Adapted From Virginia Williams'lecture notes.

COMP 285 (NC A&T, Spr '22) Lecture 31

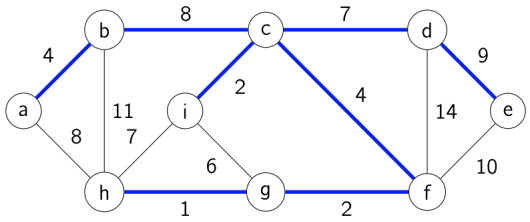
Minimum Spanning Trees

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 $w(e) \in R$ 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 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.

Algorithm 1: Template for Minimum Spanning Tree Algorithms

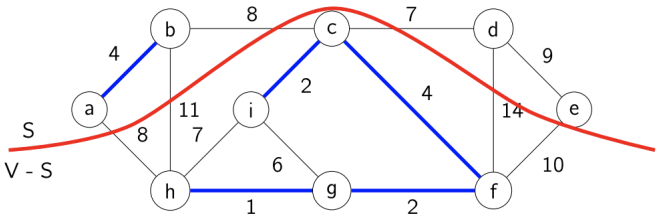
$A \leftarrow \emptyset$
while *A is not a spanning tree* **do**
  find edge (u, v) that is 'safe' for A
  $A \leftarrow A \cup \{(u, v)\}$
return A

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 $(S, V \setminus S)$ of a graph $G = (V, E)$ be a partition of V into two disjoint sets S and $V \setminus S$. From this, we can say that an edge

$$(u, v)$$

crosses the cut $(S, V \setminus S)$ if the edge has one endpoint in S and the other in $V \setminus S$. 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.



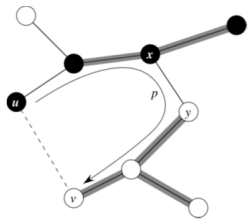
In the example above (Figure 3), let $(S, V \setminus S)$ be a cut of the graph where S contains the set of nodes above the red curve and $V \setminus S$ contains the set of nodes below it, and the set A be the set of edges colored blue. The edges which cross the cut are exactly the following: $(a, h), (b, h), (b, c), (c, d), (d, f), (e, f)$ and the only light edge which crosses $(S, V \setminus S)$ 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, $(S, V \setminus S)$ be a cut that respects A, and (u, v) be a light edge crossing $(S, V \setminus S)$.

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 $(S, V \setminus S)$. 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 $(S, V \setminus S)$ 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 $(V_C, V \setminus V_C)$.

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, $A = \{(p(v), v) : v \in (V \setminus \{r\} \setminus Q)\}$ meaning that the vertices already placed in the partial MST are those in $V \setminus Q$. For all vertices $v \in Q$, if $p(v) \neq \texttt{NIL}$, 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 $(Q, V \setminus Q)$ 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.

Algorithm 2: Prim(G)

```
key(v) ← ∞, ∀v ∈ V
key(r) ← 0
Q ← (key(v), v), ∀v ∈ V
p(v) ← NIL, ∀v inV
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
```

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: $O(logn)$
- DecreaseKey: $O(logn)$
- Total: $O(nlogn + mlogn) = O(mlogn)$

If a Fibonacci heap is used:

- ExtractMin: $O(logn)$
- DecreaseKey: $O(1)$ amortized
- Total: $O(nlogn + m)$

Example In this example, we will run through the steps fo Prim's algorithms in order to find an MST for the graph in Figure 1:

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 $p(h) = a$ as shown in Figure 2.

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

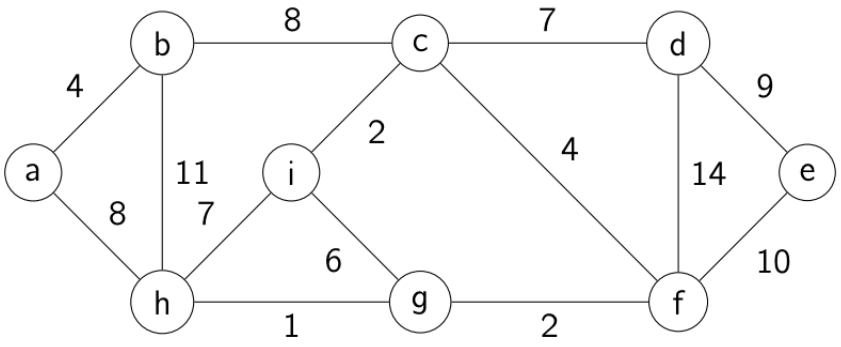


Figure 1:

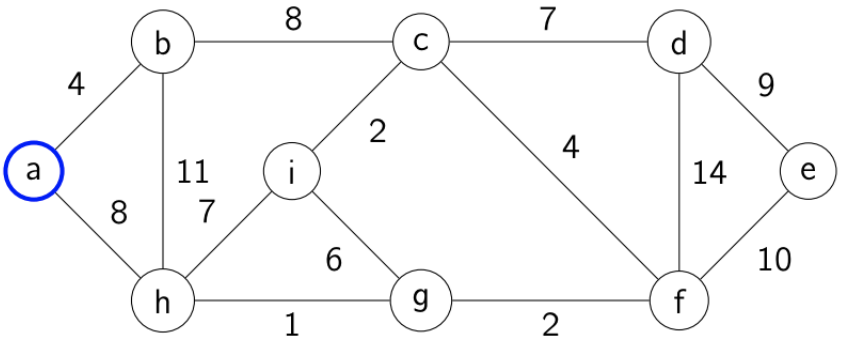


Figure 2:

have edges connecting to b. Thus we set key$(c) = 8$ and $p(c) = b$ as shown in Figure 3.

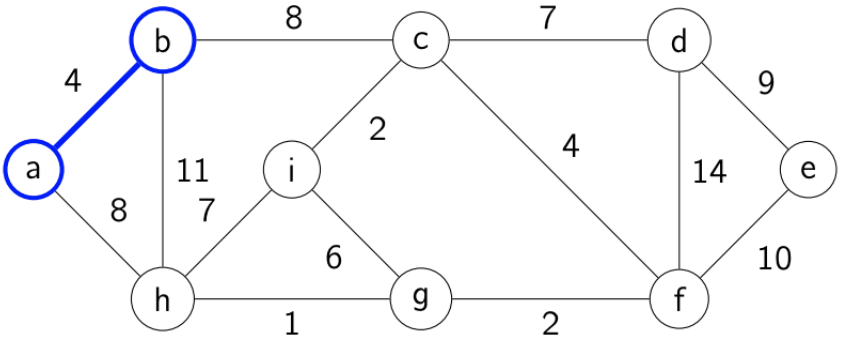


Figure 3:

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 $p(i) = c$ as shown in Figure 4.

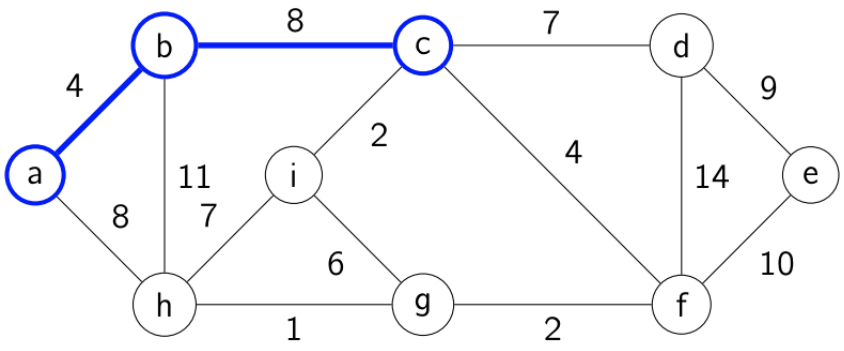


Figure 4:

key(i) is the smallest so we visit node i. Update the following: key(g) $= 6$, $p(g) = i$, key(h) $= 7$, and $p(h) = i$ as shown in Figure 5.

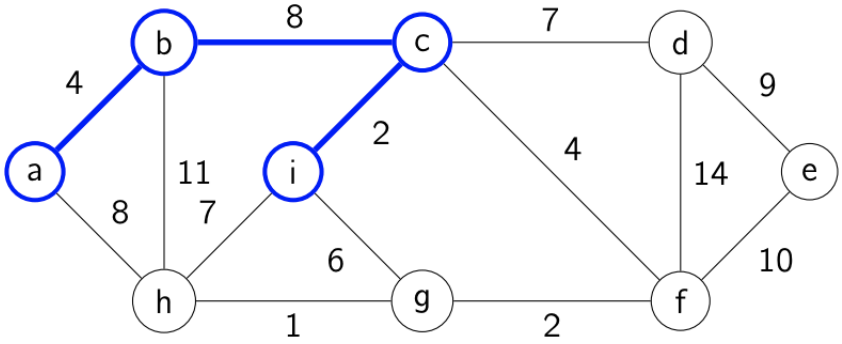


Figure 5:

key(f) is the smallest so we visit node f. Update the following: key(g) $= 2$, $p(g) = f$, key(e) $= 10$, and $p(e) = f$ as shown in Figure 6.

