Assignment 10: Maze of twisty passages

Goals: Practice working with graphs and graph algorithms by designing mazes using Kruskal's algorithm, and solving them using either breadth- or depth-first searches.

Due: Wednesday, April 17th at 9:00pm

You will be using the Impworld library — make sure that at the top of your file, you include

```
import java.util.ArrayList;
import tester.*;
import javalib.impworld.*;
import java.awt.Color;
import javalib.worldimages.*;
```

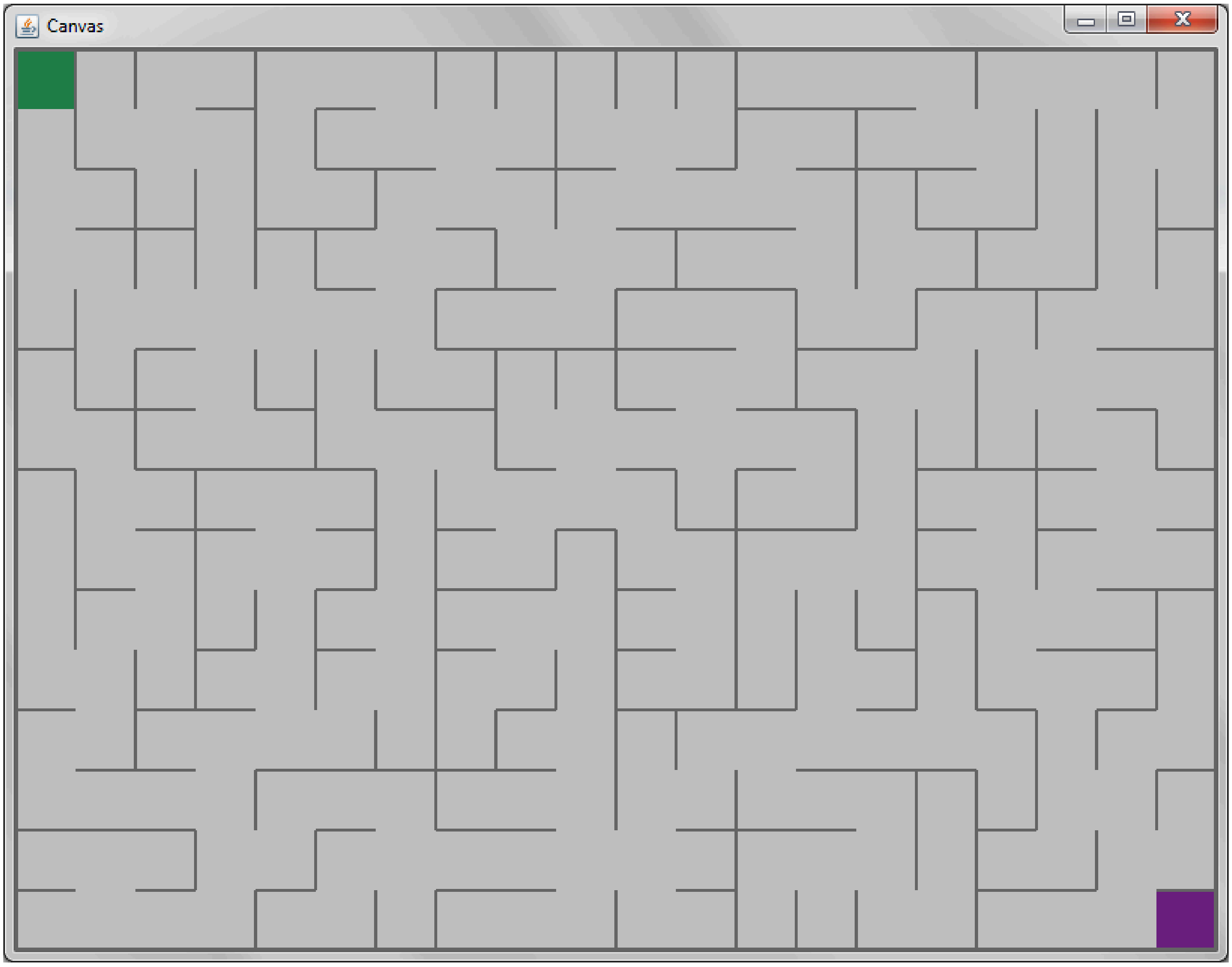
Make sure you do not name any of your files `World.java`, or else the autograder will not be able to compile your code.

