Analysis and Synthesis of Algorithms
Design of Algorithms

Minimum Cost Spanning Trees
Problem Motivation
Algorithms by Krushkal and Prim

Why Minimum Cost Spanning Trees?

- Problems in Science and Engineering
  - Formulated as directed and undirected graph problems
  - Connectivity of Network Nodes
  - Routing of Trucks Delivering Merchandise to Retailers

- Questions?
  - Can we Reach All Points of the Graph?
    - Is the graph connected?
    - If not are subsets of the nodes of the graphs connected?
    - Can each node reach any other node? (directed vs. undirected)
  - How Can we Best Reach All Nodes in the Graph?
    - Applications:
      - Route Planning.
      - Network Management in the Presence of Failures.

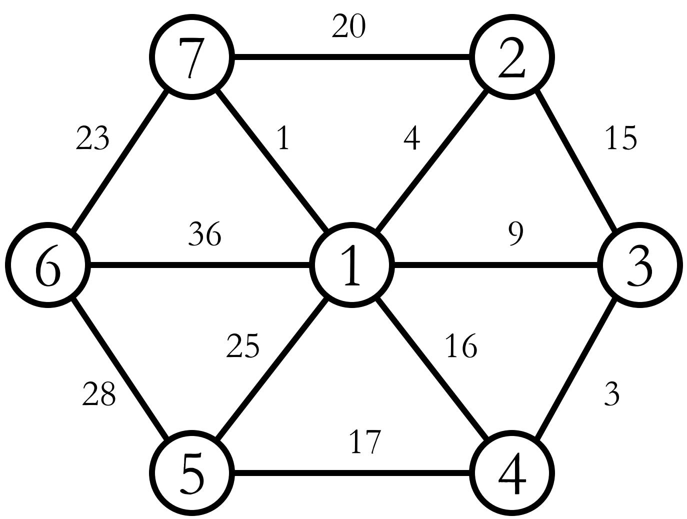
Spanning Tree Definitions

- An undirected graph G = (V, E) is **connected** if for any pair of vertices (or nodes) there exists (at least) one path connecting the two vertices.

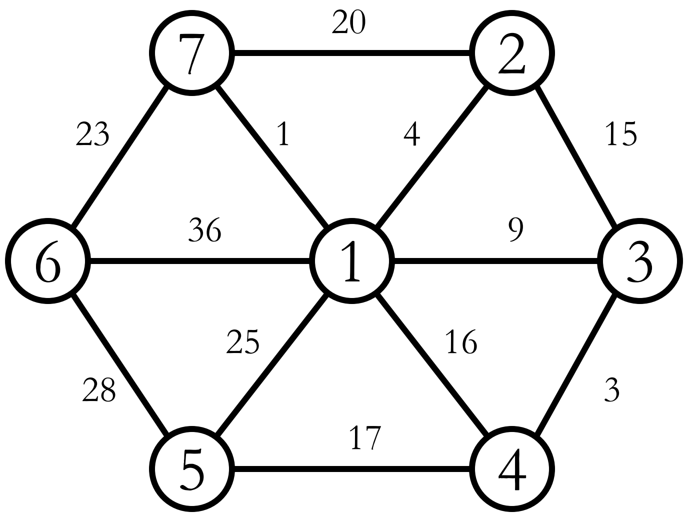
- Given an undirected, and connected, graph G = (V, E), a **spanning tree** is an acyclic, subset of the edges $T \subseteq E$ connecting all the vertices (or nodes) in G.
  - Observation: Given that G has |V| vertices the number of edges in |T| must satisfy: $|T| = |V| - 1$ (Proof: Use the pigeon principle)

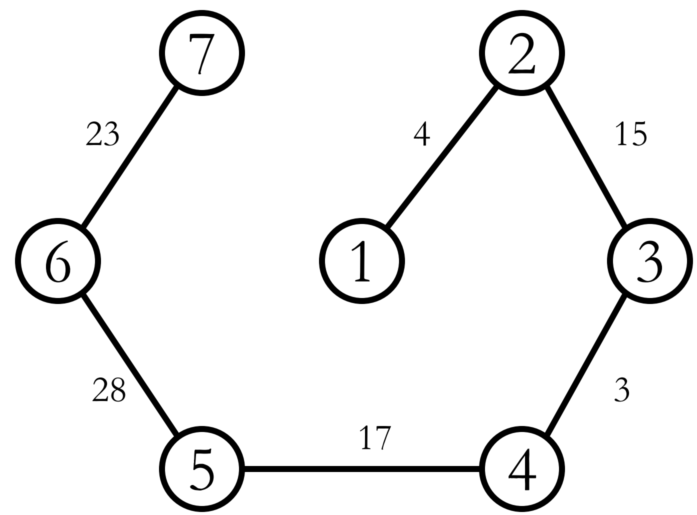
- Given a spanning tree, its **cost** is the sum of the costs associated with its edges.

Spanning Tree Example



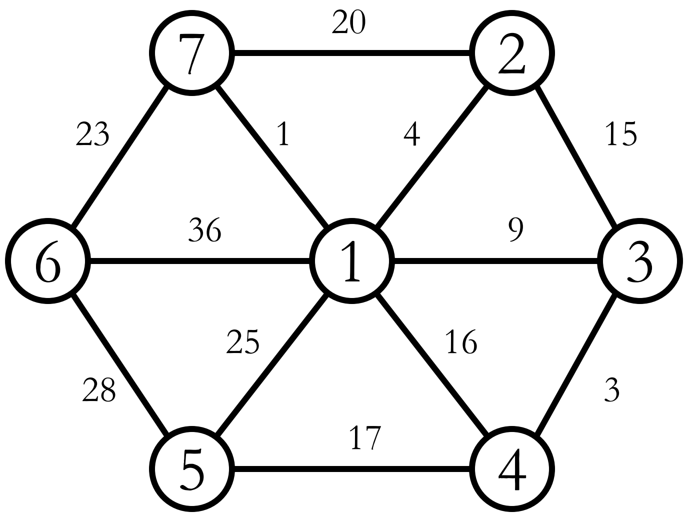
Spanning Tree Example

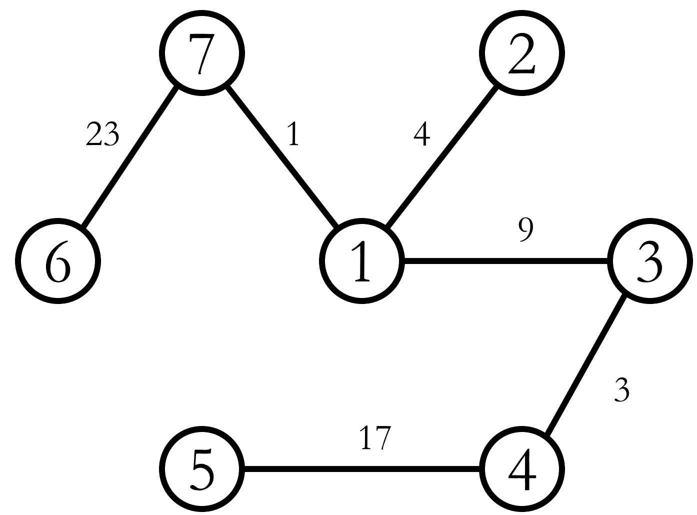




Cost: 90

Spanning Tree Example





Cost: 57

Minimum Cost Spanning Tree (MST)

