CS 161	Lecture 15: Minimum Spanning Trees
Winter 2021	Mon, Mar 8

Adapted from Virginia Williams's lecture notes. Additional credits go to Logan Short, William Chen and Mary Wootters. Some of the figures in these notes are taken from CLRS.

Please direct all typos and mistakes to Moses Charikar and Nima Anari.

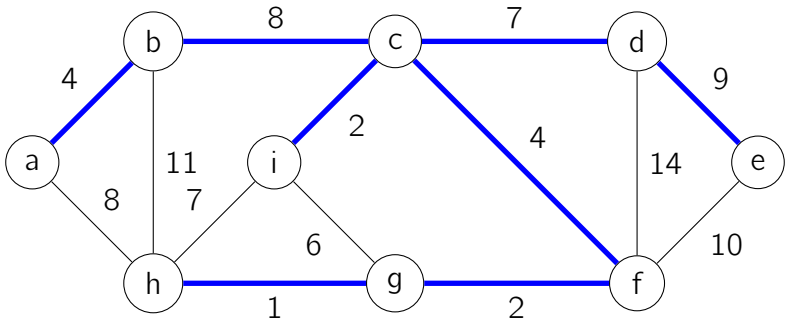
---

1 Introduction

Today we will continue our discussion of greedy algorithms, specifically in the context of computing minimum spanning trees. There are many useful applications for finding a minimum spanning tree of a graph from efficient network design to graph clustering analysis and much more. We will also show that we can compute a minimum spanning tree of a graph in polynomial time using some intuitive greedy algorithms.

The minimum spanning tree problem is formulated informally as follows: we are provided an undirected graph $G = (V, E)$ with weights $w(e) \in \mathbb{R}$ for $e \in E$ and we want to compute a subgraph of G that is a tree which connects all vertices in V (a spanning tree) and has minimum total edge weight defined as $w(T) = \sum_{e \in T} w(e)$.

Below is an example of an MST of a graph. In the example, the edges forming the MST are colored blue while edges that are not part of the MST are colored black:



2 A Template for Minimum Spanning Tree Algorithms

Let's start by introducing a basic algorithm template which will guide our discussion towards the actual algorithms for computing MSTs. These algorithms will in general follow the steps described in the template below:

We will show that the template results in a valid MST by maintaining the invariant that there exists at least one MST which contains all the edges in A. An edge is considered safe to add to A as long as it maintains this invariant. We will see that this definition of a safe edge

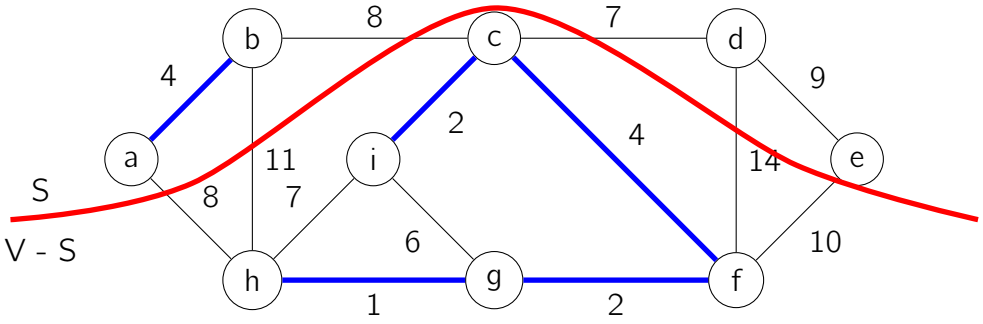
Algorithm 1: Template for Minimum Spanning Tree Algorithms

$A \leftarrow \emptyset$
while *A is not a spanning tree* **do**
    find edge (u, v) that is 'safe' for A
    $A \leftarrow A \cup \{u, v\}$
return A

can informally be defined as the edge with minimum weight which would not form a cycle if included in A. The next section will introduce new terminology to define this formally.

3 Cuts and Light Edges

We will introduce the notion of graph cuts to formally discuss which edges can be considered safe to add to the MST edge set. Let a *cut* $(S, V - S)$ of a graph $G = (V, E)$ be a partition of V into two disjoint sets S and $V - S$. From this, we can say that an edge (u, v) *crosses* the cut $(S, V - S)$ if the edge has one endpoint in S and the other in $V - S$. We can also say that a cut *respects* a subset A of edges if no edges in A cross the cut. An edge is considered a *light edge* crossing a cut if its weight is the minimum of any edge crossing the cut.



