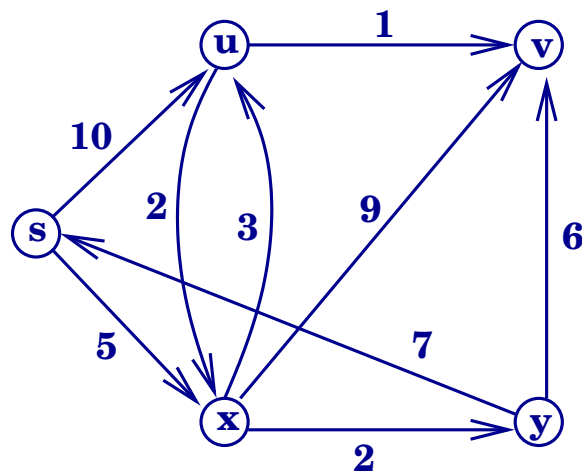


Shortest Paths

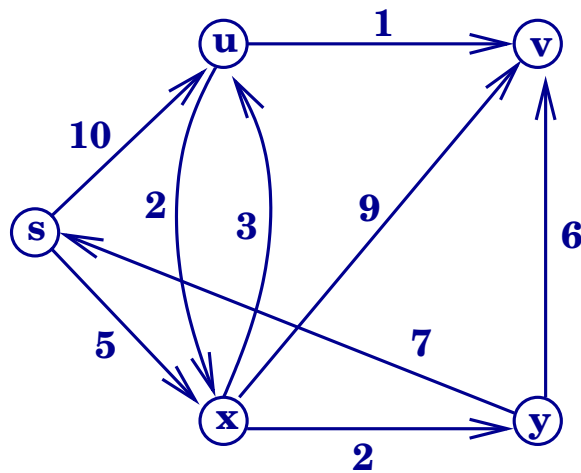
- G is a directed graph, and each edge (i, j) has a non-negative cost (or length) $c(i, j)$.
- A path $P(a, b)$ between two nodes, a and b , is a sequence of edges, starting at a , ending at b .
- The length of the path is the sum of the costs of edges on it.
- The shortest path between x and y is the path of minimum total length.



- We want to find shortest paths from node s to all other nodes.

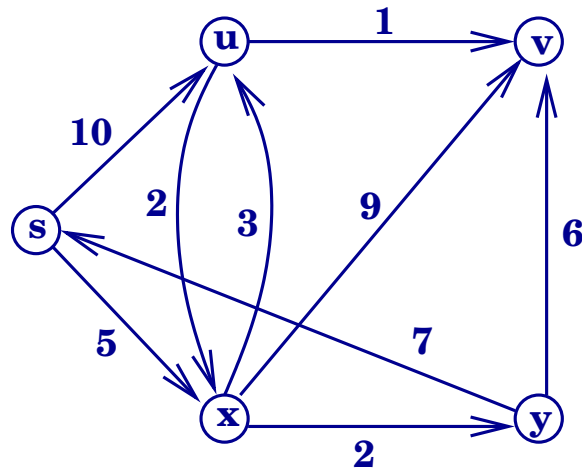
Greedy Algorithm

- Because of edge costs, there may be no relation between number of hops and total path length.
- Thus, breadth-first search by itself is not enough.
- Let us begin with the trivial path, from s to s , which has cost zero.
- What will be a good strategy to find a shortest path to another vertex?



