Data Structures and Objects
CSIS 3700

Fall Semester 2024 — CRN 41145

Project 3 — Pathfinder BFS
Due date: Tuesday, November 12, 2024

Goal

Create a program that generates a random maze, finds a solution and generates a drawing of both.

Important Note

You may work in pairs. That means two, not more than two. Two shall be the number thou shalt count, and the number of the counting shall be two. Three thou shalt not count. Four is right out.

If you wish to work in pairs, *both* of you must email me prior to starting work on the project. Failure to do so runs the risk of getting a zero grade on the project.

Details

Please note that you don't need to create the code for drawing. I will provide the code and header file for that. All you'll need to do is call the drawing function.

The program should read three numbers r c w from the command line, where $1 \leq r \leq 50$ is the number of rows in the maze, $1 \leq c \leq 50$ is the number of columns in the maze and $0 \leq w$ is the number of extra walls to remove. The program will remove $r \cdot c - 1$ walls, then remove w additional walls. There is no other input to the program.

So, for example, running your program with `$ ./project3 10 20 15` will generate a maze with `10` rows and `20` columns and remove `15` additional walls beyond the initial `199` walls needed to connect all of the cells.

Your program should be able to handle a maze with a maximum of `50` rows and `50` columns.

▹*Generating a maze*

Generating a random maze is actually not that difficult. Pick an interior wall at random and remove it, as long as removal doesn't cause a loop in the maze. Repeat until removing any interior wall creates a loop.

Theoretically, what you're doing is taking individual vertices (the cells in the maze) in a graph and connecting them (by removing walls) into a tree (connected with no loops is equivalent to having exactly one path between any two vertices).

How many times must we repeat? Tree theory tells us that a tree with v vertices has exactly $v-1$ edges. Each edge is a removed wall. If there are r rows and c columns, then there are $r \cdot c$ vertices, and we must remove $r \cdot c - 1$ walls.

How do you pick walls at random? Sampling without replacement. We did that in a previous lab.

How do you tell if you have a loop? Disjoint set (union-find) structure. We also did that in a previous lab.

▹*Sampling without replacement*

This is the `Sampler` class from the dynamic memory lab. Each element in the `Sampler` object represents a wall. To set up a `Sampler` for wall selection, we need to answer two questions:

- How many walls are there?
- How do we associate a single number (which the `Sampler` gives us) with a specific wall?

Counting walls

In this project, we are only interested in interior walls; we do not want to remove the walls along the outside edge of the maze.

There are two types of walls — vertical and horizontal. Between two adjacent columns there is one vertical wall per row, or r vertical walls per column pair. There are $c-1$ adjacent column pairs, which results in $r \cdot (c-1)$ vertical walls.

Similarly, between adjacent rows there is one horizontal wall per column, for a total of c horizontal walls per row pair. There are $r-1$ adjacent row pairs, giving a total of $c \cdot (r-1)$ horizontal walls.

Converting random samples to wall direction and location

It is possible to create a single `Sampler` object to represent all of the walls. Simply create a `Sampler` to hold all interior walls and choose walls randomly. If the number is sufficiently large, it is a horizontal wall. If not, it is a vertical wall.

Given a random value s representing a vertical wall, the row can be found using $row = \lfloor s/(c-1) \rfloor$ (note: this is all integer arithmetic, you get the floor for free) and the column can be found using $col = s \bmod (c-1)$. For horizontal walls, use $row = \lfloor (s-|v|)/c \rfloor$ and $col = (s-|v|) \bmod c$ where $|v|$ is the total number of vertical walls.

When you randomly select a vertical wall at location (row, col), you are considering removing the wall between cells (row, col) and $(row, col+1)$. For a random horizontal wall, you are considering the removal of a wall between locations (row, col) and $(row+1, col)$.

▹*Disjoint sets*

Disjoint sets are a very cool structure, and extremely easy to implement; this was also done in the dynamic memory lab. A disjoint set structure *partitions* a set of elements — every element belongs to one partition, and the intersection of any two partitions is an empty set (*i.e.*, they are disjoint).

