Data Structures and Objects
CSIS 3700

Spring Semester 2018 — CRN 21212

Project 4 — Pathfinder

Due date: Friday, April 27, 2018

Goal

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

Details

The first important detail is that you don't need to come up with 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 the number of rows and columns from the command line, with no other input to the program. So, for example, running your program with **`$ ./project3 10 20`** will generate a maze with 10 rows and 20 columns.

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 n_R rows and n_C columns, then there are $n_R \cdot n_C$ vertices, and we must remove $n_R \cdot n_C - 1$ walls.

How do you pick walls at random? Sampling without replacement. See below.

How do you tell if you have a loop? Disjoint set (union-find) structure. Also see below.

▹*Sampling without replacement*

Sampling without replacement If the universe of items isn't overly large — for our project it is 10 000 walls - there is a simple algorithm. Start by creating an array **`items[]`** containing all items in the universe and initialize a counter n to the item count. Then, to sample, use this algorithm.

Algorithm 1 Sampling without replacement

Preconditions $items$ is an array with n elements

Postconditions e is a random element from $items$, e has been removed from $items$, n is decremented

```
procedure SampleNoReplacement(items,n)
    i ← Rand mod n                          ▷ Select random position in the list
    e ← items[i]                            ▷ Store the selected item
    n ← n − 1                               ▷ Decrement n
    items[i] ← items[n]                     ▷ Move last item into selected position

    return e
end procedure
```

The algorithm selects one item at "random," then takes the last item in the list and moves it into the position vacated by the selected item. This keeps the remaining items in a (smaller) contiguous list. This is very efficient; each selection takes $O(1)$ time.

What does the universe consist of? Each cell/wall combination is one element.

Suggestion: encode the cell (r, c) and direction d as $e = r \cdot n_C + c + 4096 \cdot d$. Given e, it is easy to extract r, c and d.

Use $d \in \{0, 1, 2, 3\}$ to encode up, down, left and right. Consider using an enumeration or set of named constants.

Create an array of size $10\,000$, the largest number of combinations possible in this project. Fill $4 \cdot n_R \cdot n_C$ of them with the combinations.

▹*Disjoint sets*

Disjoint sets are a very cool structure, and extremely easy to implement. Start by making two arrays $elements[\,]$ and $rank[\,]$, with $elements[i] = i$ and $rank[i] = 0$ for each i. We will need one position for each cell in our maze, so the arrays can be declared with size 2500. It's a one-dimensional array, even though the maze has two dimensions.

A disjoint set only supports two operations: a *union* operation that joins two disjoint sets into one, and *find* which picks one element from a disjoint set. As long as a particular set does not change, the *find* operation always returns the same element.

The algorithms for union and find are given below.

Algorithm 2 Disjoint set union

Preconditions *elements* and *rank* are a disjoint set with elements a and b

Postconditions The disjoint sets containing a and b have been combined into one set

```
procedure DisjointSetUnion(elements,rank,a,b)
    a ← DisjointSetFind(a)                         ▷ Get representatives for a and b
    b ← DisjointSetFind(b)

    if a ≠ b then                                  ▷ Only union if a and b are in different sets
        if rank[a] < rank[b] then                  ▷ Set with lower rank merged into set with larger rank
            elements[a] = b
        else if rank[a] > rank[b] then
            elements[b] = a
        else                                       ▷ In case of tie, increment one set's rank
            rank[a] ← rank[a] + 1
            elements[b] = a
        end if
    end if
end procedure
```

Algorithm 3 Disjoint set find

Preconditions *elements* and *rank* are a disjoint set with element a

Postconditions Returns the representative for the set containing a

```
procedure DisjointSetFind(elements,rank,a)
    if elements[a] = a then                                    ▷ Return a if it is root of intree
        return a
    else
        elements[a] ← DisjointSetFind(elements[a])             ▷ Connect a directly to top of intree
        return elements[a]                                     ▷ Return top of intree
    end if
end procedure
```

Suggestion: Make the disjoint set structure a class. Have the constructor take a number indicating how many elements there are in the universe.

The disjoint set will have $n_R \cdot n_C$ 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 $\textsc{DisjointSetFind}(a) \neq \textsc{DisjointSetFind}(b)$. 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. The algorithm to create the maze follows.

Algorithm 4 Generate a maze

Preconditions None

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

```
procedure GenerateMaze(n_R, n_C)
    i ← 0
    for r ← 0 to n_R − 1 do
        for c ← 0 to n_C − 1 do
            maze[encode(r, c, 0)] ← 15
            for d ← 0 to 3 do
                elements[i] ← encode(r, c, d)
                i ← i + 1
            end for
        end for
    end for
    Initialize disjoint set structure with n_R · n_C elements

    i ← 0
    n ← 4 · n_R · n_C
    while i < n_R · n_C − 1 do
        do
            do
                e ← SampleNoReplacement(elements, n)
            while e references an exterior or nonexistent wall
            r_1 ← decodeCell(e)
            Set r_2 to room adjacent to r_1 in given direction
        while DisjointSetFind(r_1) = DisjointSetFind(r_2)

        DisjointSetUnion(r_1, r_2)
        i ← i − 1

        Remove wall between r_1 and r_2
    end while
end procedure
```

▹*Solving the maze*

Finding a path through a maze is a basic backtracking algorithm. The general idea is to follow some path into the maze, remembering the choices made along the way. If a dead end is reached, go back to the previous decision point and make a different decision. Eventually, the end will be reached if a path exists.

The algorithm is given below.

Algorithm 5 Maze solver

Preconditions *maze* is a maze generated by GenerateMaze

Postconditions *maze* is marked with a path from $(0,0)$ to (n_R, n_C) and dead ends are also marked

```
procedure FindPath(maze)
    S.push(encode(0, 0, 0))
    Mark (0, 0) as visited
    while true do
        (r, c, d) ← decode(S.peek())
        if r = n_R − 1 and c = nC − 1 then
            break
        end if

        if d = 4 then
            Mark (r, c) as a dead end
            S.pop()
        else
            Let (r′, c′) be the next cell in direction d
            Replace encode(r, c, d) with encode(r, c, d + 1) on top of stack
            if no wall exists in direction d and (r′, c′) is not marked as visited then
                S.push(encode(r′, c′, 0))
                Mark (r′, c′) as visited
            end if
        end if
    end while
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, the maze with the solution path drawn, and the maze with the solution path and dead ends that were encountered. 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.