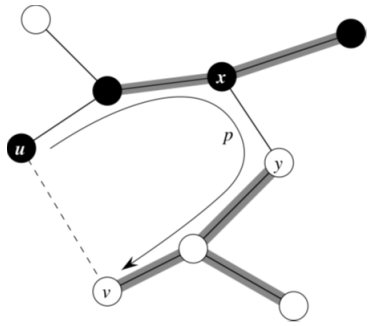
In the example above, let $(S, V - S)$ be a cut of the graph where S contains the set of nodes above the red curve and $V - S$ contains the set of nodes below it, and the set A be the set of edges colored blue. The edges which cross the cut are exactly the following: $(a, h), (b, h), (b, c), (c, d), (d, f), (e, f)$ and the only light edge which crosses $(S, V - S)$ is (c, d). Since none of these edges are contained in the set A, the cut respects the set A. Note that if we were to add any of the edges previously mentioned to A, then the cut would no longer respect A.

Given the definitions above, let $G = (V, E)$ be a connected and undirected graph with edge weights $w(e)$, A be a subset of E such that some MST of G contains A, $(S, V - S)$ be a cut that respects A, and (u, v) be a light edge crossing $(S, V - S)$.

Theorem 1. *There exists an MST that contains* $A \cup \{(u, v)\}$.

Proof. Let T be an MST containing A. As previously mentioned, (u, v) is a light edge which

crosses the cut $(S, V - S)$. Since T is already a spanning tree, note that adding any other edge of the graph to it will lead to a cycle, so in particular adding (u, v) to T produces a cycle. Consider a path p from u to v in T. There will necessarily be at least one edge (x, y) of p which crosses the cut $(S, V - S)$ where $(x, y) \notin A$ because the cut respects A. Since (u, v) is a light edge, $w(u, v) \leq w(x, y)$. Deleting (x, y) from T and adding (u, v) yields a new MST T'. The only difference between T and T' are the edges (x, y) and (u, v) so $w(T') \leq w(T)$. T' is an MST which contains $A \cup \{(u, v)\}$. □



Note that in the proof, if $w(T') \neq w(T)$, then we have that our initial assumption of T being an MST is false since we have found a spanning tree with smaller total edge weight.

Because of the theorem, we can add some additional points about the MST algorithm template.

- The MST algorithm maintains a subset A of edges with no cycles. That is, the graph represented by $G_A = (V, A)$ is a forest (a set of distinct unconnected trees).
- Any safe edge (u, v) connects two distinct connected components of G_A.
- For some connected component $C = (V_C, E_C)$ in G_A, the safe edge (u, v) is a light edge crossing $(V_C, V - V_C)$.

4 Prim's Algorithm

At a high level, the set A maintained by Prim's algorithm is a single tree. The algorithm starts with an arbitrary root r and in each step, a light edge leading out of A and connecting to a node that has not yet been connected to A is selected and added to A. Once A connects every node in the graph, it is returned as an MST of the graph.

Prim's algorithm is similar to Dijkstra's algorithm in that estimates of the distance to each node are maintained and updated as the algorithm progresses. Q is a priority queue maintaining distances of vertices not in the tree so far, key(v) is the minimum weight of edge connecting v to some vertex in the tree, and p(v) is the parent of v in the tree.

Correctness Much of the correctness of Prim's algorithm follows from Theorem 1. Notice that at the beginning of every loop iteration, $A = \{ (\text{p}(v), v) : v \in (V - \{r\} - Q)\}$ meaning that the vertices already placed in the partial MST are those in $V - Q$. For all vertices $v \in Q$,

