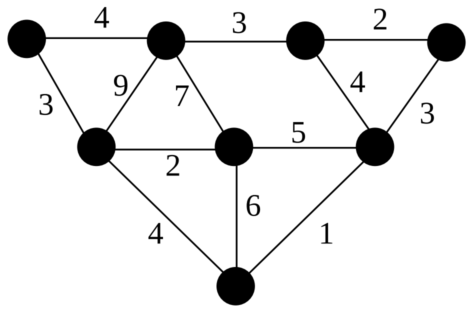
Minimum Spanning Tree

- Let $G = (V, E)$ be a connected, weighted graph.
- Recall that a weighted graph is a graph where we associate with each edge a real number, called the **weight**. graph.
- Recall that a **spanning tree** of G is a subgraph T of G which is a tree that spans G. In other words, it contains all of the vertices of G.
- The **weight** of a spanning tree T is the sum of the weights of its edges. That is,
$$w(T) = \sum_{(u,v) \in T} w(u, v).$$
- A **minimum spanning tree (MST)** of G is spanning tree T of minimum weight.
- It should be clear that a minimum spanning tree always exists.

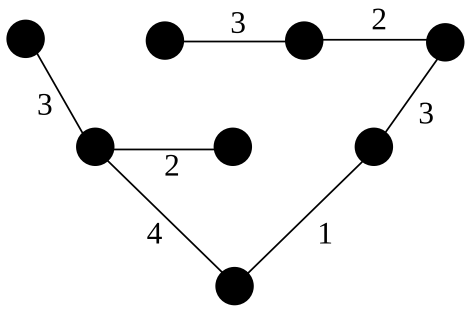
Minimum Spanning Tree Examples

- A graph G.



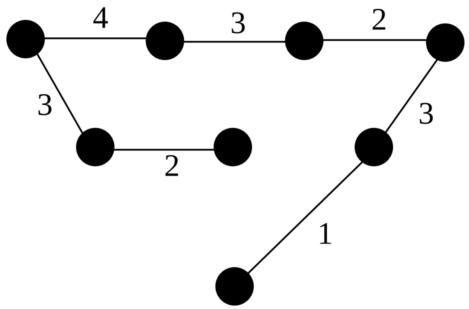
G

- T_1 is a minimum spanning tree of G.



T_1

- T_2 is another minimum spanning tree of G.



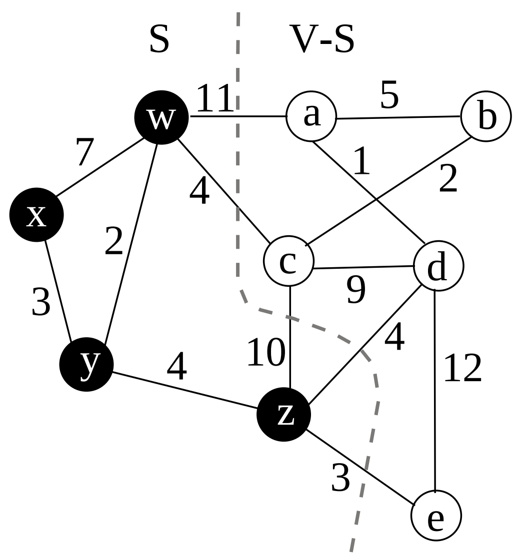
T_2

Constructing an MST

- Minimum spanning trees can be constructed in a greedy fashion.
- There are two common algorithms to construct MSTs:
  - **Kruskal's algorithm**
  - **Prim's algorithm**
- Both of these algorithms use the same basic ideas, but in a slightly different fashion.
- We will proceed as follows:
  - We will consider a general approach to solving MST.
  - We will prove that the general approach works.
  - We will show two methods of implementing the general method: Kruskal's and Prim's algorithms.

Some Terminology

- A **cut** $(S, V - S)$ of G is a partition of the vertices V.
- An edge $(u, v) \in E$ is said to **cross** the cut if one of the endpoints is in S, and the other is in $V - S$.
- The set of edges which cross a cut are the **cross edges**.
- A cut **respects** a set A of edges if A does not contain any cross edges.
- A cross edge of minimum weight is called a **light edge**.



