Minimum Spanning Trees

(CLRS 23)

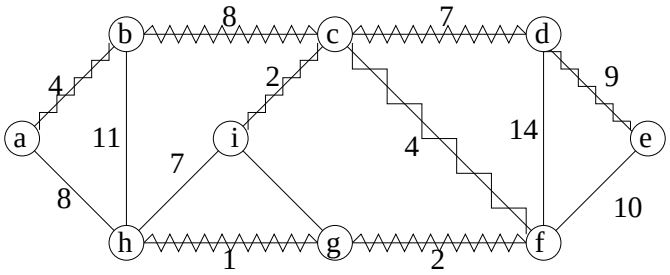
1 The problem

Recall the definition of a *spanning tree*: Given connected, undirected graph $G = (V, E)$, a subset of edges of G such that they connect all vertices in G and form no cycles is called a *spanning tree* (ST) of G. Any undirected, connected graph has a spanning tree. Actually, a graph usually has many spanning trees.

Question: How would you go about finding a ST of a given graph G? Answer: Run BFS or DFS; the resulting BFS- or DFS-tree are spanning trees of G.

The *minimum spanning tree* (MST) problem is the following: Given a connected, undirected, weighted graph G (each edge (u, v) has weight $w(u, v)$), find a spanning tree T of minimum weight: $w(T) = \sum_{(u,v)\in T} w(u, v)$. In other words, of all spanning trees of G, we want one of minimum total weights. We use *minimum spanning tree* as short for *minimum weight spanning tree*).

Example:



- Weight of MST is $4 + 8 + 7 + 9 + 2 + 4 + 1 + 2 = 37$
- Note: MST is not unique: e.g. (b, c) can be exchanged with (a, h)

The MST problem has many applications: For example, think about connecting cities with minimal amount of wire or roads (cities are vertices, weight of edges are distances between city pairs). Also, network design in general (telephones, electrical, TV cable, road networks).

2 MST algorithms

The two classical algorithms for computing MST are Kruskal's and Prim's algorithms. They both have same running time complexity, and they are both greedy algorithms, but they use complementary approaches:

- Kruskal's algorithms: Start with a T consisting of all vertices but no edges. Consider the edges in G in increasing order of weight. Add edge e to T unless doing so would create a cycle.

  This corresponds to growing the tree from a forest of V disconnected vertices. Add one edge at a time, joining two trees in the forest together. Each join reduces the number of connected components in the forest by one.

- Prim's algorithm: Start with an empyt T, and an arbitrary vertex v. Grow T one edge at a time: at each step, add to T the edges of minimum weight that has exactly one endpoint in T.

  This grows a tree one edge at a time. The current tree is always connected but only covers a part of the vertices.

3 Correctness

The correctness of both MST algorithms follows from the following general theorem:

The cut theorem: Let S be a set of vertices in a graph $G = (V, E)$, and consider all edges with one endpoint in S and the other one in $V - S$.[1] Let $e = (u, v)$ be an edge of minimum weight among all edges that cross the cut. Then there exists a MST that contains e.

Proof: Let T be a MST of G. If T contains edge $e = (u, v)$, then we are done. If T does not contain $e = (u, v)$, then:

- Because T is a tree, there is a unique path in T between any two nodes and in particular between u, v.
- Because u is on one side of the cut and v is on the other, this path must contain at least an edge $(x, y) \in T$ crossing the cut.
- When adding $e = (u, v)$ to T, this path and e form a cycle.
- If we remove edge (x, y) from T and add e instead, then:
- $T' = T - (x, y) + (u, v)$ is a spanning tree; and
- T' must have same weight as T since $w(u, v) \leq w(x, y)$
  $\Downarrow$
  Thus T' is also a MST, and thus there is a MST containing T and e.

[1] The partition of V into $(S, V - S)$ is referred to as a *cut*, and the edges with one one endpoint in S and the other one in $V - S$ are called edges that *cross the cut*.

The Cut Theorem allows us to describe a very abstract greedy algorithm for MST:

$T = \emptyset$
While $|T| \leq |V| - 1$ DO

Find cut S respecting T

Find minimal edge e crossing S

$T = T \cup \{e\}$

Both Prim's and Kruskal's algorithms follow this abstract algorithm and their correctness follows from the cut theorem by chosing a suitable cut:

- Prim's: The cut is $(T, V - T)$, that is, the current tree on one side, and the remaining vertices on the other. At each point the algorithm adds the minimum edge crossing the cut, which, by the cut theorem, we know must be part of an MST.
- Kruskal's: When adding edge (u, v) chose a cut that has the vertices in the set of u on one side and the vertices in the set of v on the other. Edge e is the smallest edge that crosses the cut and thus by the cut theorem, we know must be part of an MST.

