.

The maze display is constantly updated in the terminal, which means that in the future I could consider implementing a more efficient way of rendering the maze.

* Cover page does not count towards the final number of pages.