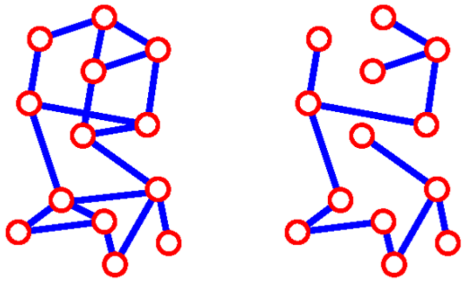
Minimum Spanning Trees

Outline: MST

- Minimum Spanning Tree
- Generic MST Algorithm
- Kruskal’s Algorithm (Edge Based)
- Prim’s Algorithm (Vertex Based)

Spanning Tree

- A **spanning tree** of *G* is a subgraph which
  - is tree (acyclic)
  - and connect all the vertices in *V*.



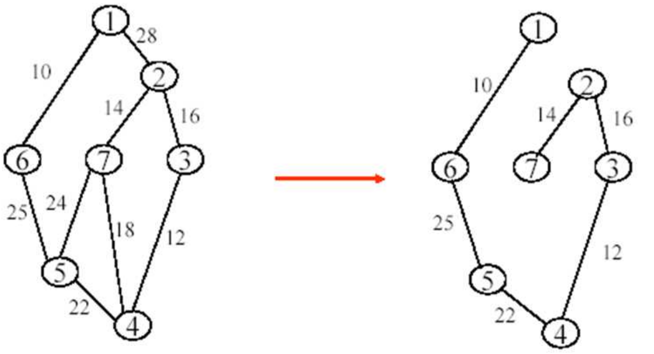
G spanning tree of **G**

  - Spanning tree has only $|V| - 1$ edges.

Minimum Spanning Tree

- **Input:**
  - Undirected connected graph $G = (V, E)$ and weight function $w : E \rightarrow \mathbf{R}$,
- **Output:**
  - A Minimum spanning tree *T*: tree that connects all the vertices and minimizes $w(T) = \sum_{(u,v)\in T} w(u,v)$
- Greedy Algorithms
  - Generic MST algorithm
  - Kruskal's algorithm
  - Prim's algorithm

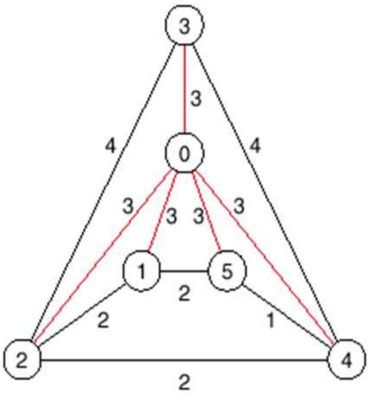
Example: MST



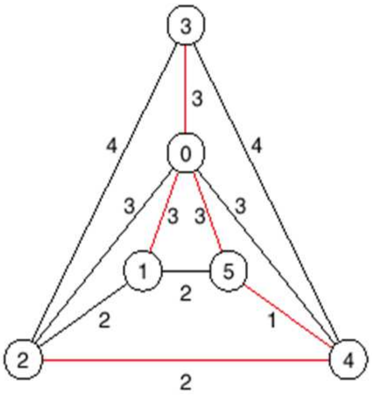
G

Minimum Cost Spanning Tree

Example: MST



Weight sum of tree = 15



Weight sum of tree = 12

Generic MST Algorithm

- Our goal is to build a spanning tree by adding one edge at a time to a **set** ***A*** in a “***greedy***” fashion.
- Basically, we just need to somehow guarantee ourselves that at each step, the current set can be “extended” to an MST.
- Strategy:
  - Grow the MST ***one edge at a time***, ensuring that the partial solution remains a subset of some MST
- How do we do that?

Generic MST Algorithm

- Let’s assume that our current set of edges ***A*** already satisfies the property that ***A*** can be extended to an MST.
- Question:
  - What edges can we add to ***A*** to maintain the property?
- Answer:
  - an edge ***e*** such that ***w(e) ≤ w(other edges)*** and
  - $A \cup \{e\}$ is ***acyclic***
- We will call such edge a ***safe edge*** if it also doesn’t create a cycle when added to ***A***.