Read the entire assignment before starting! Most of your likely questions are probably addressed in here, just not merely in the Requirements section...

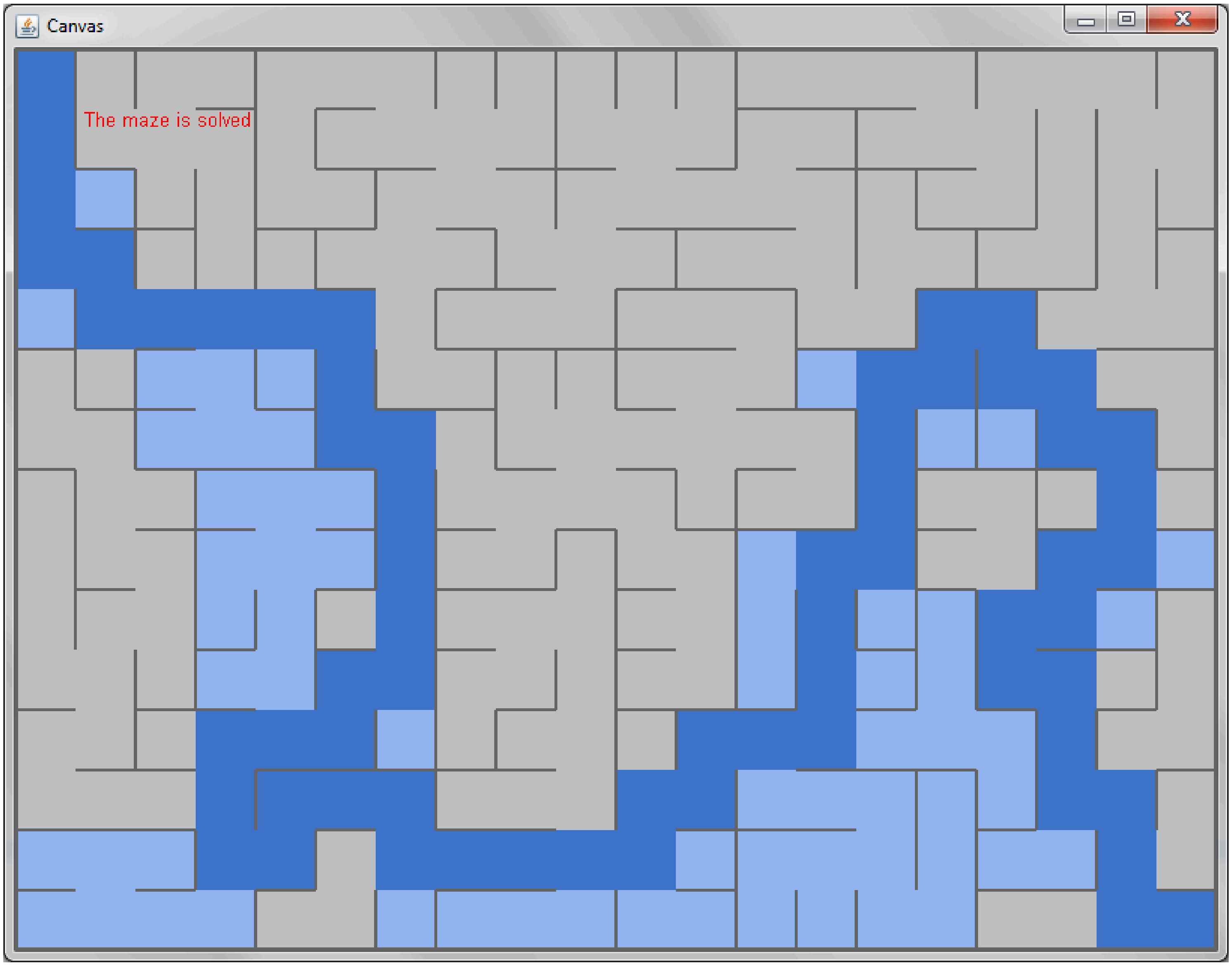
Extra credit will only count if they are convincingly and thoroughly tested, and if the rest of the assignment is completed equally thoroughly — you will not receive extra credit if the minimum functionality does not work properly. (You should again aim to implement features that demonstrate that you've mastered concepts that you got wrong on your exams.)

