Today we will discuss minimum spanning trees (MSTs) in weighted, undirected graphs.

1 The problem

Suppose we have a set of locations $V = \{v_1, v_2, \ldots, v_n\}$ and want to build a communication network on top of them. Further, suppose that for certain pairs (v_i, v_j) we may build a direct link at a positive cost (weight) $weight(v_i, v_j) > 0$. Then we may represent our input as the undirected weighted graph $G(V, E, weight)$.

Given the input graph $G(V, E, weight)$, we want to build our communication network over the vertices of G so that the network is **connected**, that is, there is a path between every pair of nodes, and it is built as cheaply as possible. Therefore, our output should be a subset $T \subseteq E$ such that

- the graph (V, T) is connected and
- $\sum_{e \in T} weight(e)$ is minimal.

Recall that in undirected graphs, a tree is a connected graph with no cycles. Then our output graph (V, T) is a tree: if it had a cycle, we could remove any edge from the cycle and still have a connected graph with smaller total cost. A tree that spans the vertices of the graph is called a **spanning** tree. So our problem is equivalent to finding the minimum weight (or cost) spanning tree, which we will henceforth just call minimum spanning tree (MST).

2 A generic algorithm for finding MSTs

A first attempt to solve this problem would be to list all possible spanning trees in a graph and output the one with the smallest weight. This approach would be feasible if the number of spanning trees in a graph were small. However, even a simple cycle on n vertices has n distinct spanning trees (*why?*) while a more complex graph like the complete graph on n vertices (K_n) has n^{n-2} spanning trees as shown in the Appendix. So exhaustive search would not yield an answer for many input graphs.

However we do not need to list every spanning tree to find an MST. In fact, lots of **greedy** algorithms work in the case of MSTs. The underlying idea is to **repeatedly add edges to a partial solution**. We will guarantee after the inclusion of each new edge that the set of selected edges forms a subset of the minimum spanning tree T. This is possible thanks to the following crucial fact.

Fact 1 (Cut property) *Assume that all edge weights are distinct. Let $S \subset V$ ($S \neq \emptyset$), and let edge $e = (u, v)$ be the minimum-weight edge with one endpoint in S and the other in $V - S$. Then every minimum spanning tree contains e.*

Proof. Let T' be a spanning tree that does not contain $e = \{u, v\}$. We will show that T' is not a minimum spanning tree by exchanging one edge of T' for the cheaper edge e.

Since T' is a spanning tree, there must be some other path P in T' from u to v. Starting at v, suppose we follow the vertices of P. Since (u, v) crosses from S to $V - S$, there must be some first vertex v' on P that is in $V - S$. Let u' be the last vertex before it in S. Then edge $e' = (u', v') \in T'$ **and** e' crosses between S and $V - S$.

Now exchange e with e' to obtain the set of edges

$$T = T' + \{e\} - \{e'\}.$$

Then T is a spanning tree because

- it is connected: any path in T' that used edge $e' = (u', v')$ can now be rerouted in T to use the portion of the path P from u' to u, then (u, v), then the portion of P from v to v'.
- T has no cycle (*why?*).

Since both e' and e cross between S and $V - S$ but e is the lightest edge with this property, $weight(e) < weight(e')$. It follows that $weight(T) < weight(T')$. □

The Cut property basically states that we can construct our tree **greedily**: simply take the lightest edge across two regions not yet connected. Therefore, we have the following generic MST algorithm

Generic MST($G = (V, E, weight)$)
  $A = \emptyset$
  while $|A| < n - 1$ **do**
    Pick $S \subseteq V$ such that no edge in A crosses between S and $V - S$.
    Let $e \in E$ be a lightest edge that crosses between S and $V - S$.
    $A = A \cup \{e\}$
  end while

Different algorithms arise by different ways of picking the set S at every step.

Assumption 1 *For all our algorithms, we will assume that the edge weights in the input graph are distinct.*

We will see in Section 4.4 how to eliminate this assumption.

