Assignment 3 Report

2020CS10375 Rohit Janbandhu
2020CS10317 Aditya Goel

Introduction

Creating and solving random mazes with graph algorithms on distributed memory

a. Sequential Graph Algorithms for Maze Solving

1. Generating the Maze

1.1 BFS

Approach - In the BFS code, we generate the maze using the following approach - First, we make a 64 by 64 grid maze of the character vector. Then, we create a path connecting any starting and endpoints. To generate the path, we can either make the path just by moving down and moving left/right, or we can create a path by moving in all directions. But we find doing down and right/left is much faster.
After generating this path, we use this path to create the maze. We generate the maze by first putting all the path cells in the queue, then pop the queue, and for that cell, we add the valid neighbour cells, which have only one neighbour in the Path. and make the popped cell the path.
We repeat these processes until the queue is empty.

Functions-

1. printMaze(const vector<vector<char>> &maze): - This function is used to print the maze represented by a 2D vector of characters.

2. printMaze(const std::vector<std::string> &maze): Overloaded version of `printMaze` specifically for printing mazes represented as vectors of strings.

3. IS_VALID(int x, int y, int row1, int row2, int cols): This is a helper function used to check whether a given cell (x, y) is within the valid range of the maze and whether it's within the specified rows and columns.

4. hasOnePathNeighbour(vector<vector<char>> &maze, int x, int y, int row1, int row2, int cols): This function checks if a given cell (x, y) has only one path neighbour (adjacent cell with ' ' or 'Q').

5. generateMazeHelper(vector<vector<char>> &maze, int x, int y, int final, int final, int row1, int row2, int cols, vector<int> &directions): This function recursively generates the maze using a modified depth-first search (DFS) algorithm. It randomly shuffles directions to explore neighbours. It backtracks when no valid path is found.

6. extendPaths(vector<vector<char>> &maze, int startX, int startY, int row1, int row2, int cols): This function extends paths in the generated maze using a breadth-first search (BFS) approach. It explores reachable cells from each empty cell in the maze and marks them as part of the maze path.

7. cycle (vector<vector<char>> &maze, int row1, int row2, int cols, int start, int start): This function checks if there's a cycle in the generated maze using a DFS-based approach. It detects if there's a closed loop in the labyrinth.

8. `BFSMazeGenerator::generateMaze(vector<vector<char>>& maze, int startX, int startY, int endX, int endY): This is the primary function responsible for generating the maze. It initialises parameters and calls helper functions to create and extend the labyrinth.

These functions work together to generate a maze represented as a 2D grid of characters, ensuring it's solvable and cycles-free.

1.2 Kruskal

Approach- In this algorithm, we generate the maze in two parts. First, we initialise the maze with *. Then, to apply the Kruskal algorithm, we must make the maze to distinguish two neighbouring Path cells belonging to different sets. We solve this by initialising the maze with alternate Path cells at each row, then alternating at columns. Then, for the edges list, we add all the edges such that it is the initial path cell and its neighbours (valid neighbours - inside the boundary). After getting the edge list, we shuffle and loop through it, making edges in the maze of two neighbouring cells. We maintain a list of parents for each cell, and for each neighbour, we look for their parents. If they are different, we join them and make their parents the same. If it is the same, it is a loop, so we don’t add that edge. We are repeating the step through the edge list. In the end, we get the maze.

Function-

1. initializeMaze(int rows, int cols): This function initialises a maze grid with dimensions rows * cols. It sets every cell in the maze as a wall (*) except for alternate cells in even rows, creating a grid pattern.

2. addEdges(const vector<vector<char>> &maze, int rows, int cols): - This function iterates over the maze grid to identify potential edges between adjacent cells. - It adds edges between neighbouring cells where a passage can be created.

3. find the root (vector<int> &parent, int cell): This function finds the root of the set to which the given cell belongs using path compression. It traverses the parent pointers until it reaches the root of the set.

4. performUnion(vector<int> &parent, int root1, int root2): - This function performs the union operation on two sets represented by their roots. It updates the parent pointer of one root to point to the other root.