4 PRIM's algorithm

GeneraL IDEA:

- Start with spanning tree containing arbitrary vertex r and no edges
- Grow spanning tree by repeatedly adding minimal weight edge connecting vertex in current spanning tree with a vertex not in the tree

Implementation:

- To find minimal edge connected to current tree we maintain a priority queue on vertices not in the tree. The key/priority of a vertex v is the weight of minimal weight edge connecting v to the tree. (We maintain pointer from adjacency list of v to v in the priority queue).
- For each node v maintain $visit(v)$ such that edge $(v, visit(v))$ is the best edge connecting v to the current tree.

PRIM(r)
/* initialize */
For each vertex $u \in V, u \neq r$ DO: INSERT(PQ, u, ∞)
INSERT$(PQ, r, 0)$, $visit(r) = NULL$
/* main loop */
WHILE PQ not empty DO

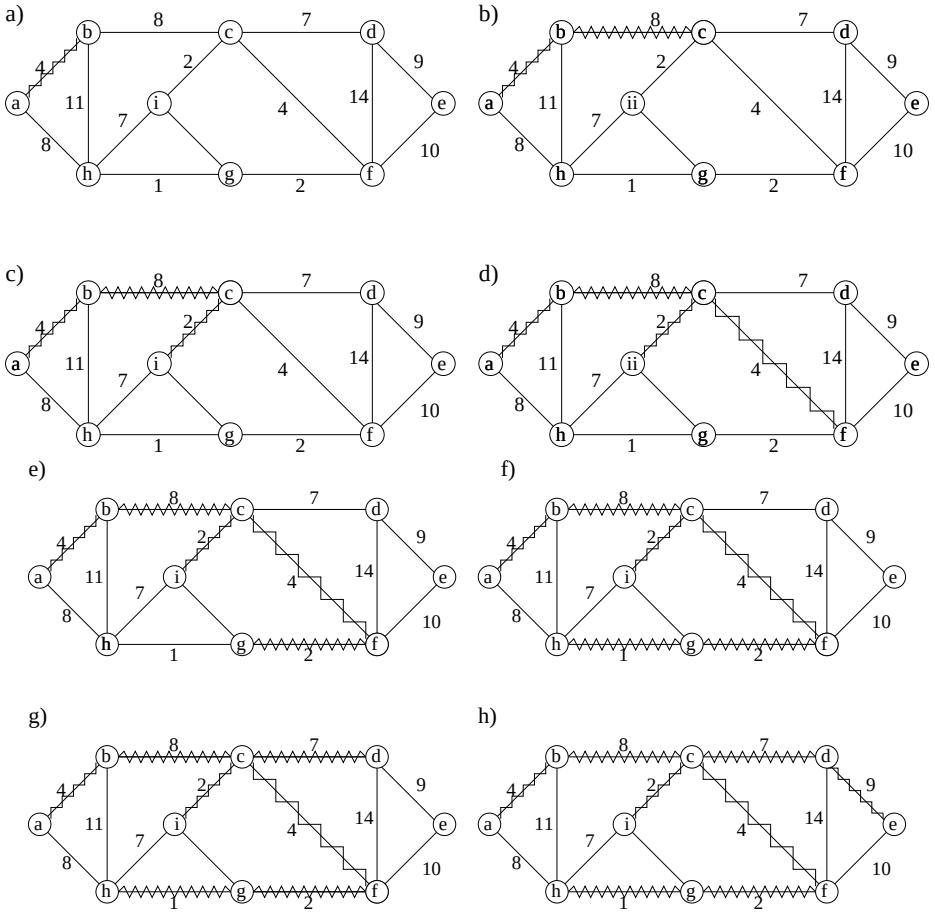
$u =$ DELETE-MIN(PQ)

For each $(u, v) \in E$ DO

IF $v \in PQ$ and $w(u, v) <$ key(v):
visit$[v] = u$
DECREASE-KEY$(PQ, v, w(u, v))$

Output edges $(u, visit(u))$ as part of MST.