3 Prim's algorithm

During the execution of Prim's algorithm, the set A maintains the edges of a partial MST. Specifically, we start with a root node s and **greedily** grow a tree outward from s. At every step, we simply add to the partial tree the node that can be attached as cheaply as possible. Thus the algorithm grows a **single** tree, adding a new vertex and edge at each iteration, until the MST is complete. More formally:

1. Maintain a set $S \subseteq V$ on which a spanning tree has been constructed so far.
2. In each iteration, grow S by one vertex v, adding the vertex $v \in V - S$ that minimizes the attachment cost:
$$\min_{e=(u,v):u \in S} weight(e)$$
and include edge e in the ST.

Prim's algorithm reminds us of Dijkstra's algorithm though it is even simpler to specify. See Figure 1 for an example.

3.1 Correctness of Prim's algorithm

The correctness of Prim's algorithm follows directly from the Cut property: at every iteration, an edge (u, v) is added such that $u \in S$ (S is the set of vertices on which a partial MST has been constructed), $v \in V - S$ and (u, v) is the lightest edge that crosses between S and $V - S$. Hence the Cut property guarantees that the algorithm only adds edges belonging to every MST.

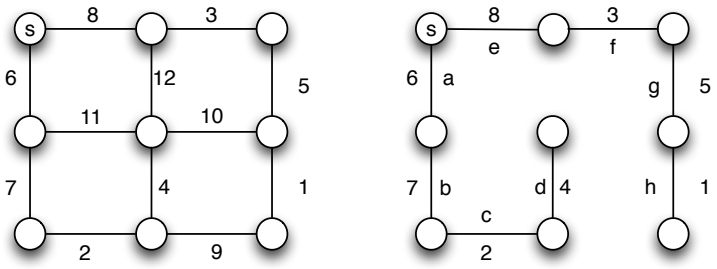

Figure 1: A graph G and the MST computed by Prim from node s. Letters a to h indicate the order in which the edges of the MST are added.

3.2 Prim's algorithm: Implementation

As in Dijkstra's algorithm, we will use a priority queue Q implemented as a binary min-heap to store the vertices in $V - S$. Initially, Q contains all the vertices and S is empty.

We will maintain two arrays: (i) *distance*: for all $v \in V - S$, $distance[v]$ maintains the weight of the lightest edge between v and any vertex in S (in Dijkstra, $distance[v]$ maintained a conservative overestimate of the distance of vertex v from vertex s); and (ii) *previous*, so we can reconstruct the edges of the MST from the pairs $(v, previous[v])$.

Prim$(G = (V, E, weight), s)$
  for $u \in V$ **do**
    $distance[v] = \infty; previous[v] = NIL$
  end for
  $distance[s] = 0$
  $Q = \{V; distance\}$ //initially the queue contains all $u \in V$ using $distance[u]$ as key
  $S = \emptyset$
  while $Q \neq \emptyset$ **do**
    $u =$ ExtractMin(Q)
    $S = S \cup \{u\}$
    for $(u, v) \in E$ and $v \in V - S$ **do**
      if $distance[v] > weight(u, v)$ **then**
        $distance[v] = weight(u, v)$
        $previous[v] = u$
      end if
    end for
  end while

Similarly to Dijkstra's algorithm, the running time of Prim's algorithm is $O(n \log n + m \log n) = O(m \log n)$. If the priority queue is implemented as a Fibonacci heap, then the algorithm requires $O(n \log n + m)$ amortized time (we will discuss amortized analysis in Section 5).

4 Kruskal's algorithm

Kruskal's algorithm works differently: instead of growing a single tree, it adds to A the **lightest** edge possible at every step. Specifically, it first sorts the edges by increasing weight. Then it examines the edges iteratively in sorted order; if the edge does not create a cycle with already included edges, it is added to A.

Another way to view how Kruskal's algorithm operates, is the following.

