Chapter 3: Greedy Algorithm Design Technique - II

Devesh C Jinwala, IIT Jammu, India

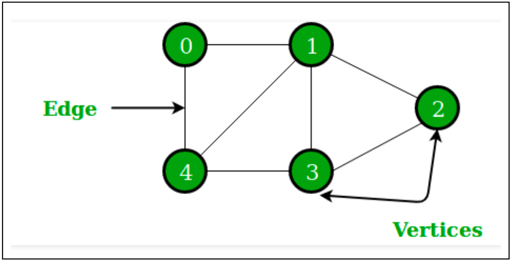
November 27, 2020

Design and Analysis of Algorithms
IIT Jammu, Jammu

Revisiting a Graph

Definition

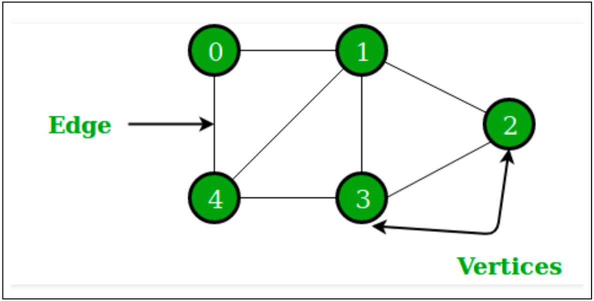
- informally, a set of vertices (nodes) and edges connecting them.
- a graph $G = (V, E)$ where, V is a set of vertices i.e. $V = v_i$ with an edge $e = v_i, v_j$ connecting two vertices with each e ϵ to a set of all such edges between two vertices of the graph viz. $E = (v_i, v_j)$. An example graph G=(V,E) shown here



Graph attributes

A Path

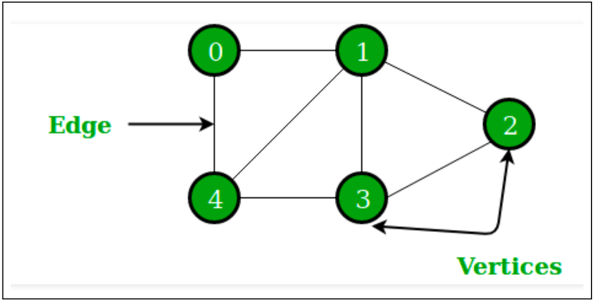
- A Path p of length k, is a sequence of connected vertices $p = \langle v_0, v_1, \ldots\ldots, v_k \rangle$ where $(v_i, v_{i+1}) \epsilon E$



Graph attributes

A Path

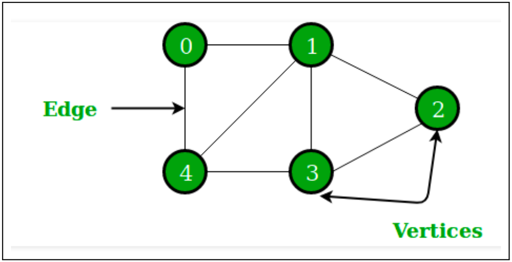
- A Path p of length k, is a sequence of connected vertices p $= < v_0, v_1, ..., v_k >$ where $(v_i, v_{i+1}) \epsilon E$
- in the figure given here various paths are :



Graph attributes : Path, Path-length

A Path and path-length

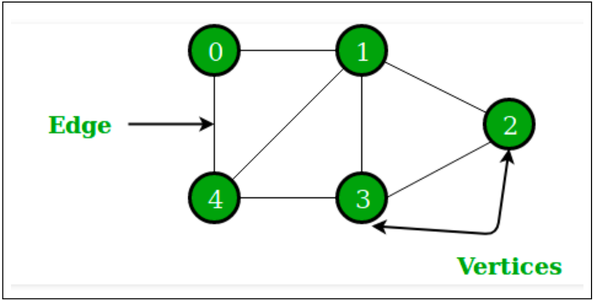
- Path-length is the number of vertices in a graph e.g. the path lengths in this graph are:



Graph attributes: Cycle

A Cycle

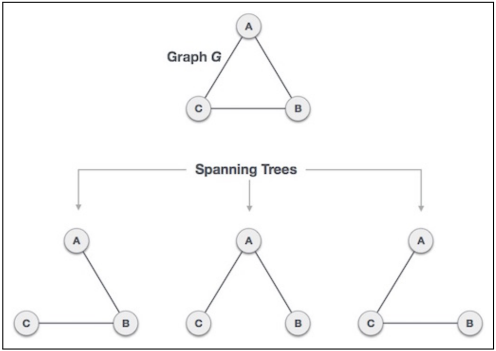
- Cycle: A graph contains no cycles if there is no path $p = < v_0, v_1, ..., v_k >$ such that $v_0 = v_k$ e.g. in the graph shown one of the cycles is :



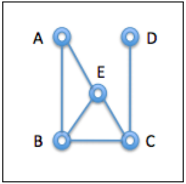
Graph attributes : A Spanning Tree

A Spanning Tree

- A Spanning Tree is a set of $|V|$ - 1 edges that connect all the vertices of a graph.



Graph attributes : A Spanning Tree...



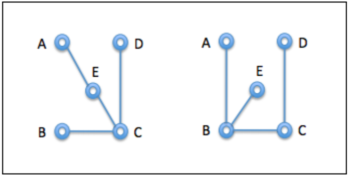


Figure: Another illustration of Spanning Tree

Graph attributes : A Spanning Tree...

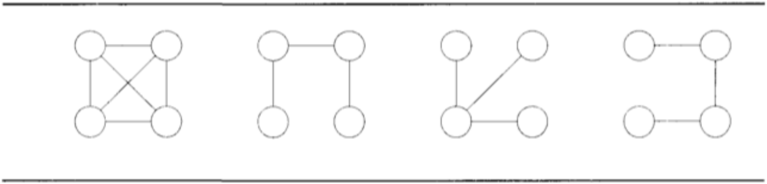
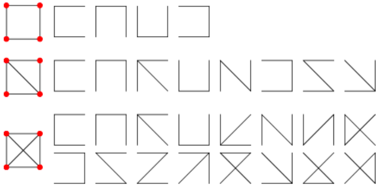


Figure 4.5 An undirected graph and three of its spanning trees

Figure: Another illustration of Spanning Tree

Graph attributes : A Spanning Tree...



A spanning tree of a graph on n vertices is a subset of $n-1$ edges that form a tree (Skiena 1990, p. 227). For example, the spanning trees of the cycle graph C_4, diamond graph, and complete graph K_4 are illustrated above.

Figure: Another illustration of Spanning Tree

Properties of a Spanning Tree

Properties of a Spanning Tree

- A connected graph G can have more than one spanning tree.

Properties of a Spanning Tree

Properties of a Spanning Tree

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.

Properties of a Spanning Tree

Properties of a Spanning Tree

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).

Properties of a Spanning Tree

Properties of a Spanning Tree

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.

Properties of a Spanning Tree

Properties of a Spanning Tree

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.

Properties of a Spanning Tree

Properties of a Spanning Tree

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.
- Spanning tree has n-1 edges, where n is the number of nodes (vertices).

Properties of a Spanning Tree

Properties of a Spanning Tree

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.
- Spanning tree has n-1 edges, where n is the number of nodes (vertices).
- From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree.

Properties of a Spanning Tree

Properties of a Spanning Tree

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.
- Spanning tree has n-1 edges, where n is the number of nodes (vertices).
- From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree.
- A complete graph can have maximum n^{n-2} number of spanning trees.

Properties of a Spanning Tree

Properties of a Spanning Tree

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.
- Spanning tree has n-1 edges, where n is the number of nodes (vertices).
- From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree.
- A complete graph can have maximum n^{n-2} number of spanning trees.
- Spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Sparse and Dense Graphs

Dense Graphs

- Dense graphs have a lot of edges compared to the number of vertices.

By comparison, a tree is sparse because $|E| = n - 1 = O(n)$.

