CS4311
Design and Analysis of Algorithms

Lecture 26: Minimum Spanning Tree

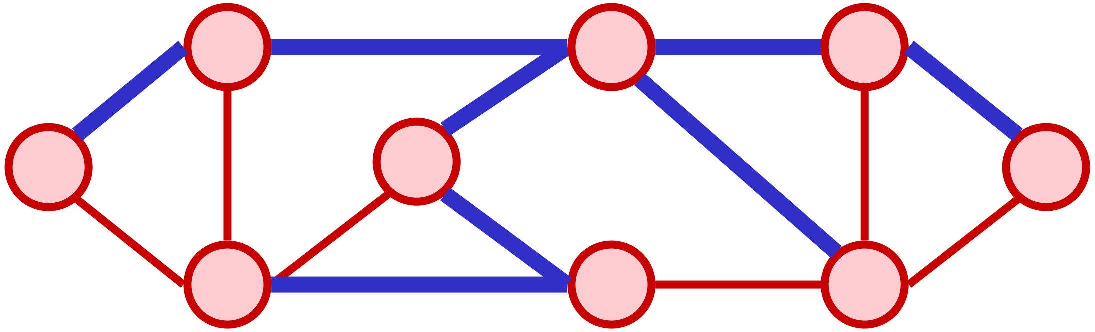
About this lecture

- What is a Minimum Spanning Tree?
- Some History

- The Greedy Choice Lemma
    - Kruskal's Algorithm
    - Prim's Algorithm
    - Borůvka's Algorithm

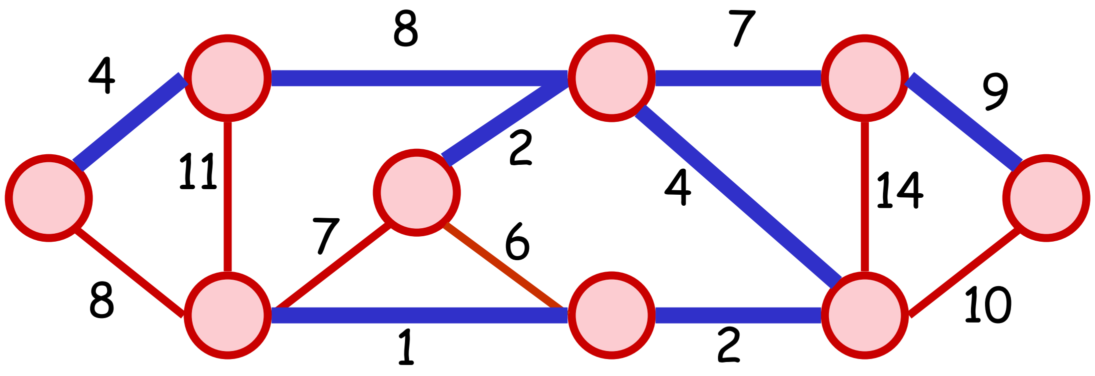
Minimum Spanning Tree

- Let $G = (V,E)$ be an undirected, connected graph
- A spanning tree of G is a tree, using only edges in E, that connects all vertices of G



Minimum Spanning Tree

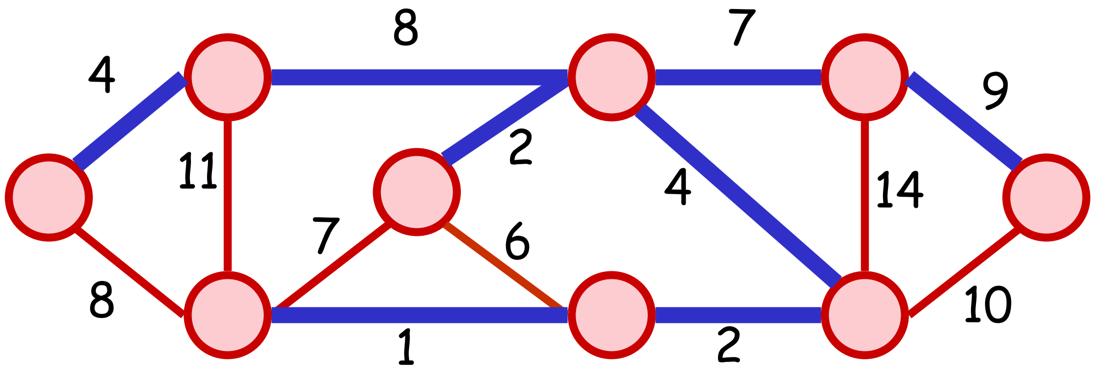
- Sometimes, the edges in G have weights
  - weight ⇔ cost of using the edge
- A minimum spanning tree (MST) of a weighted G is a spanning tree such that the sum of edge weights is minimized

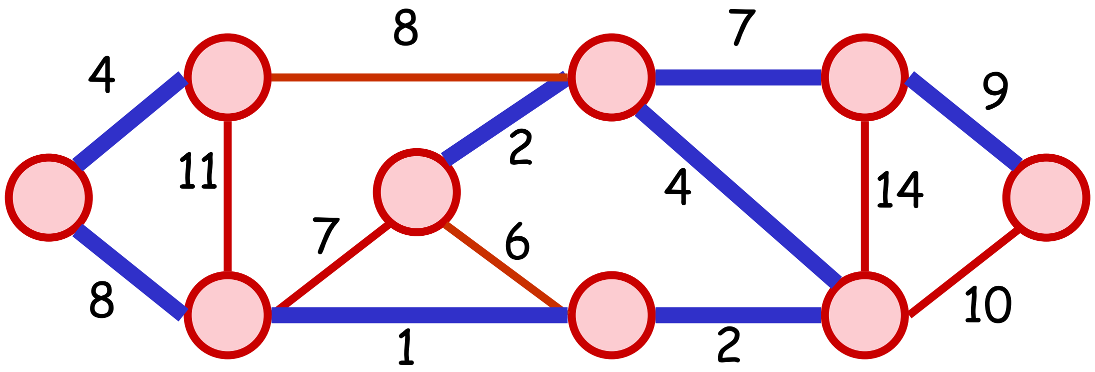


Total cost = 4 + 8 + 7 + 9 + 2 + 4 + 1 + 2 = 37

Minimum Spanning Tree

- MST of a graph may **not** be unique





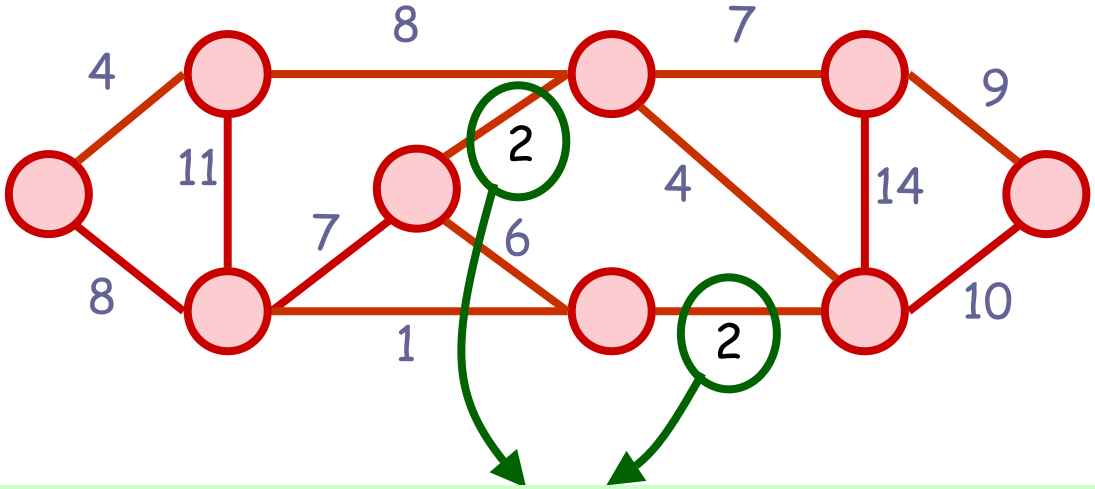
Some History

- Borůvka [1926] : First algorithm
  - for electrical coverage of Moravia
- Kruskal [1956] : Kruskal's algorithm
- Jarník [1930], Prim [1957] : Prim's algorithm
- Fredman-Tarjan [1987] : $O(E \log^*(V))$ time
- Gabow et al [1986]: $O(E \log \log^*(V)$ time
- Chazelle [1999]: $O(E \alpha(E,V))$ time

Remark: $\log^*$ = iterated log, $\alpha(m,n)$ = inverse Ackermann

Greedy Choice Lemma

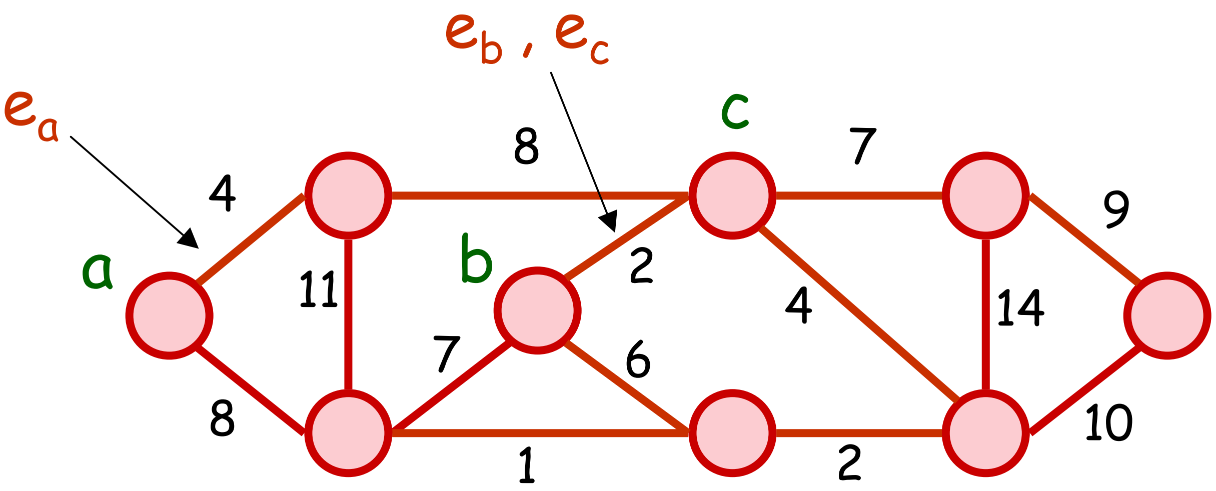
- Suppose all edge weights are distinct
  - If not, we give an arbitrary ordering among equal-weight edges
- E.g.,



