Lecture 4

Minimum Spanning Trees II

Supplemental reading in CLRS: Chapter 23; Section 16.2; Chapters 6 and 21

4.1 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.

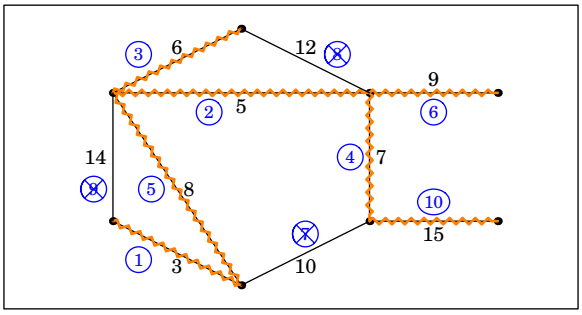


Figure 4.1. Illustration of Kruskal's algorithm.

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  ▷ Initialization and setup
 2  T ← ∅
 3  for each vertex v ∈ V do
 4      MAKE-SET(v)
 5  Sort the edges in E into non-decreasing order of weight
 6  ▷ Main loop
 7  for each edge (u, v) ∈ E in non-decreasing order of weight do
 8      if FIND-SET(u) ≠ FIND-SET(v) then
 9          T ← T ∪ {(u, v)}
10          UNION(u, v)
11  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

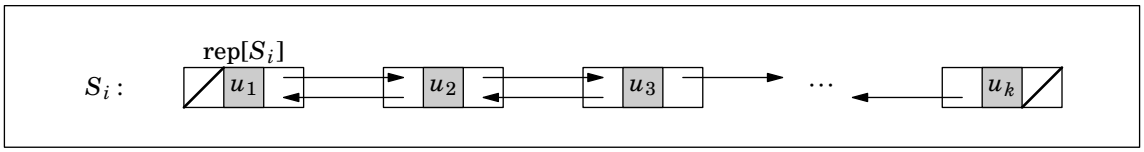
$$O(1) + VT_{\text{MAKE-SET}} + O(E \lg E) + 2ET_{\text{FIND-SET}} + O(E) + ET_{\text{UNION}}.$$ [1]

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.

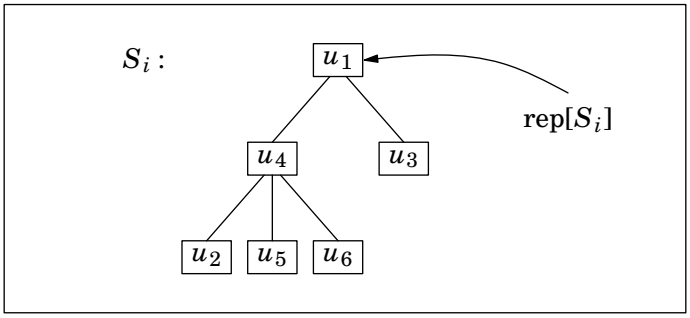
[1] Actually, we can do better. In line 10, since we have already computed rep$[S_u]$ and rep$[S_v]$, we do not need to call UNION; we need only call WEAK-UNION, an operation which merges two sets assuming that it has been given the correct representative of each set. So we can replace the ET_{UNION}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



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]



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.

[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 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 $\alpha(n) \le 4$.

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

$$\begin{array}{ll} \text{Initialize:} & O(1) + V\overbrace{T_{\text{MAKE-SET}}}^{O(1)} + O(E\lg E) \\ \text{Loop:} & 2E\underbrace{T_{\text{FIND-SET}}}_{O(\lg V)} + O(E) + E\underbrace{T_{\text{UNION}}}_{O(\lg V)} \\ \hline & O(E\lg E) + 2O(E\lg V). \end{array}$$

Since there can only be at most V^2 edges, we have $\lg E \le 2\lg V$. 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 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',v') 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 $\big(T \setminus \{(u',v')\}\big) \cup \{(u,v)\}$. □

4.2 Prim's Algorithm

We now present a second MST algorithm: **Prim's algorithm**. Like Kruskal's algorithm, Prim's algorithm depends on a method of determining which greedy choices are safe. The method is to continually enlarge a single connected component by adjoining edges emanating from isolated vertices.[6]

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

```
1 Choose an arbitrary start vertex s
2 C ← {s}
3 T ← ∅
4 while C is not the only connected component of T do
5       Select a light edge (u,v) connecting C to an isolated vertex v
6       T ← T ∪ {(u,v)}
7       C ← C ∪ {v}
8 return T
```

[6] An **isolated** vertex is a vertex which is not connected to any other vertices. Thus, an isolated vertex is the only vertex in its connected component.

Proof of correctness for Prim's algorithm. Again, we use a loop invariant:

> Prior to each iteration, T is a subset of an MST.