Algorithm 2: Prim(G)

```
key(v) ← ∞, ∀v ∈ V
key(r) ← 0
Q ← (key(v), v), ∀v ∈ V
p(v) ← NIL, ∀v ∈ V
A ← ∅
while Q is not empty do
    u ← ExtractMin(Q)
    if u ≠ r then
        A = A ∪ {(p(u), u)}
    for each neighbor v of u do
        if v ∈ Q and w(u, v) < key(v) then
            key(v) = w(u, v)
            DecreaseKey(key(v), v)
            p(v) = u
return A
```

if $p(v) \neq$ NIL, then key(v) is the minimum weight of an edge connecting v to the partial MST. This can be thought of in terms of graph cuts with partitions $(Q, V - Q)$ and the vertices in Q with non-NIL parents as being the tail of edges crossing this cut. Since in Q, only the vertices with non-NIL parents have key $\neq \infty$ (except for r in the first iteration), this means that only the edges which cross the cut are considered at each iteration and the one with minimum weight is added to A. This is exactly what the MST template algorithm does (we add a safe edge) and as such, the correctness of the algorithm follows.

Running time Prim's Algorithm can be implemented as a direct modification of Dijkstra's Algorithm and can achieve a similar running time, but its exact bound depends on the implementation of the priority queue.

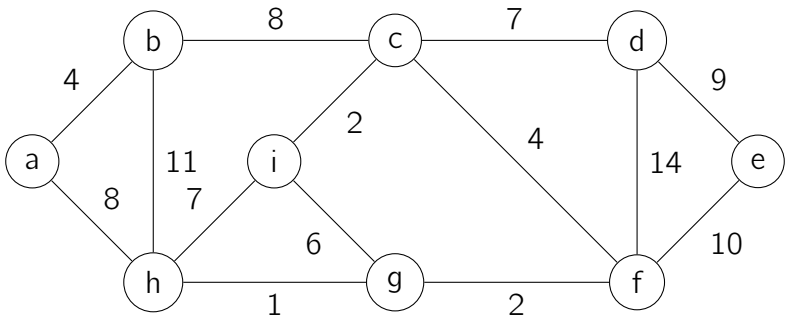
If a red-black tree or a binary heap is used:

- ExtractMin: $O(\log n)$
- DecreaseKey: $O(\log n)$
- Total: $O(n \log n + m \log n) = O(m \log n)$

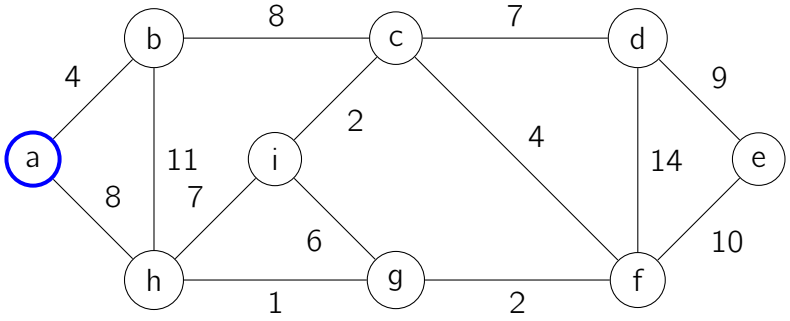
If a Fibonacci heap is used:

- ExtractMin: $O(\log n)$
- DecreaseKey: $O(1)$ amortized
- Total: $O(n \log n + m)$

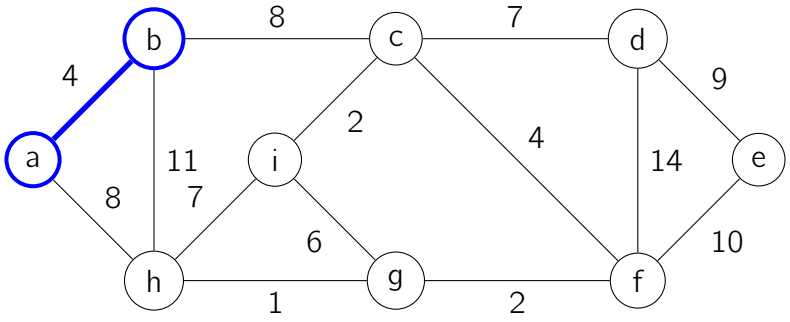
Example In this example we will run through the steps of Prim's algorithm in order to find an MST for the following graph:



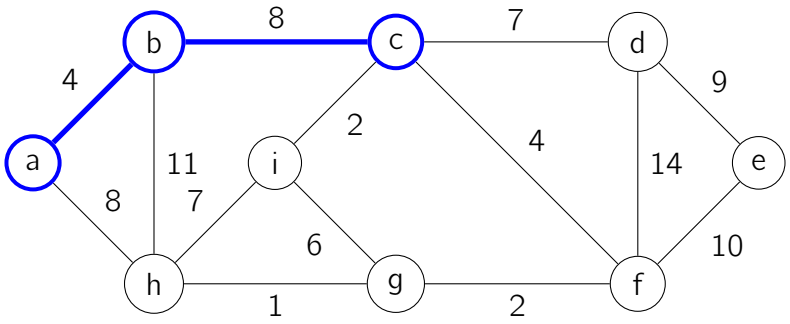
Suppose we select node a to be the source node, r. We then extract node a from Q and set key(b) = 4 , p(b) = a, key(h) = 8, and p(h) = a.



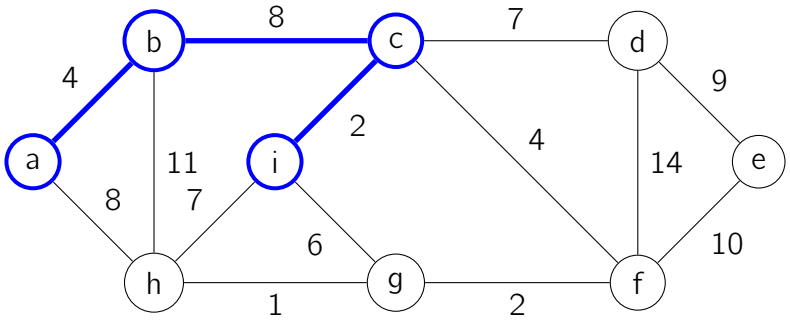
Since key(b) is now the smallest value in the priority queue, we visit node b. Because p(b) = a we add edge (a, b) to the set A. We then update the keys and parent fields of nodes that have edges connecting to b. Thus we set key(c) = 8 and p(c) = b.



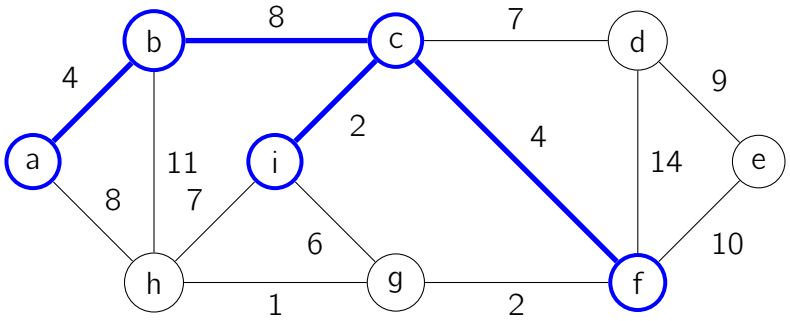
The next smallest in the priority queue is a tie between key(c) and key(h). The algorithm can pick either one – the results may be different, but both will be an MST. Let's say the algorithm arbitrarily picks c. We add edge (b, c) to A and perform the following updates: key(d) = 7, p(d) = c, key(f) = 4, p(f) = c, key(i) = 2, and p(i) = c.



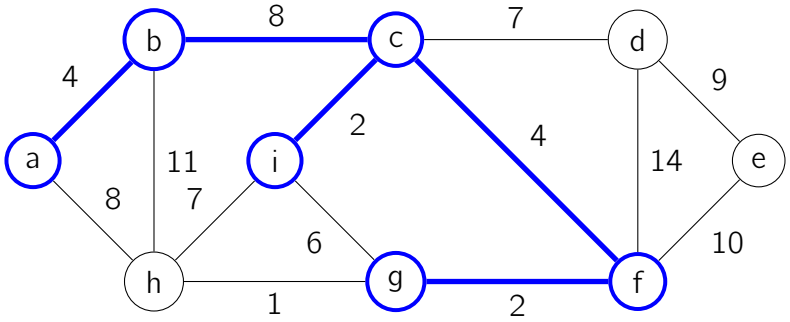
key(i) is the smallest so we visit node i. Update the following: key(g) = 6, p(g) = i, key(h) = 7, and p(h) = i.