On the example graph, the greedy algorithm would work as follows (starting at vertex a):



Prim's analysis:

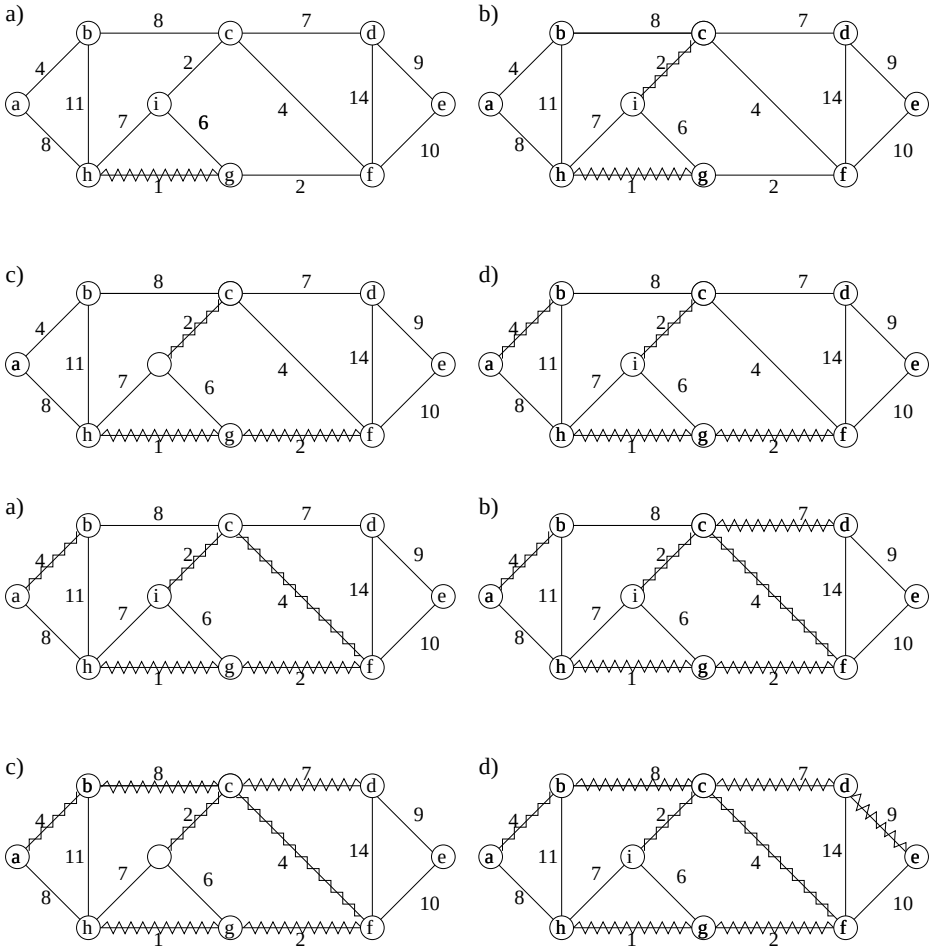
- $|V|$ INSERT's
- While loop runs $|V|$ times $\Rightarrow$ we perform $|V|$ DELETEMIN's
- We perform at most one DECREASE-KEY for each of the $|E|$ edges
- If use a heap as pqueue, inserts, delete-min and decrease-key run in $O(\lg V) \Longrightarrow O((|V| + |E|) \log |V|) = O(|E| \log |V|)$ time.

5 Kruskal's Algorithm

General idea:

- Start with a forest consisting of $|V|$ trees, each one consisting of one vertex
- Consider edges E in increasing order of their weight; add an edge if it connects two trees, ie if it does not create cycles.

Example:



To implement Kruskal's idea, we need to a structure that maintains the "partial" trees and is able to support the following operations: (a) initialize a tree as consisting of a single vertex; (b) are two vertices in the same tree?; and (c) join two trees together.

Abstracting further, we can think of each tree as a set of vertices. We need a data structure to store each set of vertices so that we are able to: (a) create a set consisting of one vertex; (b) determine of two vertices are in the same set; (c) join two sets.

This type of structure is referred to as an *union-find data structure* and its operations are referred to as:

- MAKE-SET(v): Create set consisting of v
- UNION-SET(u, v): Unite set containing u and set containing v
- FIND-SET(u): Return unique representative for set containing u

We can re-write Kruskal's algorithm to use a union-find data structure as follows:

```
KRUSKAL

T = ∅
FOR each vertex v ∈ V: MAKE-SET(v)
Sort edges of E in increasing order by weight
FOR each edge e = (u, v) ∈ E in order DO
    IF FIND-SET(u) ≠ FIND-SET(v) THEN
        T = T ∪ {e}
        UNION-SET(u, v)
```

Kruskal's analysis:

- We use $O(|E| \log |E|) = O(|E| \log |V|)$ time to sort edges and we perform $|V|$ MAKE-SET, $|V| - 1$ UNION-SET, and $2|E|$ FIND-SET operations.
- We will discuss a simple solution to the *Union-Find problem* such that MAKE-SET and FIND-SET take $O(1)$ time and UNION-SET takes $O(\log V)$ time amortized. With this implementation Kruskal's algorithm runs in time $O(|E| \log |E| + |V| \log |V|) = O((|E| + |V|) \log |E|) = O(|E| \log |V|)$ like Prim's algorithm.