S={x,y,z,w}

V-S={a,b,c,d,e}

(w,a) and (c,z) are cross edges

(z,e) is a light edge

The cut respects {(x,w),(y,w),(c,d)}

The cut does not respect {(a,b),(d,z)}

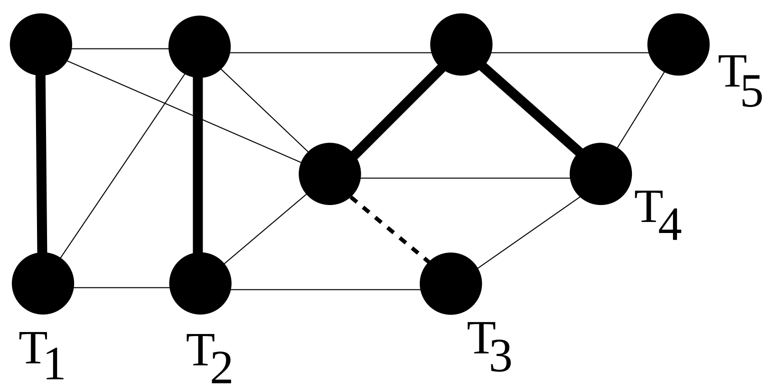
The Generic MST algorithm

- Let A be the edges a minimal spanning tree of G.
- The MST algorithm “grows” the spanning tree one edge at a time.
- It starts with set $A = \emptyset$, which is clearly a subset of every minimum spanning tree.
- At each step, the algorithm adds an edge (u, v) to A so that the set $A \cup \{(u, v)\}$ is a subset of some minimum spanning tree.
- Such an edge (u, v) is called a **safe edge**, because we can safely add it to the set A and still continue.
- The algorithm is simple:

```
MST(G)
 A=EmptyList
 While ! IsSpanningTree(G,A)
     e = SafeEdge(G,A)
     Insert(A,e)
 return A
```

Properties of A during MST

- Let A be the edges in a partial solution to MST.
- The graph $G_A = (V, A)$ is a forest.
- Each tree in the forest G_A is a **connected component**.
- Some of the trees in the forest consist of just a single node.
- At every step of the algorithm, MST adds an edge to the set A.
- The result of this is a merger of two trees into one.
- **Example**



Are There Safe Edges?

- The algorithm assumes that we can always find a safe edge.
- We can easily argue this:
  - At the beginning of the algorithm, $A = \emptyset$, and any edge in any minimum spanning tree is safe.
  - During each iteration, we add a safe edge to A.
  - Since we added a safe edge, then A is still contained in some minimal spanning tree T.
  - Thus, any edge from $T - A$ is safe for A.
- Now we know there are safe edges. How do we find them?
- Actually, it's not that hard to find safe edges, as we will see next.

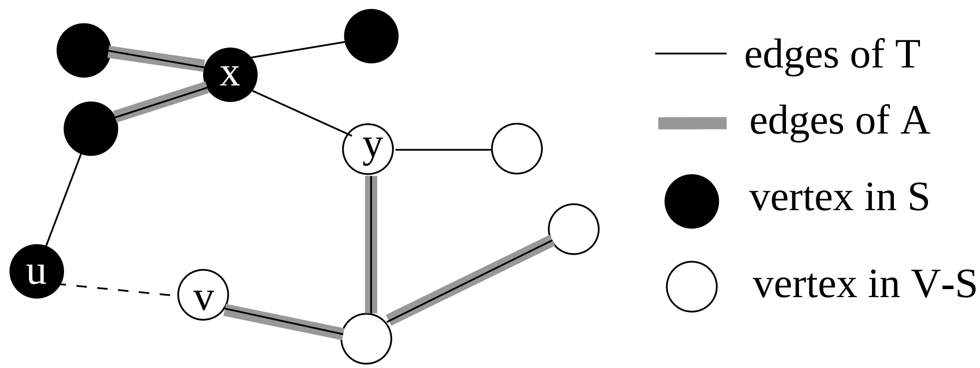
Finding Safe Edges: Part 0

- **Theorem 0:** Let
  - $G=(V,E)$ be a connected, weighted graph,
  - $A \subseteq E$ a subset of some MST for G,
  - $(S, V-S)$ be any cut of G that respects A, and
  - (u,v) be a light edge of $(S, V-S)$.

