Greedy

Greedy Algorithms

- Algorithm design technique
  - Usually more efficient than divide and conquer
  - But not always applicable
  - Constructs optimal solution incrementally by repeatedly choosing what looks promising *right now*
- Problem must satisfy the *greedy choice property*
  - *A locally optimal choice is guaranteed to lead to some globally optimal solution*

NCoins

- For coin values 1, 5, 10, 25
  - Give highest value coin you can at each step
  - This is a Greedy solution
    - Works for US coins
- When a 12 was added
  - The Greedy choice property no longer holds
  - We developed DAC/DP algorithms to solve
    - Choice was not “local,” but we “looked ahead”

END

Activity Selection

Activity Selection

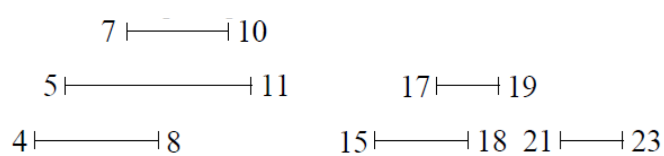
- *Input:* set $A = \{a_1, \ldots, a_n\}$ of n activities requesting *exclusive* access to a common resource

Starts at

$$a_i = [s_i, f_i)$$

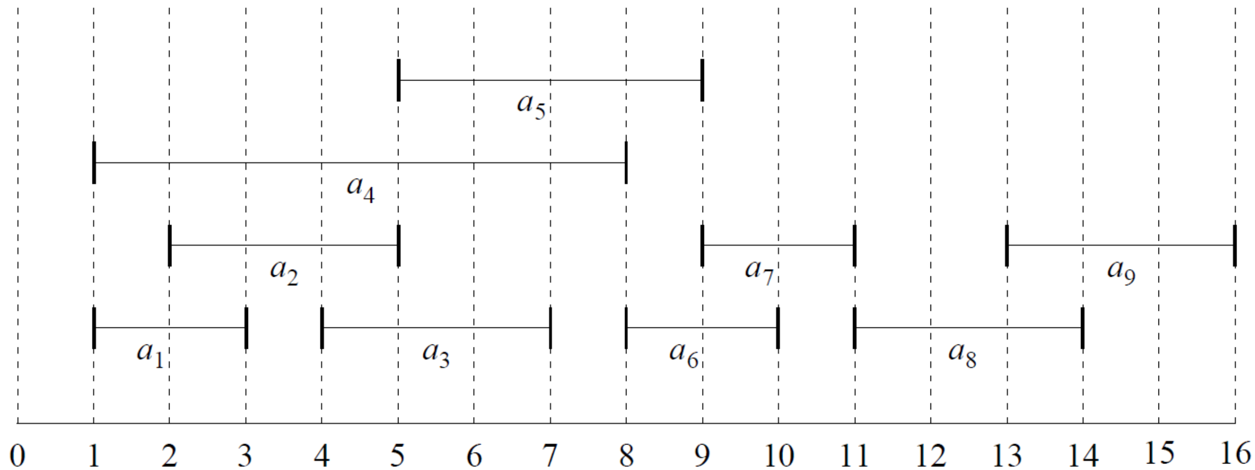
Finishes at

- *Output:* the largest cardinality set A of nonoverlapping activities
  - *Example:* schedule use of a room to maximize the number of events that use it



Example

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



Assume S sorted by finish time: $f_1 < f_2 < \ldots < f_n$

Optimal Substructure

- Let $S_{ij} = \{a_k \in S : f_i \leq s_k \leq f_k \leq s_j\}$, and let A_{ij} denote an optimal solution for S_{ij}.
- Activities in S_{ij} are compatible with:
  - Activities that finish no later than f_i
  - Activities that start no earlier than s_j
- $a_k \in A_{ij}$ generates two subproblems: S_{ik} and S_{kj}.
- Then, $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ (cut-and-paste argument).
- For convenience add sentinels $a_0 = (-\infty, -\infty)$ and $a_{n+1} = (\infty, \infty)$. Then $A = A_{0,n+1}$.

DAC

- Since optimal solution A_{ij} must contain optimal solutions to subproblems S_{ik} and $S_{kj} \Rightarrow$ can consider DAC (and DP)
  - $c[i, j]$ = size of optimal solution to S_{ij}
  - $c[i, j] = c[i, k] + c[k, j] + 1$, but what k?

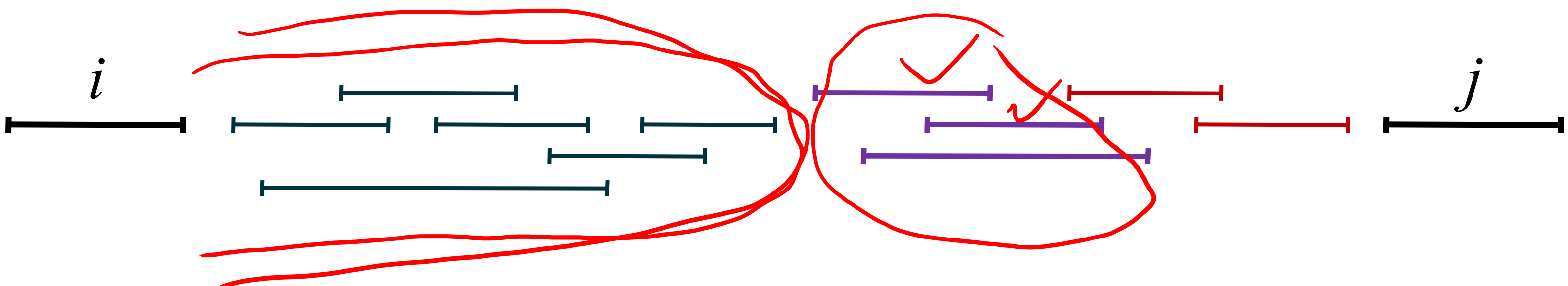
$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ 1 + \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k, j]\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

- Can develop memoized DAC or bottom-up DP

Overlapping Subproblems

- Will generate many repeated problems when evaluating

$$c[i, j] = 1 + \max_{a_k \in S_{ij}} \{c[i,k] + c[k, j]\}$$



- The blue intervals become the left subproblem of any of the purple intervals

END

Activity Selection

Greedy

Hallmark of Greedy Algorithms

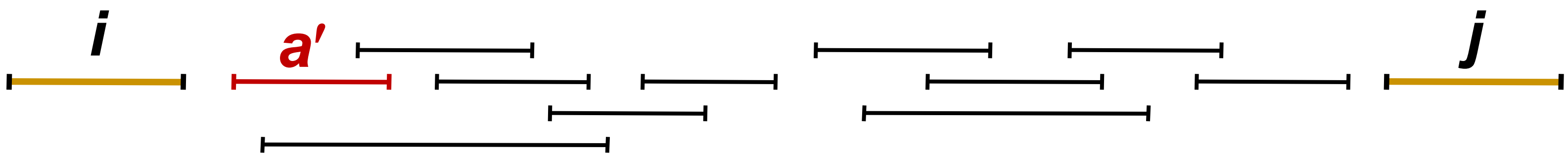
- *Greedy choice property:* a locally optimal choice leads to a globally optimal solution
  - Identify a simple heuristic to make the local choice.
  - Prove that the choices made are part of some optimal solution.
- In Activity Selection, choose an activity a_k to add to $A = A_{0,n+1}$ *before* solving the ensuing subproblems

- Some greedy choices: ~~shortest activity~~, activity that ends first, activity that ends last, activity that overlaps the fewest number of other activities, and so on

The Greedy Choice

- Among all activities in subproblem S_{ij} choose the one (a') that ends first
- No activity of S_{ij} ends before a' starts

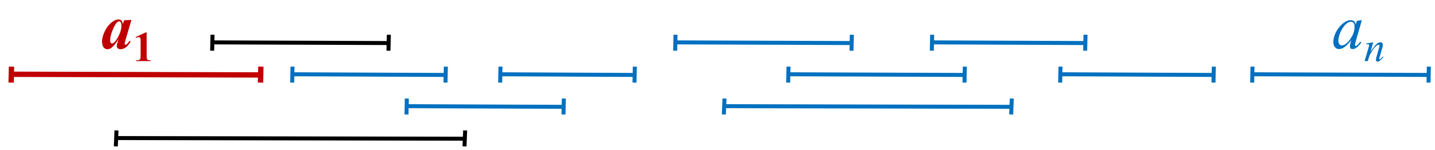
  $\Rightarrow$ Choosing a' eliminates subproblem S_{ik} and leaves S_{kj} only



- Making the greedy choice a_1 initially reduces the problem $S = S_{0,n+1}$ to problem $S_{1,n+1}$

The Greedy Choice

- Since we only have one subproblem, we can simplify the notation: let $S_k = \{a_i : s_i \geq f_k\}$
- Greedy choice $a_1 \Rightarrow \boldsymbol{S_1}$ is the only subproblem left
- By optimal substructure: If a_1 is in A, then A consists of a_1 plus all activities in an optimal solution to $\boldsymbol{S_1}$



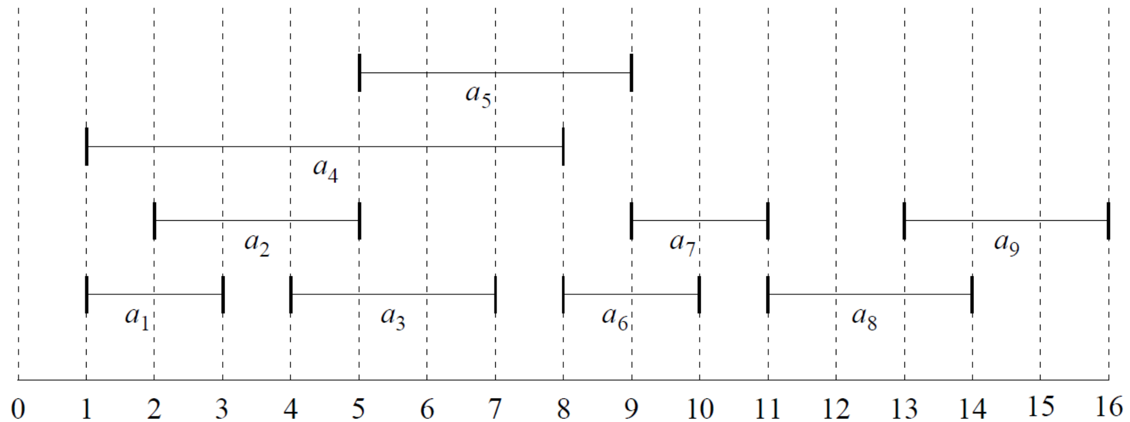
- Need to prove that a_1 is part of some optimal solution

- ***Theorem:*** If S_k is nonempty and a_m ends earliest in S_k, then a_m is part of *some* optimal solution to S_k.
- ***Proof***
- Let A_k be an optimal solution to S_k and a_r have the earliest finish time of all activities in A_k.
- If $a_r = a_m$, we are done. A_k is the desired solution.
- Otherwise, let $B_k = A_k - \{a_r\} \cup \{a_m\}$.
- The activities in B_k are disjoint.
- Since $|B_k| = |A_k|$, then B_k is optimal also and includes a_m.