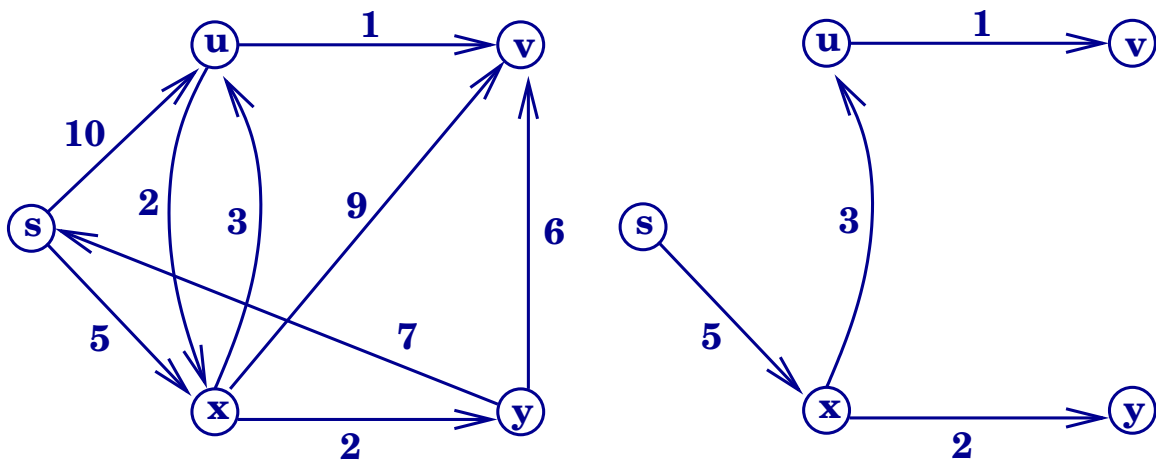
- A neighbor of s with cheapest edge from s .

Dijkstra's Algorithm



- One greedy strategy could be to always extend the current shortest path to generate the next shortest path.
- This does not work: Path $s \rightarrow x \rightarrow v$ is not the shortest path to v .
- An alternative greedy scheme is to consider all shortest paths generated so far for 1-edge extensions.
- Of all such possibilities, pick the shortest one to extend. This is Dijkstra's algorithm.

Subpath Optimality



Shortest Path Tree

- If $s \rightsquigarrow u \rightarrow v$ is a shortest path to v , then $s \rightsquigarrow u$ is a shortest path to u .
- With this property, we need not explicitly store all shortest paths.
- Instead, each node stores a pointer to its predecessor node.

High Level Description

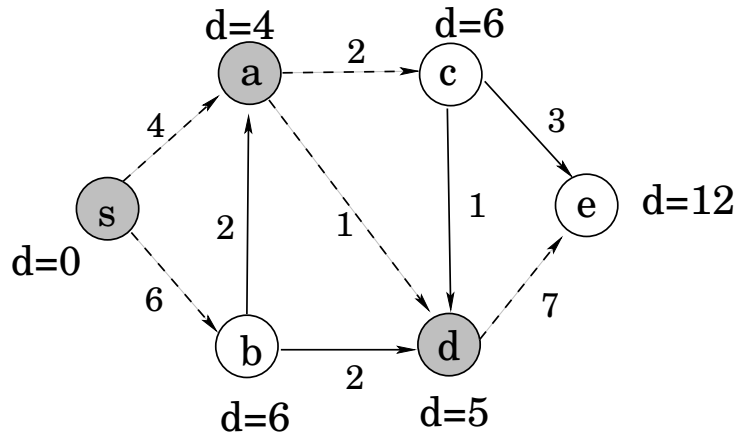
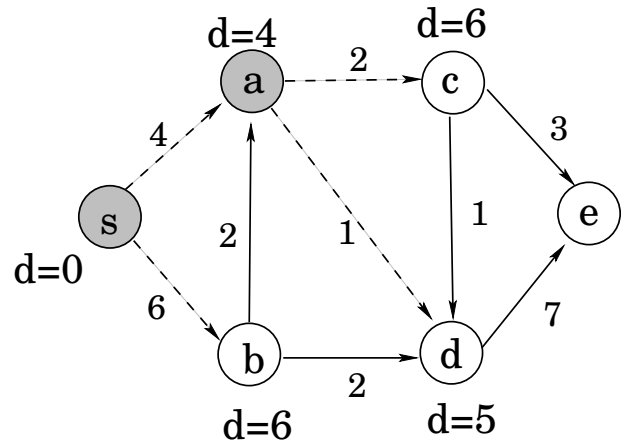
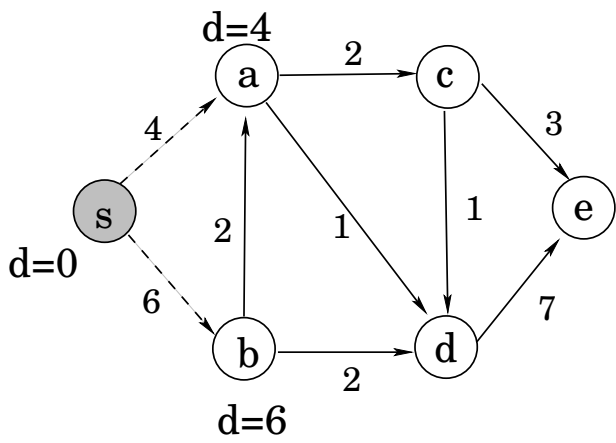
1. Initialize $d(s) = 0$, and $d(i) = \infty$, for $i \neq s$.
2. Put s into a priority queue L .
3. If L is empty, terminate; otherwise go to Step 3.
4. Delete from L the vertex i with minimum value of d . In case of ties, pick arbitrarily.
5. For each node j such that (i, j) is an edge in the graph,