  Then the edge (u,v) is safe for A.
- **Proof:**
  - Let T be a MST of G containing A.
  - If $(u,v) \in T$, then (u,v) is safe for A, and we are done.
  - If $(u,v) \notin T$, we need to find an MST T' such that $A \cup \{(u,v)\} \subseteq T'$
  - We will find an edge $(x,y) \in T$ such that the tree $T' = (T - \{(x,y)\}) \cup \{(u,v)\}$ is an MST for G that contains A.
  - This will mean that (u,v) is safe for A.

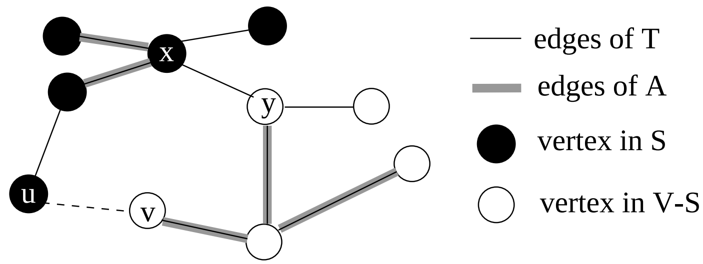
Proof of Theorem 0 Continued

- An illustration:



  - The graph $T \cup \{(u, v)\}$ contains a cycle.
  - Since (u, v) is a cross edge on the cycle, there must be another cross edge on the cycle.
  - Let (x, y) be such an edge.
  - **Claim:** $T' = (T - \{(x, y)\}) \cup \{(u, v)\}$ is an MST for G containing A, so that (u, v) is safe for A.
  - The edge (x, y) is not in A, because the cut respects A. Thus, A is a subset of T'.
  - Now all we need to show is that T' is an MST for G.

Proof of Theorem 0 Continued



- Proof that T' is an MST of G.
  - Since (u, v) is a light edge crossing $(S, V - S)$, $w(u, v) \leq w(x, y)$, since (x, y) is also a cross edge.
  - Thus $w(T') = w(T) + w(u, v) - w(x, y) \leq w(T)$
  - Since T is an MST, $w(T) \leq w(T')$.
  - Thus, $w(T) = w(T')$.
  - Then we have that T' is an MST for G.
- To summarize, we have found a tree T' such that
  - T' is an MST of G.
  - A is a subset of T'.
  - $(u, v) \in T'$, and $(u, v) \notin A$, so (u, v) is safe for A.

Finding Safe Edges: Part 1

- **Theorem 0:** Let
  - $G = (V, E)$ be a connected, weighted graph,
  - $A \subseteq E$ a subset of some MST for G,
  - $(S, V - S)$ be any cut of G that respects A, and
  - (u, v) be a light edge of $(S, V - S)$.

  Then the edge (u, v) is safe for A.
- We can use **Theorem 0** to prove:
- **Theorem 1:** Let
  - $G = (V, E)$ be a connected, weighted graph,
  - $A \subseteq E$ a subset of some MST for G, and
  - C be the edges in a connected component of $G_A = (V, A)$, and
  - (u, v) be a light edge of the cut $(C, V - C)$.

  Then (u, v) is safe for A.
- **Proof:** Since $(C, V - C)$ respects A, this follows from **Theorem 0**.

Interpreting Theorem 1

- **Theorem 1:** Let
  - $G = (V, E)$ be a connected, weighted graph,
  - $A \subseteq E$ a subset of some MST for G, and
  - C be the edges in a connected component of $G_A = (V, A)$, and
  - (u, v) be a light edge of the cut $(C, V - C)$.

  Then (u, v) is safe for A.
- **Theorem 1** basically says that if C is a subtree of an MST, and (u, v) is an edge of minimum weight with exactly one endpoint incident with C, then $C \cup \{(u, v)\}$ is a subtree of an MST for G.
- The following are applications of **Theorem 1**:
  - Let u be a vertex of G, and (u, v) an edge of minimum weight incident with u. Then (u, v) is contained in some MST of G.
  - If (u, v) is an edge of minimal weight in G, then (u, v) is contained in some MST of G.