Example

i	0	1	2	3	4	5	6	7	8	9
s		1	2	4	1	5	8	9	11	13
f	0	3	5	7	8	9	10	11	14	16

Subproblems: S_1 S_3 S_6 S_8



Greedy Activity Selection

n log n

```
def activitySelect(activities):
        assert type(activities)==list and
len(activities)>0
        actSorted = sorted(activities, key=finish)
        result = [actSorted[0]]
        last = 0
        for a in range(1,len(activities)):
                if actSorted[a].getStartTime() >= \

        actSorted[last].getFinishTime():
                        result.append(actSorted[a])
                        last = a
        return result
```

n

Time?

$\boxed{n \log n} + n$

Which Method to Use?

- Dynamic programming
  - Make a choice at each step.
  - Choice depends on solution to subproblems.
  - Solve subproblems first.
  - Solve bottom-up.
- Greedy
  - Make a choice at each step.
  - Make the choice *before* solving the subproblems (and then solve the remaining subproblems).
  - Solve top-down.

END

Minimal Spanning Tree

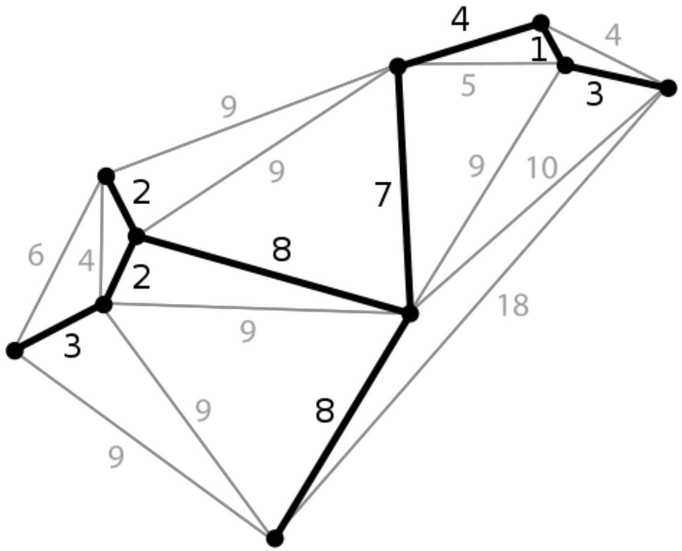
Minimum Spanning Tree

- A tree connecting all nodes of an undirected graph with minimal cost
- Many applications
  - Approximate solution of NP-hard problems
  - Basis of AT&T original billing system
  - Construction of LDPC codes
  - Image registration with Renyi entropy
  - Learning for real-time face verification
  - Reducing data storage for sequencing amino acids in proteins
  - Optimal clustering
  - Particle interactions in turbulent fluid flows
  - Autoconfig protocol to avoid cycles in a network

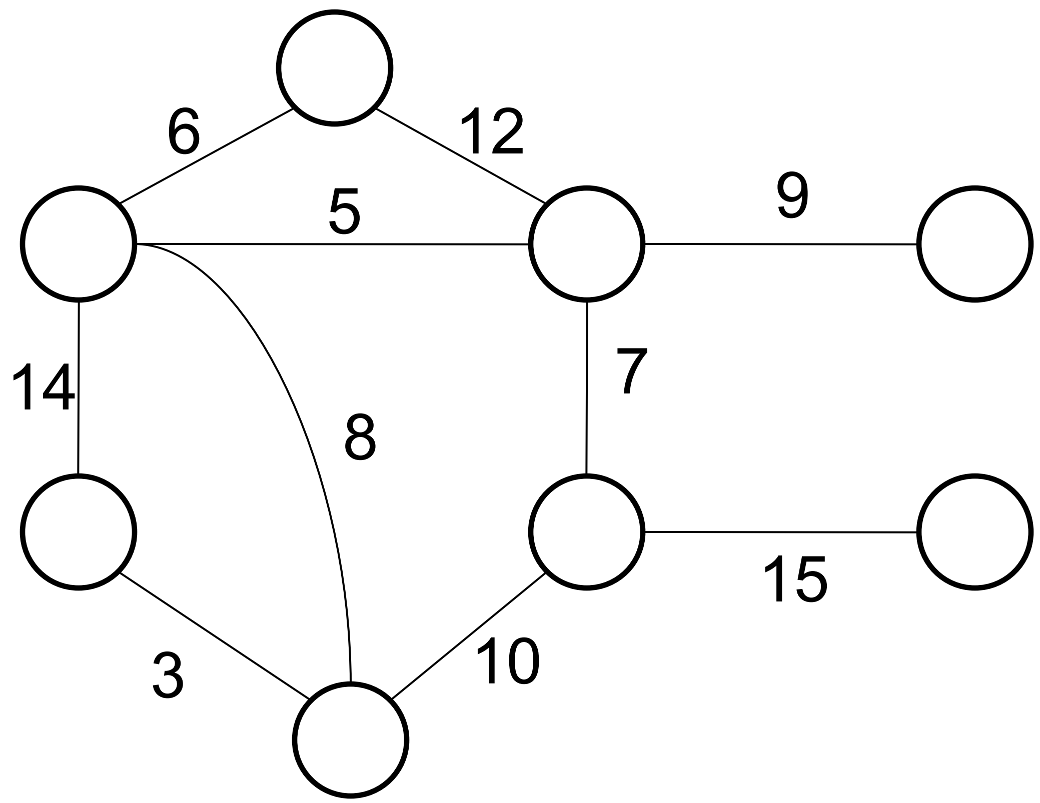
Minimum Spanning Trees

- *Input:* undirected connected graph $G(V,E)$ with weights w: $E \rightarrow R^+$
- *Output:* a tree T of minimum weight that spans V

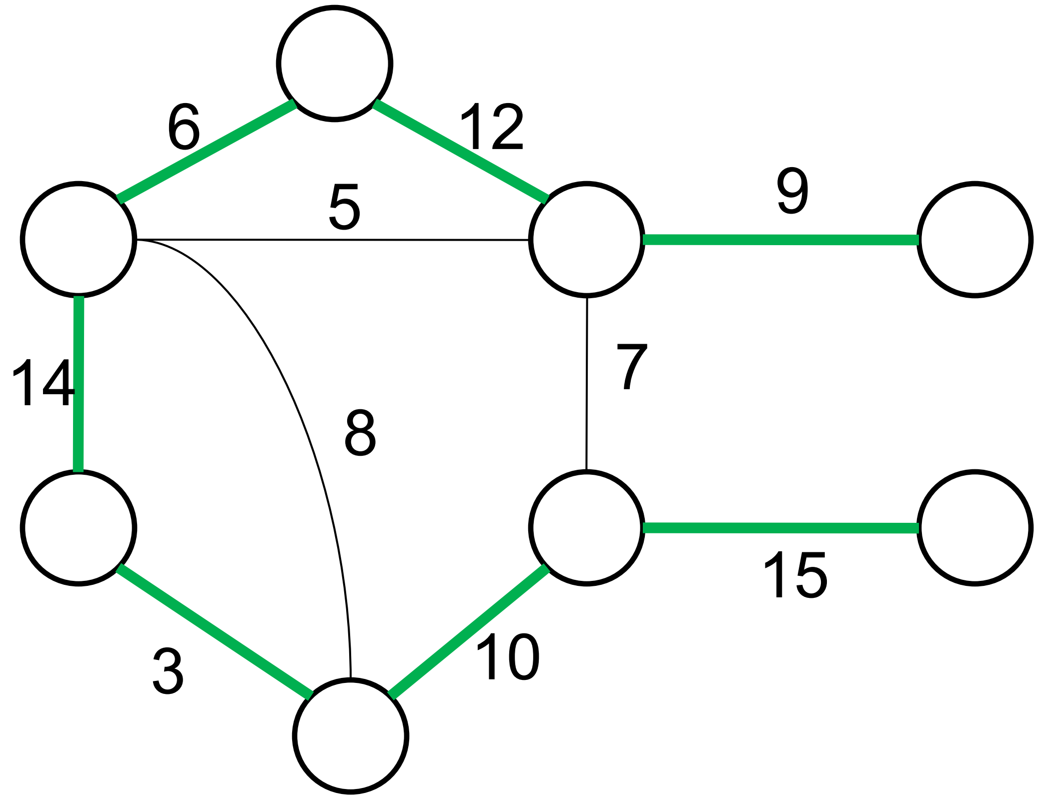
$$w(T) = \sum_{e \in T} w(e)$$



Example

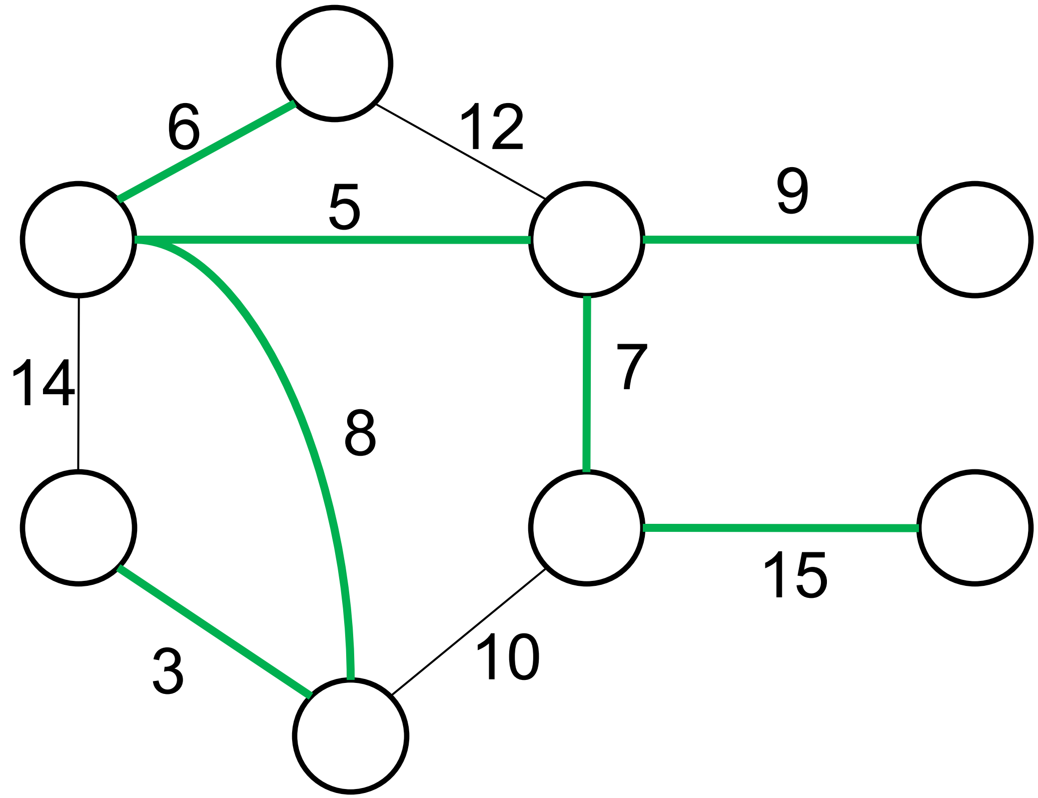


A Spanning Tree



But is this minimum?