1. Start with $A = \emptyset$ and each vertex being a trivial tree with no edges.
2. Each edge in G is examined **in order**:
   - If its endpoints are in the same tree then the edge is discarded.
   - Otherwise, it is included in A and this causes the two trees at each endpoint to merge into a single tree.

See Figure 2 for the execution of Kruskal's algorithm on the left graph of Figure 1.

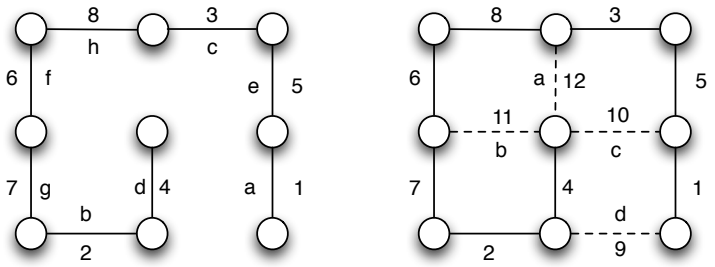

Figure 2: The MST of Kruskal's algorithm for the left graph of Figure 1. The MST on the right is produced by the algorithm in Section 4.3 for the same graph (here, letters denote the order in which edges are deleted from G.

4.1 Correctness of Kruskal's algorithm

Let (u, v) be the edge added at any iteration. Consider the set S of nodes that have a path to u by edges in A just before (u, v) is added. Obviously $u \in S$ but $v \notin S$ because then inclusion of (u, v) in A would create a cycle. Also, (u, v) must be the first edge between S and $V - S$ encountered so far: if such an edge had been encountered before, it would have been included in A previously since there was no edge between S and $V - S$ then, so its inclusion would not cause a cycle. Thus (u, v) is the lightest edge with one endpoint in S and another in $V - S$ and, by the Cut Property, belongs to every MST.

4.2 Kruskal's implementation

Kruskal's algorithm maintains a **forest** of trees at all times, starting from n empty trees. To implement Kruskal's algorithm given a forest of trees, we need a data structure that allows us to

1. given two vertices, decide whether they belong to the same tree or not.
2. update the data structure to reflect the merging of two trees into a single tree.

Thus we want our data structure to **maintain a collection of disjoint sets** and support the following three operations:

1. MakeSet(u): Given an element u, create a new set containing only u (for example, make u a trivial tree with no edges).
2. Find(u): Given an element u, find which set (tree) u belongs to.
3. Union(u, v): Merge the set containing u and the set containing v into a single set.

Assuming we could implement such a data structure so that Makeset(), Find() and Union() have worst-case times $O(1)$, $O(\log n)$ and $O(\log n)$ respectively, the pseudocode and running time analysis for Kruskal's algorithm follow.

Kruskal($G = (V, E, weight)$)
  $A = \emptyset$
  Sort(E) by $weight$
  for $u \in V$ **do** MakeSet(u)
  end for
  for $(u, v) \in E$ (by increasing $weight$) **do**
    if Find(u) $\neq$ Find(v) **then**
      $A = A \cup \{(u, v)\}$
      Union(u, v)
    end if
  end for

The running time of the algorithm is $O(m \log n)$ since

- Sorting requires $O(m \log m) = O(m \log n)$.
- Set operations require $O(n) + 2mO(\log n) + O(n \log n) = O(m \log n)$.

4.3 More algorithms for MSTs

Before we discuss the Union-Find data structure in more detail, we note that there are many natural greedy algorithms for finding MSTs besides Prim 's and Kruskal's algorithms. To see how such algorithms might be generated, we state a property analogous to the Cut property that tells us when it is safe to **not** include an edge in an MST.