In class, we have been discussing various algorithms for working with graphs, that require the use of several data structures working together. We have talked about general-purpose maze searching algorithms, like breadth- and depth-first searches, and we have talked about building minimum spanning trees on graphs.

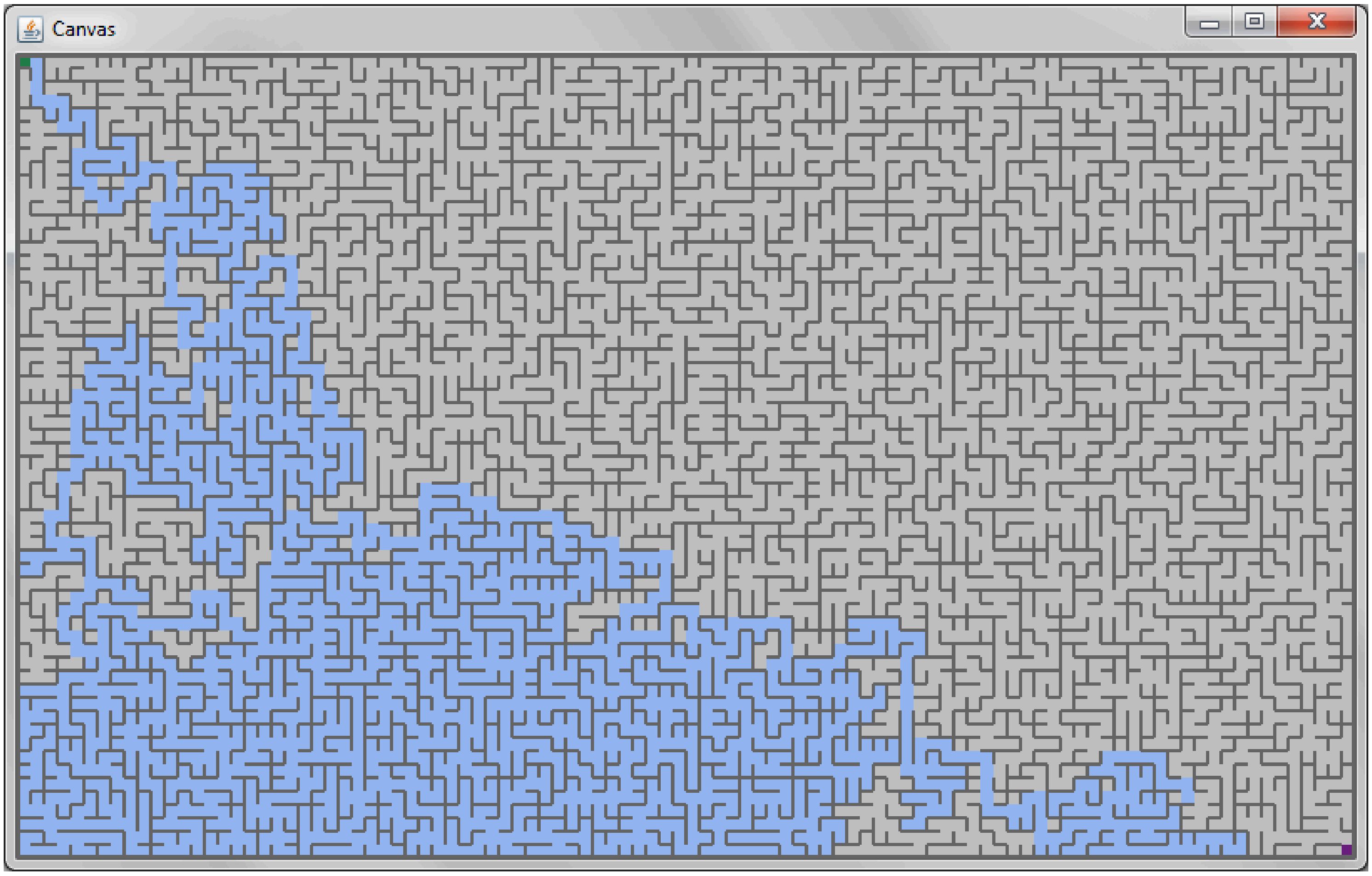
To get a visual grasp of how these algorithms work, you are going to be building and solving mazes, like this one:



