Data Structures and Objects
CSIS 3700

Spring Semester 2020 — CRN 21212 / 21213

Project 3 — Hex Pathfinder
Due date: Wednesday, May 6, 2020

Goal

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

Details

A couple of important points. First, 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 second important detail is that you may work in pairs on the project. Both students must email me indicating their choice of partner.

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*

If the universe of items isn't overly large — for our project it is 15 000 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)
    $i \leftarrow$ Rand mod n ▹ Select random position in the list
    $e \leftarrow items[i]$ ▹ Remember the selected item
    $n \leftarrow n - 1$ ▹ Decrement n
    $items[i] \leftarrow items[n]$ ▹ 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 wall in the maze is one element.

Suggestion: encode the cell (r, c) and direction $0 \le d < 6$ as $e = c + n_C \cdot (r + n_R \cdot d)$. Given e, it is easy to extract r, c and d.

Use $d \in \{0, 1, 2, 3, 4, 5\}$ to encode up, up-right, down-right, down, down-left and up-left *in that order*. That order matches the constants used in my code.

Create a **`Sampler`** class with the following methods:

- **`Sampler(int n)`**
  Creates an array of **`n`** integers. Set **`array[i]=i`** for all slots. Store **`n`** in a class variable.
- **`~Sampler()`**
  Deletes the array.
- **`int getSample()`**
  Returns one element from the array using Algorithm 1.

▹*Disjoint sets*

Disjoint sets are a very cool structure, and extremely easy to implement. 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 *find* operation also gives the same answer for any element in the same set.

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               ▷ In case of tie, increment one set's rank
                rank[a] ← rank[a] + 1
            end if
            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                         ▷ Connect a directly to top of intree
        elements[a] ← DisjointSetFind(elements[a])
    end if
    return elements[a]                              ▷ Return top of intree
end procedure
```

Make a `DisjointSet` class. It should have pointers to integers for both arrays. It should have the following methods:

- `DisjointSet(n)`
  Create the `elements` and `rank` arrays, each with `n` elements. Initialize `elements[i]=i` and `rank[i]=0` for all slots.
- `~DisjointSet()`
  Deletes the arrays.
- `find(int a)`
  Performs Algorithm 3 to find the top of the intree containing a.
- `join(int a,int b)`
  Performs Algorithm 2 to join a's and b's sets together.

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 $\textsc{GenerateMaze}(n_R, n_C)$
    $i \leftarrow 0$
    for $r \leftarrow 0$ **to** $n_R - 1$ **do**
        for $c \leftarrow 0$ **to** $n_C - 1$ **do**
            $maze[r][c] \leftarrow 63$ ▹ 63 = 0b00111111; marks all six walls present
        end for
    end for
    Initialize disjoint set object `ds` with $n_R \cdot n_C$ elements
    Initialize sampler object `sampler` with $3 \cdot nR \cdot n_C$ elements

    $i \leftarrow 0$
    while $i < n_R \cdot n_C - 1$ **do**
        do
            do
                $e \leftarrow sampler.getSample()$
            while e references an exterior or nonexistent wall
            $(r_1, c_1, dir_1) \leftarrow decodeCell(e)$
            Set (r_2, c_2) to cell adjacent to (r_1, c_1) in given direction
            $cell_1 \leftarrow encode(r_1, c_1, 0)$
            $cell_2 \leftarrow encode(r_2, c_2, 0)$
        while $\textsc{DisjointSetFind}(cell_1) = \textsc{DisjointSetFind}(cell_2)$

        $\textsc{DisjointSetUnion}(cell_1, cell_2)$
        $i \leftarrow i + 1$

        Remove wall between (r_1, c_1) and (r_2, c_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
  $(r, c, d) \leftarrow decode(S.peek())$
  if $r = n_R - 1$ **and** $c = nC - 1$ **then**
   break
  end if

  if $d = 6$ **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.