Give an arbitrary ordering among these two edges, so that one costs "fewer" than the other

Greedy Choice Lemma

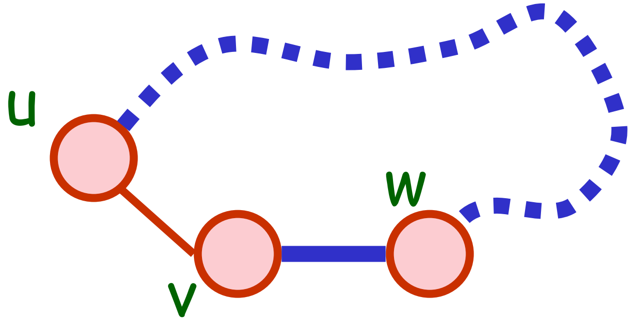
- Let e_v to be the cheapest edge adjacent to v, for each vertex v



Theorem: The minimum spanning tree of G contains every e_v

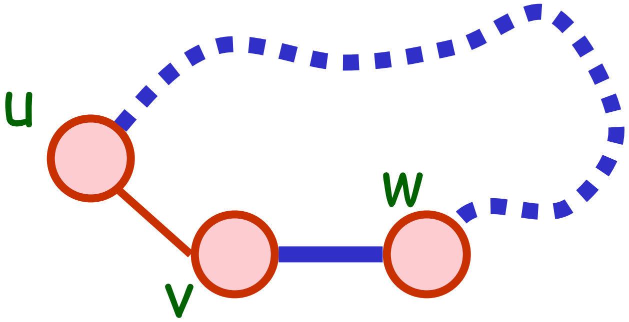
Proof

- Recall that all edge weights are distinct
- Suppose on the contrary that MST of G does not contain some edge e_v = (u,v)
- Let T = optimal MST of G
- By adding e_v = (u,v) to T, we obtain a cycle u, v, w, …, u [why??]

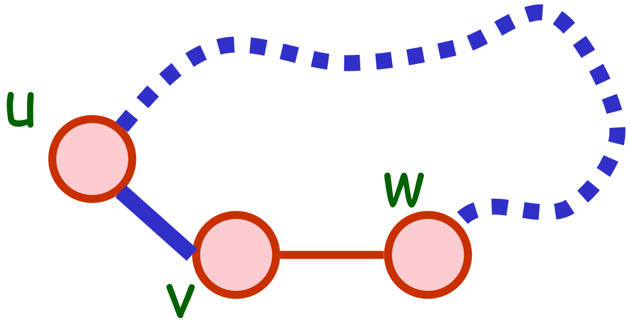


Proof

- By our choice of e_v , we must have weight of (u,v) cheaper than weight of (v,w) to T



- If we delete (v,w) and include e_v, we obtain a spanning tree cheaper than T



➔ contradiction !!

Optimal Substructure

Let E' = a set of edges which are known to be in an MST of $G = (V,E)$

Let G^* = the graph obtained by contracting each component of $G' = (V,E')$ into a single vertex