Fact 2 (The Cycle Property) *Assume that all edge costs are distinct. Let C be any cycle in T, and let edge (u, v) be the heaviest edge in C. Then e does not belong to any MST of G.*

The proof is similar to the proof of the Cut property (*exercise*).
Remark 1: We can use Fact 2 to design yet another algorithm for finding MSTs: start with a full graph and repeatedly delete edges in order of decreasing costs, so long as the graph does not become disconnected (see Figure 2 for an example). We will not discuss implementation of this algorithm (even Kruskal's algorithm will be fairly subtle to implement).
Remark 2: Basically, any algorithm that repeatedly uses the Cut property to add an edge when justified and the Cycle property to repeatedly delete an edge when justified correctly builds an MST.

4.4 Eliminating the assumption of non-distinct edge weights

Suppose some edges have equal weights. We can show that the algorithms discussed in the previous sections are still optimal by slightly perturbing all edge weights by different, tiny amounts, so that all edge weights are now distinct. Edges whose weights differed before still have the same relative order (since perturbations are tiny). Essentially, the perturbations serve as tie-breakers to resolve comparisons between equal edge weights.

5 Amortized analysis for the Union–Find data structure

The Union-Find data structure maintains disjoint sets, such as the trees in Kruskal's algorithms or the connected components of a graph. The operations supported by this data structure are: MakeSet(u) (see

Section 4.2); Find(u), which returns the **name** of the set where u belongs (the name of a set is one of the elements contained in the set); and Union(u, v), which merges the set of u and the set of v into a single set, **if** Find(u) $\neq$ Find(v).

5.1 Implementing Union-Find with arrays

We could implement the Union-Find data structure by maintaining an array Set such that for all $u \in V$

$$Set[u] = i \text{ , if } u \text{ belongs to set } i.$$

For example, $Set[u]$ may indicate the tree where u belongs. Note that

- MakeSet(u) takes $O(1)$ time and corresponds to initializing $Set[u] = u$, for all $u \in V$.
- Find(u) corresponds to accessing $Set[u]$ and takes $O(1)$ time.
- Union(u, v) corresponds to updating the values in the array Set for all elements in the set of u and all elements in the set of v to be the same. This could take $O(n)$ time: go over all elements in the array and update them.

We can implement two straightforward optimizations to the above data structure.

- First, maintain the elements in each set explicitly so we don't have to look through the whole array for elements that need updating.
- Choose the name for the union of the sets to be the name of the larger set. This way we only have to update the elements in the smaller set. To this end, maintain an additional array $size$ of size n that keeps the sizes of all sets.

The worst-case running time of Union(u, v) is still $O(n)$: suppose we need perform Union(u, v) on large sets of u, v, whose sizes are constant fractions of n. However such bad cases can't happen too often: every time two large sets are merged, the resulting set is even bigger. This motivates us to bound the **total** running time of a sequence of k Union operations instead of bounding the **worst-case** running time of a single Union operation. This kind of analysis is called **amortized** analysis and considers the cost of the algorithm over a sequence of steps, instead of considering the cost of a single operation.

Fact 3 *Using the array implementation, a sequence of k Union operations takes $O(k \log k)$ time.*

Proof. We will now investigate how our optimizations affect the time (number of updates) required for k Union operations.

Originally all n elements are their own singleton sets. Suppose we perform k Union operations. Then the maximum number of original elements that can be touched (that is, be visited, *not updated*) is $2k$: each Union touches at most two of the original elements.

Now consider a fixed element u. Every time u changes name, it is because it is merged with a set that is larger than its own set, hence the size of its set **at least doubles**. So u starts at a set of size 1 and after a sequence of k Union operations ends in a set whose maximum size is at most $2k$ (because this is the maximum number of elements that may be visited after k Union operations). Since every time u changes name, the size of the new set it belongs to is at least twice the size of the set it started at, the size of the set of u may be doubled at most $\log_2 2k$ times. Since at most $2k$ elements are involved in these k operations, a sequence of k Union operations takes at most $O(k \log k)$ time. □

Exercise: show that there is a sequence of k Union operations that requires $\Theta(k \log k)$ time (that is, $\Theta(k \log k)$ updates).