$$d(j) = \min\{d(j), d(i) + c(i, j)\}$$

If $d(j)$ changes, set $p(j) = i$, and add j to L if it is not already there. Go to Step 2.

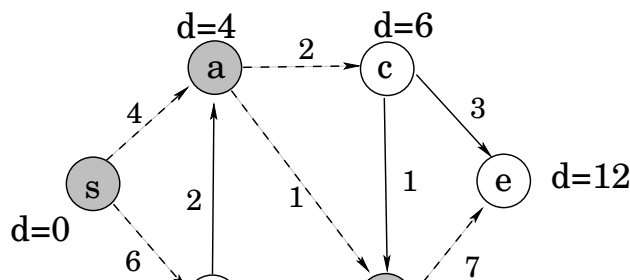
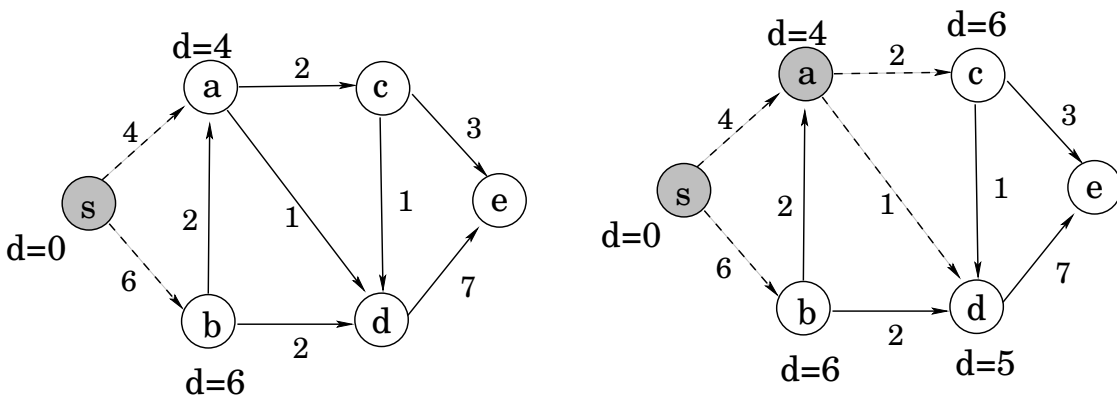
- C++ code for the algorithm in the textbook, page 645.

Illustration



Analysis

- We can store G as an adjacency matrix: $A[i, j]$ stores the information about edge (i, j) . We can store L as an unordered list.
- Choosing i with smallest d takes $O(n)$ time.
- Updating $d(j)$ for each neighbor of i takes $O(1)$ time, and there are at most n neighbors.
- Each iteration of the loop takes $O(n)$ time and it deletes one vertex from L .
- Thus, total time complexity of Dijkstra's algorithm using unordered list L and adjacency matrix is $O(n^2)$.