Generic MST Algorithm

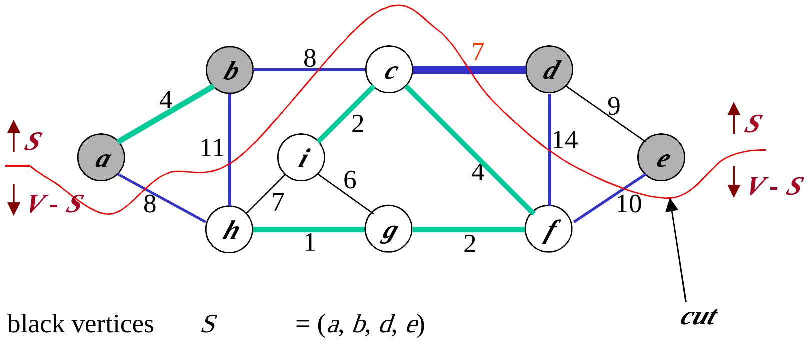
```
Generic-MST(G, w)
1. A = { }
2. while A does not form a spanning tree
3.     find an edge (u, v) that is safe for A
4.         Add (u, v) to A
5. return A
```

- How to find a **safe** edge?

Generic MST Algorithm

- Definitions:
  - A ***cut*** $(S, V - S)$ of $G = (V, E)$ is a partition of V into 2 sets
  - An edge $(u, v) \in E$ ***crosses*** the cut $(S, V - S)$ if one point is in S while the other point in $V - S$
  - A ***cut respects a set A of edges*** if **no edges** in A ***crosses*** the cut.
  - An edge is ***light*** if its weight is **minimum** of all edges crossing the cut

Generic MST Algorithm



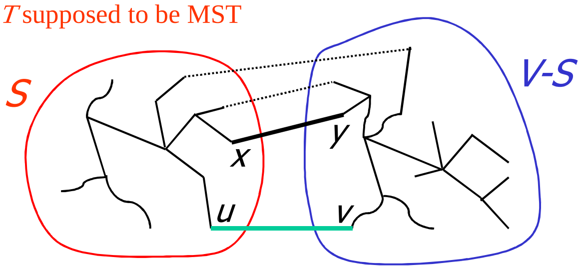
black vertices	S = (a, b, d, e)
white vertices	V-S = (c, i, h, g, f)
red line	*cut*
blue	edges crossing the cut
(*c*, *d*)	only light edge
green	a subset ***A*** of edges; which is a cut (S, V-S) **respects** to ***A***, since no edge of ***A*** crosses the cut.

Theorem 23.1 (CLRS)

- Let $G = (V, E)$ be a *connected, undirected* graph
- w is weight function $w: E \rightarrow \mathbf{R}$
- Let $A \subseteq E$ be included in some MST T for G
- Let (S, V - S) be any cut of G that respect A
- Let (u, v) be a light edge (**min-weight**) crossing the cut (S, V - S)
- Then, edge (u, v) is safe for A
    - i.e., $(u, v) \in T$, a **MST** of G

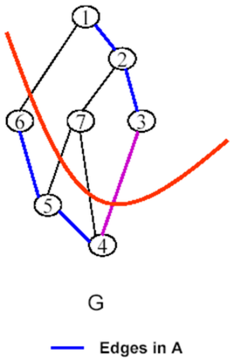
Theorem 23.1 (CLRS)

- Proof
  - suppose $(u, v) \notin T$
  - look at path from u to v in T
  - swap (x, y) with (u, v)
    - ➢ the first edge on path from u to v in T that crosses from S to $V - S$
  - this increases $w(T)$ – contradiction
    - ➢ T supposed to be MST



Generic MST Algorithm

- **Corollary 23.2 (CLRS):**
  - Let $A \subseteq E$ be included in some MST
  - Let $C_1 = (V_{C1}, E_{C1})$ and
  - $C_2 = (V_{C2}, E_{C2})$ be two distinct connected components (trees) in the forest $G_A = (V, A)$.
  - If (u, v) is a *light* edge crossing the cut = (V_{C1}, V_{C2}) then (u, v) is *safe* for A.



Generic MST Algorithm

```
Generic-MST($G$, $w$)
1.  $A$ = { }
2.  while $A$ does not form a spanning tree
3.          find an edge ($u$, $v$) that is safe for $A$
4.                  Add ($u$, $v$) to $A$
5.  return $A$
```

- As the algorithm proceeds:
  - A is always acyclic
- At any point of the execution of the algorithm,
  - Graph $G_A = (V, A)$ is a forest, and
  - Each ***connected component*** is a tree
- Also, any safe edge (u, v) for A
  - Connects distinct components of G_A,
  - Since $A \cup (u, v)$ must be acyclic

Generic MST Algorithm

```
Generic-MST($G$, $w$)
1.  $A$ = { }
2.  while $A$ does not form a spanning tree
3.          find an edge ($u$, $v$) that is safe for $A$
4.                  Add ($u$, $v$) to $A$
5.  return $A$
```