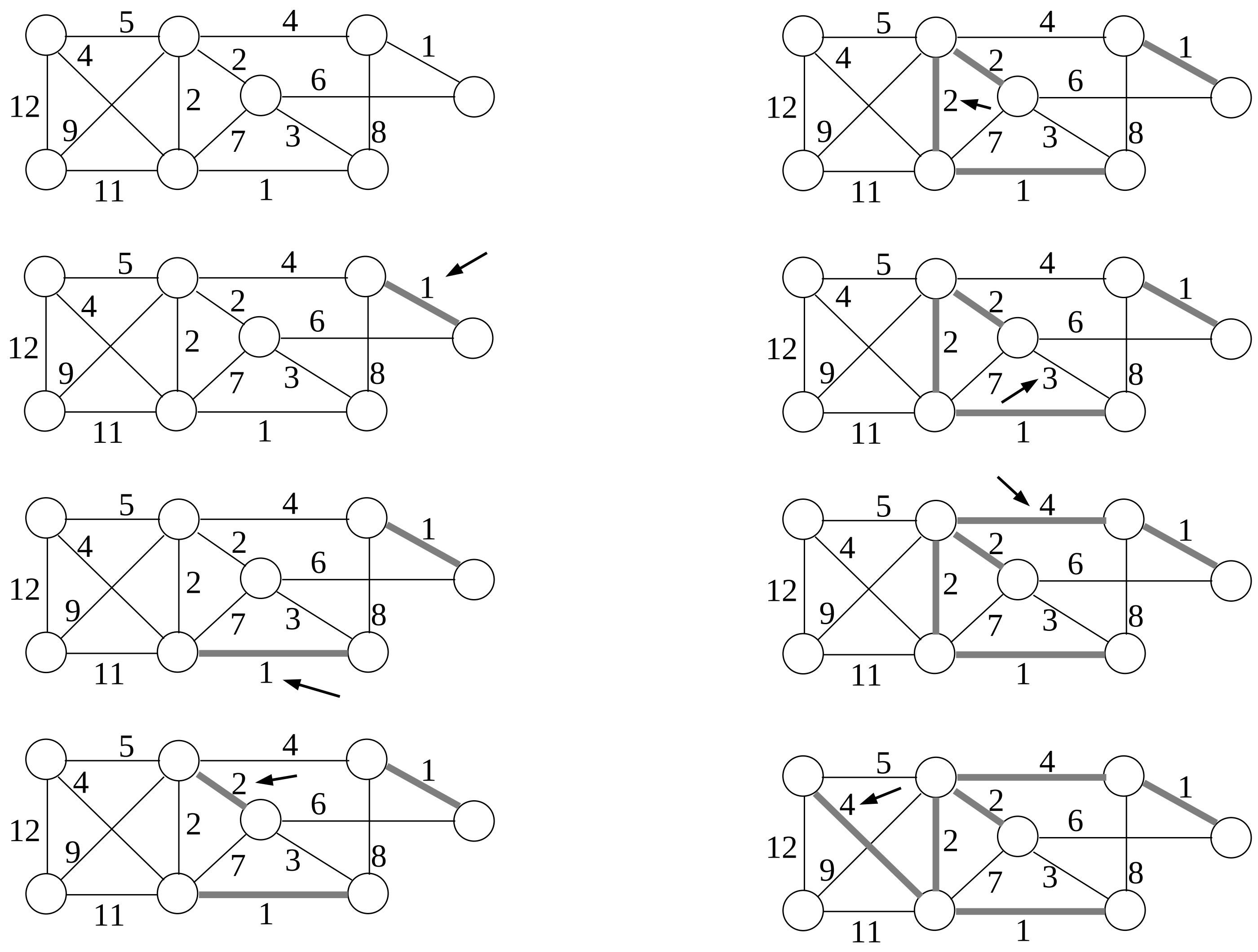
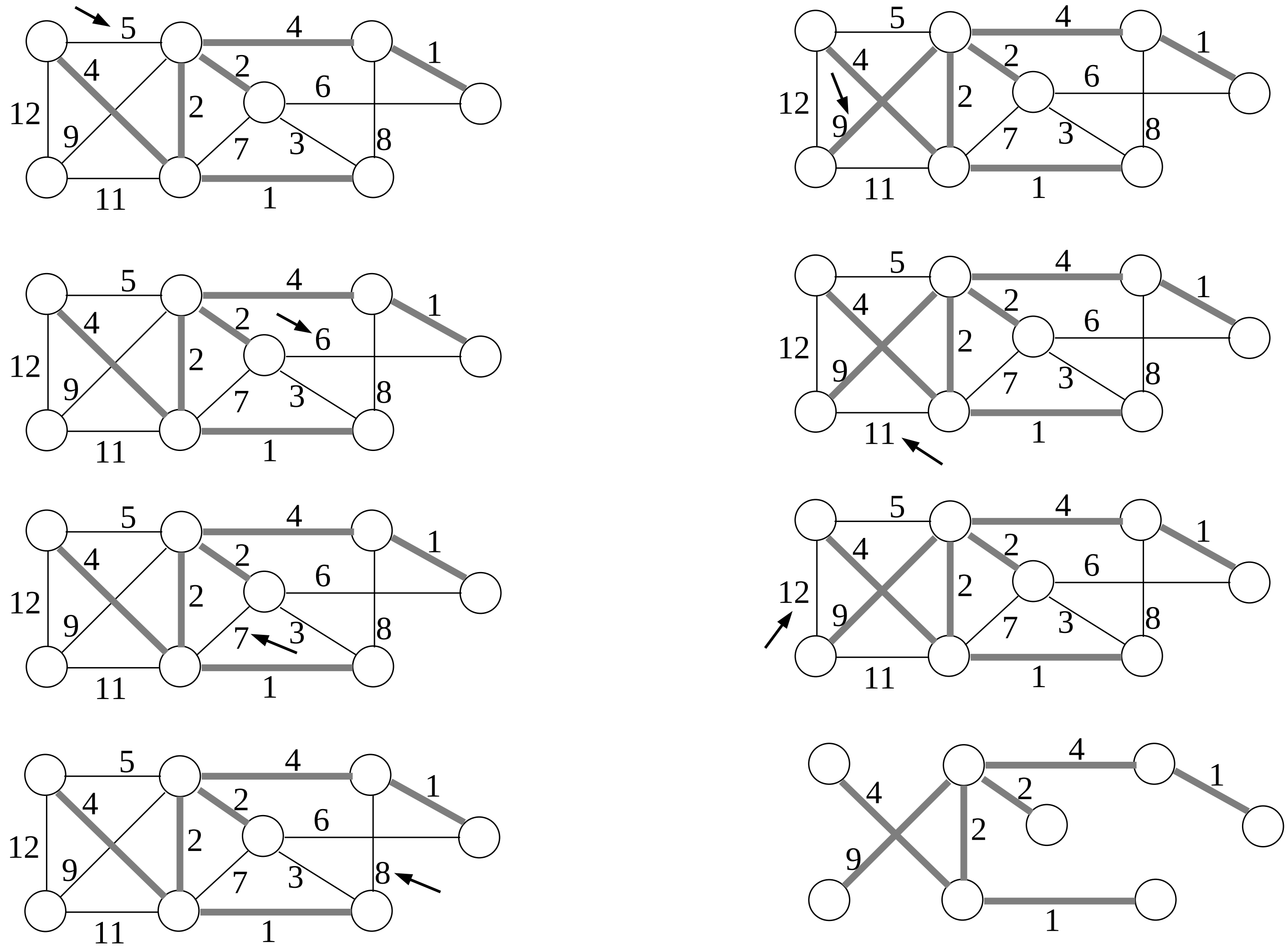
Applying Theorem 1