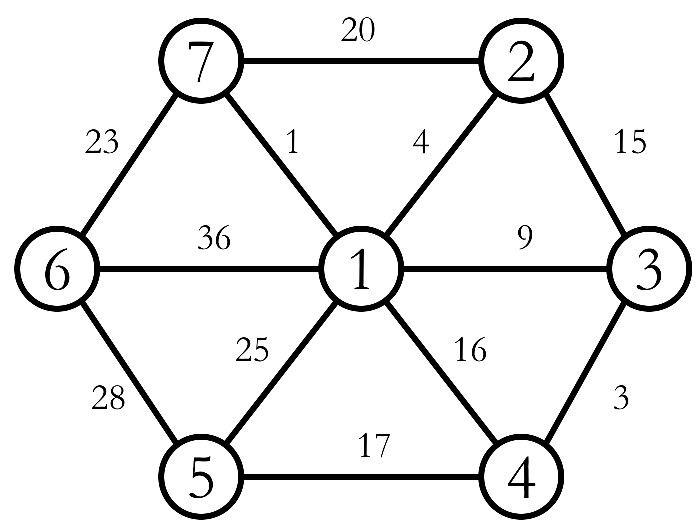
- Given the graph G = (V, E), connected, undirected, with weight function $w : E \rightarrow \mathbf{R}$, identify a spanning tree T, such that the summation of the edge weights of T is minimal

$$\min W(T) = \sum_{(u,v) \in T} w(u, v)$$

Brute-Force Algorithm

- Algorithm:

```
function MST-BruteForce(Graph G, function w)
  S = generateSpanningTrees(G);
  for each s ∈ S do
    s_c= Cost(s,w);
  select s ∈ S with min cost;
  return s
```



- Probably Not a Good Idea…
  - The number of Trees can be Huge…
  - Matrix-Tree Theorem
  - Compute the Laplacian of the adjacency matrix $t(G) = \frac{1}{n}\,\lambda_1\lambda_2 \ldots \lambda_{n-1}$
  - Number of distinct Trees is the product of all non-zero eigenvalues

Building an MST

- Greedy Approach:
  - Maintain a subset tree A of the graph G
  - Identify edge (u,v) added to A such that
    - $A \cup \{(u,v)\}$ is still a subset tree A
    - Creates no ***cycles***, *i.e.*, is a ***safe*** edge

- Algorithm:

```
function MST-Generic(Graph G, function w)
  A = ∅;
  while A is not a spanning tree do
      identify safe edge (u,v) for A;
      A = A ∪ {(u,v)};
  return A
```

MST Definitions

- A **cut** (S, V-S) of an undirected graph G=(V,E) is a partion of V into disjoint sets of nodes.

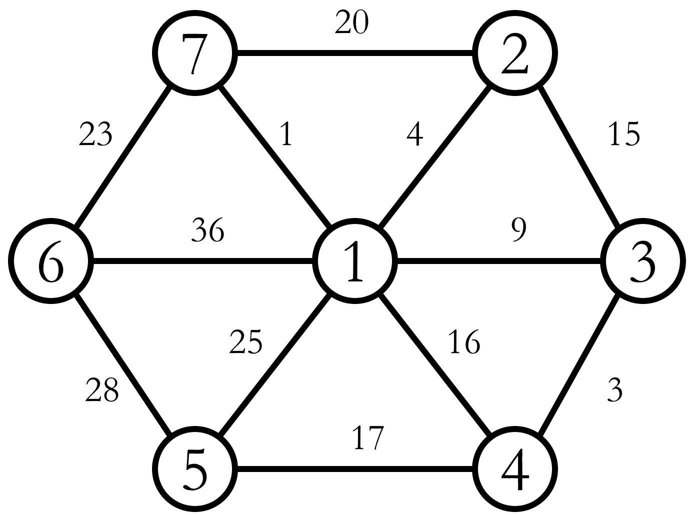
- An edge (u,v) ∈ E **crosses** the cut (S, V-S) if one of its nodes is in S and the other node is in V-S.

- A cut **contains** a set of edges A if no edge ∈ A crosses the cut.
  - all edges connect nodes in either S or (V-S).

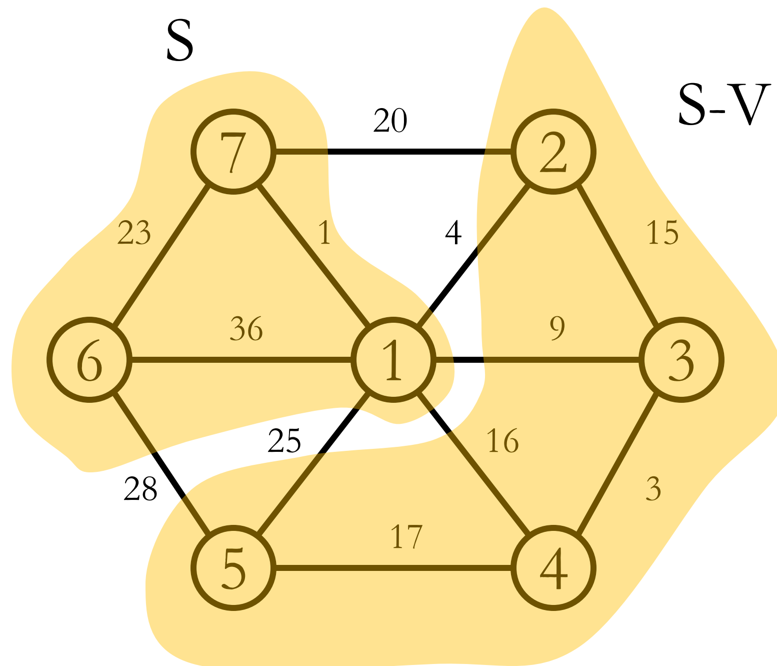
- An edge that crosses a cut with the lowest cost is designated as the **lightest edge**

- An edge is **safe** for A if its inclusion in A does not create any cycles in A

