MST Problem – Prim's Algorithm

- Whereas Kruskal's algorithm is based on growing a **forest**, Prim's is based on growing a **tree**.
- Prim's algorithm like Kruskal's utilizes the Greedy Method design strategy.
- Again the input is a weighted connected graph $G(V,E)$ and the output is a minimum spanning tree**.**

The textbook reading for Prim's algorithm is 6.5.2 (Chapter 6, Section 5, Subsection 2), pp. 266-272.

Growing a Tree using Cut

- Suppose we are growing a tree T starting with an initial or root vertex r.
- Then, at any stage of expanding T, we must choose and edge $e = \{u,v\}$ where ***u* is in the tree**, i.e., $u \in V(T)$ = the vertex set of T, and ***v* is not in the tree**, i.e., $v \in V - V(T)$.
- Otherwise, in the case where both u and v belong to the tree a cycle would be formed or in the case both u and v are not in the tree, the edge e would be disconnected from T.
- The set of edges having one end vertex in the tree and the other not in tree, which we denote by $Cut(T)$, is called a **cut** in graph theory. **Thus, in order to expand T we must choose and edge from *Cut*(*T*).**

High-Level Description of Prim's Algorithm

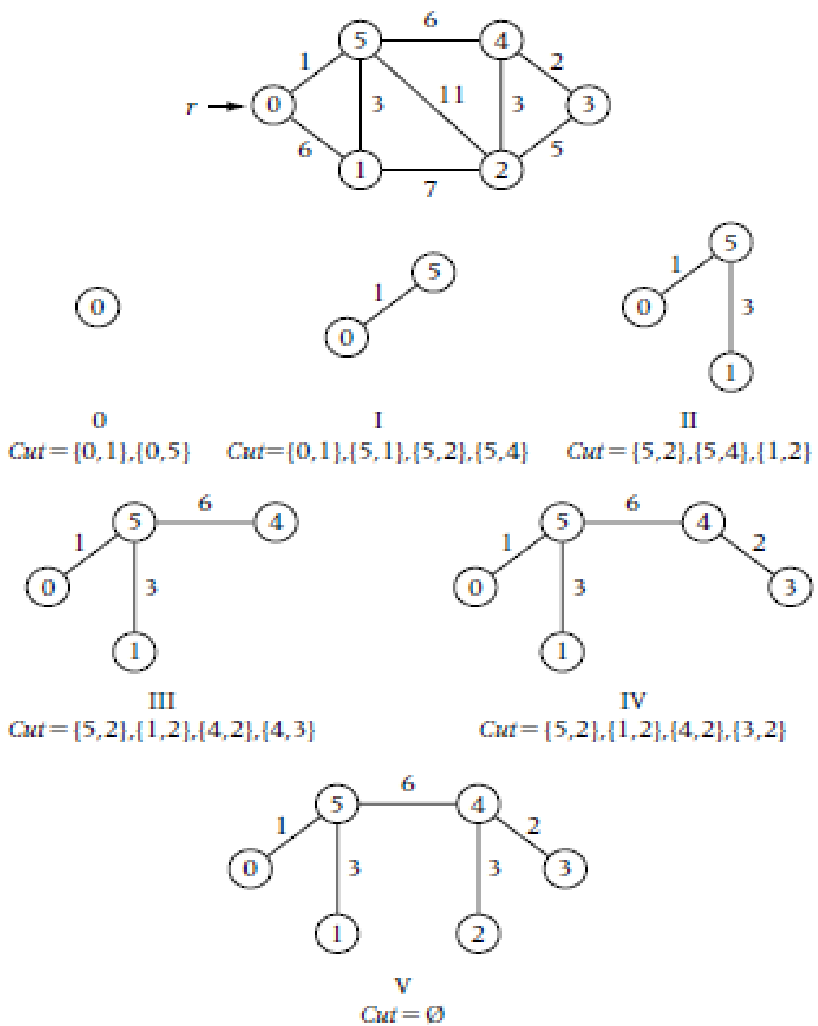
Start with an initial tree T_0 consisting of a single vertex r and no edges. Vertex r can be chosen arbitrarily to be **any vertex**.

We then grow a spanning tree rooted at r by building a sequence of $n - 1$ trees $T_1,\ldots,T_{n-1}$, where

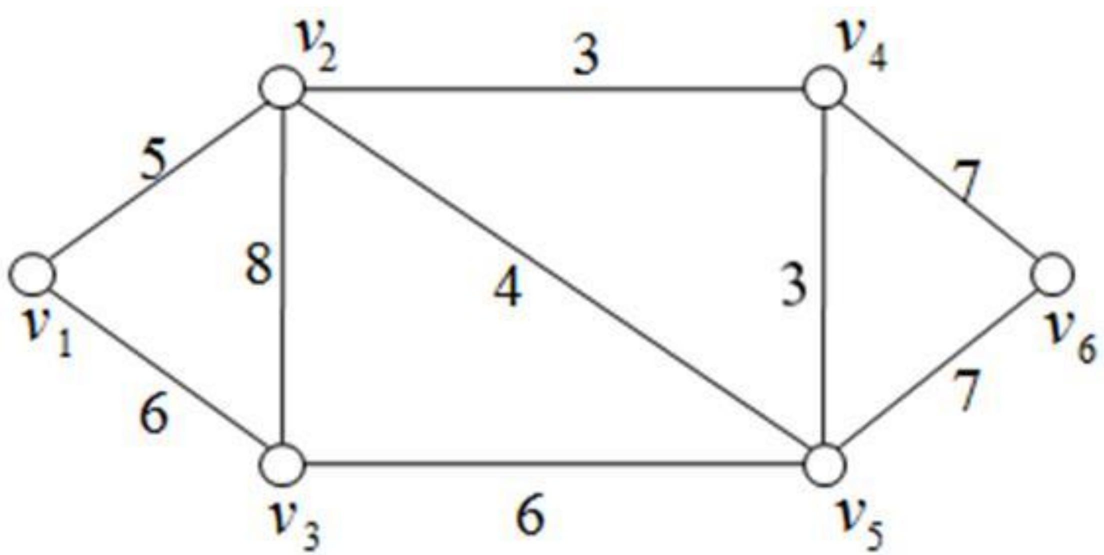
$$T_i = T_{i-1} + e_i\ , \ i = 1, 2, \ldots, n - 1,$$

where e_i is chosen so that it has **minimum weight** among all the edges in $Cut(T_i)$. The final tree T_{n-1} is a minimum spanning tree.

Action of Prim's for sample graph *G*



PSN. Find a MST in the following weighted graph using Prim's algorithm starting at vertex v_3.



Finding minimum edge in *Cut*(*T*) – Naïve Algorithm

Finding the minimum edges in $Cut(T_i)$ can be very expensive time-wise if we scan all the edges of the cut to locate the one having minimum weight. For example, if G is the complete graph K_n, then the size of a cut when computing e_i is $i * (n - i)$. To see this observe that there are i vertices in T_i and $n - i$ vertices not in T_i. Since G is complete the number of edges joining these sets is $i * (n - i)$, i.e.,

$$|\ Cut(T_i)\,| = i * (n - i)\ ,\ i = 0\ ,\ \ldots,\ n - 1.$$

Thus the computing time over all these cuts is:

$$\sum_{i=1}^{n} i * (n-i) > \sum_{i=n/4}^{3n/4} i * (n-i) > \sum_{i=n/4}^{3n/4} \frac{n}{4} * \frac{3n}{4} = \frac{n}{2} * \frac{n}{4} * \frac{3n}{4} = \frac{3n^3}{32} \in \Theta(n^3).$$

Using the naïve algorithm to compute the minimum edges in each cut by simply scanning all the edges in the cut would result in Prim's algorithm having worst-case complexity $W(n) \in \Theta(n^3)$

It turns out we can do much better than this.

Improvement using a Priority Queue

We maintain a priority queue of vertices not in the current tree T, i.e., the set of vertices $V - V(T)$. *We take the priority of*

$v \in V - V(T)$ to be

$$Nearest[v] = \text{minimum weight over all edges } \{u,v\} \text{ where } u \text{ belongs to the tree } T$$

Letting u^* denote the vertex in the tree that is nearest to v, define

$$Parent[v] = u^*$$

Note that $\{u^*,v\}$ is an edge in $Cut(T)$ that has minimum weight over all the edges in $Cut(T)$ that are incident with v.

Minimum Weight Edge in *Cut*(*T*)

Based on our last observation, it follows that

minimum weight of the an edge in *Cut*(*T*)
= minimum of *Nearest*[*v*] over all *v* not in *T*
= minimum over all *v* in the priority queue.

Thus, this minimum weight over all the edges in *Cut*(*T*) can be computed simply by performing one **dequeue** operation.

Maintaining *Nearest*

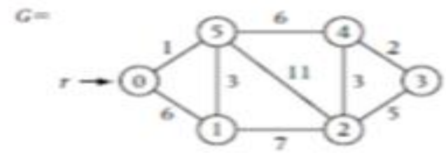
After we dequeue a vertex *u* we must update *Nearest*. The vertices for which *Nearest* will be affected are those vertices *v* not in the tree that are adjacent to *u*. They can be updated by simply scanning the neighborhood of *u* and performing the operation

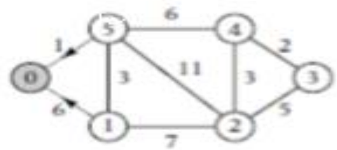
if $w(uv) < Nearest[v]$ **then**
  $Nearest[v] \leftarrow w(uv)$
  $Parent[v] \leftarrow u$
endif

Pseudocode for Prim's Algorithm

```
procedure Prim(G,w,Parent[0:n – 1])
Input:    G (a connected graph with vertex set V and edge set E)
          w (a weight function on E)
Output: Parent[0:n – 1] (parent array of a minimum spanning tree}
   for v ← 0 to n – 1 do
          Nearest[v] ← ∞
          InTheTree[v] ← .false.
   endfor
   Parent[r] ← -1          // Stage 1: root the tree at an arbitrary vertex r
   Nearest[r] ← 0
   for Stage ← 2 to n – 1 do
      Select vertex u that minimizes Nearest[u] over all u such that InTheTree[u] = .false.
      InTheTree[u] ← .true.     // add u to T
         for each vertex v such that uv ∈ E do        // update Nearest[v] and
           if .not. InTheTree[v] then                 // Parent[v] for all v ∉ V(T)
                 if w(uv) < Nearest[v] then           //that are adjacent to u
                    Nearest[v] ← w(uv)
                    Parent[v] ← u
                 endif
            endif
          endfor
   endfor
end Prim
```

Action of Prim’s Algorithm for Sample Graph *G*

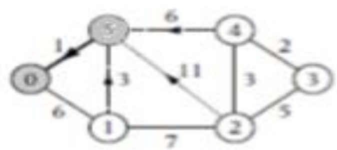




i	0	1	2	3	4	5
Nearest[i]	0	6	∞	∞	∞	1
Parent[i]	−1	0	–	–	–	0
InTheTree[i]	T	F	F	F	F	F

Weight = 0

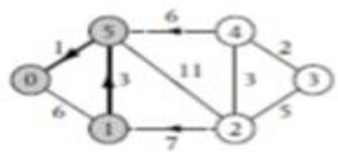
Stage 1



i	0	1	2	3	4	5
Nearest[i]	0	3	11	∞	6	1
Parent[i]	−1	5	5	–	5	0
InTheTree[i]	T	F	F	F	F	T

Weight = 1

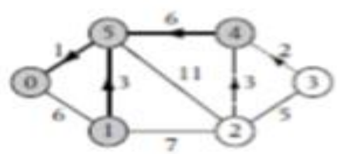
Stage 2



i	0	1	2	3	4	5
Nearest[i]	0	3	7	∞	6	1
Parent[i]	−1	5	1	–	5	0
InTheTree[i]	T	T	F	F	F	T

Weight = 4

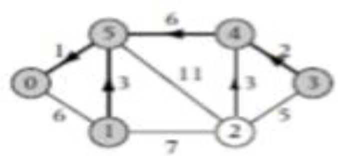
Stage 3



i	0	1	2	3	4	5
Nearest[i]	0	3	3	2	6	1
Parent[i]	−1	5	4	4	5	0
InTheTree[i]	T	T	F	F	T	T

Weight = 10

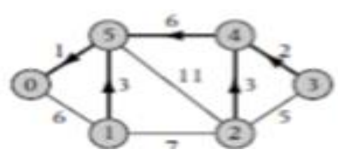
Stage 4



i	0	1	2	3	4	5
Nearest[i]	0	3	3	2	6	1
Parent[i]	−1	5	4	4	5	0
InTheTree[i]	T	T	F	T	T	T

Weight = 12

Stage 5



After stage 5, array *Parent* is now complete, and implements a minimum spanning tree with weight 15.

Slight Savings

Note that the procedure *Prim* terminates after only $n - 1$ stages, even though there are n vertices in the final minimum spanning tree. The reason is simple: After stage $n - 1$ has been completed,

InTheTree[0:$n - 1$] is **.false.** for only one vertex w. Thus, another iteration of the **for** loop controlled by *Stage* would result in no change to the arrays *Nearest*[0:$n - 1$] and *Parent*[0:$n - 1$]. In other words, the last vertex and edge in the minimum spanning tree come in for free.

Analysis of Procedure Prim

- There are $n - 1$ stages and each stage involves at most $n - 2$ comparisons of edge weights to find the minimum value of Nearest and at most $n - 1$ comparisons to update *Nearest*.
- Thus, Procedure *Prim* has worst-case complexity

$$W(n) \in \Theta\,(n^2)\ .$$

Using min-heap to implement priority queue

The dequeue operation for each vertex takes time $O(\log n)$, so

the total time for dequeuing is $O(n \log n)$.

Updating the priorities when vertex v is added to the tree T takes time $O(d(v)\log n)$ where $d(v)$ denotes the degree of v. That is because the priority may need to be updated for each vertex in the neighborhood of v and each update takes time $O(\log n)$. Thus,

the total time for updates over the entire algorithm is $O(m \log n)$.

This follows from the observation that the sum of $d(v)$ over all the vertices v equals twice the number of edges, i.e., $2m$.

Min-heap implementation of priority queue, cont'd

Thus, the time for updates dominates the time for dequeuing, so that

the computing time of Prim's algorithm is $O(m \log n)$.

Note that to achieve this, it is necessary to implement the digraph D using **adjacency lists**. Also, this result is sharp, i.e., the worst-case complexity of Prim's algorithm implemented in this way is given by

$$W(m,n) \in \Theta(m \log n).$$