Greedy Algorithms

- General principle of greedy algorithm
- Activity-selection problem
  - Optimal substructure
  - Recursive solution
  - Greedy-choice property
  - Recursive algorithm
- Minimum spanning trees
  - Generic algorithm
  - Definition: cuts, light edges, safe edges
  - Prim's algorithm

Overview

- Like dynamic programming (DP), used to solve optimization problems.
- Problems exhibit optimal substructure (like DP).
- Problems also exhibit the **greedy-choice** property.
  - » When we have a choice to make, make the one that looks best *right now*.
  - » Make a **locally optimal choice** in hope of getting a **globally optimal solution**.

Greedy Strategy

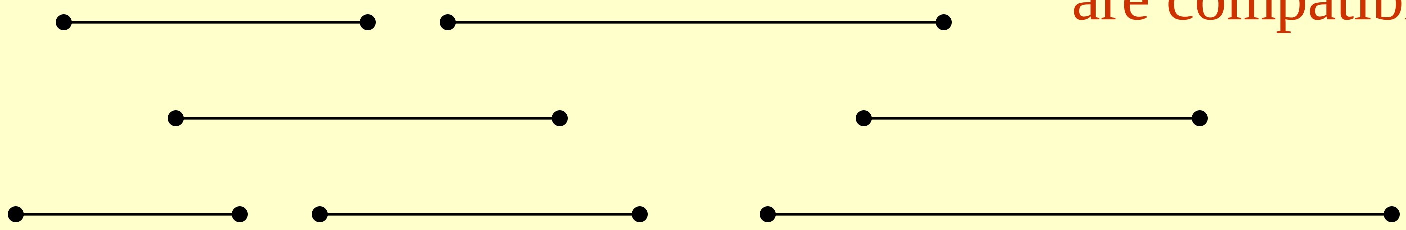
- The choice that seems best at the moment is the one we go with.
  - » Prove that when there is a choice to make, one of the optimal choices is the greedy choice. Therefore, it's always safe to make the greedy choice.
  - » Show that all but one of the subproblems resulting from the greedy choice are empty.

Activity-selection Problem

- Input: Set S of n activities, $a_1, a_2, \ldots, a_n$.
  - » s_i = start time of activity i.
  - » f_i = finish time of activity i.
- Output: Subset A of maximum number of compatible activities.
  - » Two activities are compatible, if their intervals don't overlap.

Example:

Activities in each line are compatible.



Example:

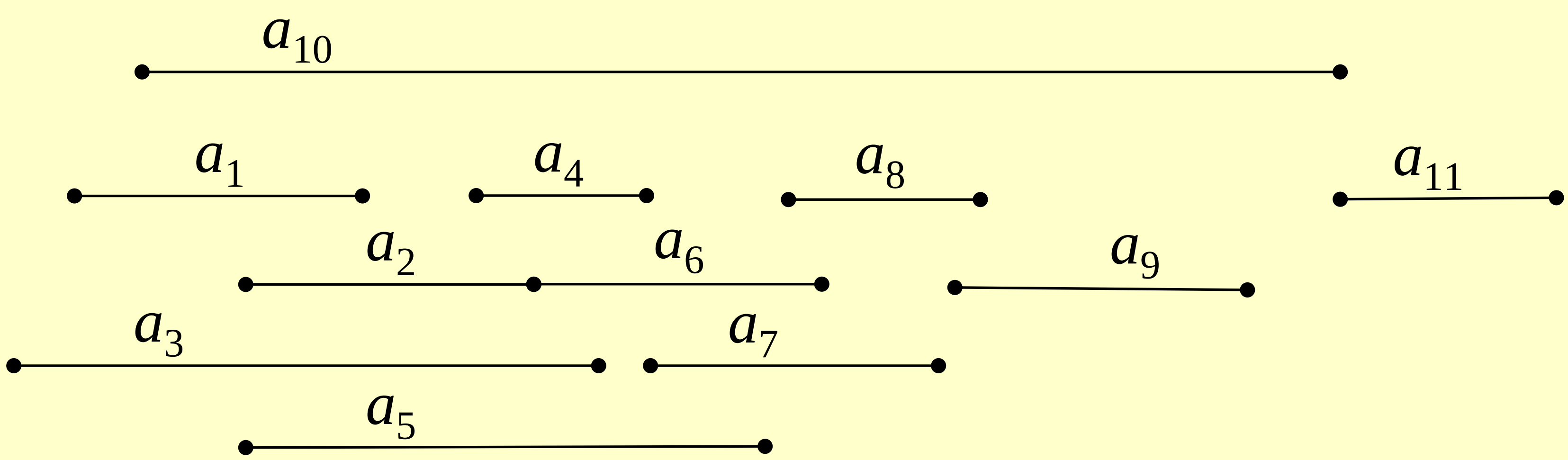
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	7	8	10	2	13
f_i	4	5	6	7	8	9	10	11	12	13	14

- $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities.

But it is not a maximal set.

- $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities.

Another largest subset is $\{a_2, a_6, a_9, a_{11}\}$.



Optimal Substructure

- Assume activities are sorted by finishing times.
  - » $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Suppose an optimal solution includes activity a_k.
  - » This generates two subproblems.
  - » Selecting from $a_1, \ldots, a_{k-1}$, activities compatible with one another, and that finish before a_k starts (compatible with a_k).
  - » Selecting from $a_{k+1}, \ldots, a_n$, activities compatible with one another, and that start after a_k finishes.
  - » The solutions to the two subproblems must be optimal.

Recursive Solution

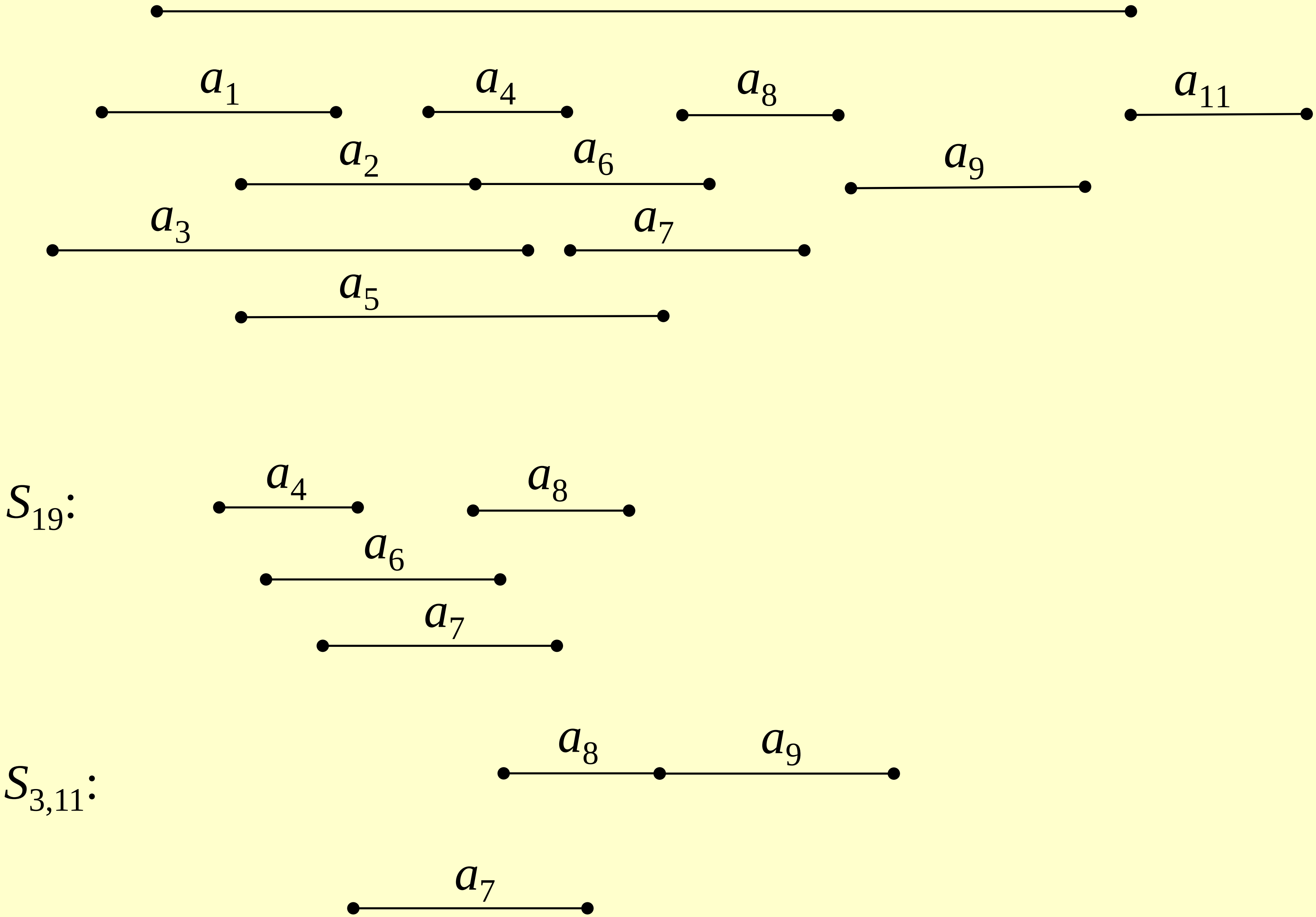
- Let S_{ij} = subset of activities in S that start after a_i finishes and finish before a_j starts.
- Subproblems: Selecting maximum number of mutually compatible activities from S_{ij}.
- Let $c[i, j]$ = size of maximum-size subset of mutually compatible activities in S_{ij}.