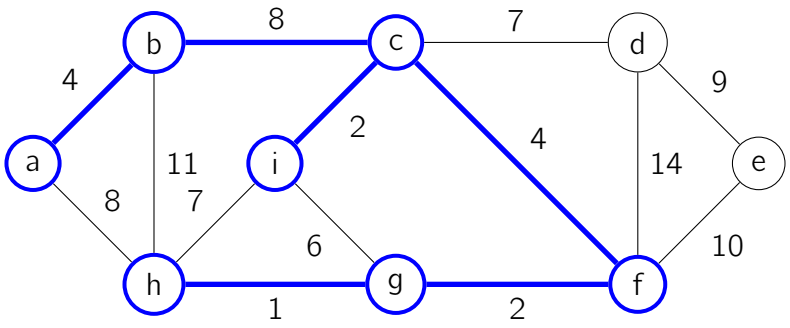
key(f) is the smallest so we visit node f. Update the following: key(g) = 2, p(g) = f, key(e) = 10, and p(e) = f.



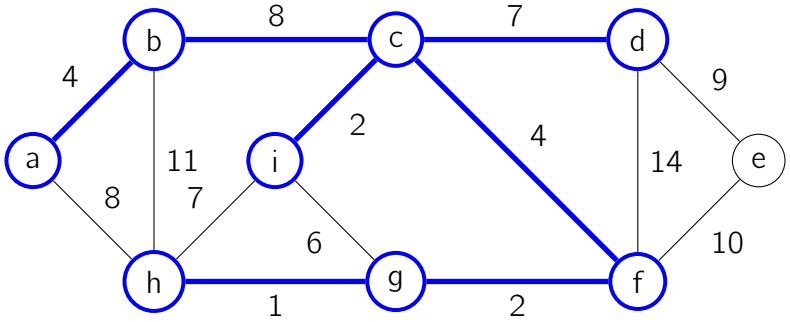
key(g) is the smallest so we visit node g. Update the following: key(h) = 1 and p(h) = g.



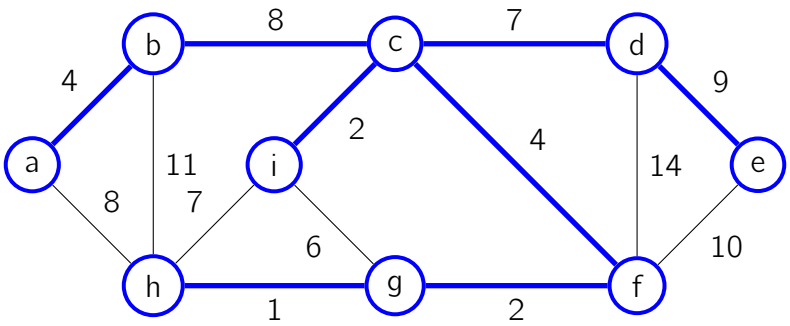
key(h) is the smallest so we visit node h. There are no updates at this step.



key(d) is the smallest so we visit node d. Update the following: key(e) = 9 and p(e) = d.



Finally, key(e) is the smallest so we visit node e. There are no updates at this step and the algorithm will detect that Q is empty at the next iteration and return.



5 Kruskal's Algorithm

At a high level, the set A maintained by Kruskal's algorithm is a set of disjoint trees. During update step i, if the ith smallest edge connects different trees, merge the two trees connected by this edge. The algorithm progresses until eventually only one tree remains at which point the set A represents an MST of the graph.

Kruskal's algorithm utilizes the union-find (aka disjoint set) data structure in order to handle the merging of the disjoint trees maintained by the algorithm. The union-find data structure supports disjoint sets with the following operations:

- **makeset(u):** creates a new set containing u provided that u is not in any other set
- **find(u):** returns the name of the set containing u
- **union(u, v):** merge the set containing u and the set containing v into one set

The algorithm itself can be structured as follows:

Algorithm 3: Kruskal(G)

```
A ← ∅
E' ← sort edges by weight in non-decreasing order
foreach v ∈ V do
    makeset(v)
foreach (u, v) ∈ E' do
    if find(u) ≠ find(v) then
        A ← A ∪ {(u, v)}
        union(u, v)
return A
```

Correctness The correctness follows from Theorem 1.