- *Initialization.* T has no edges, so trivially it is a subset of an MST.
- *Maintenance.* Suppose $T \subseteq T^*$ where T^* is an MST, and suppose the edge (u,v) gets added to T, where $u \in C$ and v is an isolated vertex. Since (u,v) is a light edge for the cut $(C, V \setminus C)$ which is respected by T, it follows by Proposition 4.1 that (u,v) is a safe edge for T. Thus $T \cup \{(u,v)\}$ is a subset of an MST.
- *Termination.* At termination, C is the only connected component of T, so by Proposition 3.1(v), T has at least $|V|-1$ edges. Since T is also a subset of an MST, it follows that T has exactly $|V|-1$ edges and is an MST. □

The tricky part of Kruskal's algorithm was keeping track of the connected components of T. In Prim's algorithm this is easy: except for the special component C, all components are isolated vertices. The tricky part of Prim's algorithm is efficiently keeping track of which edge is lightest among those which join C to a new isolated vertex. This task is typically accomplished with a data structure called a min-priority queue.

4.2.1 Min-Priority Queue

A **min-priority queue** is a data structure representing a collection of elements which supports the following operations:

> INSERT(Q,x) – Inserts element x into the set of elements Q
>
> MINIMUM(Q) – Returns the element of Q with the smallest key
>
> EXTRACT-MIN(Q) – Removes and returns the element with the smallest key
>
> DECREASE-KEY(Q,x,k) – Decreases the value of x's key to new value k.

With this data structure, Prim's algorithm could be implemented as follows:

Algorithm: PRIM-MST(G, s)

```
1  T ← ∅
2  for each u ∈ G.V do
3      u.key ← ∞ ▷ initialize all edges to "very heavy"
4      ▷ The component C will be a tree rooted at s. Once a vertex u gets added to C, u.π
       will be a pointer to its parent in the tree.
5      u.π ← NIL
6  s.key ← 0 ▷ this ensures that s will be the first vertex we pick
7  Let Q ← G.V be a min-priority queue
8  while Q ≠ ∅ do
9      u ← EXTRACT-MIN(Q)
10     if u.π ≠ NIL then
11         T ← T ∪ {(u, u.π)}
12     ▷ We assume that G is presented in the adjacency-list format*
13     for each v ∈ G.adj[u] do
14         if v ∈ Q and w(u, v) < v.key then
15             v.π ← u
16             v.key ← w(u, v) ▷ using DECREASE-KEY
17 return T
```

*For more information about ways to represent graphs on computers, see §22.1 of CLRS.

4.2.2 Running Time of Prim's Algorithm

Lines 1 through 6 clearly take $O(V)$ time. Line 7 takes $T_{\text{BUILD-QUEUE}}(V)$, where $T_{\text{BUILD-QUEUE}}(n)$ is the amount of time required to build a min-priority queue from an array of n elements. Within the "while" loop of line 8, EXTRACT-MIN gets called $|V|$ times, and the instructions in lines 14–16 are run a total of $O(E)$ times. Thus the running time of Prim's algorithm is

$$O(V) + T_{\text{BUILD-QUEUE}}(V) + V T_{\text{EXTRACT-MIN}} + O(E) T_{\text{DECREASE-KEY}}.$$

Exercise 4.1. *How can we structure our implementation so that line 14 runs in $O(1)$ time?*

Let's take a look at the performance of PRIM-MST under some implementations of the min-priority queue, in increasing order of efficiency:

Q	$T_{\text{BUILD-QUEUE}}(n)$	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Runtime of PRIM-MST
Array	$O(n)$	$O(n)$	$O(1)$	$O(V^2)$
Binary Heap	$O(n)$	$O(\lg n)$	$O(\lg n)$	$O(E \lg V)$
Fibonacci Heap	$O(n)$	$\underbrace{O(\lg n)}_{\text{amortized}}$	$\underbrace{O(1)}_{\text{amortized}}$	$O(E + V \lg V)$

Thus, for a dense graph (a graph with $\Theta(V^2)$ edges), Prim's algorithm with a Fibonacci heap outperforms Kruskal's algorithm. The best known MST algorithm to date is a randomized algorithm with $\Theta(V + E)$ expected running time, introduced by Karger, Klein and Tarjan in 1995.

4.3 Greedy Strategies

General approach:

1. Structure the problem so that we make a choice and are left with one subproblem to solve.
2. Make a greedy choice and then prove that there exists an optimal solution to the original problem which makes the same greedy choice ("safe choice").
3. Demonstrate optimal substructure.
    - After making the greedy choice, combine with the optimal solution of the remaining subproblem, giving an optimal solution to the original problem.

Note that this sounds a lot like dynamic programming. Let's now examine the key properties of greedy algorithms in comparison to those of dynamic programming.

1) *Greedy Choice Property – Locally optimal solution leads to globally optimal solution.* Each greedy local choice is made independently of the solution to the subproblem. (E.g.: Kruskal's and Prim's algorithms can choose a safe edge without having examined the full problem.) In dynamic programming, the local choice depends on the solution to the subproblem—it is a bottom-up solution.

2) *Optimal Substructure – The optimal solution to a problem contains optimal solutions to subproblems.* Both greedy strategies and dynamic programming exploit (or rely on) optimal substructures. Prim's algorithm produces (optimal) MSTs on subsets of V on the way to finding the full MST.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012