A Better Spanning Tree

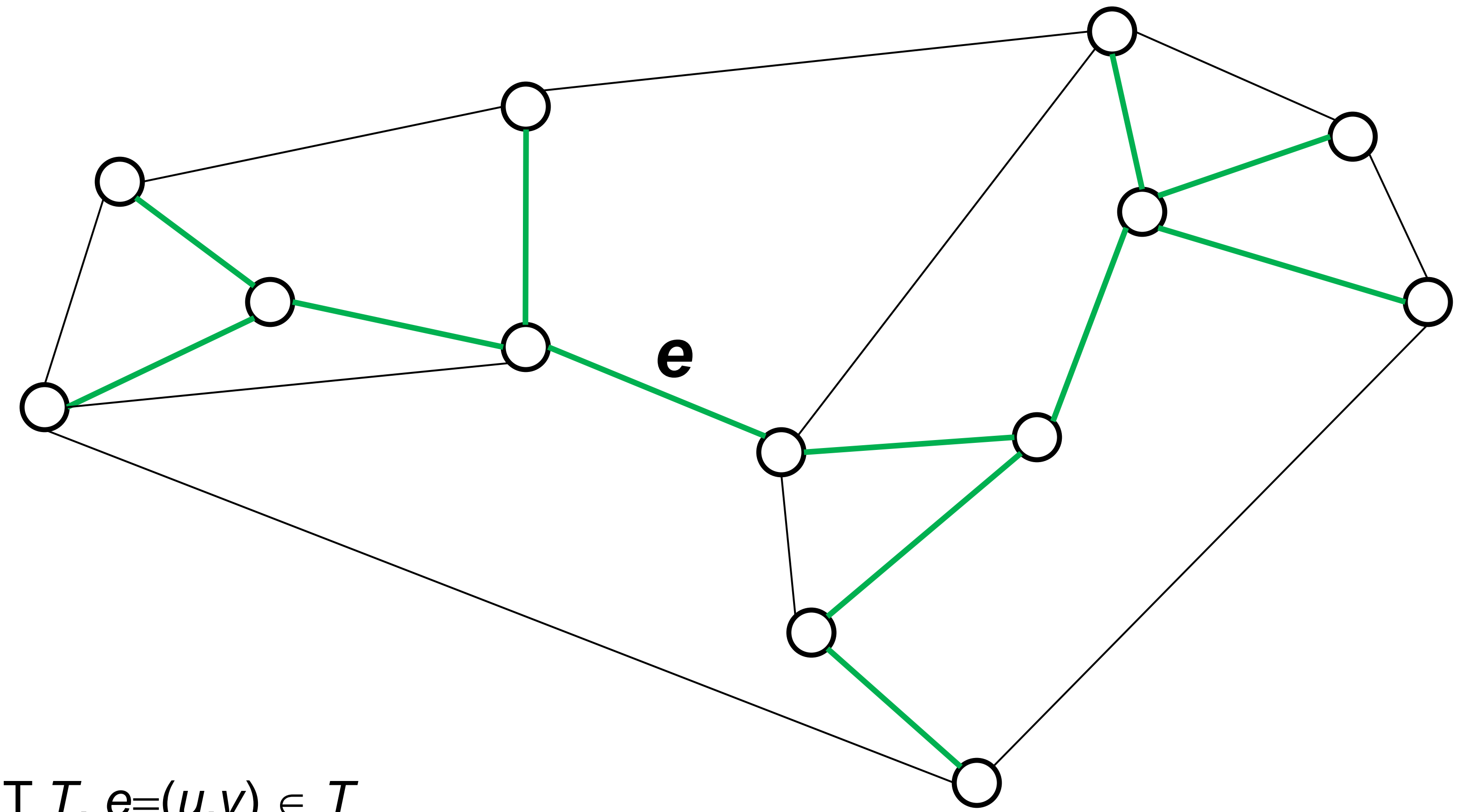


END

Minimal Spanning Trees

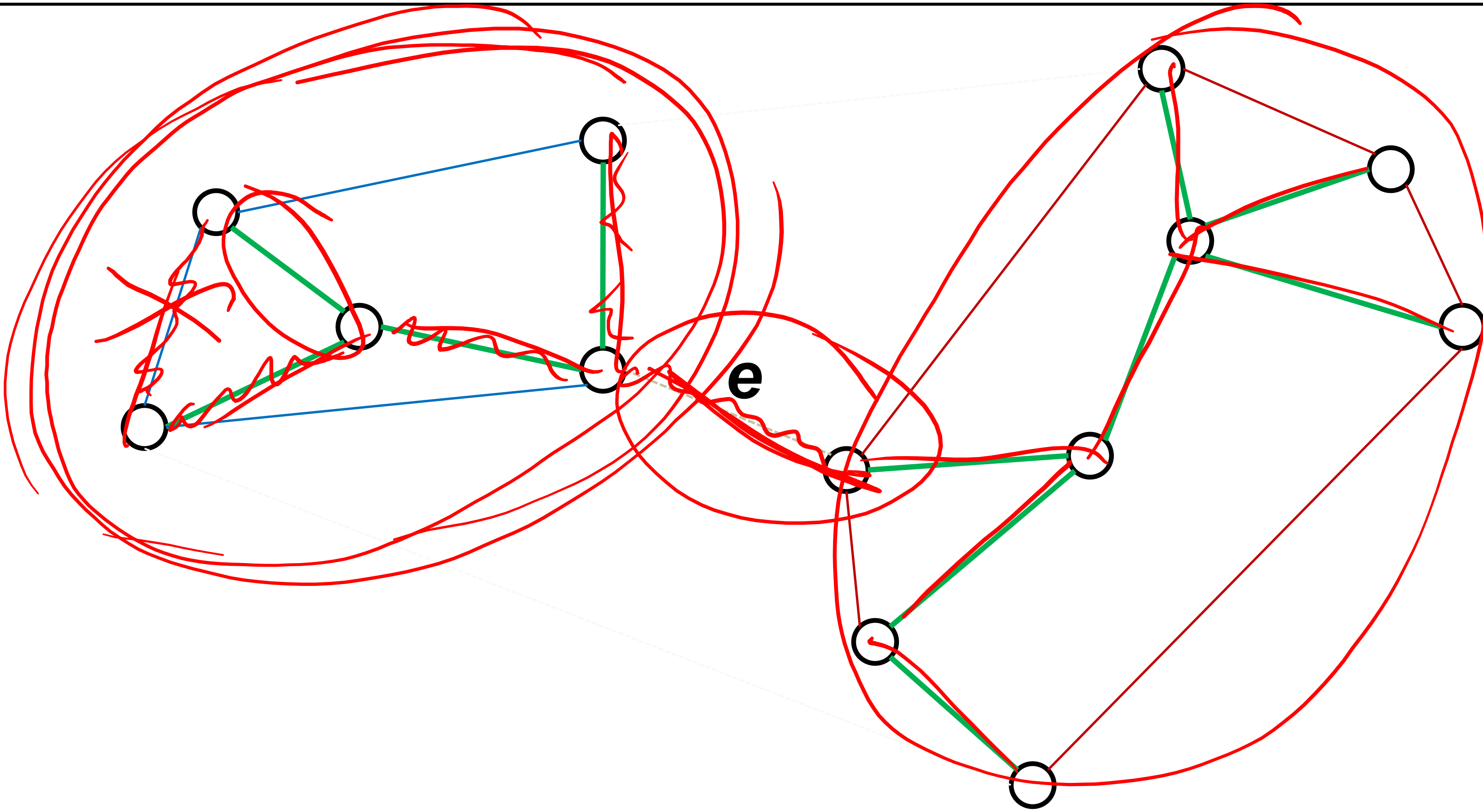
Properties

Optimal Substructure



- MST T, $e=(u,v) \in T$
- Remove e
- Get subtrees T_1 and T_2 with vertex sets V_1 and V_2

Optimal Substructure

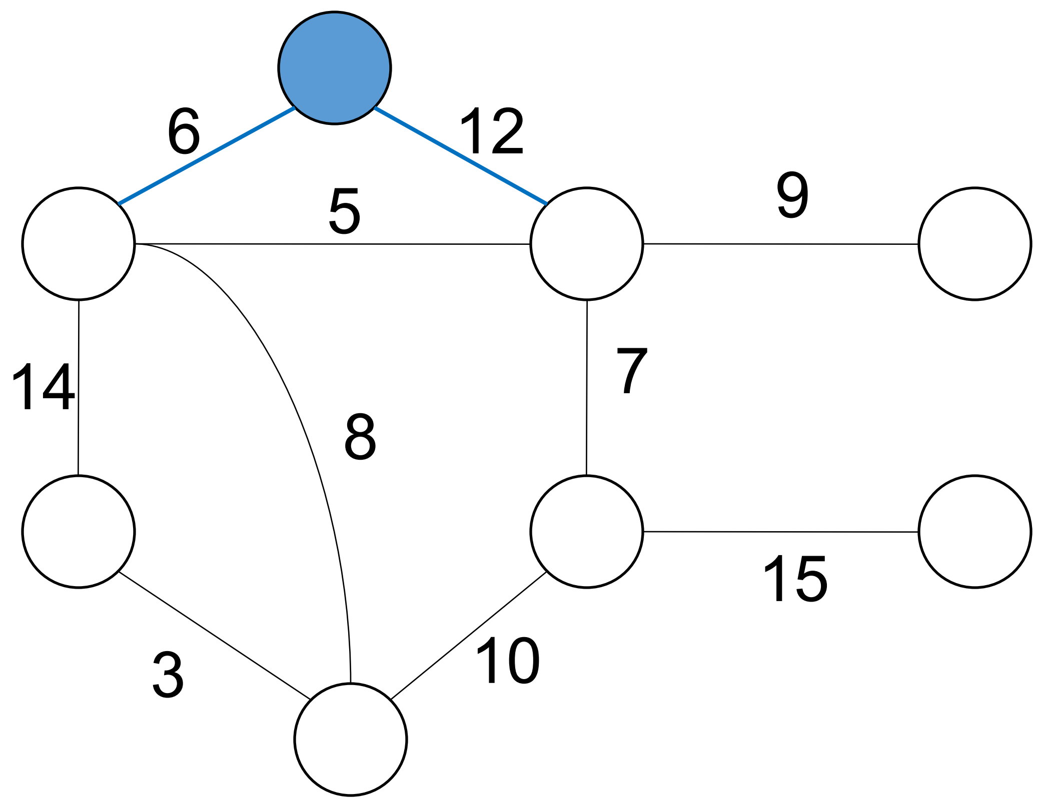


- V_i = vertices in T_i and $E_i = \{(u,v) \in E, u,v \in V_i\}$
- T_1 and T_2 are MSTs of $G(V_1,E_1)$ and $G(V_2,E_2)$, the subgraphs of G induced by V_1 and V_2