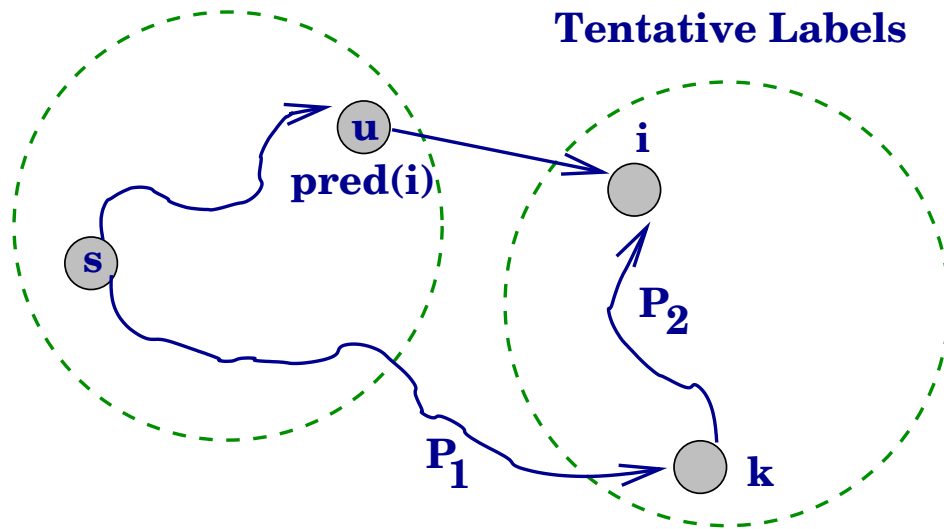
Analysis

- We improve the running time by storing L in a heap, and using the adjacency list representation of the graph.
- Choosing i with smallest d takes $O(\log n)$ time.
- Updating $d(j)$ takes $O(1)$ time, but when d changes, the heap needs to propagate the change, which takes $O(\log n)$ time.
- While a node i can have up to n neighbors, the total number of neighbors is $|E|$, the number of edges in G .
- Thus, the complexity of the steps 3–4 is $O(|E| \log n)$.

Correctness Proof

- Think of $d(i)$ as tentative distance label.
- Dijkstra's algorithm makes the distance label of one node permanent in each iteration.
- We argue that when $d(i)$ is made permanent (deleted from L), it equals the shortest path distance.

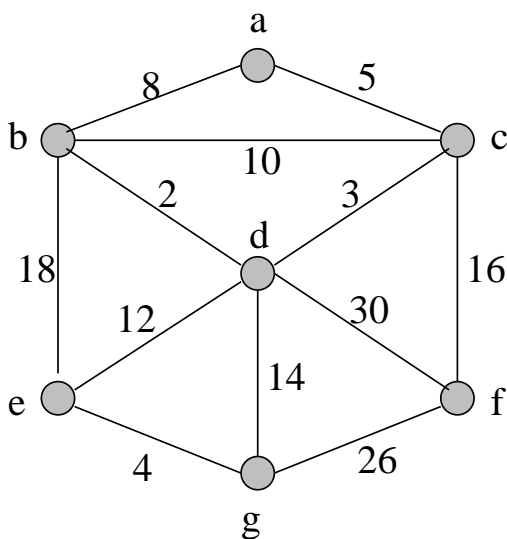
Permanent Labels



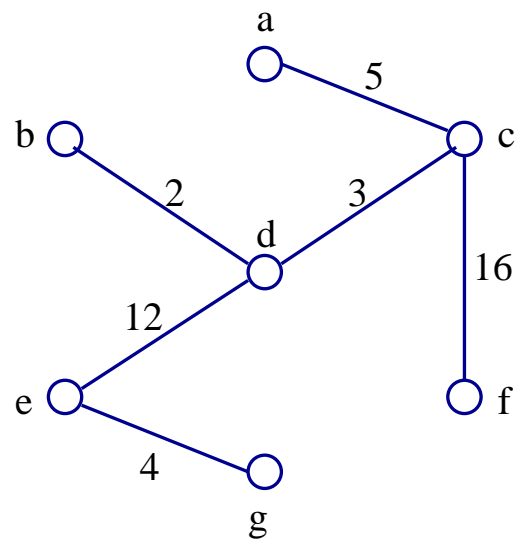
- By hypothesis, P_1 is longer than $d(u) + c(u, i)$.
- Since P_2 has positive length, no alternative path to i via k can be shorter.

Minimum Spanning Trees

- $G = (V, E)$ is an undirected graph; each edge (i, j) has a non-negative cost $c(i, j)$.
- A spanning tree $T = (V, F)$ connects all vertices of V using fewest possible edges.
- A minimum spanning tree is a spanning tree with least possible total cost.



