Implementing Kruskal's Algorithm

In the previous lecture, we outlined Kruskal's algorithm for finding an MST in a connected, weighted undirected graph $G = (V, E, w)$:

> Initially, let $T \leftarrow \emptyset$ be the empty graph on V.
>
> Examine the edges in E in increasing order of weight (break ties arbitrarily).
> - If an edge connects two unconnected components of T, then add the edge to T.
> - Else, discard the edge and continue.
>
> Terminate when there is only one connected component. (Or, you can continue through all the edges.)

Before we can write a pseudocode implementation of the algorithm, we will need to think about the data structures involved. When building up the subgraph T, we need to somehow keep track of the connected components of T. For our purposes it suffices to know which vertices are in each connected component, so the relevant information is a partition of V. Each time a new edge is added to T, two of the connected components merge. What we need is a disjoint-set data structure.

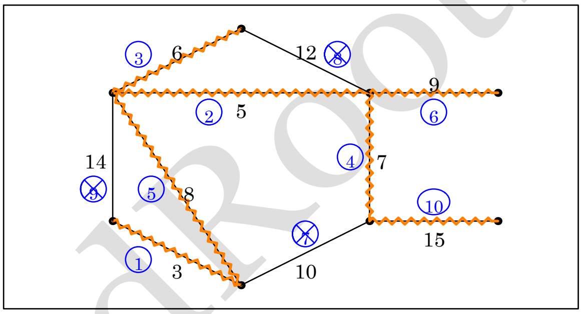


Figure 4.1. Illustration of Kruskal's algorithm.

4.1.1 Disjoint-Set Data Structure

A **disjoint-set data structure** maintains a dynamic collection of pairwise disjoint sets $\mathbf{S} = \{S_1, \ldots, S_r\}$ in which each set S_i has one representative element, rep$[S_i]$. Its supported operations are

- MAKE-SET(u): Create new set containing the single element u.
  - u must not belong to any already existing set
  - of course, u will be the representative element initially
- FIND-SET(u): Return the representative rep$[S_u]$ of the set S_u containing u.
- UNION(u, v): Replace S_u and S_v with $S_u \cup S_v$ in $\mathbf{S}$. Update the representative element.

4.1.2 Implementation of Kruskal's Algorithm

Equipped with a disjoint set data structure, we can implement Kruskal's algorithm as follows:

Algorithm: KRUSKAL-MST(V, E, w)

[1] Actually, we can do better. In line 10, since we have already computed rep$[S_u]$ and rep$[S_v]$, we do not need to call UNION; we need only call WEAK-UNION, an operation which merges two sets assuming that it has been given the correct representative of each set. So we can replace the $E T_{\text{UNION}}$ term with $E T_{\text{WEAK-UNION}}$.

▷ Initialization and setup
$T \leftarrow \emptyset$;
for each vertex $v \in V$ **do**
MAKE-SET(v)
Sort the edges in E into non-decreasing order of weight
▷ Main loop
for each edge $(u, v) \in E$ in non-decreasing order of weight **do**
if FIND-SET(u) $\neq$ FIND-SET(v) **then**
$T \leftarrow T \cup \{(u, v)\}$
UNION(u, v)
return T

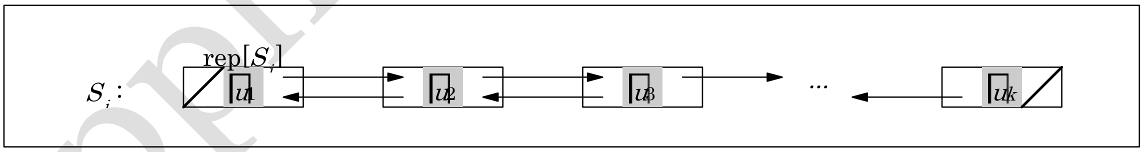
The running time of this algorithm depends on the implementation of the disjoint set data structure we use. If the disjoint set operations have running times $T_{\text{MAKE-SET}}$, T_{UNION} and $T_{\text{FIND-SET}}$, and if we use a good $O(n \lg n)$ sorting algorithm to sort E, then the running time is

$$O(1) + V\,T_{\text{MAKE-SET}} + O(E \lg E) + 2E\,T_{\text{FIND-SET}} + O(E) + E\,T_{\text{UNION}}.$$ [1]

4.1.3 Implementations of Disjoint-Set Data Structure

The two most common implementations of the disjoint-set data structure are (1) a collection of doubly linked lists and (2) a forest of balanced trees. In what follows, n denotes the total number of elements, i.e., $n = |S_1| + \cdots + |S_r|$.

Solution 1: Doubly-linked lists. Represent each set S_i as a doubly-linked list, where each element is equipped with a pointer to its two neighbors, except for the leftmost element which has a "stop" marker on the left and the rightmost element which has a "stop" marker on the right. We'll take the leftmost element of S_i as its representative.