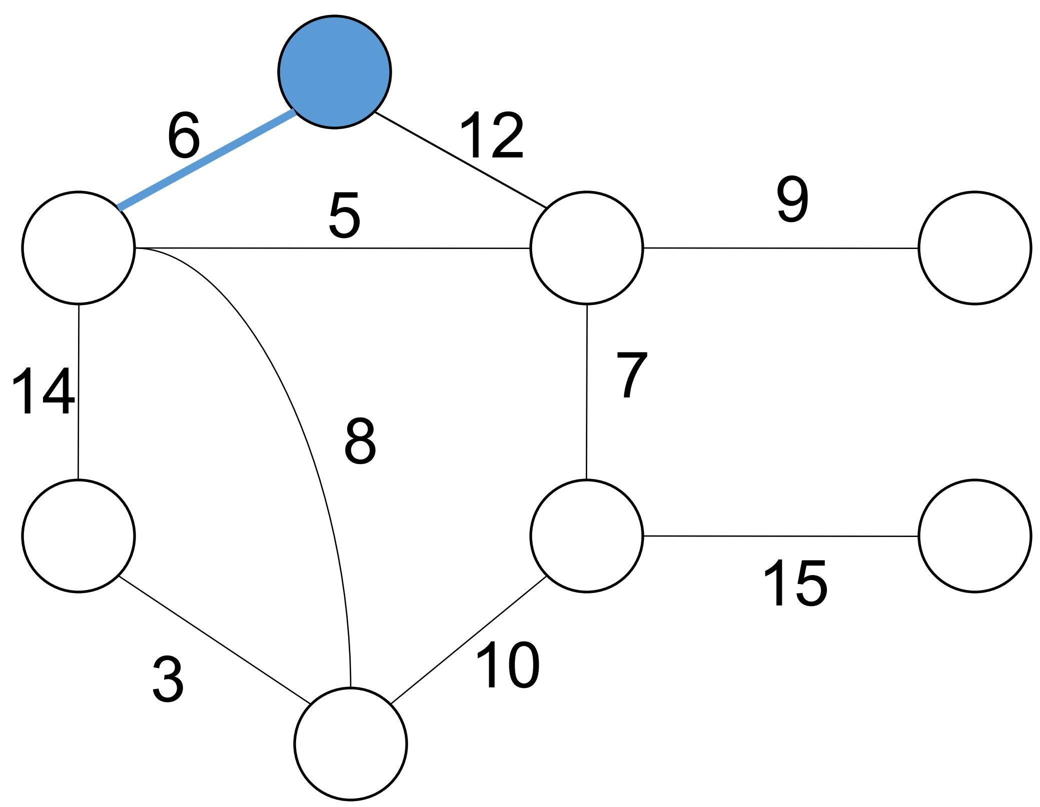
Hallmark of Greedy Algorithms

- *Greedy choice property:* A locally optimal choice leads to a globally optimal solution.
  - Identify a simple to implement heuristic to make the local choices.
  - Prove that the choices made are part of some optimal solution.
- ***Theorem (minimum cut):*** Let T be an MST of $G(V, E)$ and $A \subset V$. If $e = (u,v) \in E$ is the minimum weight edge connecting A to $V - A$, then $e \in T$.

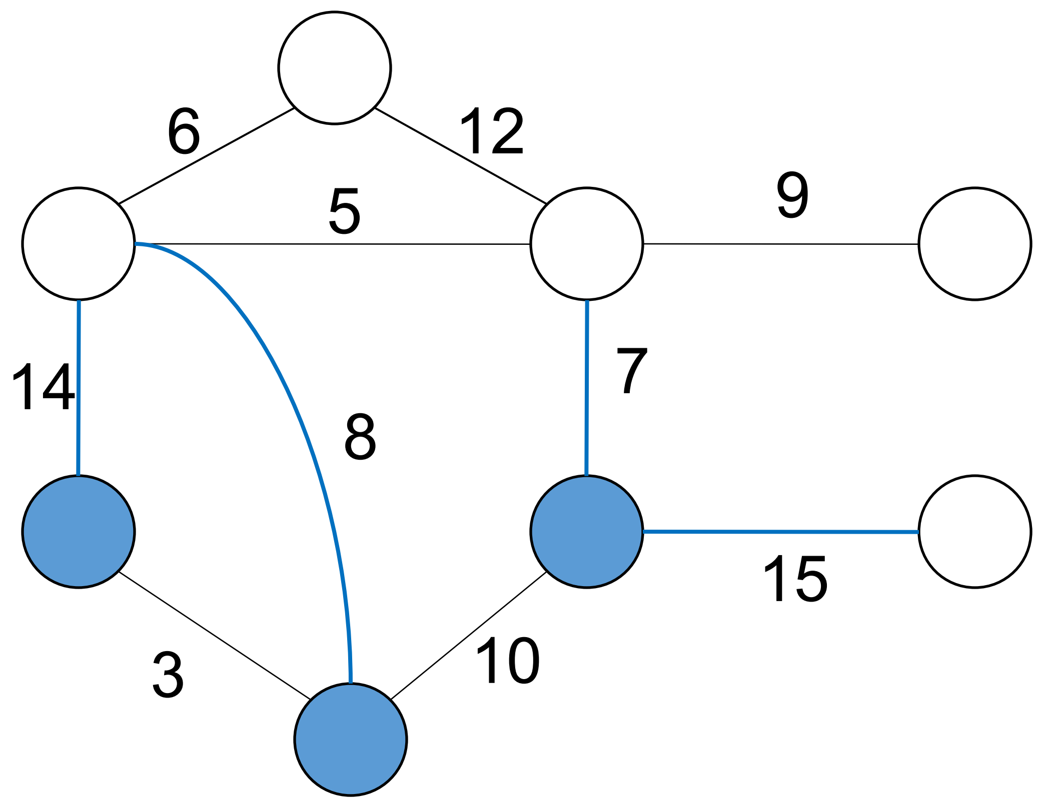
Example



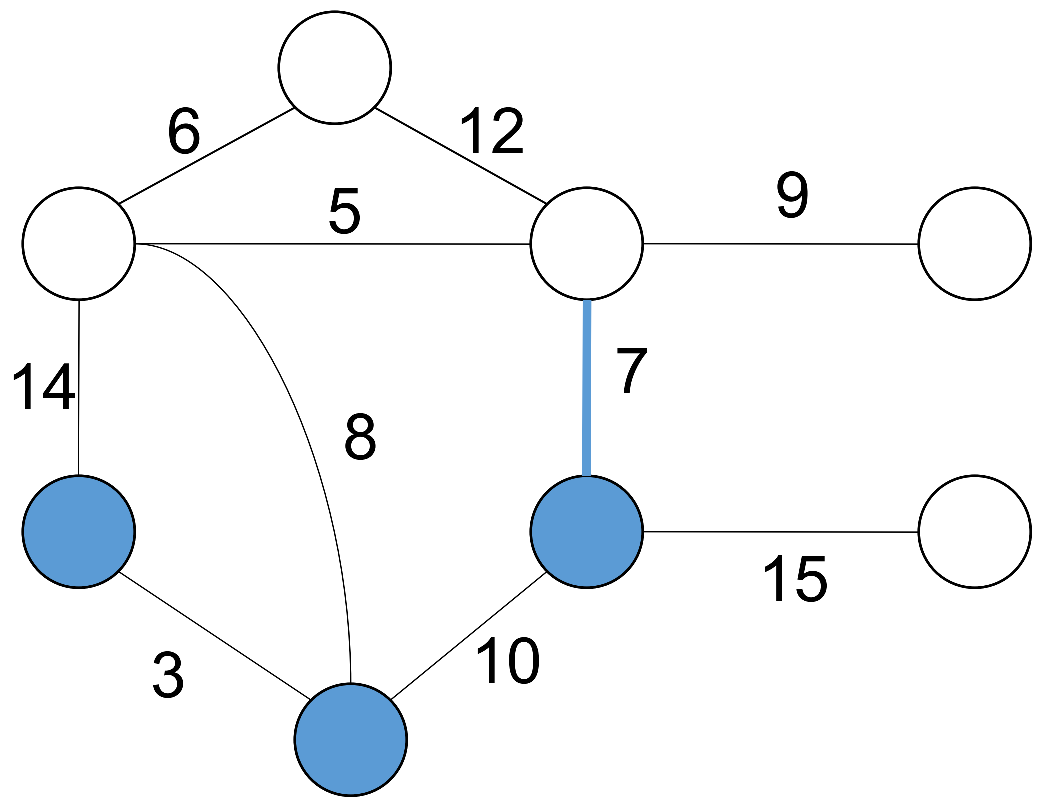
Example



Example



Example



END

Prim’s MST Algorithm

Prim's Algorithm

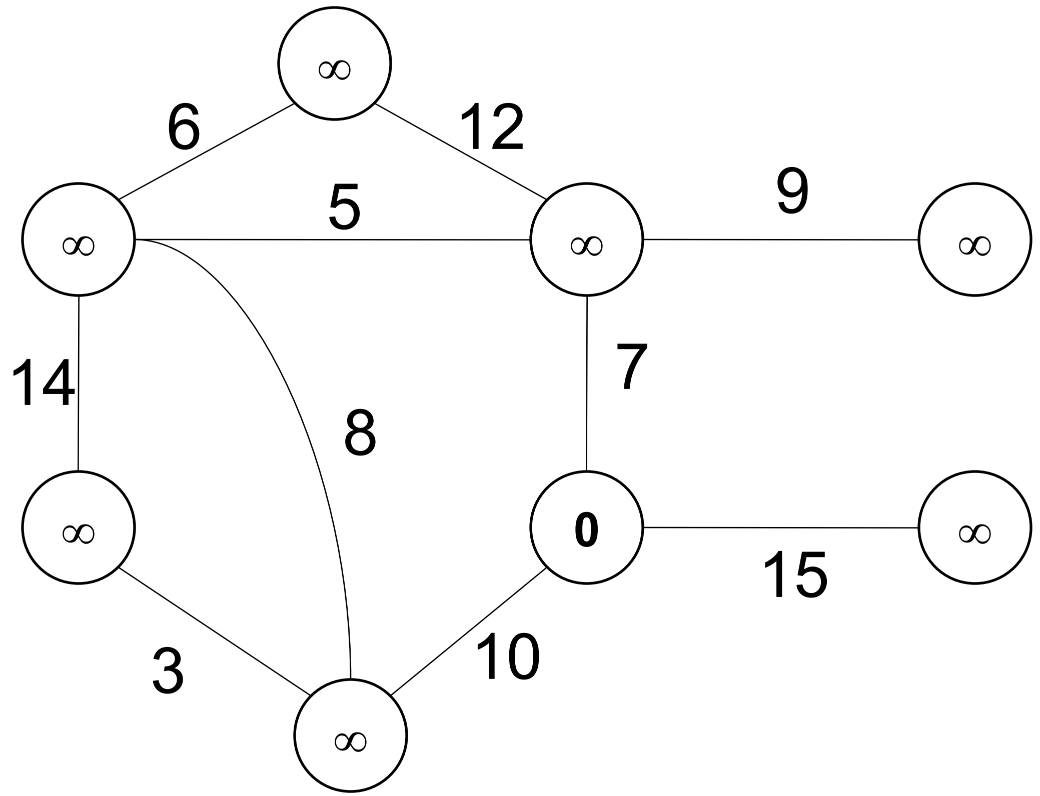
- Start with *A* containing a single arbitrary vertex and grow one vertex at a time. The cut is $(A, V - A)$.
- Keep a single growing tree of edges committed so far
- Keep $V - A$ in a priority queue *Q.*
  - Vertices in A have already been reached by the partial *T.*
- The priority of each vertex *q* in *Q* is the minimum cost required to connect *q* to a vertex in *A.*
- Repeatedly remove the minimum vertex *u* from *Q* (using ExtractMin) and add it to *A.*
- Update *Q* by (possibly) updating the priorities of *u*'s neighbors (using DecreaseKey).

