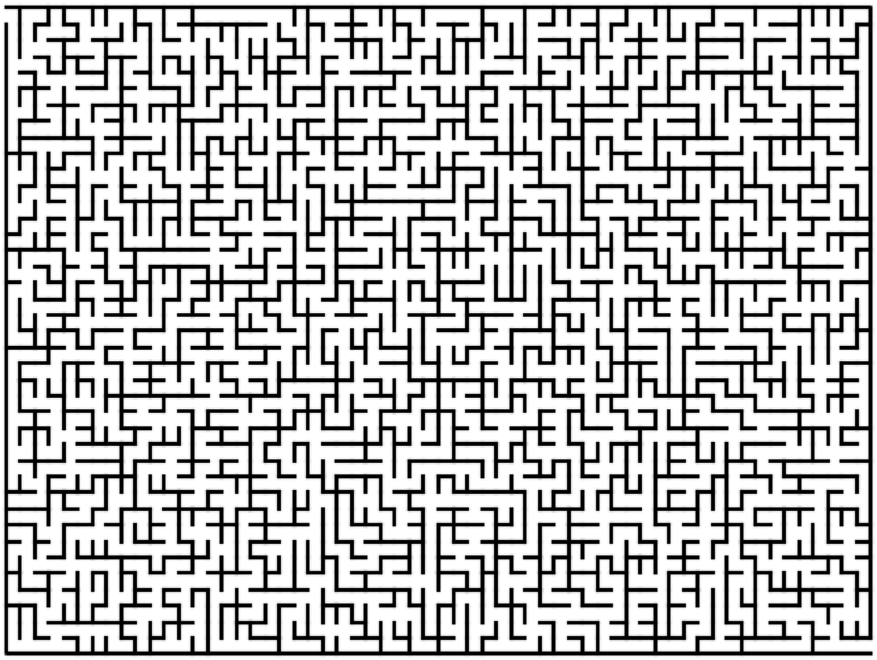
Random Maze Generation

CSCI112, Lab 9

March 16, 2023



File names: Names of files, functions, and variables, when specified, must be EXACTLY as specified. This includes simple mistakes such as capitalization.

Individual work: All work must be your own. Do not share code with anyone other than the instructor and teaching assistants. This includes looking over shoulders at screens with the code open. You may discuss ideas, algorithms, approaches, *etc.* with other students but NEVER actual code. Do not use code written by anyone else, in the class or from the internet.

Documentation: Each file should begin with a docstring that includes your name, the class number and name, the lab number, and a short description of the lab, as well as documentation pertinent to that particular file.

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 put a door (remove the wall) between them, merging their sets and getting Figure 1 (b), and we now have 24 sets of cells, all the sets not including 4 or 9, and the set $\{4, 9\}$.

We now pick another pair of adjacent cells at random, which are in *different* sets, for example 3 and 4, and put a door (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, put a door between them and merge their sets, giving Figure 1 (d).

We continue in this fashion, each time removing a wall between two random *adjacent* cells that are in *different* sets, merging the 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. If they are not adjacent putting a door between them makes no sense. If they are in the same partition, 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 grid is paired up with the cells to the right and below it, if they exist. 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 pair of cells, swap that pair with a pair at a random location. Or you can use the `random.shuffle` routine.

We use this list every time we want a new pair of random cells. We remove the first pair and check to see if the two cells are in different partitions. If they are in different partitions, we put a door there and merge the partitions. We also eliminate this pair from our list of walls. If we don't put a door there (they were in the same partition), we keep them in the `walls` list.

We can now use the resulting list of remaining `walls` to tell us 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 cells in the `walls` list.

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:

- Determine 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.

For these trees we are only going to use *parent* pointers, and they will be general trees, not binary trees. Rather than create a data structure for the cells with a parent pointer, we'll use a dictionary of parents, with the keys being the cells.

Each cell starts out in its own tree, as in Figure 2 (a). Thus, each cell has `None` as its parent.