Recursive Solution:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \phi \\ \max\limits_{i<k<j}\{ c[i,k-1]+c[k+1,j]+1\} & \text{if } S_{ij} \neq \phi \end{cases}$$

The answer: $c[1, n]$

Running time: $O(n^3)$



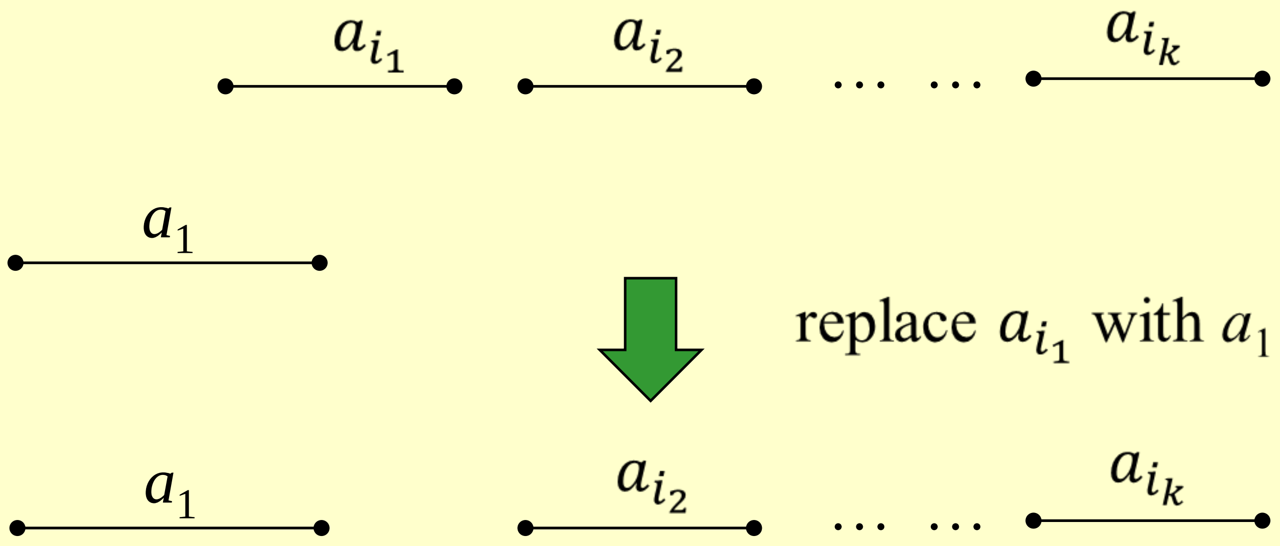
a_1
a_4
a_8
a_{11}
a_2
a_6
a_9
a_3
a_7
a_5
S_{19}:
a_4
a_8
a_6
a_7
$S_{3,11}$:
a_8
a_9
a_7

Greedy-choice Property

- The problem also exhibits the greedy-choice property.
  - » There is an optimal solution to the subproblem S_{ij}, that includes the activity with the smallest finish time in set S_{ij}.
  - » Can be proved easily.
- Hence, there is an optimal solution to S that includes a_1.
- Therefore, make this greedy choice without solving subproblems first and evaluating them.
- Solve the subproblem that ensues as a result of making this greedy choice.
- Combine the greedy choice and the solution to the subproblem.

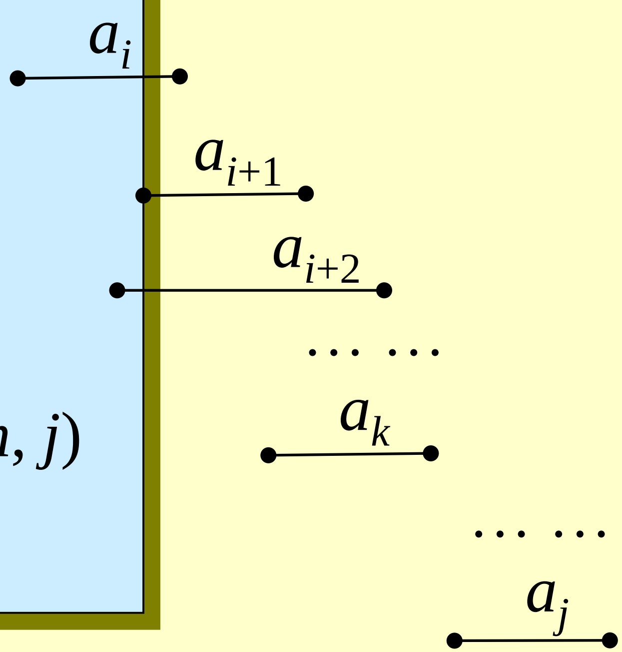
Greedy choice property:

Assume that $\{a_{i_1}, \ldots, a_{i_k}\}$ be a maximal set of compatible activities. Then, $\{a_1, \ldots, a_{i_k}\}$ must be a maximum set of compatible activities.