Prim's Algorithm

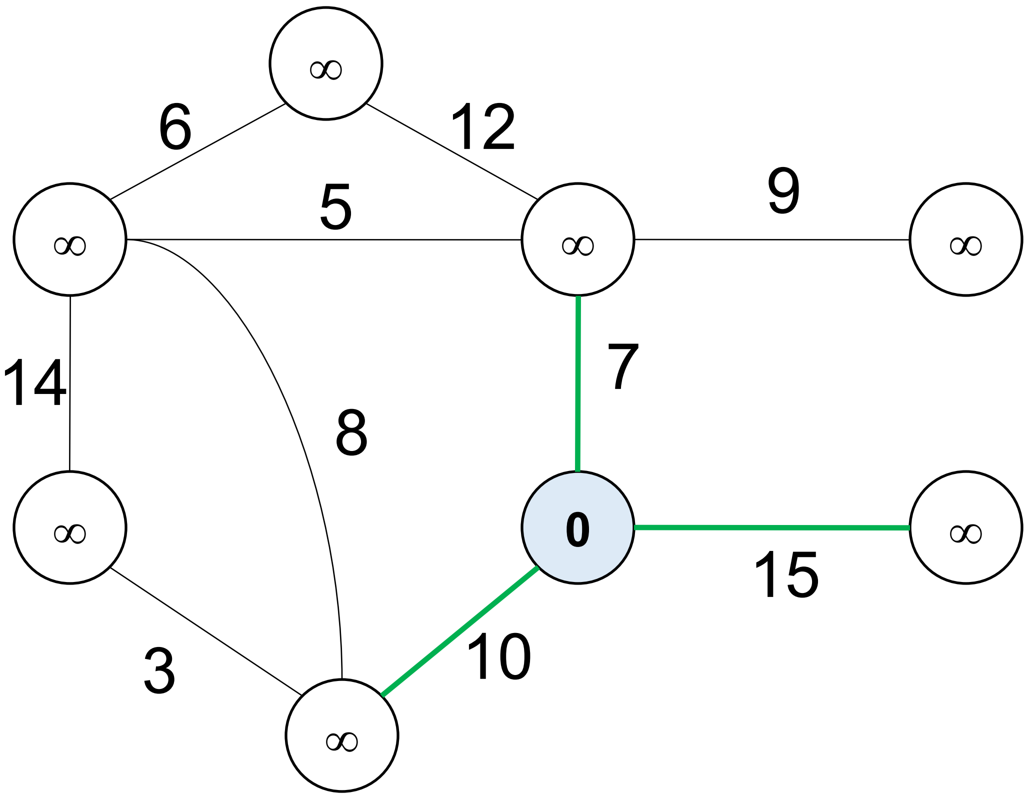
PRIM-MST(V, E, w, s)

foreach $v \in V$
    do $key[v] \leftarrow \infty$
        $P[v] \leftarrow \text{NIL}$
$key[s] \leftarrow 0$
$Q \leftarrow V$
while $Q \neq \emptyset$
    do $u \leftarrow \textsc{Extract-Min}(Q)$
        foreach $v \in Adj[u]$
            do if $v \in Q$ and $w(u, v) < key[v]$
                then $key[v] \leftarrow w(u, v)$
                    $P[v] \leftarrow u$
return P

