Lecture 26 Notes
Union-Find

15-122: Principles of Imperative Computation (Spring 2016)
Frank Pfenning

1 Introduction

Kruskal's algorithm for minimum weight spanning trees starts with a collection of single-node trees and adds edges until it has constructed a spanning tree. At each step, it must decide if adding the edge under consideration should create a cycle. If so, the edge would not be added to the spanning tree; if not, it will.

In this lecture we will consider an efficient data structure for checking if adding an edge to a partial spanning tree would create a cycle, a so-called *union-find* structure.

This lecture fits our learning goals as follows.

Computational Thinking: We complete our overview of graphs with union-find, a data structure often used when working with graphs.

Algorithms and Data Structures: Union-find is an important data structure beyond graphs as it allows to work efficiently with equivalence classes whenever we can easily designate an element as a canonical representative of the class.

Programming: We leave it to the reader to study the code that implements union-find.

2 Maintaining Equivalence Classes

The basic idea behind the data structure is to maintain equivalence classes of nodes, efficiently. An equivalence class is a set of nodes related by an *equivalence relation*, which must be *reflexive*, *symmetric*, and *transitive*. In our case, this equivalence relation is defined on nodes in a partial spanning

tree, where two nodes are related if there is a path between them. This is reflexive, because from each node u we can reach u by a path of length 0. It is symmetric because we are working with undirected graphs, so if u is connected to w, then w is also connected to u. It is transitive because if there is a path from u to v and one from v to w, then the concatenation is a path from u to w.

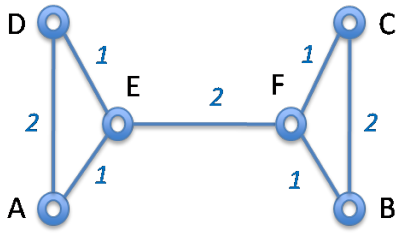
Initially in Kruskal's algorithm, each node is in its own equivalence class. When we connect two trees with an edge, we have to form the *union* of the two equivalence classes, because each node in either of the two trees is now connected to all nodes in both trees.

When we have to decide if adding an edge between two nodes u and w would create a cycle, we have to determine if u and w belong to the same equivalence class. If so, then there is already a path between u and w; adding the edge would create a cycle. If not, then there is not already such a path, and adding the edge would therefore not create a cycle.

The union-find data structure maintains a so-called *canonical representative* of each equivalence class, which can be computed efficiently from any element in the class. We then determine if two nodes u and w are in the same class by computing the canonical representatives from u and w, respectively, and comparing them. If they are equal, they must be in the same class, otherwise they are in two different classes.

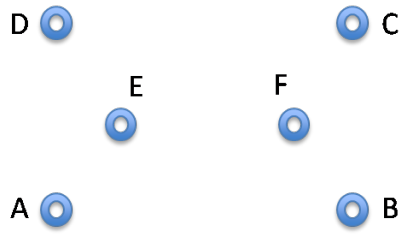
3 An Example

In order to motivate how the union-find data structure works, we consider an example of Kruskal's algorithm. We have the following graph, with the indicated edge weights.



We have to consider the edges in increasing order, so let's fix the order AE, ED, FB, CF, AD, EF, CB. We represent the nodes A–F as integers 0–5 and keep the canonical representation for each node in an array.

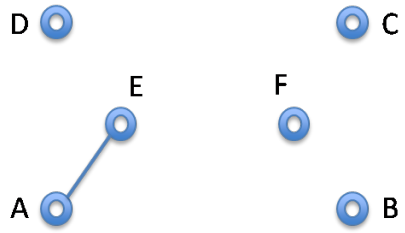
Initially, each node is in its own equivalence class.



In the array, we have the following state

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

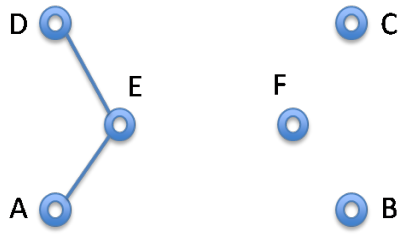
We begin by considering the edge AE, we see that they are in two different equivalence classes because $A[0] = 0$ and $A[4] = 4$, and $0 \neq 4$. This means we have to add an edge between A and E.