Hence using this array-based implementation of Union-Find for Kruskal's algorithm, the average time required for the sequence of n Union operations is $O(n \log n)$. The overall running time of the algorithm is $O(m \log n)$.

5.2 A pointer-based implementation for Union–Find

We will now improve the **worst-case** time of Union(u, v) (note that the analysis in previous section only offered a bound on the **total** time required by k Union operations). This can be accomplished by modifying the data structure as follows. We still name a set after one of its own elements (vertices). Each element u will now be contained in a record with an associated pointer to the name of the set (that is, the corresponding element) that contains u. Also, we keep the optimizations from before, that is, the array $size$ and the rule of assigning the name of the larger set to the smaller when performing a Union operation. So now the operations are as follows.

1. MakeSet(u): initializes a record for an element $u \in V$ with a pointer that points to itself to denote that u is in its own set ($O(1)$ time).

2. Union(u, v): first, note that a Union operation is always preceeded by a comparison "Find(x) =Find(y)?" that checks whether the two elements x and y belong to different sets. So suppose u is the name returned for the set where x belongs and v is the name for the set where y belongs; then, assuming $u \neq v$, we will merge the set named after u with the set named after v. If $size[u] < size[v]$, we simply make u point to v, otherwise, v points to u; this can be accomplished in $O(1)$ time. See Figure 3 for an example.

3. Find(x): to find the name of the set where element x belongs, we now need $\Omega(1)$ time. We need to follow a sequence of pointers until we find an element r that points to itself. Then r is the name of the set where x belongs (otherwise, r itself would point to some other element). How long does it take to find r (equivalently, how long does Find(x) take)? It takes the time required to traverse the sequence of pointers from x to r. In turn, the length of this sequence represents the number of times that the set where x belongs changed names from the start of the algorithm.

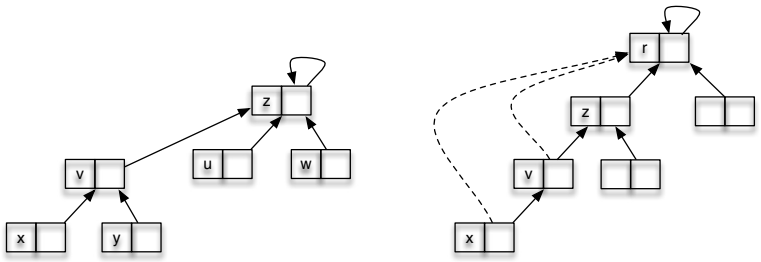

Figure 3: On the left, a sequence of Union operations: first, $\{x\}$ merged to $\{v\}$, then $\{y\}$ merged to $\{v, x\}$, then $\{v, x, y\}$ merged to $\{z, u, w\}$. On the right, an example of path compression: Find(x) updates the pointers of all nodes on the path from x to r to point to r.

Fact 4 *Using the pointer-based implementation, one Find operation takes $O(\log n)$ time.*

Proof. The time required by Find(x) equals the number of times the set that contains x changed names during the execution of the algorithm. Again, in a Union operation, the size of the set where x belongs at least doubles if its name changes. Since the set containing x starts at size 1 and cannot get larger than n, its name changes at most $\log_2 n$ times. □

So, using the pointer-based implementation, we have a worst case bound for all single operations of the data structure, which again yields a running time of $O(m \log n)$ for Kruskal's algorithm.

We can further improve this implementation —however this will not improve the $O(m \log n)$ bound for the running time of Kruskal's algorithm unless the input is sorted or can be sorted faster.

To highlight the main idea of the improvement (but not the analysis!), consider a bad setting for the Find operation where x is an element that requires $\log_2 n$ time to find which set it belongs to, and the algorithm repeatedly calls Find(x). Every call after the first one simply wastes time since after the first

Find(x) we know the name of the set, say r, where x belongs. Further, for every node v on the path from x to r, after the first Find(x), we also know the name of the set where v belongs : it is simply r. This suggests the following optimization procedure, called **path compression**: for every node v on the path followed for the Find(x) operation, reset the (potentially outdated) pointer of v to point to the current name of the set, that is, r (see Figure 3 for an example).

