Navigating the Maze: Exploring Pathfinding Algorithms

This document delves into the intricacies of solving the classic maze problem, exploring the fundamental algorithms of Breadth-First Search (BFS) and backtracking. We will delve into the data structures used to represent the maze, understand the core principles of each algorithm, and examine how they effectively navigate through mazes. Through detailed code walkthroughs, real-life applications, and illustrative examples, this document aims to provide a comprehensive understanding of these pathfinding techniques.

Team members:
Alston Samuel Peter A-2023115006
Agash-2023115009
Ebi Jershika-2023115011

Maze Representation: A 2D Grid Approach

A maze, in its essence, is a complex network of interconnected pathways and walls. To effectively model this structure, we employ a 2D grid representation. Each cell within this grid corresponds to a location in the maze, with values representing its state:

- *0: Represents an open path, allowing movement.*
- *1: Represents a wall, blocking any movement.*

This grid structure provides a simple and efficient way to store the maze's layout, allowing algorithms to process and analyze its pathways.

Data Structures for Maze Navigation

The efficiency and effectiveness of pathfinding algorithms depend on the data structures employed to store and manipulate information about the maze. Here are the crucial data structures used in BFS and backtracking:

- Queue (for BFS): BFS utilizes a queue to process cells in a level-by-level manner, ensuring that the shortest path is found first.

- Parent Array: To reconstruct the path after reaching the destination, a parent array is used. Each cell stores the index of its parent cell, allowing for tracing back to the starting point.

- Linked List (for storing paths): In backtracking, a linked list is used to keep track of each cell in the current path explored.

- Stack (for backtracking): The stack plays a crucial role in backtracking, enabling the program to retrace its steps and try alternate paths when needed.

- Trees(BFS model): The root of the tree is the starting cell. Children of a node represent valid neighboring cells. Branches form as BFS or backtracking explores different paths. Dead ends correspond to leaf nodes, while the destination is another leaf node.

These data structures work in tandem to provide a robust and efficient framework for maze exploration.

Breadth-First Search(BFS): Level-by-Level Exploration

BFS operates on a simple yet powerful principle: explore the maze level by level, starting from the initial point. This process ensures that the shortest path to the destination is found because it processes all cells at a given depth before moving on to the next. This approach, often referred to as a "breadth-first" traversal, guarantees a minimum distance solution.

The algorithm involves enqueuing neighboring cells of the current cell being processed. The algorithm proceeds by dequeuing the first cell in the queue and examining its neighbors. If a neighbor is not yet visited and is not a wall, it is marked as visited and added to the queue. The process continues until the destination cell is reached or the queue is empty, indicating that no path exists.

Backtracking: Exploring All Possibilities

In contrast to BFS, which focuses on finding the shortest path, backtracking explores all possible paths from the starting point to the destination. It follows a "try and backtrack" approach, where the program tries each possible move and then backtracks if it leads to a dead end or a previously visited cell.

Backtracking uses a stack to keep track of the current path. When a cell is visited, it is pushed onto the stack. If the destination is reached, the path stored in the stack represents a successful path. If a dead end or previously visited cell is encountered, the algorithm backtracks by popping cells off the stack and exploring alternative paths. This exhaustive approach ensures that all possible paths are considered, although it can be computationally expensive for large mazes.

