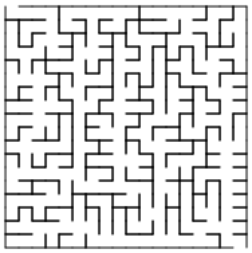
CSCI 241, Project #3, Random Maze Generation

Geoffrey Matthews

April 21, 2017

Note: This is still incomplete.



Due date: Midnight, May 26.

Turn in: Be sure to **zip** all your source files together before submission. Canvas downloads mangle the names, and java likes to have the names of the files the same as the public class in the file.

To make it easier for me to write scripts to unpack, compile, and run your file:

- Use **zip**, not tar or 7z or any other archiving software.
- Zip the folder, not the individual files in the folder.
- Name the zip file `lab03.zip`.
- Nmae the file (and class) with your `main` method in it `lab03.java`.

Input and output: The number of rows and the number of columns as arguments. The output will be a graphical image of the maze (see the java example from the previous lab to see how to make images).

Maze algorithm: To draw a maze, consider a rectangular grid with each cell labelled and walls between all adjacent cells. as in Figure 1 (a). In the beginning, we regard each cell as in its own set by itself (since it is separated from the others by walls). Thus, we have a *partition* of the set of n cells into n sets. We will now gradually reduce the size of this partition to 1 by merging two sets at a time.

We pick two adjacent cells from different sets at random, for example, 4 and 9. We remove the wall between them, merging their sets and getting Figure 1 (b), and we now have 24 sets of cells.

We now pick another pair of adjacent cells at random, which are in *different* sets, for example 3 and 4, and remove a wall between them. This merges the sets $\{3\}$ and the set $\{4, 9\}$ into the set $\{3, 4, 9\}$, and results in Figure 1 (c).

Now we pick a different pair of random adjacent cells from different sets, for example 20 and 21, and merge them, giving Figure 1 (d).

We continue in this fashion, each time removing a wall between two random *adjacent* cells that are in *different* sets, until we have only one set remaining, as in Figure 1 (e).

At this point, we have a maze if we just remove the cell labels and leave an entrance at the upper left, and an exit at the lower right, as in Figure 1 (f).

It is crucial that we always pick a pair of adjacent cells from *different* sets, because otherwise our maze would have cycles in it.

Picking random adjacent cells: As the algorithm progresses, there will be fewer and fewer adjacent cells in different partitions. If we just pick cells at random, it will take longer and longer to find a candidate to merge. To solve this we take the following approach.

Before we begin, we form a complete list of all adjacent cell pairs. Each cell in the range (0... n-1) is paired up with the cells to the right and below it. For example, for the cells in Figure 1, the list would start out looking like this:

$$((0,1),(0,5),(1,2),(1,6),(2,3),(2,7),(3,4),(3,8),(4,9),(5,10),\ldots,(21,22),(22,23),(23,24))$$

Note that some cells (on the right side and the bottom) have only one adjacent cell, and the last cell has no adjacent cell.

Now we randomize this list. The algorithm is simple: for each cell, swap that cell with one at a random location.

We use this list every time we want a new pair of random cells. If we merge the cells (because they were in disjoint partitions), we keep them in one list, and if we don't merge them (they were in the same partition), we keep them in a different list.

The reason we keep the cells that were *not* merged, is because that is where all the internal walls are drawn in the final maze. Having this list makes it easy to draw the final maze: draw the outside border (leaving the entrance and exit open), and then draw a wall segment between each pair of *unmerged* cells.

Merging and detecting disjoint sets: Now that we have a random pair of cells, another problem is maintaining the partition of the cells, with the following operations as fast as possible:

- Tell if two cells are in the same partition.
- Merge two partitions into one.

If we do this inefficiently, every time we have two cells, i and j, and want to know if they are in the same set, we may have to use an $O(n)$ process to find this out. We can do a lot better than that.

To accomplish this, wre going to use a forest of trees to represent the partition. We are going to have to insert and delete trees from this forest, so pick an efficient data structure for that.

We are also going to have to quickly find a node in the tree, given just its index, so keep an array $(0,\ldots,n-1)$ of pointers to the individual nodes as well.

For these trees we are only going to use *parent* pointers, and they will be general trees, not binary trees.