In the array of canonical representatives, we either have to set $A[0] = 4$ or $A[4] = 0$, depending on whether we choose 0 or 4 as the representative. Let's assume it's 0. The array then would be the following:

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

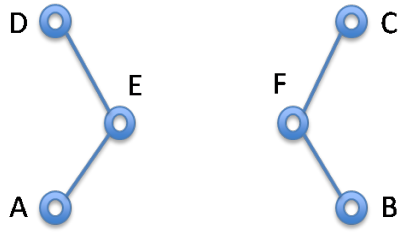
Next we consider ED. Again, this edge should be added because $A[4] = 0 \neq 3 = A[3]$.



If we want to maintain the array it is clearly easier to change $A[3]$ to 0, than to change $A[4]$ and $A[0]$ to 3, since the latter would require two changes. In general, the representative of the larger class should be the representation of the union of two classes.

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

We now combine two more steps, because they are analogous to the above, adding edges FB and CF.



The array:

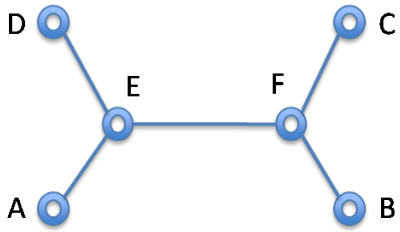
A	B	C	D	E	F
0	1	2	3	4	5
0	5	5	0	0	5

Next was the edge AD. In the array we have that $A[0] = 0 = A[3]$, so A and D belong to the same equivalence class. Adding the edge would create a cycle, so we ignore it and move on.

The next edge to consider is EF. Since $A[4] = 0 \neq 5 = A[5]$ they are in different equivalence classes. The two classes are of equal size, so we have to decide which to make the canonical representative. If it is 0, then we would need to change $A[1]$, $A[2]$, and $A[5]$ all to be 0. This could take up to $n/2$ changes in the array, potentially multiple times at different stages during the algorithm.

In order to avoid this we make one representative "point" to the other (say, setting $A[5] = 0$), but we do not change $A[1]$ and $A[2]$. Now, to find the canonical representative of, say, 1 we first look up $A[1] = 5$. Next we lookup $A[5] = 0$. Then we lookup $A[0] = 0$. Since $A[0] = 0$ we know that we have found a canonical element and stop. In essence, we follow a chain of pointers until we reach a root, which is the canonical representative of the equivalence class and looks like it points to itself.

Taking this step we now have



and a union-find structure which is as follows:

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

At this point we can stop, and we don't even need to consider the last edge BC. That's because we have already added $5 = n - 1$ edges (where n is the number of nodes), so we must have a spanning tree at this stage.

Examining the union-find structure we see that the representative for all nodes is indeed 0, so we have reduced it to one equivalence class and therefore a spanning tree.

In this algorithm we are trying to keep the chain from any node to its representative short. Therefore, when we are applying the union to two classes, we want the one with shorter chains to point to the one with longer chains. In that case, we will only increase the size of the longest chain if both are equal.

In this case, we can show relatively easily that the worst-case complexity of a sequence of n find or union operations is $O(n \log n)$ (see Exercise 1).

4 An Implementation

Instead of developing the implementation here, we refer the reader to the code on the course web site. There is a simple implementation, `unionfind-lin.c`, which does not try to maintain balance, and is therefore linear in the worst case. It does perform a weak form of *path compression*: a postcondition of `ufs_find(eqs, i)` is that `eqs->A[i] == ufs_find(eqs, i)`. That is, before returning the representative for `i`, the implementation stores that representative at `A[i]`. This shortens the search time for subsequent find operations on `i`. (See the exercises for strong path compression.)

This second implementation, `unionfind-log.c` changes the representation we used above slightly. In the above representation, an element i is canonical if $A[i] = i$. In the improved representation, $A[i] = -d$, where d is the maximal length of the chain leading to the representative i. This allows us to make a quick decision how to pick a representative for the union.

Exercises

Exercise 1. *Prove that after n union operations, the longest chain from an element to its representative is $O(\log n)$ if we always take care to have the class with longer chains be the canonical representative of the union. This is without any form of path compression.*

Exercise 2. *Modify the simple implementation in `unionfind-lin.c` so it does strong path compression, which means that on every find operation, every intermediate node will be redirected to point directly to its canonical representative.*

Exercise 3. *Modify the more efficient implementation at `unionfind-log.c` to do path compression. Note that this may require loosening the invariants, since in the straightforward implementation the stored number is only a bound on the longest path and may not be exact (since the path may be compressed).*