L A B E R I N T O

Luis Cedillo Maldonado - 0250009

Tarea

Desarrollar un programa que simule el comportamiento de un robot en un laberinto. Se creará una matriz de tamaño fijo que simulará un laberinto en el cual un robot será posicionado y este deberá moverse y hasta llegar a una meta. El robot será representado con una “R”, los espacios vacíos (Por los cuales es posible moverse) representado por “0”, las paredes son representadas por una “P” y la meta final por una letra “M”. El mapa será generado de manera estática al iniciar el programa

Solución

Para este problema, se dividió el problema en dos secciones. Una clase para el laberinto y otra clase para el robot, el cual se encargará de resolver el laberinto.

Clase Laberinto

```
class Laberynth
{
private:
    const int width, height;
    const vector<int> DIR_X = {0, 0, -1, 1};
    const vector<int> DIR_Y = {-1, 1, 0, 0};

    int **generate_map()
    {
        int **maze = new int *[width];
        for (int i = 0; i < width; i++)
        {
            maze[i] = new int[height];
            fill(maze[i], maze[i] + height, 0);
        }
        maze[0][0] = 1;

        vector<pair<int, int>> cellsToExplore;
        cellsToExplore.push_back({0, 0});

        default_random_engine rng(random_device{}());

        while (!cellsToExplore.empty())
        {
            int currentX = cellsToExplore.back().first;
            int currentY = cellsToExplore.back().second;
            cellsToExplore.pop_back();

            vector<int> possibleDirections = {0, 1, 2, 3};
            shuffle(possibleDirections.begin(), possibleDirections.end(), rng);

            for (int i : possibleDirections)
            {
                int newX = currentX + 2 * DIR_X[i];
                int newY = currentY + 2 * DIR_Y[i];

                if (newX >= 0 && newX < width && newY >= 0 && newY < height && maze[newX][newY] == 0)
                {
                    cellsToExplore.push_back({newX, newY});
                    maze[newX][newY] = 1;
                    maze[currentX + DIR_X[i]][currentY + DIR_Y[i]] = 1;
                }
            }
        }
```

```
        return maze;
    }

    int **map;

public:
    Laberynth(int w, int h) : width(w), height(h)
    {
        map = generate_map();
    }

    ~Laberynth()
    {
        for (int i = 0; i < width; i++)
        {
            delete[] map[i];
        }
        delete[] map;
    }

    int **get_map() const
    {
        return map;
    }

    int get_width() const
    {
        return width;
    }

    int get_height() const
    {
        return height;
    }

    void print_map()
    {
        cout << "┌";
        for (int j = 0; j < height; ++j)
        {
            cout << "—";
        }
        cout << "┐" << endl;

        for (int i = 0; i < width; ++i)
        {
            cout << "│";
            for (int j = 0; j < height; ++j)
            {
                if (map[i][j] == 1)
                {
                    cout << " ";
                }
                else if (map[i][j] == 2)
                {
                    cout << "\033[1;31m"
                         << "◍"
                         << "\033[0m"; // Red robot
                }
                else
                {
                    cout << "■";
                }
            }
            cout << "│\n";
        }

        cout << "└";
        for (int j = 0; j < height; ++j)
        {
            cout << "—";
        }
        cout << "┘" << endl;
    }
};
```

Propiedades

- `width` y `height`: Estas dos propiedades almacenarán el ancho y el alto del laberinto respectivamente. Están declaradas como constantes y se inicializan en el constructor de la clase. El laberinto se generará con estas dimensiones.
- `DIR_X` y `DIR_Y`: Son vectores constantes que representan las direcciones posibles en el eje X y el eje Y respectivamente. En este caso, se utilizan para moverse en horizontal y vertical dentro del laberinto.
- `map`: Es un puntero a una matriz bidimensional de enteros (`int**`) que almacenará el laberinto generado. Cada celda puede contener uno de los siguientes valores:
  - `0`: Pared, celda no transitable.
  - `1`: Camino, celda transitable.
  - `2`: Posición del robot.

