15-122: Principles of Imperative Computation

Recitation 25 — Josh Zimmerman

Kruskal's algorithm

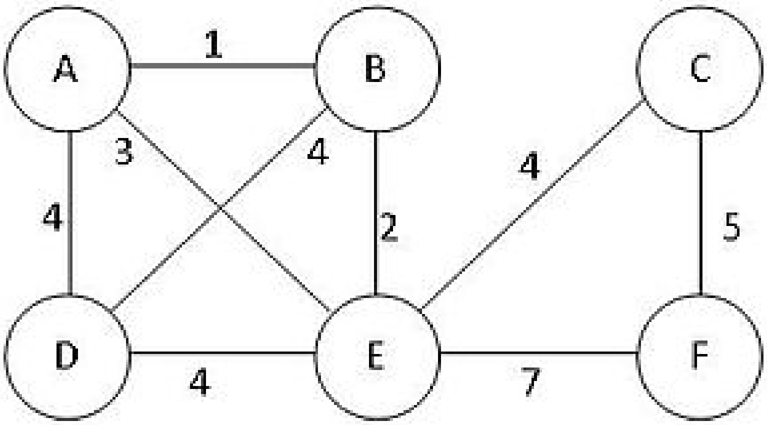
Kruskal's algorithm is an algorithm to find a *minimum weight spanning tree* (often called a *minimum spanning tree*) on a graph.

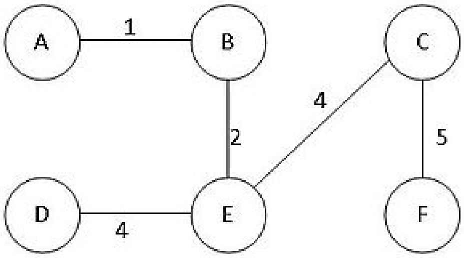
A spanning tree of a graph is a subgraph that is a tree and that connects all vertices of the graph. (Remember that a tree is a connected graph with no cycles.)

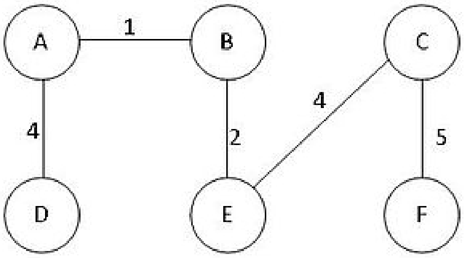
A *minimum* spanning tree (MST) is simply a spanning tree whose edges have the minimum total weight of any spanning tree on the graph.

For any given graph, there may be multiple different MSTs.

For instance, on the graph below we have two different minimum spanning trees, which are shown below it.







Minimum spanning trees are useful in a variety of applications. One classic example is if a phone company is laying cables to connect its customers to the phone network. It wants to connect them all with the minimum possible cost in terms of amount of wire, and doesn't want to use any additional wiring. If the phone company makes a graph where the vertices are houses and the edges are all possible places the company could lay wires, a minimum spanning tree of the graph will give the places they should lay wires to connect everyone to the phone network with minimum possible cost.

Kruskal's algorithm is an algorithm that finds the minimum spanning tree for a graph. The algorithm works as follows:

1. Sort all edges by weight, from smallest weight to largest weight.

2. Go through the edges in order. If adding an edge would not create a cycle in the graph, add it. When we have a minimum spanning tree (when the number of edges we've added is one less than the number of vertices), we're done.

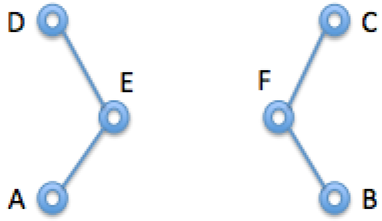
Let's look at a visualization of this algorithm, available at `http://www.cs.usfca.edu/~galles/visualization/Kruskal.html`. (The visulation on the right is the union-find algorithm.)