Graph G

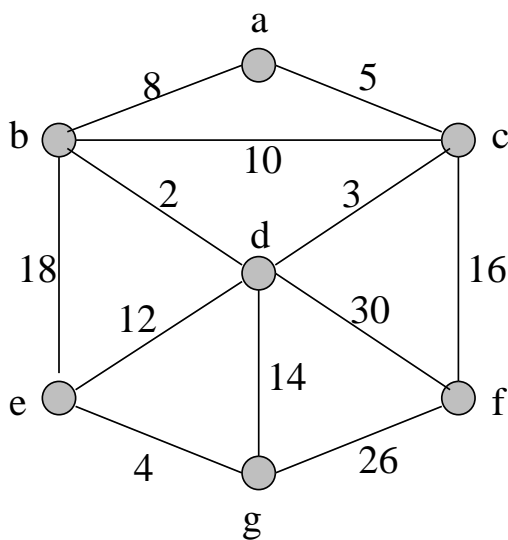


MST

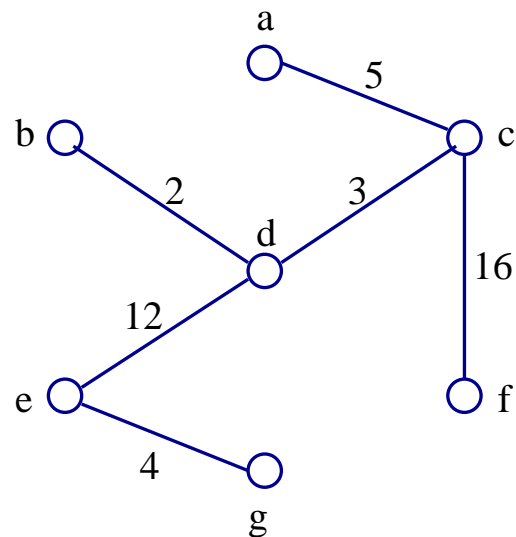
- All spanning trees on n nodes have $n - 1$ edges. The problem is to choose the cheapest collection that spans the nodes.

Kruskals' Algorithm

- Sort the edges in non-decreasing order of cost: $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$.
- Set T to be the empty tree.
- For $i = 1$ to m , add edge e_i to T if it does not create a cycle.
- Output T as the MST.



Graph G



MST

- Sorted order:
2, 3, 4, 5, 8, 10, 12, 14, 16, 18, 26, 30.

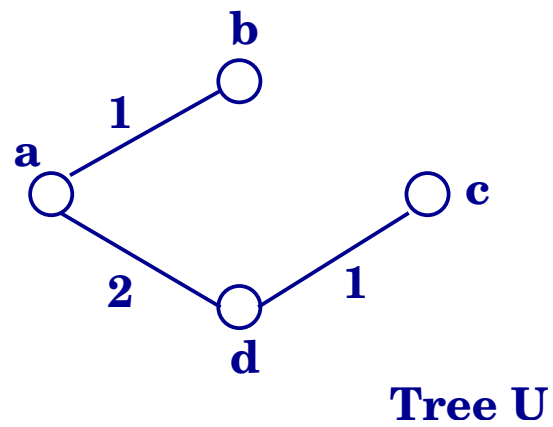
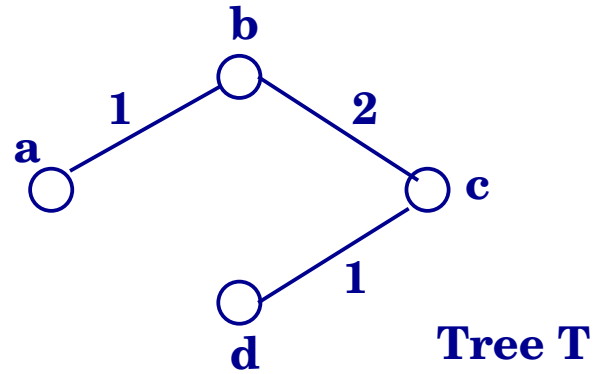
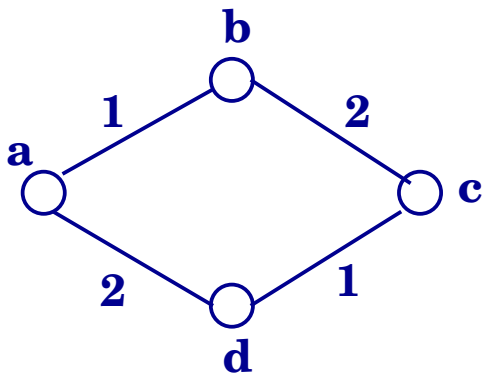
Correctness Proof

- An undirected graph G has a spanning tree if and only if G itself is connected.
- The only edges rejected by Kruskal's algorithm are those that form a cycle with previously chosen edges.
- Removing any edge from a cycle leaves the remaining subgraph connected.
- So, if G is connected, Kruskal's algorithm indeed produces a spanning tree.
- In order to argue that it produces a **minimum** spanning tree, we use the by now standard swapping argument.
- Many edges of G can have the same cost, so MST need not be unique. We show that no spanning tree can have lower cost than output of Kruskal.

Correctness of Kruskal

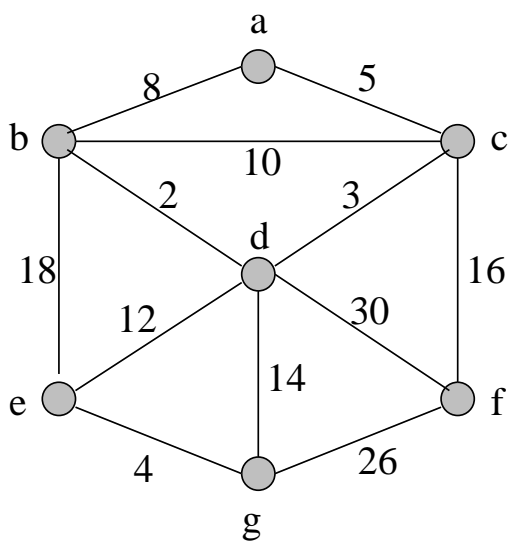
- Suppose T is the output of Kruskal's algorithm. Let U be another MST claimed to have smaller cost than T .
- We transform U into T without increasing its cost, thereby refuting the claim.
- Let e be the cheapest edge of T that is not in U .
- Adding e to U creates a unique cycle C .
- Let f be any edge of cycle C that is not in T ; such an f must exist.
- Since Kruskal's algorithm scans edges in sorted order, and it chose e but rejected f , we must have $c(e) \leq c(f)$.
- We remove f from U and add e instead. This keeps U connected, and does not increase its cost.
- Repeat until $T = U$.

Correctness of Kruskal

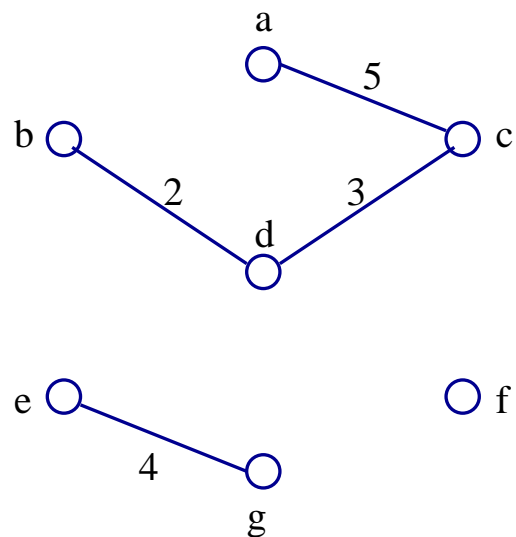


Data Structures

- Initial sorting of edges takes $O(e \log e)$ time, where $e = |E|$.
- The non-obvious operation is to detect whether an edge (u, v) forms a cycle with the previously accepted edges.



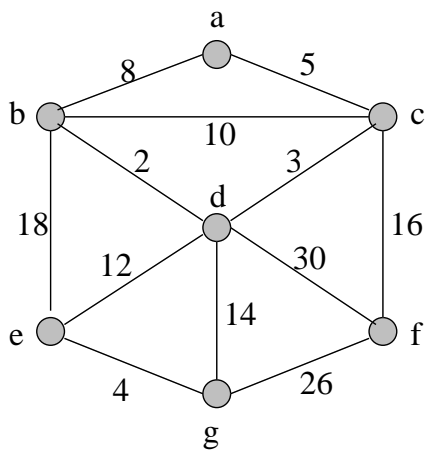
Graph G



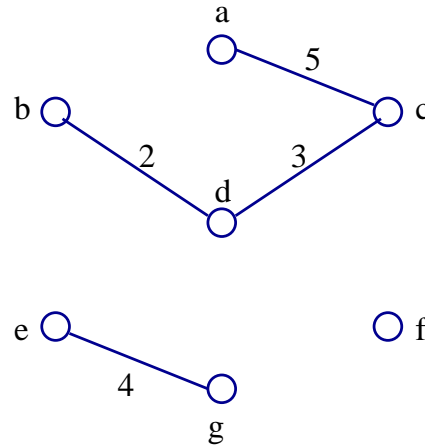
**Collection of trees
after scanning 4 edges.**

Data Structures

- Initially, forest has n singleton trees.
- After i edges have been added, there are $n - i$ components.



Graph G



**Collection of trees
after scanning 4 edges.**

- When considering edge (u, v) , we can perform the **reachability test** (DFS, BFS) to see if v is reachable from u in the current forest.
- This could take $O(n)$ time per test, and will lead to $O(ne)$ time for the overall algorithm.
- We can do the cycle test faster using the **Union-Find** data structure.

Union Find

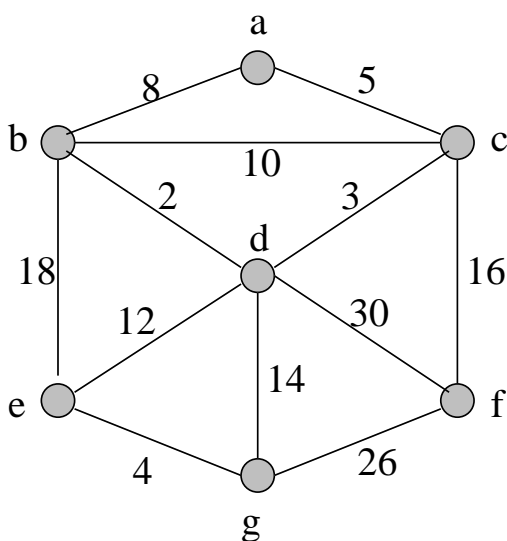
- Each component as a set, with one vertex acting as “representative”.
- The operation $Find(x)$ returns the name of the set containing x .
- If $Find(u) = Find(v)$, then u, v are in the same tree \Rightarrow edge (u, v) forms a cycle.
- Otherwise, we merge the sets containing u and v . (The **union** operation.) This requires renaming all elements of at least one set.
- Store sets as rooted trees, and using the Union-by-Rank heuristic we can achieve $O(\log n)$ cost for both Find and (amortized) Union.
- With this data structure, Kruskal’s algorithm runs in $O(e \log e)$ worst-case time.

Improved Union Find

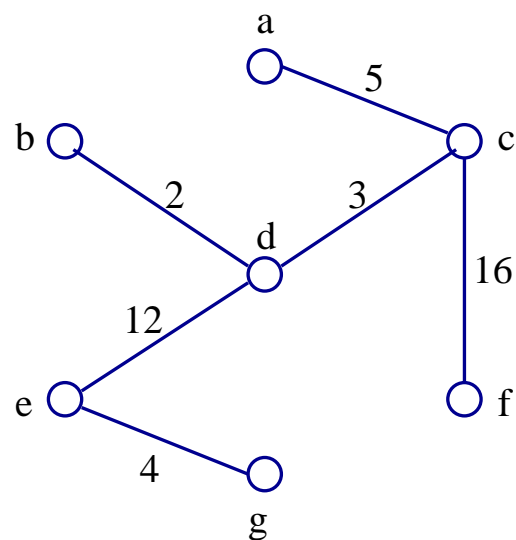
- There is an improved version of Union-Find data structure.
- Besides union-by-rank, it uses path compression.
- Suppose we perform a sequence of m operations, of which at most n are make-set; others are unions and finds.
- Total time complexity is $O(m\alpha(m, n))$, where $\alpha(m, n)$ is extremely slow growing function, called Inverse Ackermann function.