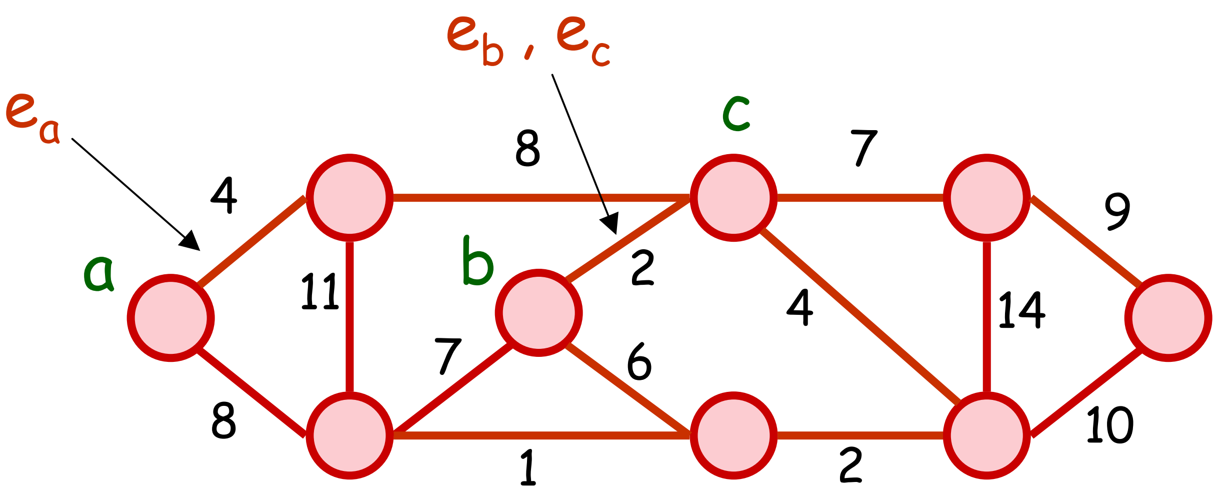
Let T^* be (the edges of) an MST of G^*

Theorem: $T^* \cup E'$ is an MST of G

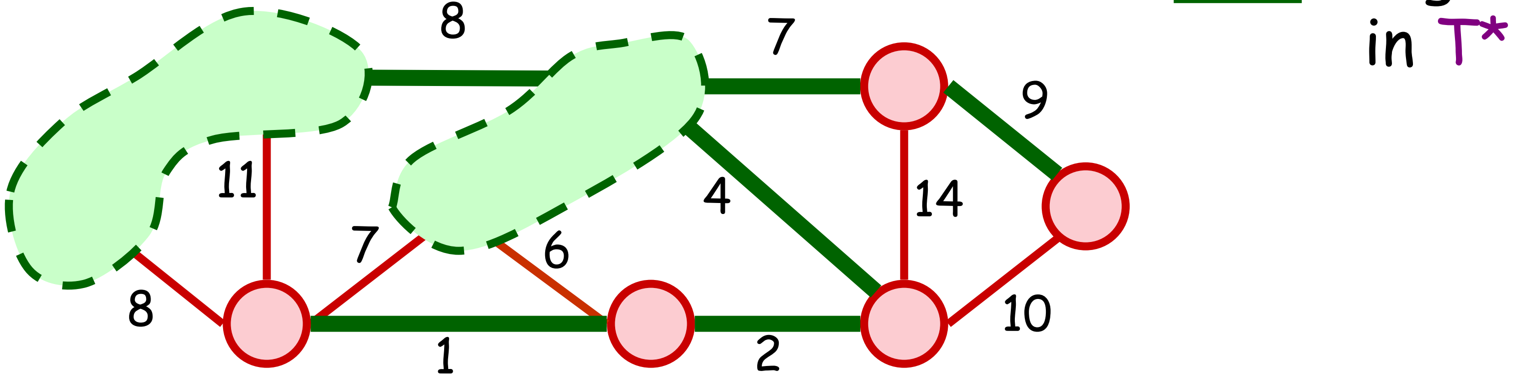
Proof: By contradiction

Example

G



G*

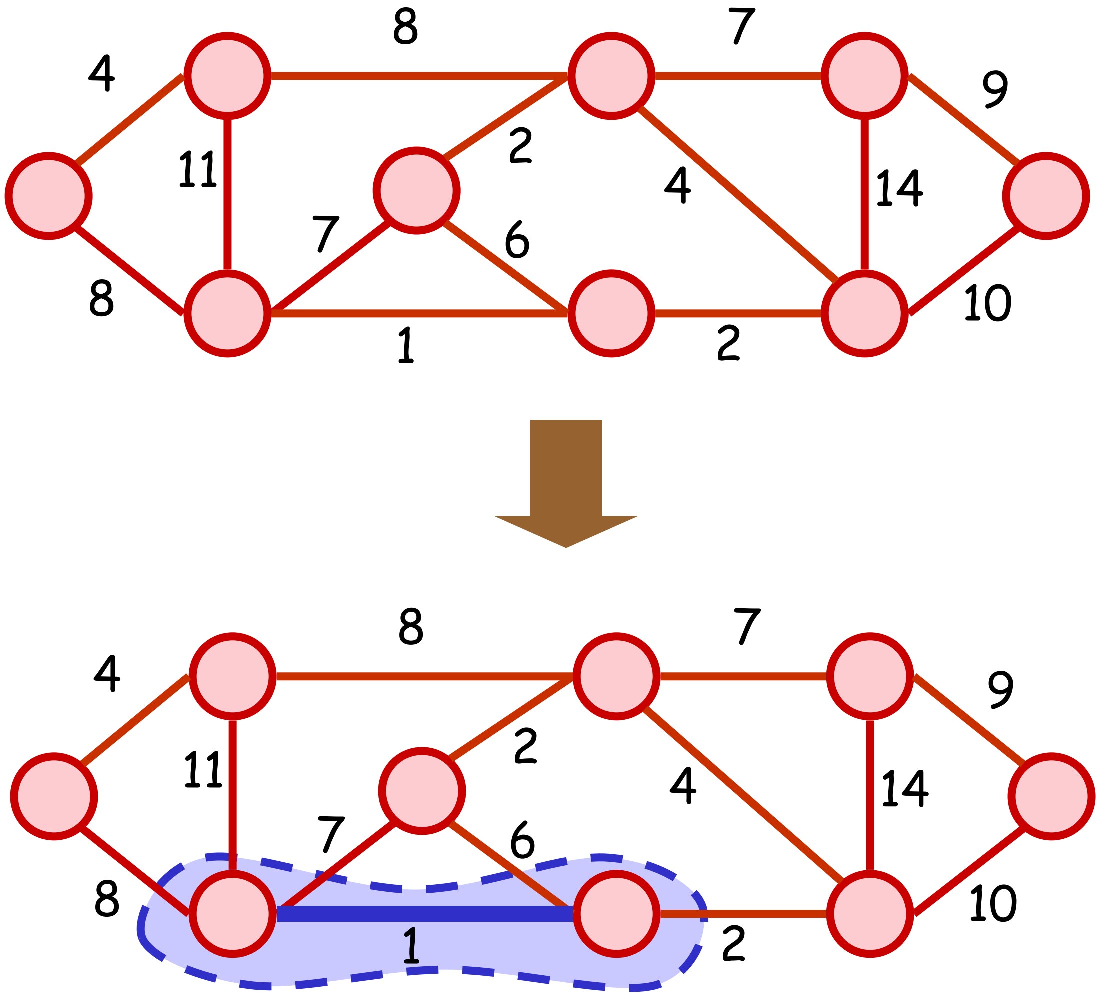


Kruskal's Algorithm

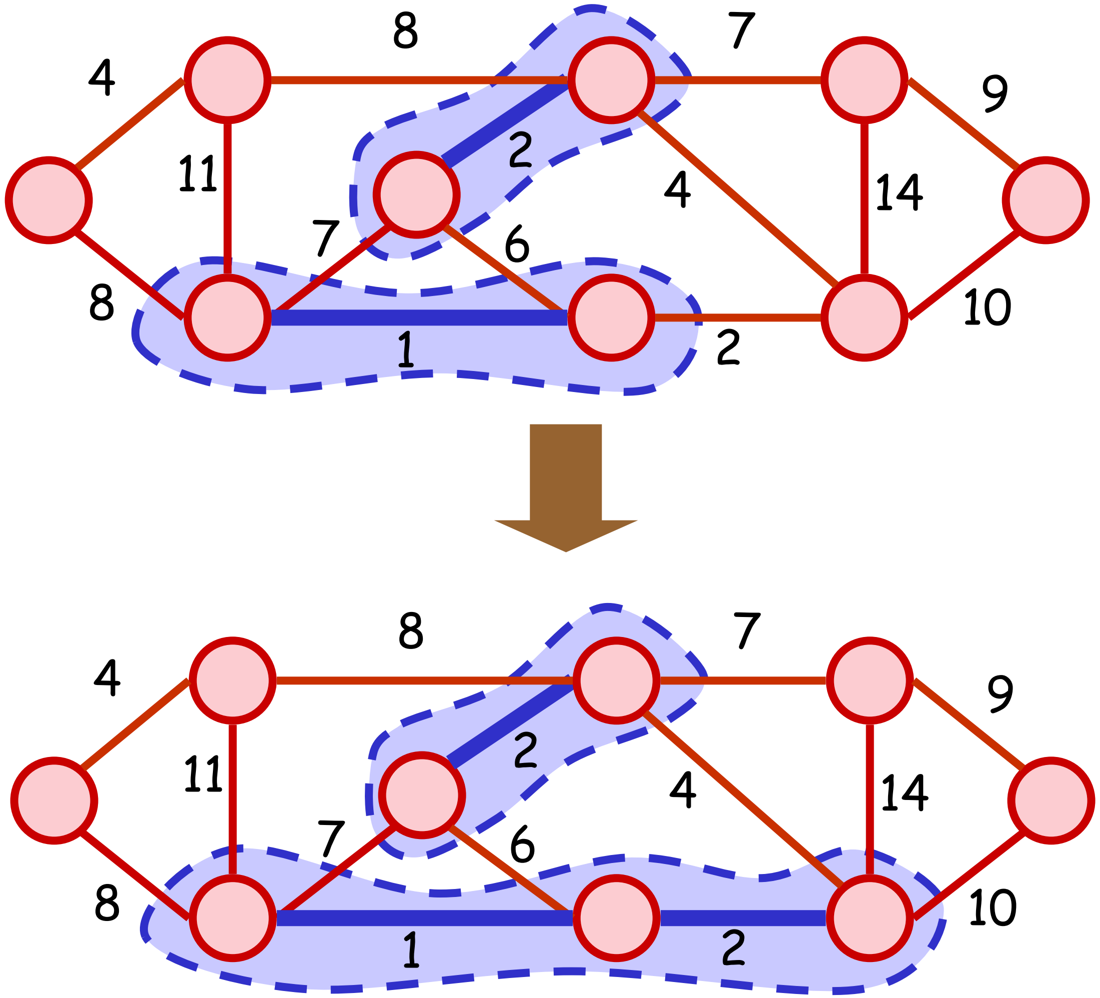
Kruskal-MST(G)

- Find the cheapest (non-self-loop) edge (u,v) in G
- Contract (u,v) to obtain G*
- Kruskal-MST(G*)