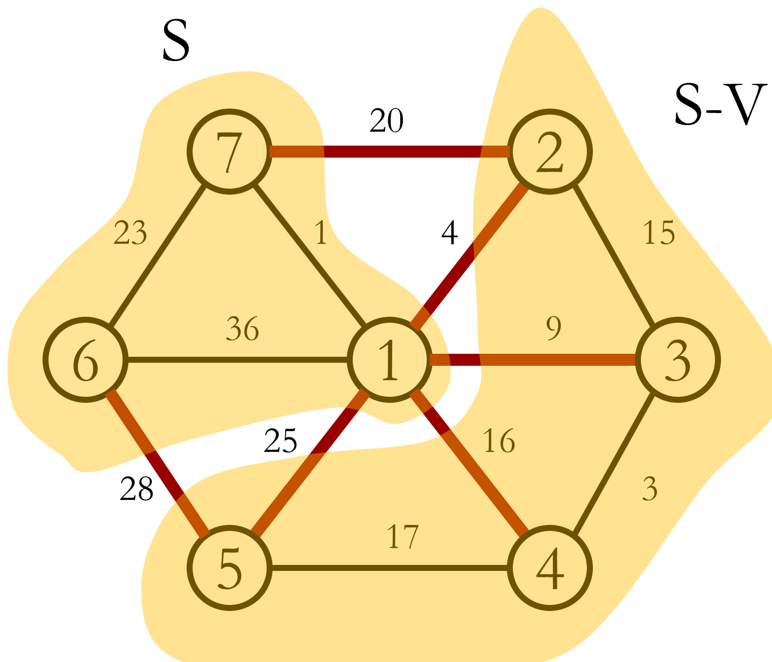
MST Definitions: Examples



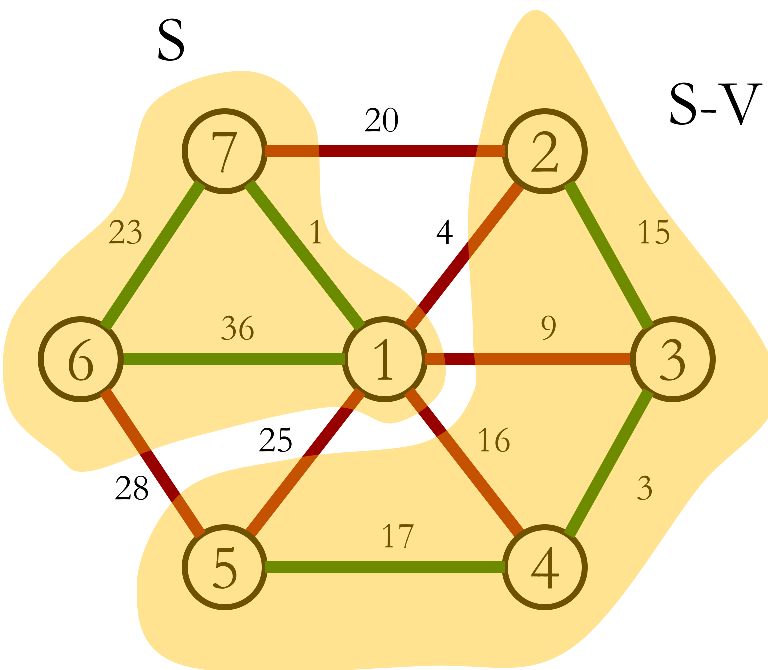
MST Definitions: Examples



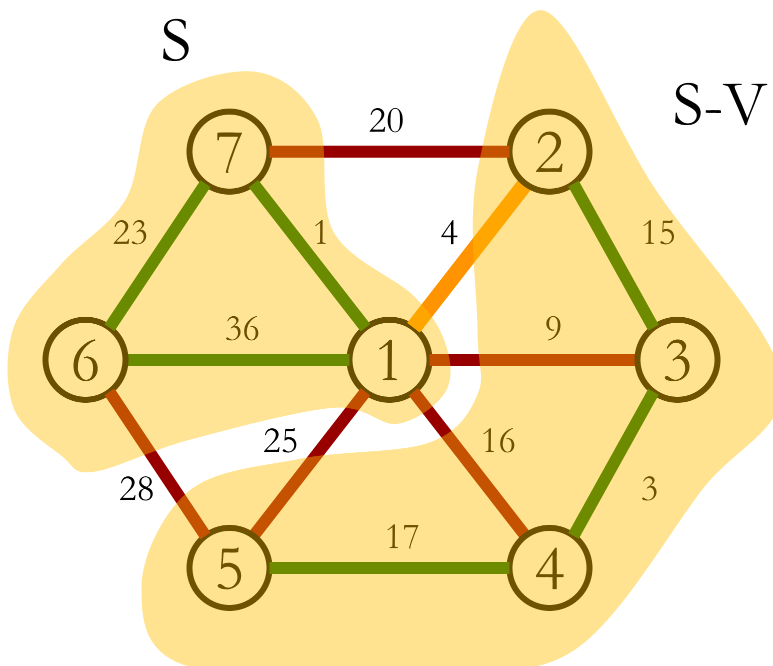
MST Definitions Examples



MST Definitions Examples



MST Definitions Examples



MST Construction Properties

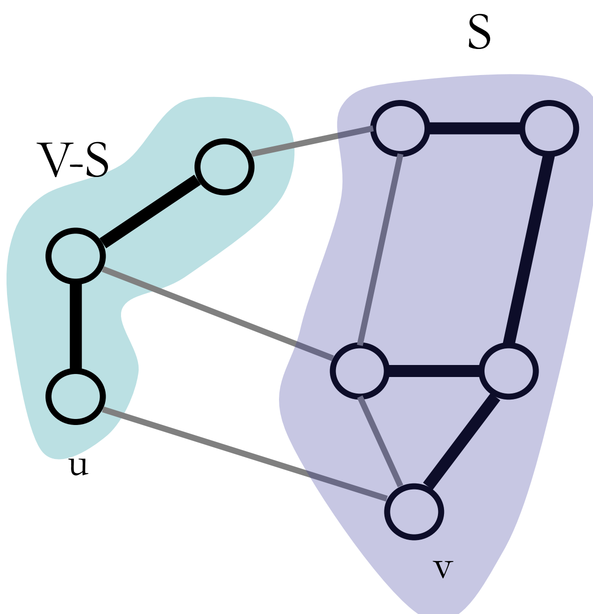
- MSTs satisfy two very useful properties:
  - **Cycle Property:** The heaviest edge along a cycle is NEVER part of an MST.
  - **Cut Property:** Split the vertices of the graph any way you want into two sets A and B. The lightest edge with one endpoint in A and the other in B is ALWAYS part of an MST.

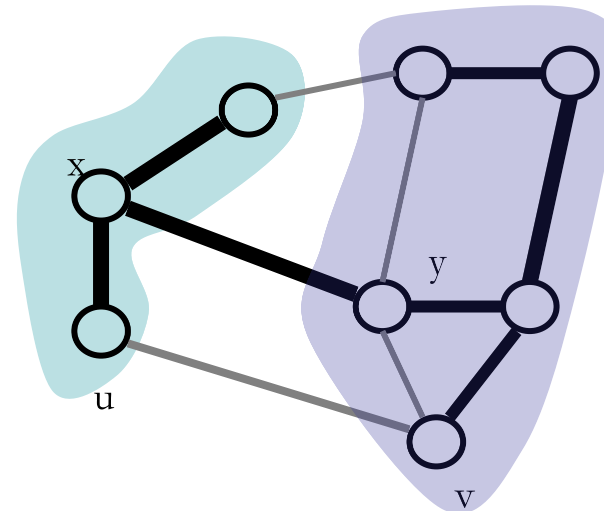
- **Observation:** If you add an edge to a tree and you create (exactly) one cycle, you can then remove any edge from that cycle and get another tree out.
- This observation, combined with the cycle and cut properties form the basis of all of the greedy algorithms for MSTs.

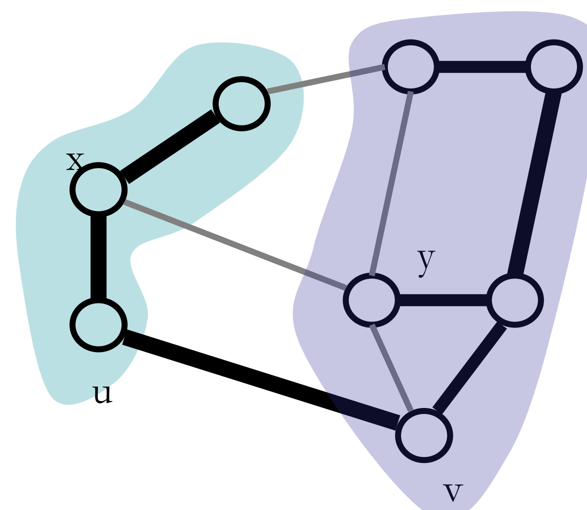
MST Optimality

- Let G=(V, E) be a connected, undirected graph with edge weight function w and let A $\subset$ E be included in a MST T, then for any cut (S, V-S) that **contains** A
  – If (u,v) is the **lightest edge** that **crosses** (S, V-S)
  – Then (u,v) is a **safe edge** for A
- **Proof:**
  – Given that (u,v) crosses the (S, V-S) its addition to A does not create a cycle as either u or v are not in either S or V-S and the cut contains A
  – The resulting tree of adding (u,v) to A has the lowest cost
    - If there was an edge (x,y) with lower cost that edge would also have been crossing the cut (see next slide)
    - The resulting spanning tree would not have the lowest cost as we would have picked (u,v) at that particular step.
- MST Algorithm Correctness by Induction