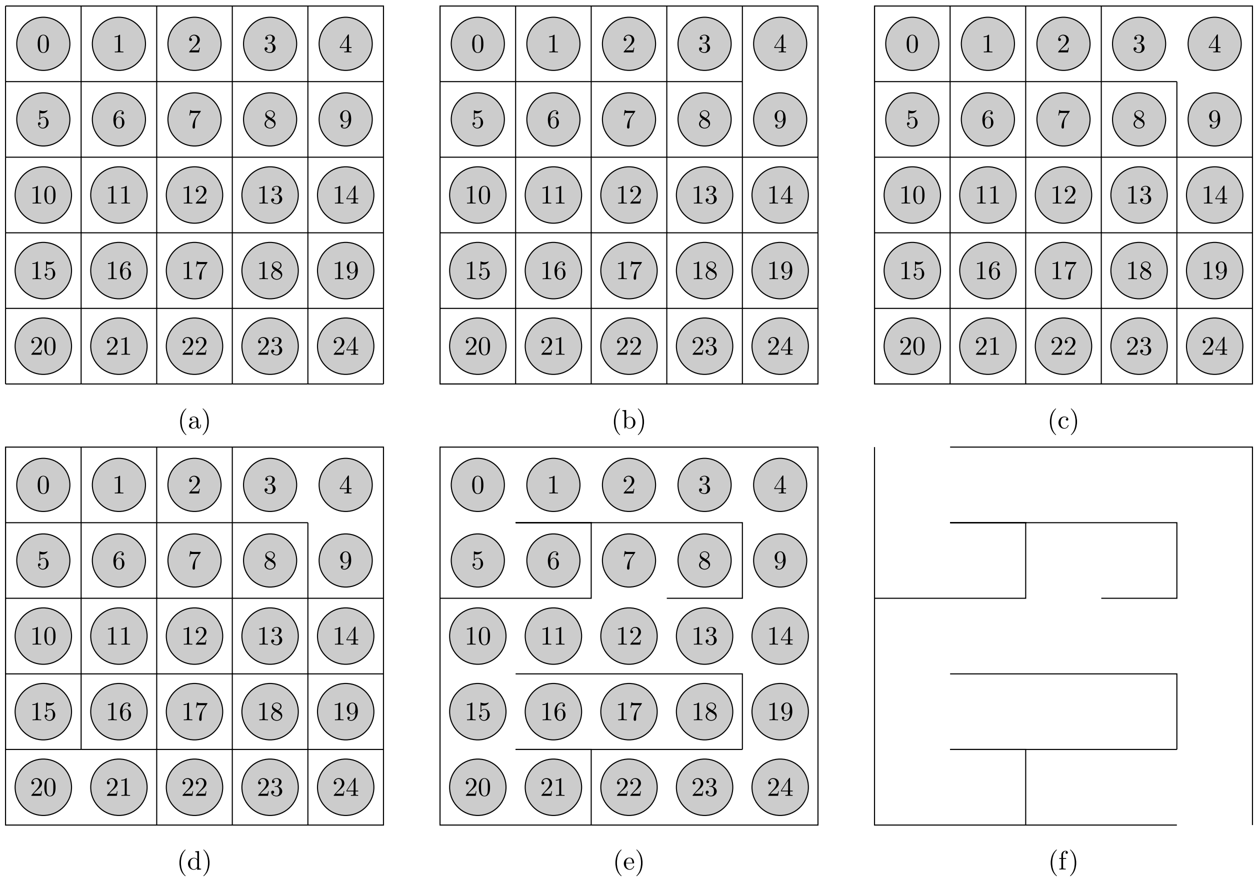


Figure 1: Merging cells to make a maze. The original partition is (a), after merging $\{4\}$ and $\{9\}$ we get (b), after merging $\{3\}$ and $\{4, 9\}$ we get (c), and after merging $\{20\}$ and $\{21\}$ we get (d). A full merge down to one set is shown in (e), and drawn as a maze in (f).

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 find their roots, and then make the root of one be the parent of the other.

But what about determining if two nodes are in the same partition?

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 (h), cells 3 and 20 are in different sets, because $\text{root}(3) = 19 \neq 21 = \text{root}(20)$.

Implementation: We can implement these ideas with just two major data structures: `walls` and `parents`.

`parents` is a dictionary with each cell in the maze being a key, and the value is either `None` or another cell in the maze. This is initialized to all `None`, indicating that each cell is alone in its own class.

`walls` is a list of *pairs* of cells, indicating that there is a wall between the two cells. To begin with, there is a wall between every adjacent pair of cells in the maze.

Since we only need to represent a wall once, we don't need to store both the north and south walls, for example, or both the east and west walls. We arbitrarily decide that each cell will hold only the south and east neighbors (as in the algorithm discussion, above).

