COL 380 - Assignment 3

Manshi Sagar 2020CS50429
Rajat Mahaur 2020CS50534
Richa Yadav 2020CS50438

March 2024

1 Algorithms

1.1 BFS maze generator

1.1.1 Main Function

1. The main function initializes MPI and retrieves the rank and size of the MPI communicator. It sets up a random number generator using the rank of each process to ensure different random numbers are generated for each process. It declares variables to store the entry and exit columns of the maze for each process.

2. Process with rank 0 generates 3 random integers between 1 and ‘cols - 1‘, where ‘cols‘ is the number of columns in the maze. These random integers represent potential entry and exit columns for the local mazes that each of the process held. We divide the maze row-wise among the different processes. It ensures that different processes work on different parts of the maze, thereby reducing the overall computation time. The random integers are then sent to all other processes using MPI_Send.

3. Processes with rank other than 0 receive the 3 random integers from process 0 using MPI_Recv. Based on the received random integers, each process determines its entry and exit columns.

4. The local maze for each process is initialized as a 2D vector with dimensions ‘localRows‘ x ‘cols‘. Breadth-First Search (BFS) algorithm is used to generate a path from the entry point to the exit point within each process's local maze. The use of BFS ensures that the generated maze has a path from the entry point to the exit point.

5. Local maze data from each process is flattened into a 1D vector ('localMazeData') for MPI_Gather. MPI_Gather is used to gather the local maze data from all processes into a single global maze ('globalMazeData') on process 0.

6. Process 0 reconstructs the global maze from the gathered data and prints it out. MPI_Finalize is called to clean up MPI resources.

1.1.2 BFS

1. Local Variables:

   (a) localrows: Represents the number of rows assigned to each process (assuming this is used in a parallel setting).

   (b) timing: Variable used to control the probability of adding walls to the maze based on some threshold.

   (c) ranklocal: Rank of the MPI process.

2. BFS Algorithm:

   (a) The function starts by initializing a queue (q) with the entry point of the maze inserted in the queue.

(b) While the queue is not empty, the function continues to explore the maze. In each iteration, it dequeues a cell from the queue (current cell). It then explores the neighbouring cells (of current cell) in random order (to introduce randomness in the maze generation).

(c) For each neighbouring cell, it checks whether the neighbour cell can be opened(made a non-wall cell). If all(3) adjacent cells of this neighbouring cell (except the current cell) are walls, then with some probability (0.2), we open this neighbouring cell (make it non-wall cell). And add it to queue.

(d) MPI_Barrier are used after local maze generation so that global maze gathering happens after each of the local mazes are generated.

3. Probability Control: There is a section of code (wanttoincludeprob == 1) that introduces probability control to the maze generation process. If enabled, it randomly decides whether to add a wall to the maze for open walk based on a probability threshold.

1.2 Kruskal Maze Generation

The Main Function for this is also same as the previous BFS Code, where we divide the maze in row-wise manner among the processes. MPI_Barrier are used after local maze generation from kruskal algorithm so that global maze gathering happens after each of the local mazes are generated.

1.2.1 Kruskal

1. The DisjointSet class is implemented to maintain the disjoint sets for efficient union-find operations. It uses the union by rank and path compression techniques. The parent vector holds the parent node of each cell in the maze, while the rank vector maintains the rank of each set. The maze follows that if the two cells are connected by any path in the maze itself then they will be in same set or will have the same parent. So this way we will create a perfect maze.

2. Kruskal function generates a maze using the Kruskal's algorithm, which is a minimum spanning tree algorithm.

3. It starts by creating a list of all possible walls in the maze. Walls are represented as pairs of adjacent cells. They can be vertical or horizontal walls.

4. The algorithm then shuffles the list of walls to introduce randomness in maze generation. It initializes an instance of the DisjointSet class to keep track of the sets of cells and their parent relationships. Next, it iterates through the shuffled list of walls. For each wall:

5. If the cells separated by the wall belong to different sets (i.e., they are not in the same connected component), the wall is removed, and the cells are connected.