Recursive Algorithm

```
Recursive-Activity-Selector (s, f, i, j)
    m ← i + 1
    while m < j and s_m < f_i
        do m ← m + 1
    if  m < j
        then return {a_m} ∪
                Recursive-Activity-Selector(s, f, m, j)
        else return ϕ
```

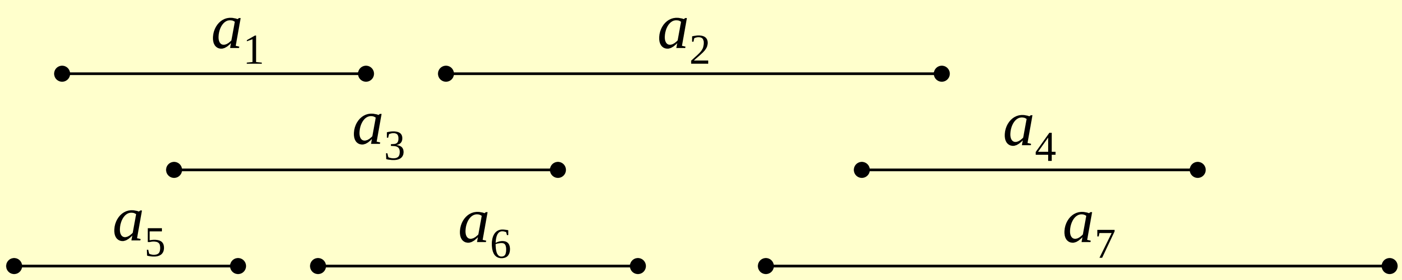


Initial Call: Recursive-Activity-Selector (s, f, 0, n + 1)

Complexity: $\Theta(n)$

Straightforward to convert the algorithm to an iterative one.
See the text.

Example:



sorted sequence according to *f* values:

$$a_5 \longrightarrow a_1 \longrightarrow a_3 \longrightarrow a_6 \longrightarrow a_2 \longrightarrow a_4 \longrightarrow a_7$$

$$a_5 \longrightarrow a_1 \longrightarrow a_3 \longrightarrow a_6 \longrightarrow a_2 \longrightarrow a_4 \longrightarrow a_7$$

step 1:

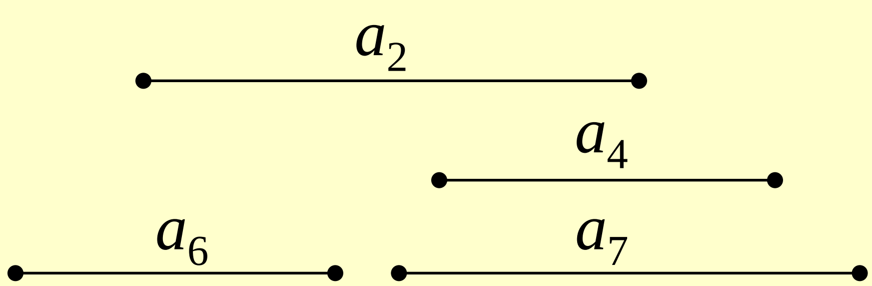
result = $\{a_5\}$. Removed all those activities not compatible with a_5.

$$a_6 \longrightarrow a_2 \longrightarrow a_4 \longrightarrow a_7$$

step 2:

result = $\{a_5, a_6\}$. Removed all those activities not compatible with a_6.

$$a_4 \longrightarrow a_7$$



step 3:

result = $\{a_5, a_6 , a_4\}$.

Typical Steps

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
- Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.
- Make the greedy choice and **solve top-down**.
- May have to preprocess input to put it into greedy order.
  - » Example: Sorting activities by finish time.

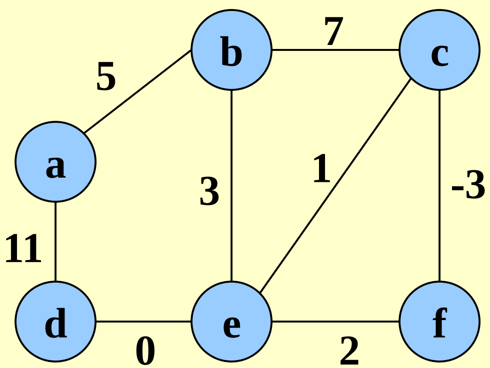
Elements of Greedy Algorithms

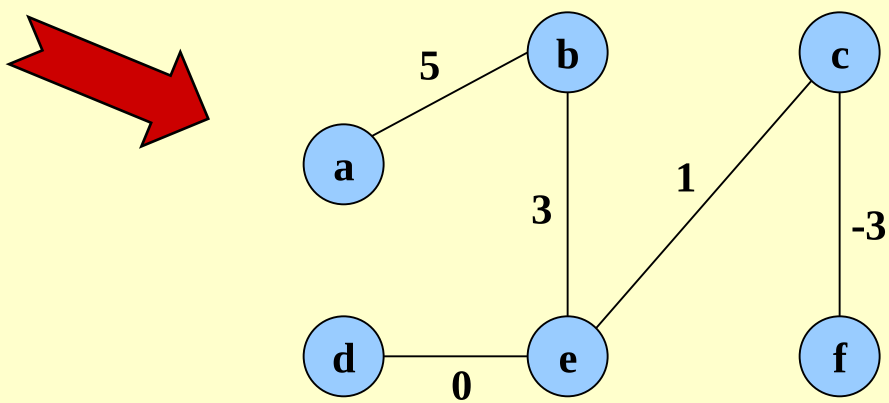
- Greedy-choice Property.
  - » A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- Optimal Substructure.

Minimum Spanning Trees

- **Given:** Connected, undirected, weighted graph, *G*
- **Find:** Minimum - weight spanning tree, *T*
- **Example:**

Acyclic subset of edges(*E*) that connects all vertices of *G*.



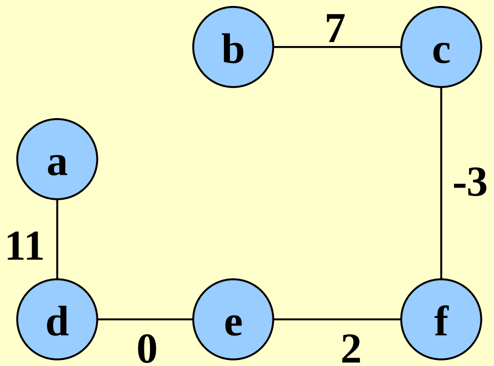


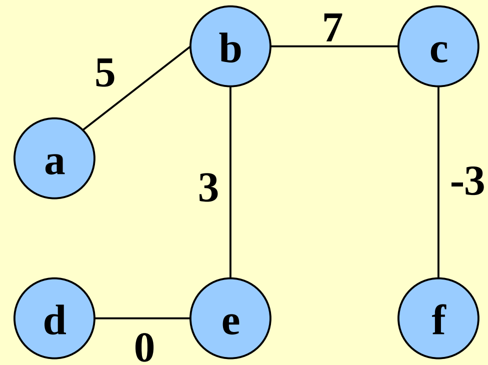
weight of *T*:

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

Minimum Spanning Trees

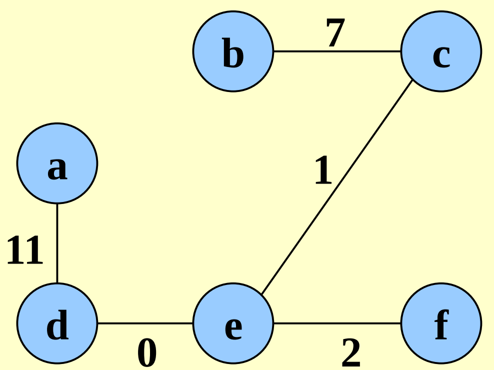
17

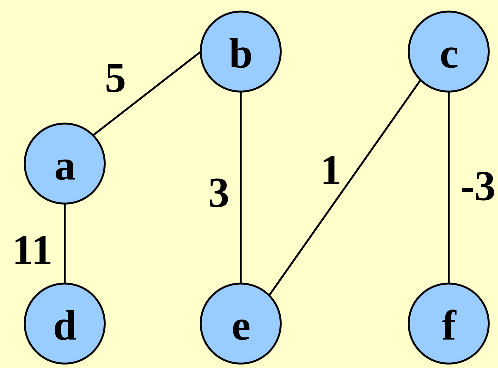




12

21



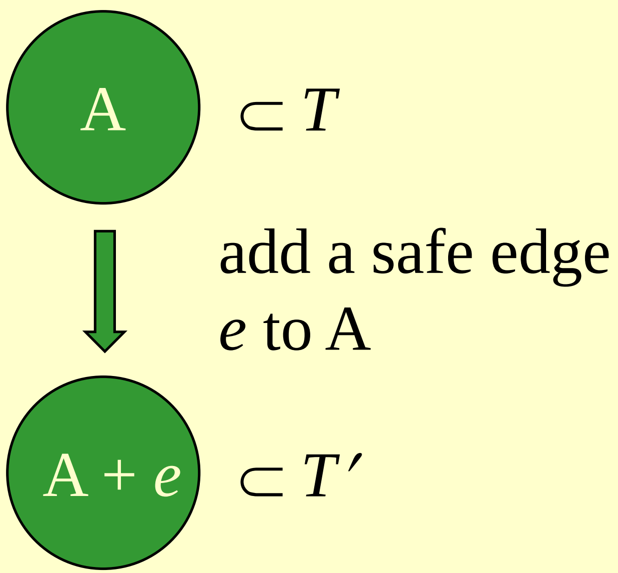


17

Generic Algorithm

- *A* - subset of some Minimum Spanning tree (MST).
- "Grow" *A* by adding "safe" edges one by one.

- Edge is "safe" if it can be added to *A* without destroying this invariant.



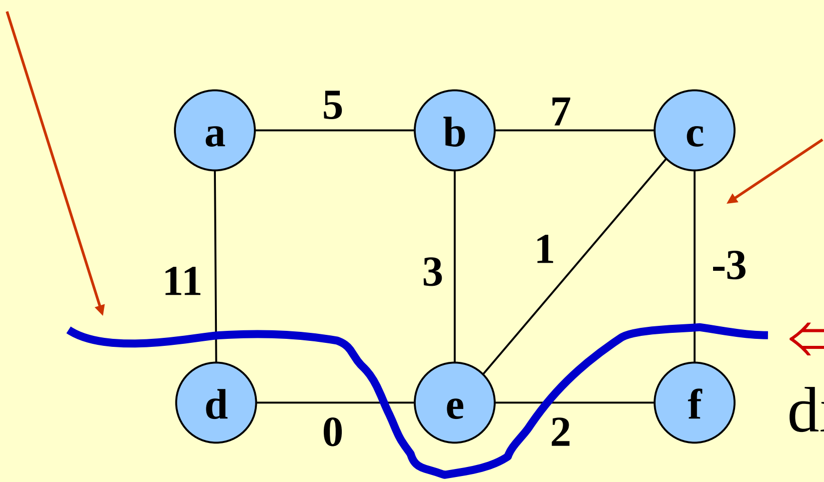
$A := \emptyset$;
while *A* not complete tree **do**
    find a safe edge (u, v);
    $A := A \cup \{(u, v)\}$
od

T' may be different from T.

Definitions

- Cut – A cut (*S*, *V* – *S*) of an undirected graph *G* = (*V*, *E*) is a partition of *V*.
- A cut respects a set *A* of edges if no edge in *A* crosses the cut.
- An edge is a light edge crossing a cut if its weight is the minimum of any edge crossing the cut.

cut that **respects** an edge set *A* = {(a, b), (b, c)}



a **light** edge crossing cut (could be more than one)

⇐ **cut** partitions vertices into disjoint sets, *S* and *V* – *S*.

Theorem 23.1

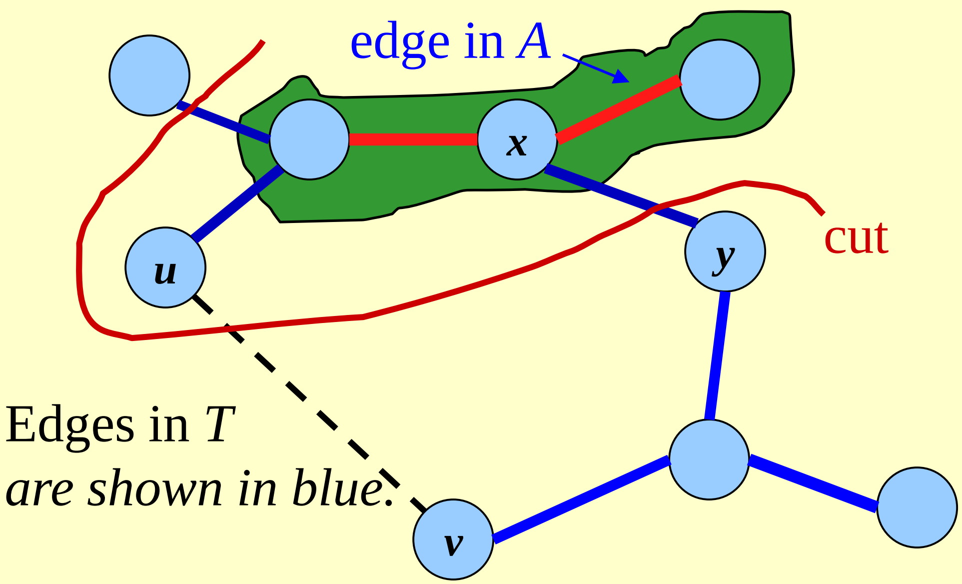
Theorem 23.1: Let $(S, V - S)$ be any cut that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then, (u, v) is safe for A.

Proof:

Let T be an MST that includes A.

Case 1: (u, v) in T. We're done.

Case 2: (u, v) not in T. We have the following:



(x, y) (in T) crosses cut.

Let $T' = \{T - \{(x, y)\}\} \cup \{(u, v)\}$.

Because (u, v) is light for cut,

$w(u, v) \leq w(x, y)$. Thus,

$w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$.

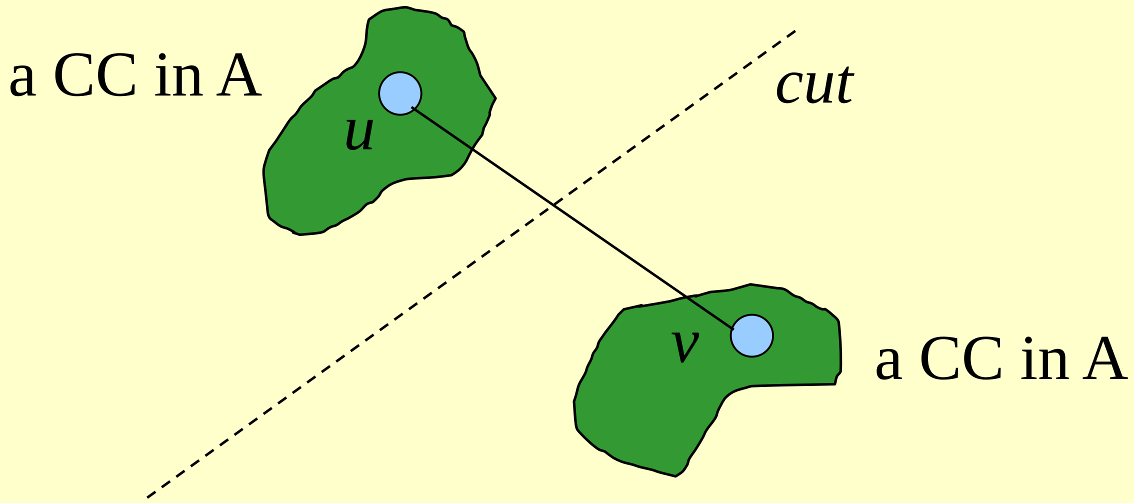
Hence, T' is also an MST.

So, (u, v) is safe for A.