- **Theorem 1** is used by the two most common MST algorithms.
- **Kruskal's Algorithm**
  - Let $A = \emptyset$.
  - While A is not an MST
    - Add to A a minimum weight edge that does not form a cycle.
- **Prim's algorithm:**
  - Pick some vertex x.
  - Let $A = \{(x, y)\}$, where edge (x, y) has minimum weight of edges incident with x.
  - While A is not an MST
    - Add to A an minimum weight edge which has one endpoint incident with A
- We will take a closer look at each of these.

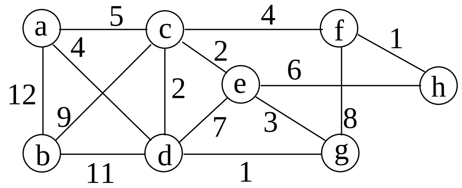
Kruskal's Algorithm

- Kruskal's algorithm is as follows.
  - Sort E in ascending order.
  - Set $A = \emptyset$
  - For I = 1 to $|E|$
    If $A \cup \{E[I]\}$ does not contain a cycle
    $A = A \cup \{E[I]\}$
  - Return A.
- When does $A \cup \{E[I]\}$ contains a cycle?
  - As the algorithm progresses, A is a forest.
  - Edges connecting two vertices in the same tree will create a cycle.
  - Edges that goes from one tree to another will not create a cycle.
  - We will store each tree in a separate set.
  - Adding an edge connect two trees, so we merge the sets.
  - We can now rewrite Kruskal's algorithm.