Prim' Algorithm

- Prim's algorithm grows a single tree T , one edge at a time, until it becomes a spanning tree.
- We initialize T to be a singleton node, and no edges.
- At each step, Prim's algorithm adds the cheapest edge with one endpoint in T and other not in T .
- Since each edge adds one new vertex to T , after $n - 1$ additions, T becomes a spanning tree.



Graph G



MST

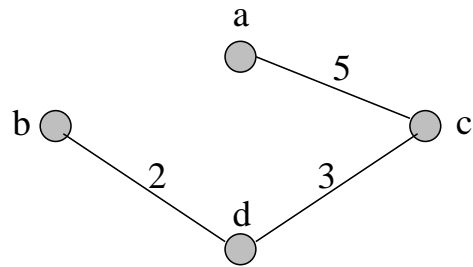
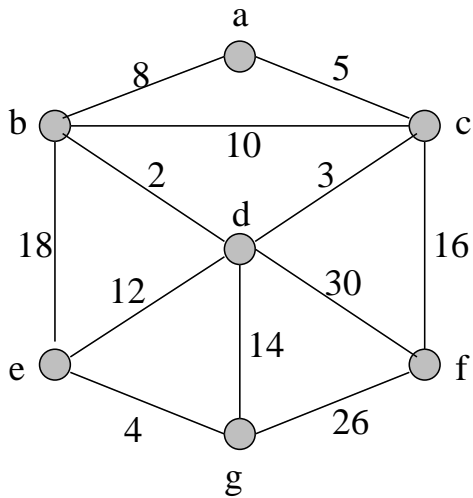
Correctness Proof

- Suppose there is a MST U that is claimed to be cheaper than T .
- We use contradiction. Suppose among all tree cheaper than T , U differs from T in least number of edges.
- Let e be the first edge added to T that is not in U .
- Just before e was added, let X be the set of nodes connected by T , and let $Y = V - X$ be the remaining nodes.
- Since U spans all the vertices, it contains at least one edge, f , with one endpoint in each of X and Y .
- By the choice of e , we have $c(e) \leq c(f)$.
- We add e to U and remove f from it. This does not increase the cost of U , but now U differs from T in one fewer edge.
- This contradicts the choice of U . So, T is optimal.

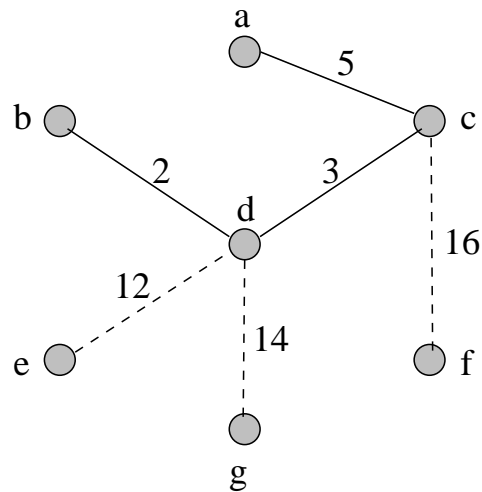
Data Structures

- Use adjacency list representation of graph G .
- Vertices not in T are stored in a heap, where the $key(v)$ is the cost of the cheapest edge from v to some node in T .
- Initially we put one node s in T , and make $key()$ of all neighbors of s finite. All other vertices have infinite keys.
- Do a DeleteMin to find the cheapest edge. If vertex v is the node connected by the cheapest edge, we add v to T .
- We then scan all edges incident to v , and update the keys of their other endpoints, if necessary.

Illustration



After 3 steps



Blue edges and keys after 3 steps

Time Complexity

- DeleteMin operations takes $O(\log n)$ time, and there are at most $n - 1$ such operations.
- When a node v is pulled into T , we need to scan all neighbors of v , and potentially update their keys.
- There can be at most e such updates, and each update takes $O(\log n)$ time.
- Using a binary heap, Prim's algorithm can be implemented in $O(e \log n)$ time.