Analysis and Design of Algorithms

CS3230

Week 9

Greedy Algorithms

Arnab Bhattacharyya & Ken Sung

Admin

- Programming Assignment 2 OUT
  - Submit written part to the box in undergrad office COM1-02-19 by Thurs Apr 11 before 6 pm (next week)
  - Submit online part as usual
- Midterms will be handed back in tutorials next week

Recap of last lecture

- Dynamic programming – algorithm design pattern for problems displaying optimal substructure and overlapping subproblems

- Examples:
  - Shortest path in a DAG
  - Longest common subsequence in a pair of strings
  - Knapsack problem

Shortest Path Lengths in DAGs

initialize all dist(·) values to ∞

$dist(s) \leftarrow 0$

for $v \in V \setminus \{s\}$ in linearized order:

$\quad dist(v) = \min\{dist(u) + \ell(u,v) : (u,v) \in E(G)\}$

- There are n many subproblems $\{dist(v) : v \in V\}$
- Subproblems processed in the topological order, and each subproblem only needs solutions to previous ones
- Running time: $\cancel{O(n)}$ $O(|E| + |V|)$

Recursive Solution

Let $m[i, w]$ be the maximum value that can be obtained using:

- a subset of items in $\{1,2, \ldots, i\}$
- with total weight no more than w

$w \geq w_i$

$$m[i, w] = \begin{cases} 0, & \text{if } i = 0 \text{ or } w = 0 \\ \max\{m[i-1, w-w_i] + v_i, m[i-1, w]\}, & \cancel{\leq w_i} \\ m[i-1, w], & \text{otherwise} \end{cases}$$

Today: Greedy Algorithms

A very general technique, like divide-and-conquer and dynamic programming

Technique is to recast the problem so that only **one** subproblem needs to be solved at each step. Beats divide-and-conquer and dynamic programming, **when it works**.

Today: Greedy Algorithms

- Fractional Knapsack problem

- Activity Selection problem

- Prim’s algorithm for Minimum Spanning Tree

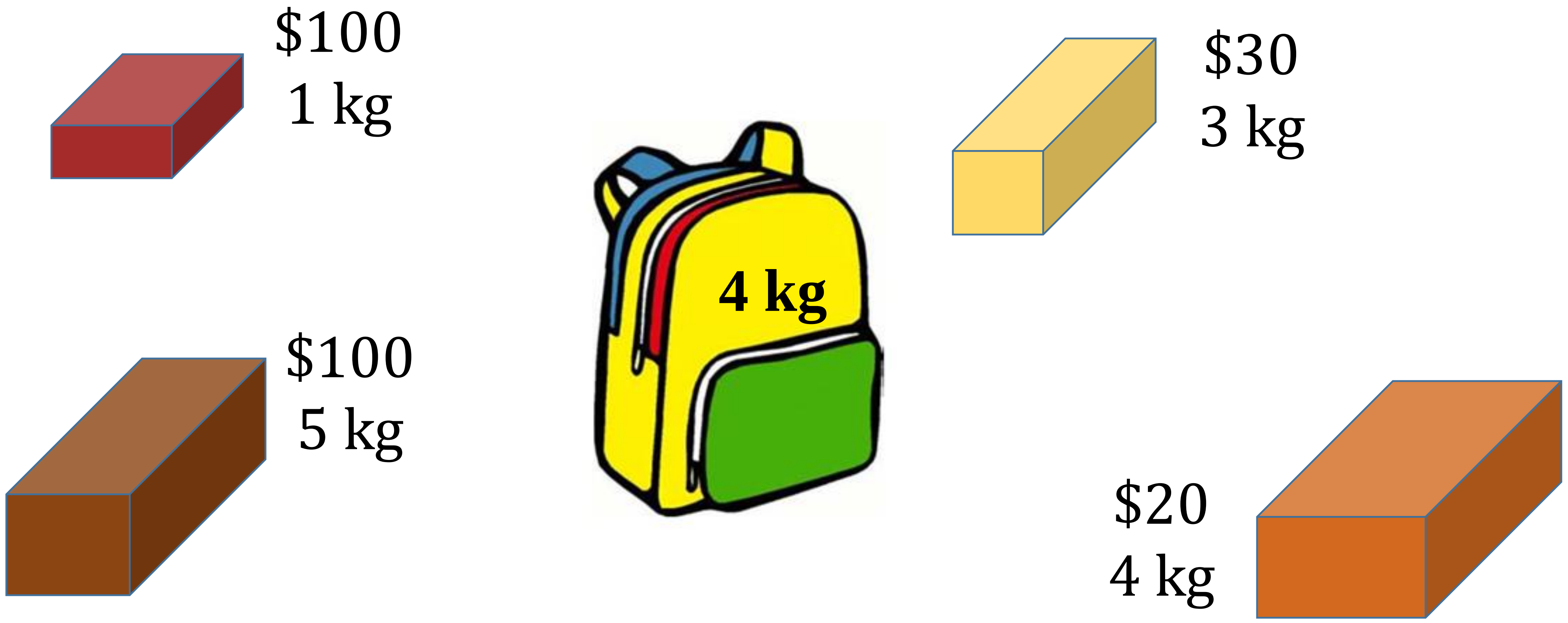
Today: Greedy Algorithms

- **Fractional Knapsack problem**

- Activity Selection problem

- Prim’s algorithm for Minimum Spanning Tree

Fractional Knapsack



Fractional Knapsack

Input:

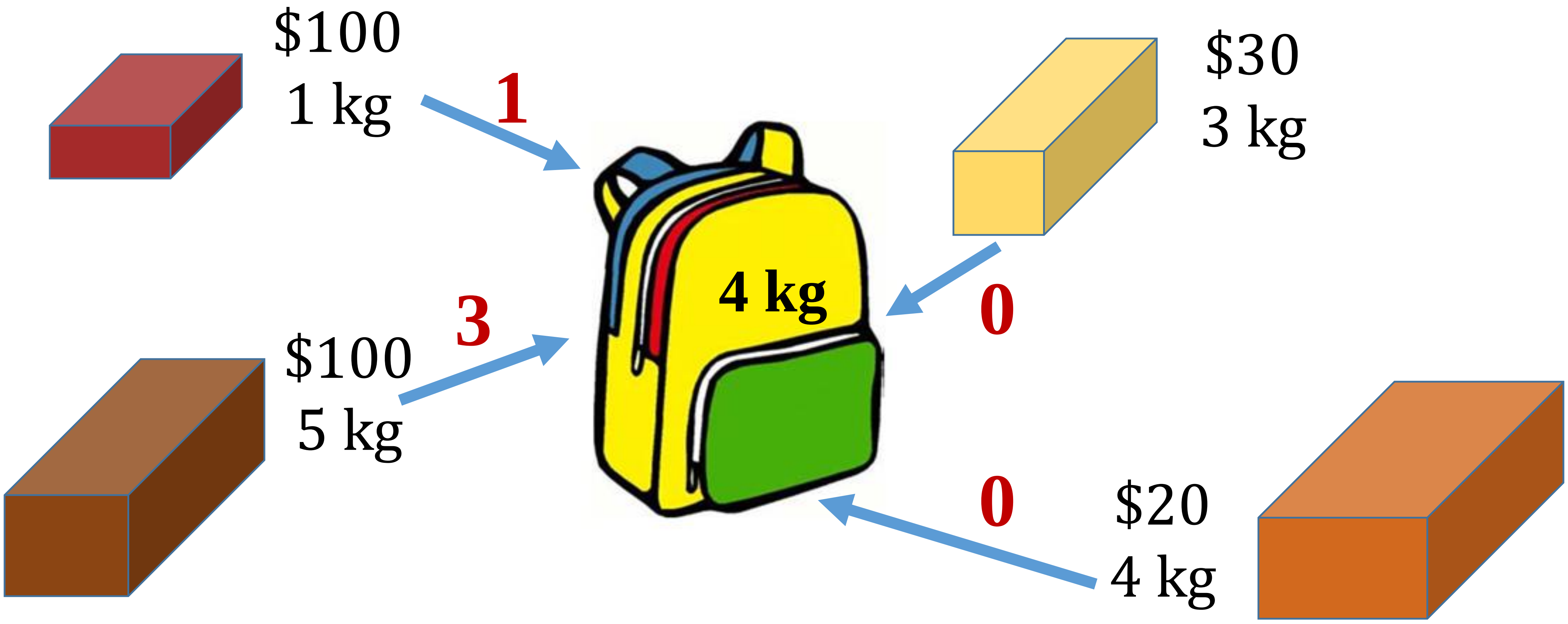
$(w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n)$ and T

Output:

Weights $x_1, \ldots, x_n$ that maximize $\sum_i v_i \cdot \frac{x_i}{w_i}$ subject to:

$$\sum_i x_i \leq T \text{ and } 0 \leq x_j \leq w_j \text{ for all } j \in [n].$$