Sparse and Dense Graphs

Dense Graphs

- Dense graphs have a lot of edges compared to the number of vertices.
- With $n = |V|$ for the number of vertices, how many edges maximally a dense graph can have ?

By comparison, a tree is sparse because $|E| = n - 1 = O(n)$.

Sparse and Dense Graphs

Dense Graphs

- Dense graphs have a lot of edges compared to the number of vertices.
- With $n = |V|$ for the number of vertices, how many edges maximally a dense graph can have ?
- That is $|E| = n^2$

By comparison, a tree is sparse because $|E| = n - 1 = O(n)$.

Constructing of a Spanning Tree

Algorithm1: Node-centric ST Algorithm

- Pick an arbitrary node and mark it as being in the tree.

- We iterate $(n1)$ times in Step 2, because there are $(n1)$ vertices that have to be added to the tree.
- The efficiency of the algorithm is determined by how efficiently we can find a qualifying w.

Constructing of a Spanning Tree

Algorithm1: Node-centric ST Algorithm

- Pick an arbitrary node and mark it as being in the tree.
- Repeat until all nodes are marked as in the tree:

- We iterate $(n1)$ times in Step 2, because there are $(n1)$ vertices that have to be added to the tree.
- The efficiency of the algorithm is determined by how efficiently we can find a qualifying w.

Constructing of a Spanning Tree

Algorithm1: Node-centric ST Algorithm

- Pick an arbitrary node and mark it as being in the tree.
- Repeat until all nodes are marked as in the tree:
  - Pick an arbitrary node u in the tree with an edge e to a node w not in the tree.

- We iterate $(n1)$ times in Step 2, because there are $(n1)$ vertices that have to be added to the tree.
- The efficiency of the algorithm is determined by how efficiently we can find a qualifying w.

Constructing of a Spanning Tree

Algorithm1: Node-centric ST Algorithm

- Pick an arbitrary node and mark it as being in the tree.
- Repeat until all nodes are marked as in the tree:
  - Pick an arbitrary node u in the tree with an edge e to a node w not in the tree.
  - Add e to the spanning tree and mark w as in the tree.

- We iterate $(n1)$ times in Step 2, because there are $(n1)$ vertices that have to be added to the tree.
- The efficiency of the algorithm is determined by how efficiently we can find a qualifying w.

Constructing of a Spanning Tree

Algorithm2: Edge-centric ST Algorithm

1. Start with the collection of singleton trees, each with exactly one node.

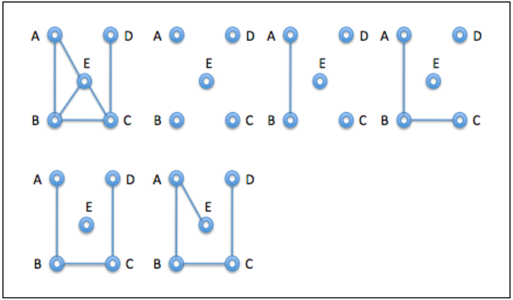


Figure: Another illustration of Spanning Tree

Constructing of a Spanning Tree

Algorithm2: Edge-centric ST Algorithm

1. Start with the collection of singleton trees, each with exactly one node.
2. As long as we have more than one tree, connect two trees together with an edge in the graph.

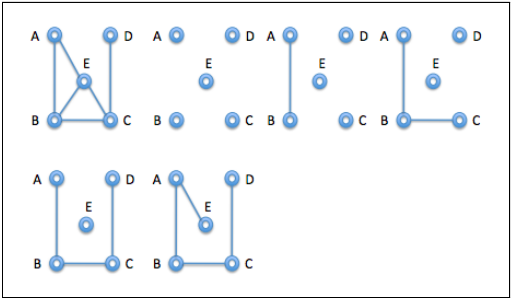


Figure: Another illustration of Spanning Tree

Minimum Spanning Tree of a Graph

Definition

If a cost c_{ij} is associated with an edge $e_{ij} = (v_i, v_j)$ then the minimum spanning tree is the set of edges E_{span} such that C=$\sum_{for\ all\ e_{ij}\ \epsilon\ E_{span}} c_{ij}$ is a minimum

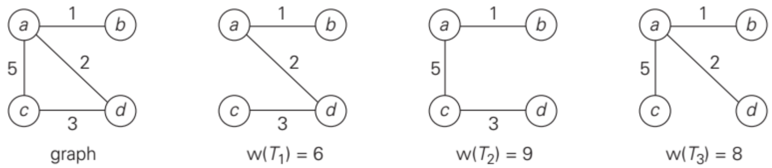


FIGURE 9.2 Graph and its spanning trees, with T_1 being the minimum spanning tree.

Figure: Another illustration of Spanning Tree

Generic Approach for computing Minimum Spanning Tree

- We shall consider two algorithms viz. Kruskal and Prim's algorithm for computing the MST.

Generic Approach for computing Minimum Spanning Tree

- We shall consider two algorithms viz. Kruskal and Prim's algorithm for computing the MST.
- Both are based on a generic approach to compute the MST. This is based on the following loop invariant viz. *Prior to each iteration, A is a subset of some minimum spanning tree.*

Generic Approach for computing Minimum Spanning Tree

- We shall consider two algorithms viz. Kruskal and Prim's algorithm for computing the MST.
- Both are based on a generic approach to compute the MST. This is based on the following loop invariant viz. *Prior to each iteration, A is a subset of some minimum spanning tree.*
- The generic method manages a set of edges A, maintaining this loop invariant and does the following:

Generic Approach for computing Minimum Spanning Tree

- We shall consider two algorithms viz. Kruskal and Prim's algorithm for computing the MST.
- Both are based on a generic approach to compute the MST. This is based on the following loop invariant viz. *Prior to each iteration, A is a subset of some minimum spanning tree.*
- The generic method manages a set of edges A, maintaining this loop invariant and does the following:

Generic Approach

At each step, we determine an edge (u, v) that we can add to the set A without violating the loop invariant, in the sense that A U {(u,v)} is also a subset of a minimum spanning tree.

Generic Approach for computing Minimum Spanning Tree

- We shall consider two algorithms viz. Kruskal and Prim's algorithm for computing the MST.
- Both are based on a generic approach to compute the MST. This is based on the following loop invariant viz. *Prior to each iteration, A is a subset of some minimum spanning tree.*
- The generic method manages a set of edges A, maintaining this loop invariant and does the following:

Generic Approach

At each step, we determine an edge (u, v) that we can add to the set A without violating the loop invariant, in the sense that A U {(u,v)} is also a subset of a minimum spanning tree.

- The approach uses the concepts of safe edges, light edges and a cut of a graph

Generic Approach for computing Minimum Spanning Tree

- We shall consider two algorithms viz. Kruskal and Prim's algorithm for computing the MST.
- Both are based on a generic approach to compute the MST. This is based on the following loop invariant viz. *Prior to each iteration, A is a subset of some minimum spanning tree.*
- The generic method manages a set of edges A, maintaining this loop invariant and does the following:

Generic Approach

At each step, we determine an edge (u, v) that we can add to the set A without violating the loop invariant, in the sense that A U {(u,v)} is also a subset of a minimum spanning tree.

- The approach uses the concepts of safe edges, light edges and a cut of a graph
- Let us investigate these concepts.

Safe edge and Generic MST approach

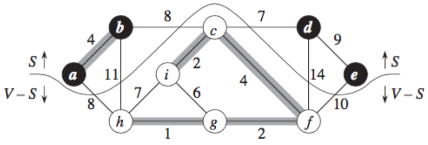
Safe edge

Given the loop invariant that *Prior to each iteration, A is a subset of some minimum spanning tree* at each step, we determine an edge (u, v) that can be safely added to the set A - we call such an edge a safe edge for A, since we can add it safely to A while maintaining the invariant.