Corollary

In general, *A* will consist of several connected components (CC).

Corollary: If (u, v) is a light edge connecting one CC in $G_A = (V, A)$ to another CC in G_A, then (u, v) is safe for *A*.

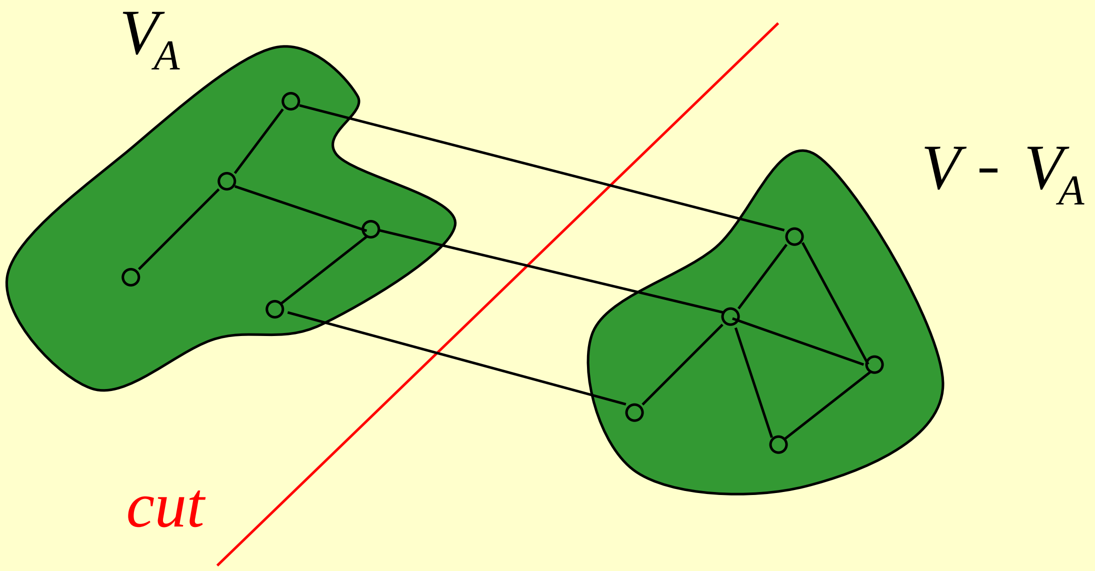


Kruskal's Algorithm

- Starts with each vertex in its own component.
- Repeatedly merges two components into one by choosing a light edge that connects them (i.e., a light edge crossing the cut between them).
- Scans the set of edges in monotonically increasing order by weight.
- Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.

Prim's Algorithm

- Builds **one tree**. So A is always a tree.
- Starts from an arbitrary "root" r.
- At each step, **adds a light edge** crossing cut $(V_A, V - V_A)$ to A.
  - » V_A = vertices that A is incident on.

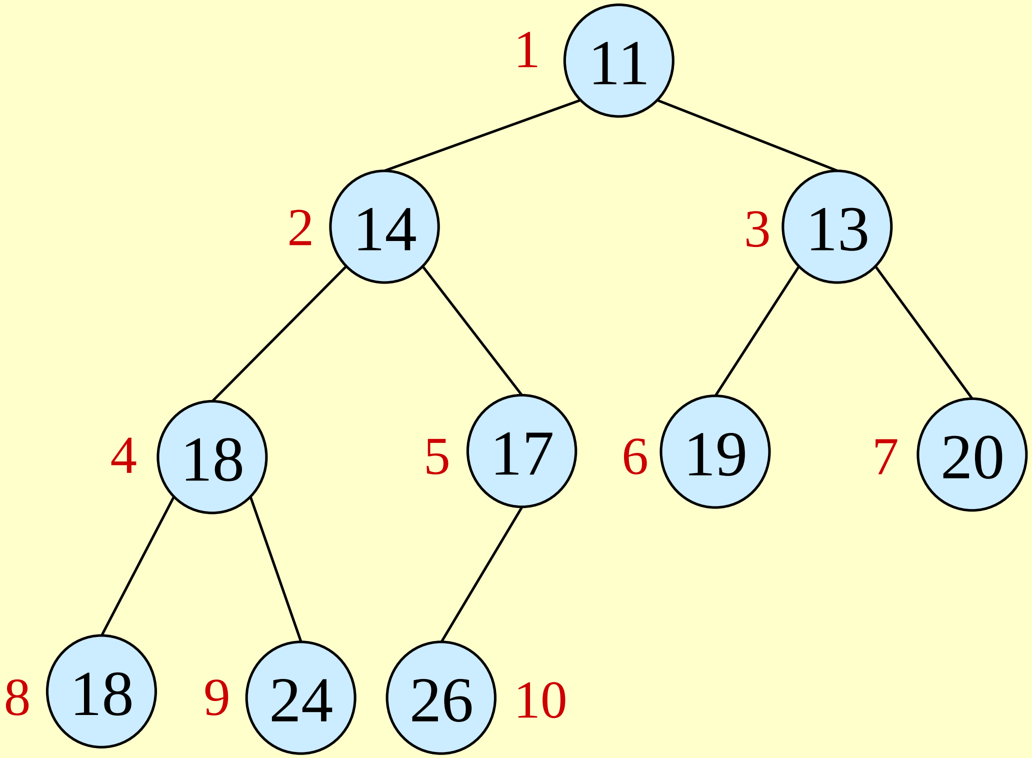


Prim's Algorithm

- Uses a **priority queue *Q*** to find a light edge quickly.
- Each object in Q is a vertex in $V - V_A$.

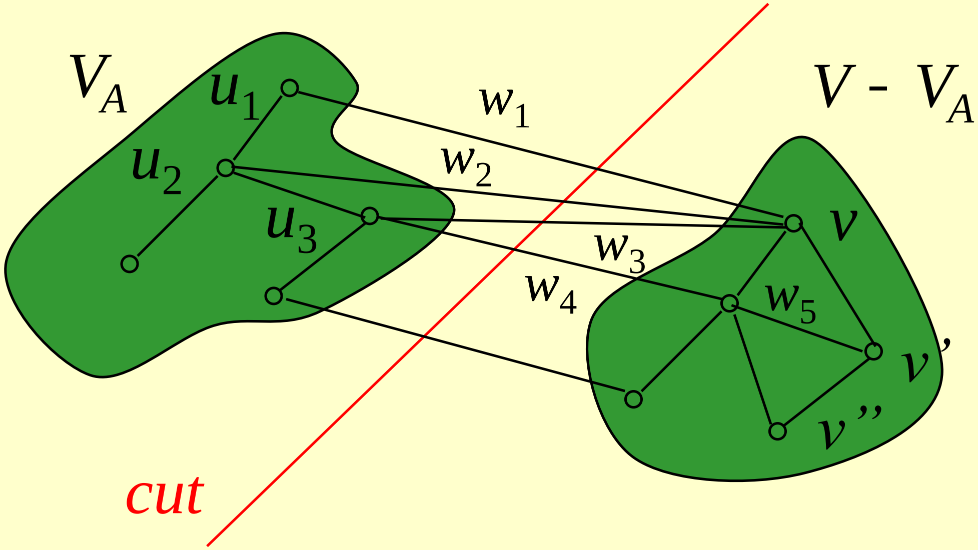
implemented as a min-heap

Min-heap as a binary tree.



Prim's Algorithm

- *key*(v) (key of $v \in V - V_A$) is minimum weight of any edge (u, v), where $u \in V_A$.
- Then the vertex returned by Extract-Min operation is v such that there exists $u \in V_A$ and (u, v) is light edge crossing (V_A, $V - V_A$).
- *key*(v) is ∞ if v is not adjacent to any vertex in V_A.