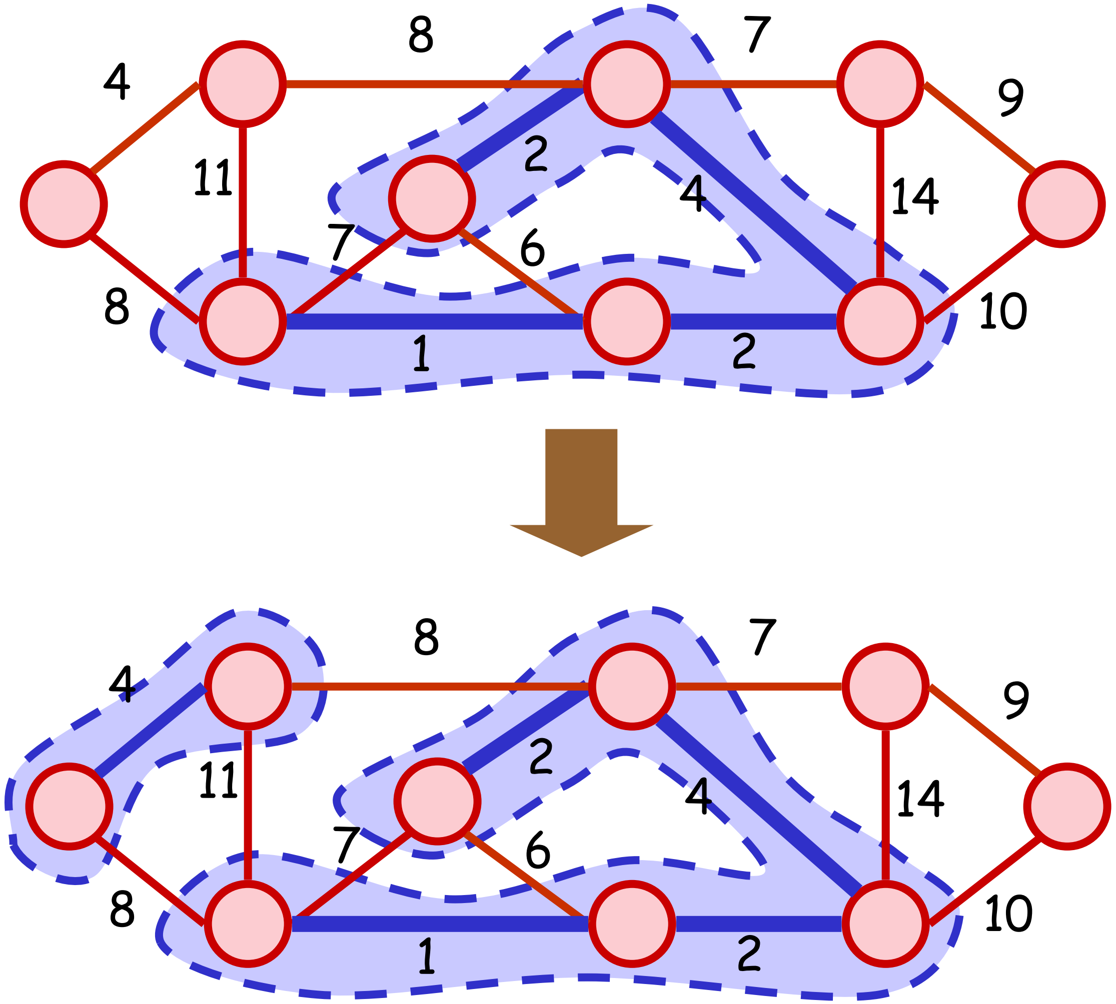
Example



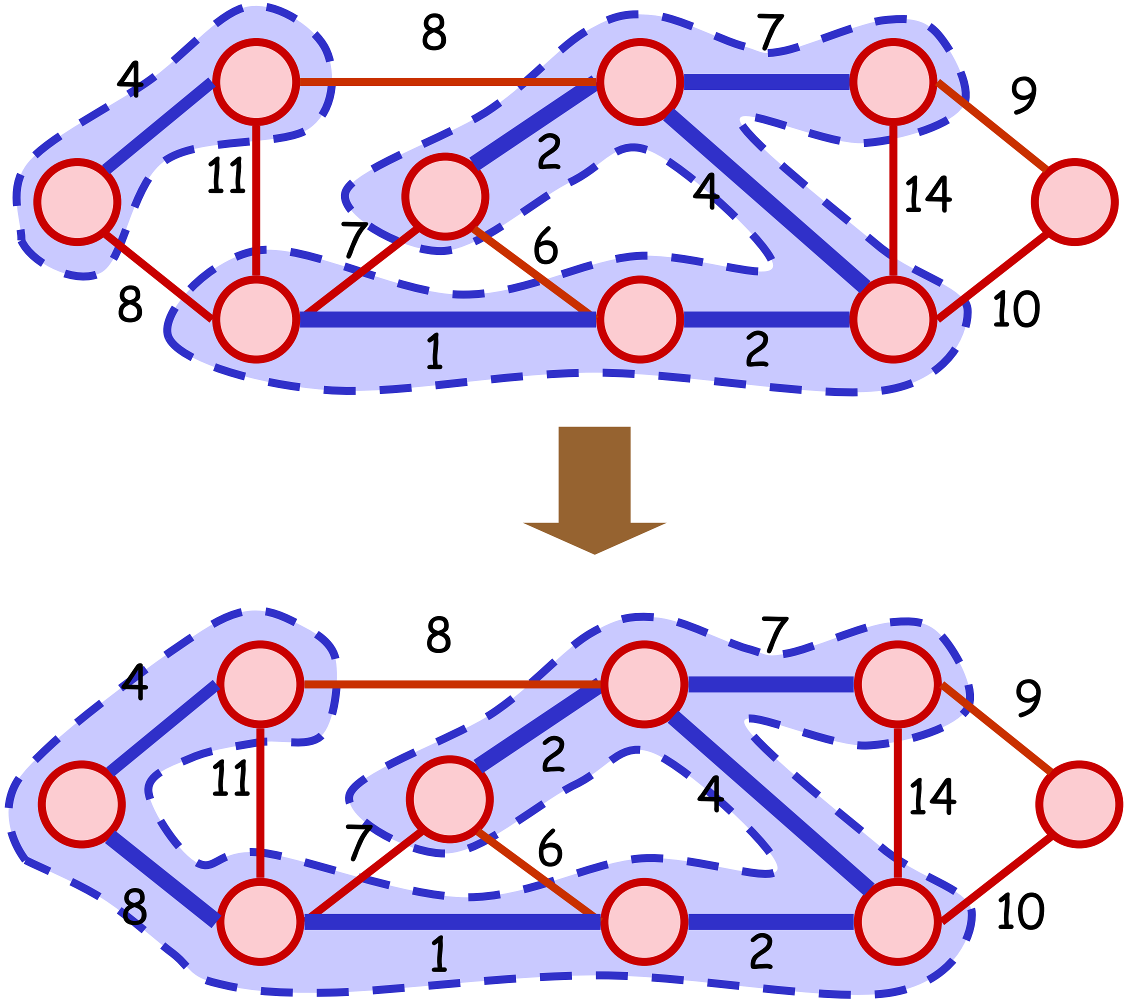
Example



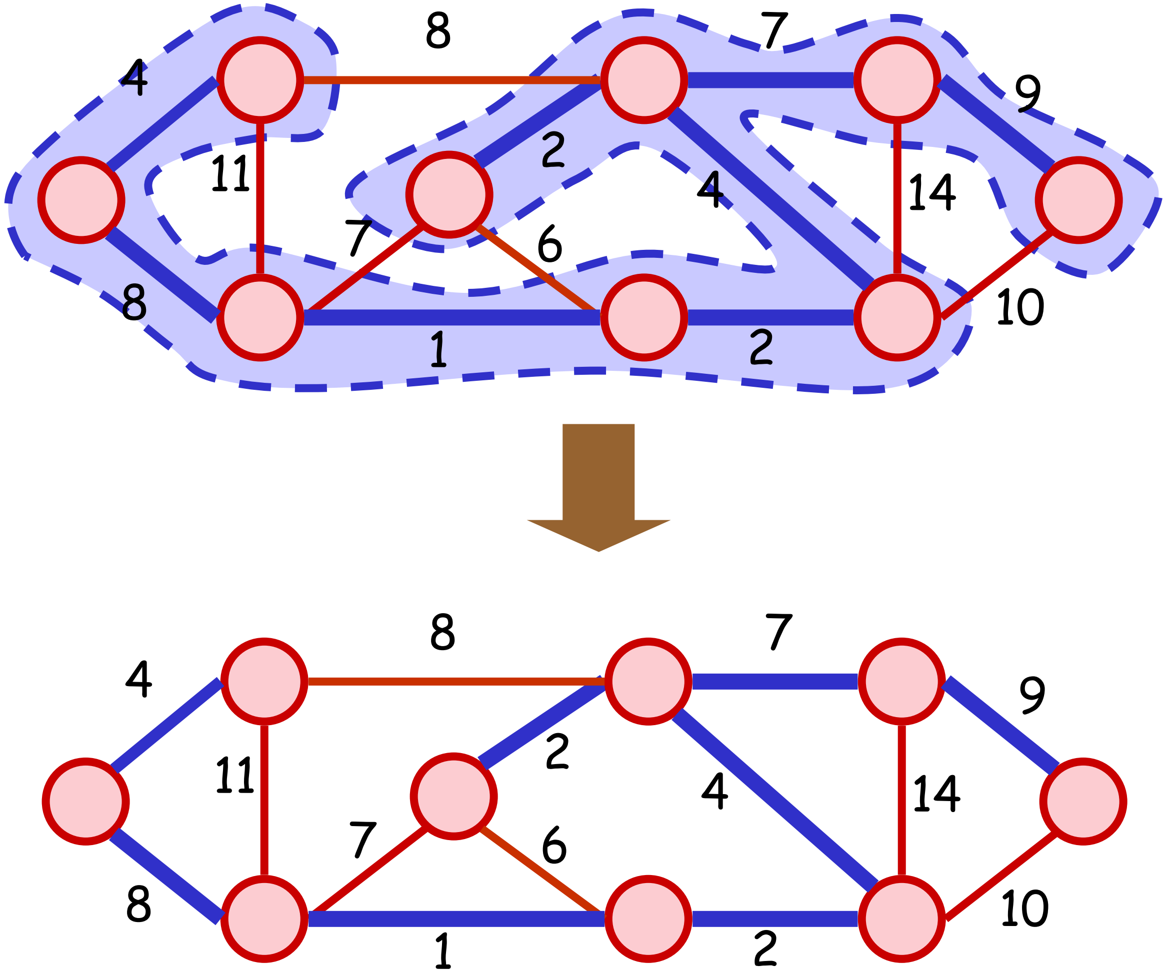
Example



Example



Example



Performance

- Kruskal's algorithm can be implemented efficiently using Union-Find :
- First, sort edges according to the weights
- At each step, pick the cheapest edge
    - If end-points are from different component, we perform Union (and include this edge to the MST)

➔ Time for Union-Find = $O(E\alpha(E))$

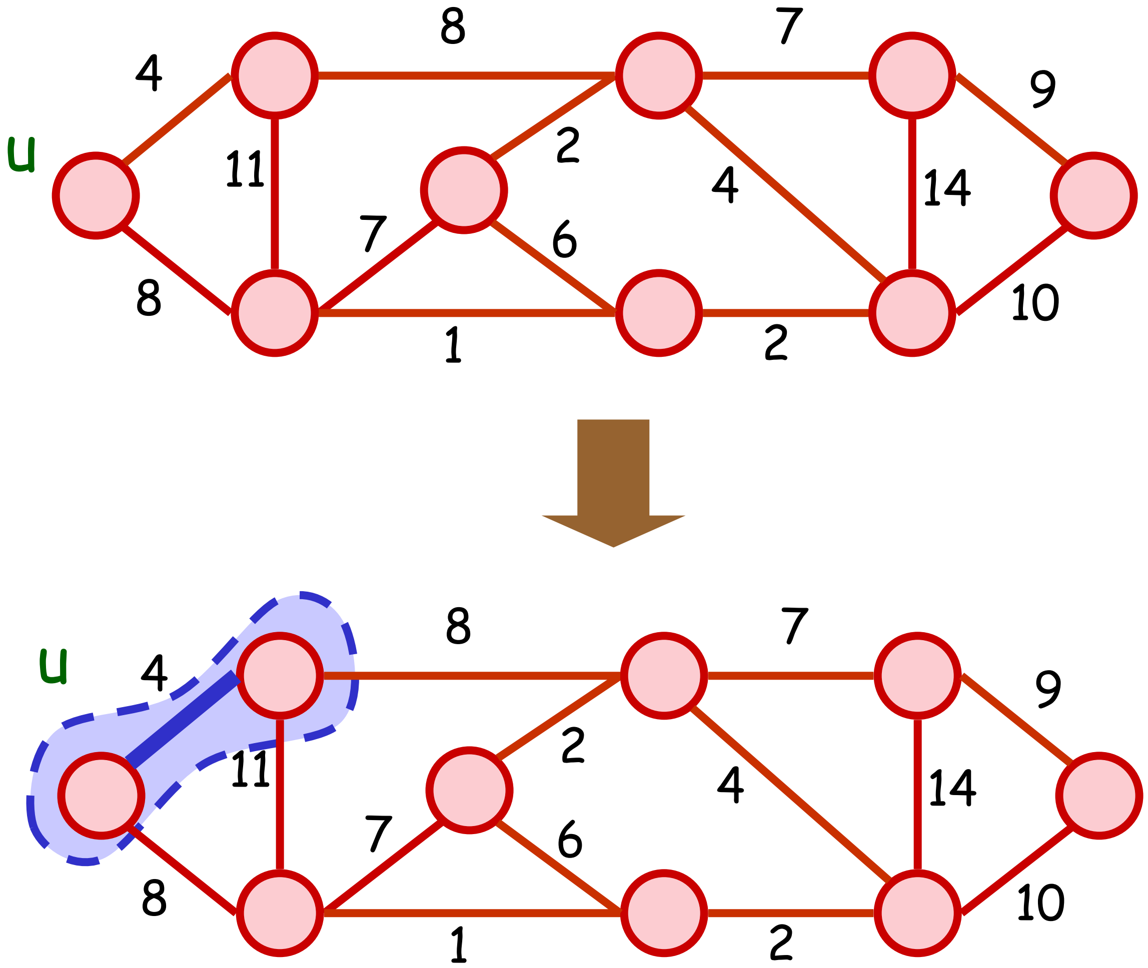
Total Time: $O(E \log E + E\,\alpha(E)) = O(E \log V)$

Prim's Algorithm

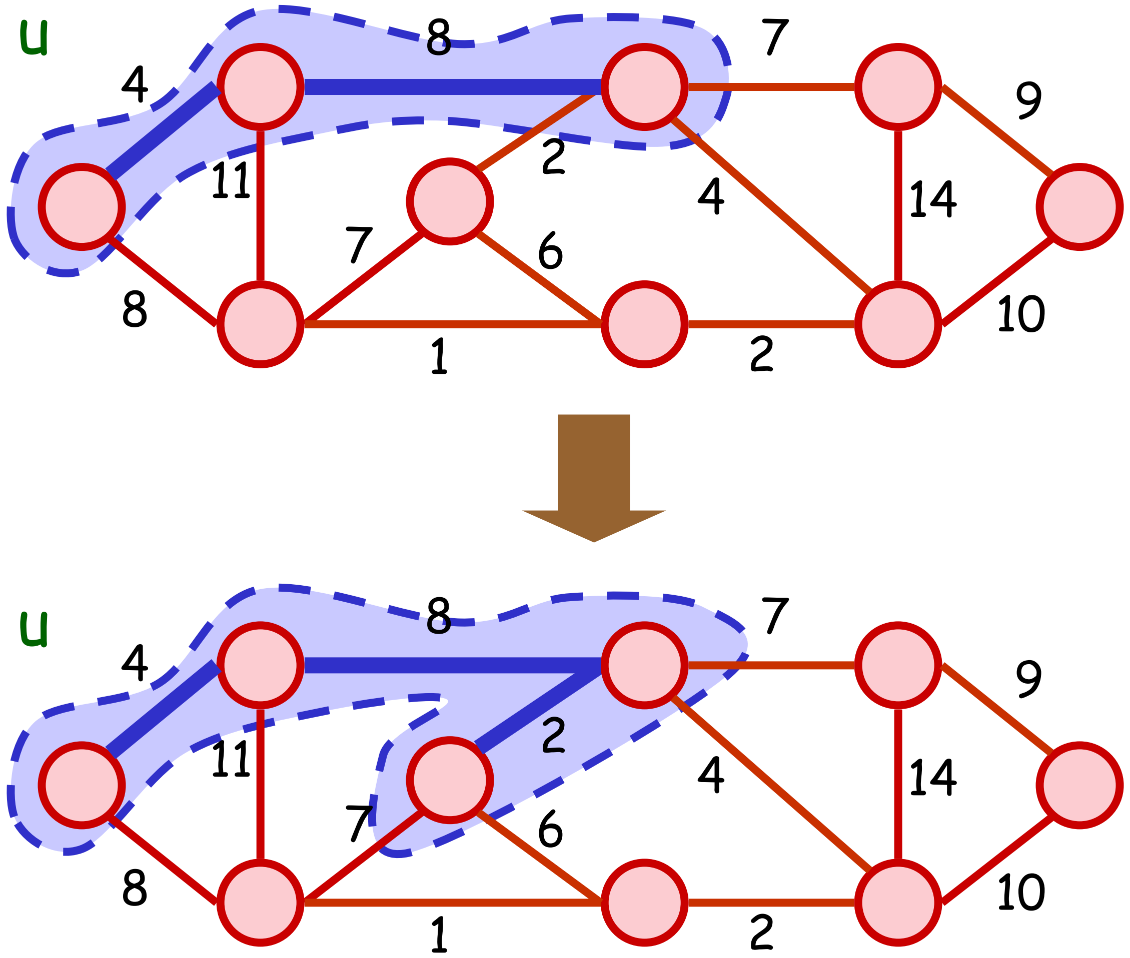
Prim-MST(G, u)