Code Walkthrough: Implementing BFS and Backtracking

Let's examine how the core functionalities of BFS and backtracking are implemented in a code snippet. This will provide a concrete understanding of how the algorithms operate on a programmatic level

```
#include <stdio.h>
#include <stdlib.h>

// ---------------- Linked List for Explored Path ----------------
typedef struct Node {
  int x, y;
  struct Node* next;
} Node;

Node* createNode(int x, int y) {
  Node* newNode = (Node*)malloc(sizeof(Node));
  newNode->x = x;
  newNode->y = y;
  newNode->next = NULL;
  return newNode;
}

// ---------------- Stack for Backtracking ----------------
typedef struct Stack {
  int x, y;
  struct Stack* next;
} Stack;

Stack* top = NULL;
```

```
void push(int x, int y) {
  Stack* newNode = (Stack*)malloc(sizeof(Stack));
  newNode->x = x;
  newNode->y = y;
  newNode->next = top;
  top = newNode;
}

void pop(int* x, int* y) {
  if (top == NULL) return;
  Stack* temp = top;
  *x = top->x;
  *y = top->y;
  top = top->next;
  free(temp);
}

// ---------------- Queue for BFS ----------------
typedef struct Point {
  int x, y;
} Point;

typedef struct Queue {
  int front, rear, size;
  int capacity;
  Point* array;
} Queue;

Queue* createQueue(int capacity) {
  Queue* queue = (Queue*)malloc(sizeof(Queue));
  queue->capacity = capacity;
  queue->front = queue->size = 0;
  queue->rear = capacity - 1;
  queue->array = (Point*)malloc(queue->capacity * sizeof(Point));
  return queue;
}
```

```
int isFull(Queue* queue) {
 return (queue->size == queue->capacity);
}

int isEmpty(Queue* queue) {
 return (queue->size == 0);
}

void enqueue(Queue* queue, int x, int y) {
 if (isFull(queue)) return;
 queue->rear = (queue->rear + 1) % queue->capacity;
 queue->array[queue->rear].x = x;
 queue->array[queue->rear].y = y;
 queue->size++;
}

Point dequeue(Queue* queue) {
 Point p = {-1, -1}; // Default value
 if (isEmpty(queue)) return p;
 p = queue->array[queue->front];
 queue->front = (queue->front + 1) % queue->capacity;
 queue->size--;
 return p;
}

// ---------------- Maze and Helper Structures ----------------
int rows, cols; // Maze dimensions
int** maze; // Dynamic maze
int** visited; // Dynamic visited array
int dx[] = {0, 0, -1, 1}; // Directions for movement
int dy[] = {-1, 1, 0, 0};

// ---------------- Function Prototypes ----------------
void displayMaze();
int isSafe(int x, int y);
void bfsSolveMaze();
void findAllPaths(int x, int y, Node* path);
```

```
void resetVisited();
void freePath(Node* path);

// ---------------- Utility Functions ----------------
void displayMaze() {
 printf("Maze:\n");
 for (int i = 0; i < rows; i++) {
 for (int j = 0; j < cols; j++) {
 printf("%d ", maze[i][j]);
 }
 printf("\n");
 }
 printf("\n");
}

int isSafe(int x, int y) {
 return (x >= 0 && x < rows && y >= 0 && y < cols && maze[x][y] == 0 &&
!visited[x][y]);
}

void resetVisited() {
 for (int i = 0; i < rows; i++) {
 for (int j = 0; j < cols; j++) {
 visited[i][j] = 0;
 }
 }
}
void freePath(Node* path) {
  Node* temp;
  while (path) {
    temp = path;
    path = path->next;
    free(temp);
  }
}
```

```
// ---------------- BFS SolveMaze ----------------
void bfsSolveMaze() {
 Queue* queue = createQueue(rows * cols);
 enqueue(queue, 0, 0);
 visited[0][0] = 1;

 // Parent array to reconstruct the path
 Point parent[rows][cols];
 parent[0][0] = (Point){-1, -1}; // Start point has no parent

 while (!isEmpty(queue)) {
 Point p = dequeue(queue);
 int x = p.x;
 int y = p.y;

 // Check if we reached the end
 if (x == rows - 1 && y == cols - 1) {
 // Reconstruct the path
 Node* path = NULL;
 int currX = x, currY = y;

 while (currX != -1 && currY != -1) {
 Node* newNode = createNode(currX, currY);
 newNode->next = path;
 path = newNode;

 Point prev = parent[currX][currY];
 currX = prev.x;
 currY = prev.y;
 }

 // Display the path
 printf("Shortest Path using BFS:\n");
 Node* temp = path;
 while (temp) {
 printf("(%d, %d) -> ", temp->x, temp->y);
 temp = temp->next;
 }
```

```
printf("END\n");

// Free the path
freePath(path);
return;
}

  // Explore neighbors
  for (int i = 0; i < 4; i++) {
    int newX = x + dx[i];
    int newY = y + dy[i];

    if (isSafe(newX, newY)) {
      visited[newX][newY] = 1;
      enqueue(queue, newX, newY);
      parent[newX][newY] = (Point){x, y}; // Set the parent
    }
  }
 }
 printf("No path found using BFS.\n");
}

// ---------------- Find All Paths ----------------
void findAllPaths(int x, int y, Node* path) {
 if (x == rows - 1 && y == cols - 1) {
  // Print the current path
  Node* temp = path;
  printf("Path: ");
  while (temp) {
    printf("(%d, %d) -> ", temp->x, temp->y);
    temp = temp->next;
  }
  printf("END\n");
  return;
 }
```

```
// Explore neighbors
for (int i = 0; i < 4; i++) {
int newX = x + dx[i];
int newY = y + dy[i];

if (isSafe(newX, newY)) {
visited[newX][newY] = 1;

// Add to path
Node* newPathNode = createNode(newX, newY);
newPathNode->next = path;
findAllPaths(newX, newY, newPathNode);
// Backtrack
visited[newX][newY] = 0;
free(newPathNode);
}
}
}

// ---------- Main Function ----------
int main() {
// Input maze dimensions
printf("Enter number of rows in the maze: ");
scanf("%d", &rows);
printf("Enter number of columns in the maze: ");
scanf("%d", &cols);

// Allocate memory for maze and visited arrays
maze = (int*)malloc(rows * sizeof(int));
visited = (int*)malloc(rows * sizeof(int));
for (int i = 0; i < rows; i++) {
maze[i] = (int*)malloc(cols * sizeof(int));
visited[i] = (int*)malloc(cols * sizeof(int));
}

// Input maze configuration
printf("Enter the maze row by row (0 for path, 1 for wall):\n");
```

```
for (int i = 0; i < rows; i++) {
for (int j = 0; j < cols; j++) {
scanf("%d", &maze[i][j]);
}
}

displayMaze();
resetVisited();
printf("Finding all paths:\n");
Node* initialPath = createNode(0, 0); // Start from (0, 0)
visited[0][0] = 1; // Mark the starting point as visited
findAllPaths(0, 0, initialPath);
freePath(initialPath);

resetVisited();
printf("Finding shortest path using BFS:\n");
bfsSolveMaze();

 // Free allocated memory
 for (int i = 0; i < rows; i++) {
   free(maze[i]);
   free(visited[i]);
 }
 free(maze);
 free(visited);

 return 0;
}
```