The mazes you construct should start in the upper-left corner (shown in green) and end in the lower-right corner (shown in purple). As you solve the mazes, you should color in the cells you have explored. Once you have reached the solution, you must backtrack the path from the end to the start, and draw it as well. A fully-solved maze might look like this:



Your code must handle mazes of arbitrary sizes, up to at least 100x60:



In particular, your code must not crash with stack overflows...

10.1 Requirements

All your classes' fields must be marked `private`. As many as possible must also be marked `final`. You should avoid any gratuitous getter methods: in other words, a purpose statement of "getFoo(): Gets the foo field" is *not* a sufficient reason for that method to exist. You may try "Gets the foo field, as needed by somethingElse()", but you'll need to justify why somethingElse really needs access to that field.

If you cannot make a field `final` or `private`, leave a comment explaining what other code is permitted to modify or access it, and why. (Remember that `final` is shallow: even if an `ArrayList` variable is marked `final`, you may still mutate the *contents* of that list; all you are prevented from doing is *reassigning* that variable.)

Your program should support at minimum the following features:

- Construct random mazes using Kruskal's algorithm and Union/Find (below)
- Display the maze graphically and animate the search for the path.
- Allow the user to choose one of two algorithms for finding the path: Breadth-First Search or Depth-First Search (below).
- Provide an option for designing a new random maze.
- Allow the user to traverse the maze manually - using the keys to select the next move, preventing illegal moves and notifying the user of completion of the game.

- Display the solution path connecting the start and end, once it's found (either automatically or by the user).

Be sure to submit documentation for your code, so the graders know how to run and play your game. As always, be sure to test your code thoroughly.

Recommended timeline:

- By Friday, April 12th, be able to create mazes and draw them
- By Wednesday, April 17th, be able to solve (and animate the solutions of) mazes using either breadth- or depth-first search

Additionally, you may attempt bells and whistles for extra credit:

Whistles:

- Provide an option to toggle the viewing of the visited paths.
- Allow the user the ability to start a new maze without restarting the program.
- Keep the score of wrong moves — for either the automatic solutions or manual ones — and maybe keep statistics on which one of the two algorithms had fewer steps for each maze.

Bells:

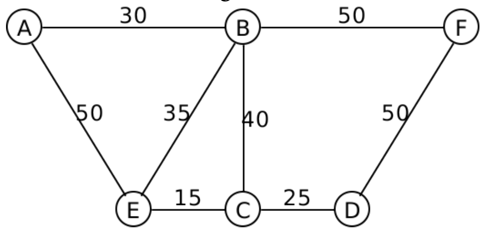
- In addition to animating the solution of the maze, also animate the construction of the maze: on each tick, show a single wall being knocked down.
- Color every square with a gradient of colors indicating how far it is from the start of the maze. E.g. red means very close to the start, and blue means very far.
- Color every square with a gradient of colors indicating how far it is from the exit of the maze. E.g. red means very close to the exit, and blue means very far. (Are these colors exactly the opposite of the previous ones, or is this a different color pattern altogether?)
- *(Tricky)* Construct mazes with a bias in a particular direction —a preference for horizontal or vertical corridors. (Hint: you might wish to play tricks with the edge weights here.)
- **Hard! (But very cool)** Instead of constructing a rectangular maze, try constructing a hexagonal one. To do this, you'd need:
  - To figure out how to render a hexagon. See The Image Library.
  - To figure out how to represent a hexagonal grid. You'll need to update your maze-cell class to have six neighbors instead of four. You'll also need to figure out how to represent a hexagonal grid using a normal `ArrayList<ArrayList<YourMazeCell>>`. (Hint: if you look at the rows of a hexagonal grid, every other row is "shifted" by half a cell-width, but there are the same number of cells in every row, so a regular `ArrayList<ArrayList<YourMazeCell>>` should still work.)
  - To figure out how to render a hexagonal grid. You'll need a little bit of math to figure out the centers of each hexagon.
  - To figure out how to do maze generation. Your initial graph will need to be a bit different...
  - To figure out user-input controls. I suggest using the letter 'a', 'w', 'e', 'd', 'x' and 'z', to mean their "obvious" directions (relative to the letter 's' on the keyboard). If you *also* implement two-player mode, you'll need to come up with different letters for that player and/or the maze-resetting letters.

- **Tricky** Construct two intertwined mazes, and allow two players to race from their starting points to their ending points. Choosing the start points for both mazes is easy; choosing end points is harder. Even more impressive is ensuring that it's a fair race, and both mazes have the same length path from start to finish... (Hint: how can you force Kruskal's algorithm to produce two distinct connected mazes? Talk to an instructor if you choose to do this, and are not sure how to proceed.)

Spend careful thought planning ahead and designing your classes: if your design is too brittle, you'll have a very hard time completing the algorithms. And as always, have fun!

10.2 Kruskal's Algorithm for constructing Minimum Spanning Trees

Here is Kruskal's algorithm illustrated on a particular example graph:



(The edges are drawn without directional arrows; in your mazes, every maze cell will be connected to its four neighbors, so edges are effectively undirected. Edge weights are notated as numbers on the edges.)

Kruskal's algorithm begins by sorting the list of edges in the graph by edge weight, from shortest to longest:

```
(E C 15)
(C D 25)
(A B 30)
(B E 35)
(B C 40)
(F D 50)
(A E 50)
(B F 50)
```

At each step we remove the shortest edge from the list and add it to the spanning tree, provided we do not introduce a cycle. In practice, this may produce *many* trees during the execution of the algorithm (so in fact, the algorithm produces a spanning *forest* while it runs), but they will eventually merge into a single spanning tree at the completion of the algorithm.

For this particular graph, we add the edges `(E C 15)`, `(C D 25)`, `(A B 30)` and `(B E 35)`. When we try to add the edge `(B C 40)` we see that it would make a cycle, so this edge is not needed and we discard it. We then add edge `(F D 50)`. This connects the last remaining unconnected node in the graph, and our spanning tree is complete. In very high-level pseudocode, the algorithm is quite short and elegant:

```
while (we do not yet have a complete spanning tree)
  find the shortest edge that does not create a cycle
  and add it to the spanning tree
```

Determining if we have a complete spanning tree is easy: for n nodes, we need $n - 1$ edges to connect them all.

> ***Do Now!***
>
> Why can't we have fewer edges? Why can't we have more?

We can represent the spanning tree itself by a list of edges. Adding an edge to that list is as easy as `Cons`’ing it on, or `add`ing it, depending on which representation of lists you choose to use. Finding the shortest edge is easy, since we began by sorting the list of edges by their weights. The only tricky part in this algorithm is figuring out whether a given edge creates a cycle with the edges we have already selected. For this we use the Union/Find data structure.