MST Optimality







Kruskal's Algorithm

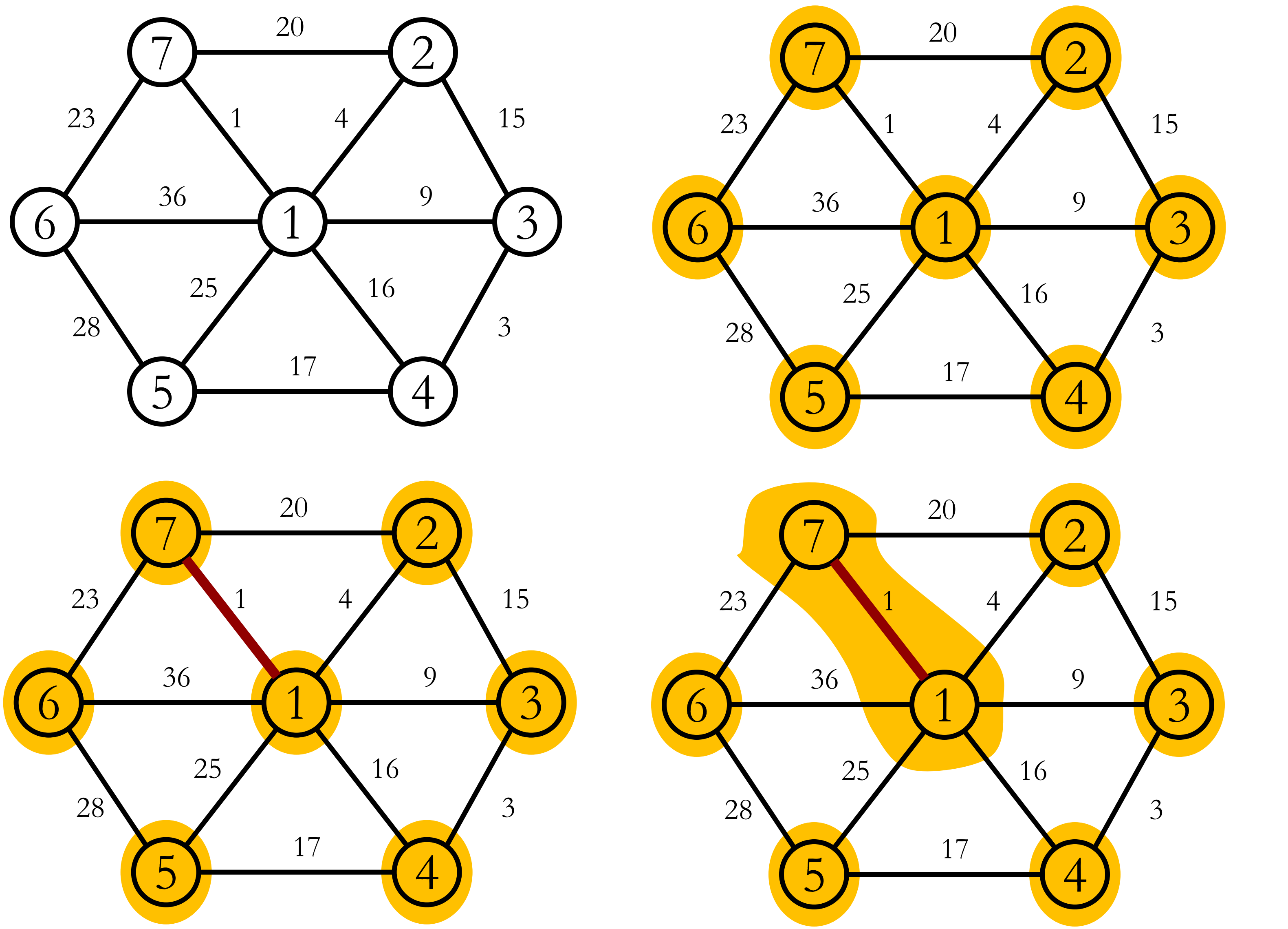
- Algorithm Outline:
  - Start with each isolated node as its own cluster
  - Pick the lightest edge $e \in E$
    - if e connects two nodes in different clusters, then e is added to the MST and the clusters merged,
    - otherwise ignore it
  - Continue until $|V| - 1$ edges are added
- Implementation:
  - Algorithm maintains a forest of trees or subset $A \subset E$
  - Uses a disjoint-sets data structure for representing and merging clusters
  - Each set or cluster represents a sub-tree of the final MST

Kruskal's Algorithm

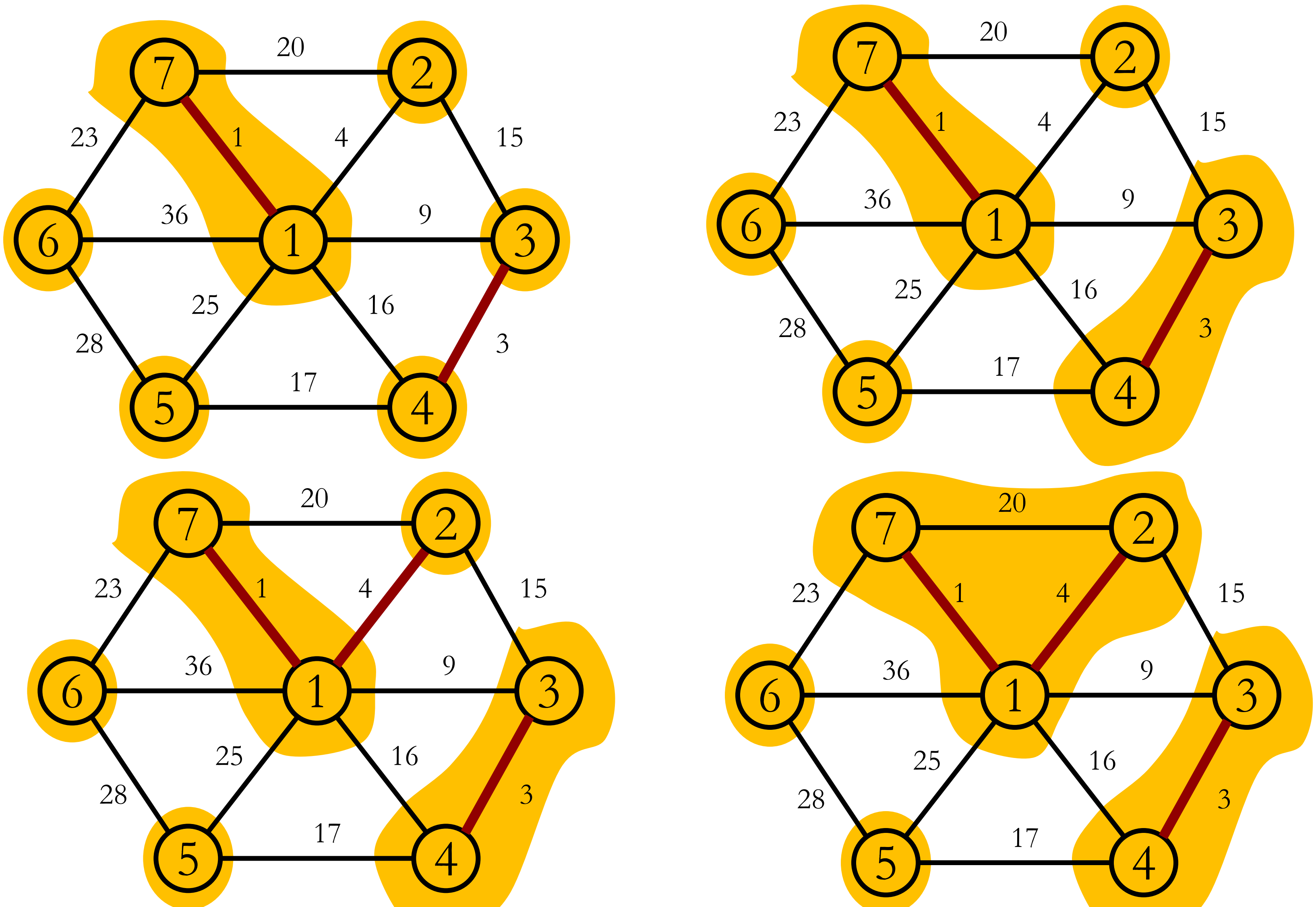
- Algorithm:

```
function MST-Kruskal(G,w)
  A = ∅;                                // initialization of MST as empty
  foreach v ∈ V[G] do
    MakeSet(v);                         // creates a cluster for each v
  sort edges ∈ E by non-decreasing order of weight;
  foreach (u,v) ∈ E[G] in sorted order do
    if FindSet(u) ≠ FindSet(v) then
        // (u,v) is the lightest and safe edge for A
        A = A ∪ {(u,v)};
        Union(u,v);                     // merge clusters for u and v
        if (|A| = |V(G)|-1)             // early exit
          break;
return A;
```