- The loop in lines 2-4 is executed $|V|$-1 times
- Initially when $A = \{\}$, there are $|V|$ trees in G_A
  - Each tree has only one vertex
- When G_A (forest) contains only a single tree, the algorithm terminates

Kruskal's Algorithm

- **Edge based** algorithm
- Greedy strategy:
  - From the remaining edges, select a ***least-cost*** edge that ***does not result in a cycle*** when added to the set of already selected edges
  - Repeat $|V|$-1 times

Kruskal's Algorithm

- INPUT:
  - edge-weighted graph $G = (V, E)$, with $|V| = n$
- OUTPUT:
  - a spanning tree A of G
    - touches all vertices,
    - has n-1 edges
    - of minimum cost (= total edge weight)
- Algorithm:
  - Start with A empty,
  - Add the edges one at a time, in increasing weight order
  - An edge is accepted, if it connects vertices of distinct trees (if the edge does not form a cycle in A)
  - until A contains n-1 edges

Kruskal's Algorithm

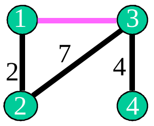
```
MST-Kruskal($G$, $w$)
1 $A \leftarrow \emptyset$
2 for each vertex $v \in V[G]$ do
3    Make-Set($v$)
4 sort the edges of $E$ by nondecreasing weight $w$
5 for each $(u, v) \in E$, in nondecreasing of weight do
6   if Find-Set($u$) $\neq$ Find-Set($v$) then
7      $A \leftarrow A \cup \{(u, v)\}$
8      Union(Set($u$),Set($v$))
9 return $A$
```

Kruskal's Algorithm

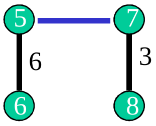
- **Lines 1-3 initialize the set *A* to empty set and create |V| trees, one containing each vertex.**
- **The edges in *E* are sorted into nondecreasing order by weight in line 4.**
- **The for loop in lines 5-8 checks, for each (u, v), whether the endpoints u and v belong to the same tree.**
    - **If they do, then the edge (u, v) cannot be added to the forest without creating a cycle, and the edge is discarded.**
    - **Otherwise, the two vertices belong to different trees.**
    - **In this case, the edge (u, v) is added to A in line 7, and the vertices in the two trees are merged in line 8.**

Data Structures For Kruskal’s Algorithm

- Does the addition of an edge (u, v) to A result in a cycle?
- Each component of A is a tree.
  - When u and v are in the
    - **same** component, the addition of the edge (u, v) creates a **cycle**.

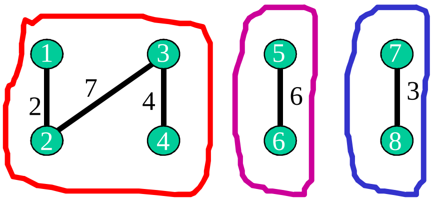


    - **different** components, the addition of the edge (u, v) **does not** create a cycle.



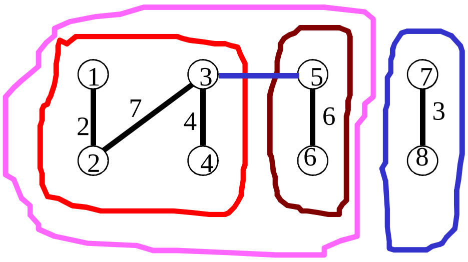
Data Structures For Kruskal’s Algorithm

- Each component of A is defined by the vertices in the component.
- Represent each component as a set of vertices.
  - {1, 2, 3, 4}, {5, 6}, {7, 8}
- Two vertices are in the same component iff they are in the same set of vertices.



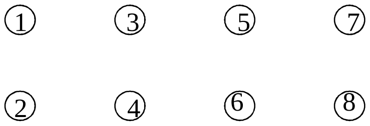
Data Structures For Kruskal’s Algorithm

- When an edge (u, v) is added to A, the two components that have vertices u and v combine to become a single component
- In our set representation of components, the set that has vertex u and the set that has vertex v are united.
  - {1, 2, **3**, 4} + {**5**, 6} ➔ {1, 2, 3, 4, 5, 6}



