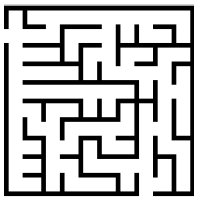
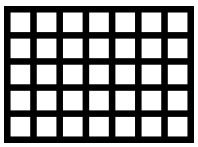
Using the JPD algorithm to draw a random maze



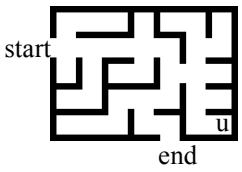
We discuss modifying the JPD algorithm to generate random mazes —the one on the right was created by our Java program. This modification of JPD creates random spanning trees instead of minimum-cost spanning trees. This is a simple change.

But there is more to this story. This is a neat example of how data structures can be mightily simplified when the models they represent have special, simple forms. But before we get into data structures, we show you exactly what the graph is and what its spanning tree looks like.



To the right is a graph `G`, with lines separating nodes: Each of the 35 white squares in the grid is a node of the graph. Also, implicitly, each node is connected by undirected edges to its neighbors to the North, East, West, and South. We don't show the edges, we just tell you: Each pair of neighbors is connected by an undirected edge. That's graph `G`.

Next, construct a random spanning tree of the graph. That's done using a modification of JPD, which we discuss below.



Finally, draw the maze in two steps. First draw the grid in a GUI, as done above, with lines separating adjacent nodes. Then, if two nodes are connected by a spanning-tree edge, *remove the black line separating them.* For example, node `u` in the southeast corner is connected by an edge of the spanning tree to the node to its west, so the black line between them has been deleted. A spanning-tree edge doesn't connect `u` to node north of `u`, so the black line remains. That's it!

There are two processes going on here. First, construct a minimum spanning tree of the graph. Second, based on the graph and its spanning tree, draw the grid in a GUI and remove lines in the grid between each pair of nodes that are connected by an edge of the spanning tree. We have also marked two nodes as the start and end nodes and deleted the border lines adjacent to them.

Because of the way the GUI represents the spanning tree of the graph, it's guaranteed that there is one simple path from *start* to *end* —or between any pair of nodes, for that matter.

Implementing the graph and the spanning tree

We use a 2-dimensional array of nodes `m[width][height]`. Node `m[0][0]` is the upper left node, we call it the northwest node. The node east (to the right) of that is `m[1][0]`, and the one south (down) of `m[0][0]` is `m[0][1]`. This is the standard way to label pixels in a GUI `JPanel`.

We want `m` to contain graph `G` and the spanning tree, as it is being constructed. In the JPD algorithm, the spanning tree being constructed, as we have described JPD, is component `Cw`, and initially it contains start node `w`. Nodes adjacent to `Cw` in the graph are in a set `F`. And, finally, there are all the other one-node trees.

We give an example using an array `m[4][3]`. Its elements will be values of an `enum NodeType`. We use `NodeType.w` for the value of the node containing `w`, `NodeType.F` for nodes in set `F`, i.e. nodes that are adjacent to `Cw`, and `NodeType.O` for the other nodes. With `m[1][0]` as start node `w`, the initial value of array `m` is as shown to the right.

```
F w F O
O F O O
O O O O
```

Each iteration of the JPD algorithm places one node from set `F` into component `Cw`. In this version of JPD, a random node is chosen from `F`. In this example, we randomly choose node (1, 1), which is south of node `w`. We have to change the value `m[1][1]` to indicate that it is now in `Cw`. At the same time, we have to indicate that edge `{(1, 1), (1, 1)}` is in `Cw`. How can we do this? Here's the trick. The edge runs **N**orth from node `m[1][1]` to node `w`, which is `m[1][0]`. So let's put a value `NodeType.N` in `m[1][1]` so that the array is now as shown on the right.

```
F w F O
O N O O
O O O O
```

We see that five different values are used to represent nodes in `Cw`. Start node `w` is represented by value `w` of `enum NodeType`. Any other node in `Cw` is represented by a value `N`, `E`, `W`, or `S` to indicate the direction in which a spanning-tree edge goes from that node. Neat!

Here's the declaration of enum variable `NodeType`:

```
public enum NodeType {w, // Node is in Cw and it's the start node
                      N, // Node is in Cw. Other node of spanning-tree edge is to the north
                      E, // Node is in Cw. Other node of spanning-tree edge is to the east
                      W, // Node is in Cw. Other node of spanning-tree edge is to the west
                      S, // Node is in Cw. Other node of spanning-tree edge is to the south
                      F, // Node is in frontier F
                      O} // Not in Cw or F
```

The two variables main of the algorithm

The algorithm uses these two variables:

```
NodeType m[width][height] // It's discussed above
ArrayList<Point> F        // list of indices of nodes not in Cw that are adjacent to nodes in Cw
```

Class `Point` has two **int** fields to give the indices (`x`, `y`) of a node. For example, for the initial array `m` above, with `w` in `m[1][0]`, `F` contains three `Point` values: (0, 0), (2, 0), and (1, 1).

Each iteration of the algorithm

Each iteration of the JPD algorithm to build a spanning tree for a maze does this:

1. Choose a random `Point p` in `ArrayList F` and remove it from `F`[1].
2. Suppose `p` is (`x`, `y`). We know that `m[x][y]` = `NodeType.F`. Make a list `L` of nodes that are adjacent to `p` (look north, east, west, and south) and that are in `Cw`.
3. Choose a random node `u` from list `L`.
4. Put node `m[x][y]` and the edge from `m[x][y]` to `u` into `Cw`. Do this simply like this: Change `m[x][y]` to `NodeType` value `N`, `E`, `W`, or `S`, depending on the direction from `m[x][y]` to u.
5. Add to set `F` (and change their `NodeType` value to `F`) all nodes that are adjacent to `m[x][y]` and have value `NodeType.O`.

Time complexity

Suppose graph `G` contains `n` nodes. Then `n-1` iterations, as described above, are executed to add edges to the spanning tree. We analyze the time it takes to execute one iteration.

`G` is sparse, with at most 4 edges leaving each node. Therefore, list `L` in step 2 above has size ≤ 4. In step 4, determining the direction requires looking in at most 4 directions. Finally, step 5 requires looking at the 4 nodes that are adjacent to `m[x][y]`. Therefore, it takes O(`1`) time to execute one iteration of the algorithm.

Therefore, the algorithm takes time O(`n`).

[1] Here, removing `F[i]` from `ArrayList F` can be done in constant time: Put `F [F.size()-1]` into `F[i]` and remove `F [F.size()-1]`.