Métodos

- `generate_map()`: Este es un método privado que se encarga de generar el laberinto. Explora nodos y conecta caminos mientras se asegura de no crear bucles en el laberinto. Retorna el laberinto generado como una matriz de enteros.

  1. Comienza con una matriz de nodos, donde inicialmente todas los nodos están marcadas como paredes (valor `0`).
  2. Se selecciona una celda inicial (generalmente en la esquina superior izquierda) y se marca como un camino transitable (valor `1`).
  3. Se crea una pila o una lista de nodos aún no exploradas. La celda inicial se agrega a esta lista.
  4. El bucle principal se ejecuta mientras haya nodos sin explorar en la lista:
     - Se selecciona la última celda en la lista.
     - Se obtienen las direcciones posibles (arriba, abajo, izquierda, derecha) y se mezclan de manera aleatoria.
     - Por cada dirección, se calcula la nueva celda a explorar (dos nodos en la dirección seleccionada).
     - Si la nueva celda está dentro de los límites del laberinto y no se ha explorado (es una pared), se marca como camino (valor `1`), así como la celda intermedia (para crear un camino entre nodos).
     - La nueva celda se agrega a la lista de nodos para explorar.

     Este proceso crea caminos aleatorios a medida que explora los nodos, evitando formar bucles. Cuando una celda no tiene direcciones posibles para explorar, se retrocede a la última celda que tiene direcciones sin explorar.
  5. Una vez que no hay más nodos en la lista de nodos por explorar, el laberinto está completo.

El método `generate_map()` implementa este algoritmo en el código proporcionado. Aquí hay un resumen de los pasos clave dentro del bucle while:

- Se selecciona la última celda en la lista de `cellsToExplore` para explorarla.
- Se mezclan de manera aleatoria las direcciones posibles.
- Por cada dirección, se calcula la nueva celda a explorar.
- Si la nueva celda es válida y no ha sido explorada, se marca como camino y la celda intermedia también se marca como camino.
- La nueva celda se agrega a la lista de nodos para explorar.

Este proceso se repite hasta que no queden más nodos en la lista para explorar, lo que asegura que todas los nodos posibles del laberinto se conecten formando caminos, sin crear bucles.

- `Laberynth(int w, int h)`: El constructor de la clase. Toma el ancho `w` y el alto `h` como argumentos y crea un laberinto utilizando estas dimensiones.
- `~Laberynth()`: El destructor de la clase. Libera la memoria ocupada por la matriz del laberinto.
- `get_map()`, `get_width()`, `get_height()`: Métodos de acceso para obtener el puntero al laberinto, el ancho y el alto respectivamente.
- `print_map()`: Imprime el laberinto en la consola. Utiliza caracteres ASCII para representar las paredes, el camino y la posición del robot. Los nodos que contienen un valor de `1` son caminos transitables, las nodos con `2` representan la posición del robot y las nodos con `0` son paredes.