6. This connection is done by marking the wall as part of the maze and merging the sets of the two cells using the merge function of the DisjointSet class. The merging process effectively connects the two cells and removes the wall between them.

7. The process continues until all walls have been processed.

8. Handling Horizontal and Vertical Walls, The code distinguishes between horizontal and vertical walls to ensure proper maze generation. For horizontal walls, it checks if the cells above and below the wall belong to different sets. If so, it connects them and adds the wall to the maze. Similarly, for vertical walls, it checks if the cells on the left and right sides of the wall belong to different sets before connecting them.

1.3 DFS Maze Solving

1.3.1 Main Function

1. Each of the processes solves the maze in their area. The maze is divided in a row-wise fashion and each of the processes solves the part in it. We define the exits and entries of p process to $p + 1$ process by the iteration from the last row of one process to first row of next process. This way we can create the separate entry and exit points of each of the processes.

2. After solving the maze locally on each process, we gather the local maze data to the root process using MPI_Gather. However, we ensure that the buffer sizes and offsets are correctly set to avoid segmentation faults or incorrect data gathering.
3. After gathering the maze data on the root process, we reconstruct the global maze to display it.

1.3.2 Solving Maze by DFS

1. It contains a public method solveMaze which takes a 2D vector maze representing the maze and returns a vector of pairs representing the path from the start 'S' to the end 'E'. It also contains a private method dfs which is the depth-first search algorithm used to traverse the maze recursively. Another private method isValid checks whether a given cell is valid for movement (not a wall '#', not already visited 'X', and within the bounds of the maze).
2. If the starting point is found, it initializes an empty path vector and calls the dfs method to find the path from the starting point to the end point 'E'. If a solution is found, it returns the path, otherwise, it returns an empty path in form of cell coordinates.
3. It takes the current position (x, y) in the maze and recursively explores all possible directions. It first checks if the current position is the end point 'E'. If so, it adds the current position to the path and returns true. It marks the current cell as visited by changing its value to 'X'. It defines the four possible movements (up, down, left, right) using arrays dx and dy. It iterates over each possible movement and recursively calls itself for each valid movement. If a valid movement leads to the end point, it returns true. Otherwise, it backtracks by removing the last cell from the path and continues exploring other directions. If no solution is found from the current position, it returns false.

1.4 Dijkstra Maze Solving

- Input = matrix of size 64 x 64 (some cells are wall cells and some are non-wall cells)
- We convert this input matrix into a weighted adjacency matrix (size pxp) of non-wall cells (non-reachable cells have infinite distance and reachable cells have 1 distance between them)
- To convert the input matrix to new adjacency matrix, each non-wall cell is made a new vertex. The mapping from (I, J) in input matrix to new vertex id is stored in a map.
- Then we add padding in this adjacency matrix so that we can divide it equally into 4 sub-matrices (column partition)
- Each processor runs Dijkstra on its part of the matrix. Each process calculates the minimum distance from the global source vertex to their assigned vertices.
- When one process finds a new global minimum distance, it updates all processes by broadcasting this information.
- The process finding the global minimum vertex marks it as visited to avoid revisiting it.
- Each process updates its local distance to its assigned vertices based on the new global minimum distance.
- Repeat the process for n - 1 rounds, marking one vertex in each round, ensuring termination.
- After completion, the minimum distance from the source vertex to all other vertices is computed.
- We look at the path between source and destination, and use the mapping of (I,J) to vertex id to reconstruct the input matrix with the required path.

2 References

- https://blog.stackademic.com/solving-mazes-in-java-depth-first-search-dfs-approach-fb86ec585772
- https://github.com/Lehmannhen/MPI-Dijkstra
- https://www.geeksforgeeks.org/find-whether-path-two-cells-matrix/