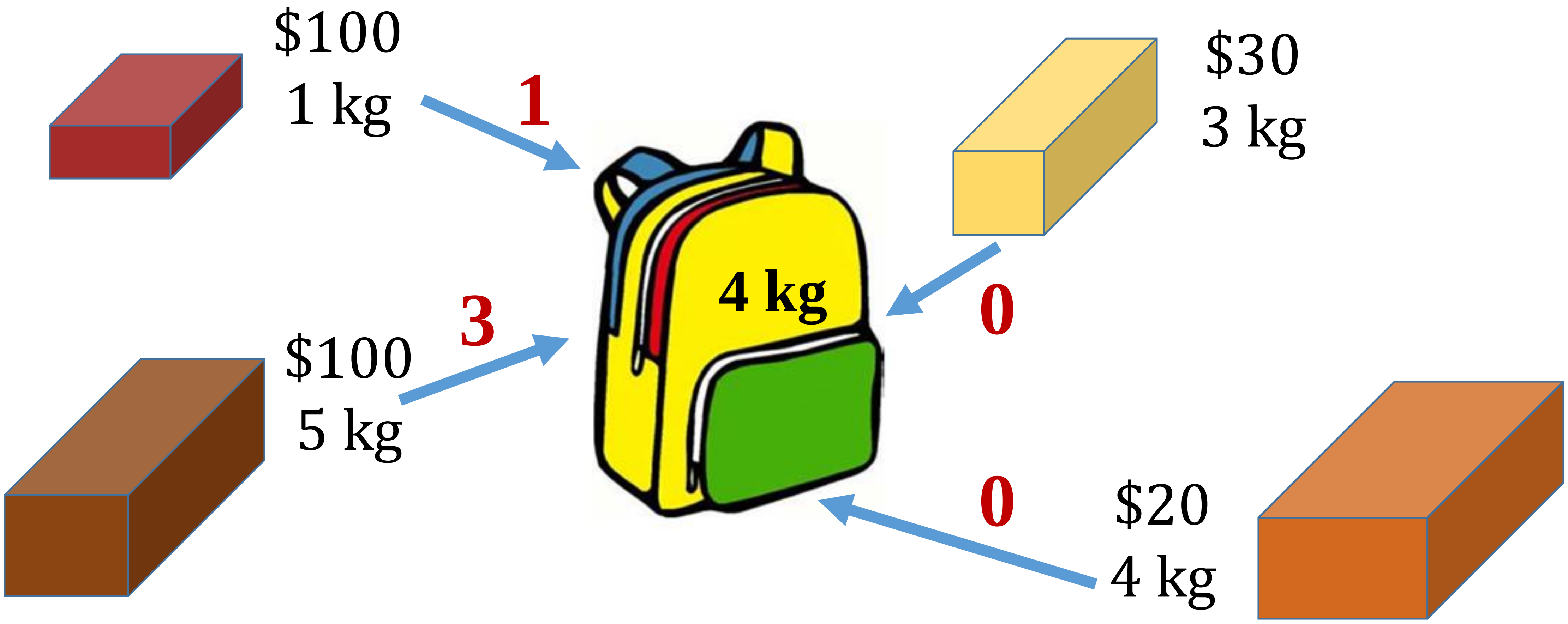
Fractional Knapsack



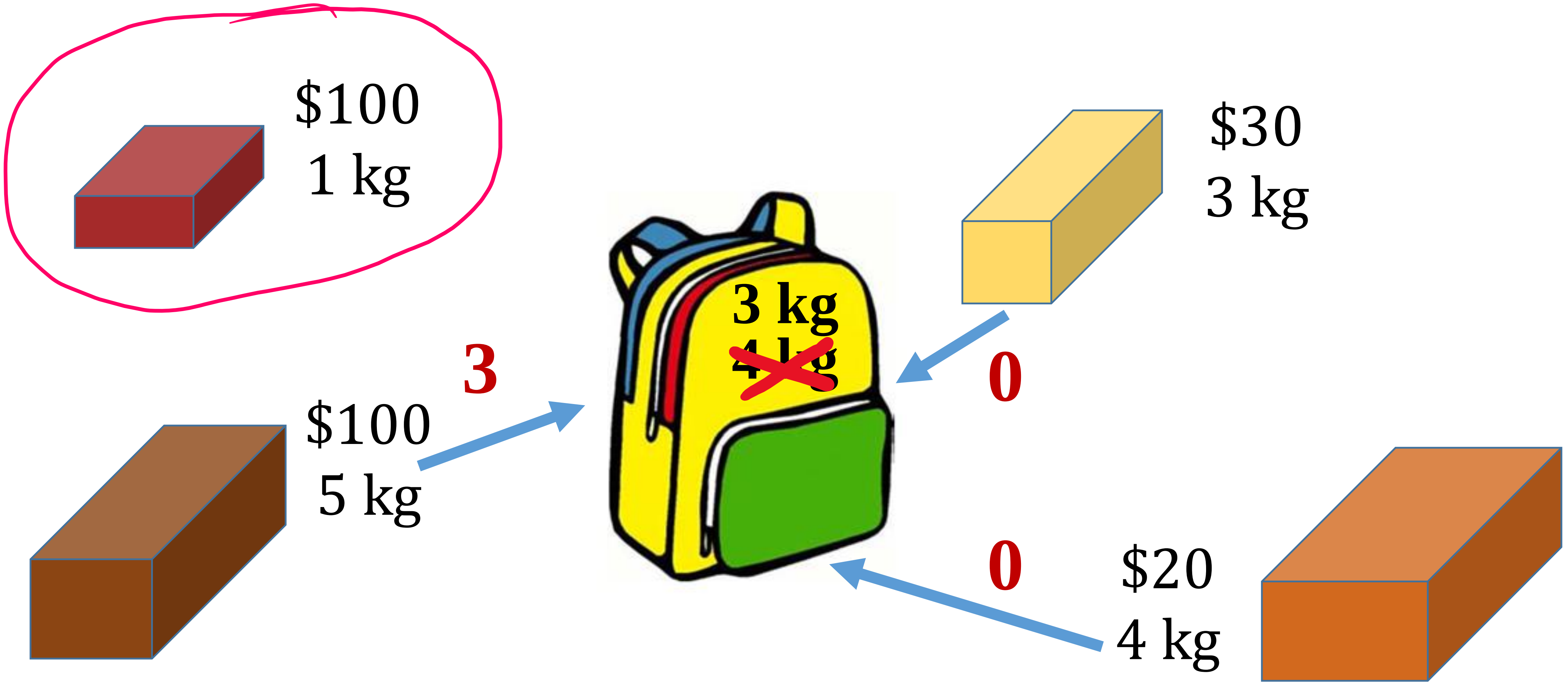
Optimal Substructure

If we remove w kgs of one item j from the optimal knapsack, then the remaining load must be the optimal knapsack weighing at most $T - w$ kgs that one can take from the $n - 1$ original items and $w_j - w$ kgs of item j.

Fractional Knapsack



Fractional Knapsack



Optimal Substructure: Why?

Optimal Substructure: Why?

- Let X be the value of the optimal knapsack. Suppose that the remaining load after removing w kgs of item j was not the optimal knapsack weighing at most $T - w$ kgs that one can take from the $n - 1$ original items and $w_j - w$ kgs of item j.

Optimal Substructure: Why?

- Let X be the value of the optimal knapsack. Suppose that the remaining load after removing w kgs of item j was not the optimal knapsack weighing at most $T - w$ kgs that one can take from the $n - 1$ original items and $w_j - w$ kgs of item j.

- This means that there is a knapsack of value $> X - v_j \cdot \frac{w}{w_j}$ with weight $\leq T - w$ kgs among the $n - 1$ other items and $w_j - w$ kgs of item j.

Optimal Substructure: Why?

- Let X be the value of the optimal knapsack. Suppose that the remaining load after removing w kgs of item j was not the optimal knapsack weighing at most $T - w$ kgs that one can take from the $n - 1$ original items and $w_j - w$ kgs of item j.

- This means that there is a knapsack of value $> X - v_j \cdot \frac{w}{w_j}$ with weight $\leq T - w$ kgs among the $n - 1$ other items and $w_j - w$ kgs of item j.

- Combining with w kgs of item j gives knapsack of value $> X$ and weight at most T for original input. Contradiction!

Optimal Substructure: Why?

- Let X be the value of the optimal knapsack. Suppose that the remaining load after removing w kgs of item j was not the optimal knapsack weighing at most $T - w$ kgs that one can take from the $n - 1$ original items and $w_j - w$ kgs of item j.

- This means that there is a knapsack of value $> X - v_j \cdot \frac{w}{w_j}$ with weight $\leq T - w$ kgs among the $n - 1$ other items and $w_j - w$ kgs of item j

- Combining with w kgs of item j gives knapsack of value $>$ weight at most T for original input. Contradiction!

Cut-and-paste argument

Dynamic Programming?

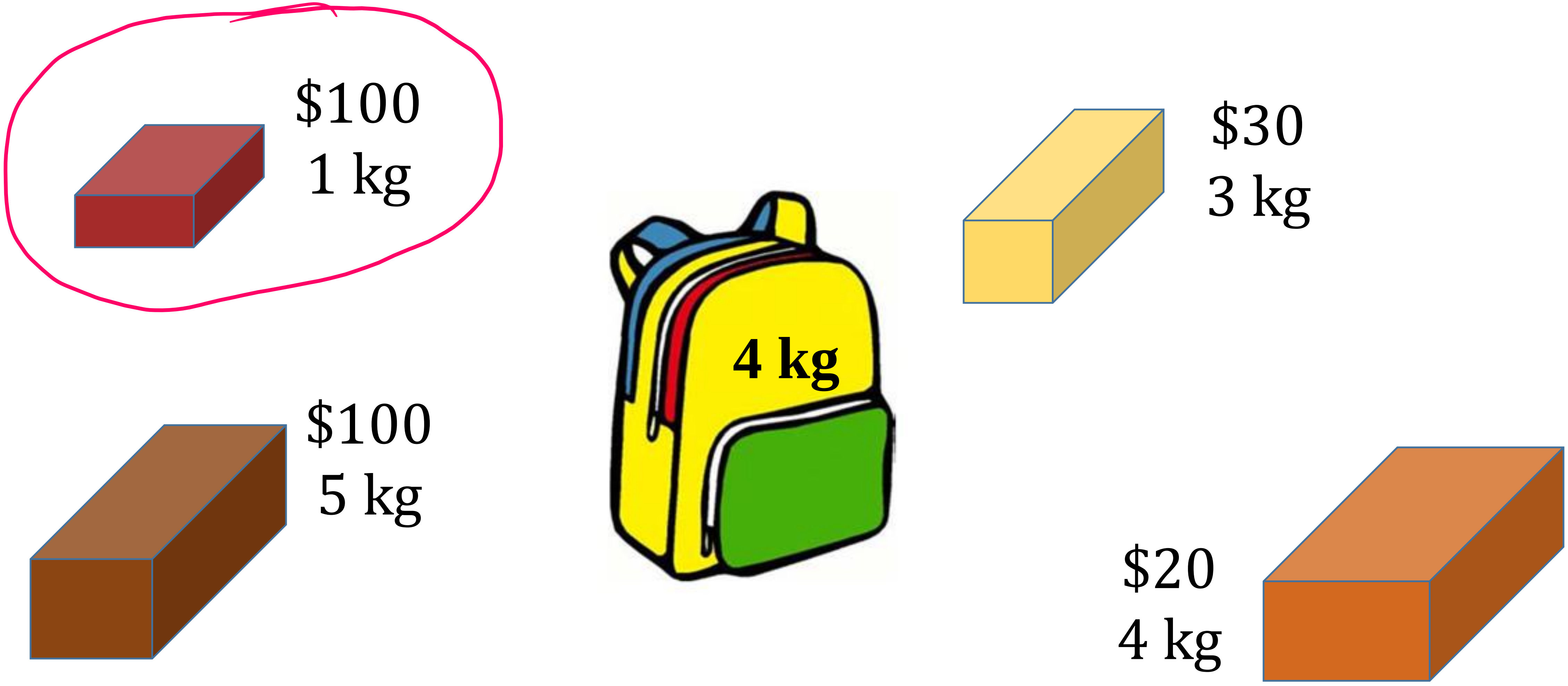
In integral knapsack problem, we used the optimal substructure to formulate DP for deciding whether to add item j.

But in this case, we can do better….

Greedy-choice Property

Claim: Let j^* be the item with the maximum value/kg, v_j/w_j. Then, there exists an optimal knapsack containing $\min\left(w_{j^*}, T\right)$ kgs of item j^*.

Max value/kg item



Greedy-choice Property: Why?

Claim: Let j^* be the item with the maximum value/kg, v_j/w_j. Then, there exists an optimal knapsack containing $\min(w_{j^*}, T)$ kgs of item j^*.

Greedy-choice Property: Why?

Claim: Let j^* be the item with the maximum value/kg, v_j/w_j. Then, there exists an optimal knapsack containing $\min(w_{j^*}, T)$ kgs of item j^*.

- Suppose an optimal knapsack contains x_1 kgs of item 1, x_2 kgs of item 2, ..., x_n kgs of item n such that:

$$x_1 + x_2 + \cdots + x_n = \min(w_{j^*}, T)$$

Greedy-choice Property: Why?

Claim: Let j^* be the item with the maximum value/kg, v_j/w_j. Then, there exists an optimal knapsack containing $\min(w_{j^*}, T)$ kgs of item j^*.

- Suppose an optimal knapsack contains x_1 kgs of item 1, x_2 kgs of item 2, ..., x_n kgs of item n such that:
$$x_1 + x_2 + \cdots + x_n = \min(w_{j^*}, T)$$
- Replace this weight by $\min(w_{j^*}, T)$ kgs of item j^*.

Greedy-choice Property: Why?

Claim: Let j^* be the item with the maximum value/kg, v_j/w_j. Then, there exists an optimal knapsack containing $\min(w_{j^*}, T)$ kgs of item j^*.

- Suppose an optimal knapsack contains x_1 kgs of item 1, x_2 kgs of item 2, ..., x_n kgs of item n such that:

$$x_1 + x_2 + \cdots + x_n = \min(w_{j^*}, T)$$

- Replace this weight by $\min(w_{j^*}, T)$ kgs of item j^*.
- Total weight does not change. Total value does not decrease because value/kg of j^* is maximum. So, knapsack stays optimal.

Strategy for Greedy Algorithm

- Use greedy-choice property to put $\min(w_{j^*}, T)$ kgs of item j^* in knapsack.

- If knapsack weighs T kgs, we are done.

- Otherwise, use optimal substructure to solve subproblem where all of item j^* is removed and knapsack weight limit is $T - w_{j^*}$.

Recursive greedy algorithm

```
REC-FRAC-KNAPSACK(v, w, T):
        if T == 0:
                return
        n ← v.length()
        jmax ← 1
        for i = 2 to n:
                if v[i]/w[i] > v[jmax]/w[jmax]:
                        jmax ← i
        if w_jmax ≥ T:
                print "T kgs of jmax"
        else:
                print "w_jmax kgs of jmax"
                REC-FRAC-KNAPSACK(v.remove(jmax), w.remove(jmax), T − w_jmax)
        return
```

Recursive greedy algorithm

REC-FRAC-KNAPSACK(v, w, T):

  if $T == 0$:

    return

  $n \leftarrow v$.length()

  $jmax \leftarrow 1$

  for $i = 2$ to n:

    if $v[i]/w[i] > v[jmax]/w[jmax]$:

      $jmax \leftarrow i$

  if $w_{jmax} \geq T$:

    print "T kgs of $jmax$"

  else:

    print "w_{jmax} kgs of $jmax$"

    REC-FRAC-KNAPSACK(v.remove($jmax$), w.remove($jmax$), $T - w_{jmax}$)

  return

$O(n^2)$

Iterative greedy algorithm

ITER-FRAC-KNAPSACK(v, w, T):

- $valperkg \leftarrow [1,2,\ldots,n]$
- Sort $valperkg$ using comparison operator $\preccurlyeq$ where $i \preccurlyeq j$ if $\frac{v[i]}{w[i]} \leq \frac{v[j]}{w[j]}$
- **for** $i = 1$ to n:
  - **if** $T == 0$: **break**
  - $j \leftarrow valperkg[i]$
  - $w \leftarrow \min(w[j], T)$
  - **print** "w kgs of item j"
  - $T \leftarrow T - w$
- **return**

Iterative greedy algorithm

ITER-FRAC-KNAPSACK(v, w, T):

$valperkg \leftarrow [1,2,\ldots,n]$

Sort $valperkg$ using comparison operator $\preccurlyeq$ where $i \preccurlyeq j$ if $\frac{v[i]}{w[i]} \leq \frac{v[j]}{w[j]}$

for $i = 1$ to n:

if $T == 0$: **break**

$j \leftarrow valperkg[i]$

$w \leftarrow \min(w[j], T)$

print "w kgs of item j"

$T \leftarrow T - w$

return

$O(n \log n)$

Paradigm for greedy algorithms

1. Cast the problem where we have to make a choice and are left with one subproblem to solve.

2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so the greedy choice is safe.

3. Use optimal substructure to show that we can combine an optimal solution to the subproblem with the greedy choice to get an optimal solution to the original problem.

Today: Greedy Algorithms

- Fractional Knapsack problem
- **Activity Selection problem**
- Prim's algorithm for Minimum Spanning Tree

Activity Selection Problem

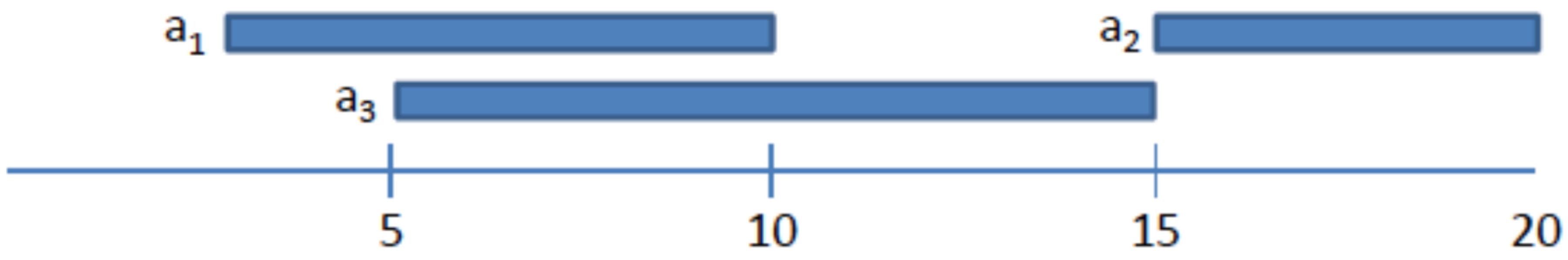
Given a set of activities $S = \{a_1, a_2, \dots, a_n\}$:

- Each activity takes place during $[s_i, f_i)$
- Two activities a_i and a_j are **compatible** if their time intervals don't overlap: $s_i \geq f_j$ or $s_j \geq f_i$.

Problem: Find a largest subset of mutually compatible activities.

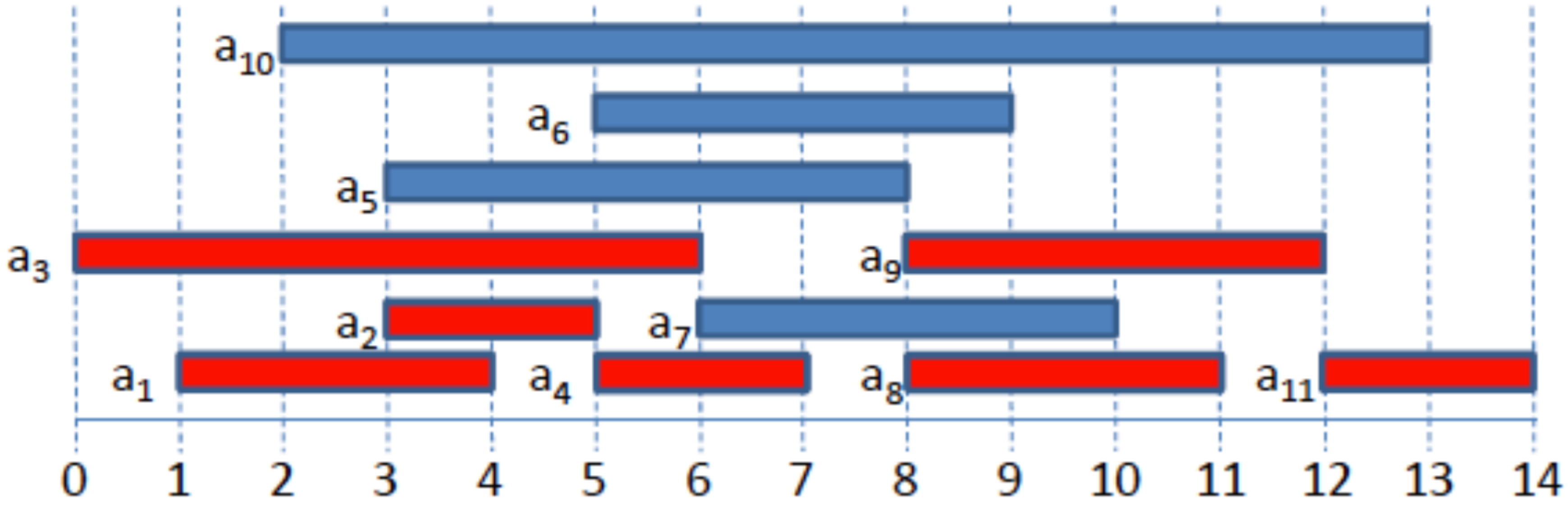
Example: a_1=[3, 10), a_2=[15, 20), a_3=[5, 15)

- {a_1 and a_2} and {a_2 and a_3} are compatible
- {a_1 and a_3} are not compatible



Example: Find a max-size subset of mutually compatible activities

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



$\{a_3, a_9, a_{11}\}$ are mutually compatible

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

Another largest set is $\{a_2, a_4, a_9, a_{11}\}$

Optimal Substructure

Suppose an optimal scheduling S contains activity a_j. Let:

$$\text{Before}_j = \{a_i : f_i \leq s_j\}$$
$$\text{After}_j = \{a_i : s_i \geq f_j\}$$

Then, S also contains an optimal scheduling for Before_j and an optimal scheduling for After_j.

Optimal Substructure: Why?

- Suppose S does not include an optimal schedule for Before_j.

- Then, by replacing the set of activities from Before_j in S with the optimal scheduling for Before_j, we can increase the size of S and maintain compatibility. Contradiction!

- Same argument for After_j

Using optimal substructure

- Value of the optimal scheduling is then:

$$OPT(a_1, \ldots, a_n) = \max_{j \in [n]} \left(1 + OPT\left(\text{Before}_j\right) + OPT\left(\text{After}_j\right)\right)$$

where we are searching for a_j that belongs in an optimal solution.

- Can set this up as a DP with running time $O\left(n^3\right)$. Exercise!

- But greedy choice simplifies the search for a_j…

Greedy-choice property

Claim: There exists an optimal scheduling that contains the activity a^* with the earliest finishing time.

Greedy-choice property

Claim: There exists an optimal scheduling that contains the activity a^* with the earliest finishing time.

Note that $\text{Before}_{a^*} = \emptyset$, so:

$$OPT(a_1, \ldots, a_n) = 1 + OPT(\{a_1, \ldots, a_n\} \setminus \{a^*\})$$

Greedy-choice property: Why?

Claim: There exists an optimal scheduling that contains the activity a_m with the earliest finishing time.

Greedy-choice property: Why?

Claim: There exists an optimal scheduling that contains the activity a_m with the earliest finishing time.

- Consider an optimal scheduling S and let a be the activity in S that has the earliest finishing time.

Greedy-choice property: Why?

Claim: There exists an optimal scheduling that contains the activity a_m with the earliest finishing time.

- Consider an optimal scheduling S and let a be the activity in S that has the earliest finishing time.

- Replace a by a^*. The schedule remains compatible and number of activities is the same.

Greedy-choice property: Why?

Claim: There exists an optimal scheduling that contains the activity a_m with the earliest finishing time.

- Consider an optimal scheduling S and let a be the activity in S that has the earliest finishing time.
- Replace a by a^*. The schedule remains compatible and number of activities is the same.
- The new schedule is also optimal.

Greedy algorithm

GREEDY-ACTIVITY-SELECTOR(s,f)

$n \leftarrow$ length[s], $A \leftarrow \{a_1\}$, $i \leftarrow 1$

for $m \leftarrow 2$ **to** n

do if $s_m \geq f_i$

then $A \leftarrow A \cup \{a_m\}$

$i \leftarrow m$

return A

Linear time, assuming activities already ordered in increasing finish time

GREEDY-ACTIVITY-SELECTOR(s,f)

$n \leftarrow \text{length}[s], A \leftarrow \{a_1\}, i \leftarrow 1$

for $m \leftarrow 2$ **to** n

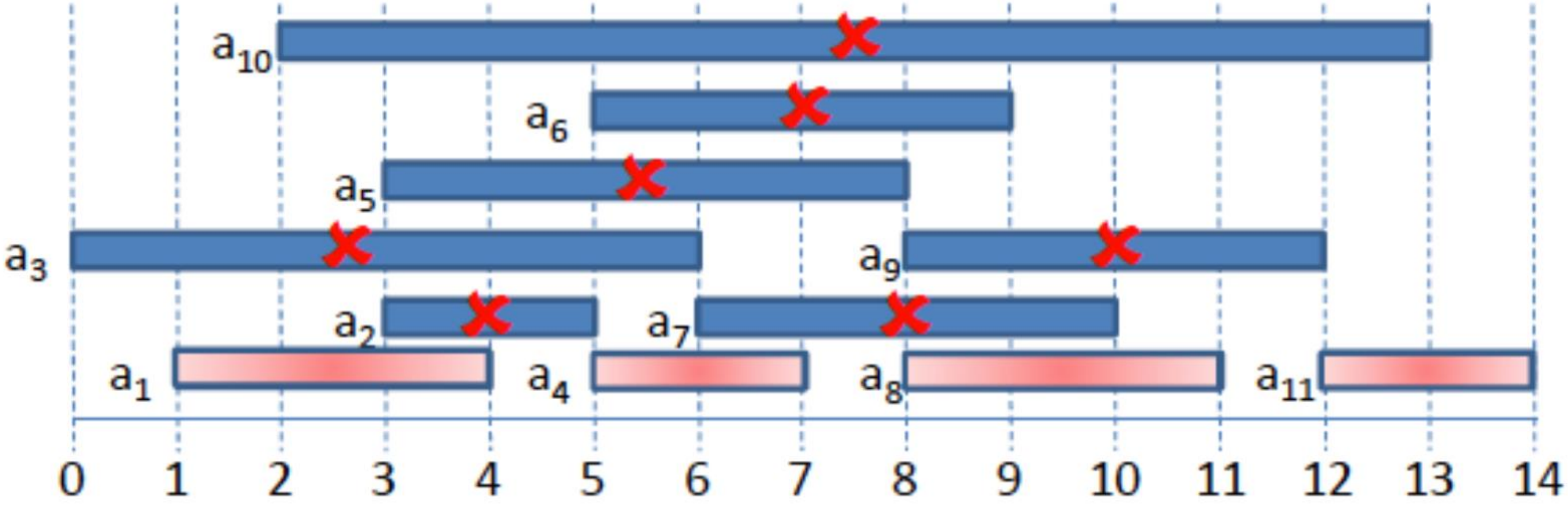
  do if $s_m \geq f_i$

    then $A \leftarrow A \cup \{a_m\}$

      $i \leftarrow m$

return A

Example: Greedy algorithm selects 4 activities: a_1, a_4, a_8, a_{11}



Today: Greedy Algorithms

- Fractional Knapsack problem

- Activity Selection problem

- **Minimum Spanning Tree**

Minimum spanning trees

Input: A connected, undirected graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$.

- For simplicity, assume that all edge weights are distinct. (CLRS covers the general case.)

Minimum spanning trees

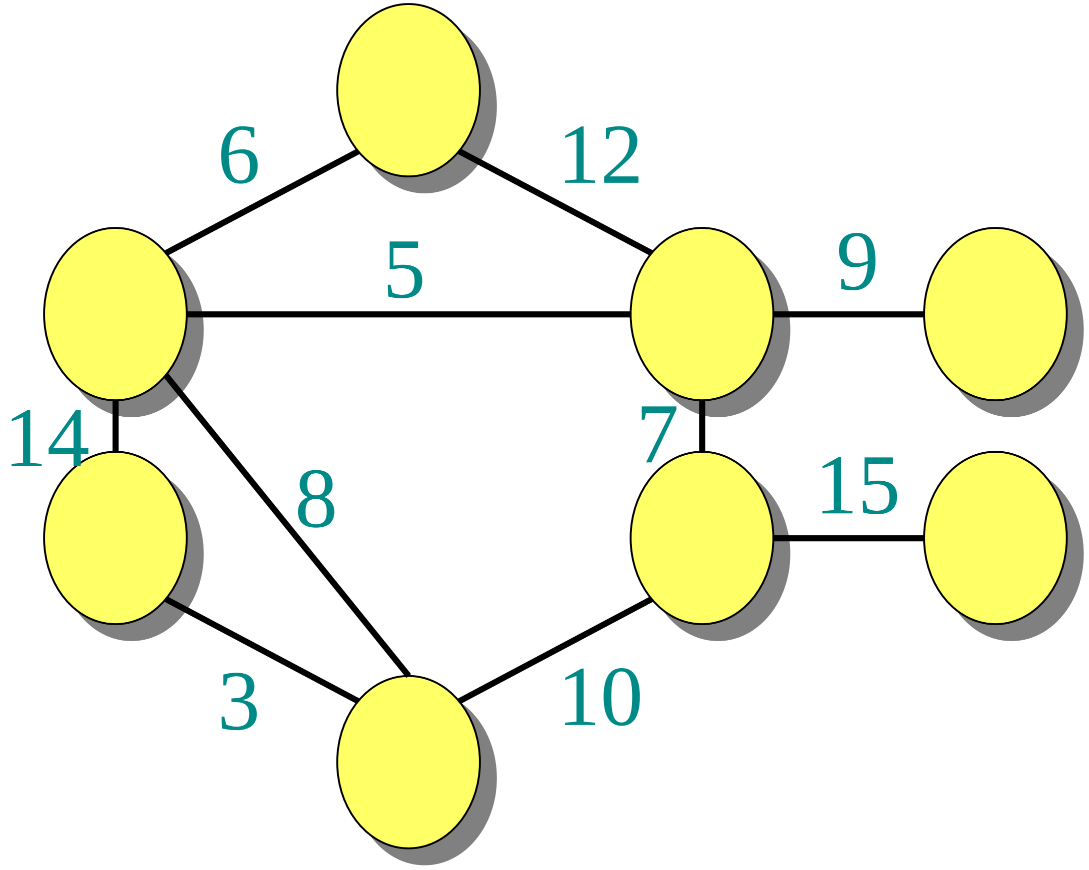
Input: A connected, undirected graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$.