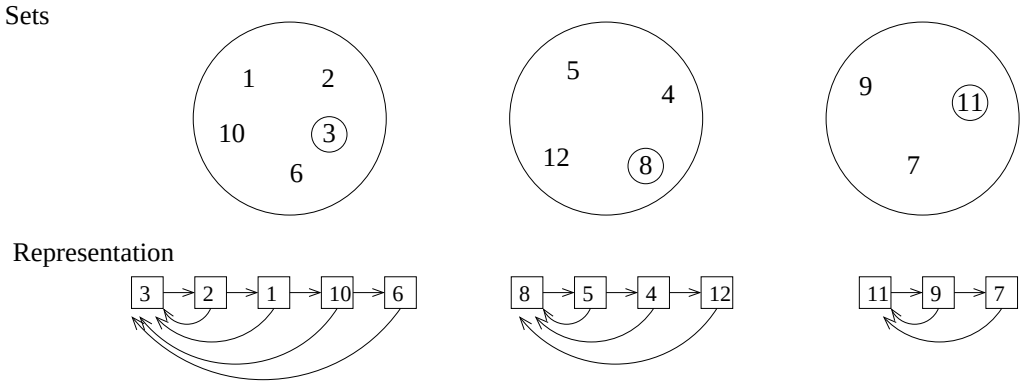
6 Union-Find

Kruskal's MST algorithm can be expressed in terms of an union-find data structure. As it turns out this is a general-purpose structure that has other applications. Generally speaking the *Union-Find problem* is the following: We want to provide a data structure to maintain a set of elements under the following operations:

- MAKE-SET(v): Create set consisting of v
- UNION-SET(u, v): Unite set containing u and set containing v
- FIND-SET(u): Return unique representative for set containing u

6.1 Simple solution

Perhaps the simplest idea that comes to mind is to maintain elements in same set as a linked list with each element having a pointer to the first element in the list (unique representative). Example:



- MAKE-SET(v): Make a list with one element $\Rightarrow O(1)$ time
- FIND-SET(u): Follow pointer and return unique representative $\Rightarrow O(1)$ time
- UNION-SET(u, v): Link first element in list with unique representative FIND-SET(u) after last element in list with unique representative FIND-SET(v) $\Rightarrow O(|V|)$ time (as we have to update all unique representative pointers in list containing u)

With this simple solution the $|V| - 1$ UNION-SET operations in Kruskal's algorithm may take $O(|V|^2)$ time.

6.2 Improved solution:

We can improve the performance of UNION-SET with a very simple modification:

Always link the smaller list after the longer list (update the pointers of the smaller list)

If you think: this is simple, you are right! As we'll show below, such a beautifully simple and intuitive idea results in an amortized bound of $O(\log |V|)$ per UNION-SET, amortized. This is algorithmic elegance at its best!

Why is this efficient?

- One UNION operation takes time proportional to number of pointer changes it needs to do
- One UNION-SET operation can still take $O(|V|)$ time (It is possible that we join a list of V/2 vertices to a list of V/2 vertices, which requires V/2 pointer updates).
- Kruskal's algorithm performs $|V| - 1$ UNION operations in total
- The trick is to think *over all the* UNION *operations performed by Kruskal's algorithm.*
- Consider element u:

  After pointer for u is updated, u belongs to a list of size at least double the size of the list it was in before the union
  $\Downarrow$
  After k pointer changes, u is in list of size at least 2^k
  $\Downarrow$
  Since the size of the largest set is $|V|$, the pointer of u can be changed at most $\log |V|$ times.
- In other words, over the course of all the UNION operations, each element can have its pointer changed only $O(\lg V)$ times.
- This means that $O(V)$ UNION operations will take $O(|V| \log |V|)$ time altogether in the worst case

Notes: With improvement, Kruskal's algorithm runs in time $O(|E| \log |E| + |V| \log |V|) = O((|E| + |V|) \log |E|) = O(|E| \log |V|)$ like Prim's algorithm.

7 Sate-of-the-art

A Union-Find data structure with better bounds is possible; however even when using an improved UF structure Kruskal's algorithm stays at $O(|E| \log |V|)$ and its running time will be dominated by sorting.

The best upper bound for computing MST is $O(|E|\alpha(|E|, |V|))$ where $\alpha()$ is the inverse Ackerman function; $\alpha()$ is a very slowly growing function, in practice a constant. MST can be computed in expected $O(V + E)$ time using a randomized algorithm—the algorithm was a breakthrough when first presented by Tarjan etal in 1995. Computing MST in deterministic linear time (using comparisons) is a big open problem.