- Set u as the source vertex
- Find the cheapest (non-self-loop) edge from u, say, (u,v)
- Merge v into u to obtain G*
- Prim-MST(G*, u)

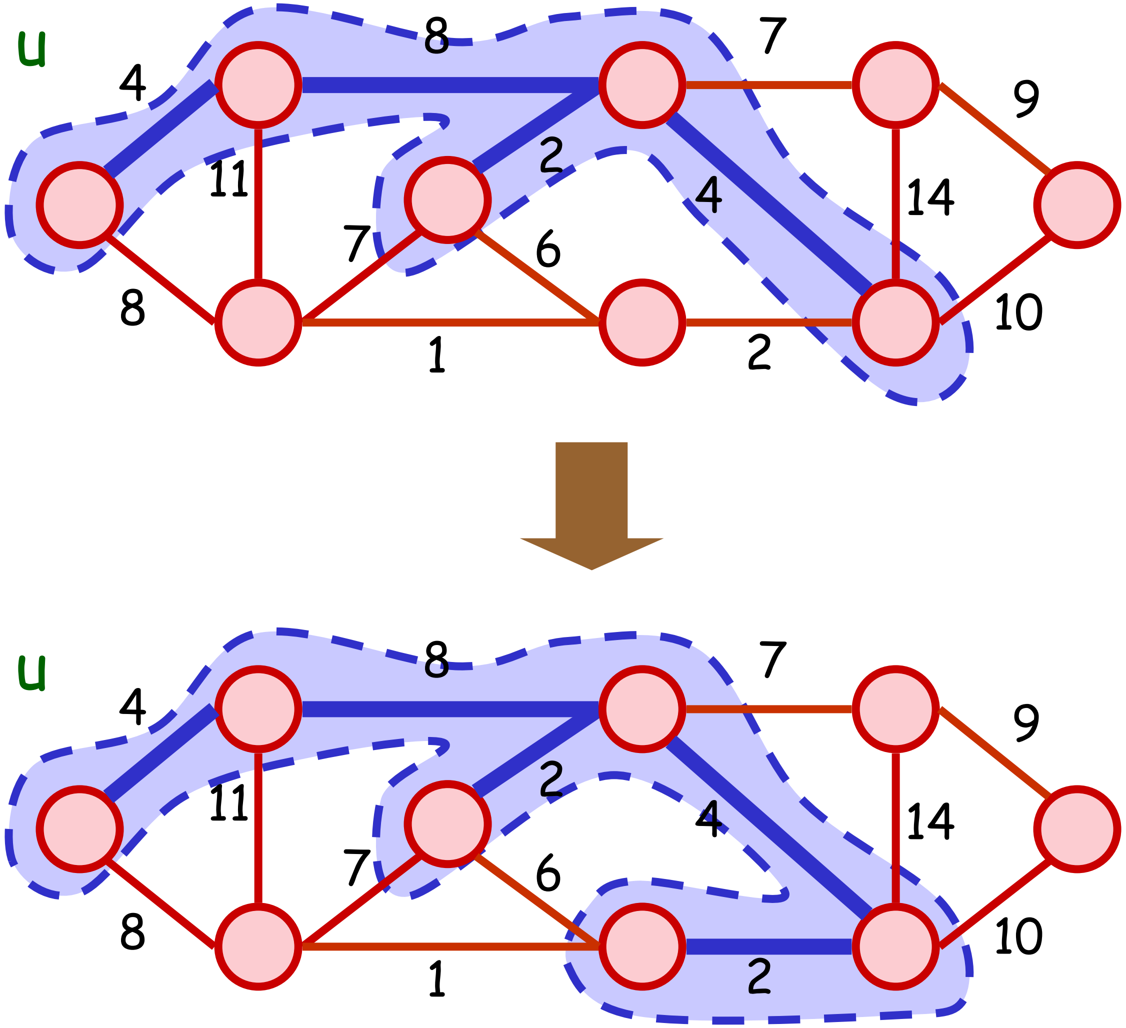
Example



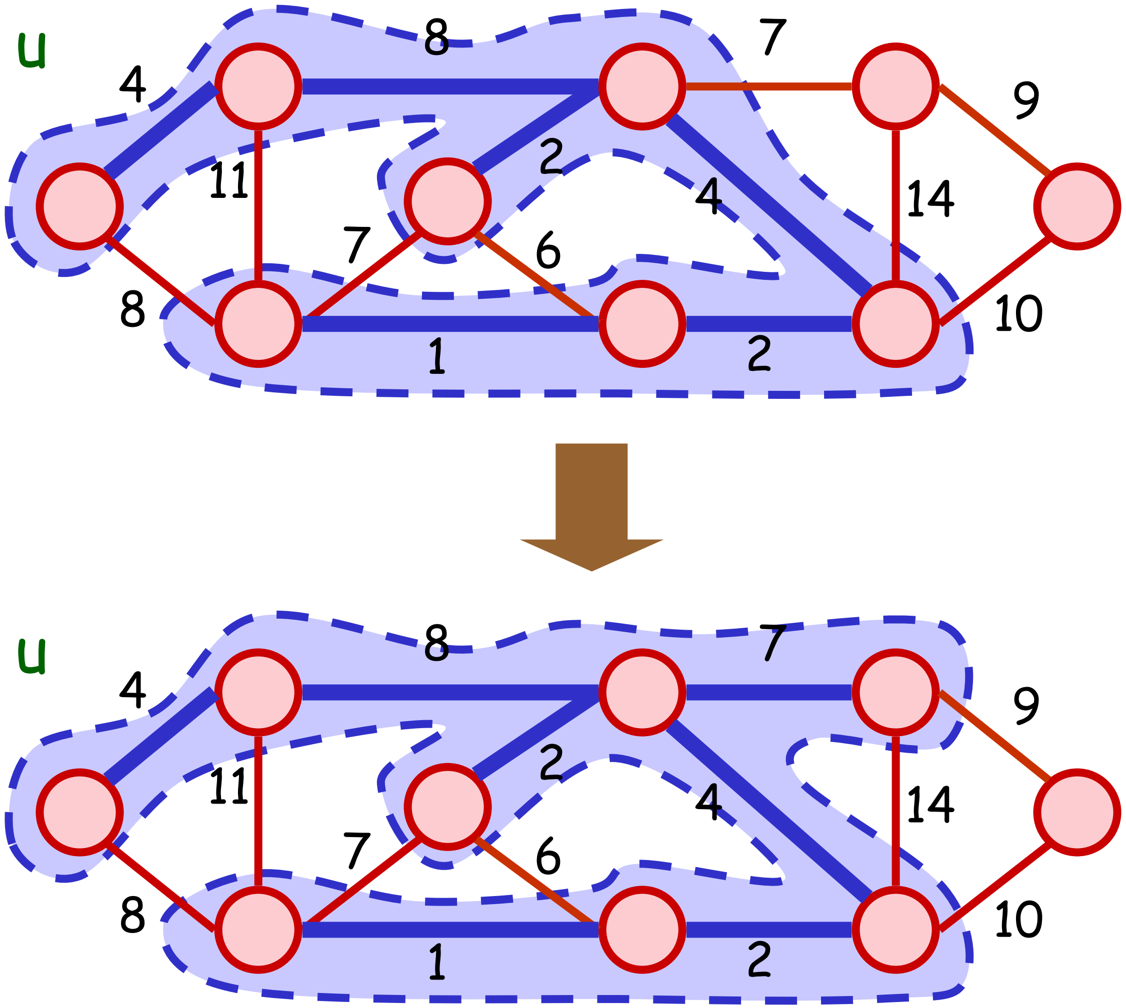
Example



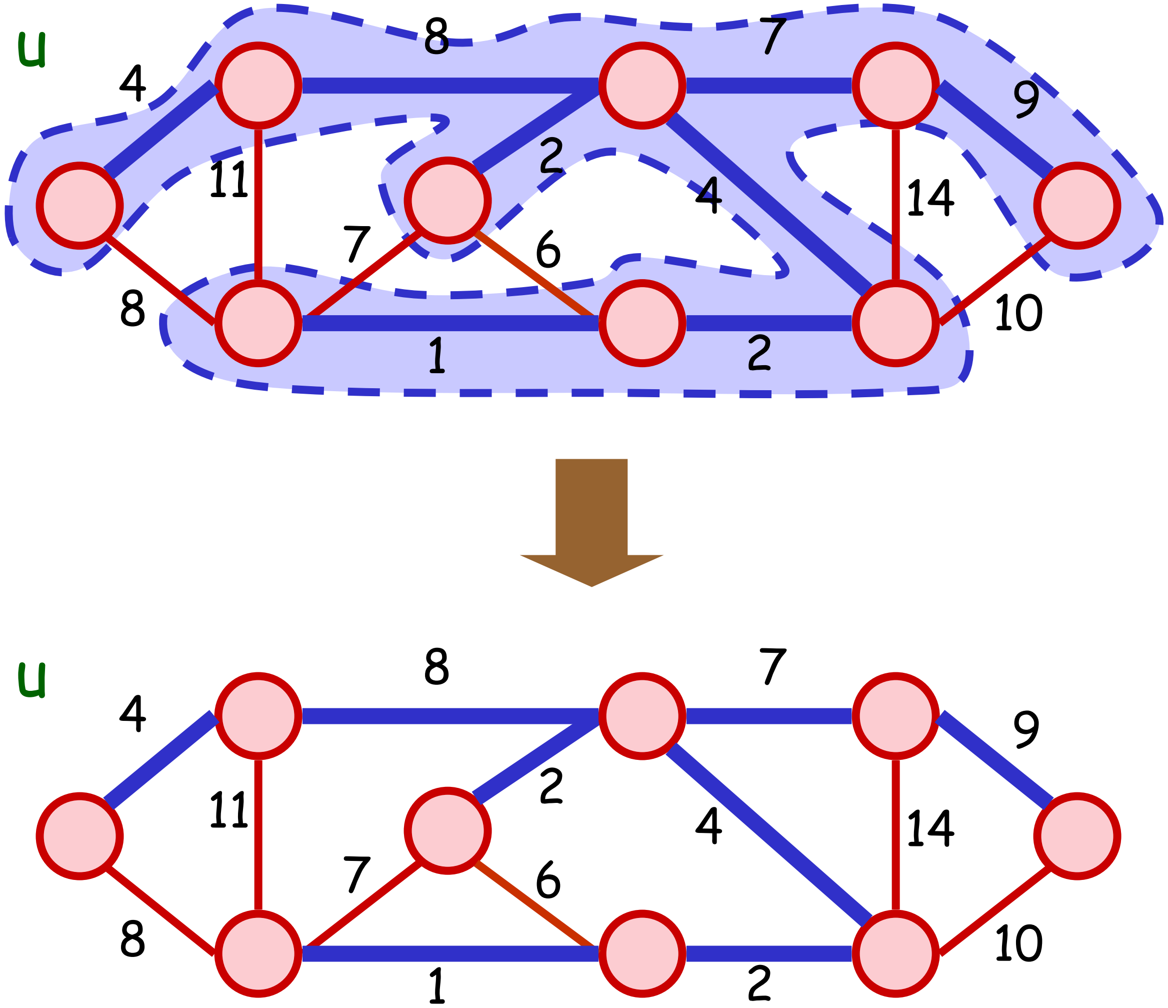
Example



Example



Example



Performance

- Prim's algorithm can be implemented efficiently using Binary Heap H:
- First, insert all edges adjacent to u into H
- At each step, extract the cheapest edge
  - If an end-point, say v, is not in MST, include this edge and v to MST
    - Insert all edges adjacent to v into H
- At most O(E) Insert/Extract-Min

➔ Total Time: O(E log E) = O(E log V)

Performance (speed-up)

- In fact, Prim's algorithm can be sped up using a Fibonacci Heap F
  - Instead of keeping edges in the heap, we keep distinct vertices