- For simplicity, assume that all edge weights are distinct. (CLRS covers the general case.)

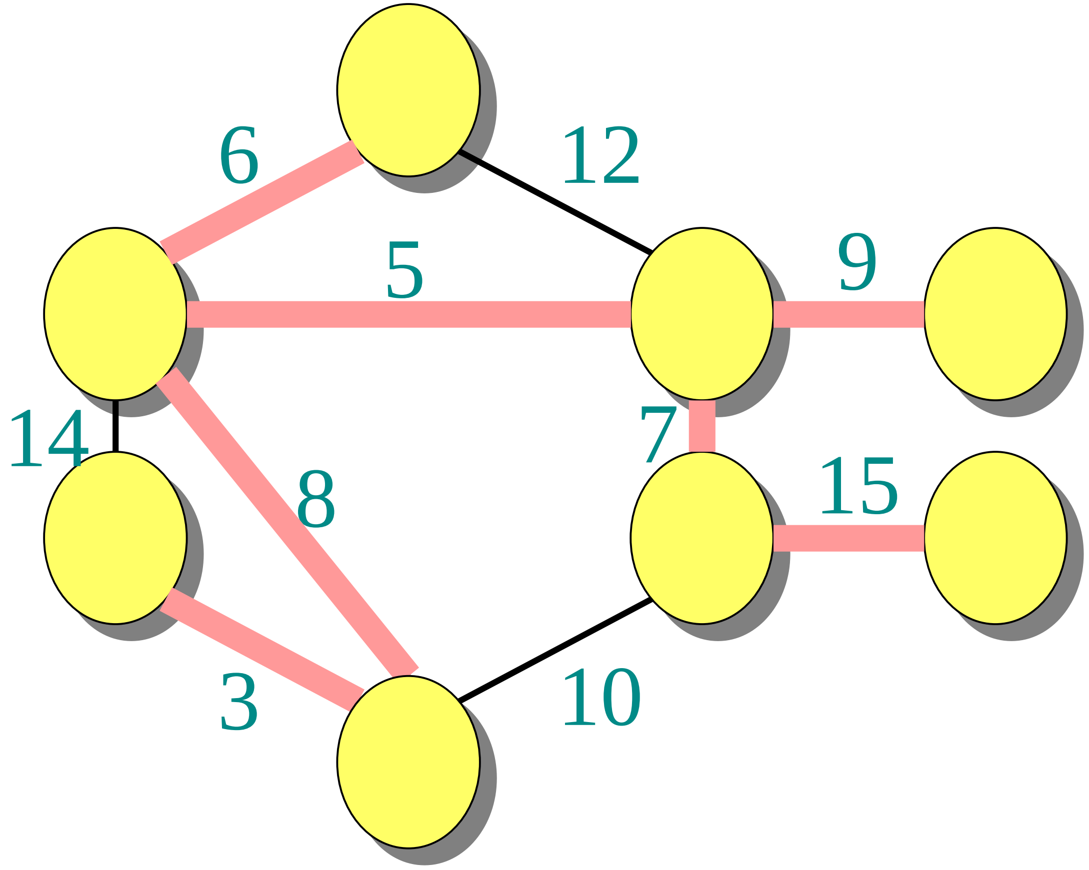
Output: A ***spanning tree*** T — a tree that connects all vertices — of minimum weight:

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

Example of MST



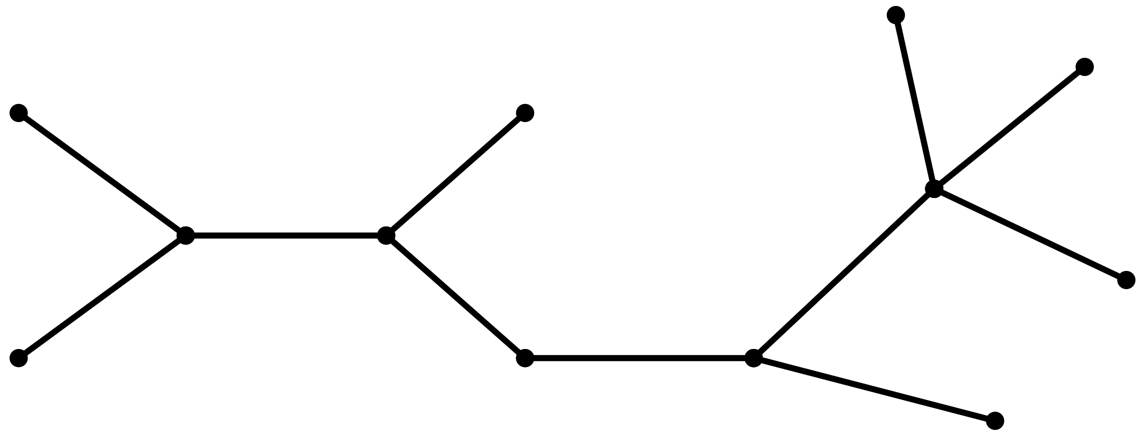
Example of MST



Optimal substructure

MST *T*:

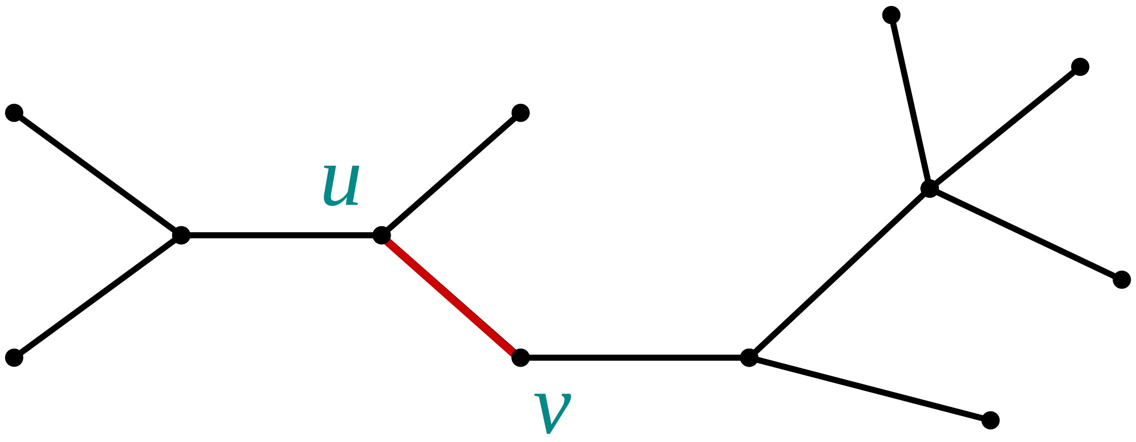
(Other edges of *G* are not shown.)



Optimal substructure

MST T:

(Other edges of G are not shown.)

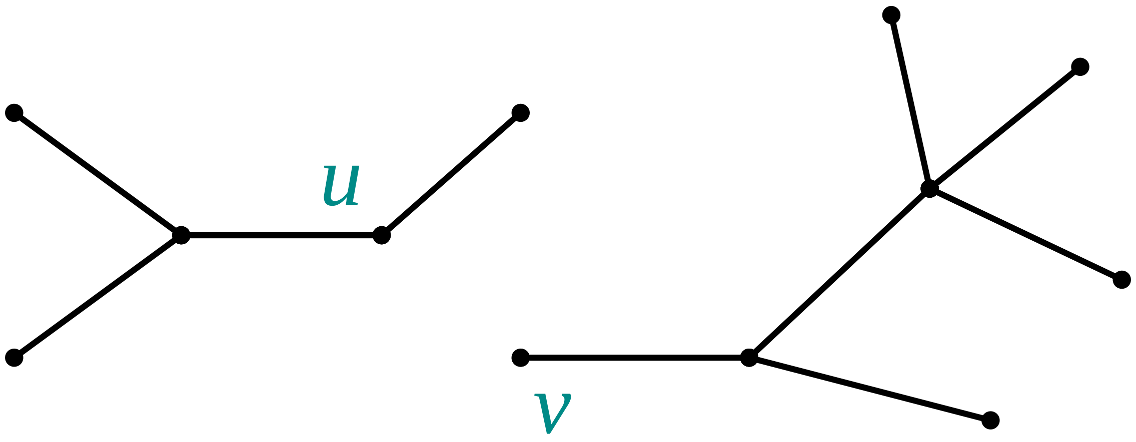


Remove any edge $(u, v) \in T$.

Optimal substructure

MST T:

(Other edges of G are not shown.)