This code demonstrates the fundamental steps of BFS, including queue management, neighbor exploration, and parent cell tracking. The backtracking implementation (omitted for brevity) utilizes a stack to explore possible paths.

Maze Problem

Maze Representation:
S = Start (0, 0)
E = End (2, 2)
1 = Wall
0 = Path

Maze: S 0 1
0 0 1
1 0 E

Maze as a 2D Array:

maze = [[0, 0, 1], [0, 0, 1], [1, 0, 0]]

Objective: Find a path from **(0, 0)** to **(2, 2)**.

Data Structures in Action

1. Arrays

- **Maze Array:**
  - Represents the maze.
  - Helps check whether a cell is a wall or a path.
  - Example: maze[2][1] == 0 (path), maze[0][2] == 1 (wall).

- **Visited Array:**
  - Prevents revisiting cells.
  - Initially:
  - visited = [[False, False, False], [False, False, False], [False, False, False]]
- Updates when a cell is visited.
- After visiting (0, 0):
- visited = [[True, False, False], [False, False, False], [False, False, False]]

2. Stacks(DFS)

- **Purpose**: Used to explore paths by backtracking when hitting dead ends.
- **Initial State:**
  - Push start (0, 0) onto the stack.
    - Stack: [(0, 0)]
- **Iteration:**
  - **Pop** (0, 0). Explore neighbors: (1, 0) (down) and (0, 1) (right).
    - Stack: [(1, 0), (0, 1)]
  - **Pop** (0, 1). Neighbor (1, 1) is valid.
    - Stack: [(1, 0), (1, 1)]
  - **Pop** (1, 1). Neighbor (2, 1) is valid.
    - Stack: [(1, 0), (2, 1)]
  - **Pop** (2, 1). Neighbor (2, 2) is valid (destination).
    - Stack: [(1, 0), (2, 2)]

- **Path Traversed:** (0, 0) → (0, 1) → (1, 1) → (2, 1) → (2, 2)

3. Queue (BFS)

- **Purpose**: Used for finding the shortest path.
- **Initial State:**
  - Enqueue start (0, 0).
    - Queue: [(0, 0)]
  - Parent Array: parent = { (0, 0): None }
- **Iteration:**
  - **Dequeue** (0, 0). Enqueue neighbors (1, 0) and (0, 1).
    - Queue: [(1, 0), (0, 1)]
    - Parent: { (0, 0): None, (1, 0): (0, 0), (0, 1): (0, 0) }
  - **Dequeue** (1, 0). Enqueue (1, 1).
    - Queue: [(0, 1), (1, 1)]
    - Parent: { (0, 0): None, (1, 0): (0, 0), (0, 1): (0, 0), (1, 1): (1, 0) }

- **Dequeue** (0, 1). Enqueue (2, 1).
  - Queue: [(1, 1), (2, 1)]
  - Parent: { (0, 0): None, (1, 0): (0, 0), (0, 1): (0, 0), (1, 1): (1, 0), (2, 1): (1, 1) }
- **Dequeue** (1, 1). Enqueue (2, 2) (destination).
  - Queue: [(2, 1), (2, 2)]
  - Parent: { (0, 0): None, (1, 0): (0, 0), (0, 1): (0, 0), (1, 1): (1, 0), (2, 1): (1, 1), (2, 2): (2, 1) }

- **Shortest Path:**
  - Backtrack from (2, 2) using the parent array: (2, 2) → (2, 1) → (1, 1) → (1, 0) → (0, 0)
  - Reverse the path: (0, 0) → (1, 0) → (1, 1) → (2, 1) → (2, 2)

4. Linked List

- Purpose: Used in DFS to store paths dynamically.
- Initial State:
  - Start at (0, 0) with a single node.
- Iteration:
  - Add node (0, 0) → (1, 0).
  - Add node (1, 0) → (1, 1).
  - Add node (1, 1) → (2, 1).
  - Add node (2, 1) → (2, 2).