$key(v) = min\{w_1, w_2, w_3\}$

$key(v') = key(v'') = \infty$

Prim's Algorithm

```
Q := V[G];
for each u ∈ Q do
      key[u] := ∞
od;
key[r] := 0;
π[r] := NIL;
while Q ≠ ∅ do
      u := Extract-Min(Q);
      for each v ∈ Adj[u] do
            if v ∈ Q ∧ w(u, v) < key[v] then
                  π[v] := u;
                  key[v] := w(u, v)
            fi
      od
od
```

← decrease-key operation

Complexity:

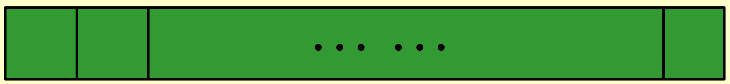
Using binary heaps: $O(E \lg V)$.

- Initialization – $O(V)$.
- Building initial queue – $O(V)$.
- V Extract-Min's – $O(V \lg V)$.
- E Decrease-Key's – $O(E \lg V)$.

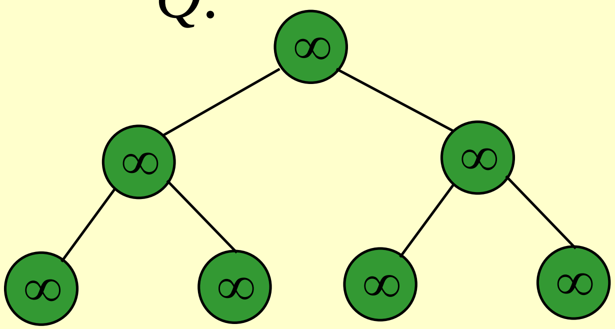
Using min-heaps: $O(E + V\lg V)$.
(see book)

Note: $A = \{(\pi[v], v) : v \in V - \{r\} - Q\}$.

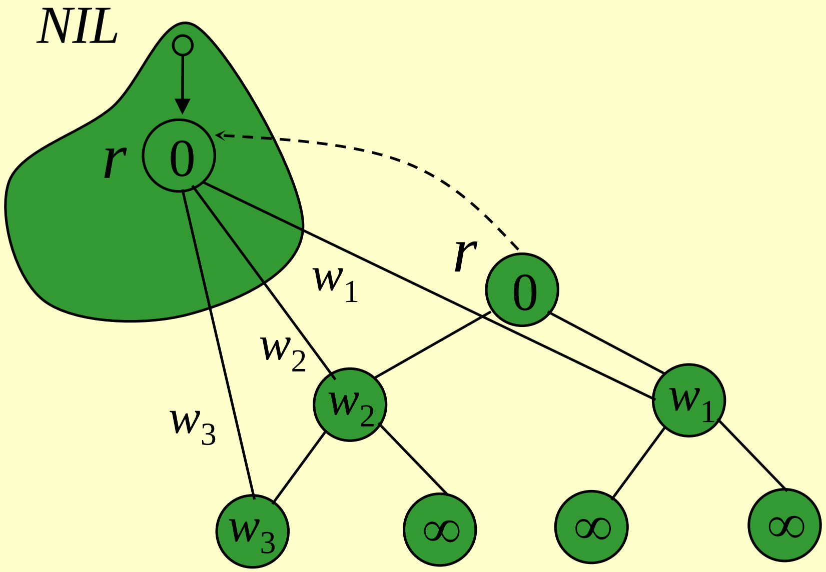
Q:

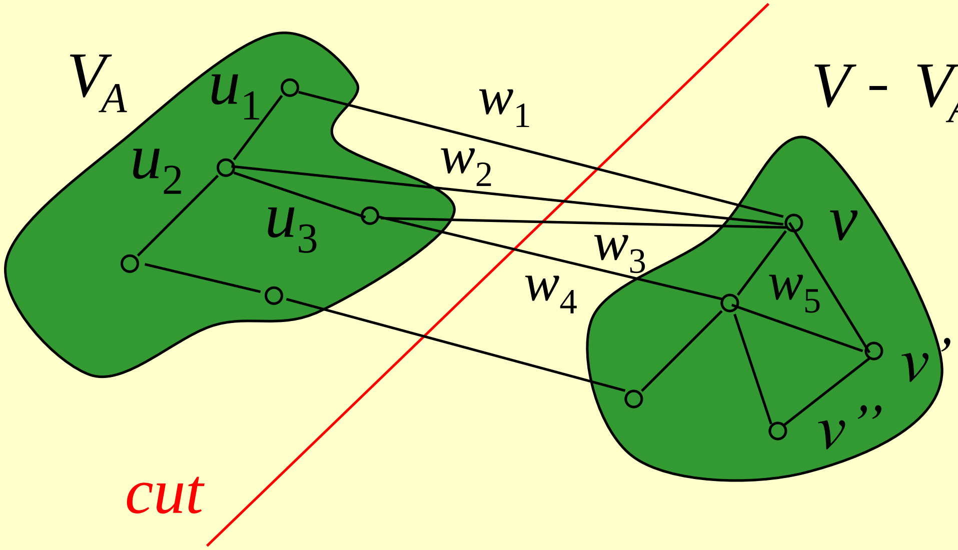


Q:



A:



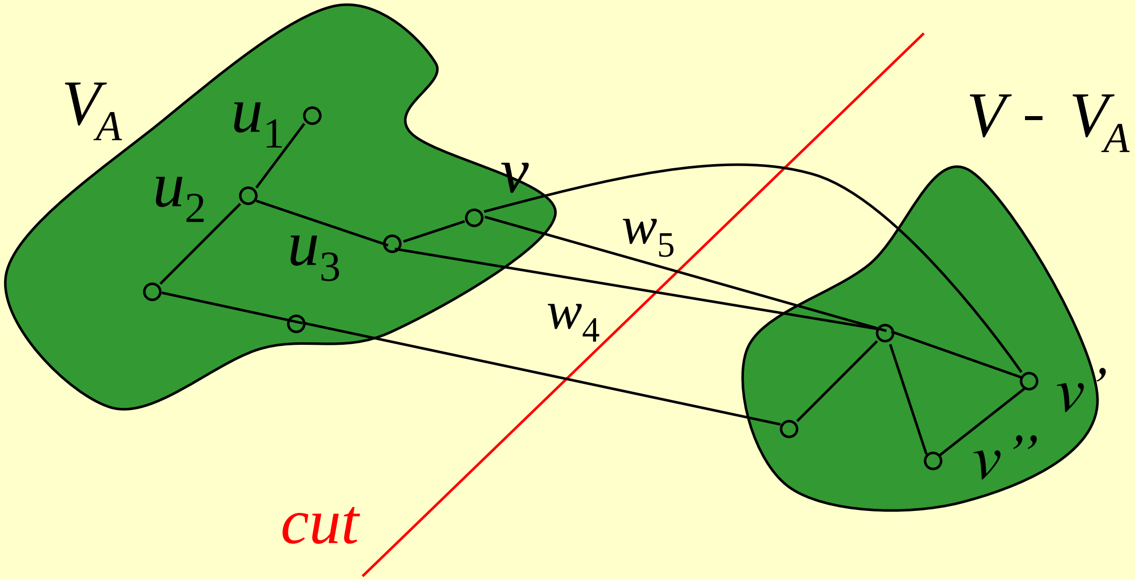


$key(v) = min\{w_1, w_2, w_3\}$

$key(v') = key(v'') = \infty$

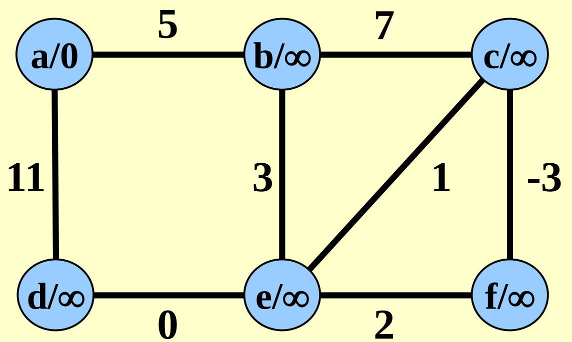
Assume that u_3 is u, chosen by the *extract-min* operation.
$key(v)$ should be changed:

$$key(v) \leftarrow \min\{key(v), w_3\}.$$



V_A
u_1
u_2
u_3
v
V - V_A
w_5
w_4
v'
v''
cut

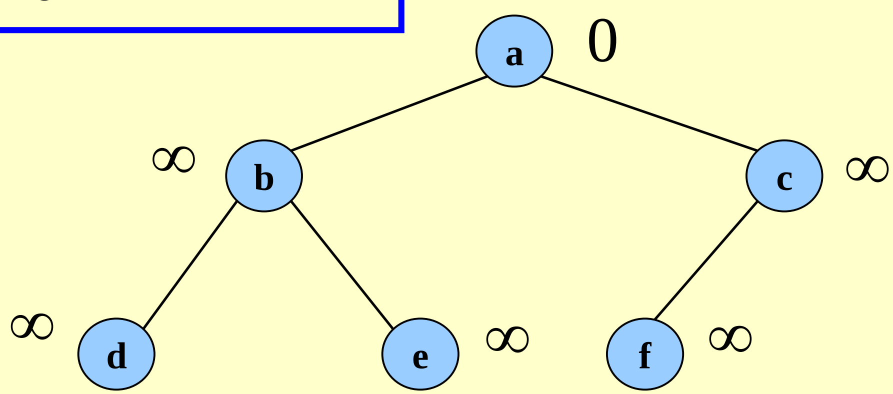
Example of Prim's Algorithm



Not in tree

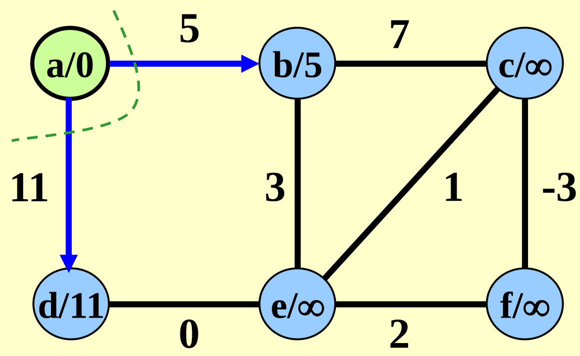
Q = a b c d e f

0 ∞ ⟷ ∞

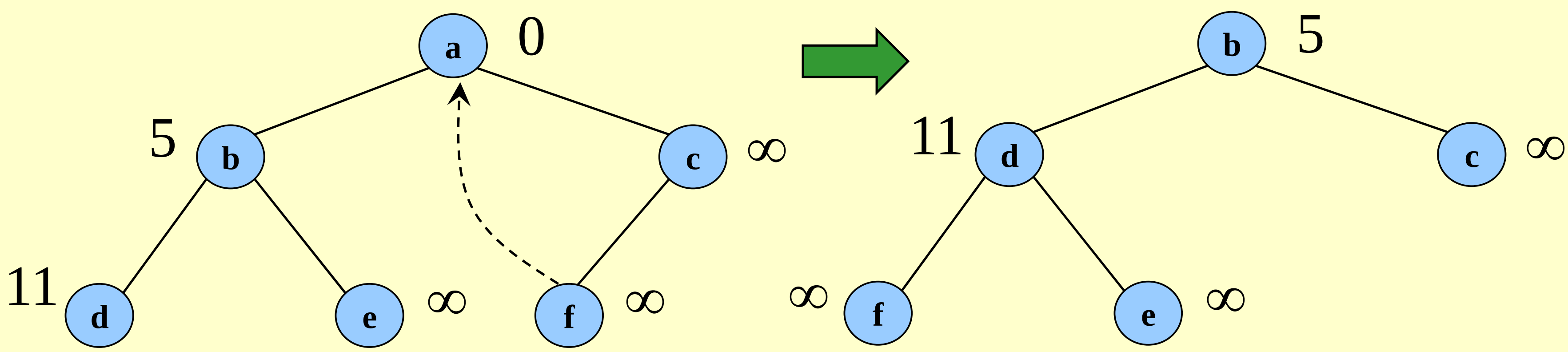


$Q := V[G]$;
for each $u \in Q$ **do**
    $key[u] := \infty$
od;
$key[r] := 0$;
$\pi[r] := NIL$;
while $Q \neq \varnothing$ **do**
    $u :=$ Extract-Min(Q);
    for each $v \in Adj[u]$ **do**
        if $v \in Q \wedge w(u, v) < key[v]$
    then
            $\pi[v] := u$;
            $key[v] := w(u, v)$
        fi
    od
od