5. generateMazeH(vector<vector<char>> &maze, int rows, int cols): This function implements the main logic of Kruskal's algorithm to generate a maze. It shuffles the edges to randomise the order in which they are considered. It iterates over the edges and performs union operations on sets represented by cells connected by the edge. It adds passages between cells if they are not in the same set and do not create cycles. It updates the maze grid accordingly.

6. KruskalMazeGenerator::generateMaze(vector<vector<char>>& maze, int startX, int startY, int endX, int endY): This is the primary function responsible for generating the maze using Kruskal's algorithm. It initialises parameters and calls helper functions to create the maze. It copies the generated maze to the specified portion of the output maze grid.

These functions work together to implement Kruskal's algorithm for maze generation sequentially. They identify edges between cells, perform unions to connect cells and prevent cycles, and update the maze grid to create passages between cells.

2. Solving the Maze

1.1 DFS

Approach- It starts from the given root position (starting point) and explores neighbouring cells recursively until it reaches the endpoint (destination). During the traversal, it marks visited cells to avoid revisiting them and ensures that it does not visit cells that are walls or have already been visited. Once the endpoint is reached, it marks the path from the starting point to the endpoint as 'P' in the maze grid.

Functions-

1. isVALID(std::vector<std::vector<char>>& maze, int x, int y): - Checks if the given cell `(x, y)` is within the bounds of the maze and if it is a valid path cell (' ' or 'E').

2. DFSMazeSolver::solveMaze(std::vector<std::vector<char>>& maze, std::vector<std::vector<char>>& visited, int root, int rootY): Recursively solves the maze starting from the root position (root, rootY). Marks the current cell as visited in the `visited` grid. Check if the current cell is the endpoint ('E'), which marks the path from the root to the endpoint as 'P' and returns. Explores neighbouring cells recursively, marking them as part of the path if they lead to the endpoint. If the current cell is part of the path ('P'), it also marks the root cell as part of the path. This function modifies the maze grid in place to mark the path.

1.2 Djikstra

Approach: Dijkstra's algorithm is a famous graph traversal algorithm used to find the shortest path from a source vertex to all other vertices in a weighted graph. In the context of maze solving, each cell is considered a vertex, and the distance between neighbouring cells is regarded as the weight of the edges between vertices.- The algorithm maintains a priority queue to explore cells with the smallest distance from the starting point. It iteratively explores neighbouring cells, updating their distances if a shorter path is found. Once the endpoint is reached, the algorithm terminates, and the shortest path from the starting point to the endpoint is reconstructed using predecessor information.

Functions:

1.DijkstraMazeSolver::solveMaze(std::vector<std::vector<char>>& maze, std::vector<std::vector<char>>& visited, int root, int rootY): This function implements Dijkstra's algorithm to solve the maze. It initialises distance and predecessor arrays to store the shortest distances from the starting point to each cell and the predecessor cell on the shortest path, respectively. The function iterates through cells in the maze, exploring neighbouring cells and updating their distances if a shorter path is found. Once the endpoint is reached, the function reconstructs the shortest path from the starting point to the endpoint using predecessor information. Finally, it marks the cells on the shortest route with 'P' in the maze grid.

-

2. Parallelisation using MPI

Approach- In the Generation of the maze, we divide the work of generation among the four processes, where each process generates the subparts of the labyrinth, which are then stacked vertically; we also leave a row between these subparts of the maze so that we can handle the case for the cycle. We do this so that there is only one path between the layers.

In solving, we divide the work into four or more subtrees, each handled differently. The subtree that finds the end point successfully prints the maze.

2.1. Synchronisation Primitives and Correctness

We use MPI_Barrier to synchronise the processes upon completion of the maze generation process.

2.2. MPI Blocking v non-blocking calls

We use Blocking MPI calls to transfer data between processes. Tensuressure correctness even though it has some extra time overhead compared to non-blocking calls.

2.3. MPI Reductions

We don’t use any MPI reductions.

2.4. Optimisation for Sparsity of the Maze Graph

Yes, we use the fact that any cell can have at most four neighbours for maze generation and solving.

4. Analytical Speedup and Efficiency Analysis

4.1. Speedup Analysis

- Generation: 64x
- Solving: 4x

4.2. Efficiency Analysis

- Generation: 64x
- Solving: 4x