How does path compression affect the running time for a Find operation? For some elements x, Find(x) may now take $2\log_2 n$ time: after finding the name for x, we go up through the same path and update the pointers for all intermediate elements. (Of course, this is just a constant factor increase compared to $\log_2 n$). Similarly to the array-based implementation analysis, the real gain comes from considering subsequent calls to Find and for this, amortized analysis is used again. However this analysis (due to Tarjan) is quite complex so we will just state that to bound the **total** time of $2m$ Find operations (rather than the worst-case time for any of them), we need an amount of time that is extremely close to linear, specifically

$$O(m\alpha(m, n)),$$

where $\alpha(m, n)$ is a function that for all practical purposes is at most 3 or 4 (and is similar to the inverse Ackermann function). Hence in the improved Union-Find data structure with path compression, the sequence of $2m$ Find and $n - 1$ Union operations requires almost $O(m)$ amortized time.

5.3 Appendix: How many spanning trees does a complete graph on n vertices have?

We remind that a directed tree is a rooted graph that has a simple path from the root to every vertex in the graph. A tree has $n - 1$ edges. We will show that the number of spanning trees T_n in the complete graph on n vertices is given by

$$T_n = n^{n-2}.$$

This fact is known as Cayley's formula. There are alternative proofs for Cayley's formula, but here we will use the technique of double counting. That is, we will compute a quantity in two different ways to derive an expression for T_n.

The quantity we will compute in two different ways is the number ν different sequences of directed edges that can be added to an empty graph on n vertices to yield a rooted tree.
1. One way to think about ν explicitly involves T_n. That is,

1. start with a spanning tree on this graph (T_n choices);
2. pick a root for the tree (n choices);
3. given the root, the directionalities of the edges are fully determined (recall that a rooted tree must have a path from the root to every vertex). So now we have $n - 1$ directed edges to insert in any order in our graph and there are $(n - 1)!$ ways to order them.
4. So, in total, there are $T_n \cdot n \cdot (n - 1)!$ different sequences of directed edges to add in a graph so as to form a directed rooted tree.

2. Another way to compute ν is by adding edges one at a time and counting our available choices at every step.
In the beginning our graph is a forest of n rooted trees that are empty (that is, contain no edges).

- For the 1st edge: we may pick **any** of the n vertices as the tail, and direct the edge to **any** of the $n - 1$ remaining roots: thus there are $n(n - 1)$ choices for the first edge.

- For the 2nd edge: the graph is now a forest with $n-1$ rooted trees. We may pick **any** of the n vertices as the tail of the edge, and add a directed edge to the **root of any tree** (so that the resulting graph remains a **rooted** tree) except for the root of the tree where the tail belongs (so as to guarantee no cycles). This leaves us with $n-2$ choices for the head of the edge. So in total, there are $n(n-2)$ choices for the second edge.
- For the k-th edge: the reasoning is entirely similar. After addition of the $(k-1)$-st edge, there are $n-(k-1)$ rooted trees in the forest (by construction, every edge we add reduces the number of trees in the graph by 1). The k-th edge may leave any vertex (so n choices for the tail of the edge), but there are only $n-(k-1)-1$ choices for the head (*why?*). So there are $n(n-k)$ choices for the k-the edge.
- Finally, when we add the $n-1$-st edge, there are only 2 rooted trees in the forest hence $n \cdot 1$ choices for this last edge (*why?*).
- In total, there are $n^{n-1}(n-1)!$ ways to add the edges.

Equating the two expressions we obtain: $T_n = n^{n-2}$.
For arbitrary graphs, the number of spanning trees is computable in polynomial time.