The Real Kruskal's Algorithm

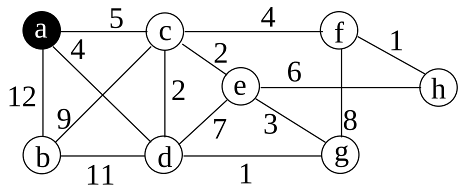
- ```
  Kruskal_MST(G)
     A=EmptySet
     ForAll v in V[G]
         Create_Set(v)
     SortAscending(E[G])
     ForAll edges e=(u,v) //sorted order
         If Set(u) != Set(v)
            Insert(A,e)
            Set_Union(u,v)
     Return A
  ```
- Let $n = |V|$ and $m = |E|$.
- It is possible to implement the sets so that the combined cost of the set operations is $O(m \log m)$ (we won't go into the details here).
- The sorting takes $O(m \log m)$ time.
- Thus, the the complexity of Kruskal's Algorithm is $O(m \log m)$= $O(m \log n)$ (since $O(\log m) = O(\log n^2) = O(\log n)$).
  ```

Kruskal's Algorithm Example



Kruskal's Algorithm Example (continued)



Prim’s Algorithm Background

- Unlike Kruskal’s algorithm, with Prim’s algorithm we grow a single tree A into a minimum spanning tree.
- An arbitrary vertex r is picked, and the tree is grown from that vertex.
- At each step a light edge of the cut $(A, v - A)$ is added to A.
- Thus, we add a node and an edge to A at each step.
- Since A is a tree, it remains a tree with the added edge and node.
- We need to have an efficient (greedy) way to determine the light edge at each step.
- Notice that according to **Theorem 1**, this method will produce an MST.

Prim's Algorithm–More Details

- For each node x, we will store
  - The predecessor $p(x)$. This is the vertex y in A which we join x to when edge (x, y) is added to A.
  - The $key(x)$. This is the minimum weight edge that connects x to some vertex in A.
- $key(r) = 0$, and $p(r) = NULL$ throughout.
- Each node $x \neq r$ starts with $key(x) = \infty$.
- The value $key(x)$ only changes if some neighbor of x is added to A.
- Thus, when we add a node y to A, we need to update the key values of the nodes adjacent to y.
- We will store the vertices in $V - A$ in a priority queue Q based on $key(x)$. This allows us to pick the minimum weight edge to add to A.
- We don't explicitly store A. The MST is reconstructed using the predecessors $p(x)$ for all $p \neq r$.

Prim's Algorithm

- ```
  Prim_MST(G,r)
    PriorityQueue Q=V[G]
    ForAll u in Q
        key[u]=Max_Int
    key[r]=0
    p[r]=NULL
    While NotEmpty(Q)
        u=ExtractMin(Q)
        ForAll v adjacent to u
          if(v in Q and w(u,v) < key[v])
              key[v]=w(u,v) // not constant (why?)
              p[v]=u
  ```
- Notice that at each step we add a vertex with minimum key, and then update the key values for its neighbors.
- The complexity is $O(n \log n + m \log n) = O(m \log n)$, assuming we use a binary heap to implement the priority queue, and an auxiliary array to keep track of the vertices that are still in the priority queue.
  ```

Prim's Algorithm Example



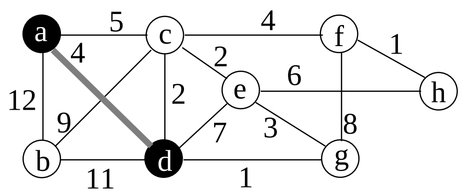
v	a	b	c	d	e	f	g	h
k	0	∞	∞	∞	∞	∞	∞	∞
p	nil	?	?	?	?	?	?	?

Q=[a,b,c,d,e,f,g,h]



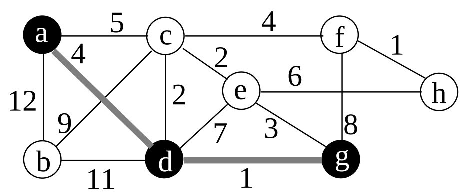
v	a	b	c	d	e	f	g	h
k	~~0~~	12	5	4	∞	∞	∞	∞
p	nil	a	a	a	?	?	?	?