Initially, every element is in its own singleton partition. Over time, partitions are merged — the *union* operation. During the structure's lifetime, queries are made to identify if two elements are in the same partition — the *find* operation.

The find operation has the property that within one partition, all elements will give the same answer to a query. Note that in the lab example, as the program progressed, when two groups were joined, they all pointed to the same value afterward.

For this project, we will need one disjoint set structure. The elements in the structure represent cells in the maze. If two cells are in the same partition, then there is currently a path without walls between the two cells. We don't know what the path is, but that is not important for the disjoint set's usage. All that is important is that we know that a path exists. If two cells are in different partitions, then there is currently no path between them.

Side note: This is one of the main (and one of the few!) uses of a disjoint set structure.

The disjoint set will have $r \cdot c$ elements in it, one for each cell in the maze. For unions, a and b are adjacent cells and we would be considering removing the wall between them. If removing the wall does not create a loop, then $\textsc{Find}(a) \neq \textsc{Find}(b)$. If removing the wall creates a loop, then the two finds would be equal.

One very important note: Each interior wall appears twice, once for each cell. When you remove a wall, make sure you remove both copies of it.

▹*Generating the maze*

The items discussed in the previous subsections provide the necessary tools to generate a maze. The maze itself is a two-dimensional array of characters; in other languages, a single-byte integer would be used.

Creating the maze consists of two parts. The first part removes just enough walls to connect all cells into a tree. The second part removes additional walls to provide loops within the maze.

Algorithm 1 Generate a tree maze

Preconditions None

Postconditions *maze* contains a single-entry, single-exit maze with no loops

```
1:  procedure GenerateTreeMaze(r, c)
2:      for row ← 0 to r − 1 do                          ▷ Every cell gets four walls
3:          for col ← 0 to c − 1 do
4:              maze[row][col] ← 15          ▷ 15 = (1111)_2 represents all four walls around the cell
5:          end for
6:      end for

7:      Initialize disjoint set object ds with r · c elements

8:      nVWalls ← r · (c − 1)
9:      nHWalls ← (r − 1) · c
10:     Initialize sampler object walls with nVWalls + nHWalls elements

11:     for i ← 0 to r · c − 1 do
12:         do
13:             e ←walls.GetSample()                     ▷ Select random wall
14:             if e < nVWalls then                      ▷ Removing a vertical wall
15:                 r_1 ← ⌊e/(c − 1)⌋                    ▷ Find the two cells on either side of wall
16:                 c_1 ← e mod (c − 1)
17:                 Set r_2, c_2 to cell to the right of r_1, c_1

18:                 cell_1 ← r_1 · c + c_1               ▷ Convert locations to single number
19:                 cell_2 ← r_2 · c + c_2

20:                 wall_1 ← WALL_RIGHT                  ▷ Remember which walls are being removed
21:                 wall_2 ← WALL_LEFT
22:             else                                     ▷ Removing a horizontal wall
23:                 e ← e − nVWalls                      ▷ Skip over the vertical walls
24:                 r_1 ← ⌊e/c⌋                          ▷ Find the two cells on either side of wall
25:                 c_1 ← e mod c
26:                 Set r_2, c_2 to cell below r_1, c_1

27:                 cell_1 ← r_1 · c + c_1               ▷ Convert locations to single number
28:                 cell_2 ← r_2 · c + c_2

29:                 wall_1 ← WALL_DOWN                   ▷ Remember which walls are being removed
30:                 wall_2 ← WALL_UP
31:             end if
32:         while Find(cell_1) =Find(cell_2)             ▷ Continue until cells aren't already connected

33:         Union(cell_1, cell_2)                        ▷ Connect cells

34:         Remove wall wall_1 from cell at r_1, c_1
35:         Remove wall wall_2 from cell at r_2, c_2
36:     end for
37: end procedure
```