Each cell starts out in its own tree, as in Figure 2 (a). In order to merge cells 4 and 9, we simply make the root of one the parent of the root of the other; for example as in Figure 2 (b), where we made 9 the parent of 4. In (c) we join 3 and 4 by merging $\{3\}$ with $\{4,9\}$, by making 9 the parent of 3 (since 9 is the root of $\{4,9\}$). The next few merges are shown in Figure 2 (d) to (h). (Which cells were merged at each step?)

Clearly, given two cells, merging their sets is a fast operation. We merely make the parent of one point to the other. But what about determining if two cells are in the same set?

The beauty here is that it is going to be very fast to find the root of each cell's tree. Two cells are in the same set if they have the same root. For example, in Figure 2 (d), cells 9 and 20 are in different sets, because $\text{root}(9) = 4 \neq 21 = \text{root}(20)$.

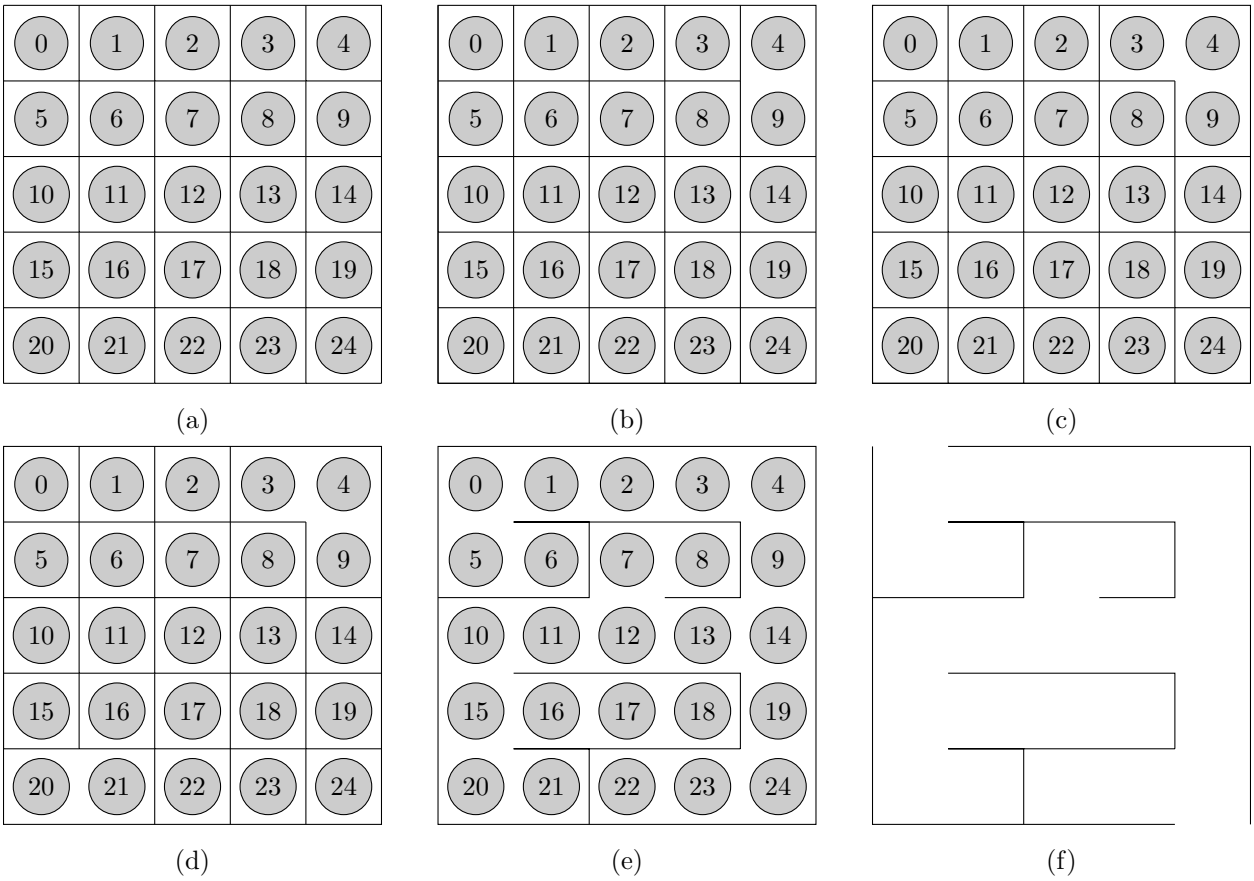


Figure 1: Merging cells to make a maze. The original partition is (a), after merging $\{4\}$ and $\{9\}$ we get (b), after merging $\{3\}$ and $\{4, 9\}$ we get (c), and after merging $\{20\}$ and $\{21\}$ we get (d). A full merge down to one set is shown in (e), and drawn as a maze in (f).