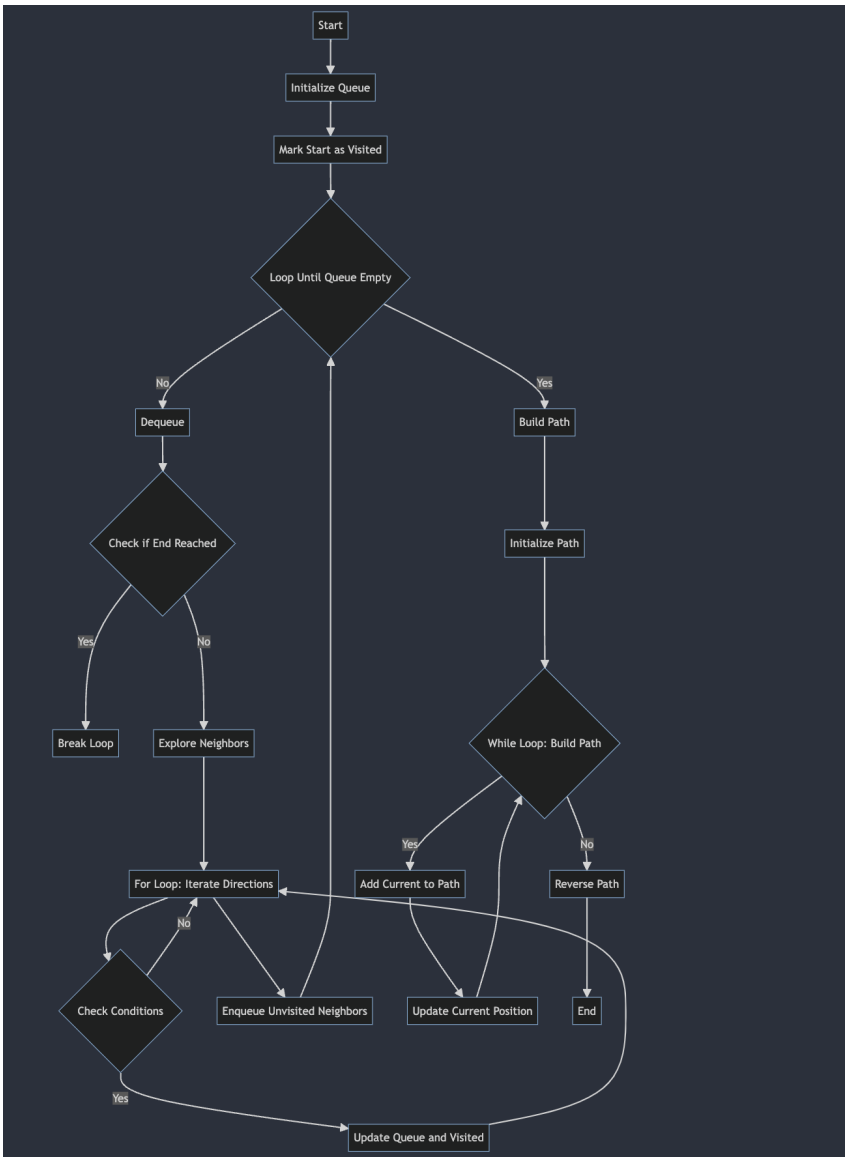
Clase Robot

```
class Robot
{
private:
    static const vector<pair<int, int>> DIRECTIONS;

    static void clearScreen()
    {
#ifdef _WIN32
        system("cls");
#else
        system("clear");
#endif
    }

    void displayMaze(int **maze, int width, int height, int x, int y) const
    {
        clearScreen();
        for (int i = 0; i < width; ++i)
        {
            for (int j = 0; j < height; ++j)
            {
                if (i == x && j == y)
                {
                    cout << "\033[1;31m"
                         << "◍"
                         << "\033[0m"; // Red robot
                }
                else
                {
                    cout << (maze[i][j] == 0 ? "■" : " "); // Wall or space
                }
            }
            cout << '\n';
        }
        this_thread::sleep_for(chrono::milliseconds(25));
    }

public:
    vector<pair<int, int>> solve(int **maze, int width, int height, bool shouldDisplayMaze = false)
    {
        vector<vector<bool>> visited(width, vector<bool>(height, false));

        // visited: mantiene el track de que nodos ya fueron visitados

        vector<vector<pair<int, int>>> previous_position(width, vector<pair<int, int>>(height, {-1, -1}));

        // previous_position: mantiene el track de la posicion anterior de cada nodo

        queue<pair<int, int>> q;

        pair<int, int> start = {0, 0};
        pair<int, int> end = {width - 1, height - 1};
```

```
        q.push(start);
        visited[start.first][start.second] = true;



        while (!q.empty()) // Mientras la cola no este vacia
        {
            pair<int, int> current = q.front();
            int x = current.first;
            int y = current.second;
            q.pop();

            // Si la posicion actual no es igual a la posicion final, se saca un nodo del frente de la pila y se
            // obtienen las coordenadas de la posicion actual

            if (current == end)
                break;

            // Si la posicion en el nodo dentro de la pila es igual a la posicion final, se termina el ciclo

            for (size_t i = 0; i < DIRECTIONS.size(); i++) // Se recorren las direcciones posibles desde la posicion actual
            {
                int newX = x + DIRECTIONS[i].first;
                int newY = y + DIRECTIONS[i].second;
                if (newX >= 0 && newX < width && newY >= 0 && newY < height &&
                    !visited[newX][newY] && maze[newX][newY] == 1)
                {
                    q.push({newX, newY});
                    visited[newX][newY] = true;
                    previous_position[newX][newY] = current;
                }

                // Esta condicional dice lo siguiente:
                // Si la nueva posicion en x es mayor o igual a 0 y menor que el ancho del laberinto
                // y la nueva posicion en y es mayor o igual a 0 y menor que el alto del laberinto
                // y no se ha visitado la nueva posicion y la nueva posicion es un espacio (1) o sea no es una pared (0)
                // entonces se agrega la nueva posicion a la pila, se marca como visitada y se guarda la posicion anterior
            }
            // Uncomment the below line to display maze while solving
            // bool shouldDisplayMaze = true; // Set this to false if you don't want to display the maze

            if (shouldDisplayMaze)
                displayMaze(maze, width, height, x, y);
        }

        vector<pair<int, int>> path; // Almacena un vector de pares de enteros que representan el camino
        pair<int, int> current_position = end;
        while (current_position != make_pair(-1, -1)) // Mientras la posicion actual no sea igual a -1, -1, o sea, mientras no se llegue a
        {
            // Se agrega la posicion actual al vector de camino y se actualiza la posicion actual a la posicion anterior
            path.push_back(current_position);
            current_position = previous_position[current_position.first][current_position.second];
        }
        // Se invierte el vector para que el camino vaya desde la posicion inicial hasta la posicion final
        reverse(path.begin(), path.end());

        // Si el primer elemento del vector es igual a la posicion inicial, se regresa el vector
        return (path[0] == start) ? path : vector<pair<int, int>>{};
    }
};
//                                                  UP      DOWN    LEFT    RIGHT
const vector<pair<int, int>> Robot::DIRECTIONS = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
```