  - This avoids $\Theta(E)$ Extract-Min in the worst case
- At the beginning, each vertex (except source) is inserted into the heap, with key = ∞
  - key represents distance between u and the vertex

Performance (speed-up)

- Next, we scan all adjacent edges in u and update the distance of the corresponding vertices (using Decrease-Key)

  ➔ the vertex with the smallest key must be joined to u with the cheapest edge

  (since key = distance from u)

- So, we extract the minimum vertex, scan all its adjacent edges, and update corresponding vertices …

Performance (speed-up)

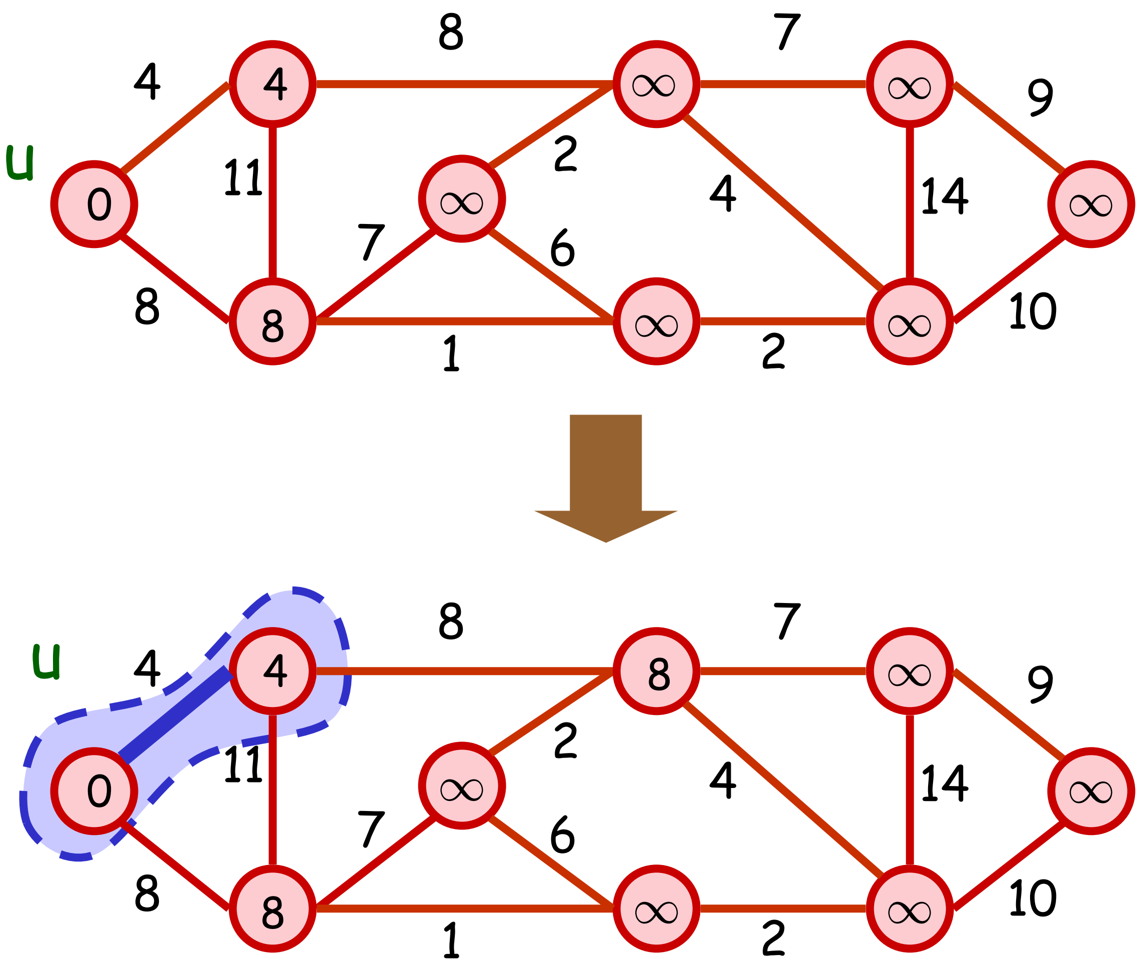
- The process is repeated until all vertices in the heap are gone

  ➔ MST obtained !

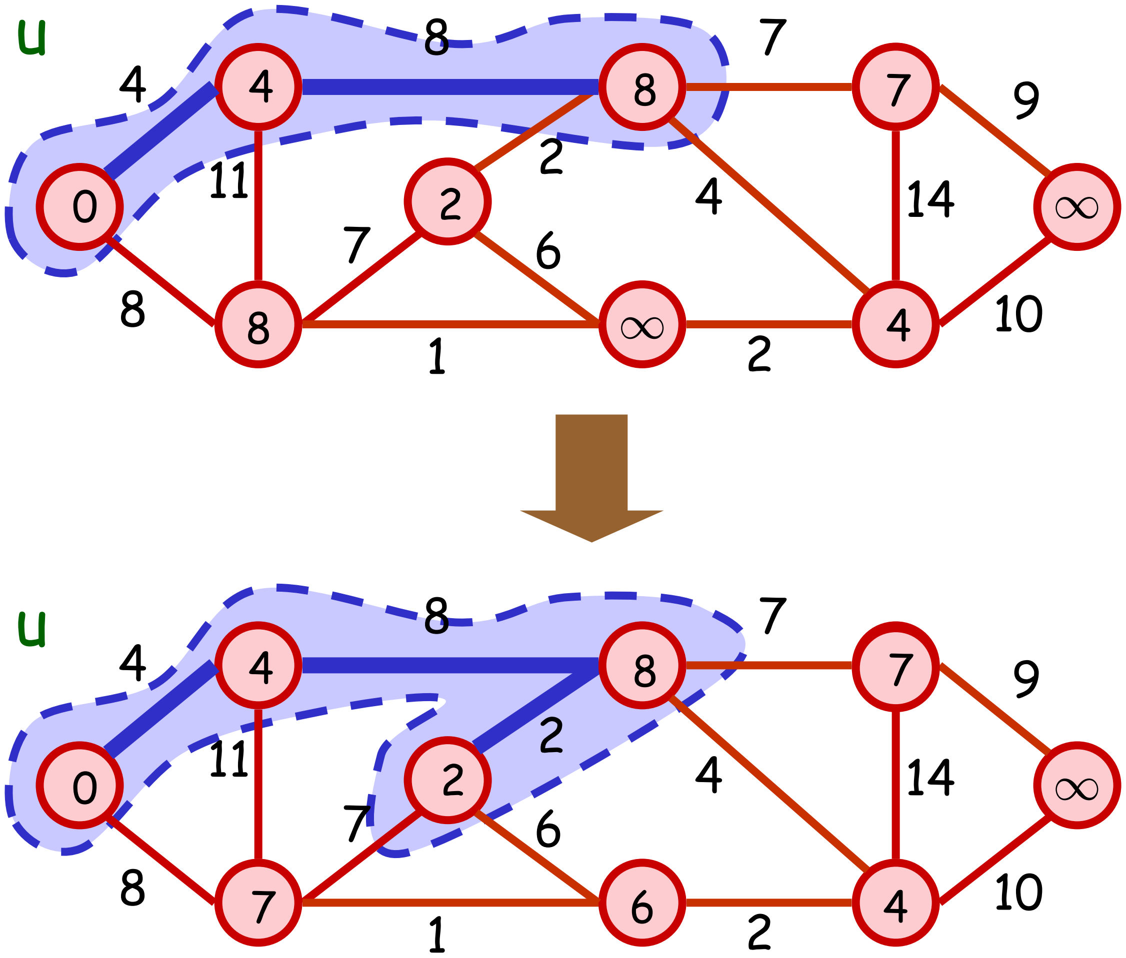
- Running Time:
  - O(V) Insert/Extract-Min
  - At most O(E) Decrease-Key

