Minimum Spanning Trees

Module: Graphs

1 MST: Definition

Given a 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. If the graph has more than one connected component, each component will have a spanning tree (and the union of these trees will form a spanning forest for the graph). The spanning tree of G is not unique.

Question: How would you go about finding a spanning tree of a connected and undirected graph G?

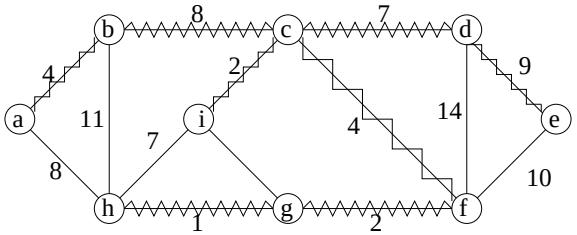
Answer: Run BFS or DFS; the resulting BFS-tree and DFS-tree are spanning trees of G.

The problem becomes more interesting when the graph is *weighted*: assume each edge (u, v) has weight $w(u, v)$. In this case, a spanning tree T will have a weight which is defined as the sum of the weights of the edges in the tree:

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

Of all possible spanning trees of G, we would like to find one of minimum total weight. This is called the *minimum spanning tree* (MST) of G. Note that 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 is considered one of the oldest, fundamental problems in graph algorithms. A spanning tree is essentially the "smallest" possible sub-graph that still keeps the same connectivity

as the original graph. When edges have weights, we naturally want the spanning tree of minimum total weight. 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).

- Draw a small graph and find the MST by hand.
- Does it have optimal sub-structure?
- How would you approach computing an MST? Can you express finding an MST recursively? Consider the with-without pattern.
- Another way to think about it: each vertex in G will have at least one incident edge in the MST. Can you say anything about this edge?
- Consider a graph that's a cycle. How many edges will NOT be in the MST? Which one(s)?

2 Properties of MST

To keep things simple, we'll assume that the edge weighst are distinct; that is, for any pair of edges e, e', we have $w(e) \neq w(e')$. This guarantees that the MST is unique. Without this condition, there may be several minimum spanning trees (e.g. when all edges have weight 1, all spanning trees are minimum spanning trees).

For any set S, 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: For any set S of vertices in a graph $G = (V, E)$, let $e = (u, v)$ be the *lightest* edge with exactly one endpoint in S. Then e must be in the MST.

Proof: Let T be a MST of G, let S be a set of vertices and let (u, v) be the lightest edge that crosses the cut. 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
- We know that e is the lightest edge that crosses the cut, therefore $w(u, v) < w(x, y)$
- This means that T' must have smaller weight than T.

- This is not possible, because T is the MST.
- Therefore it is not possible that T does not contain $e = (u, v)$.

From the proof above, it follows that if the weights are distinct, the lightest edge that crosses the cut *must* be in the MST, otherwise we get a contradiction.

If the edge weights are not distinct,the cut theorem can be stated as: For any set S of vertices in a graph $G = (V, E)$, let $e = (u, v)$ be the *lightest* edge with exactly one endpoint in S. There exists an MST that contain e.

Proof: As above, we know that $w(u, v) \leq w(x, y)$ and $w(T') \leq w(T)$ therefore since T is an MST, it must be that T' must have the same weight as T. T' is also a MST, and thus there is a MST containing T and e.

3 Generic MST algorithm

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 //i.e. T is on one side of the cut

    Find smallest edge e crossing S

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

4 Kruskal and Prim MST algorithms at a glance

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 algorithm: 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 empty 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.

Both Prim's and Kruskal's algorithms follow the abstract MST algorithm (above) 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.

5 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

- To find the smallest edge we use a priority queue containing the *vertices* not in the tree yet:

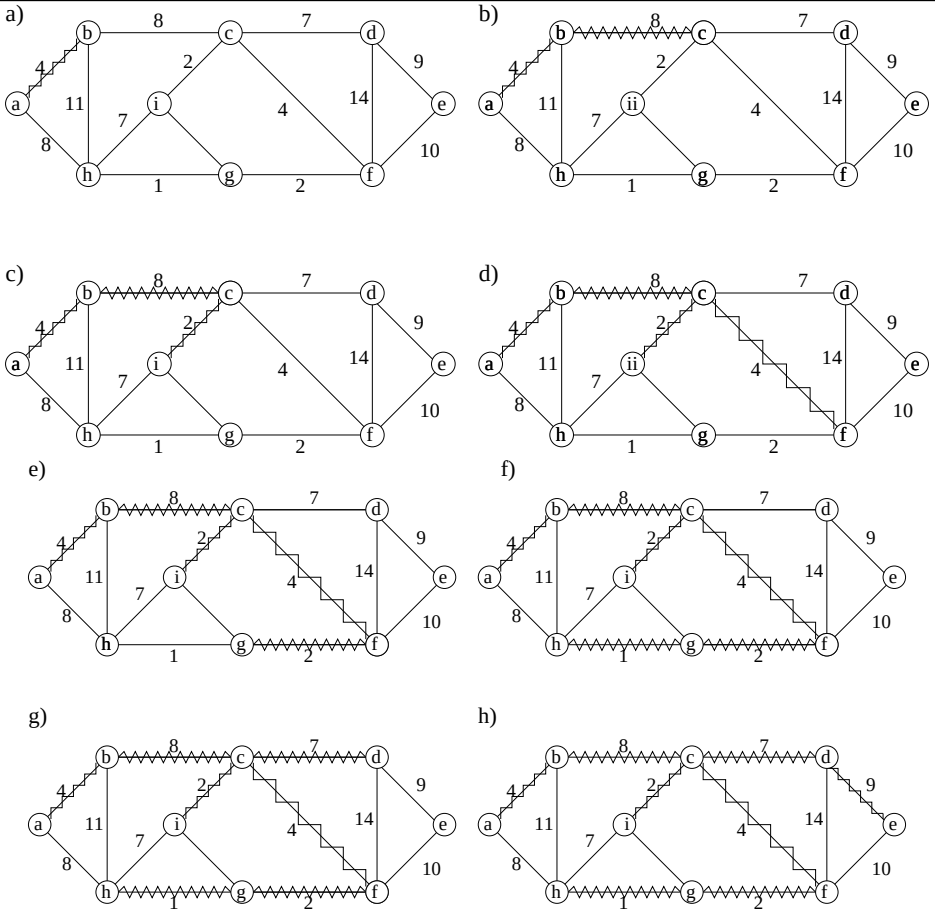
- The key/priority $d[v]$ of a vertex v is the weight of the smallest edge connecting v to the tree (implementation note: the priority of a vertex will be stored in an array $d[v]$ and also in the priority queue $(v, d[v])$; in order to be able to decrease the prioriy of a vertex we store a pointer to v's location in the priority queue)

- For each vertex v we store the edge that connects it to the tree; we call the other vertex of this edge by $pred(v)$

```
PRIM(G)
 1  // initialize
 2  Pick arbitrary vertex r and set d[r] = 0, PQ.INSERT(r, 0), pred(r) = NULL
 3  For each vertex u ∈ V(u ≠ r): d[u] = ∞, PQ.INSERT(u, ∞)

 4  while PQ not empty
 5        u = PQ.DELETE-MIN() // u is vertex closest to the tree
 6        For each adjacent edge (u, v)
 7              IF v in PQ and w_uv < d[v]
 8                    PQ.DECREASE-KEY(v, w_uv)
 9                    pred[v] = u
10  Output the edges (u, pred(u)) as the MST.
```

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



Prim's analysis:

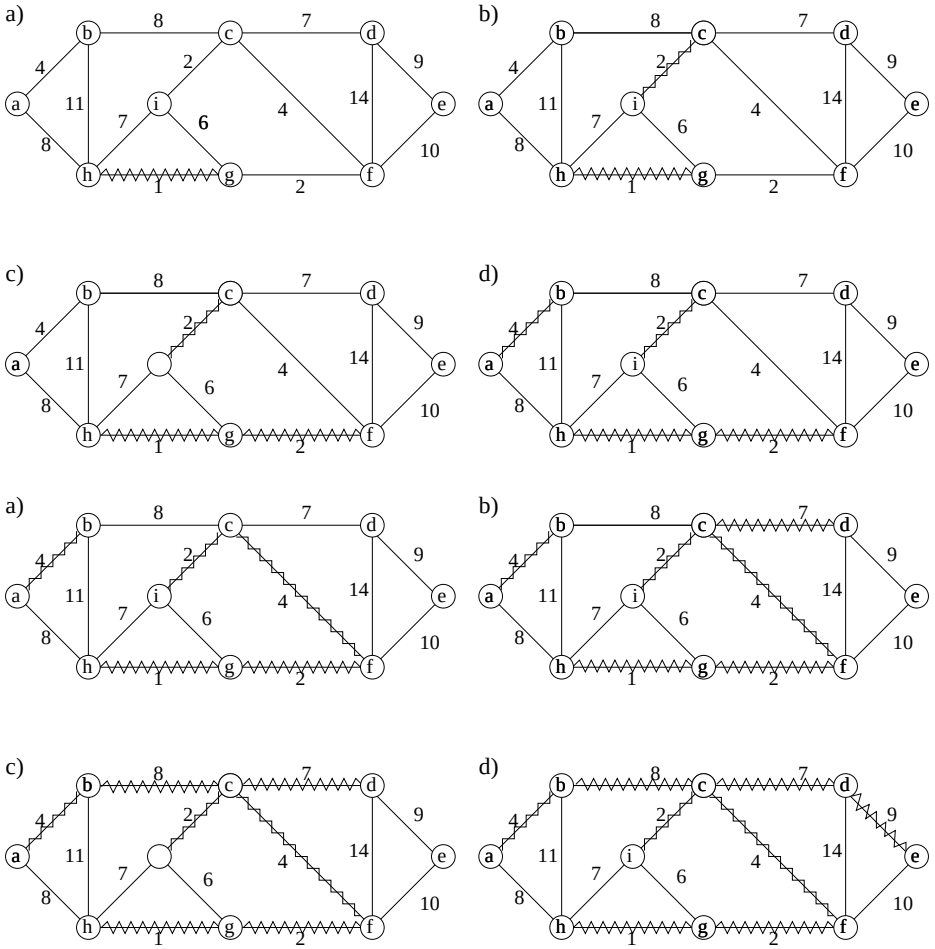
- $|V|$ INSERT's
- While loop runs $|V|$ times $\Rightarrow$ we perform $|V|$ DELETEMIN's
- We perform at most one DECREASE-KEY for each of the $|E|$ edges
- If use a heap as pqueue, inserts, delete-min and decrease-key run in $O(\lg V) \Longrightarrow O((|V| + |E|) \log |V|) = O(|E| \log |V|)$ time.

6 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.

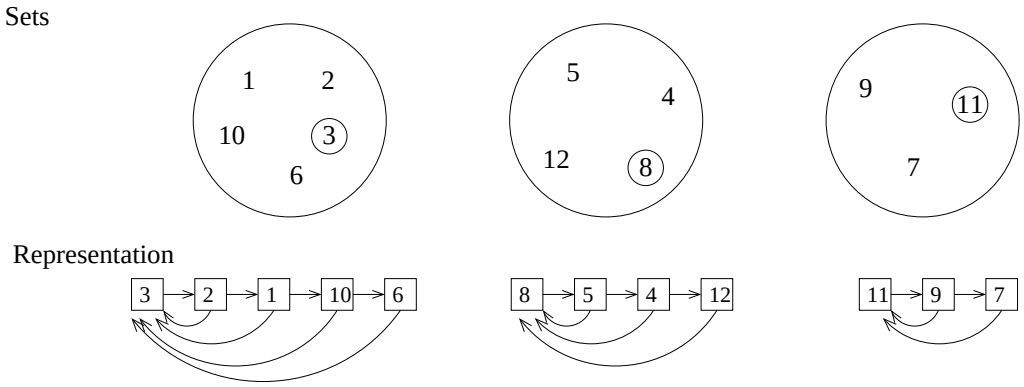
7 Union-Find

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

7.1 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.

7.2 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
  ⇓
  After k pointer changes, u is in list of size at least 2^k
  ⇓
  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.

Note: Kruskal and Prim's algorithms can be improved by plugging in improved Union-Find and priority queue data structures; however even when using an improved union-find structure Kruskal's algorithm stays at $O(|E| \log |V|)$ and its running time is dominated by sorting.

8 MST in linear time ?

MST is considered by some the oldest open problem in computer science. Improved algorithms that run in $O(|E| \lg \lg |V|)$ were given in the mid seventies, followed by more improvements.

In 1995 Karger, Klein and Tarjan described a randomized algorithm for computing an MST in *expected* $O(V + E)$ time. This results was a breakthrough.

Subsequently, it was shown that if the edge costs are integers it is possible to find the MST with a deterministic algorithm in worst-case linear time.

An MST algorithm that runs in deterministic $O(V+E)$ time and without making any assumption on the edges weights has been a long-standing open problem.

At present, 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. This result is due to Bernard Chazelle, from Princeton, and dates back to 1999. It does not resolve the open question, but it takes a big step and comes pretty close, and, perhaps more importantly, it introduces a non-greedy approach to MST (the idea is to compute a sub-optimal tree and progressively refine it).