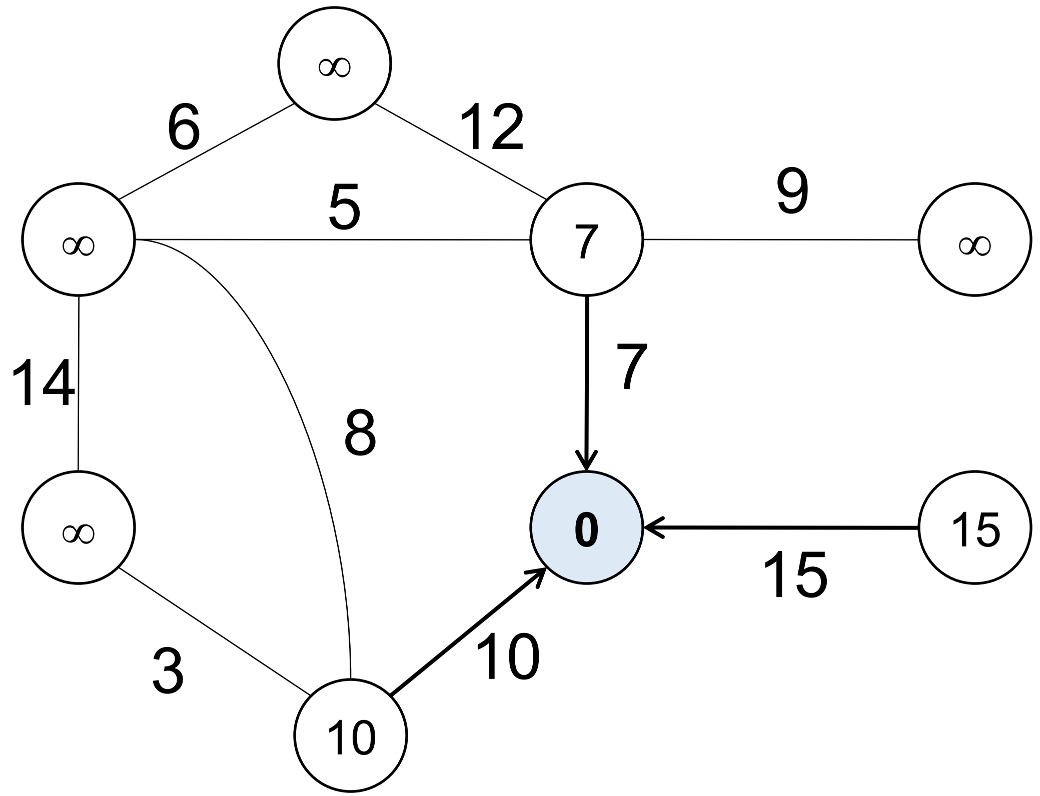
Example



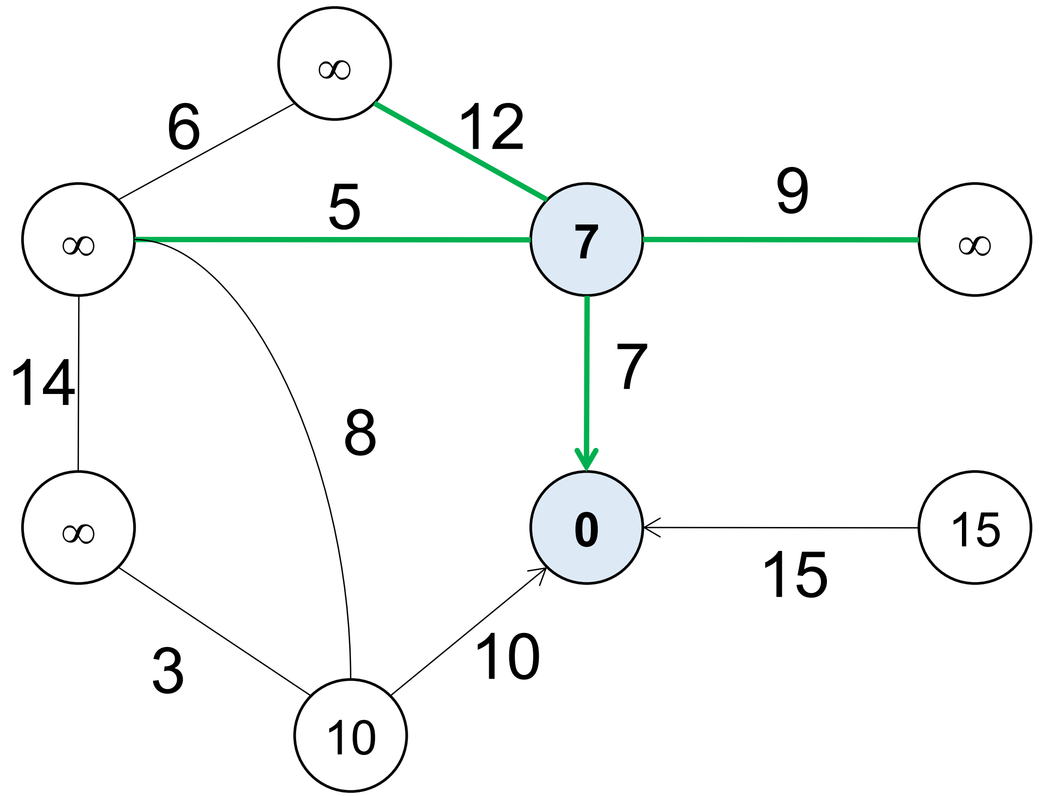
Example



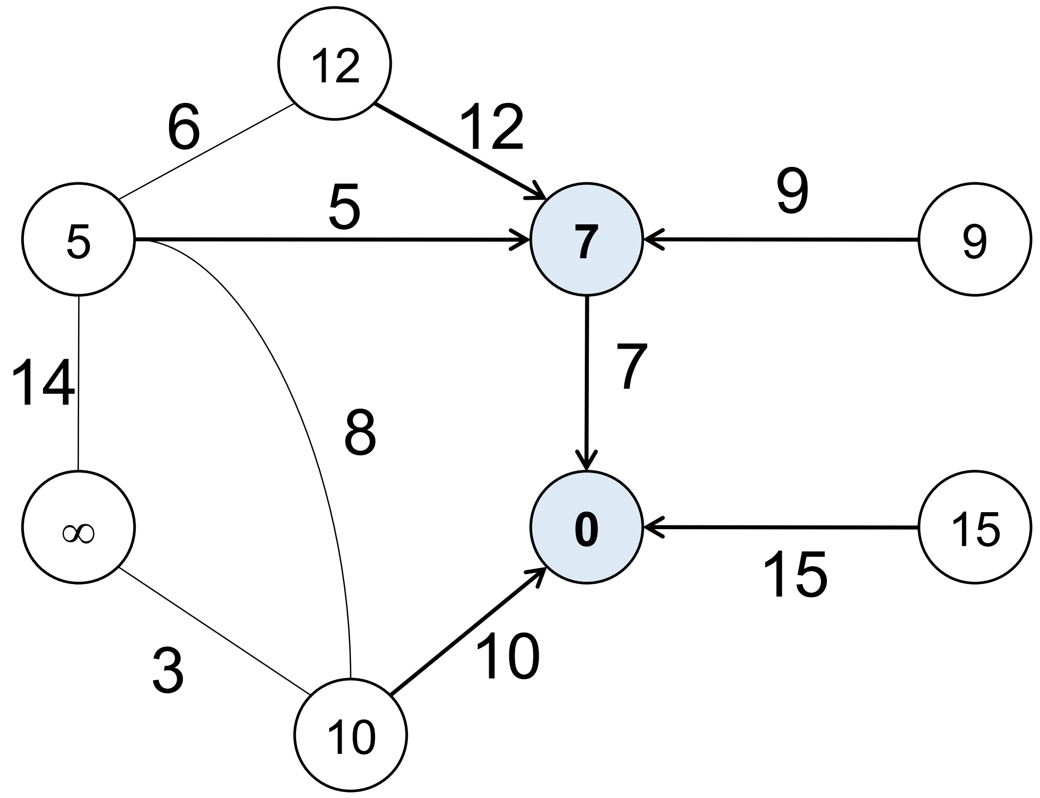
Example



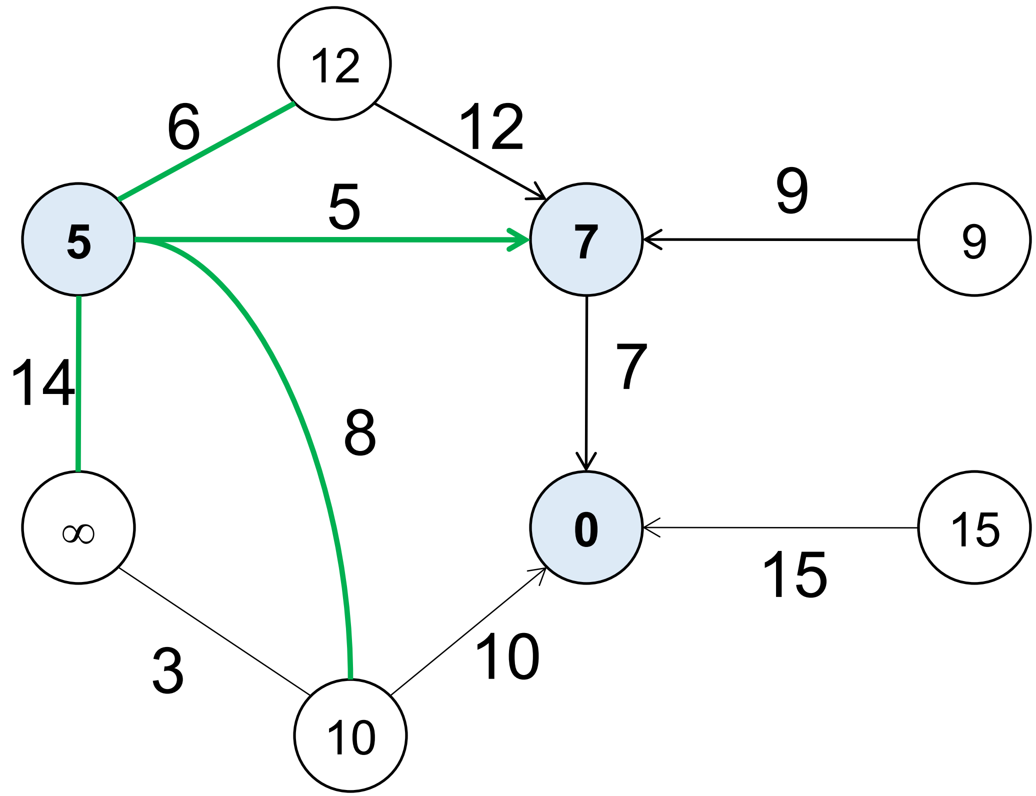
Example



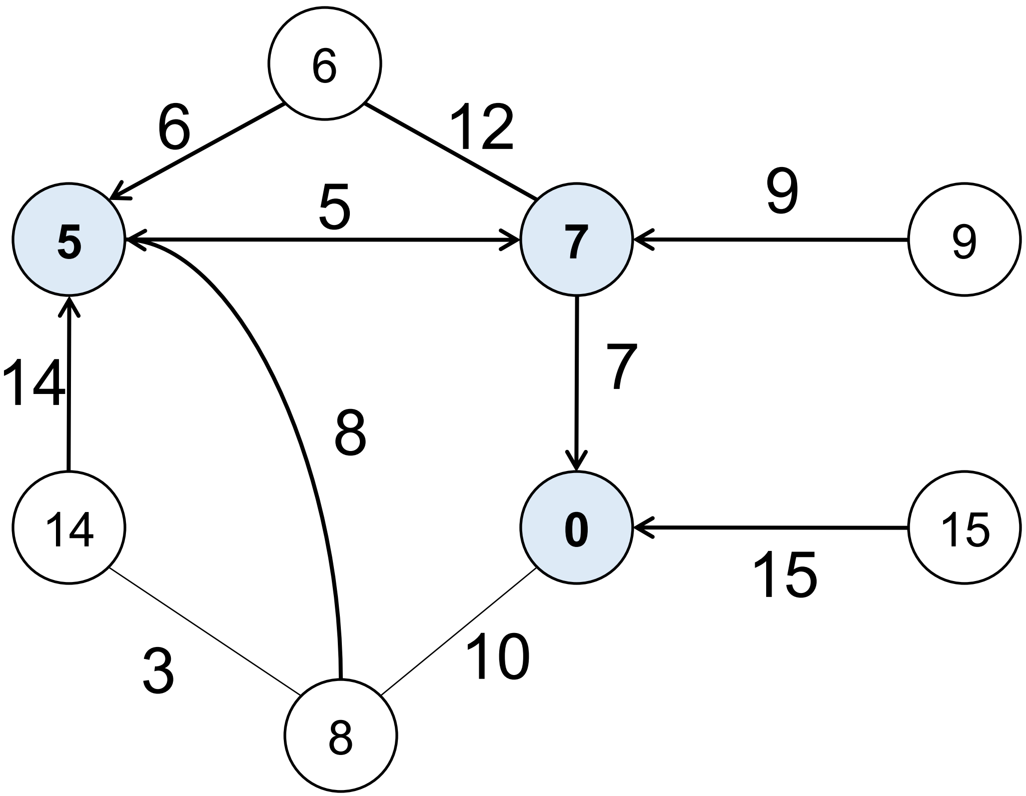
Example



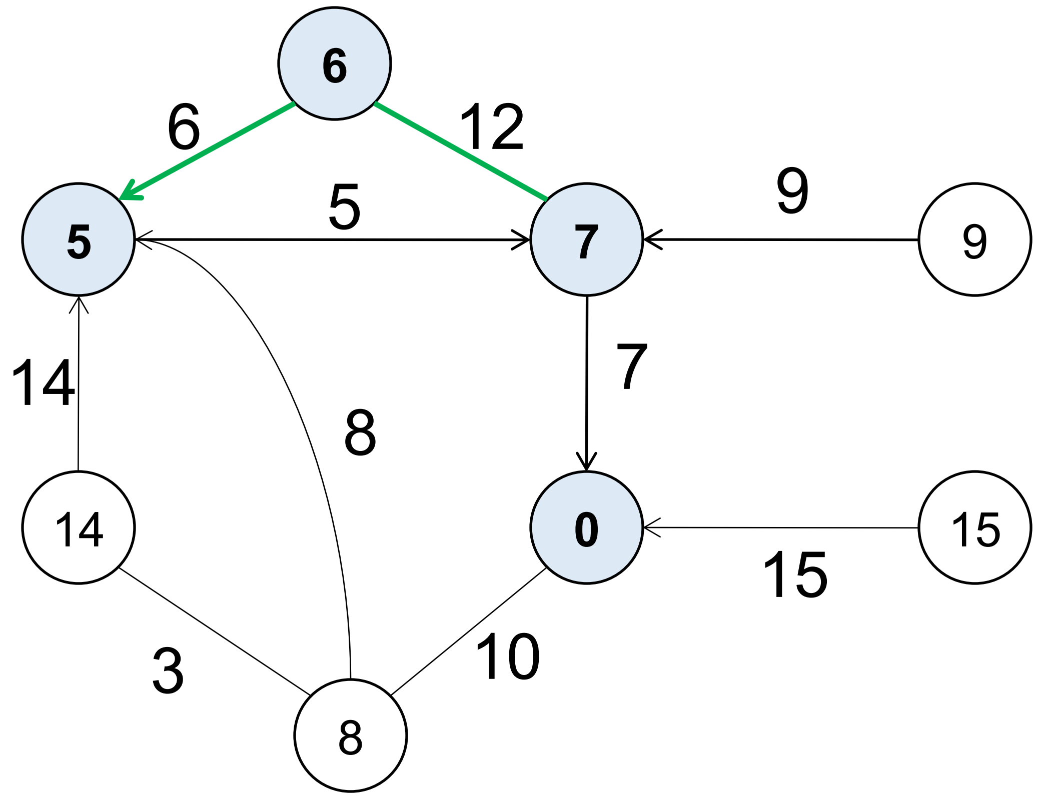
Example



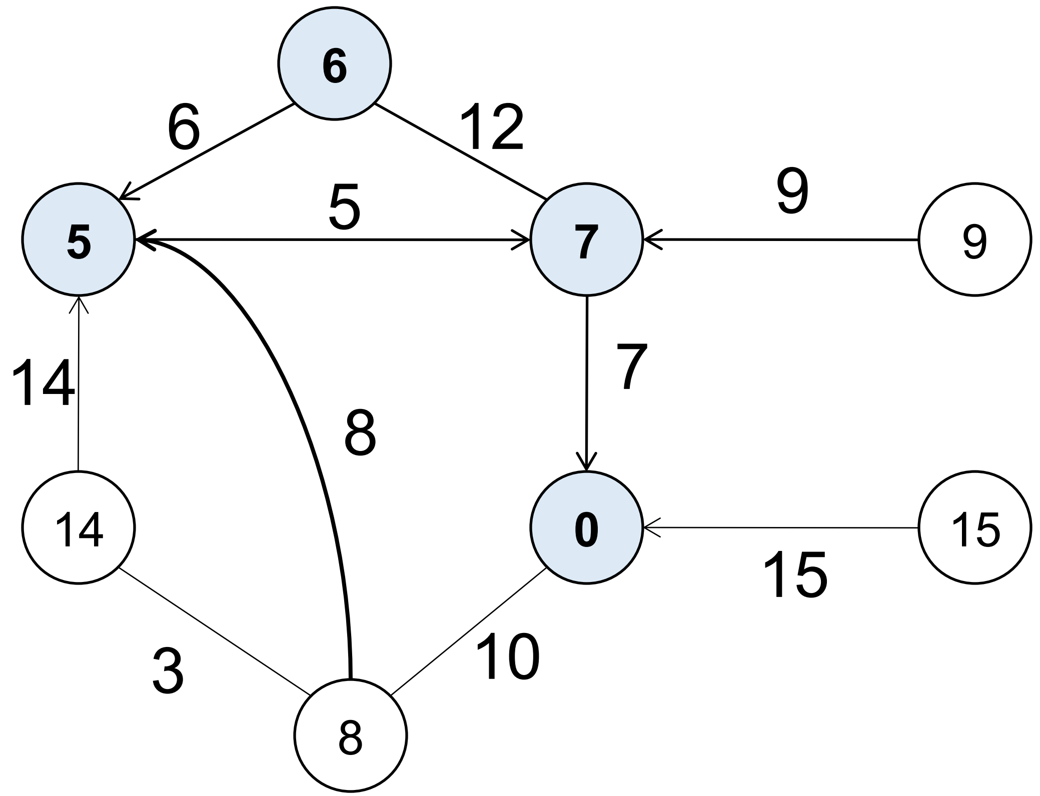
Example



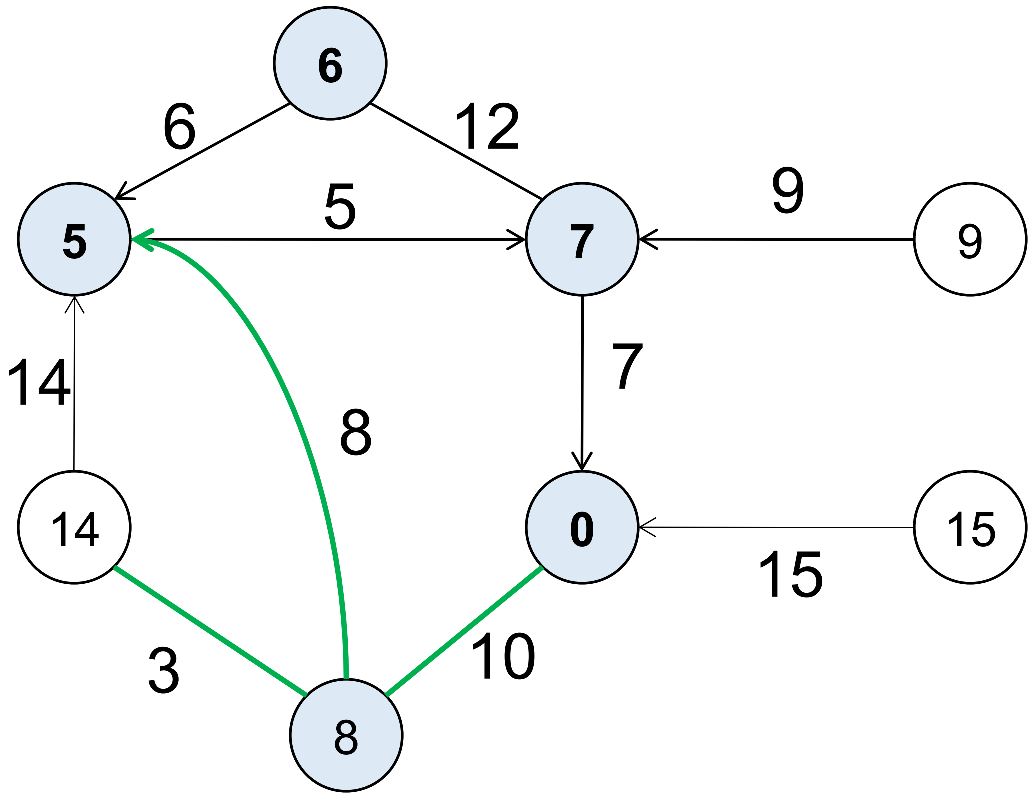
Example



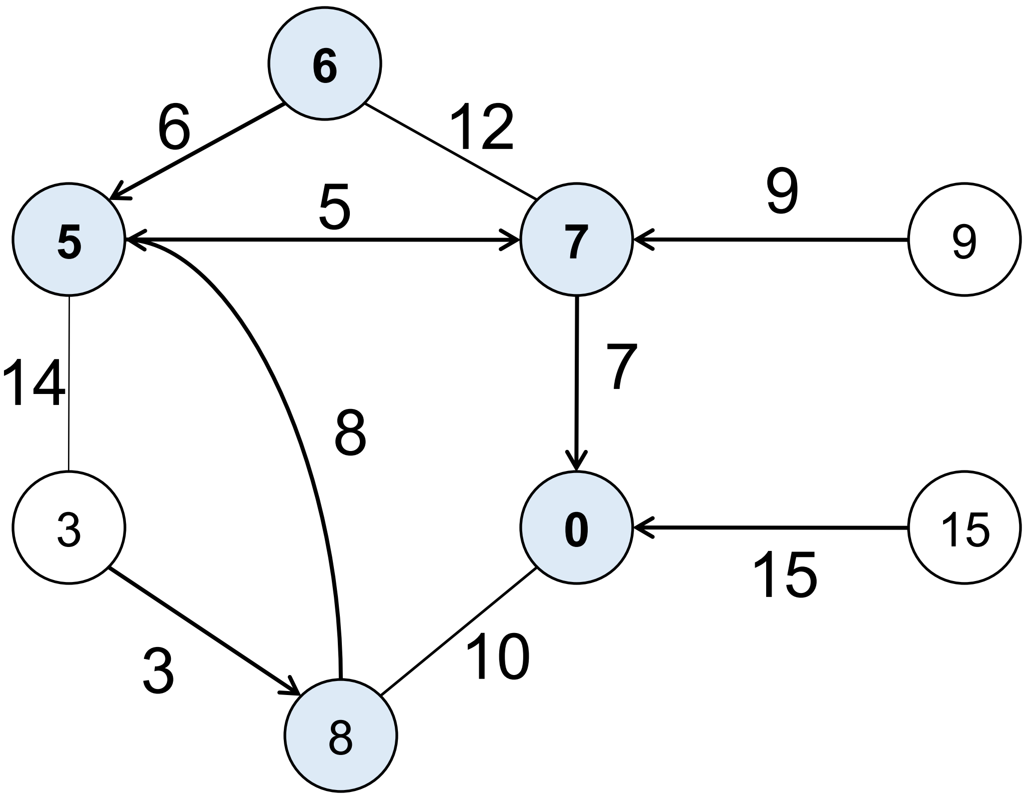
Example



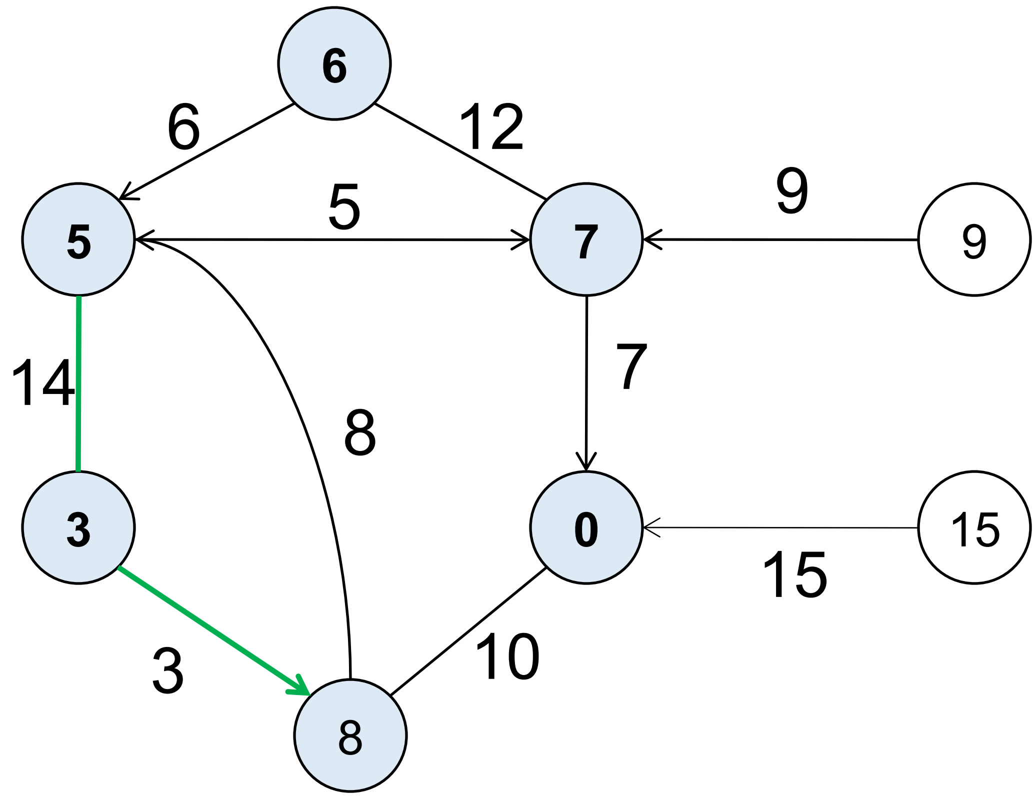
Example



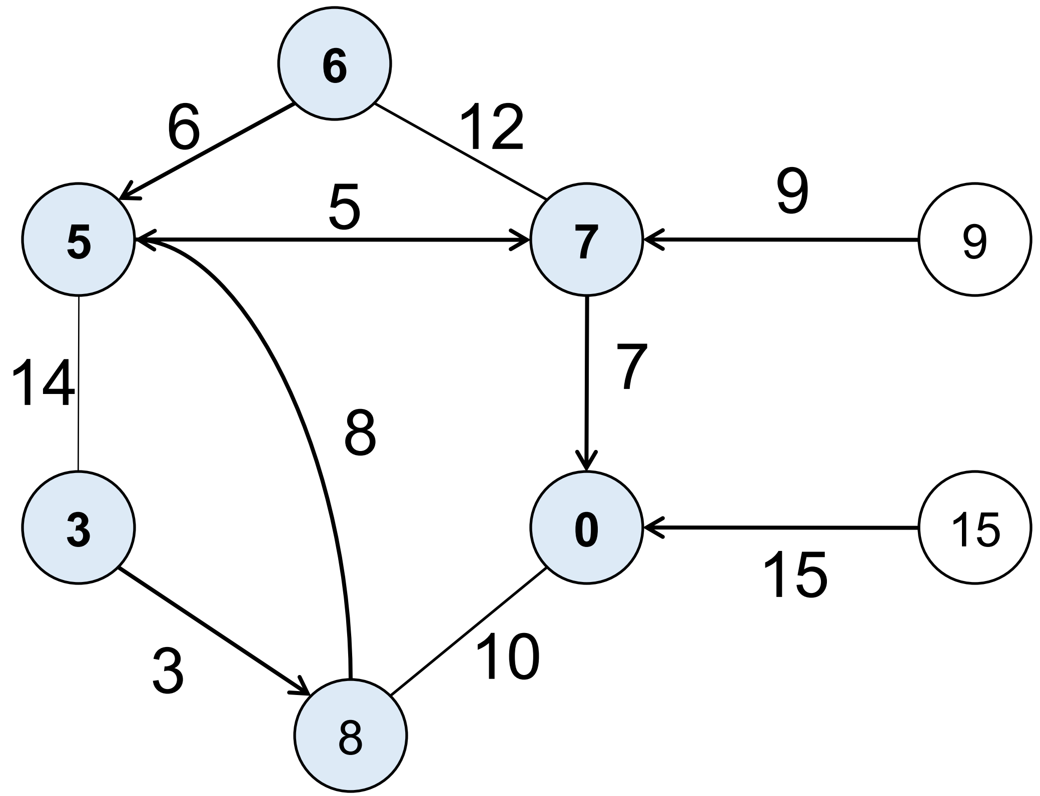
Example



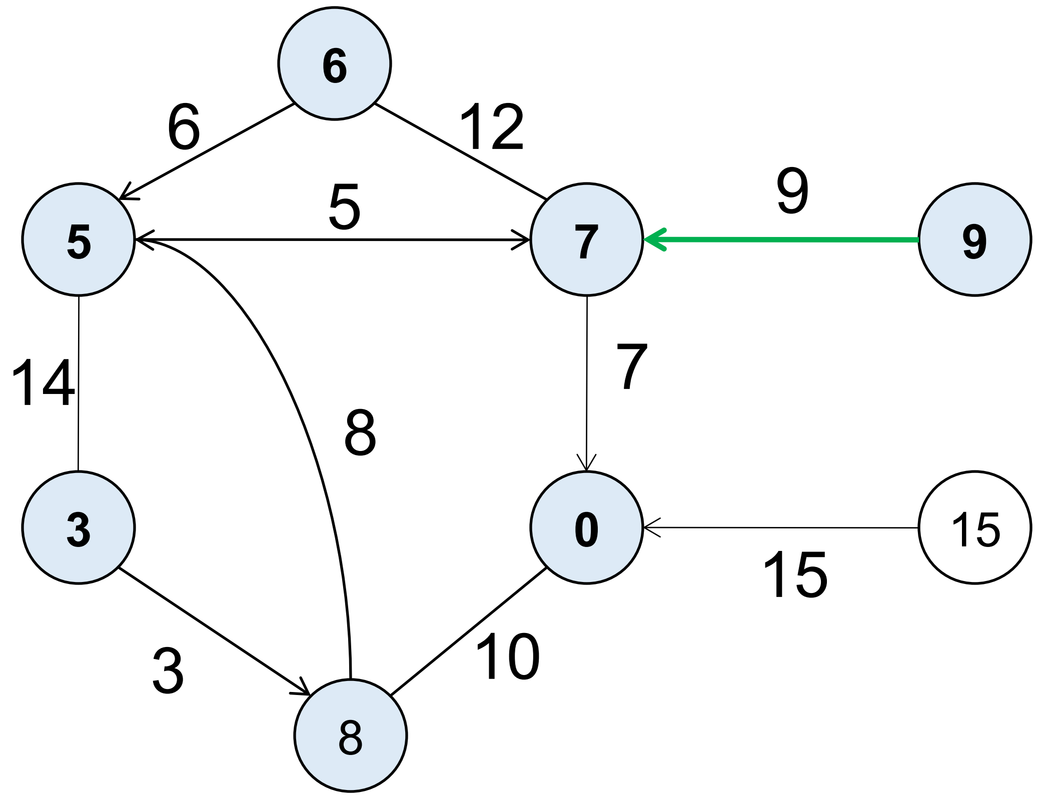
Example



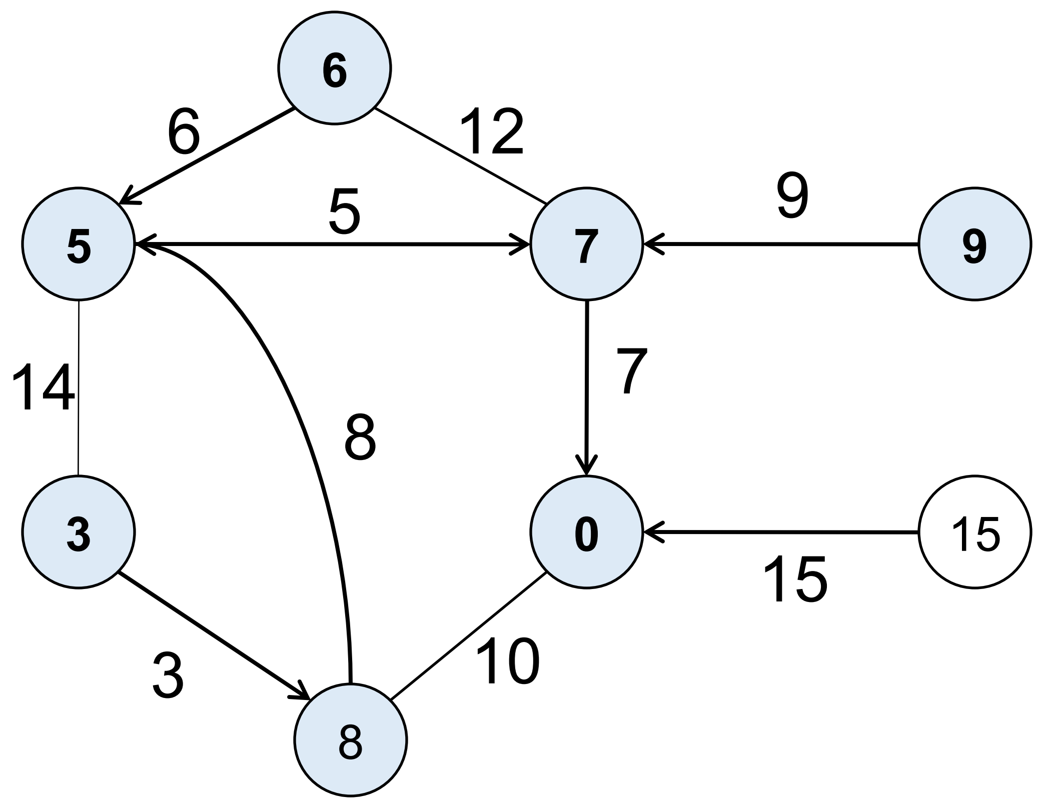
Example



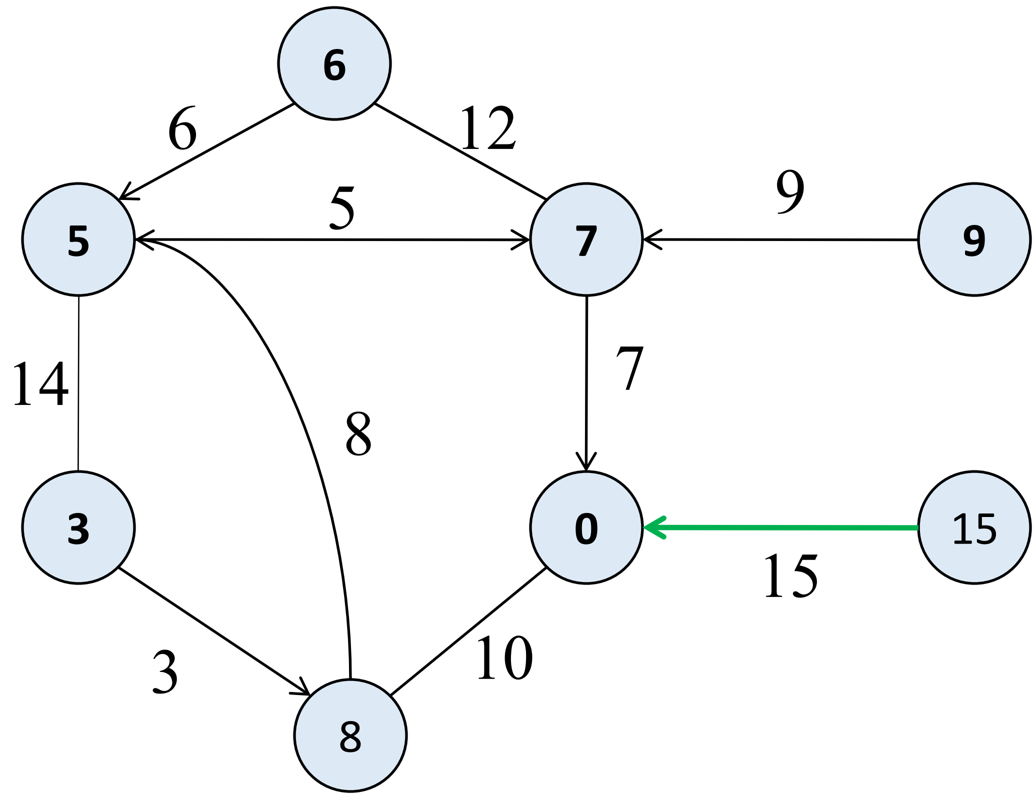