  ➔ Total Time: O(E + V log V)

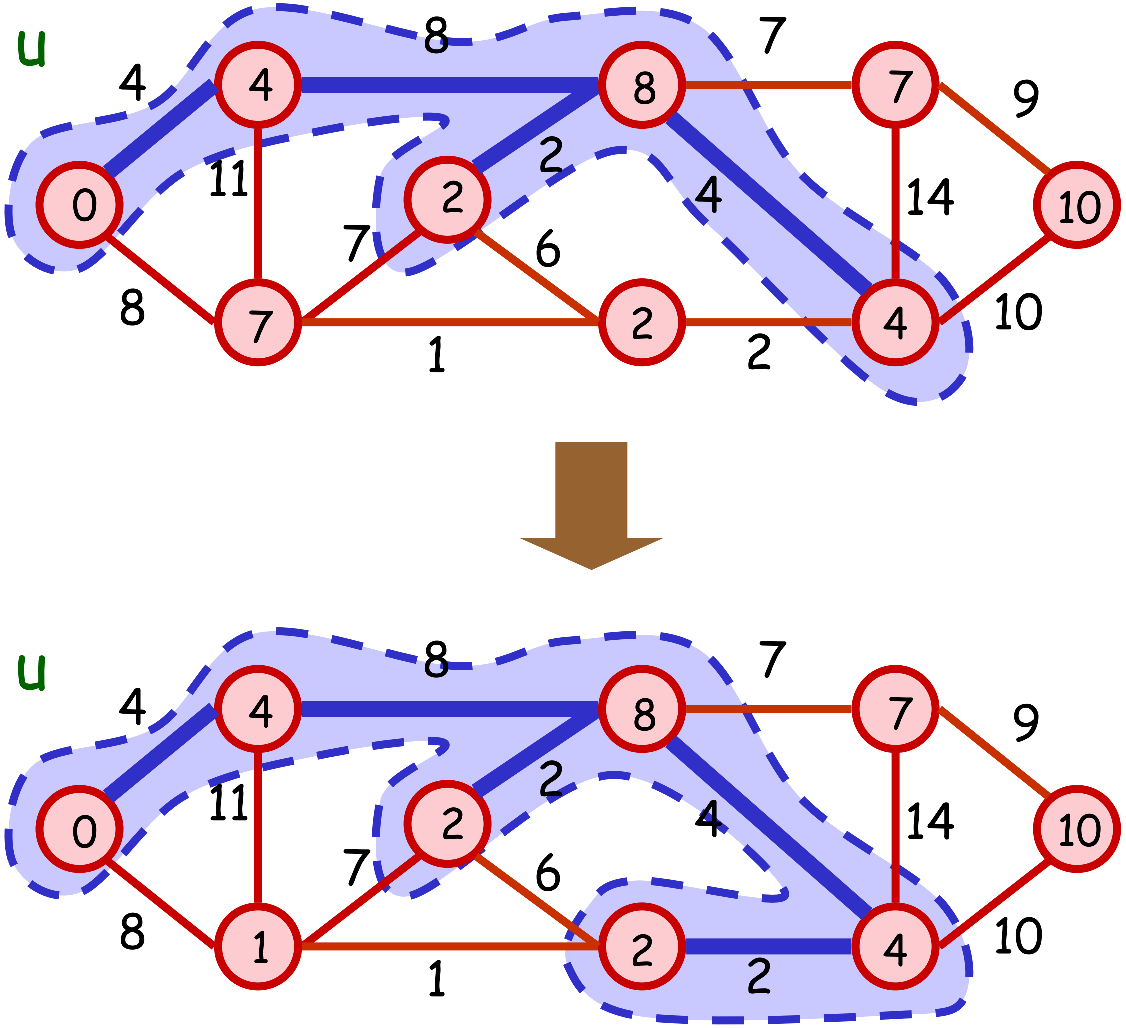
Example



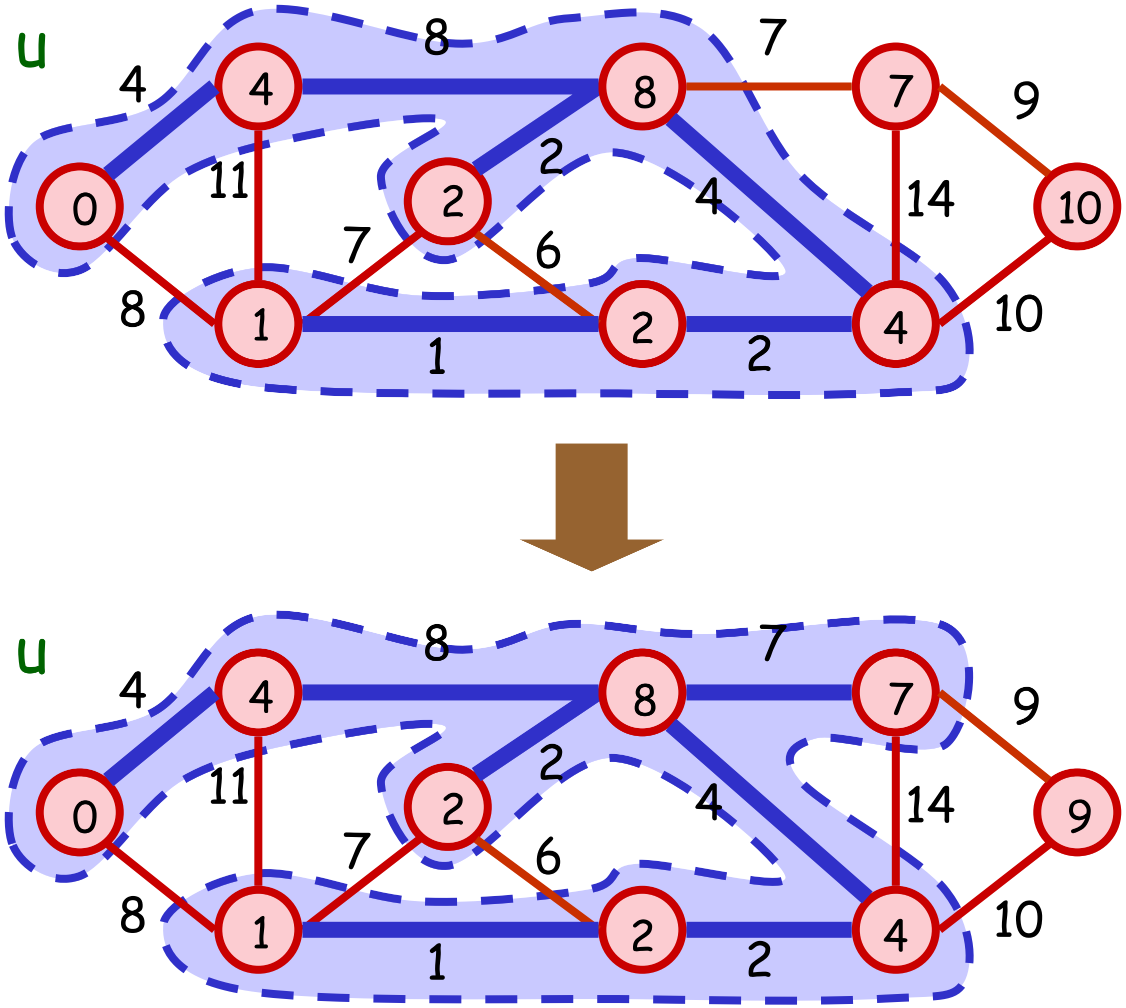
Example


u
4
4
8
8
7
7
9
0
11
2
2
4
14
∞
7
6
8
8
1
∞
2
4
10
u
4
4
8
8
7
7
9
0
11
2
2
4
14
∞
7
6
8
7
1
6
2
4
10

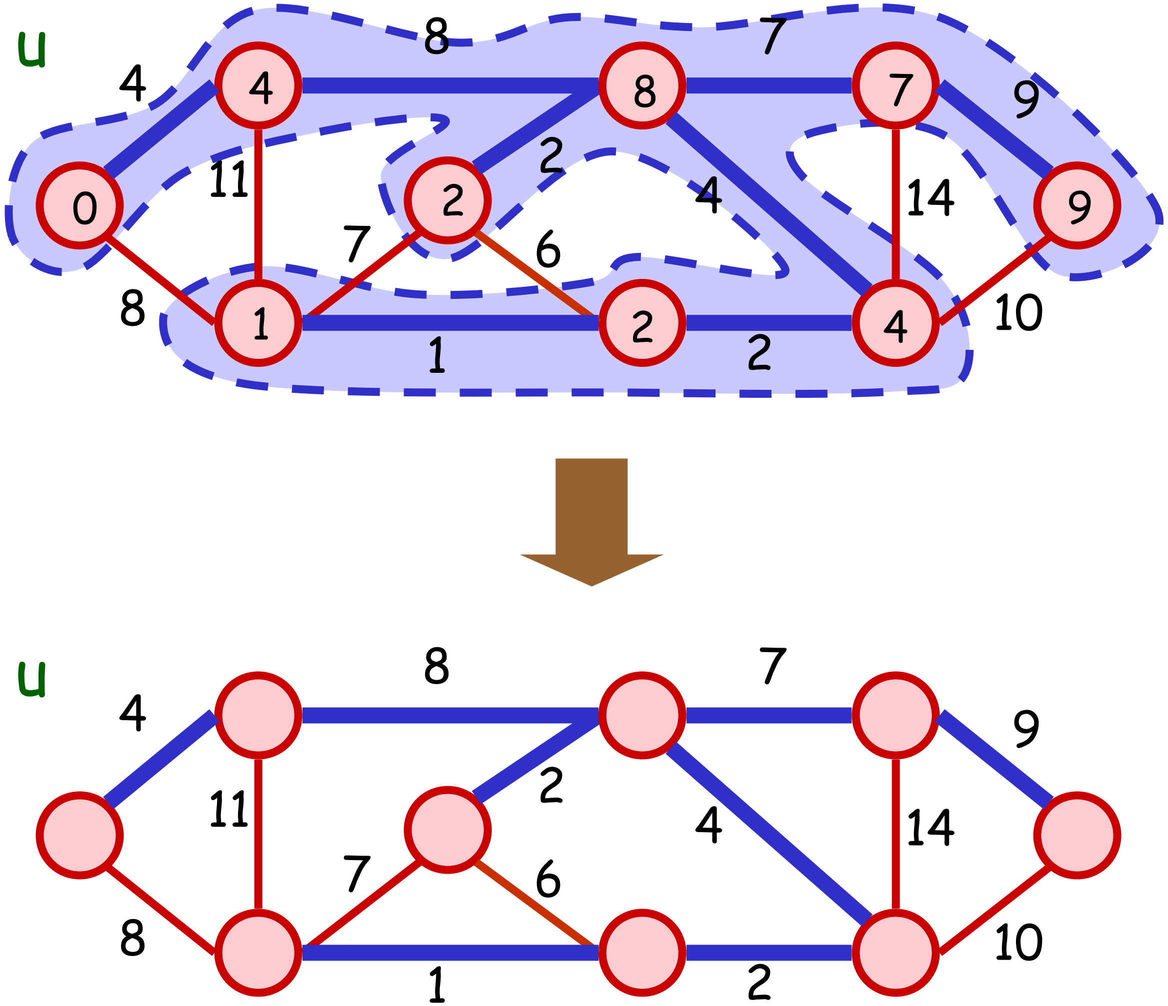
Example


u
4
4
8
8
7
7
9
0
11
2
2
4
14
10
7
6
8
7
1
2
2
4
10
u
4
4
8
8
7
7
9
0
11
2
2
4
14
10
7
6
8
1
1
2
2
4
10