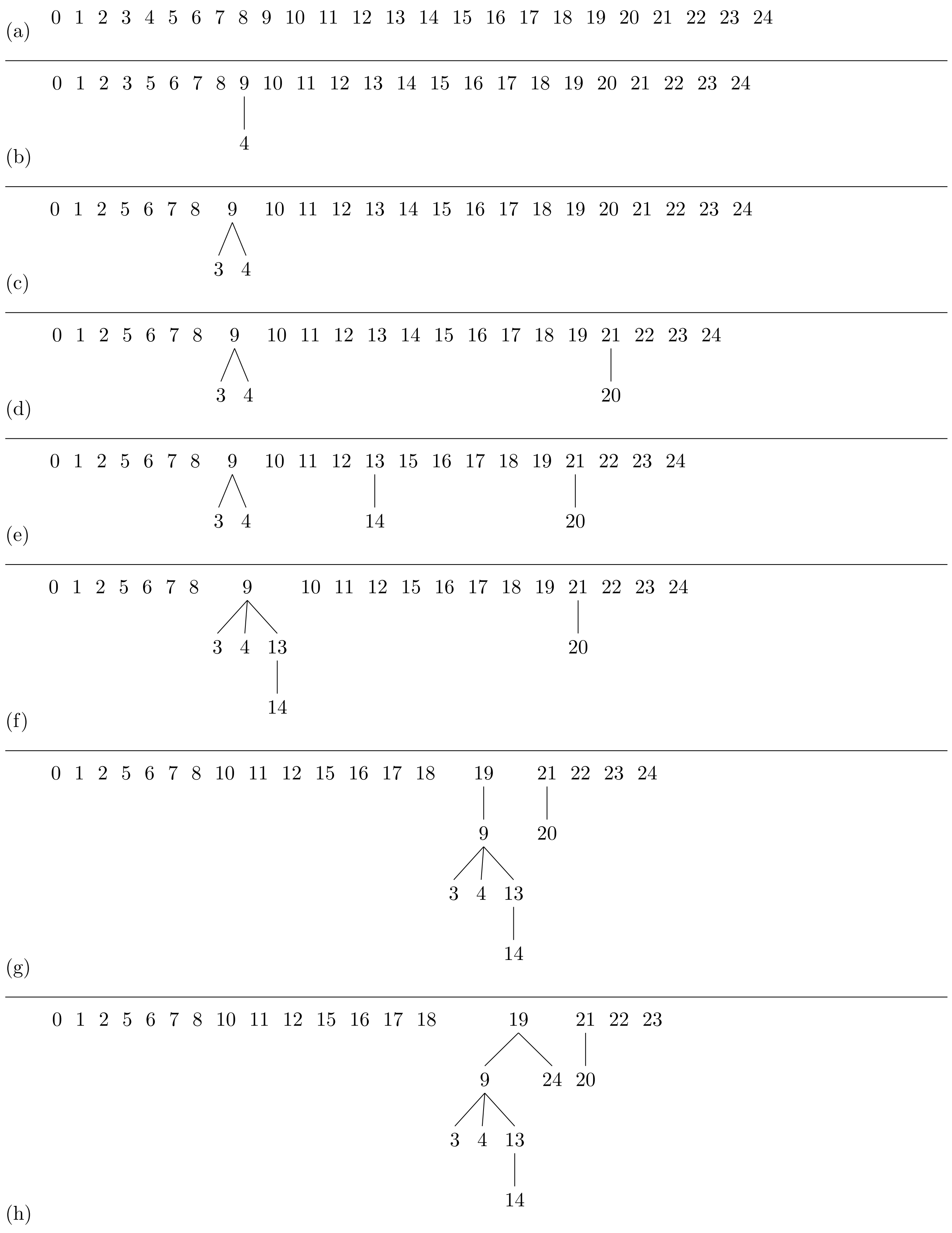

(a) 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
(b) 0 1 2 3 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24; 4
(c) 0 1 2 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24; 3 4
(d) 0 1 2 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 23 24; 3 4; 20
(e) 0 1 2 5 6 7 8 9 10 11 12 13 15 16 17 18 19 21 22 23 24; 3 4; 14; 20
(f) 0 1 2 5 6 7 8 9 10 11 12 15 16 17 18 19 21 22 23 24; 3 4 13; 14; 20
(g) 0 1 2 5 6 7 8 10 11 12 15 16 17 18 19 21 22 23 24; 9; 20; 3 4 13; 14
(h) 0 1 2 5 6 7 8 10 11 12 15 16 17 18 19 21 22 23; 9 24 20; 3 4 13; 14

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

Make sure that the cells along the edges do not have neighbors that are off the board!

With these data structures it is easy to implement the functions we need.

- `root`: find the root of a cell from the `parents`
- `join`: join the two sets of cells in different sets into one set
- `eraseWalls`: loop through the list of walls and eliminate them one by one as we unify cells, in line with the above algorithm
- `drawMaze` one simple way to draw a set of lines is with `matplotlib`. Import the plotting object with

  ```
  import matplotlib.pyplot as plt
  ```

  Here is my code:

  ```
  def drawMaze(walls, mazeSize):
      nrows, ncols = mazeSize
      fig,ax = plt.subplots()
      ax.axis('off')
      plt.plot((0,ncols,ncols),
               (nrows,nrows,1),
               linestyle='-',color='black')
      plt.plot((ncols,0,0),
               (0,0,nrows-1),
               linestyle='-',color='black')
      for wall in walls:
          i,j = wall
          y1,x1 = position(i,mazeSize)
          y2,x2 = position(j,mazeSize)
          if x1 == x2:
              plt.plot((x1, x1+1), (y1,y1), linestyle='-',color='black')
          else:
              plt.plot((x1,x1), (y1,y1+1), linestyle='-',color='black')
      plt.show()
  ```

  In my implementation I represented cells as single integers, as in the notes above, hence I needed to translate an integer into a two dimensional position with the `position` function. If you represent cells as pairs, you may not need this step.

Analysis: Note that the whole algorithm is $O(n \log n)$, where n is the number of cells. There are about $2n$ walls, and for each wall we check whether the endpoints are in the same class with about $\log n$ steps, thanks to the forest of trees. If we have to join two sets, it is a $O(1)$ process, again thanks to the forest of trees. Without this optimization, it is very easy for these kinds of algorithms to become $O(n^2)$ or worse.