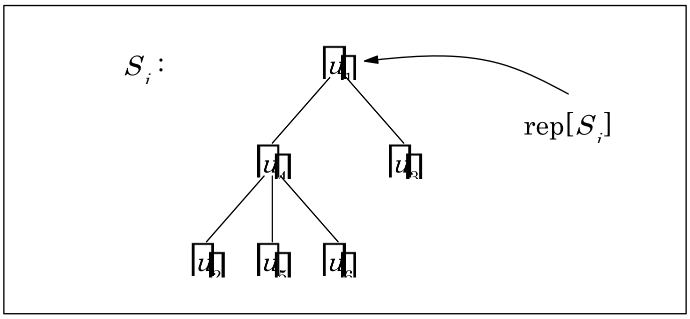


MAKE-SET(u) –	initialize as a lone node	$\Theta(1)$
FIND-SET(u)	–walk left from u until you reach the head	$\Theta(n)$ worst-case
UNION(u, v)	–walk right from u and left from v. Reassign pointers so that the tail of S_u and the head of S_v become neighbors. The representative is updated automatically.	$\Theta(n)$ worst-case

These can be improved upon—there exist better doubly-linked list implementations of the disjoint set data structure.

Solution 2: Forest of balanced trees.[2]

[2] A **rooted tree** is a tree with one distinguished vertex u, called the root. By Proposition 3.1(vi), for each vertex v there exists a unique simple path from u to v. The length of that path is called the **depth** of v. It is common to draw



MAKE-SET(u) – initialize new tree with root node u $\Theta(1)$

FIND-SET(u) –walk up tree from u to root $\Theta(\text{height}) = \Theta(\lg n)$ best-case

UNION(u, v) –change rep[S_v]'s parent to rep[S_u] $O(1) + 2T_{\text{FIND-SET}}$

The forest of balanced trees will be our implementation of choice. With a couple of clever tricks[3], the running times of the operations can be greatly improved: In the worst case, the improved structure has an amortized (average) running time of $\Theta(\alpha(n))$ per operation[4], where $\alpha(n)$ is the inverse Ackermann function, which is technically unbounded but for all practical purposes should be considered bounded.[5] So in essence, each disjoint set operation takes constant time, on average.

In the analysis that follows, we will not use these optimizations. Instead, we will assume that FIND-SET and UNION both run in $\Theta(\lg n)$ time. The asymptotic running time of KRUSKAL-MST is not affected.

As we saw above, the running time of KRUSKAL-MST is

$$\text{Initialize:}\quad O(1) + V \cdot \overbrace{T_{\text{MAKE-SET}}}^{O(1)} + O(E \lg E)$$

$$\text{Loop:}\quad \underbrace{2E\,T_{\text{FIND-SET}}}_{O(\lg V)} + O(E) + \underbrace{E\,T_{\text{UNION}}}_{O(\lg V)} = O(E \lg E) + 2\,O(E \lg V).$$

Since there can only be at most V^2 edges, we have $\lg E \leq 2 \lg V$. Thus the running time of Kruskal's algorithm is $O(E \lg V)$, the same amount of time it would take just to sort the edges.

4.1.4 Safe Choices

Let's philosophize about Kruskal's algorithm a bit. When adding edges to T, we do not worry about whether T is connected until the end. Instead, we worry about making "safe choices." A **safe choice** is a greedy choice which, in addition to being locally optimal, is also part of some globally optimal

rooted trees with the vertices arranged in rows, with the root on top, all vertices of depth 1 on the row below that, etc.

[3] The tricks are called union-by-rank and path compression. For more information, see Lecture 16.

[4] In a 1989 paper, Fredman and Saks proved that $\Theta(\alpha(n))$ is the optimal amortized running time.

[5] For example, if n is small enough that it could be written down by a collective effort of the entire human population before the sun became a red giant star and swallowed the earth, then $\alpha(n) \leq 4$.

solution. In our case, we took great care to make sure that at every stage, there existed some MST T_* such that $T \subseteq T_*$. If T is safe and $T \cup \{(u,v)\}$ is also safe, then we call (u,v) a "safe edge" for T. We have already done the heavy lifting with regard to safe edge choices; the following theorem serves as a partial recap.

Proposition 4.1 (CLRS Theorem 23.1). *Let $G = (V,E,w)$ be a connected, weighted, undirected graph. Suppose A is a subset of some MST T. Suppose $(U, V \setminus U)$ is a cut of G that is respected by A, and that (u,v) is a light edge for this cut. Then (u,v) is a safe edge for A.*

Proof. In the notation of Corollary 3.4, the edge (u_0, v_0) does not lie in A because A respects the cut $(U, V \setminus U)$. Therefore $A \cup \{(u,v)\}$ is a subset of the MST $\left(T \setminus \{(u_0,v_0)\}\right) \cup \{(u,v)\}$. □