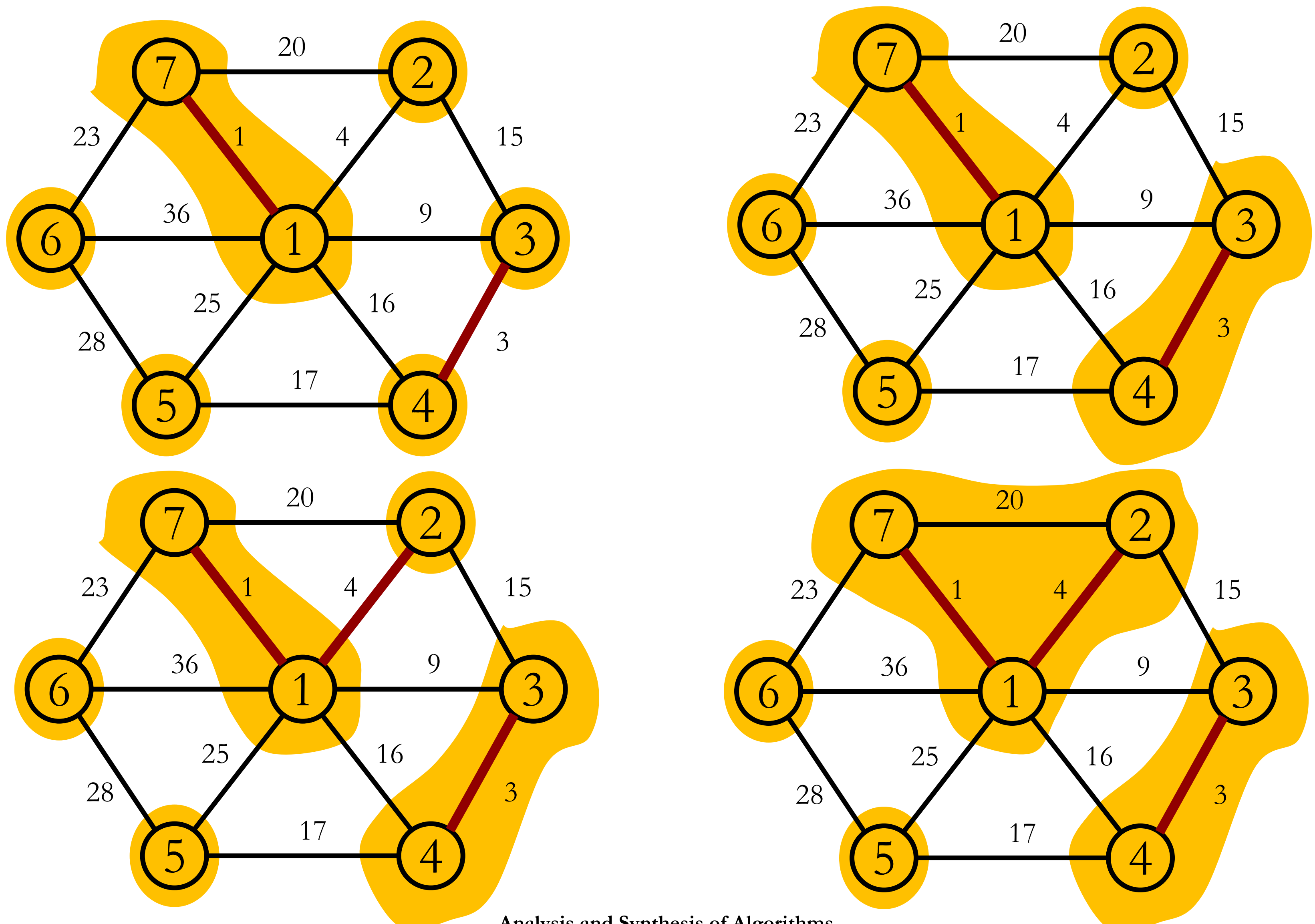
Kruskal's Algorithm



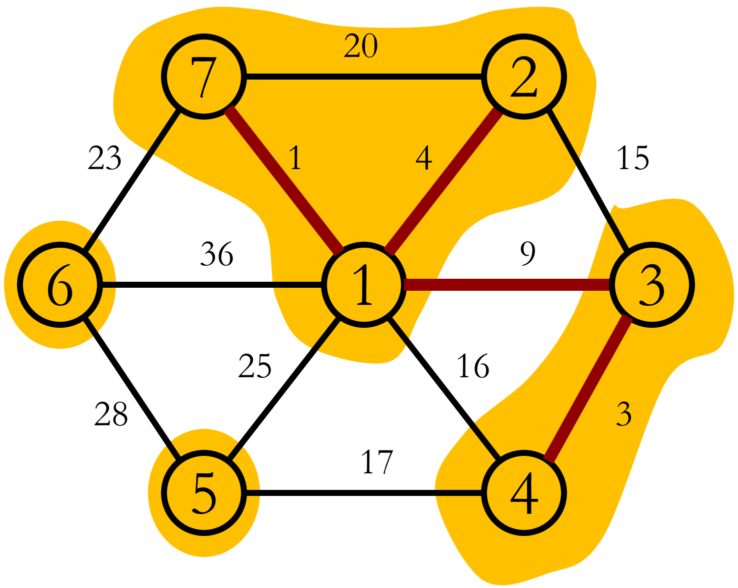
Kruskal’s Algorithm

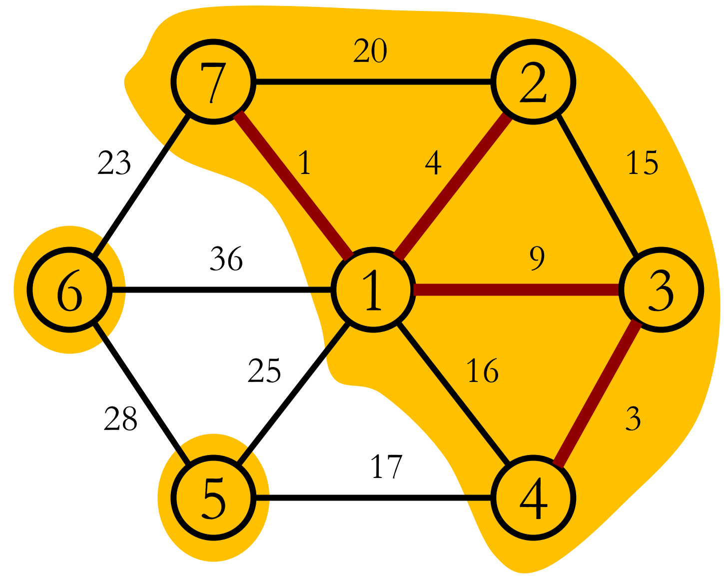


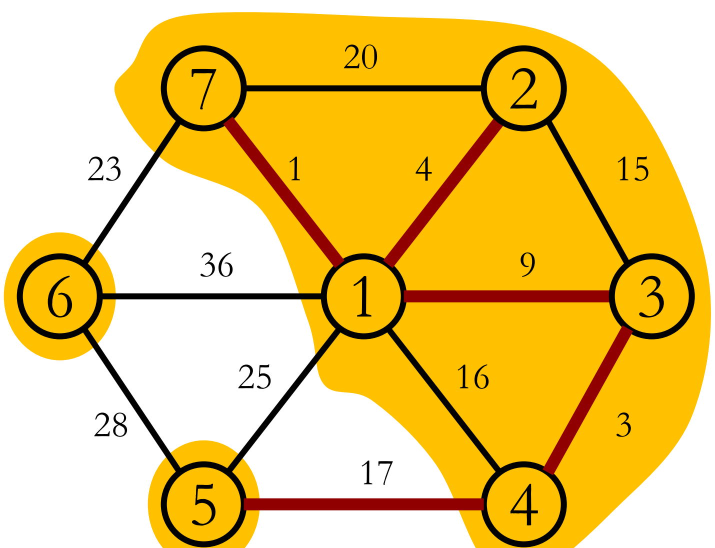
Kruskal's Algorithm

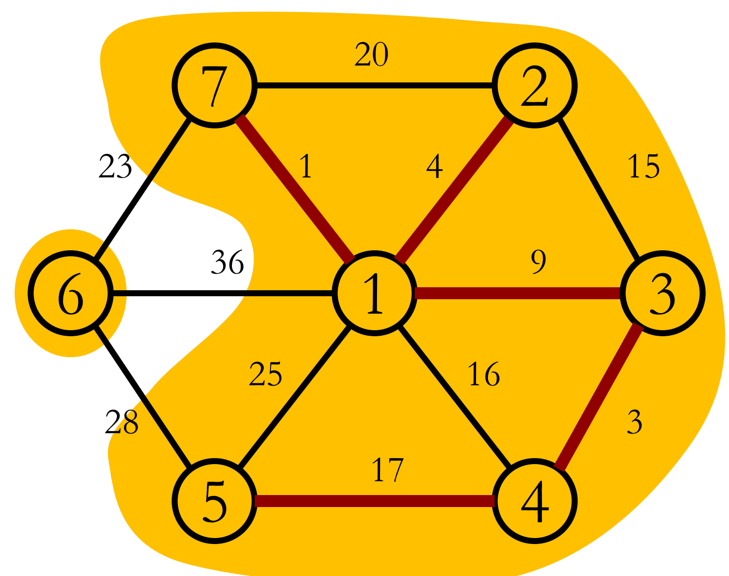


Kruskal's Algorithm

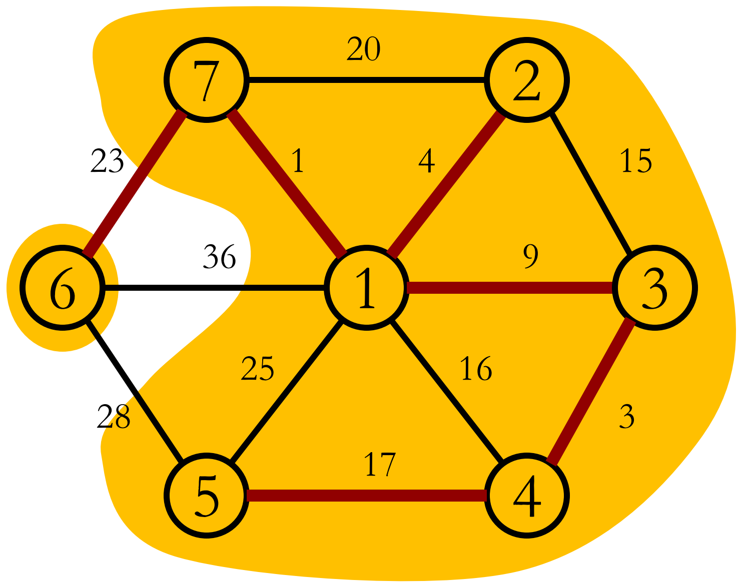


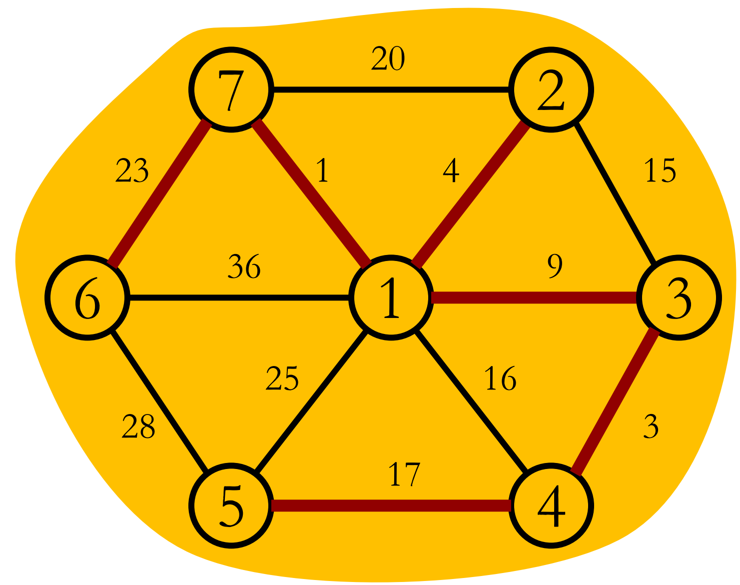


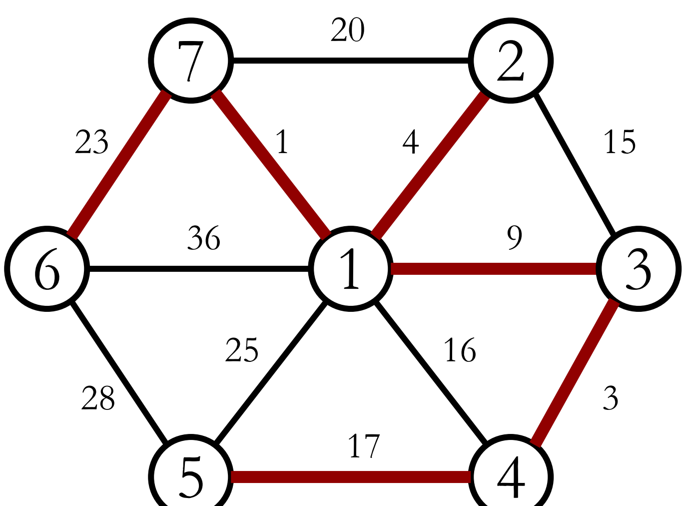


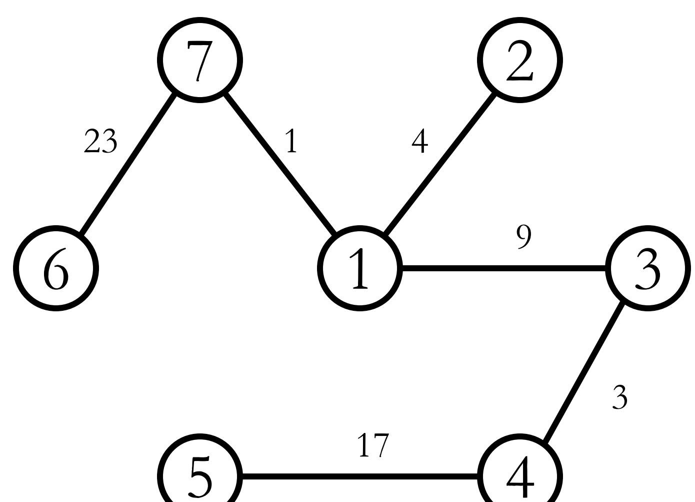


Kruskal's Algorithm









Kruskal's Algorithm

- Time Complexity:
  - Depends on the implementation of disjoint sets
  - Initialization: **O(E log E)** due to edge sorting
  - Construction:
    - Use lists to represent sets: **O(E V)**
      - Union of lists that define each set
    - **Obs:** It is possible to define a bound of **O(E log* E)** or **O(E α(E,V))** using adequate data structures

$\therefore$ Possible to define O(E log E)

$\therefore$ Given that $E < V^2$, we get O(E log V)

DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Prim's Algorithm

- History: Jarník 1930, Dijkstra 1957, Prim 1959
- Builds MST from a Root node
  - Algorithm Starts with the Root node
  - Expands Tree one Edge at a time
  - **At each step the Algorithm choose the Lightest Safe Edge**
- Using Priority Queue Q
- key[v]:
  - lowest edge weight connecting v to <u>a node </u>in the Tree
- pred[v]:
  - predecessor of v in the Tree