Example



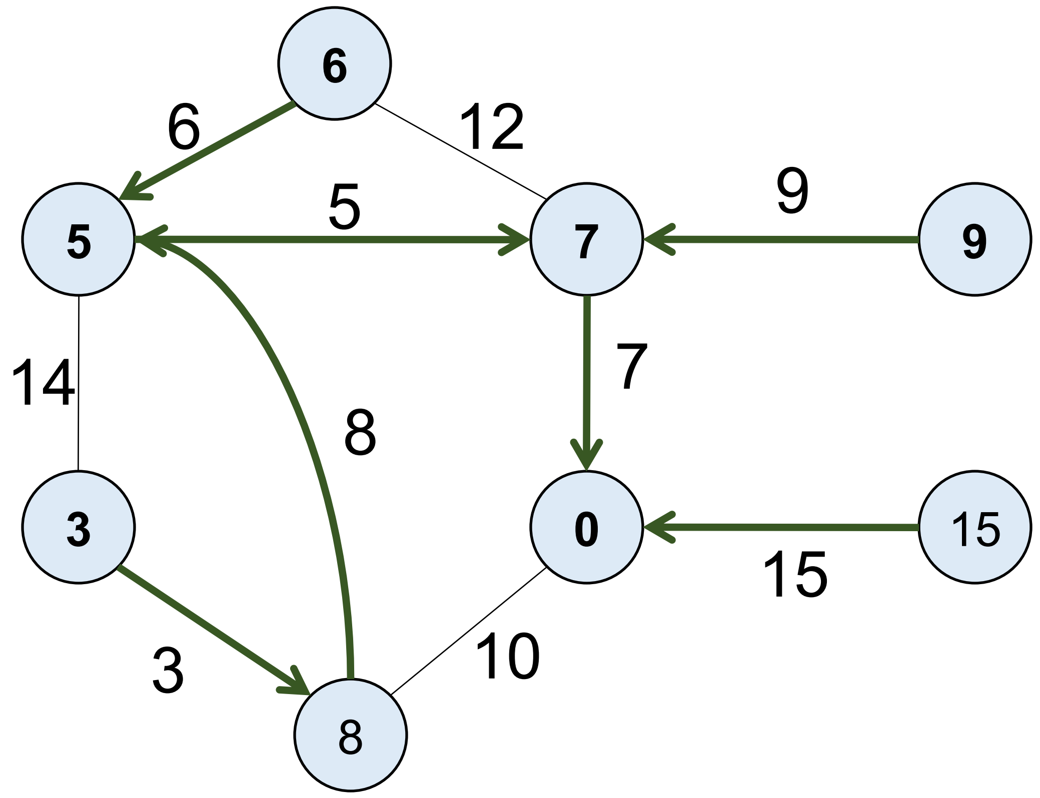
Example



Example



Example



Analysis

Frequency count

- 1–3 n times
- 4–5: once
- 6–7: n times
- 8–11: 2m times
- 12: once

PRIM-MST(V, E, w, s)

```
1   foreach v ∈ V
2       do key[v] ← ∞
3          P[v] ← NIL
4   key[s] ← 0
5   Q ← V
6   while Q ≠ ∅
7       do u ← EXTRACT-MIN(Q)
8          foreach v ∈ Adj[u]
9              do if v ∈ Q and w(u, v) < key[v]
10                    then key[v] ← w(u, v)
11                         P[v] ← u
12  return P
```

$$T(n,m) = \mathrm{T}_{\mathrm{build}} + n\mathrm{T}_{\mathrm{extract}} + m\mathrm{T}_{\mathrm{decrease}}$$

Analysis

Actual time depends on implementation of *Q*.

$$T(n,m) = \mathrm{T}_{\text{build}} + n\mathrm{T}_{\text{extract}} + m\mathrm{T}_{\text{decrease}}$$

Q	Build	Extract	DecreaseKey	Total
Python list	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n^2)$
Heap	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta((n+m)\log n)$
Fibonacci heap	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(m + n\log n)$

- Known results
  - Best deterministic: $O(m\,\alpha(m,n))$ (Chazelle, 2000)
  - Best randomized: $O(n+m)$ expected (Karger et al., 1995)
  - Holy grail: $O(n+m)$ worst case, open

END

Acknowledgements

- Introduction to Algorithms, 3rd edition, by T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein; MIT Press, 2009