Example


u
4
8
7
9
4
8
7
11
2
4
14
0
10
7
6
8
10
1
2
4
1
2
u
4
8
7
9
4
8
7
11
2
4
14
0
9
7
6
8
10
1
2
4
1
2

Example

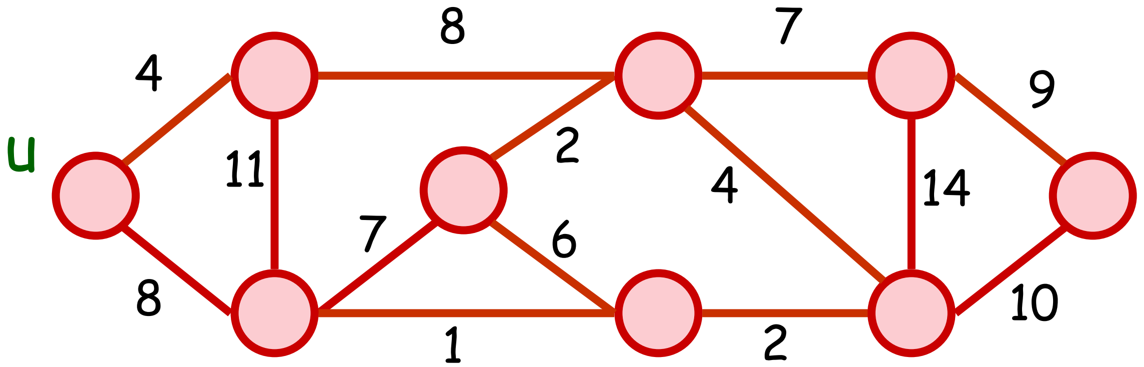


Borůvka's Algorithm

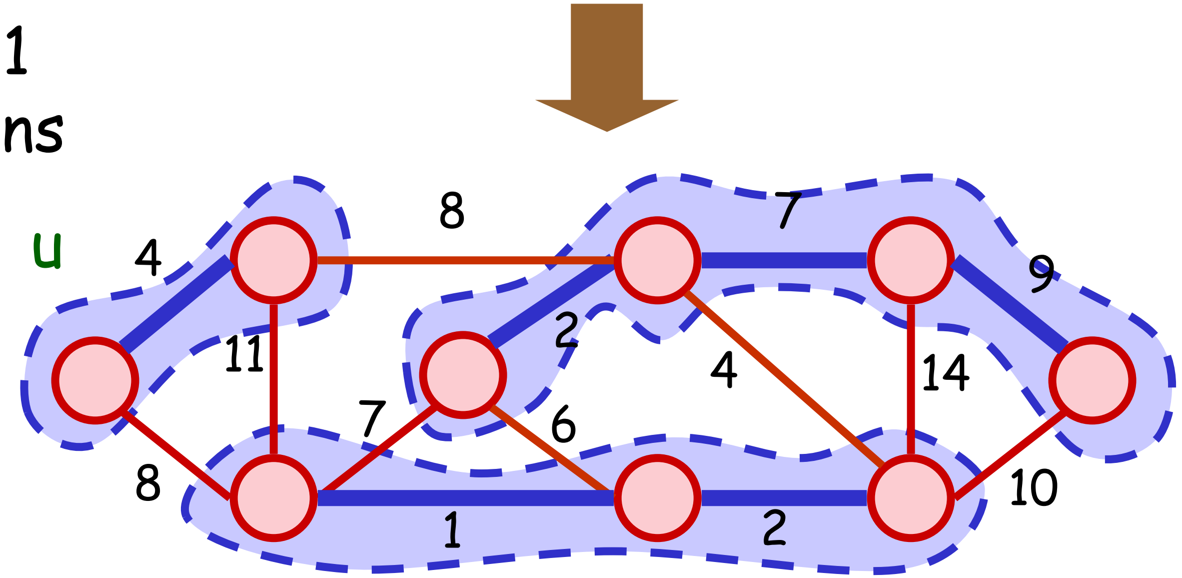
Borůvka-MST(G)

- Find cheapest adjacent edge e_v for each vertex v
- Contract all e_v to obtain G^*
- Borůvka-MST(G^*)

Example

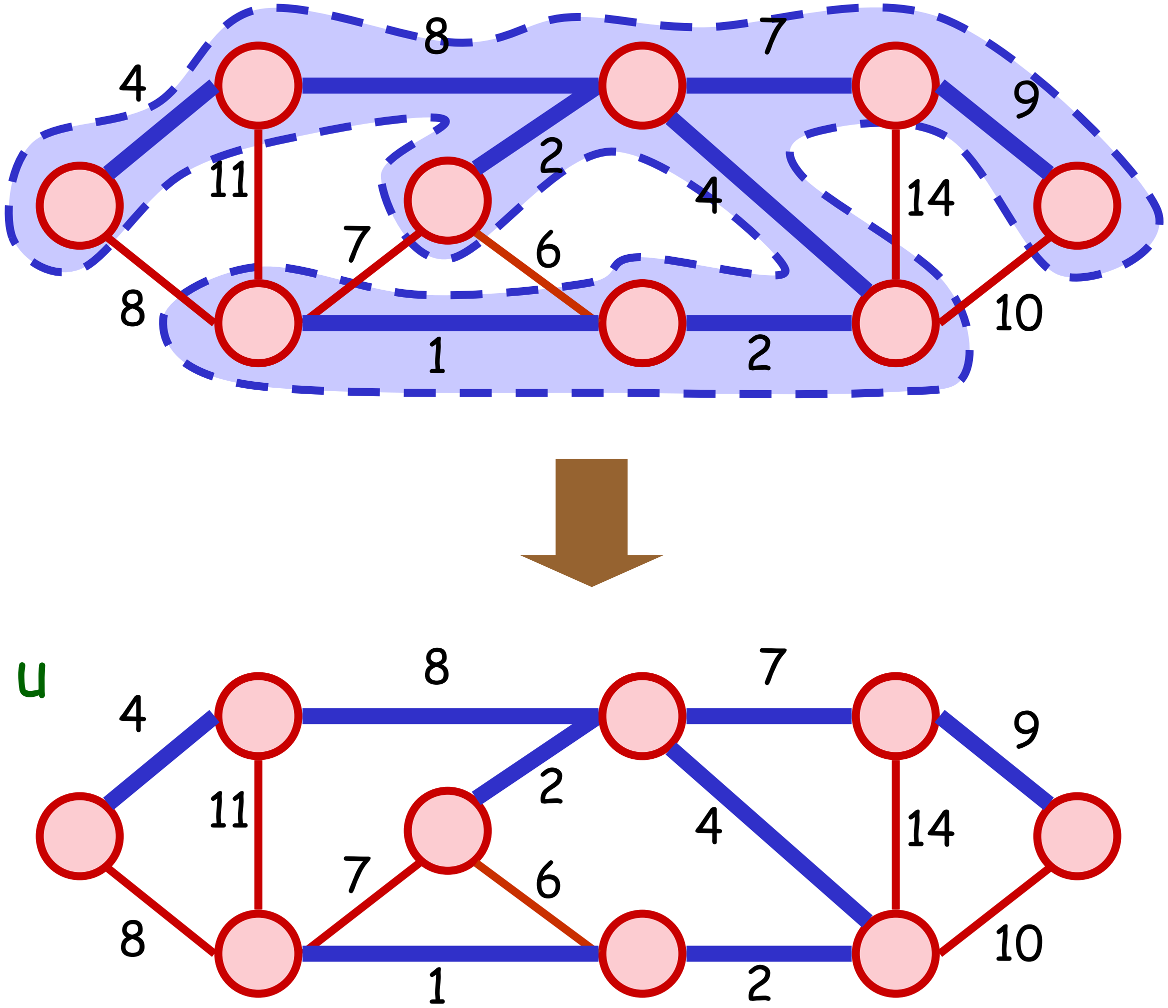


After 1
iterations



Example

After 2
iterations



Performance

- In Step 1 of Borůvka's algorithm, each vertex v needs to find e_v
    - can be done in O(E) time, without sorting of edges
- In Step 2, when all e_v are contracted, we need to re-label the end-points of the edges so that they refer to the new vertices in G^*
    - can be done in O(E) time, using DFS to find connected components

Performance

- After Step 2, each new vertex of G^* represents at least two vertices of G
  - #vertices in $G^* \leq V/2$

➔ In general, if Borůvka-MST() is called for k iterations,

#vertices in $G^* \leq V/2^k$

➔ At most $O(\log V)$ iterations

Total time: $O(E \log V)$

In practice, #iterations can be much smaller than $O(\log V)$

Modifying Borůvka

- Now, suppose we run Borůvka-MST() for only $k = \log \log V$ iterations

$$\#\text{vertices in } G^* \le V/2^{\log \log V} = V/\log V$$

$$\#\text{edges in } G^* \le E$$

- Then, we switch back to Prim
- Running Time:

$$\underbrace{O(E \log \log V)}_{\text{Borůvka}} + \underbrace{O(E + (V/\log V) \log V)}_{\text{Prim}}$$

$= O(E \log \log V)$ ← could be better than both !!