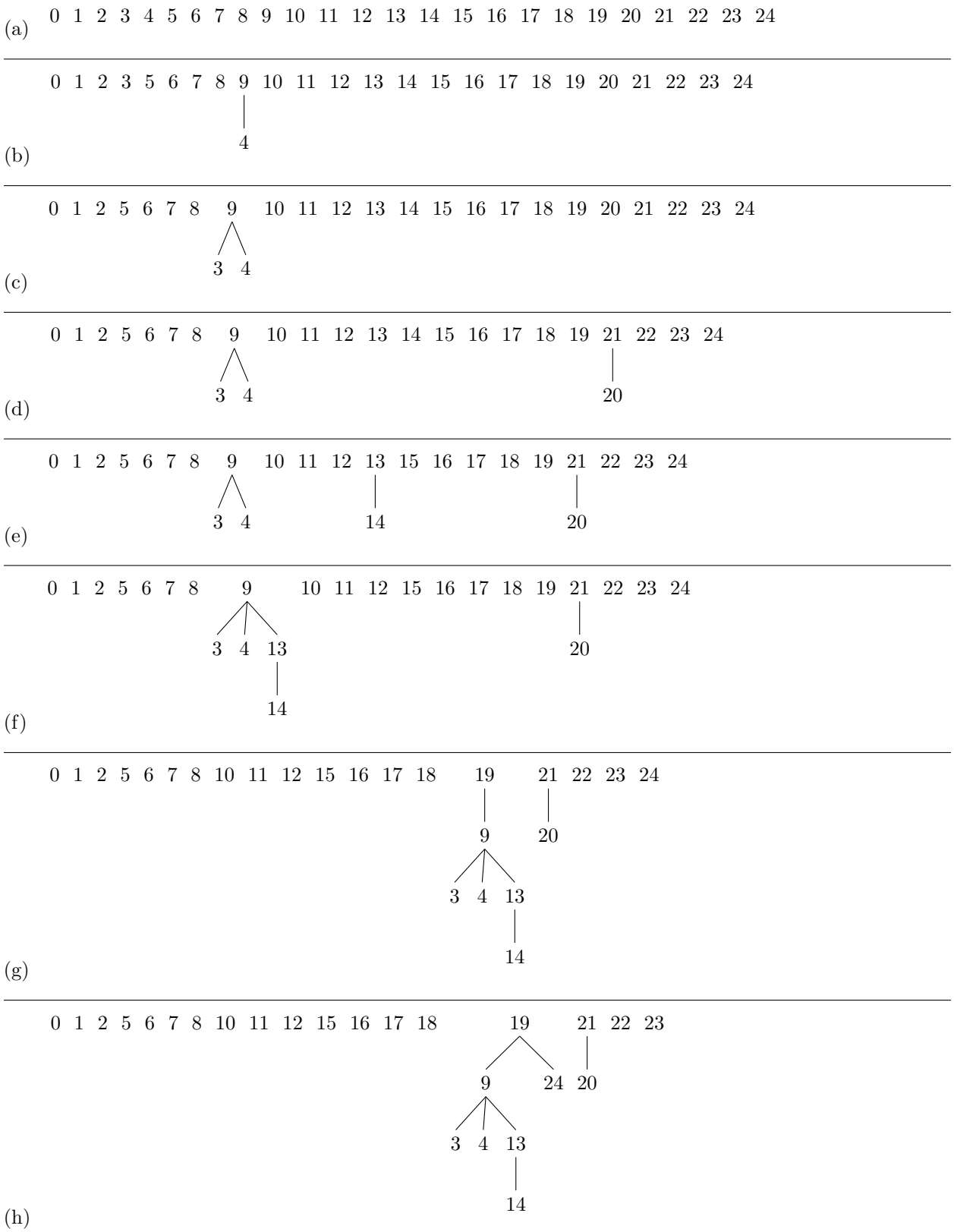


Figure 2: Merging sets with trees. Which two cells are merged at each stage?