Union-Find

When talking about Kruskal's algorithm, a key operation is deciding whether or not adding an edge to a graph will create a cycle. At the time, we didn't discuss how to do this.

The basic idea behind union-find is that we keep a single representative vertex of each connected component of the graph.

For instance, in this graph, the vertices A, E, and D would all have the same representative vertex and the vertices B, C, and F would all have the same representative vertex, but those two sets of vertices would have different representative vertices from each other.



Another, more mathematically rigorous way of thinking about this idea is that we define an equivalence relation on vertices such that two vertices u and v are related if and only if there is a path from u to v. When we're considering adding an edge we must first check whether u and v are related.

The union find algorithm uses a canonical representative of an equivalence class (one element of the equivalence class that we can compute from any other) to determine whether adding an edge would form a cycle.

Canonical representatives

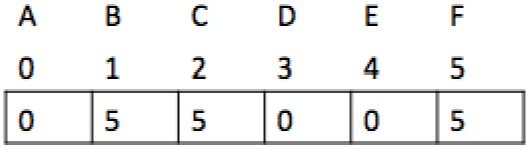
A canonical representative of an equivalence class is just one vertex we choose mostly arbitrarily to represent that equivalence class.

To compute the canonical representative from a specific vertex we can just do an array lookup. The array element that is equal to its own index (`A[i] == i`) is the canonical representative of a class, and if `A[i] != i`, then we can look at `A[A[i]]` to find the canonical representative. It's important to note that this isn't the most efficient way to do union-find, but it's the simplest, so we'll talk about it first.

We initially start with a forest of disjoint vertices, so each one is the canonical representative of its own equivalence class.

When we consider adding an edge, we make sure that the canonical representatives of the two vertices on either end of the edge are not the same. If they're not, we add the edge.

Here's an example of these representative vertices using the graph above. We see that B, C, and F are all in the same class as are A, D, and E, which is what we wanted.



A	B	C	D	E	F
0	1	2	3	4	5
0	5	5	0	0	5

Now comes an interesting question: how do we update this array once we've added another edge?

Intuitively, we need to update every vertex in one equivalence class to have the canonical representative of the other one, so that the canonical representation stays correct. However, this is pretty slow.

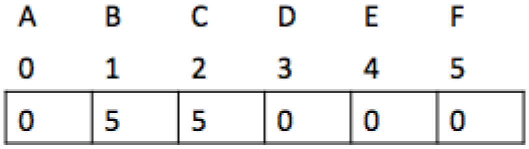
What's the worst case cost of doing this operation, assuming we never update canonical representations of the vertices in the larger equivalence class? (see the exercises section for another copy of this question).

Representatives that "point" to each other.

A more efficient way to merge different equivalence classes is to create a chain that will eventually get us to the unique canonical vertex for the equivalence class.

To do this, we simply update the entry in the table for one equivalence class's canonical vertex to be the index of the canonical vertex in the other equivalence class and keep following indices until we get to an entry where `A[i] == i`.

Here's an example:



A	B	C	D	E	F
0	1	2	3	4	5
0	5	5	0	0	0

To find the canonical representative for the group B is in, we look in the relevant array index repeatedly.

At index 1, we see 5, so we go to index 5. At index 5, we see 0, so we go to index 0. At index 0, we see 0. $0 = 0$, so we've found the canonical representative of the class that B is in.

If we use this representation we can merge multiple equivalence classes without having to update everything in one of them. This allows us to more quickly merge equivalence classes, but it slightly (only slightly!) slows down the amount of time it takes to find the canonical representative of an equivalence class.

Exercises

1. Verify that the relation for union-find (u and v are related if and only if there's a path from u to v) is an equivalence relation by showing that it is reflexive, symmetric and transitive. (You can assume we're working with undirected graphs.)

2. What's the worst case cost of adding an edge, assuming we never update canonical representations of the vertices in the larger equivalence class and that we have n vertices total?