Running time The runtime of Kruskal's algorithm depends on two factors: the time to sort the edges by weight and the runtime of the union-find data structure operations. While $\Omega(m \log n)$ time is required for sorting the edges if we use comparison-based sorting, in many cases, we may be able to sort the edges in linear time. (Recall, that RadixSort can be used to sort the edges in $O(m)$ time if the weights are given by integers bounded by a polynomial in m.) In this case, the runtime is bounded by the runtime of the union-find operations and is given by $O(nT(\text{makeset}) + mT(\text{find}) + nT(\text{union}))$. The best known data structure supporting the union-find operations runs in amortized time $O(\alpha(n))$ where $\alpha(n)$ is the inverse Ackermann function. Interestingly, the value of the inverse Ackermann is tiny for all practical purposes:

$$\alpha(n) \leq 4, \quad \forall n < \texttt{\# atoms in the universe}$$

and thus for all practical purposes, the union-find operations run in constant time. Thus, in many settings, the runtime of Kruskal's algorithm is nearly linear in the number of edges.

The actual definition of $\alpha(n)$ is $\alpha(n) = \min\{k \mid A(k) \geq n\}$, where $A(k)$ is the Ackermann function evaluated at k. $A(k)$ itself is defined using the more general Ackermann function as $A(k) = A_k(2)$. $A_k(x)$ is defined recursively:

$$A_m(x) = \begin{cases} x+1 & m = 0 \\ A_{m-1}(1) & m > 0, x = 0 \\ A_{m-1}(A_m(x-1)) & else \end{cases}$$

For example

- $A_0(x) = 1 + x$, so $A_0(2) = 3$.
- To compute $A_1(x)$, we see:

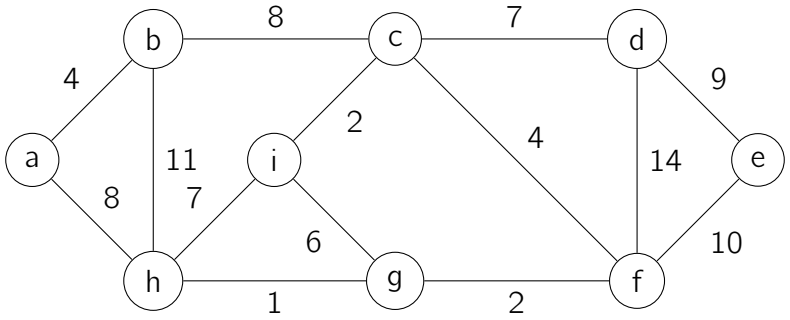
  $$A_1(x) = A_0(A_1(x-1)) = A_0(A_0(A_1(x-2))) = \cdots = A_0(A_0(\cdots(A_0(A_1(0))))) = A_0(A_0(\cdots(A_0(2)))),$$

  that is, we have "iterated" A_0 x times. If we work it out, we get

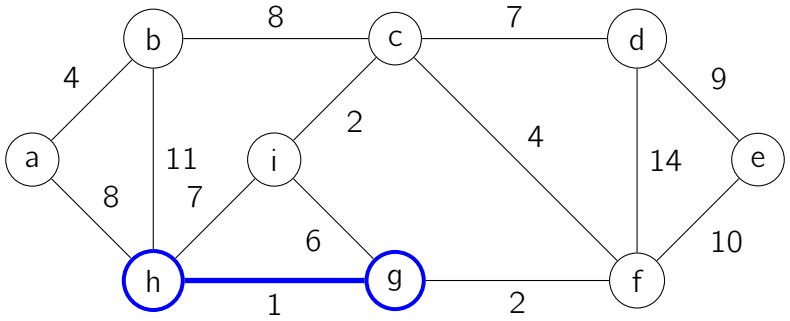
  $$A_1(x) = 2x.$$

  Thus, $A_1(2) = 4$.
- $A_2(x) = 2^x x$ (A_1 iterated x times), so $A_2(2) = 8$
- $A_3(x) \geq 2^{2^{2^{2^{\cdots}}}}$, a "tower" of x 2s (A_2 iterated x times); it turns out $A_3(2) \geq 2^{11}$
- $A_4(x)$ is larger than the total number of atoms in the known universe, and also larger than the number of nanoseconds since the Big Bang. (Thus, $\alpha(n) \leq 4$ for all practical purposes.)