- Path Traversal:
  - Traverse the linked list: (0, 0) → (1, 0) → (1, 1) → (2, 1) → (2, 2)

Summary:

- **DFS (Stack):** Explores paths deeply, but may not always find the shortest one.
- **BFS (Queue):** Guarantees the shortest path by exploring level by level.
- **Linked List (DFS):** Dynamically stores the path as DFS progresse

Working

Enter number of rows in the maze: 5
Enter number of columns in the maze: 5
Enter the maze row by row (0 for path, 1 for wall):
0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0
Maze:
0 1 0 0 0
0 1 0 1 0
0 0 0 1 0
0 1 0 0 0
0 0 0 1 0

Finding all paths:
Path: (4, 4) -> (3, 4) -> (2, 4) -> (1, 4) -> (0, 4) -> (0, 3) -> (0, 2) -> (1, 2) > (2, 2) -> (2, 1) -> (2, 0) -> (1, 0) -> (0, 0) -> END
Path: (4, 4) -> (3, 4) -> (3, 3) -> (3, 2) -> (2, 2) -> (2, 1) -> (2, 0) -> (1, 0) -> (0, 0) -> END
Path: (4, 4) -> (3, 4) -> (3, 3) -> (3, 2) -> (4, 2) -> (4, 1) -> (4, 0) -> (3, 0) -> (2, 0) -> (1, 0) -> (0, 0) -> END
Path: (4, 4) -> (3, 4) -> (2, 4) -> (1, 4) -> (0, 4) -> (0, 3) -> (0, 2) -> (1, 2) -> (2, 2) -> (3, 2) -> (4, 2) -> (4, 1) -> (4, 0) -> (3, 0) -> (2, 0) -> (1, 0) -> (0, 0) -> END

Finding shortest path using BFS: Shortest Path using BFS: (0, 0) -> (1, 0) -> (2, 0) -> (2, 1) -> (2, 2) -> (3, 2) -> (3, 3) -> (3, 4) -> (4, 4) -> END

Real-World Applications of Pathfinding Algorithms

Pathfinding algorithms, like BFS and backtracking, are not just theoretical concepts; they have profound implications in real-world applications across various fields. Let's explore some prominent examples:

- GPS Navigation: When you use a GPS system to navigate from point A to point B, these algorithms are employed to calculate the shortest routes, taking into account obstacles like traffic, roadblocks, and construction zones.
- Robot Navigation: Robots in various industries, from manufacturing to warehouse management, rely on pathfinding algorithms to navigate efficiently through their environments, avoiding obstacles and finding optimal paths to complete their tasks.
- Game Development: In maze games, pathfinding algorithms are crucial for calculating paths for AI characters or enemies to move through complex levels, adding a layer of challenge and dynamism to the gameplay.

These examples highlight the versatility and importance of pathfinding algorithms in our increasingly interconnected world.

Queue and Parent Array: Visualizing the Path

The queue and parent array play critical roles in BFS, enabling the algorithm to efficiently explore the maze and reconstruct the shortest path to the destination. The queue manages the level-by-

level exploration, ensuring that cells at the same depth are processed in order. Once the destination is reached, the parent array allows for backtracking from the destination to the starting point by following the chain of parent cells.

Consider a simple maze with the start point (0, 0) and the end point (2, 2):

0	1	0
0	0	0
0	1	0

Using BFS, the queue would process cells level by level, while the parent array would track the path leading to the destination. For example, after reaching the destination, the parent array might contain information like this (where each value represents the index of the parent cell):

0	-1	1
1	0	3
2	-1	5

By backtracking from the destination (2, 2), using the parent array, we can reconstruct the shortest path: (2, 2) -> (1, 2) -> (0, 2) -> (0, 1) -> (0, 0).

Conclusion: The Power of Path Finding Algorithms

This exploration of pathfinding algorithms, specifically BFS and backtracking, has revealed their powerful capabilities in solving complex navigation problems. BFS, with its level-by-level exploration, guarantees finding the shortest path, making it ideal for applications where efficiency is paramount. Backtracking, on the other hand, provides a comprehensive approach by exploring all possible paths, ensuring that no potential solution is overlooked.

From GPS navigation to robot path planning and game development, pathfinding algorithms play a vital role in our technological landscape. Their versatility and adaptability make them essential tools for solving a wide range of real-world problems. As technology advances, we can expect to see even more innovative applications of these algorithms, further shaping the future of navigation and problem-solving.