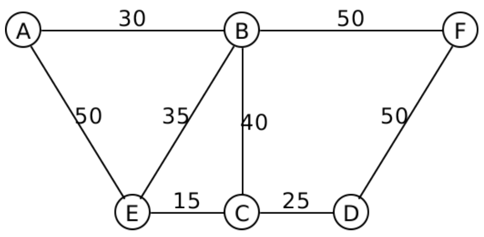
10.3 The Union/Find data structure

The goal of the union/find data structure is to allow us to take a set of items (such as nodes in a graph) and partition them into groups (such as nodes connected by spanning trees) in such a way that we can easily *find* whether two nodes are in the same group, and *union* two disjoint groups together. Intuitively, we accomplish this by *naming* each group by some *representative element*, and then two items can be checked for whether they are in the same group by checking if they have the same representative element.

10.3.1 Example

In class, we represented every node of the graph as a class with a `String` name field. (For this assignment, `String` names will be inconvenient; you will need to come up with some other uniquely-identifying feature of each cell in a maze that can serve the same role as a name.) Then the union-find data structure was a `HashMap<String, String>` that mapped (the name of) each node to (the name of) a node that it is connected to. Initially, every node name is mapped to itself, signifying that every node is its own representative element, or equivalently, that it is not connected to anything.

Recall the example from above:



Our `HashMap` will map every node name to itself:

Representatives as table							**Representatives, visually**					
Node:	A	B	C	D	E	F	A	B	C	D	E	F
Link:	A	B	C	D	E	F						

Spanning tree so far:

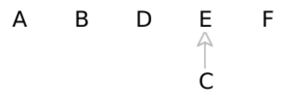
Kruskal’s algorithm begins by sorting the list of edges in the graph by edge weight, from shortest to longest:

```
(E C 15)
(C D 25)
(A B 30)
(B E 35)
(B C 40)
(F D 50)
(A E 50)
(B F 50)
```

When we add edge `(EC 15)`, nodes `E` and `C` are now connected:

Representatives as table **Representatives, visually**

Node:	A	B	C	D	E	F
Link:	A	B	E	D	E	F



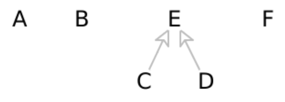
Spanning tree so far:

`(C E)`

We next add edge `CD(25)`. Since `C`’s representative is `E`, and `D`’s representative is `D`, they are currently separate, so adding this edge would not create a cycle. We can therefore *union* them and set `D`’s representative’s representative to `C`’s representative:

Representatives as table **Representatives, visually**

Node:	A	B	C	D	E	F
Link:	A	B	E	E	E	F



Spanning tree so far: `(C D) (C E)`

> ***Do Now!***
>
> Careful! Why must we union the representatives of two nodes, and not the nodes themselves?

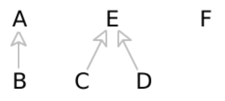
Suppose we goofed, and connected the two nodes we care about directly, rather than connecting their representatives. If one of the nodes already was connected to some other nodes, then changing this connection would *disconnect* it from the nodes it was already unioned with, which would break our data structure and produce an inconsistent state. Our example so far hasn’t yet reached this stage, but it will, soon.

(Note that we could have chosen the other ordering, and set `C`’s representative’s representative to `D`’s representative. This would *mean* the same thing, because both `C` and `D` would be part of the same unified group, but in this case it would create a taller tree, which means it would take longer in the future to find the representative for `C`.)

Next we add edge `AB(30)`. We’ll set `B`’s representative’s representative to `A`’s representative:

Representatives as table **Representatives, visually**

Node:	A	B	C	D	E	F
Link:	A	A	E	E	E	F



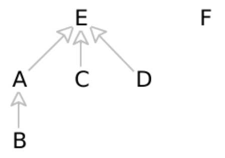
Spanning tree so far: `(A B) (C D) (C E)`

We now have three connected components: Nodes `B` and `A` form one of them, node `F` is a singleton, and nodes `C`, `D`, and `E` are in the third component.