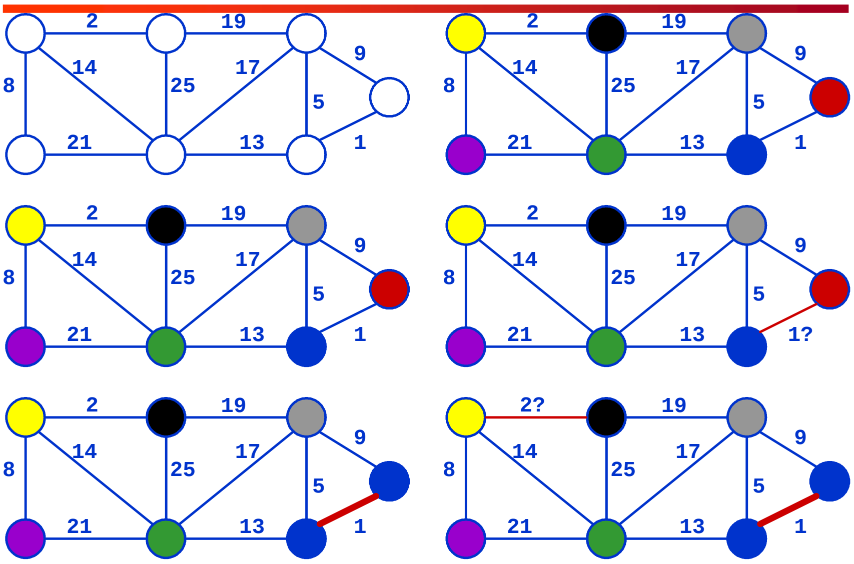
Data Structures For Kruskal’s Algorithm

- Initially, A is empty

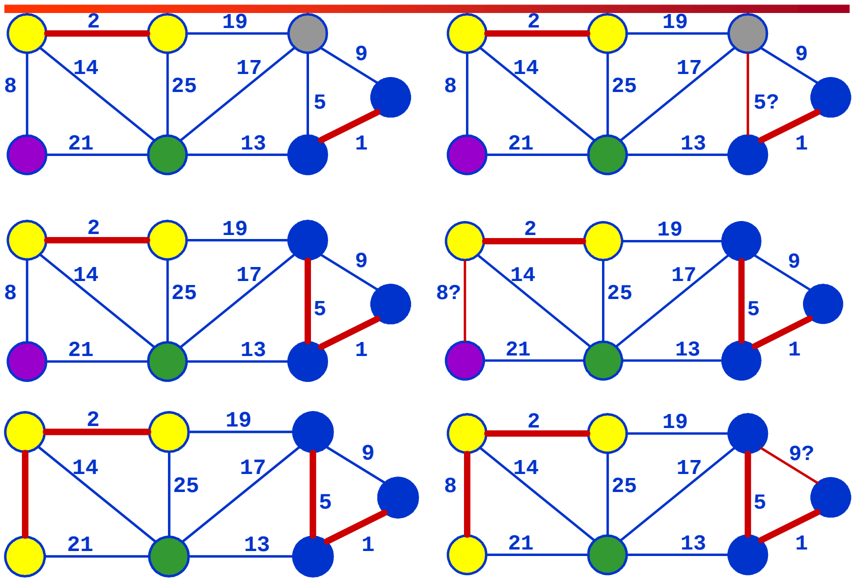


- Initial sets are:
  - {1} {2} {3} {4} {5} {6} {7} {8}
- Does the addition of an edge (u, v) to A result in a cycle? If not, add edge to A
  - s_1 = Find-Set(u); s_2 = Find-Set(v);
  - if ($s_1 \neq s_2$) then Union(s_1, s_2);