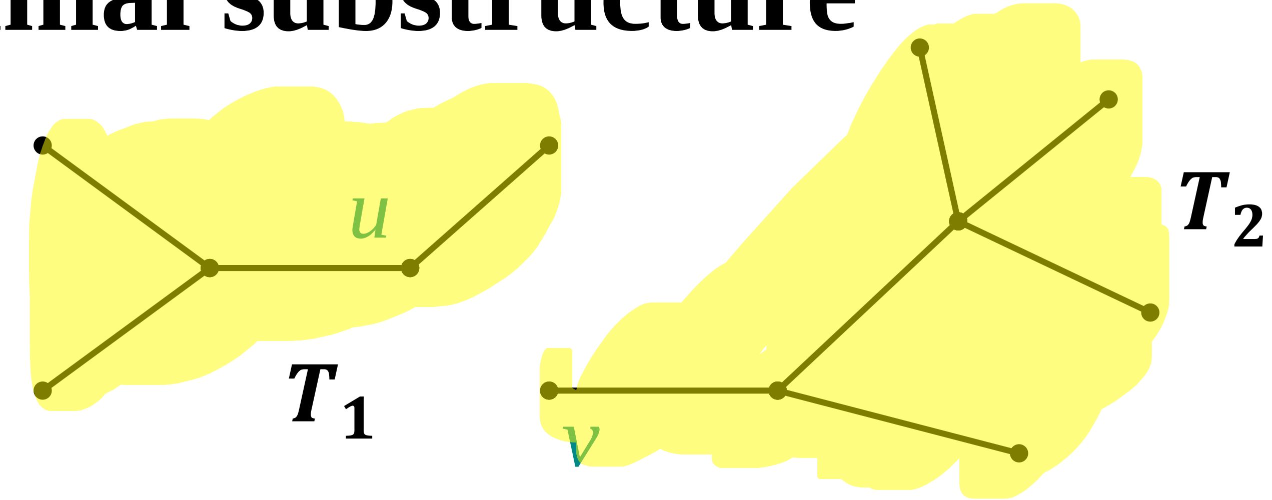


Remove any edge $(u, v) \in T$.

Optimal substructure

MST T:

(Other edges of G are not shown.)

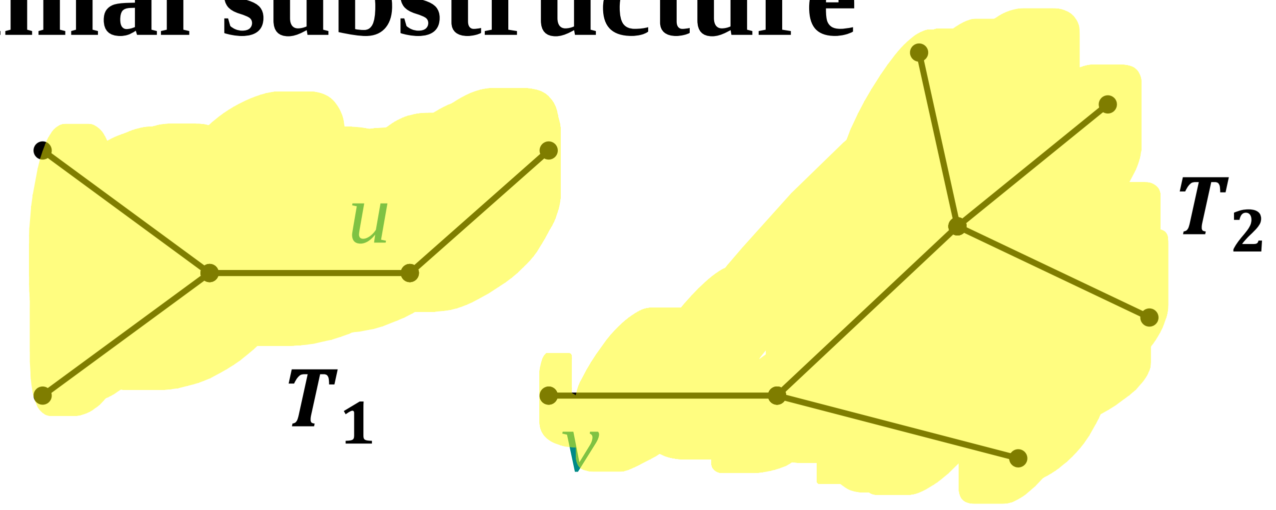


Remove any edge $(u, v) \in T$. Then, T is partitioned into two subtrees T_1 and T_2.

Optimal substructure

MST T:

(Other edges of G are not shown.)



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

$$V_1 = \text{vertices of } T_1,$$

$$E_1 = \{ (x, y) \in E : x, y \in V_1 \}.$$

Similarly for T_2.

Optimal substructure: Why?

Proof. Cut and paste:

$$w(T) = w(u, v) + w(T_1) + w(T_2).$$

If T_1' were a lower-weight spanning tree than T_1 for G_1, then $T' = \{(u, v)\} \cup T_1' \cup T_2$ would be a lower-weight spanning tree than T for G. $\square$

Proof of optimal substructure

Proof. Cut and paste:

$$w(T) = w(u, v) + w(T_1) + w(T_2).$$

If T_1' were a lower-weight spanning tree than T_1 for G_1, then $T' = \{(u, v)\} \cup T_1' \cup T_2$ would be a lower-weight spanning tree than T for G. ☐

Can use DP! The DP algorithm would search for which edge (u, v) to add and then recurse on T_1 and T_2.

Hallmark for “greedy” algorithms

Greedy-choice property

A locally optimal choice is globally optimal.

Hallmark for "greedy" algorithms

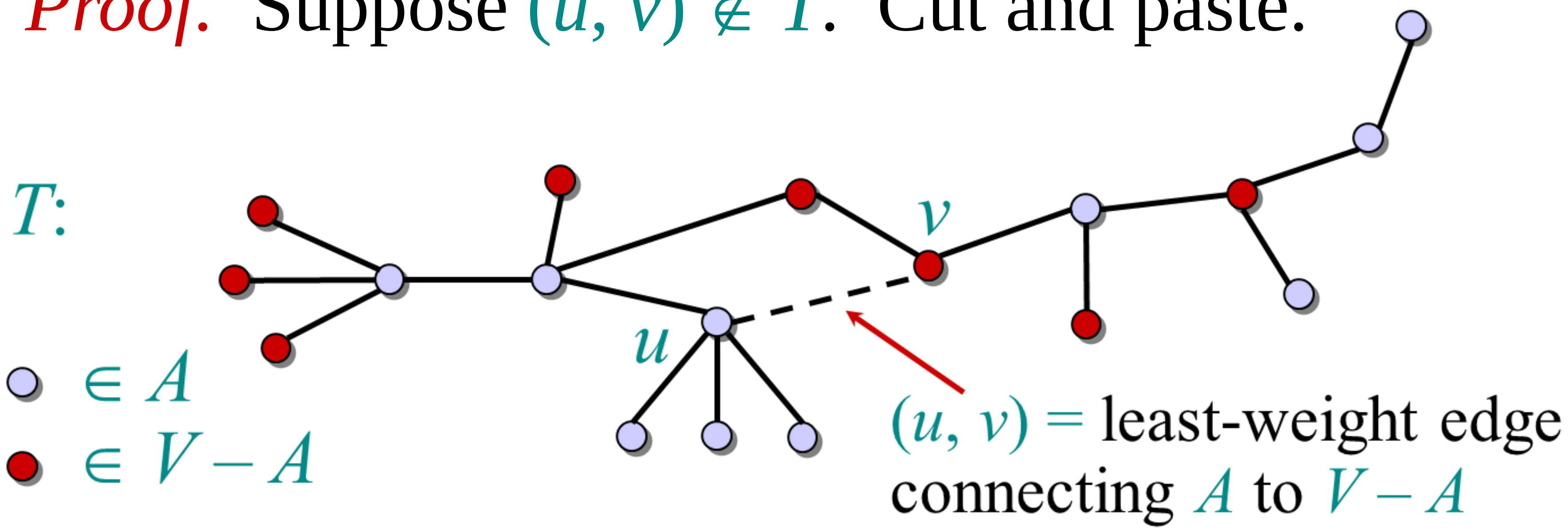
Greedy-choice property

A locally optimal choice is globally optimal.

Theorem. Let T be a MST of $G = (V, E)$, and let A be **any** subset of vertices. Suppose that $(u, v) \in E$ is the least-weight edge connecting A to $V - A$. Then, $(u, v) \in T$.

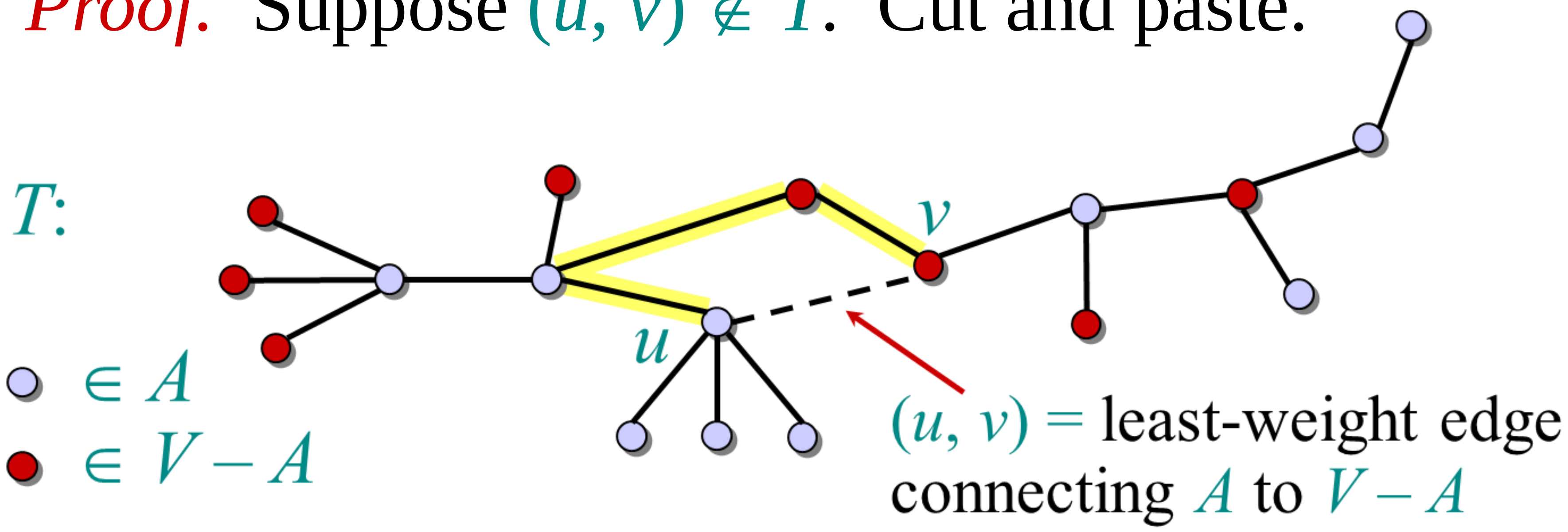
Greedy-choice property: Why?