We add edge `BE(35)`. That means we add a link between the representative for `B` (which is `A`) and the representative for node `E` (which is `E`). This time for variety, we’ll choose to connect `A` to `E` instead of the other order:

Representatives as table **Representatives, visually**

Node:	A	B	C	D	E	F
Link:	E	A	E	E	E	F



Spanning tree so far: `(A B) (B E) (C D) (C E)`

> *Do Now!*
>
> We chose arbitrarily to connect `A` to `E`. Draw the result if we had connected `E` to `A` instead.

> *Do Now!*
>
> Again, be careful! Why must we union the representatives of two nodes, and not the nodes themselves?

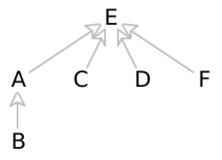
At this point if we had incorrectly connected `B` directly to `E`, then we would *disconnect* it from `A`, and so we would lose track of the fact that `A` is connected to `E` (by way of the edge `BE`).

We still have two components. When we try to add the edge `BC(40)` to the graph, we notice that the representative for node `C` is the same as the representative for the node `B`. Therefore adding this edge would create a cycle, so we discard it.

Finally, we add the edge `FD(50)`. After this, every node has the same representative, and therefore all nodes are connected:

Representatives as table **Representatives, visually**

Node:	A	B	C	D	E	F
Link:	E	A	E	E	E	E



Spanning tree so far: `(A B) (B E) (C D) (C E) (D F)`

10.4 Putting the union/find data structure to work

The full Kruskal's algorithm needs a union/find data structure to handle efficiently connecting components, and also needs a list of the edges used by the algorithm:

```
HashMap<String, String> representatives;
List<Edge> edgesInTree;
List<Edge> worklist = all edges in graph, sorted by edge weights;

initialize every node's representative to itself
While(there's more than one tree)
  Pick the next cheapest edge of the graph: suppose it connects X and Y.
  If find(representatives, X) equals find(representatives, Y):
    discard this edge  // they're already connected
  Else:
    Record this edge in edgesInTree
    union(representatives,
          find(representatives, X),
          find(representatives, Y))
Return the edgesInTree
```

To `find` a representative: if a node name maps to itself, then it is the representative; otherwise, “follow the links” in the representatives map, and recursively look up the representative for the current node’s parent.

> There are additional heuristics for speeding this algorithm up in practice, and they make for a *very* efficient algorithm. Unfortunately, analyzing these heuristics is beyond the scope of this course, but you can look up the “path-compression” heuristic if you are curious.

To `union` two representatives, simply set the value of one representative’s representative to the other.

> ***Do Now!***
>
> Again, why must we only ever union two representatives, and not two arbitrary nodes?

10.5 Breadth- and depth-first search

As we worked through in class, breadth- and depth-first searches are very closely related algorithms. The essential steps of the algorithm are the same; the only difference is whether to use a queue or a stack.

```
HashMap<String, Edge> cameFromEdge;
???<Node> worklist; // A Queue or a Stack, depending on the algorithm

initialize the worklist to contain the starting node
While(the worklist is not empty)
  Node next = the next item from the worklist
  If (next has already been processed)
    discard it
  Else If (next is the target):
    return reconstruct(cameFromEdge, next);
  Else:
    For each neighbor n of next:
      Add n to the worklist
      Record the edge (next->n) in the cameFromEdge map
```

The `cameFromEdge` map is used to record which edge of the graph was used to get from an already-visited node to a not-yet-visited one. This map is used to reconstruct the path from the source to the given target node, simply by following the edges *backward*, from the target node to the node that it came from, and so on back to the source node. Unlike Kruskal’s algorithm, the worklist here is a collection of *nodes* (rather than edges). Like the union/find algorithm, there is a recursive traversal from one node to a previous one, using node names as the keys into the auxiliary map that accumulates the ongoing state of the algorithm.