Algorithm 2 Removing additional walls

Preconditions *maze* contains a single-entry, single-exit maze with no loops

Postconditions *maze* contains a single-entry, single-exit maze that has loops unless $w = 0$

```
1:  procedure RemoveAdditionalWalls(r, c, w)
2:      nVWalls ← r · (c − 1)
3:      nHWalls ← (r − 1) · c
4:      Initialize sampler object walls with nVWalls + nHWalls elements

5:      for i ← 0 to w − 1 do
6:          haveWall ← false
7:          while not haveWall do
8:              e ←walls.getSample()                          ▷ Select random wall
9:              if e < nVWalls then                           ▷ Removing a vertical wall
10:                 r1 ← ⌊e/(c − 1)⌋                          ▷ Find the two cells on either side of wall
11:                 c1 ← e mod (c − 1)
12:                 Set r2, c2 to cell to the right of r1, c1

13:                 wall1 ← WALL_RIGHT                        ▷ Remember which walls are being removed
14:                 wall2 ← WALL_LEFT
15:             else                                          ▷ Removing a horizontal wall
16:                 e ← e − nVWalls                           ▷ Skip over the vertical walls
17:                 r1 ← ⌊e/c⌋                                ▷ Find the two cells on either side of wall
18:                 c1 ← e mod c
19:                 Set r2, c2 to cell below r1, c1

20:                 wall1 ← WALL_DOWN                         ▷ Remember which walls are being removed
21:                 wall2 ← WALL_UP
22:             end if

23:             if wall1 exists in cell r1, c1 then
24:                 haveWall ← true
25:             end if
26:         end while

27:         Remove wall wall1 from cell at r1, c1
28:         Remove wall wall2 from cell at r2, c2
29:     end for
30: end procedure
```

▹*Solving the maze*

There are two common methods for finding a path through a maze. One is to use a basic backtracking algorithm. The other is to use a *breadth-first search*. We will use this latter method.

To perform a BFS, begin with marking each cell as being "unnumbered." Number the ending cell 0 and add it to a queue. Then, while the queue is not empty, pull a cell from the queue, look at its neighbors and add any unnumbered neighbors to the queue after giving them a number. Since the maze is connected, eventually all cells will be numbered.

After the queue empties, the number given to the starting cell is the length of a shortest path from the starting cell to the end. Beginning with the starting cell, look for any neighbor whose number is one less than the current cell and move there. Repeat until you reach the ending cell.

The algorithm to do this follows.

Algorithm 3 Maze solver

Preconditions *maze* is a maze generated by GENERATETREEMAZE and REMOVEADDITIONALWALLS

Postconditions *maze* is marked with a path from $(0, 0)$ to $(r-1, c-1)$

```
1:  procedure FINDPATH(maze, r, c)
2:      for row ← 0 to r − 1 do
3:          for col ← 0 to c − 1 do
4:              count[row][col] ← −1
5:          end for
6:      end for
7:      e ← (r − 1) · c + c − 1
8:      q.enqueue(e)
9:      count[r − 1][c − 1] ← 0

10:     while q not empty do
11:         e ← q.dequeue()
12:         row ← ⌊e/c⌋
13:         col ← e mod c

14:         for each neighbor (row′, col′) of (row, col) do
15:             if no wall between (row, col) and (row′, col′) and count[row′][col′] = −1 then
16:                 e ← row′ · c + col′
17:                 q.enqueue(e)
18:                 count[row′][col′] = count[row][col] + 1
19:             end if
20:         end for
21:     end while

22:     Mark (0, 0) as visited
23:     row ← 0
24:     col ← 0

25:     while count[row][col] ≠ 0 do
26:         for each connected neighbor (row′, col′) of (row, col) do
27:             if count[row′][col′] = count[row][col] − 1 then
28:                 row ← row′
29:                 col ← col′

30:                 break
31:             end if
32:         end for

33:         Mark (row, col) as visited
34:     end while
35: end procedure
```

▸*Putting it all together*

Generate the maze, then solve the maze. Then, call my **`printMaze()`** function; it will generate the file **`maze.ps`** which will consist of the original maze and the maze with the solution path drawn. The file can be viewed with the document viewer or printed.

What to turn in

Turn in your source code and **`Makefile`**. If you are using an IDE, compress the folder containing the project and submit that.