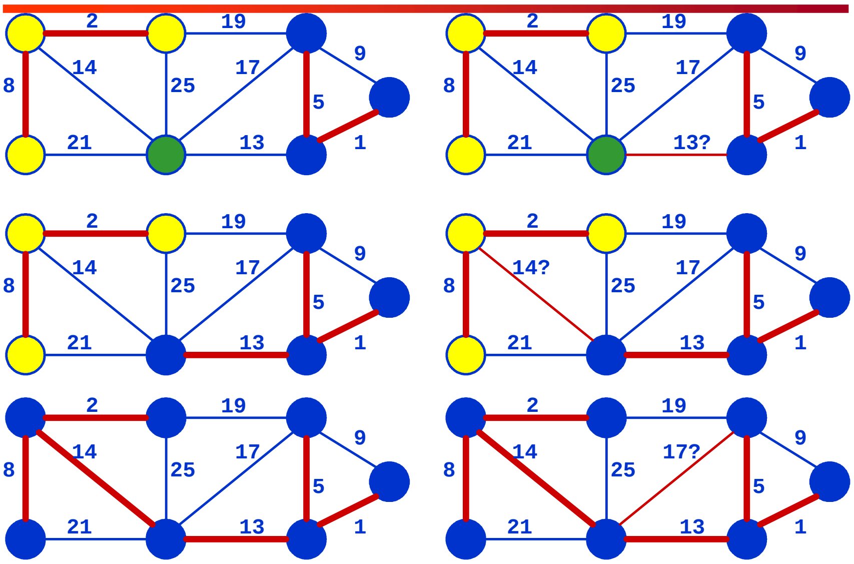
Kruskal's Algorithm



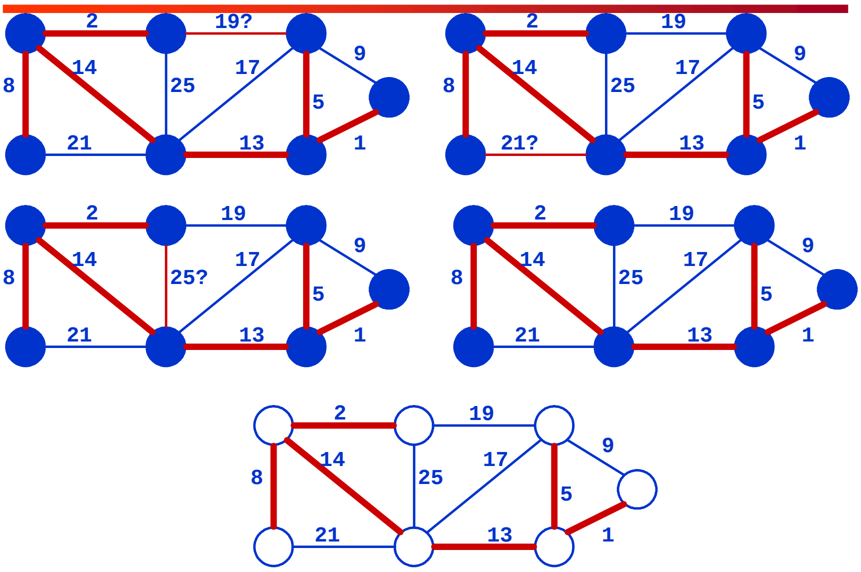
Kruskal's Algorithm



Kruskal’s Algorithm



Kruskal’s Algorithm



Kruskal's Algorithm

```
MST-Kruskal(G, w)
1 A ← ∅
2 for each vertex v ∈ V[G] do
3    Make-Set(v)
4 sort the edges of E by nondecreasing weight w
5 for each (u, v)∈E, in nondecreasing of weight do
6   if Find-Set(u) ≠ Find-Set(v) then
7      A ← A ∪ {(u, v)}
8      Union(Set(u),Set(v))
9 return A
```

Running Time of Kruskal's Algorithm

- Kruskal's Algorithm consists of two stages.
  - Initializing the set A in line 1 takes $O(1)$ time.
  - Sorting the edges by weight in line 4.
    - takes $O(E \lg E)$
  - Performing
    - $|V|$ MakeSet() operations   for loop in lines 2-3.
    - $|E|$ FindSet(),   for loop in lines 5-8.
    - $|V|$ - 1 Union(),   for loop in lines 5-8.
    - which takes $O(V + E)$
- The total running time is
  - $O(E \lg E)$
  - Observing that $|E| < |V|^2$ we have $\lg |E| = O(\lg V)$,
    So total running time becomes $O(E \lg V)$.

Prim’s Algorithm

- Prim’s algorithm has the property that edges in the set A always form a single tree.
- The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V.
- At each step, a light edge is added to the tree A that connects A to an isolated vertex of $G_A = (V, A)$.
  - Adds only edges that are safe for A.
  - When algorithm terminates, edges in A form MST.
  - MST A for G: $A=\{(v, p[v]): v \varepsilon V-\{r\}\}$.
- **Vertex based** algorithm.
- Grows one tree, **one vertex at a time**

Prim’s Algorithm

```
MST-Prim(G,w,r) //G: graph with weight w and a root vertex r
for each u ∊ V[G]
  key[u] ← ∞
  p[u] ← NIL
key[r] ← 0
Q = BuildMinHeap(V,key); // Q – vertices out of T
while Q ≠ ∅ do
  u ← ExtractMin(Q)  // making u part of T
     for each v ∊ Adj[u] do
        if v ∊ Q and w(u,v) < key[v] then
            p[v]      ← u
            key[v]    ← w(u,v)
```

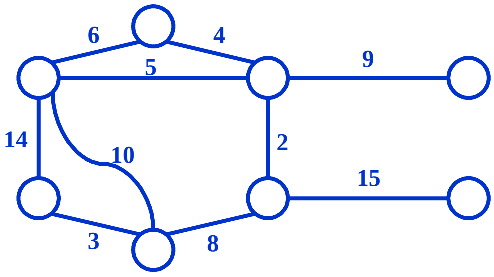
updating keys

- All vertices that are not in tree reside in min-priority queue Q based on a key field.
- For each vertex v, key[v] is min weight of any edge connecting v to a vertex in tree.
- key[v]= ∞ if there is no edge and p[v] names parent of v in tree.
- When algorithm terminates the min-priority queue Q is empty.

