

# Data Structures and Objects

## CSIS 3700

Fall Semester 2018 — CRN 41930

---

### Project 4 — Pathfinder

Due date: Thursday, December 13, 2018 at noon

#### 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 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 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

```

1: procedure SAMPLENoREPLACEMENT(items,n)
2:   i ← RAND mod n
3:   e ← items[i]
4:   n ← n − 1
5:   items[i] ← items[n]
6:   return e
7: end procedure

```

---

▶ Select random position in the list  
 ▶ Remember the selected item  
 ▶ Decrement *n*  
 ▶ Move last item into selected position

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  $d \in \{0, 1, 2, 3\}$  as  $e = d + 4(c + n_C \cdot r)$ . Given *e*, it is easy to extract *r*, *c* and *d*.

Use  $d \in \{0, 1, 2, 3\}$  to encode up, right, down and 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

```

1: procedure DISJOINTSETUNION(elements,rank,a,b)
2:   a ← DISJOINTSETFIND(a)                                ▶ Get representatives for a and b
3:   b ← DISJOINTSETFIND(b)

4:   if a ≠ b then                                          ▶ Only union if a and b are in different sets
5:     if rank[a] < rank[b] then                               ▶ Set with lower rank merged into set with larger rank
6:       elements[a] ← b
7:     else
8:       if rank[a] = rank[b] then                             ▶ In case of tie, increment one set's rank
9:         rank[a] ← rank[a] + 1
10:      end if
11:      elements[b] ← a
12:    end if
13:  end if
14: 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*

```

1: procedure DISJOINTSETFIND(elements,rank,a)
2:   if elements[a] ≠ a then                                ▶ Connect a directly to top of intree
3:     elements[a] ← DISJOINTSETFIND(elements[a])
4:   end if
5:   return elements[a]                                       ▶ Return top of intree
6: 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  $\text{DISJOINTSETFIND}(a) \neq \text{DISJOINTSETFIND}(b)$ . 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

```

1: procedure GENERATEMAZE( $n_R, n_C$ )
2:    $i \leftarrow 0$ 
3:   for  $r \leftarrow 0$  to  $n_R - 1$  do
4:     for  $c \leftarrow 0$  to  $n_C - 1$  do
5:        $\text{maze}[r][c] \leftarrow 15$ 
6:     end for
7:   end for
8:   Initialize disjoint set object ds with  $n_R \cdot n_C$  elements
9:   Initialize sampler object sampler with  $4 \cdot n_R \cdot n_C$  elements

10:   $i \leftarrow 0$ 
11:  while  $i < n_R \cdot n_C - 1$  do
12:    do
13:      do
14:         $e \leftarrow \text{sampler.getSample}()$ 
15:        while  $e$  references an exterior or nonexistent wall
16:           $(r_1, c_1, \text{dir}_1) \leftarrow \text{decodeCell}(e)$ 
17:          Set  $(r_2, c_2)$  to cell adjacent to  $(r_1, c_1)$  in given direction
18:        while  $\text{DISJOINTSETFIND}(r_1) = \text{DISJOINTSETFIND}(r_2)$ 

19:         $\text{DISJOINTSETUNION}(r_1, r_2)$ 
20:         $i \leftarrow i + 1$ 

21:      Remove wall between  $(r_1, c_1)$  and  $(r_2, c_2)$ 
22:    end while
23:  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

```

1: procedure FINDPATH(maze)
2:   S.push(encode(0, 0, 0))
3:   Mark (0, 0) as visited
4:   while true do
5:      $(r, c, d) \leftarrow \text{decode}(\text{S.peek}())$ 
6:     if  $r = n_R - 1$  and  $c = n_C - 1$  then
7:       break
8:     end if

9:     if  $d = 4$  then
10:      Mark  $(r, c)$  as a dead end
11:      S.pop()
12:     else
13:      Let  $(r', c')$  be the next cell in direction  $d$ 
14:      Replace encode( $r, c, d$ ) with encode( $r, c, d + 1$ ) on top of stack
15:      if no wall exists in direction  $d$  and  $(r', c')$  is not marked as visited then
16:        S.push(encode( $r', c', 0$ ))
17:        Mark  $(r', c')$  as visited
18:      end if
19:     end if
20:   end while
21: 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.