Q=[d,c,b,e,f,g,h]



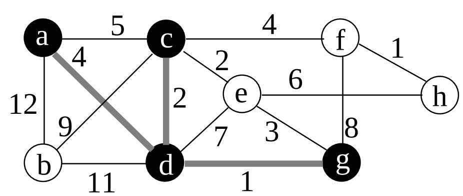
v	a	b	c	d	e	f	g	h
k	~~0~~	11	2	~~4~~	7	∞	1	∞
p	nil	d	d	a	d	?	d	?

Q=[g,c,e,b,f,h]



v	a	b	c	d	e	f	g	h
k	~~0~~	11	2	~~4~~	3	8	~~1~~	∞
p	nil	d	d	a	g	g	d	?

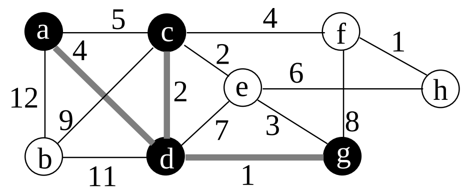
Q=[c,e,f,b,h]



v	a	b	c	d	e	f	g	h
k	~~0~~	9	~~2~~	~~4~~	2	4	~~1~~	∞
p	nil	c	d	a	c	c	d	?

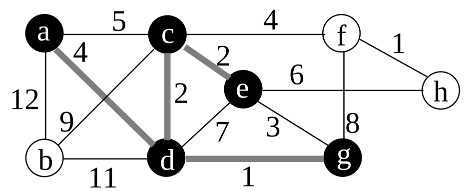
Q=[e,f,b,h]

Prim’s Algorithm Example (continued)



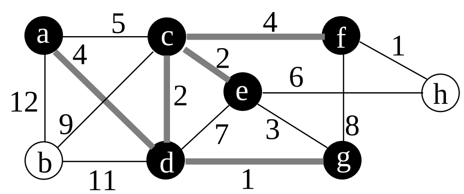
v	a	b	c	d	e	f	g	h
k	~~0~~	9	~~2~~	~~4~~	2	4	~~1~~	∞
p	nil	c	d	a	c	c	d	?

Q=[e,f,b,h]



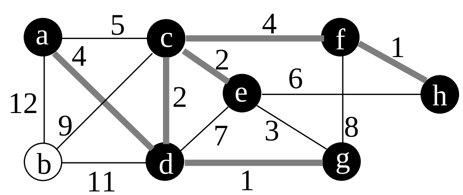
v	a	b	c	d	e	f	g	h
k	~~0~~	9	~~2~~	~~4~~	~~2~~	4	~~1~~	6
p	nil	c	d	a	c	c	d	e

Q=[f,h,b]



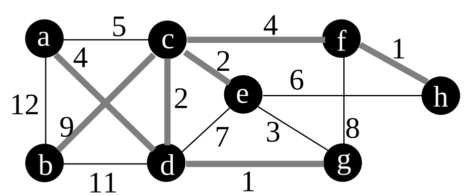
v	a	b	c	d	e	f	g	h
k	~~0~~	9	~~2~~	~~4~~	~~2~~	~~4~~	~~1~~	1
p	nil	c	d	a	c	c	d	f

Q=[h,b]



v	a	b	c	d	e	f	g	h
k	~~0~~	9	~~2~~	~~4~~	~~2~~	~~4~~	~~1~~	~~1~~
p	nil	c	d	a	c	c	d	f

Q=[b]



v	a	b	c	d	e	f	g	h
k	~~0~~	~~9~~	~~2~~	~~4~~	~~2~~	~~4~~	~~1~~	~~1~~
p	nil	c	d	a	c	c	d	f

Q=[]

Dijkstra’s Algorithm

Dijkstra’s shortest path algorithm is almost identical to Prim’s algorithm (changes marked by `<--`).

```
Dijkstra(G,r)
  PriorityQueue Q=V[G]
  ForAll u in Q
      key[u]=Max_Int
  key[r]=0
  p[r]=NULL
  While NotEmpty(Q)
      u=ExtractMin(Q)
      ForAll v adjacent to u
        if(key[u] + w(u,v) < key[v])  <--
            key[v]=key[u] + w(u,v)    <--
            p[v]=u
```