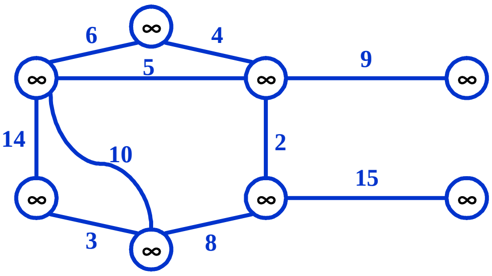
Prim's Algorithm

- Lines 1-5 set the key of each vertex to ∞ (except root r, whose key is set to 0 first vertex processed). Also, set parent of each vertex to NIL, and initialize min-priority queue Q to contain all vertices.
- Line 7 identifies a vertex $u \in Q$ incident on a light edge crossing cut (V-Q, Q) (except first iteration, u=r due to line 4).
- Removing u from set Q adds it to set Q- V of vertices in tree, thus adding (u, p[u]) to A.
- The for loop of lines 8-11 update *key* and p fields of every vertex v adjacent to u but not in tree.

Run on example graph

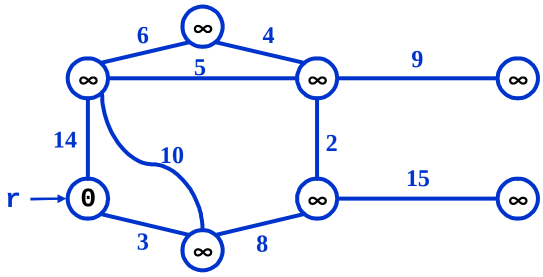


Run on example graph



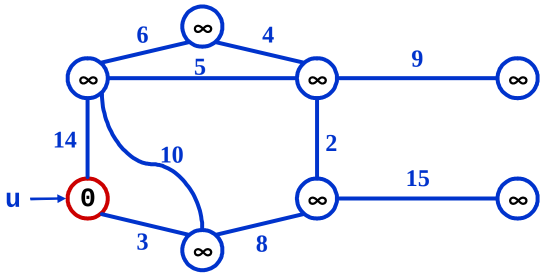
key[*u*] = ∞

Run on example graph



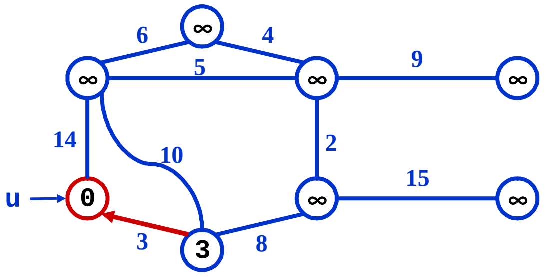
Pick a start vertex r

Run on example graph



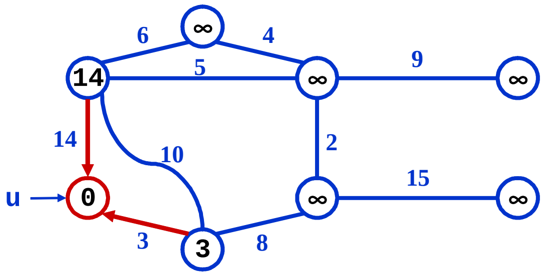
Red vertices have been removed from Q

Run on example graph

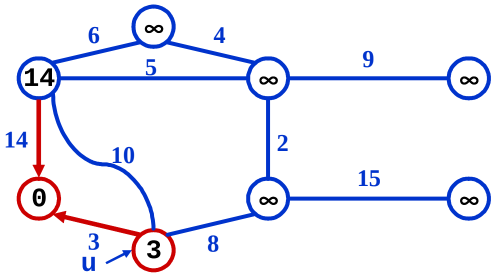