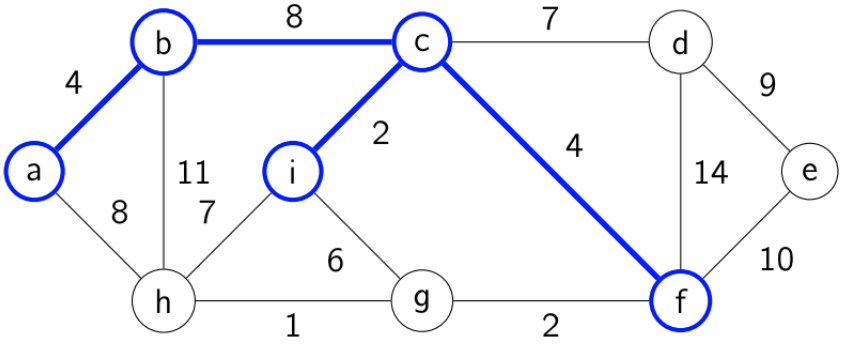


Figure 6:

key(g) is the smallest so we visit node g. Update the following: key(h) $= 1$ and $p(h) = g$ as shown in Figure 7.

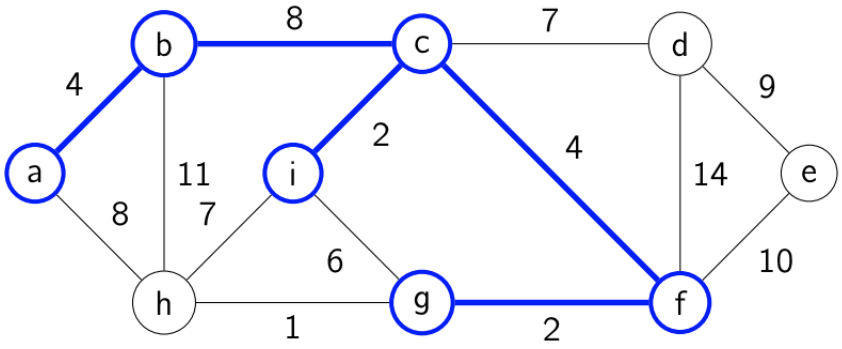


Figure 7:

key(h) is the smallest so we visit node h. There are no updates at this step as shown in Figure 8.

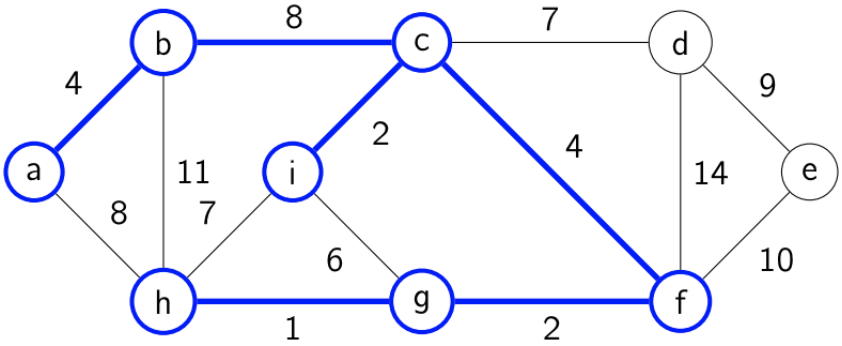


Figure 8:

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

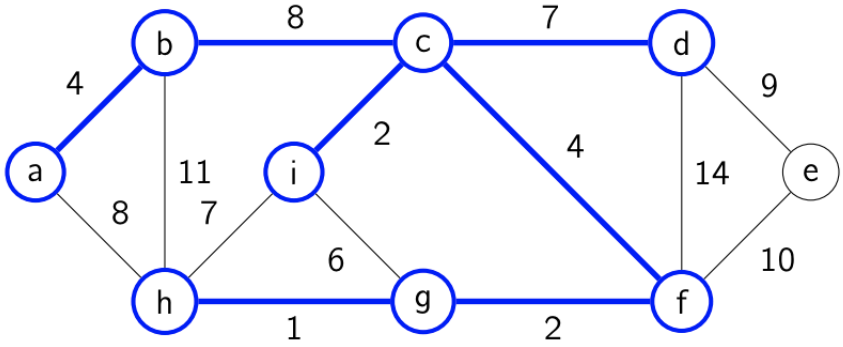


Figure 9:

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 as shown in Figure 10.

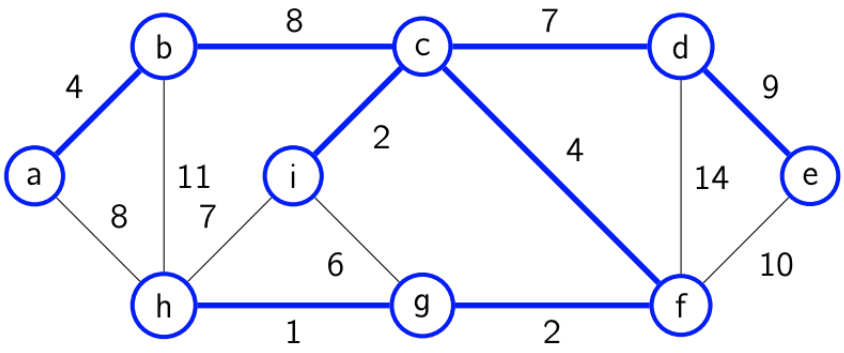


Figure 10: