# Efficient Algorithms

Ekkapot Charoenwanit

Software Systems Engineering
TGGS
KMUTNB

# Lecture 8: Greedy Algorithms

## Greedy Algorithms

***Characteristics:***

A greedy algorithm constructs a solution to an optimization problem in a ***piece-by-piece*** manner by making a sequence of choices that are

- ***feasible*** in the sense that the requirements specified by the problem are met
- ***locally optimal*** by picking the best choice at the moment of consideration
- ***irrevocable***, which means any change cannot be done to any previous partial solution

In this lecture, we will focus on only greedy algorithms that ***always*** yield ***optimal solutions*** to the problems they solve.

## Activity Selection Problem

***Input:*** A set of $n$ activities $S = \{a_1, a_2, \ldots, a_n\}$

Each activity $a_i$ has a ***start time*** denoted by $s_i$ and a ***finish time*** denoted by $f_i$ such that $0 \leq s_i < f_i < \infty$.

Two activities $a_i$ and $a_j$ are ***mutually compatible*** if and only if their time intervals do not overlap:

$f_i \leq s_j$ ($a_i$ finishes before $a_j$ starts.)

or

$f_j \geq s_i$ ($a_j$ finishes before $a_i$ starts.)

***Output:*** A maximal-size subset of mutually compatible activities

## Activity Selection Problem

***Optimal Substructure***

Let $S_{ij}$ denote the set of activities that start after $a_i$ finishes and that finish before $a_j$ starts.

Suppose that $A_{ij}$ is a maximal-size subset of mutually compatible activities in $S_{ij}$. Additionally, suppose further that some activity $a_k \in A_{ij}$.

By including $a_k$ into an optimal solution, we are left with ***two independent subproblems*** to solve, namely, finding a set of mutually compatible activities in $S_{ik}$ and finding a set of mutually compatible activities in $S_{kj}$.

## Activity Selection Problem

***Optimal Substructure (Continued)***

$S_{ik}$ is the set of activities in $S_{ij}$ that finishes before $a_k$ starts.
$S_{kj}$ is the set of activities in $S_{ij}$ that starts after $a_k$ finishes.

Thus, we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$.

Since $A_{ik}$, $\{a_k\}$ and $A_{kj}$ are mutually disjoint,

$$|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$$

## Activity Selection Problem

***Optimal Substructure (Continued)***

We will show that $A_{ik}$ and $A_{kj}$ are also maximal-size set of mutually compatible activities in $S_{ik}$ and $S_{kj}$, respectively, using a ***cut-and-paste argument***.

***Proof:*** We will prove by contradiction as follows.

Suppose that $A_{ik}$is ***not optimal***.

Then, there exists some set $A'_{ik}$ of mutually compatible activities in $S_{ik}$ where $|A'_{ik}| > |A_{ik}|$.

Let us denote the new solution by $A'_{ij}$.

[***Cut-and-Paste***]Replacing $A_{ik}$ with $A'_{ik}$ will increase the size of the solution for $S_{ij}$:

$|A'_{ij}| = |A'_{ik}| + |A_{kj}| + 1 > |A_{ij}|$, which ***contradicts*** the optimality of $A_{ij}$.

Therefore, $A_{ik}$ is a maximal-size set of mutually compatible activities in $S_{ik}$.

A symmetric argument applies to the optimality of $A_{kj}$. ■

## Activity Selection Problem

Since the activity selection problem exhibits ***optimal substructure***, we can solve the problem via ***DP*** based on the following recursive formulation, where $c[i, j]$ denotes the maximal size of an optimal solution to $S_{ij}$.

$$c[i,j] = \begin{cases} 0, & S_{ij} = \emptyset \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\}, & S_{ij} \neq \emptyset \end{cases}$$

Here, we must examine all the activities $a_k \in S_{ij}$ and pick one that ***maximizes*** the number of mutually compatible activities in $S_{ij}$.

## Activity Selection Problem

We will now argue that, in fact, we need not examine all the activities $a_k \in S_{ij}$ at each step as we would do via DP at all.

We will show that the activity selection problem exhibits another key property called ***the greedy choice property***, that is, the algorithm makes a ***locally best choice*** at each step, and it is guaranteed that it will eventually arrive at a ***globally optimal solution***.

With ***the greedy choice property***, the recursive formulation can be simplified since $a_k$ is always an activity with the ***minimum finish time***, and we are left with ***only one smaller subproblem*** to solve, instead of two smaller subproblems if DP is used.

## Activity Selection Problem

***Greedy Choice Property***

Intuition suggests that we should probably choose an activity that leaves the resource available for as many other activities as possible.

Therefore, if we follow this intuition, an activity with the ***smallest finish time*** should be first picked since this will leave the resource available for scheduling as many other activities as possible.

If there is more than one activity with the smallest finish time, we pick any one arbitrarily to break the tie.

# Activity Selection Problem

***Greedy Choice Property (continued):***

Assume that we sort the activities in $S = \{a_1, a_2, \dots, a_n\}$ in ***non-decreasing*** order of their finish times so that $f_1 \le f_2 \le \cdots \le f_n$.

Therefore, our greedy choice would be $a_1$ since it has the smallest/earliest finish time $f_1$.

Therefore, after picking $a_1$, we are left with ***one smaller subproblem*** to solve: finding activities that start after $a_1$ finishes because all activities that are mutually compatible with $a_1$ must start after $a_1$ finishes.

Let $S_k = \{a_i | \, s_i \ge f_k\}$ be the set of activities that start after $a_k$ finishes. With our greedy choice strategy, $S_1 = \{a_i | \, s_i \ge f_1\}$ is the ***only smaller subproblem*** that remains to be solved after $a_1$ is picked.

# Activity Selection Problem

***Greedy Choice Property (continued):***

We have already established that the activity selection problem exhibits ***optimal substructure***.

***Optimal substructure*** says that if there is an optimal solution that includes $a_1$, then the solution to the subproblem $S_1$ must also be optimal.

Now comes a question: is our intuition correct that $a_1$ is always part of some optimal solution?

We will now show that $a_1$ is always part of some optimal solution.

# Activity Selection Problem

***Theorem:*** Given any non-empty subproblem $S_k$, let $a_m$ be an activity in $S_k$ with the smallest finish time. Then, $a_m$ is included in ***some*** maximal-size subset of mutually compatible activities of $S_k$.

***Proof:*** We will prove using an ***exchange argument***.

Let $A_k$ be a maximal subset of mutually compatible activities in $S_k$ and let $a_j$ be the activity in $A_k$ with the smallest finish time.

***Case I:*** $a_m = a_j$

We are done because we have just shown that $a_m$ is included in some maximal subset, which is $A_k$.

# Activity Selection Problem

***Proof:* (continued)**

***Case II:*** $a_m \neq a_j$

[***Exchange Argument***]

We will show that optimality does not change by replacing $a_j$ in $A_k$ with $a_m$.

We know that $a_j$ is mutually compatible with the other activities that start after $a_j$ finishes since all activities in $A_k$ are mutually compatible.

$a_m$ is an activity with the smallest finish time so $f_m \leq f_j$.

Therefore, $a_m$ must also be mutually compatible with the activities in the set $A_k - \{a_j\}$ .

By construction, there is some mutually compatible set $A'_k = (A_k - \{a_j\}) \cup \{a_m\}$ .

Since $|A_k| = |A'_k|$, ***optimality remains***.

Hence, there is some maximal subset $A'_k$ of mutually compatible activities that includes the greedy choice $a_m$. ■

## Activity Selection Problem

Let us state the algorithm a bit more formally.
We will use $S$ to denote the set of activities that we have neither accepted nor rejected yet, and use $A$ to denote the set of accepted activities.

```
procedure ActivitySelection($S$)
    $A = \emptyset$
    while $S$ is not $\emptyset$ do
        choose $a_i \in A$ with the minimum finish time $f_i$.
        add $a_i$ to $A$
        delete all the activities from $S$ that are not compatible with $a_i$
    return $A$
```

## Activity Selection Problem

```
1: procedure ITERATIVE-ACTIVITY-SELECTION($s, f$)
2:     $n = s.length$
3:     $A = \{a_1\}$
4:     $k = 1$
5:     for $m = 2 \to n$ do
6:         if $s[m] \geq f[k]$ then
7:             $A = A \cup \{a_m\}$
8:             $k = m$
9:     return $A$
```

The algorithm takes arrays $s$ and $f$ storing start and finish times, respectively, sorted in ***non-decreasing*** order of finish times: ***sorting*** requires $O(n \log n)$ time.

The algorithm examines exactly $n - 1$ activities with the for loop in ***line 5.***

The body of the for loop (***lines 6-8***) takes $\Theta(1)$ time.

Therefore, the algorithm takes $(\mathrm{n} - 1) \cdot \Theta(1) = \Theta(n)$ time.

Therefore, the total time complexity is $O(n \log n)$ if sorting is taken into account.

## Minimum Spanning Tree

***Recall:*** A tree is a connected undirected acyclic graph.

A ***graph*** $G = (V, E)$ is connected if, for any pair of vertices $u, v \in V$, there exists a simple path connecting $u$ and $v$.

*A* ***tree*** is a connected graph where, for any pair of vertices $u, v \in V$, there exists exactly one simple path connecting $u$ and $v$.

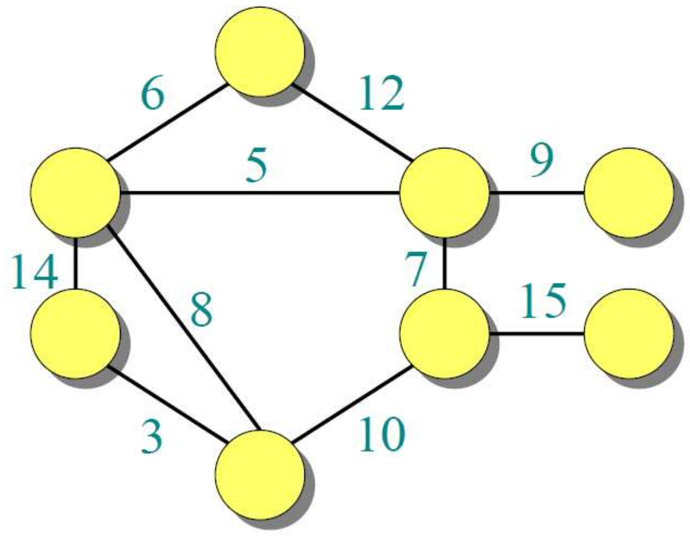
The number of edges of a tree is one smaller than the number of vertices, i.e., $|E| = |V| - 1$.

## Minimum Spanning Tree

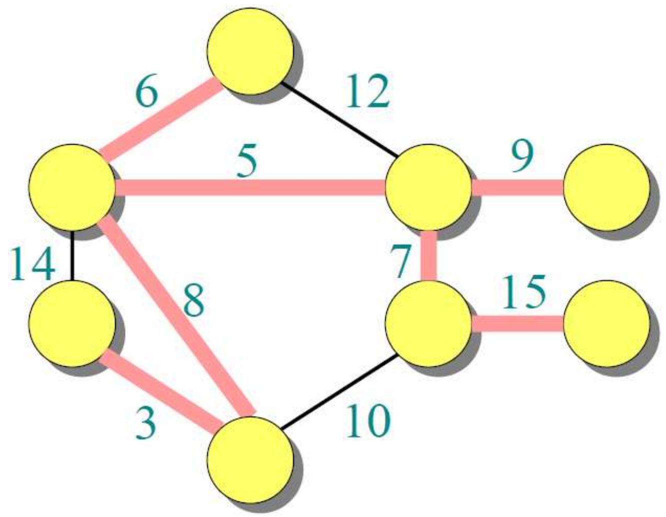
A ***spanning*** tree of a graph $G = (V, E)$ is a tree whose edges $A \subseteq E$ cover (***span***) every vertex in $V$.

The ***Minimum Spanning Tree (MST)*** problem is defined as follows:
Given a graph $G = (V, E)$ and weights $w: E \mapsto R$, compute a spanning tree $T$ with the minimum weight $\sum_{e \in T} w(e)$.

## Minimum Spanning Tree



Graph *G*

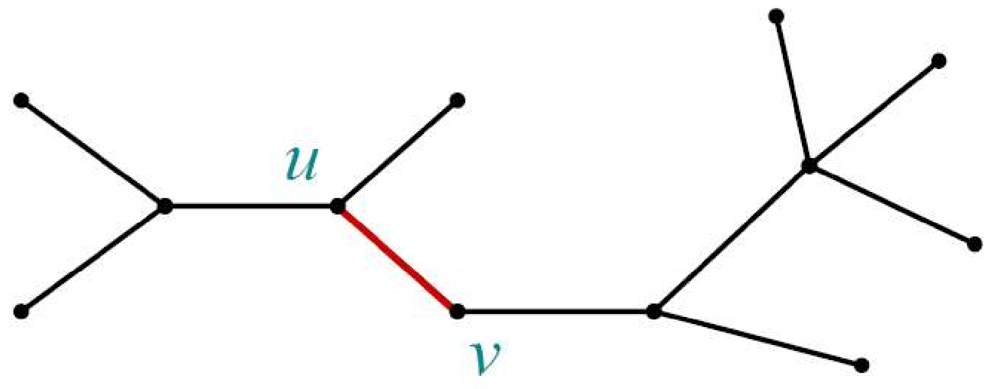


MST of *G*

# Minimum Spanning Tree

***Optimal Substructure***

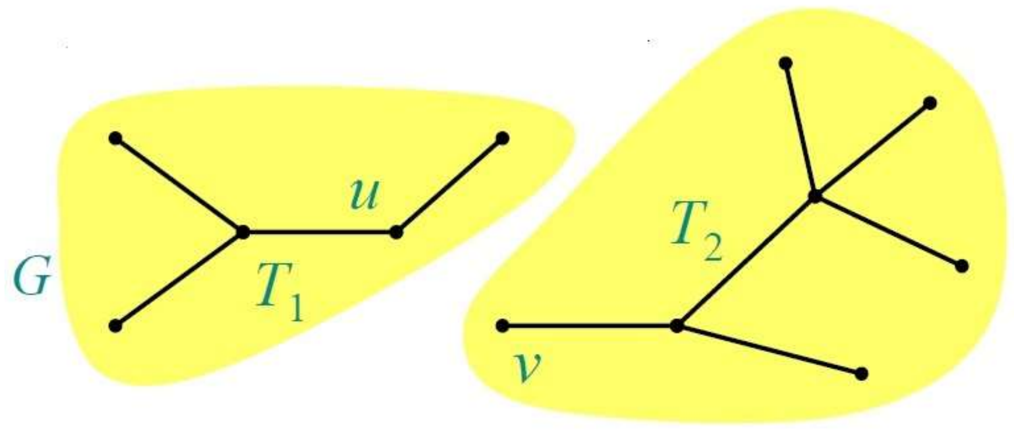
Suppose there is some MST $T^*$ of a graph $G = (V, E)$ and some edge $e = (u, v)$ belongs to $T^*$.



***Note:*** ***The other edges of*** $G$ ***that do not form the MST*** $T^*$ ***are omitted.***

## Minimum Spanning Tree

Removing $e$ partitions the MST $T^*$ into two subtrees $T_1$ and $T_2$.



# Minimum Spanning Tree

***Claim:*** The resulting subtree $T_1$ is an MST of $G_1 = (V_1, E_1)$, which is the subgraph of $G$ induced by $V_1$:

$$E_1 = \{(x, y) | x, y \in V_1\}$$

Similarly for $T_2$.

***Proof:*** We will use a ***cut-and-paste*** argument.

We have $w(T^*) = w(T_1) + w(T_2) + w(e)$.

Suppose for FPOC that $T_1$ is not an MST.

Then, there is some subtree $T'_1$with $w(T'_1) < w(T_1)$.

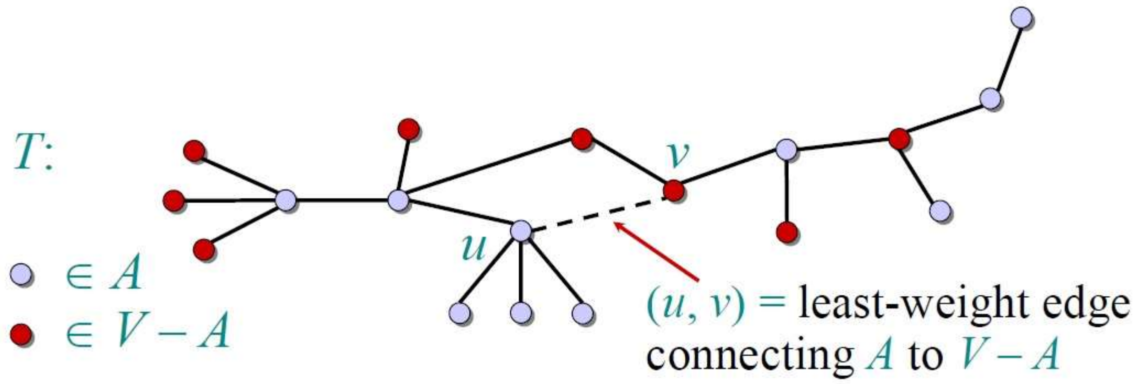
Replacing $T_1$ with $T'_1$ would result in a new spanning tree $T^{**}$ with $w(T^{**}) < w(T^*)$, which contradicts the optimality of $T^*$.

Symmetric argument applies to $T_2$. ■

## Minimum Spanning Tree

***Greedy-Choice Property:***

***Claim:*** Given any vertex cut $(A, V - A)$, suppose $e$ is an edge with the smallest weight that crosses the cut $(S, V - A)$: $e = (u, v)$ with $u \in A$ and $v \in V - A$. Then, $e$ is an edge in some MST.



## Minimum Spanning Tree

***Proof:*** We will use an ***exchange argument***.

Let $T^*$ be an MST of $G$.

If $e \in T^*$, we are done.

If $e \notin T^*$, we will show that we can modify $T^*$ to include $e$ without increasing the value of the total minimum weight to obtain that new MST.

## Minimum Spanning Tree

***Proof*: (Continued)**

Since $e = (u, v) \notin T^*$, there must be a simple path $p$ from $u$ to $v$.

Since $u$ and $v$ are on different sides of the cut, this means there must be at least an edge $e'$ on path $p$ that crosses the cut.

Therefore, we can ***safely*** exchange $e'$ for $e$.

The resulting graph $T^{**}$ is still a ***spanning tree:***

- the number of edges in $T^*$ remains the same so the condition $|E| = |V| - 1$ still holds
- $T^{**}$ is still connected
- $T^*$ still spans the entire vertex set $V$

## Minimum Spanning Tree

***Proof*: (Continued)**

Now we want to show that the new spanning tree $T^{**}$ is also an MST.

$$T^{**} = T^{*} - \{e'\} \cup \{e\}$$

Therefore, $w(T^{**}) = w(T^{*}) - w(e') + w(e)$

Since $w(e) \leq w(e')$,

$$w(T^{**}) \leq w(T^{*}).$$

Therefore, $w(T^{**}) = w(T^{*})$

$T^{**}$ is also an MST. ■

## Prim's Algorithm

***Prim's algorithm*** relies on the greedy-choice property we have just proven.

The ***vertex cut*** that Prim's algorithm computes and maintains is as follows:

$S$ is a set of vertices whose edges form a ***single tree*** $T_s$

$V - S$ is a set of vertices that have not been included as part of the tree $T_s$

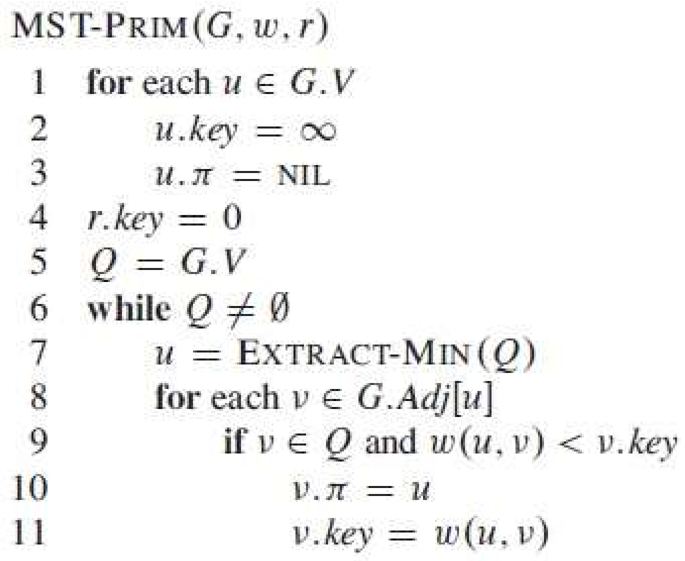
The algorithm starts with $S = \emptyset$, and therefore $V - S = V$.

## Prim’s Algorithm

Prim’s Algorithm uses a ***min priority queue*** $Q$ to maintain the invariant that we always pick an edge with the minimum-weight that crosses the cut.

Initially, we assign every vertex with key equal to infinity (***Line 2***), except for an arbitrarily chosen vertex $r$ whose key is set to $0$ (***Line 4***).

Therefore, $r$ will be picked first by removing it from $V - S$ and adding it to $S$.



```
MST-PRIM$(G, w, r)$
1  for each $u \in G.V$
2      $u.key = \infty$
3      $u.\pi = $ NIL
4  $r.key = 0$
5  $Q = G.V$
6  while $Q \neq \emptyset$
7      $u = $ EXTRACT-MIN$(Q)$
8      for each $v \in G.Adj[u]$
9          if $v \in Q$ and $w(u, v) < v.key$
10             $v.\pi = u$
11             $v.key = w(u, v)$
```

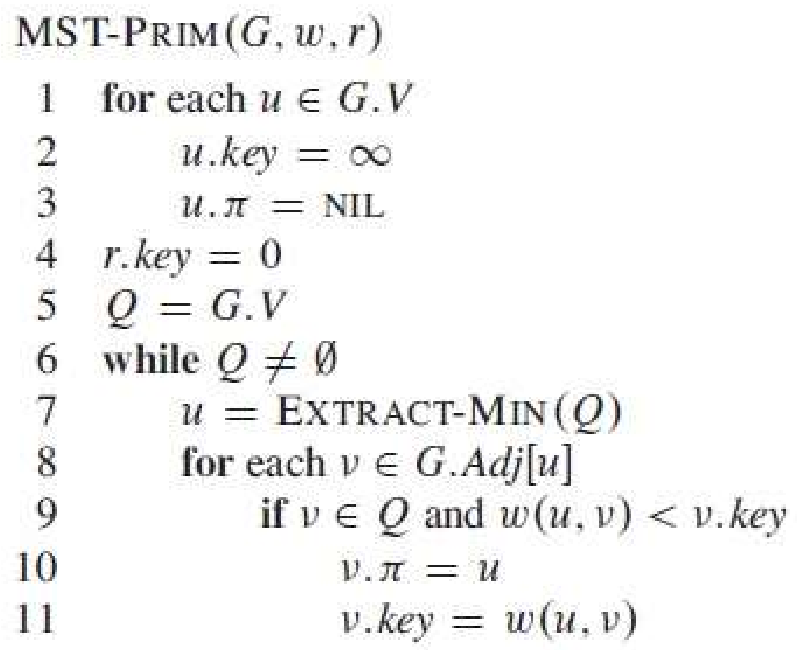
# Prim’s Algorithm

After removing $r$, we need to examine all the adjacent neighbors $v$ and only update their keys if they belong to $V - S$ and their current key value is less than $w(r, v)$.

Then, we keep extracting a minimum-weight crossing edge from the priority queue, updating their adjacent neighbors’ keys if necessary.

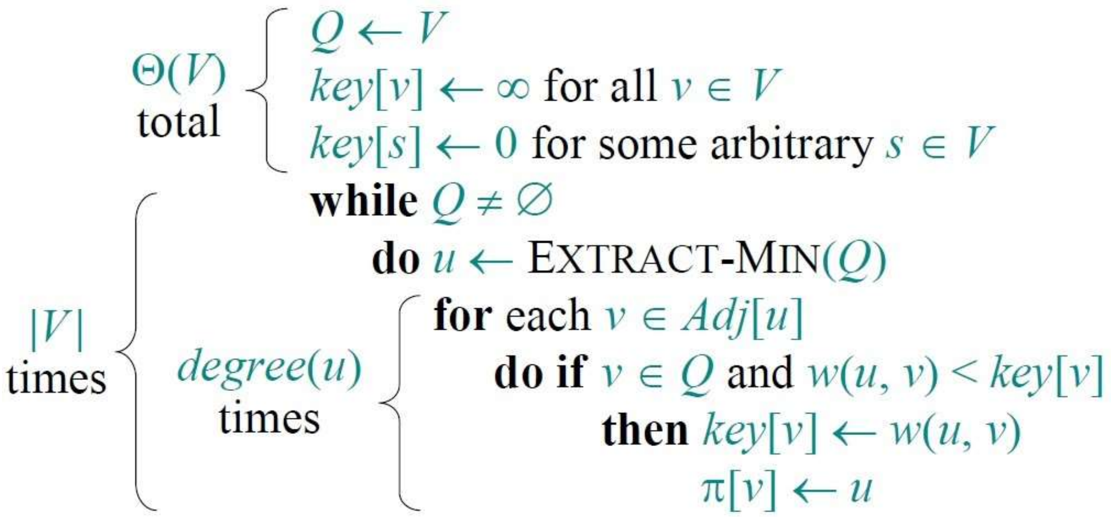
***Note:*** $v.key = w(u, v)$ is, in fact, a ***decrease-key*** operation.

The algorithm stops when there is no more edge to examine, that is, when the priority queue is empty. In other words, Prim’s Algorithm terminates when $V - S = \emptyset$ and $S = V$.



```
MST-PRIM($G, w, r$)
for each $u \in G.V$
    $u.key = \infty$
    $u.\pi =$ NIL
$r.key = 0$
$Q = G.V$
while $Q \neq \emptyset$
    $u =$ EXTRACT-MIN($Q$)
    for each $v \in G.Adj[u]$
        if $v \in Q$ and $w(u, v) < v.key$
            $v.\pi = u$
            $v.key = w(u, v)$
```

## Prim’s Algorithm



```
Θ(V) total {
    $Q \leftarrow V$
    $key[v] \leftarrow \infty$ for all $v \in V$
    $key[s] \leftarrow 0$ for some arbitrary $s \in V$
}
|V| times {
    while $Q \neq \emptyset$
        do $u \leftarrow$ EXTRACT-MIN($Q$)
        degree(u) times {
            for each $v \in Adj[u]$
                do if $v \in Q$ and $w(u, v) < key[v]$
                    then $key[v] \leftarrow w(u, v)$
                         $\pi[v] \leftarrow u$
        }
}
```

The running time = $\Theta(|V|)T_{Extract-Mi} + \Theta(|E|)T_{Decrease-Key}$

## Prim’s Algorithm

***Correctness:*** We need to show that the following ***invariant*** holds prior to the start of any iteration.

At the start of any iteration, the tree $T_s$ within $S$ is contained within some MST $T^*$ of $G$.

***Proof:*** We will prove by induction on the loop invariant.

***Initialization:*** Initially, $S = \emptyset$. The invariant trivially holds since the empty set is a trivial subset of any MST.

## Prim’s Algorithm

***Maintenance:*** Assuming the loop invariant holds at the current iteration, we have $T_s \subseteq T^*$.

Prim’s algorithm extracts a vertex $u$ with the ***minimum key value***, which corresponds to a edge $e = (v, u)$ among all edges that cross the cut.

Therefore, $T_s' = T_s \cup \{e\}$.
By the greedy-choice property, there is some MST $T^{**}$ that contains $e$.
If $e \in T^*$, we are done.
If $e \notin T^*$, we need to show that we can convert $T^*$ into $T^{**}$ and we have to make sure that $T^{**}$ contains both $T_s$ and $e$.

## Prim’s Algorithm

By the greedy-choice property, we can obtain $T^{**}$ by ***exchanging*** another crossing edge $e'$ for $e$.

***Observation:*** The exchanging process cannot not affect $T_s$ since $e'$ is a crossing edge so it cannot be any edge of $T_s$.

Therefore, we can construct $T^{**}$ that includes $T_s \cup \{e\}$.

This proves the invariant. ■ (Invariant holds)

***Termination:*** Up on termination $S = V$ so $\mathrm{T_s}$ spans the entire set of vertices $V$, meaning that $\mathrm{T_s}$ itself is an MST. ■ (Prim’s algorithm guarantees optimality)

## Summary

Today we have covered the topic of Greedy Algorithms with the help of the following optimization problems:

- ***Activity Selection Problem***
- ***Minimum Spanning Tree Problem***

There are two key properties we must show to guarantee correctness and optimality of a greedy algorithm:

- ***Optimal Substructure***
- ***Greedy-Choice Property***