Prim's Algorithm

```
function MST-Prim(G,w,r)
  Q = V[G];                 // Priority queue Q
  foreach u ∈ Q  do  // Initialization
      key[u] = ∞;
  key[r] = 0;
  pred[r] = NIL;            // Keep track of tree A
  while Q ≠ ∅ do
    u = ExtractMin(Q); // Pick closest unprocessed node
    // ∃ (u,v) safe and light edge, for tree A
    foreach v ∈ Adj[u] do
      if (v ∈ Q and w(u,v) < key[v]) then  // Check if node is not in MST
        pred[v] = u;
        key[v] = w(u,v);  // Min Heap Q is updated!
```

root

weights

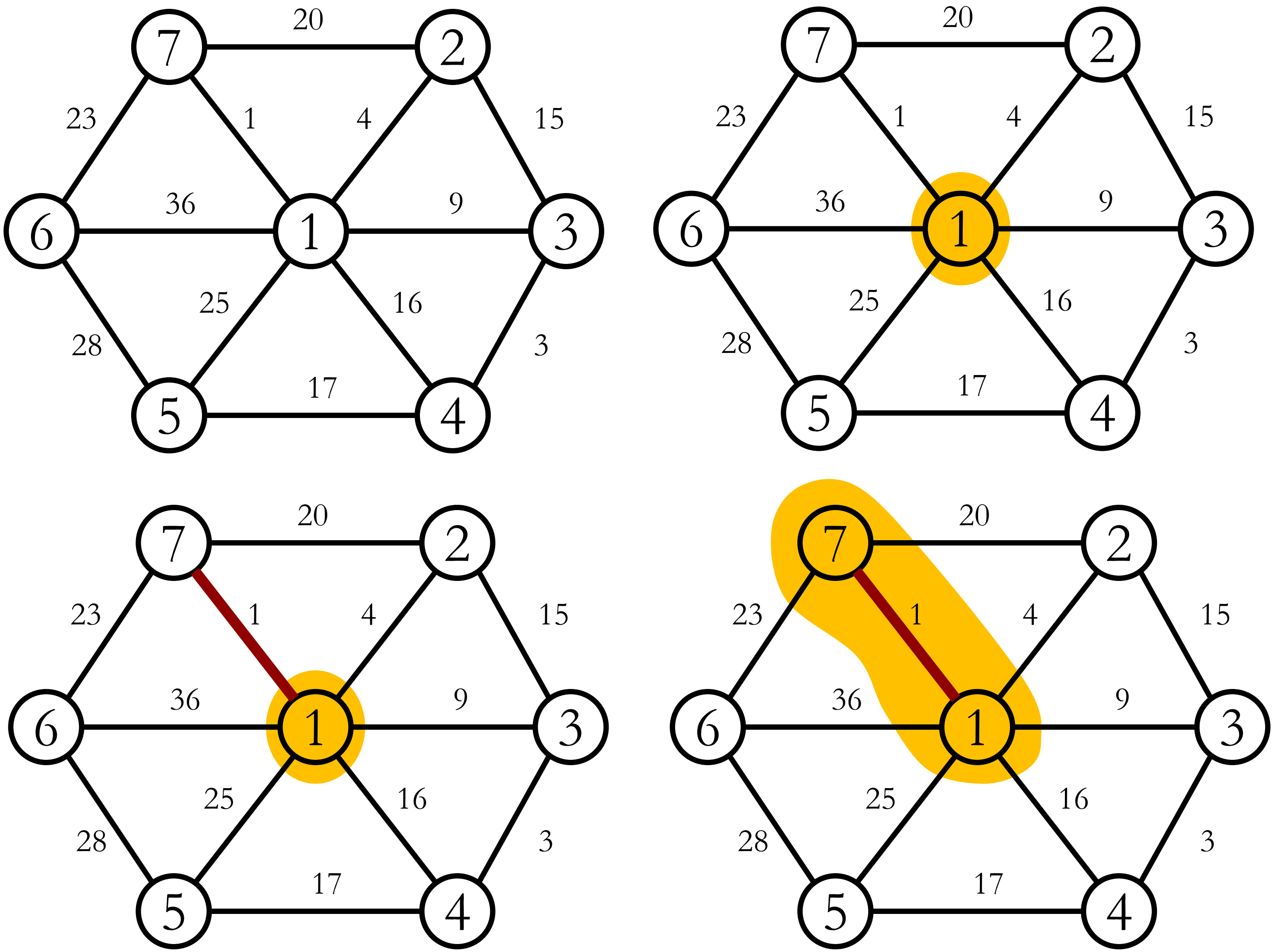
O(E)

O(V)

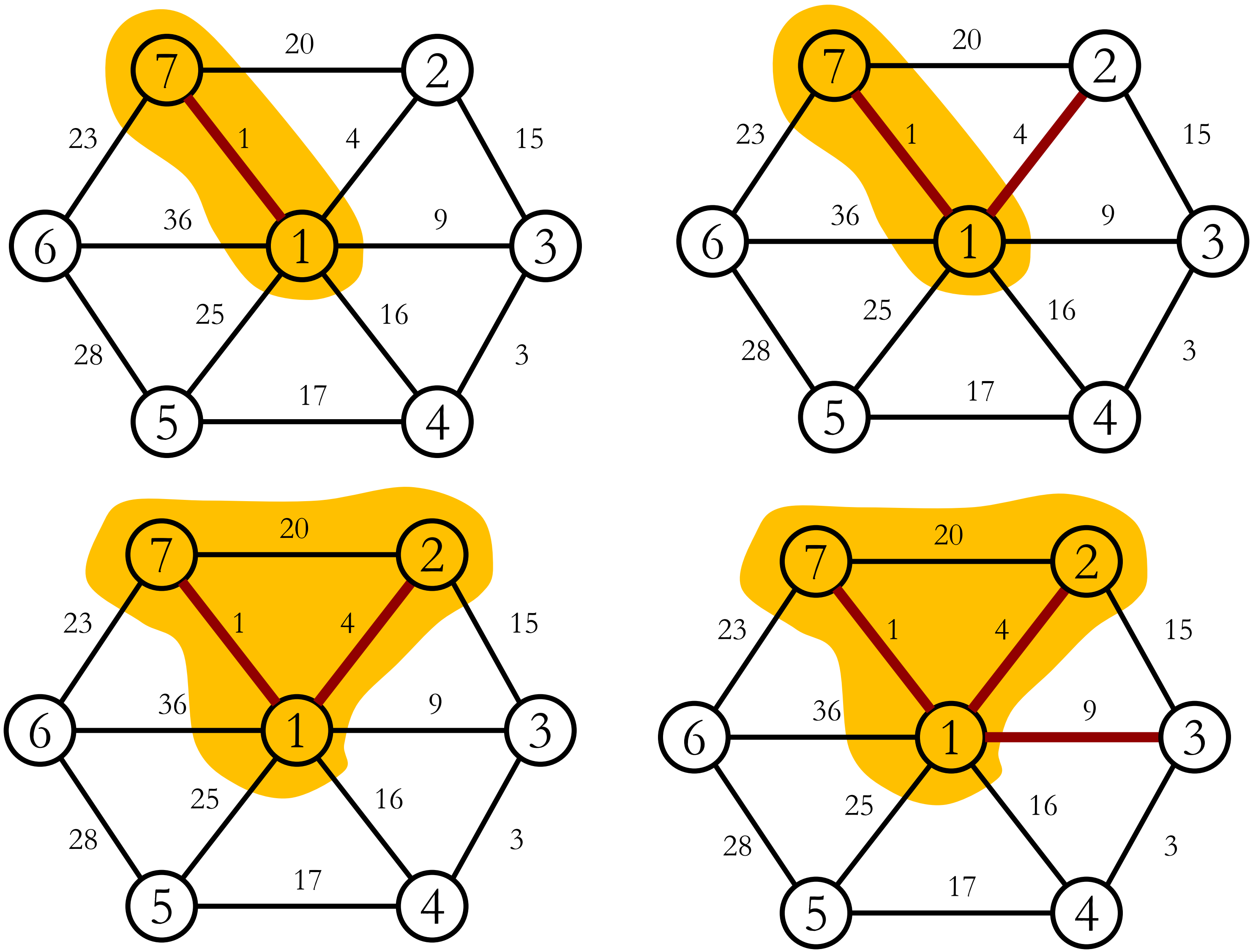
O(log V)

O(log V)