1. `clearScreen()`: Es un método privado estático que se utiliza para borrar la pantalla de la consola. Esto se logra utilizando comandos específicos del sistema operativo, como `system("cls")` en Windows y `system("clear")` en sistemas basados en Unix.
2. `displayMaze()`: Este método privado muestra el laberinto en la consola. Si `shouldDisplayMaze` es `true`, muestra el laberinto con una pequeña pausa entre cada actualización. La posición del robot se resalta en rojo (`●`), las paredes se representan como bloques (`■`), y los caminos transitables se representan como espacios en blanco.

3. `solve()`: Este es el método principal del robot para resolver el laberinto. Toma como entrada la matriz del laberinto, su ancho y alto, y un parámetro booleano opcional `shouldDisplayMaze` que determina si se debe mostrar el proceso de resolución paso a paso.
    - Primero, inicializa una matriz de `visited` para realizar un seguimiento de las celdas que se han visitado durante la búsqueda.
    - También inicializa una matriz de `previous_position` para realizar un seguimiento de la posición anterior de cada celda visitada.
    - Crea una cola `q` que almacenará las celdas por explorar. Agrega la posición de inicio a la cola y la marca como visitada.
    - Utiliza un bucle `while` que se ejecuta mientras la cola `q` no está vacía.
    - En cada iteración, toma el primer elemento de la cola, calcula las nuevas celdas adyacentes, las agrega a la cola si son válidas y actualiza las matrices `visited` y `previous_position`.
    - Además, si `shouldDisplayMaze` es `true`, llama al método `displayMaze()` para mostrar el laberinto en cada iteración.



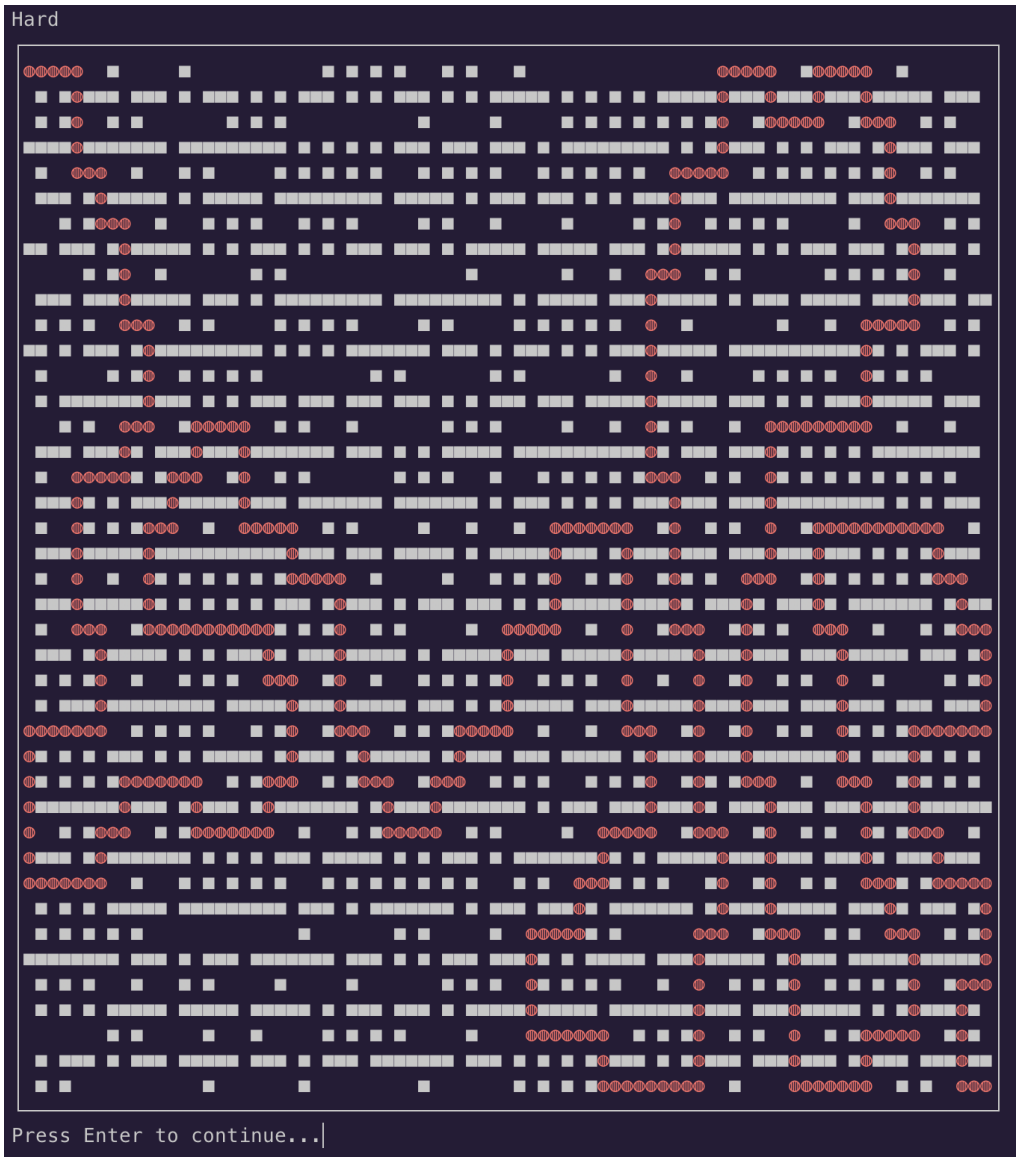
Este es un diagrama que explica a grandes rasgos como funciona el algoritmo de búsqueda.

1. `DIRECTIONS`: Es un vector estático constante de pares de enteros que representa las cuatro direcciones posibles en el laberinto: arriba, abajo, izquierda y derecha.

Walkthrough

💡 Cada uno de las opciones genera un laberinto cuadrado, la diferencia radica en las dimensiones generadas por cada opción. La opción 4 te permite enviar argumentos personalizados además de ver como avanza el robot a través del laberinto

Ejemplo:



El algoritmo muestra la ruta más viable que encontró primero, no obstante, este algoritmo genera varias rutas dependiendo de si colisiona o no.