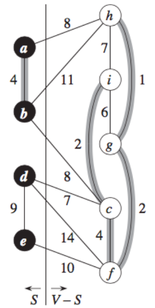
```
Algorithm GENERIC-MST(G,w)
A=Φ
while A does not form the spanning tree
    find an edge (u,v) that is safe for A
    A = A U {(u,v)}
return A
```

Definitions

Cut of a graph : A cut (S, V S) of an undirected graph G = (V, E) is a partition of the nodes of the graph V in such a way that an edge (u,v) ϵ E crosses the cut (S, V-S) if one of its endpoints is in S and the other endpoint is in V-S. We say, a cut respects a set A of edges if no edge in A crosses the cut.



(a)

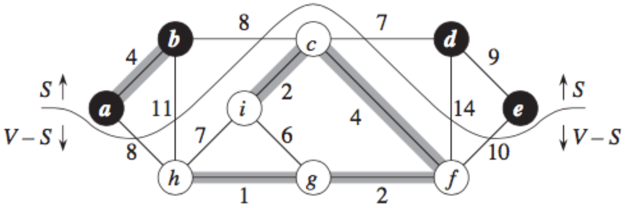


(b)

Theorem: Determining Safe edge

Theorem

Let G=(V,E) be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let (S, V-S) be any cut of G that respects A, and let (u,v) be a light edge crossing (S, V-S). Then, edge (u,v) is safe for A.



(a)

How to find a safe edge ?

Determining a safe edge

- Calculate the minimum spanning tree
- Put all the vertices into single node trees by themselves
- Put all the edges in a priority queue
- Repeat until we have constructed a spanning tree
- Extract the cheapest edge
- If it forms a cycle, ignore it, else add it to the forest of trees - (it will join two trees into a larger tree)
- Return the spanning tree

Pseudocode: How to find a safe edge ?

```
Algorithm MinimumSpanningTree(Graph G(V,E),
                                          Weight_function w)
1. for each vertex v ϵ V[G]
2.      Construct a single-vertex forest from G
3. Sort the edges of E into
                 nondecreasing order by weight w ϵ E
4. for each edge (u,v)
5.               do extract the cheapest edge
6.                         add it to A
7.               while (it does not form a Cycle)
8.      return A;
```

How to find detect a cycle ?

Cycle detection

- Uses a Union-find structure
- Union-Find data structure is based on partitioning of a set
- How do we view Partition formally ?
- Parition is a set of subsets of elements of a set where
  - every element belongs to one of the sub-sets
  - no element belongs to more than one sub-set
  - that is, given that Set, S = s_i ,
  - Partition(S)= P_i , where $P_i = s_i$
  - $\forall\ s_i \in S_i,\ s_i \in P_i$
  - $\forall\ j, k,\ P_j \cap P_k = \phi$
  - $S = \cup P_j$

How to find detect a cycle ?...

Equivalence Class

- The sets of a partition are equivalence classes

How to find detect a cycle ?...

Equivalence Class

- The sets of a partition are equivalence classes
- The elements of each set of a partition are related by an equivalence relation

How to find detect a cycle ?...

Equivalence Class

- The sets of a partition are equivalence classes
- The elements of each set of a partition are related by an equivalence relation
- Equivalence relations are

How to find detect a cycle ?...

Equivalence Class

- The sets of a partition are equivalence classes
- The elements of each set of a partition are related by an equivalence relation
- Equivalence relations are
    - reflexive i.e. if $x \sim x$

How to find detect a cycle ?...

Equivalence Class

- The sets of a partition are equivalence classes
- The elements of each set of a partition are related by an equivalence relation
- Equivalence relations are
  - reflexive i.e. if $x \sim x$
  - transitive i.e. if $x \sim y$ and $y \sim z \implies x \sim z$

How to find detect a cycle ?...

Equivalence Class

- The sets of a partition are equivalence classes
- The elements of each set of a partition are related by an equivalence relation
- Equivalence relations are
  - reflexive i.e. if $x \sim x$
  - transitive i.e. if $x \sim y$ and $y \sim z \implies x \sim z$
  - symmetric i.e. if $x \sim y$ then $y \sim x$

How to find detect a cycle ?...

Equivalence Class

- The sets of a partition are equivalence classes
- The elements of each set of a partition are related by an equivalence relation
- Equivalence relations are
  - reflexive i.e. if $x \sim x$
  - transitive i.e. if $x \sim y$ and $y \sim z \implies x \sim z$
  - symmetric i.e. if $x \sim y$ then $y \sim x$
- Each element of the set is related to every other element

Applying Partitions and Union Find in MST

In a typical MST algorithm

- The sets of a partition are equivalence classes

Applying Partitions and Union Find in MST

In a typical MST algorithm

- The sets of a partition are equivalence classes
- The connected vertices form equivalence classes

Applying Partitions and Union Find in MST

In a typical MST algorithm

- The sets of a partition are equivalence classes
- The connected vertices form equivalence classes
- *Being connected* is the equivalence relation

Applying Partitions and Union Find in MST

In a typical MST algorithm

- The sets of a partition are equivalence classes
- The connected vertices form equivalence classes
- *Being connected* is the equivalence relation
- Initially, each vertex is in a class by itself

Applying Partitions and Union Find in MST

In a typical MST algorithm

- The sets of a partition are equivalence classes
- The connected vertices form equivalence classes
- *Being connected* is the equivalence relation
- Initially, each vertex is in a class by itself
- As edges are added, more vertices become related and the equivalence classes grow, until finally all the vertices are in a single equivalence class

Applying Partitions and Union Find in MST

Representatives of Equivalence Classes

- One vertex in each class may be chosen as the representative of that class

Now we are ready to apply these concepts for cycle determination

Applying Partitions and Union Find in MST

Representatives of Equivalence Classes

- One vertex in each class may be chosen as the representative of that class
- We arrange the vertices in lists that lead to the representative

Now we are ready to apply these concepts for cycle determination

Applying Partitions and Union Find in MST

Representatives of Equivalence Classes

- One vertex in each class may be chosen as the representative of that class
- We arrange the vertices in lists that lead to the representative
- This is the union-find structure

Now we are ready to apply these concepts for cycle determination

How to find detect a cycle ?...

In a typical MST algorith, cycle termination occurs as follows:

- If two vertices have the same representative, they are already connected and additing a further connection between them is pointless

How to find detect a cycle ?...

In a typical MST algorith, cycle termination occurs as follows:

- If two vertices have the same representative, they are already connected and additing a further connection between them is pointless
- Hence, for each end-point of the edge that you are going to add

How to find detect a cycle ?...

In a typical MST algorith, cycle termination occurs as follows:

- If two vertices have the same representative, they are already connected and additing a further connection between them is pointless
- Hence, for each end-point of the edge that you are going to add
  - follow the lists and find its representative

How to find detect a cycle ?...

In a typical MST algorith, cycle termination occurs as follows:

- If two vertices have the same representative, they are already connected and additing a further connection between them is pointless
- Hence, for each end-point of the edge that you are going to add
  - follow the lists and find its representative
  - if the two representatives are equal, then the edge will form a cycle

Applying Union-find structure to find MST

Let us apply the concept of equivalence classes to determine MST in the given graph

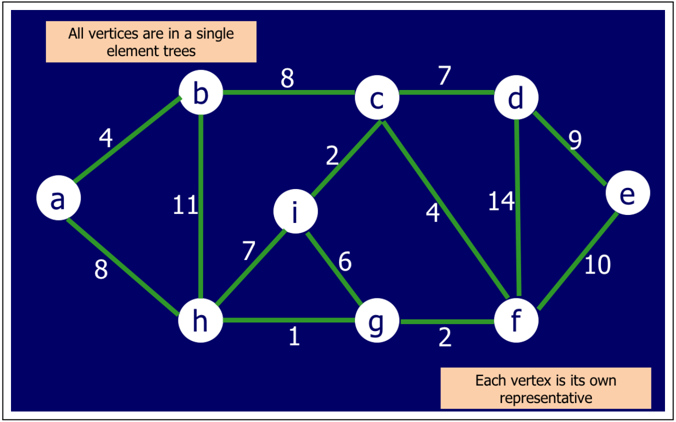


Figure: MST determination (a)

Applying Union-find structure to find MST...

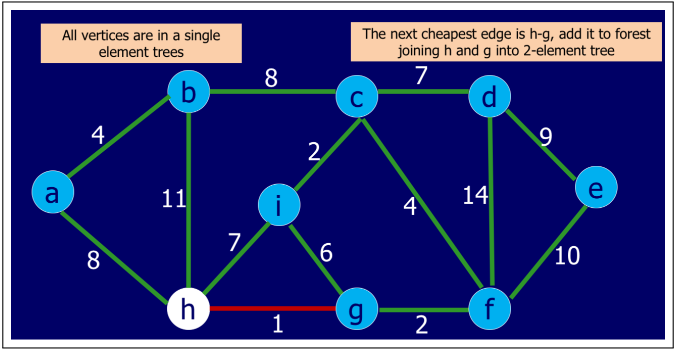


Figure: MST determination (b)

Applying Union-find structure to find MST...

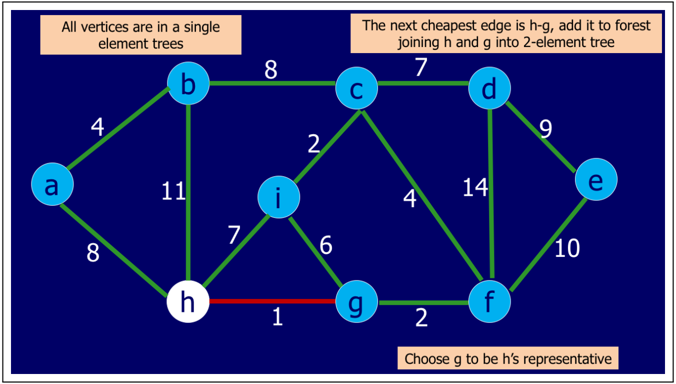


Figure: MST determination (c)

Applying Union-find structure to find MST...

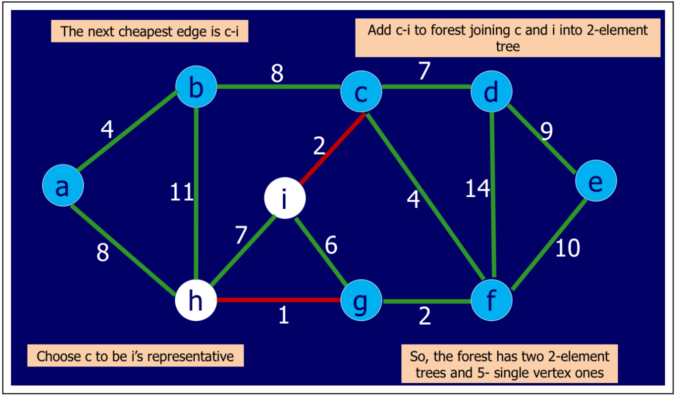


Figure: MST determination (d)

Applying Union-find structure to find MST...

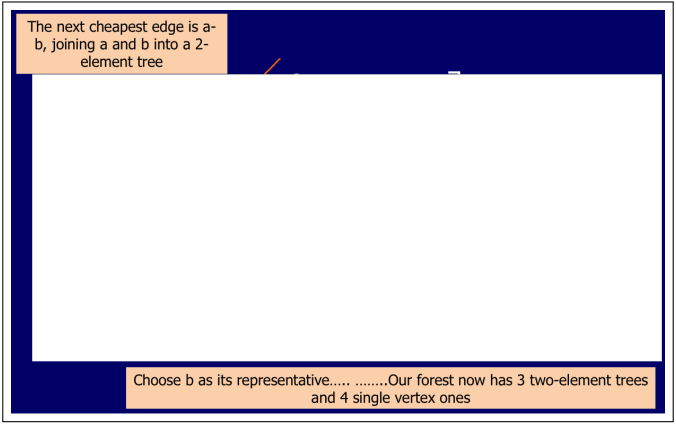


Figure: MST determination (e)

Applying Union-find structure to find MST...

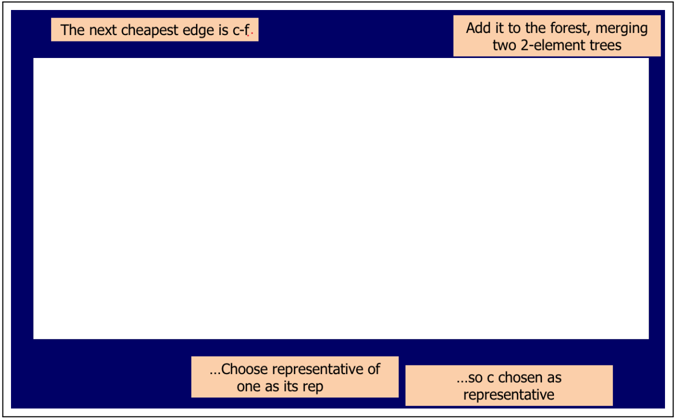


Figure: MST determination (f)

Applying Union-find structure to find MST...

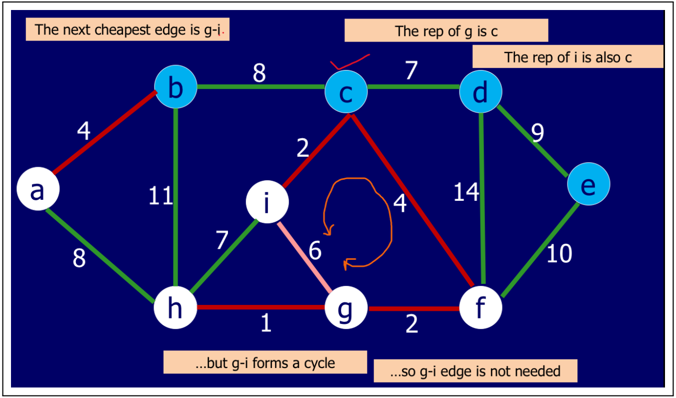


Figure: MST determination (g)

Applying Union-find structure to find MST...

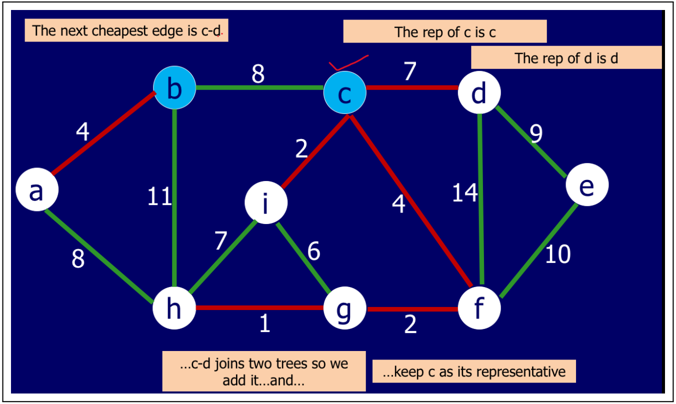


Figure: MST determination (h)

Applying Union-find structure to find MST...

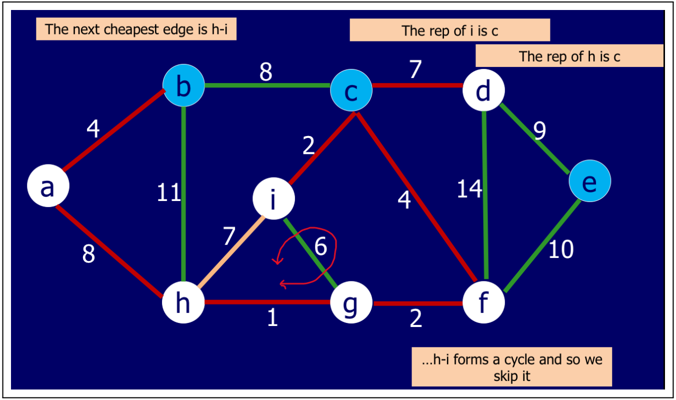


Figure: MST determination (i)

Applying Union-find structure to find MST...

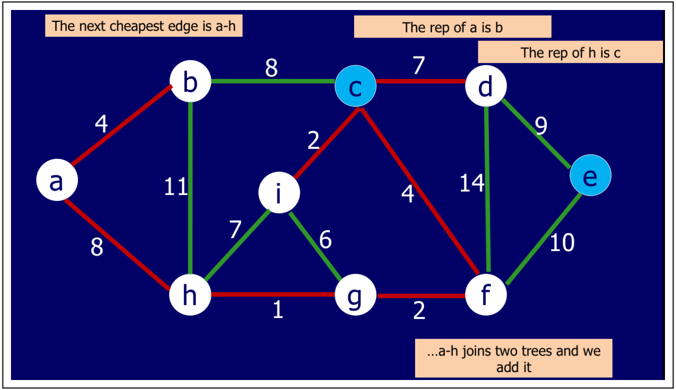


Figure: MST determination (j)

Applying Union-find structure to find MST...

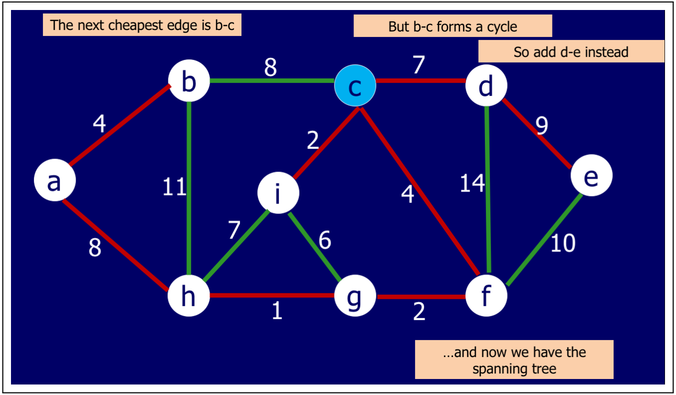


Figure: MST determination (k)

Applying Union-find structure to find MST...

Illustrate the algorithm discussed on the following graph to compute its MST

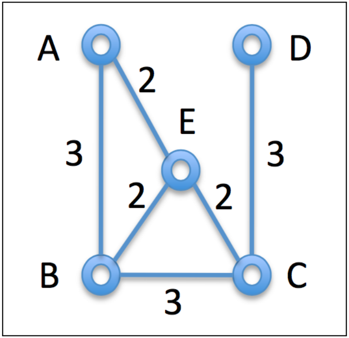


Figure: MST determination (k)

Applying Union-find structure to find MST...

Illustrate the algorithm discussed on the following graph to compute its MST

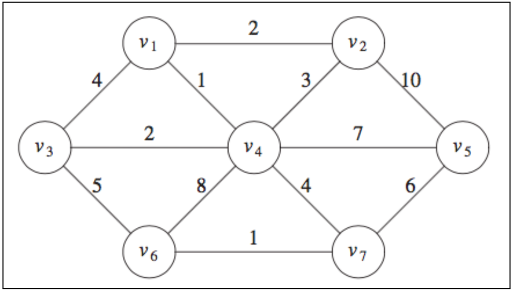
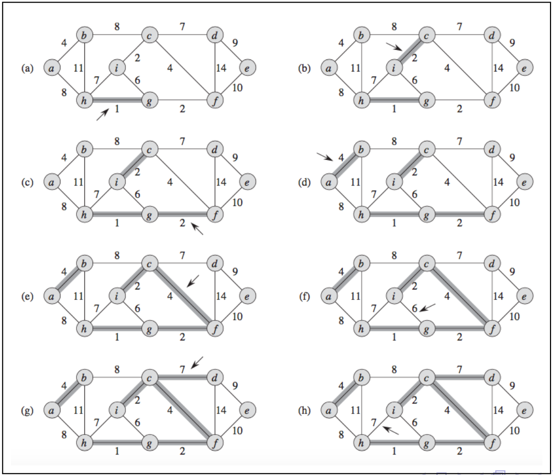


Figure: MST determination (k)

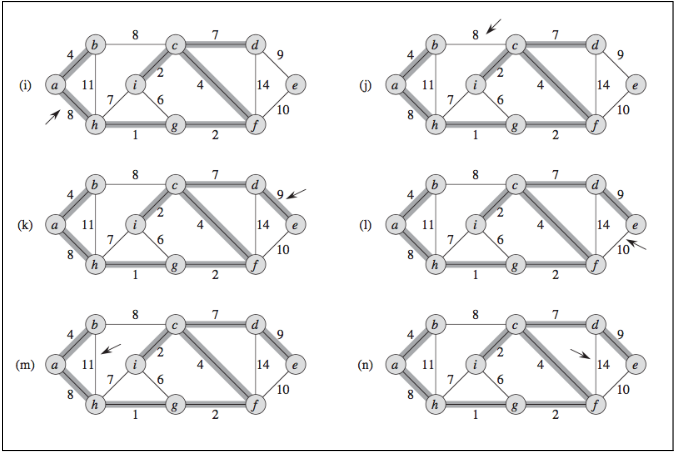
Applying Union-find structure to find MST...

Illustrate the algorithm discussed on the following graph to compute its MST



Applying Union-find structure to find MST...

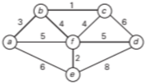
Illustrate the algorithm discussed on the following graph to compute its MST



Applying Union-find structure to find MST...

bf
4

bc	ef	ab	bf	cf	af	**df**	ae	cd	de
1	2	3	4	4	5	5	6	6	8



df
5

Kruskal's Algorithm

```
Algorithm Kruskal(Graph G(V,E), Weight_function w)
1. A=ϕ
2. for each vertex v ϵ V[G]
3.                do MAKE-SET(v)
4. Sort the edges of E into nondecreasing order
                                          by weight w
5. for each edge (u,v) ϵ E,
6.                if FIND-SET(u) ≠ FIND-SET(u)
7.                         A =A ∪ {(u,v)}
8.                         UNION(u,v)
9.        return A
```

Proving Kruskal's Algorithm

Proof by contradiction

- Note that any edge creating a cycle is not needed

Proving Kruskal's Algorithm

Proof by contradiction

- Note that any edge creating a cycle is not needed
- each edge must join two sub-trees

Proving Kruskal's Algorithm

Proof by contradiction

- Note that any edge creating a cycle is not needed
- each edge must join two sub-trees
- Suppose that the next cheapest edge, e_x, would join trees T_a and T_b

Proving Kruskal's Algorithm

Proof by contradiction

- Note that any edge creating a cycle is not needed
- each edge must join two sub-trees
- Suppose that the next cheapest edge, e_x, would join trees T_a and T_b
- Suppose that instead of $e_x$we choose e_z - a more expensive edge, which joins T_a and T_c

Proving Kruskal's Algorithm

Proof by contradiction

- Note that any edge creating a cycle is not needed
- each edge must join two sub-trees
- Suppose that the next cheapest edge, e_x, would join trees T_a and T_b
- Suppose that instead of $e_x$we choose e_z - a more expensive edge, which joins T_a and T_c
- But we still need to join T_b to T_a or some other tree to which T_a is connected.

Proving Kruskal's Algorithm

Proof by contradiction

- Note that any edge creating a cycle is not needed
- each edge must join two sub-trees
- Suppose that the next cheapest edge, e_x, would join trees T_a and T_b
- Suppose that instead of $e_x$we choose e_z - a more expensive edge, which joins T_a and T_c
- But we still need to join T_b to T_a or some other tree to which T_a is connected.
- What would be the cheapest way to do so ?

Proving Kruskal's Algorithm

Proof by contradiction

- Note that any edge creating a cycle is not needed
- each edge must join two sub-trees
- Suppose that the next cheapest edge, e_x, would join trees T_a and T_b
- Suppose that instead of $e_x$we choose e_z - a more expensive edge, which joins T_a and T_c
- But we still need to join T_b to T_a or some other tree to which T_a is connected.
- What would be the cheapest way to do so ?
- The cheapest way to do this is to add e_x - because e_x is the cheapest edge

Proving Kruskal's Algorithm

Proof by contradiction

- Note that any edge creating a cycle is not needed
- each edge must join two sub-trees
- Suppose that the next cheapest edge, e_x, would join trees T_a and T_b
- Suppose that instead of $e_x$we choose e_z - a more expensive edge, which joins T_a and T_c
- But we still need to join T_b to T_a or some other tree to which T_a is connected.
- What would be the cheapest way to do so ?
- The cheapest way to do this is to add e_x - because e_x is the cheapest edge
- So we should have added e_x instead of e_z

Proving Kruskal's Algorithm

Proof by contradiction

- Note that any edge creating a cycle is not needed
- each edge must join two sub-trees
- Suppose that the next cheapest edge, e_x, would join trees T_a and T_b
- Suppose that instead of $e_x$we choose e_z - a more expensive edge, which joins T_a and T_c
- But we still need to join T_b to T_a or some other tree to which T_a is connected.
- What would be the cheapest way to do so ?
- The cheapest way to do this is to add e_x - because e_x is the cheapest edge
- So we should have added e_x instead of e_z
- This proves that the greedy approach is correct for MST

Theorems Associated

Lemma

Let F be a forest, that is any undirected acyclic graph. Let e = (v, w) be an edge that is NOT in F. Then, there is a cycle in F, consisting of edges of F and edge e, iff v and w are in the same connected component of F.

Theorem

Let G = (V,E) be a connected, undirected graph with a real-valued weight function w defined on the set of edges E. Let A be a subset of E that is included in some MST for G. Let (S, V-S) be a cut of G that respects A and let (u,v) be a light edge crossing (S, V-S). Then, edge (u,v) is safe for A.

Theorems Associated

Theorem

Let G=(V,E) be a connected, undirected graph and G'=(V',E') be a partial graph formed by the nodes of G and the edges in E'. Let there be n nodes in V. Then, prove that the set E' with n or more edges cannot be optimal. Also, prove that E' must have exactly (n-1) edges and so E' must be a tree.

Theorem

Prove that Kruskal's algorithm is correct and it finds its MST.

Tutorial Assignment Proofs

Theorem

- Let T be a MST for a graph G, let e be an edge in T and let T' be T with e removed. Show that e is a minimum weight edge between components of T'.
- Comment on the validity of the statement. If all the weights in G are distinct, distinct spanning trees of G have distinct weights. Give a counterexample.
- Let T and T' be two STs of a connected graph G. Suppose that an edge e is in T but not in T'. Show that there is an edge e' in T', but not in T, such that (T-e ∪ e') and (T'-e' ∪ e) are STs of G.

Time Complexity

Steps

- Initialise forest O($|V|$)
- Sort edges O($|E|\log|E|$)
    - Check edge for cycles O($|V|$) x
    - Number of edges O($|V|$) i.e. O($|V|^2$)
- Total O($|V| + |E| log |E| + |V|^2$)
- Since $|E| = O(\ |V|^2)$
  Total $= O(|V|^2\ log|V|)$

- Thus, Kruskal's algorithm is tagged as On^2 log n) algorithm for a graph of n vertices
- This is an upper bound, some improvements on this are known...
- Prim's algorithm can be O($|E| + |V| log |V|$) using Fibonacci heaps.
- Even better variants are know for restricted cases, such as sparse graphs ($|E| \approx |V|$)

Prim's Algorithm

PRim's Algorithm

Approach

- Follows the natural greedy approach starting with the source vertex to create the spanning tree,
- add an edge to the tree that is attached at exactly one end to the tree & has minimum weight among all such edges.
- Prim's algorithm starts from one vertex and grows the rest of the tree an edge at a time.
- As a greedy algorithm, which edge should we pick?

Prim's Algorithm

Points to note

- In Kruskal, the selection function uses a greedy approach, i.e. chooses an edge that is minimum weighted edge, then in increasing order.

Prim's Algorithm

Points to note

- In Kruskal, the selection function uses a greedy approach, i.e. chooses an edge that is minimum weighted edge, then in increasing order.
    - does not worry too much about their connection to the previously chosen edges, except watching for cycles.

Prim's Algorithm

Points to note

- In Kruskal, the selection function uses a greedy approach, i.e. chooses an edge that is minimum weighted edge, then in increasing order.
    - does not worry too much about their connection to the previously chosen edges, except watching for cycles.
    - the result is a sort of forest of trees that grow haphazardly and later merge into a tree.

Prim's Algorithm

Points to note

- In Kruskal, the selection function uses a greedy approach, i.e. chooses an edge that is minimum weighted edge, then in increasing order.
  - does not worry too much about their connection to the previously chosen edges, except watching for cycles.
  - the result is a sort of forest of trees that grow haphazardly and later merge into a tree.
- As compared in Prim's the MST grows in a natural manner, staring from an arbitrary root.

Prim's Algorithm

Points to note

- In Kruskal, the selection function uses a greedy approach, i.e. chooses an edge that is minimum weighted edge, then in increasing order.
    - does not worry too much about their connection to the previously chosen edges, except watching for cycles.
    - the result is a sort of forest of trees that grow haphazardly and later merge into a tree.
- As compared in Prim's the MST grows in a natural manner, staring from an arbitrary root.
- At each, a new edge is added to a tree already constructed.

PRim's Algorithm

Approach

- Consider B - a set of nodes, T - a set of edges

PRim's Algorithm

Approach

- Consider B - a set of nodes, T - a set of edges
- initially B contains a single arbitrary node

PRim's Algorithm

Approach

- Consider B - a set of nodes, T - a set of edges
- initially B contains a single arbitrary node
- at each step, Prim's algorithm looks for the shortest edge $\{u,v\}$ such that u ϵ B and v ϵ N - B.

PRim's Algorithm

Approach

- Consider B - a set of nodes, T - a set of edges
- initially B contains a single arbitrary node
- at each step, Prim's algorithm looks for the shortest edge $\{u,v\}$ such that u ϵ B and v ϵ N - B.
- then it adds v to B and $\{u, v\}$ to T.

PRim's Algorithm

Approach

- Consider B - a set of nodes, T - a set of edges
- initially B contains a single arbitrary node
- at each step, Prim's algorithm looks for the shortest edge $\{u,v\}$ such that u ϵ B and v ϵ N - B.
- then it adds v to B and $\{u, v\}$ to T.
- in this way, in T at any instant an MST for the nodes in B is formed.

Prim's Algorithm

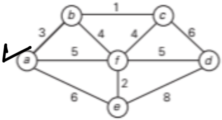
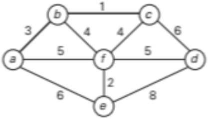
```
Algorithm Prim(Graph G(V,E), Weight_function w)
{initialization}
1. T ϵ ϕ
2. B ← {an arbitrary member of V}
3. while B ≠ N do
4.        find E={u, v} of minimum weight such that
                     u ϵ B and v ϵ V–B
5.        T ← T ∪ {e}
6.        B ← B ∪ {v}
7. return T
```

How to determine the edge with the minimum weight ?

Prim's Algorithm Refined

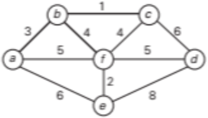
```
Algorithm Prim(Graph G(V,E), Weight_function w, r)
/* the connected graph G and the root r of the minimum
spanning tree to be grown are inputs to the algorithm.*/
1. for each u ϵ G.V
2.        u.key=∞
3.        u.π=NIL
4. r.key=0
\*  During execution of the algorithm, all vertices that
are not in the tree reside in a min-priority queue Q
based on a key attribute.*/
5. Q=G.V
6. while Q ≠ ϕ
7.                u=EXTRACT-MIN(Q)
8.                for each v ϵ G.Adj[u]
9.                        if v ϵ Q and w(u,v) < v.key
10.                          v.π = u
11.                          v.key = w(u,v)
```

Applying Prim's Algorithm...

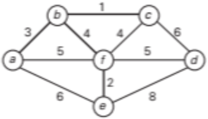
Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	

Applying Prim's Algorithm...

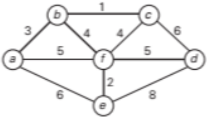
c(b, 1) d(c, 6) e(a, 6) **f(b, 4)**



f(b, 4) d(f, 5) **e(f, 2)**

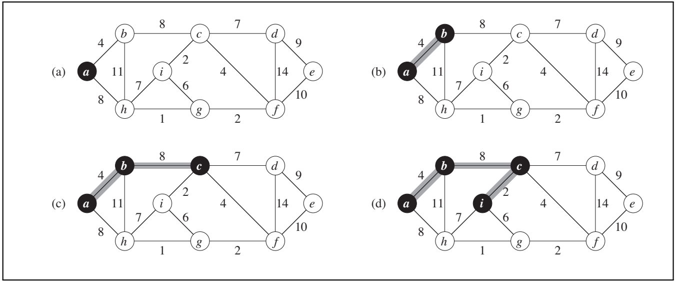


e(f, 2) **d(f, 5)**



d(f, 5)

Applying Prim's Algorithm...



Applying Prim's Algorithm...

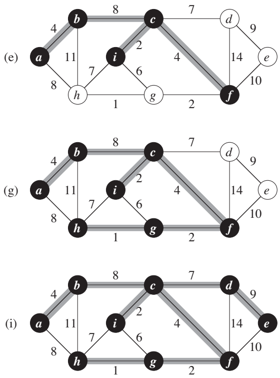
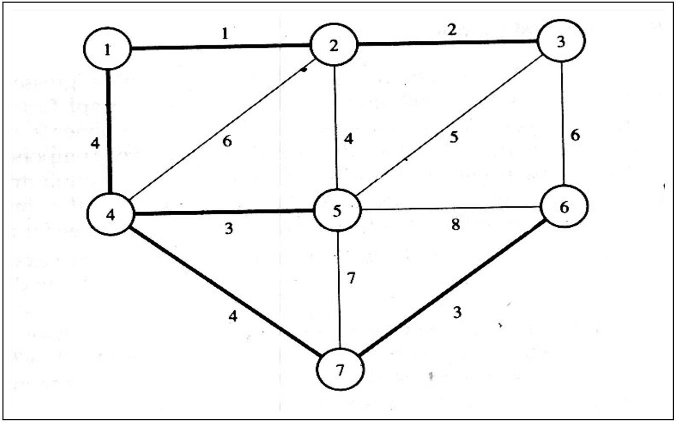
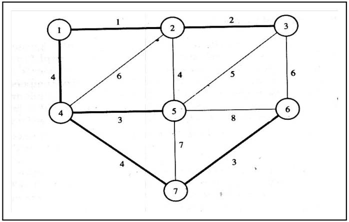


Figure 23.5 The execution of Prim's algorithm on the graph from Figure 23.1. The root vertex is a. Shaded edges are in the tree being grown, and black vertices are in the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. In the second step, for example, the algorithm has a choice of adding either edge (b, c) or edge (a, h) to the tree since both are light edges crossing the cut.

Comparing Kruskal and Prim by applying them

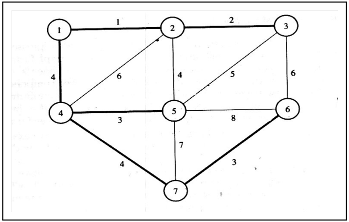


Running Prim's on the given graph



Step	EdgeConsidered	ConnectedComponents
Init	-	{1}
1	{1 2}	{1 2}
2	{2 3}	{1 2 3}
3	{1 4}	{1 2 3 4 }
4	{4 5}	{1 2 3 4 5 }
5	{4 7}	{1 2 3 4 5 7}
6	{5}	Rejected
7	{4 7}	{1 2 3 4 5 6 7}

Running Kruskal's on the given graph



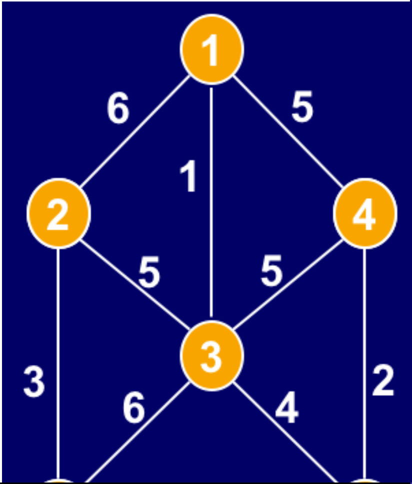
dense ?
sparse ?

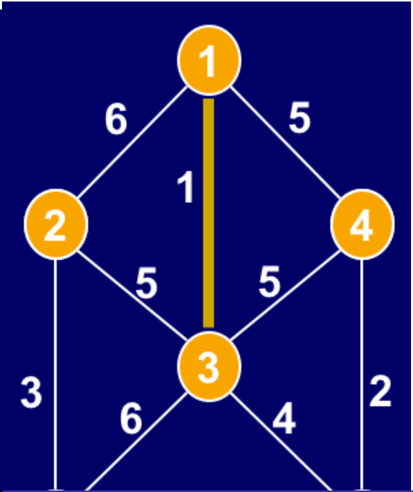
Step	EdgeConsidered	ConnectedComponents
Init	-	{1} {2} {3} {4} {5} {6} {7}
1	{1 2}	{1 2} {3} {4} {5} {6} {7}
2	{2 3}	{1 2 3} {4} {5} {6} {7}
3	{4 5}	{1 2 3} {4 5} {6} {7}
4	{6 7}	{1 2 3} {4 5} {6 7}
5	{1 4}	{1 2 3 4 5} {6 7}
6	{2 5}	Rejected
7	{4 7}	{1 2 3 4 5 6 7}

← forest

Applying Prim's Algorithm

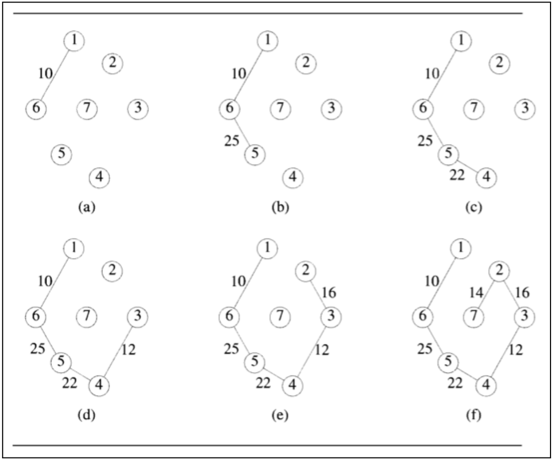
Illustrate the PRim's algorithm in the following graph to compute its MST





Applying Prim's Algorithm...

Illustrate the PRim's algorithm in the following graph to compute its MST



Applying Prim's Algorithm...

Applying Prim's Algorithm...

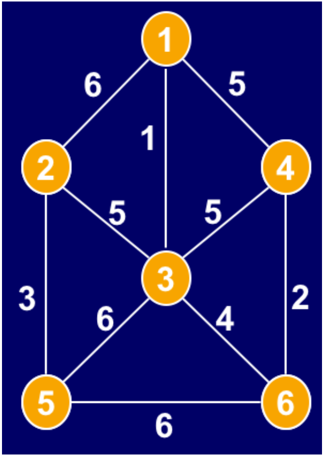


Figure: Various iterations of Prim

Applying Prim's Algorithm...

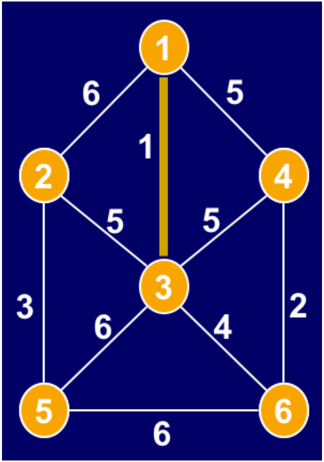


Figure: Various iterations of Prim

Applying Prim's Algorithm...

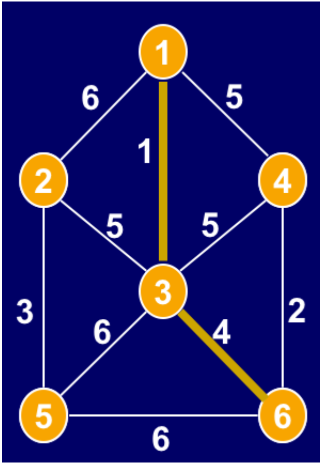


Figure: Various iterations of Prim

Applying Prim's Algorithm...

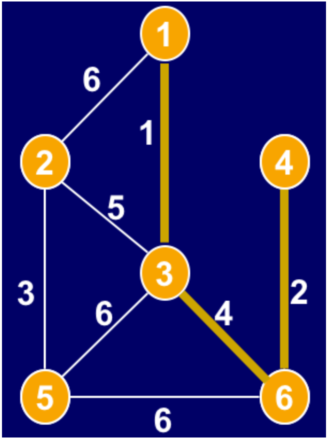


Figure: Various iterations of Prim

Applying Prim's Algorithm...

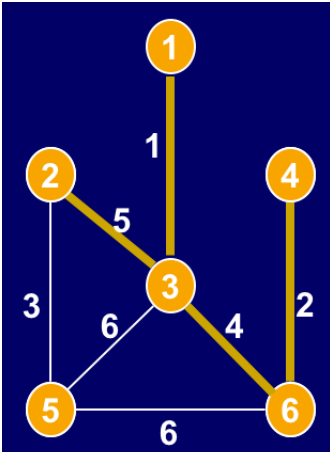


Figure: Various iterations of Prim

Applying Prim's Algorithm...

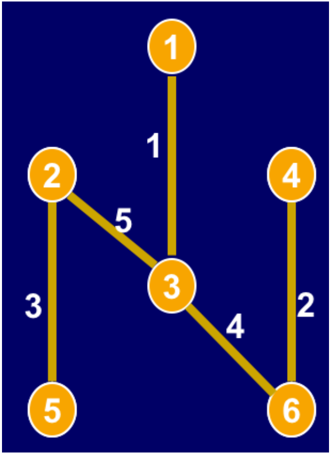
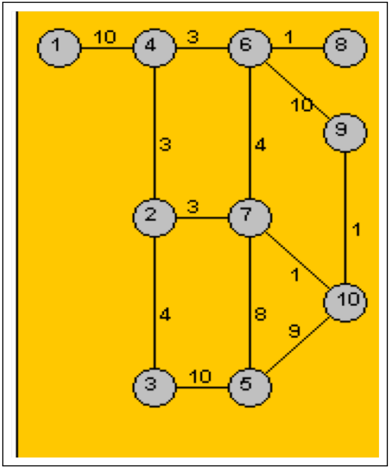


Figure: Various iterations of Prim

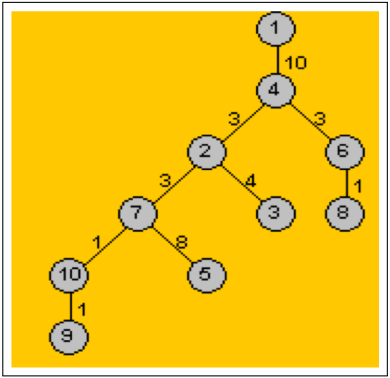
Applying Prim's Algorithm...

Illustrate the PRim's algorithm in the following graph to compute its MST



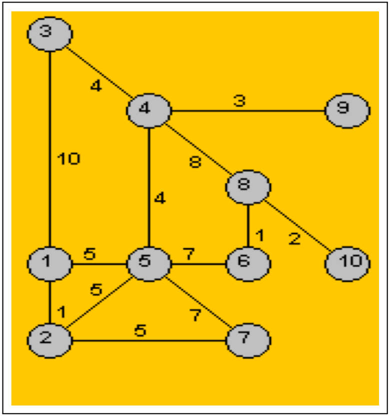
Applying Prim's Algorithm...

Answer...



Applying Prim's Algorithm...

Illustrate the PRim's algorithm in the following graph to compute its MST



Prim's Algorithm Refined...repeated here

```
Algorithm Prim(Graph G(V,E), Weight_function w, r)
/* the connected graph G and the root r of the minimum
spanning tree to be grown are inputs to the algorithm.*/
1. for each u ϵ G.V
2.        u.key=∞
3.        u.π=NIL
4. r.key=0
\*  During execution of the algorithm, all vertices that
are not in the tree reside in a min−priority queue Q
based on a key attribute.*/
5. Q=G.V
6. while Q ≠ ϕ
7.        u=EXTRACT−MIN(Q)
8.             for each v ϵ G.Adj[u]
9.                   if v ϵ Q and w(u,v) < v.key
10.                        v.π = u
11.                        v.key = w(u,v)
```

Prim's Algorithm: Timing Analysis

- Assume that minimum priority queue Q is implemented as a min-heap.

Prim's Algorithm: Timing Analysis

- Assume that minimum priority queue Q is implemented as a min-heap.
- Then, building it in the lines 1-5 of the algorithm takes O(V) time.

Prim's Algorithm: Timing Analysis

- Assume that minimum priority queue Q is implemented as a min-heap.
- Then, building it in the lines 1-5 of the algorithm takes O(V) time.
- Statements 6-11 executed for each vertex i.e. $|V|$ times and since EXTRACT-MIN(Q) takes O(lg V) times and so total time is O(V lg V)

Prim's Algorithm: Timing Analysis

- Assume that minimum priority queue Q is implemented as a min-heap.
- Then, building it in the lines 1-5 of the algorithm takes O(V) time.
- Statements 6-11 executed for each vertex i.e. $|V|$ times and since EXTRACT-MIN(Q) takes O(lg V) times and so total time is O(V lg V)
- for loop in 8-12 executed O(E) times

Prim's Algorithm: Timing Analysis

- Assume that minimum priority queue Q is implemented as a min-heap.
- Then, building it in the lines 1-5 of the algorithm takes O(V) time.
- Statements 6-11 executed for each vertex i.e. $|V|$ times and since EXTRACT-MIN(Q) takes O(lg V) times and so total time is O(V lg V)
- for loop in 8-12 executed O(E) times
- while the statement 11 requires O(lg V) times and so

Prim's Algorithm: Timing Analysis

- Assume that minimum priority queue Q is implemented as a min-heap.
- Then, building it in the lines 1-5 of the algorithm takes O(V) time.
- Statements 6-11 executed for each vertex i.e. $|V|$ times and since EXTRACT-MIN(Q) takes O(lg V) times and so total time is O(V lg V)
- for loop in 8-12 executed O(E) times
- while the statement 11 requires O(lg V) times and so
- total time of for loop is O(E lg V).

Prim's Algorithm: Timing Analysis

- Assume that minimum priority queue Q is implemented as a min-heap.
- Then, building it in the lines 1-5 of the algorithm takes O(V) time.
- Statements 6-11 executed for each vertex i.e. $|V|$ times and since EXTRACT-MIN(Q) takes O(lg V) times and so total time is O(V lg V)
- for loop in 8-12 executed O(E) times
- while the statement 11 requires O(lg V) times and so
- total time of for loop is O(E lg V).
- Therefore, Prim takes O(V lg V + E lg V) time.

Blank

Blank