Proof. Suppose $(u, v) \notin T$. Cut and paste.



Greedy-choice property: Why?

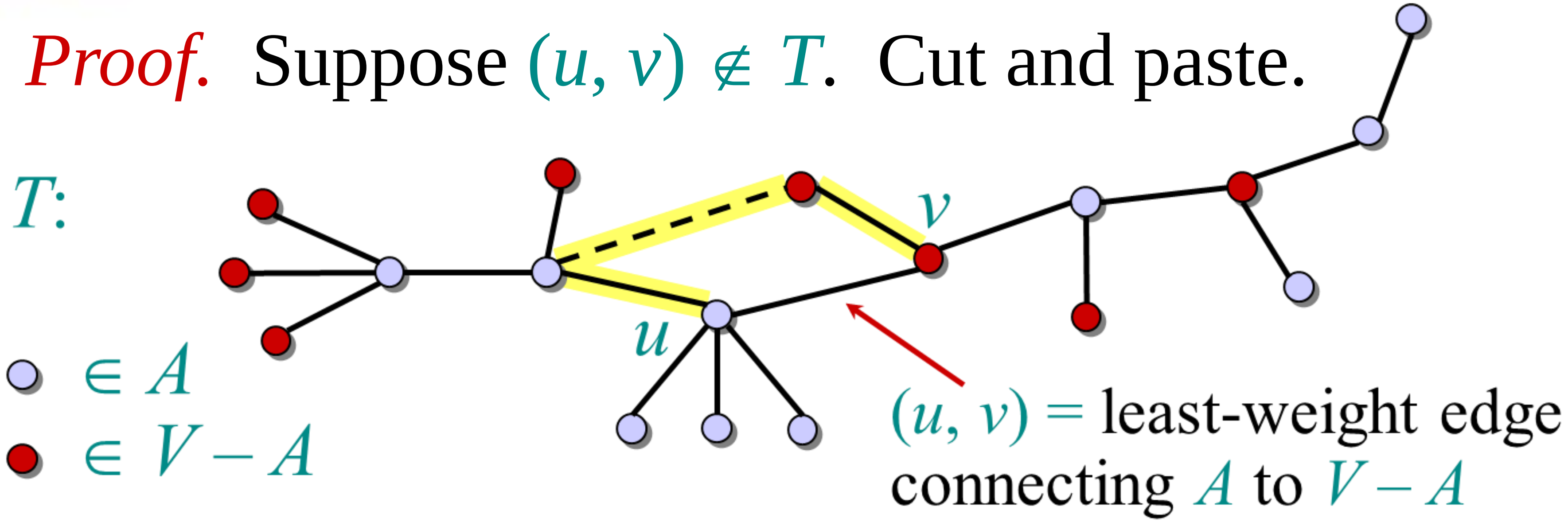
Proof. Suppose $(u, v) \notin T$. Cut and paste.



Consider the unique simple path from u to v in T.

Greedy-choice property: Why?

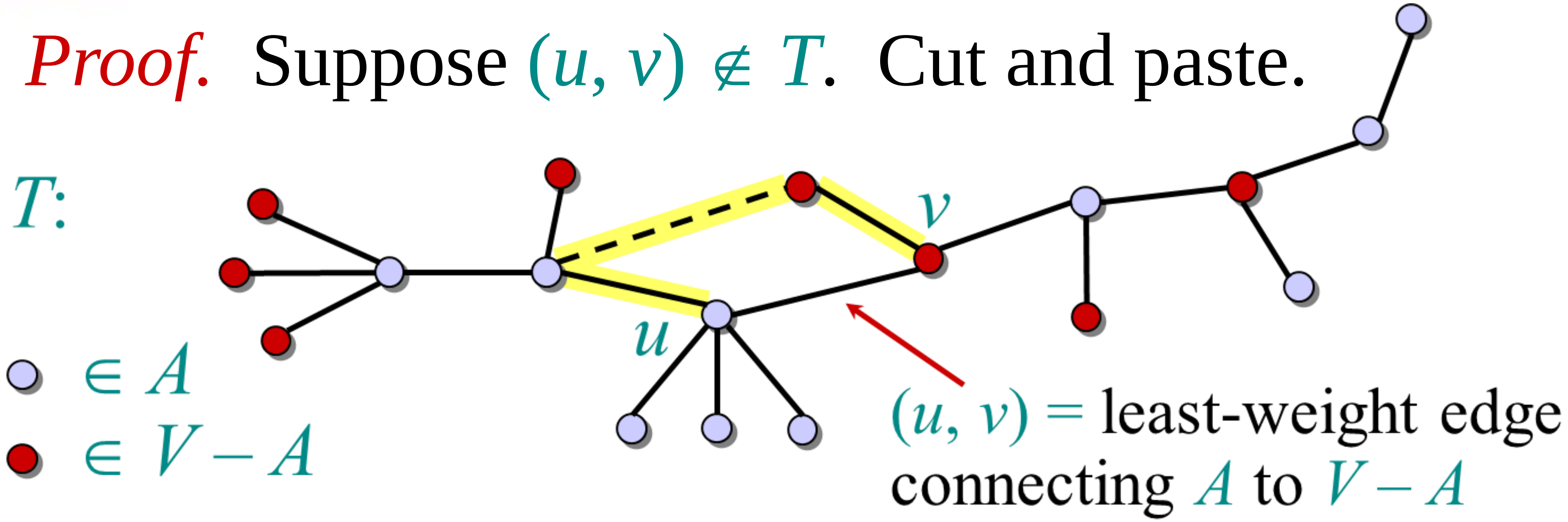
Proof. Suppose $(u, v) \notin T$. Cut and paste.



Swap (u, v) with the first edge on this path that connects a vertex in A to a vertex in $V - A$.

Greedy-choice property: Why?

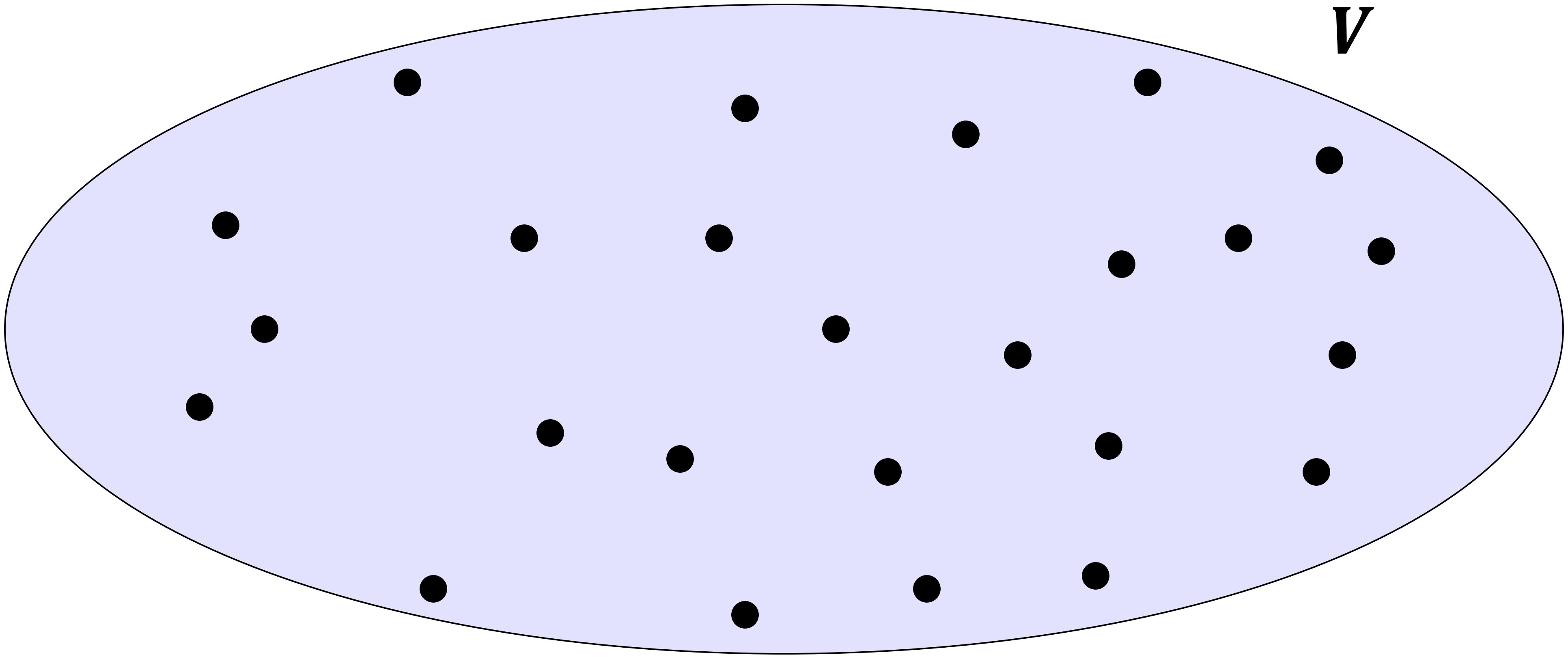
Proof. Suppose $(u, v) \notin T$. Cut and paste.