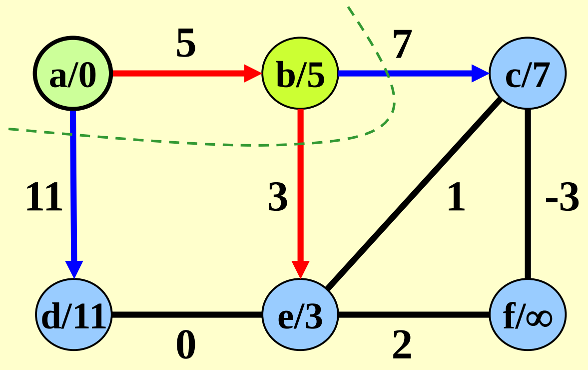
Example of Prim’s Algorithm



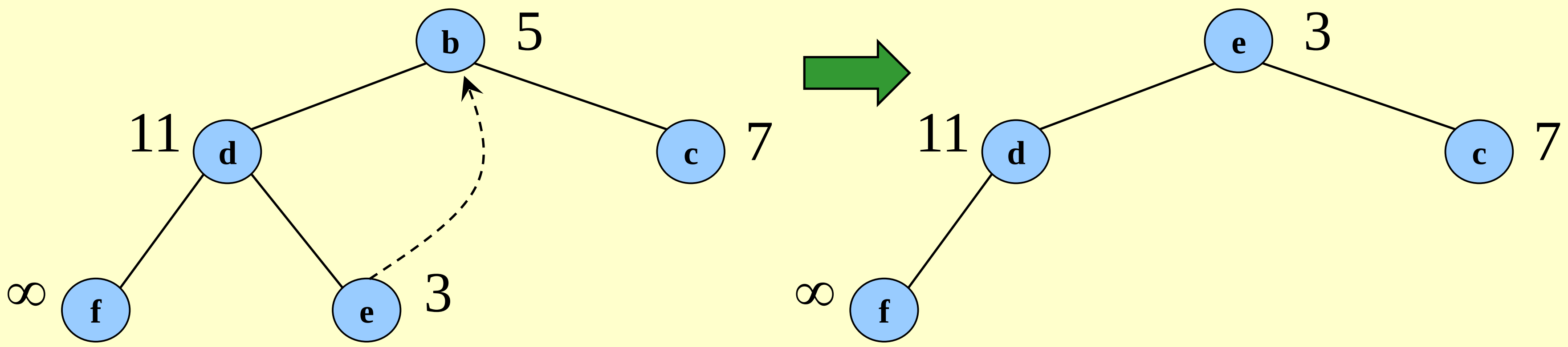
Q = b d c e f
5 11 ∞ ↔ ∞



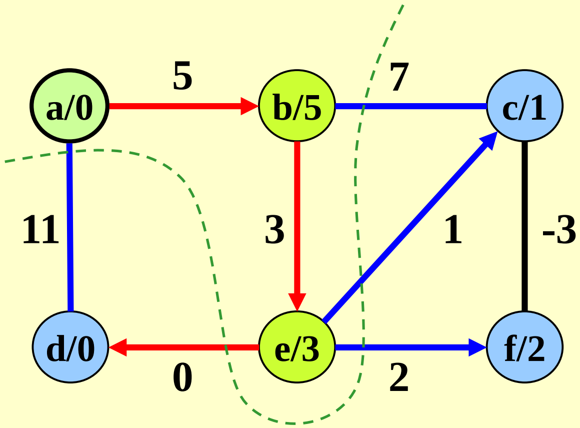
Example of Prim's Algorithm



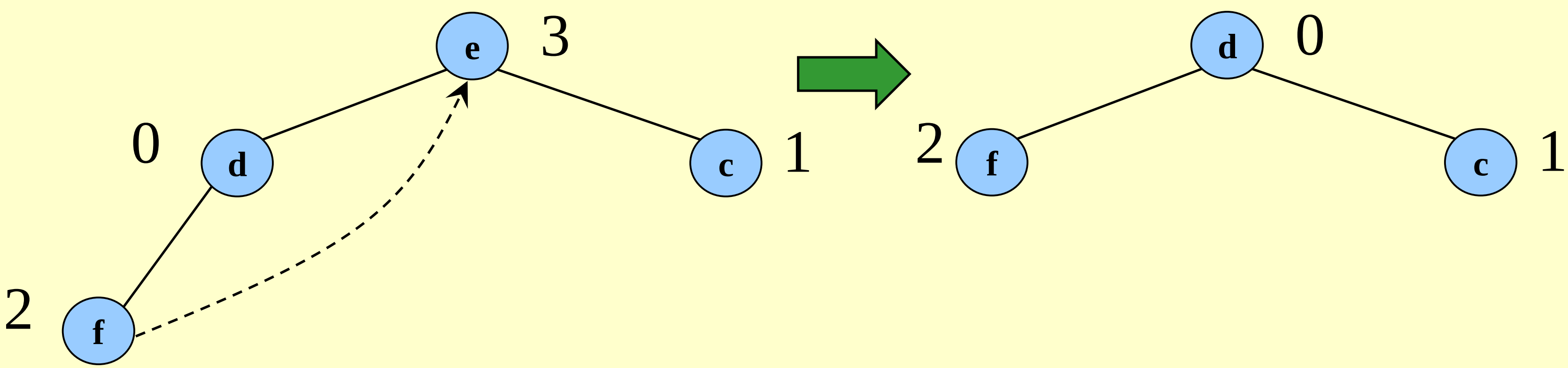
Q = e c d f
3 7 11 ∞



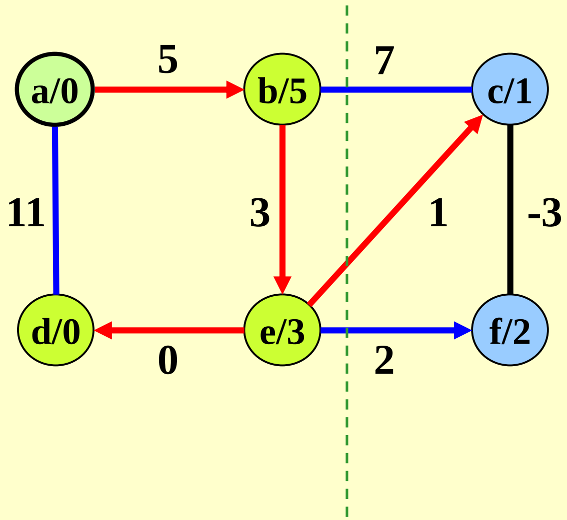
Example of Prim’s Algorithm



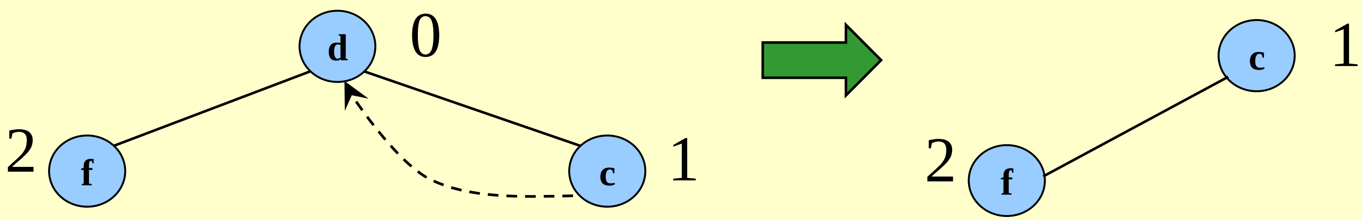
Q = d c f
0 1 2



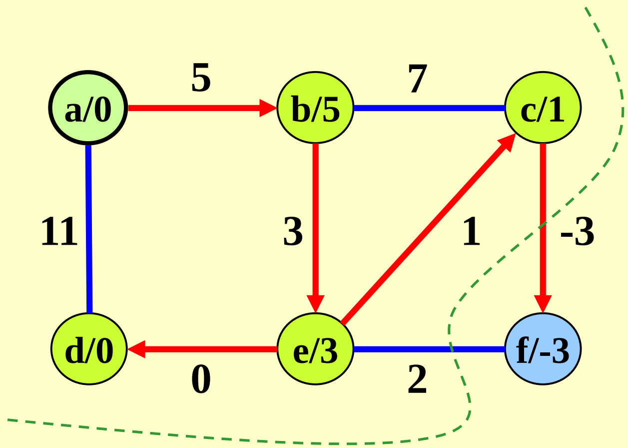
Example of Prim's Algorithm



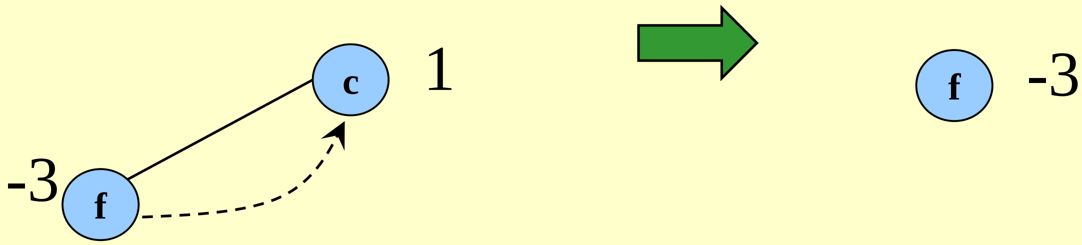
Q = c f
      1 2



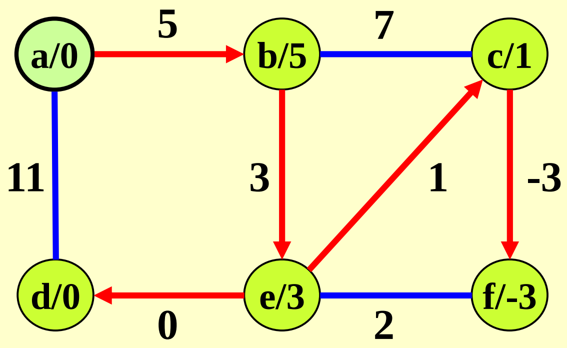
Example of Prim's Algorithm



Q = f
-3



Example of Prim's Algorithm



$Q = \varnothing$

Example of Prim's Algorithm