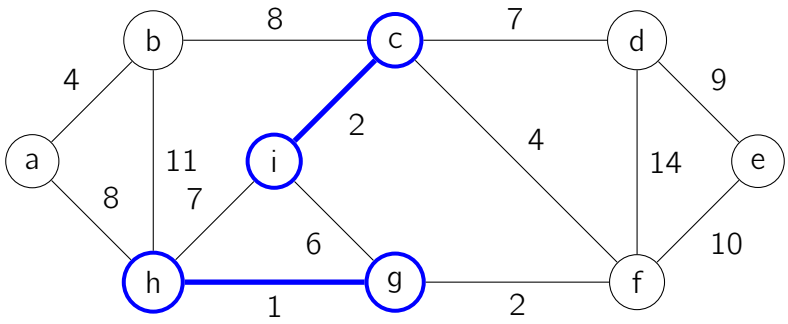
Example In this example we will run through the steps of Kruskal's algorithm in order to find an MST for the following graph:



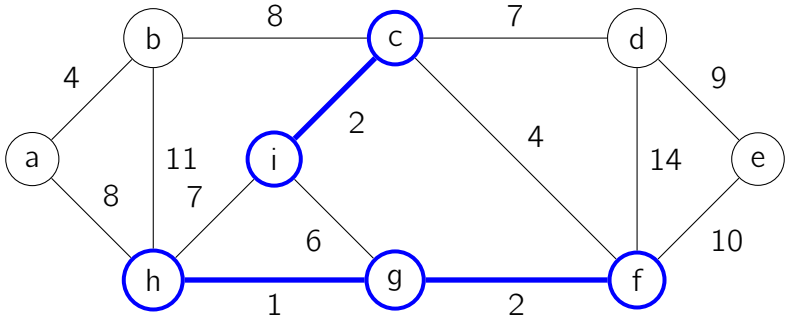
We begin by creating a new set for each node in the graph. We then begin iterating over the edges in non-decreasing order. The first edge we examine is (g, h). This edge connects nodes g and h which are currently not part of the same set. We thus include this edge in A and union the sets containing g and h.



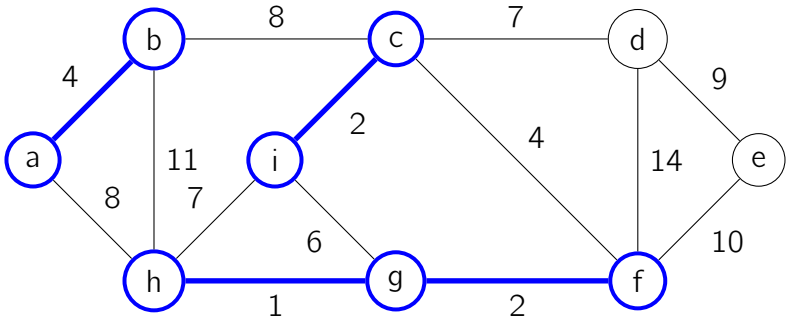
The next edge in the sorted order is a tie between edges (c, i) and (f, g). Picking either one will yield a correct result, so let's say the algorithm picks (c, i). Since c and i are not part of the same set we include this edge in A and union the sets containing c and i.



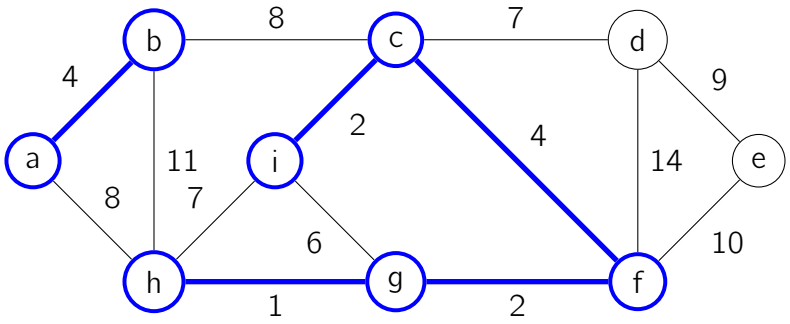
The next edge in the sorted order is (f, g). We union the sets containing f and g and add edge (f, g) to A.



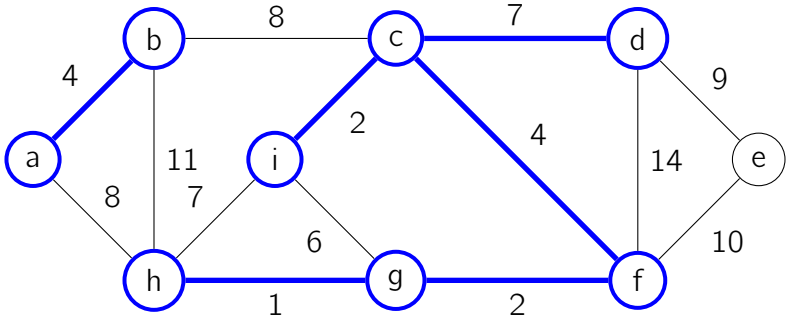
The next edge in the sorted order is a tie between edges (a, b) and (c, f). Let's say the algorithm picks (a, b). We union the sets containing a and b and add edge (a, b) to A.



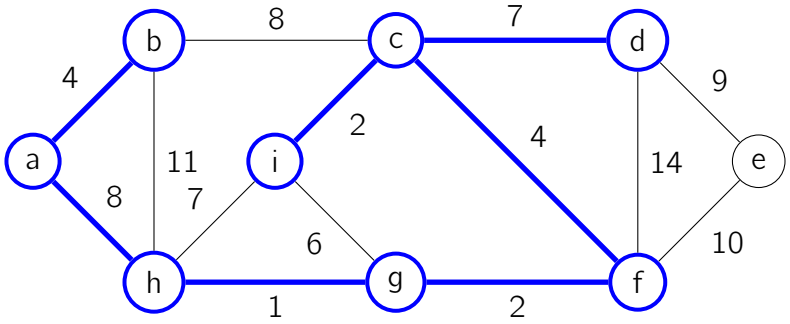
The next edge in the sorted order is (c, f). We union the sets containing c and f and add edge (c, f) to A.



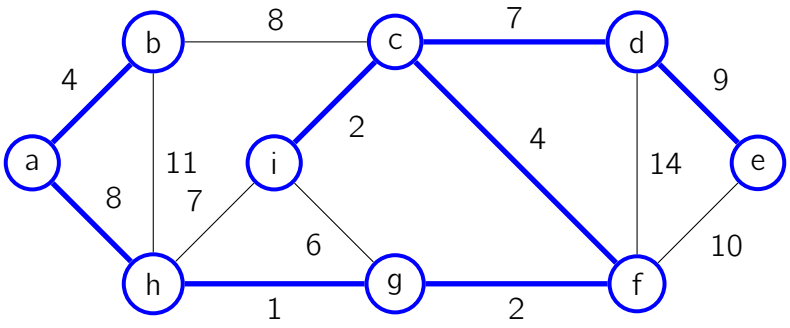
The next edge in the sorted order is (i, g). Note that i and g are contained in the same set which means that adding the edge (i, g) would lead to a cycle in A. We therefore ignore (i, g) and pick the next edge in sorted order which is a tie between (c, d) and (h, i). Let's say the algorithm picks (c, d). We union the sets containing c and d and add edge (c, d) to A.



The next edge in the sorted order is (h, i), but h and i are contained in the same set so we ignore it. The next edge is a tie between edges (a, h) and (b, c). Let's say the algorithm picks (a, h). We union the sets containing a and h and add edge (a, h) to A.



The next edge in the sorted order which has both vertices in different sets is (d, e). We union the sets containing d and e and add edge (d, e) to A. At this point all nodes are contained in the same set, so no further edges are added to A.



6 The Latest and Greatest Algorithms

While the greedy algorithms mentioned above are reasonably efficient ways to compute a minimum spanning tree of a graph, recent research has yielded more efficient algorithms. In 1995, Karger, Klein, and Tarjan discovered a randomized linear time ($O(E + V)$) algorithm based on Borůvka's algorithm and the reverse-delete algorithm. In 2000, Chazelle discovered the current fastest determistic algorithm which runs in time $O(E\alpha(V))$ using soft heaps where α is the inverse Ackermann function.