Red arrows indicate parent pointers

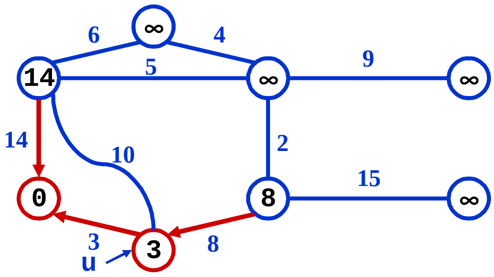
Run on example graph



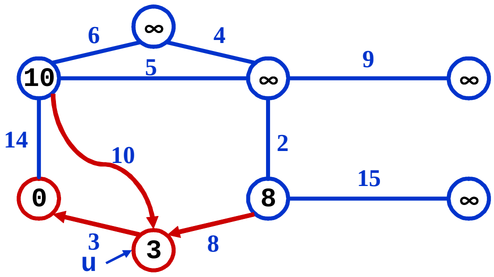
Run on example graph



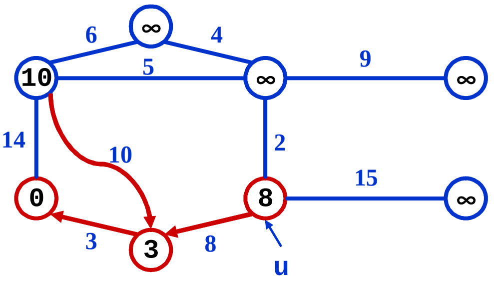
Run on example graph



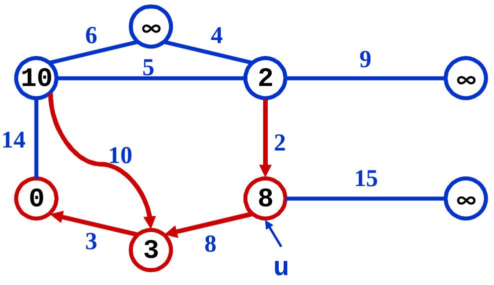
Run on example graph



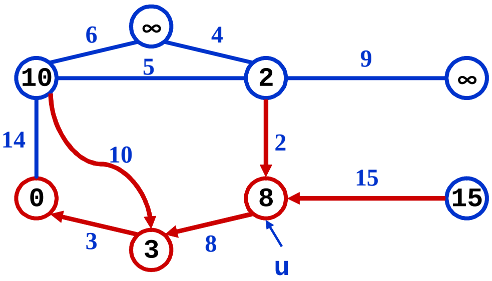
Run on example graph



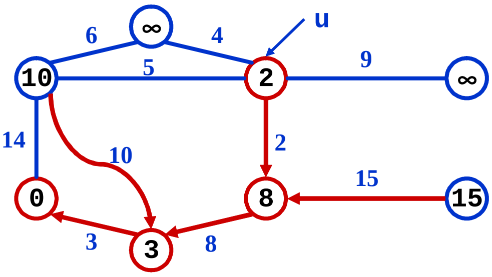
Run on example graph



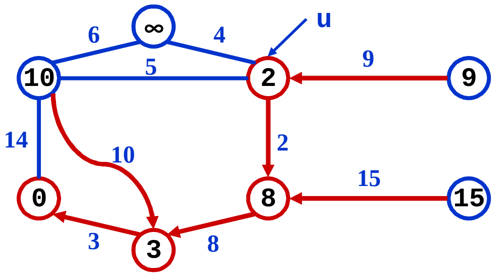
Run on example graph



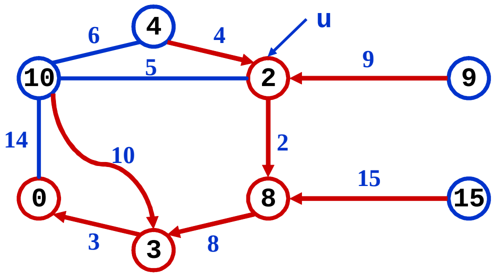
Run on example graph



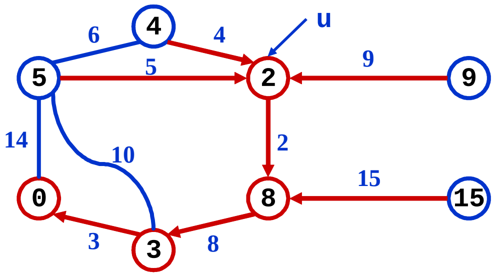
Run on example graph



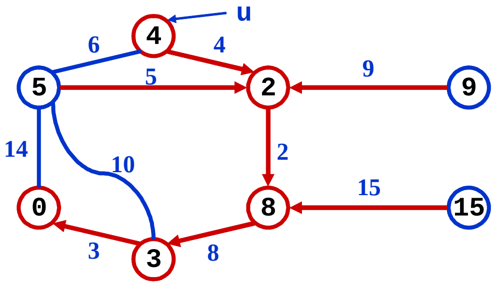
Run on example graph



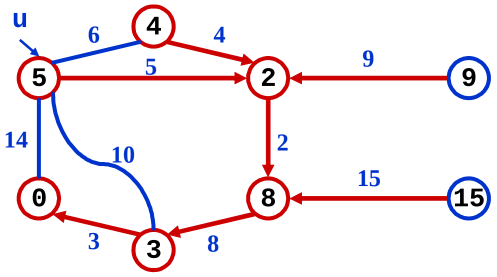
Run on example graph



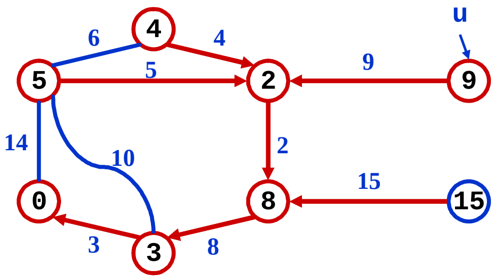
Run on example graph



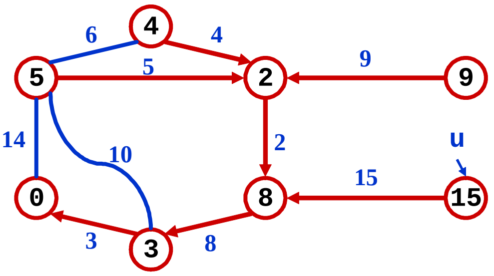
Run on example graph



Run on example graph



Run on example graph



Prim's Running Time

- *What is the hidden cost in this code?*