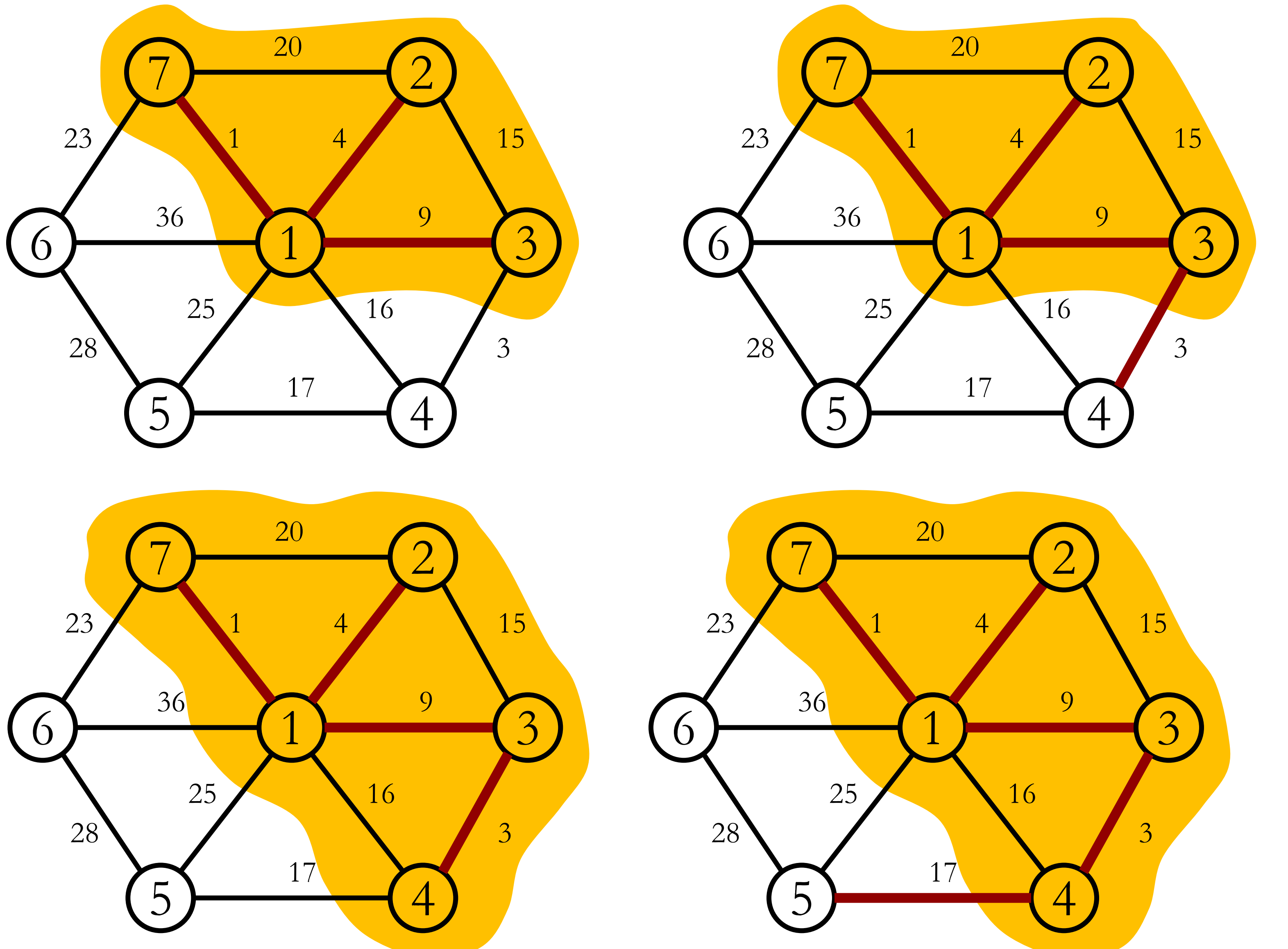
Prim's Algorithm



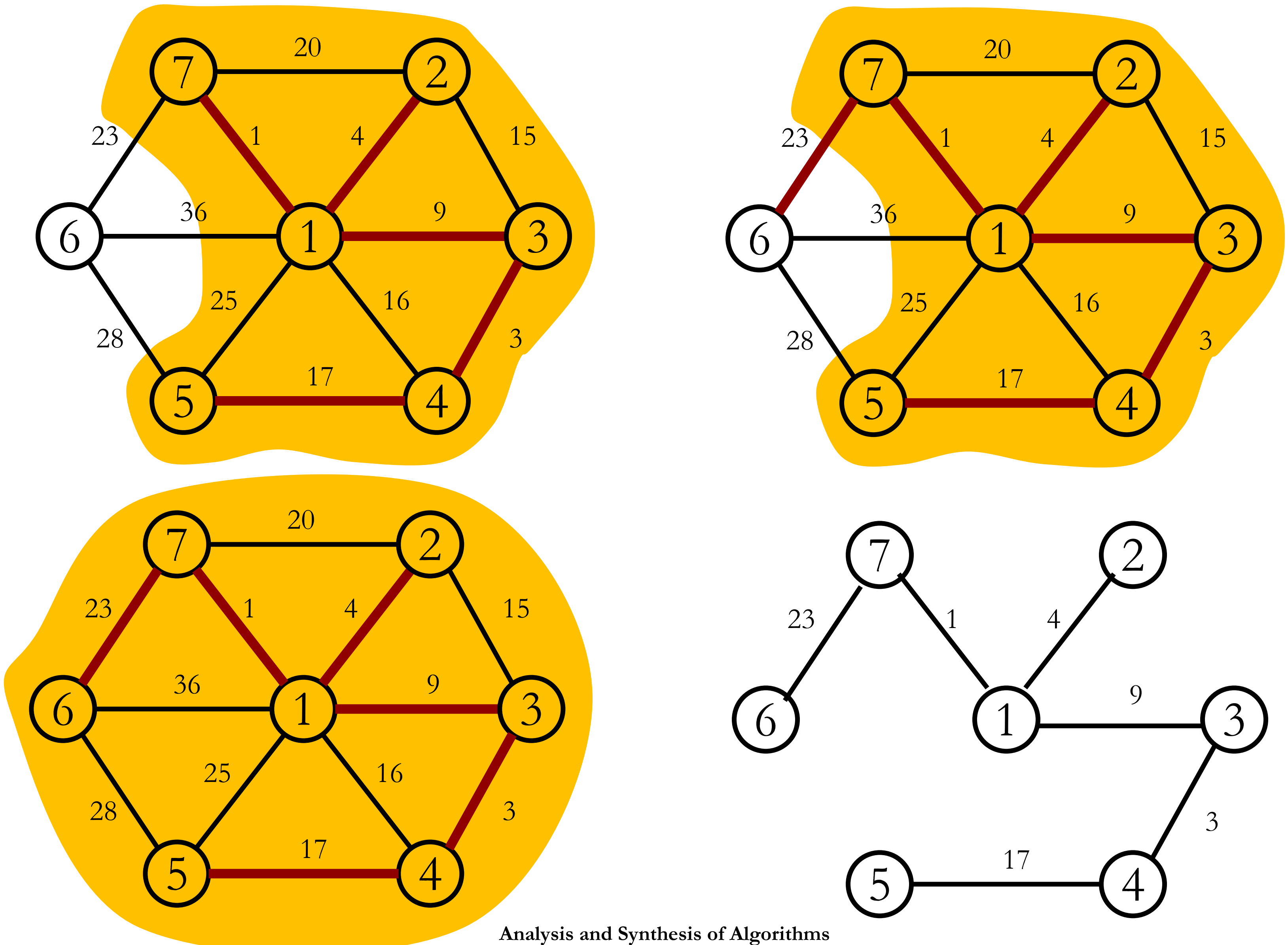
Prim’s Algorithm



Prim's Algorithm



Prim's Algorithm



Prim's Algorithm

- **Complexity:** O(E log V)
  - Priority queue using a heap
  - For each edge (*i.e.*, O(E)) exists in the worst case an update of Q with cost O(log V)

Comparison

- Although each of the above algorithms has the same worst-case running time, each one achieves this running time using different data structures and different approaches to build the MST.

There is no clear Winner among these 2 algorithms