Extract-Min is executed |*V*| times

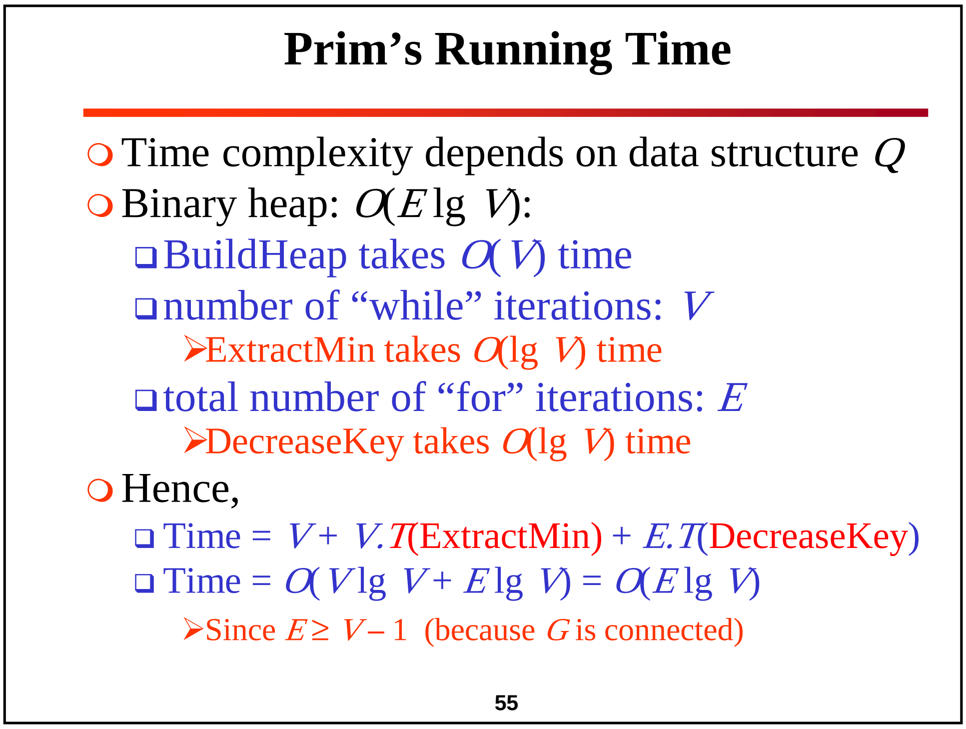
Decrease-Key is executed *O*(|*E*|) times

while loop is executed |*V*| times

```
MST-Prim(G,w,r)
for each u ∈ V[Q]
  key[u] ← ∞
  p[u] ← NIL
key[r] ← 0
Q = BuildHeap(V,key); //Q – vertices out of T
while Q ≠ ∅ do
  u ← ExtractMin(Q)   // making u part of T
     for each v ∈ Adj[u] do
          if v ∈ Q and w(u,v) < key[v] then
               p[v]     ← u
               key[v]   ← w(u,v)
```

updating keys

`DecreaseKey(v, w(u,v));`



Prim’s Running Time

- Time complexity depends on data structure Q
- Binary heap: $O(E \lg V)$:
  - BuildHeap takes $O(V)$ time
  - number of “while” iterations: V
    - ExtractMin takes $O(\lg V)$ time
  - total number of “for” iterations: E
    - DecreaseKey takes $O(\lg V)$ time
- Hence,
  - Time = $V + V.T(\text{ExtractMin}) + E.T(\text{DecreaseKey})$
  - Time = $O(V \lg V + E \lg V) = O(E \lg V)$
    - Since $E \geq V - 1$ (because G is connected)