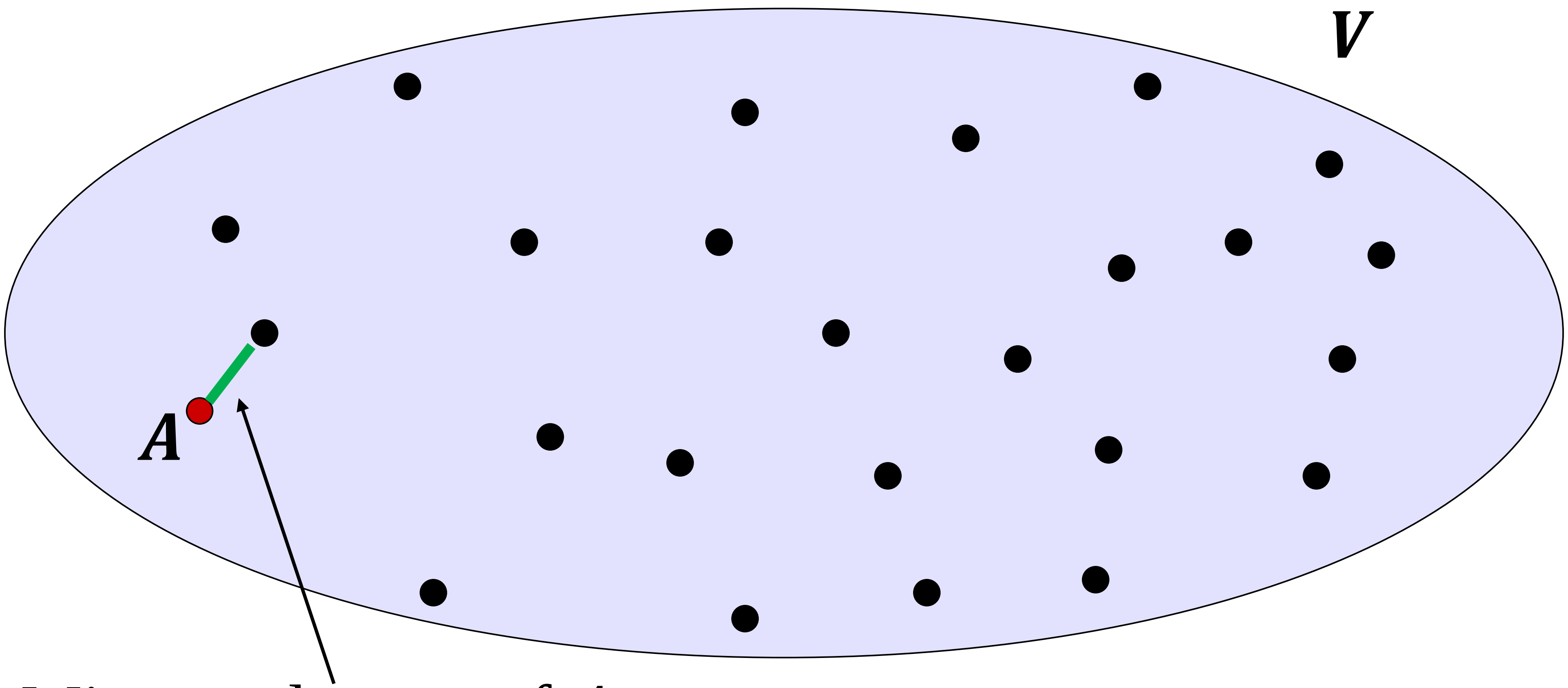
Swap (u, v) with the first edge on this path that connects a vertex in A to a vertex in $V - A$.

A lighter weight spanning tree than T results. ☐

Greedy Algorithm

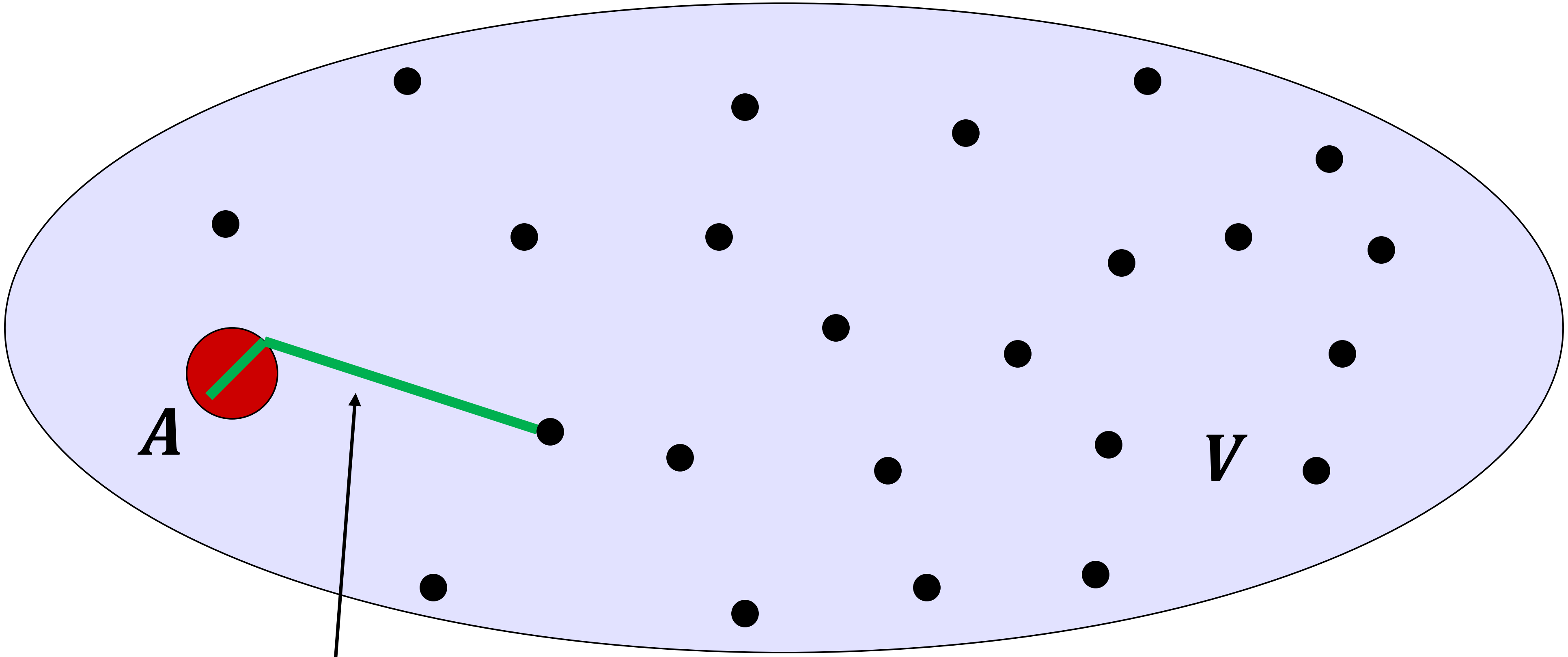


Greedy Algorithm



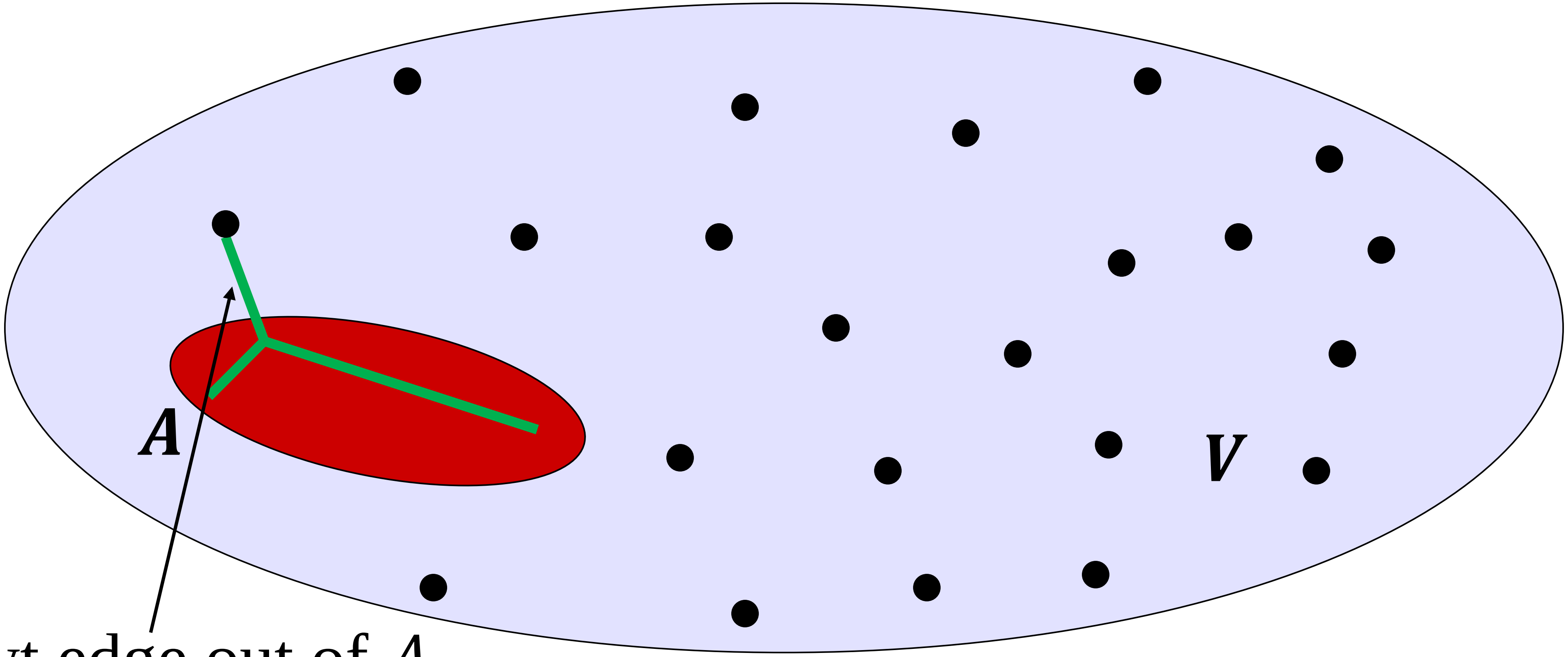
Min wt edge out of A

Greedy Algorithm



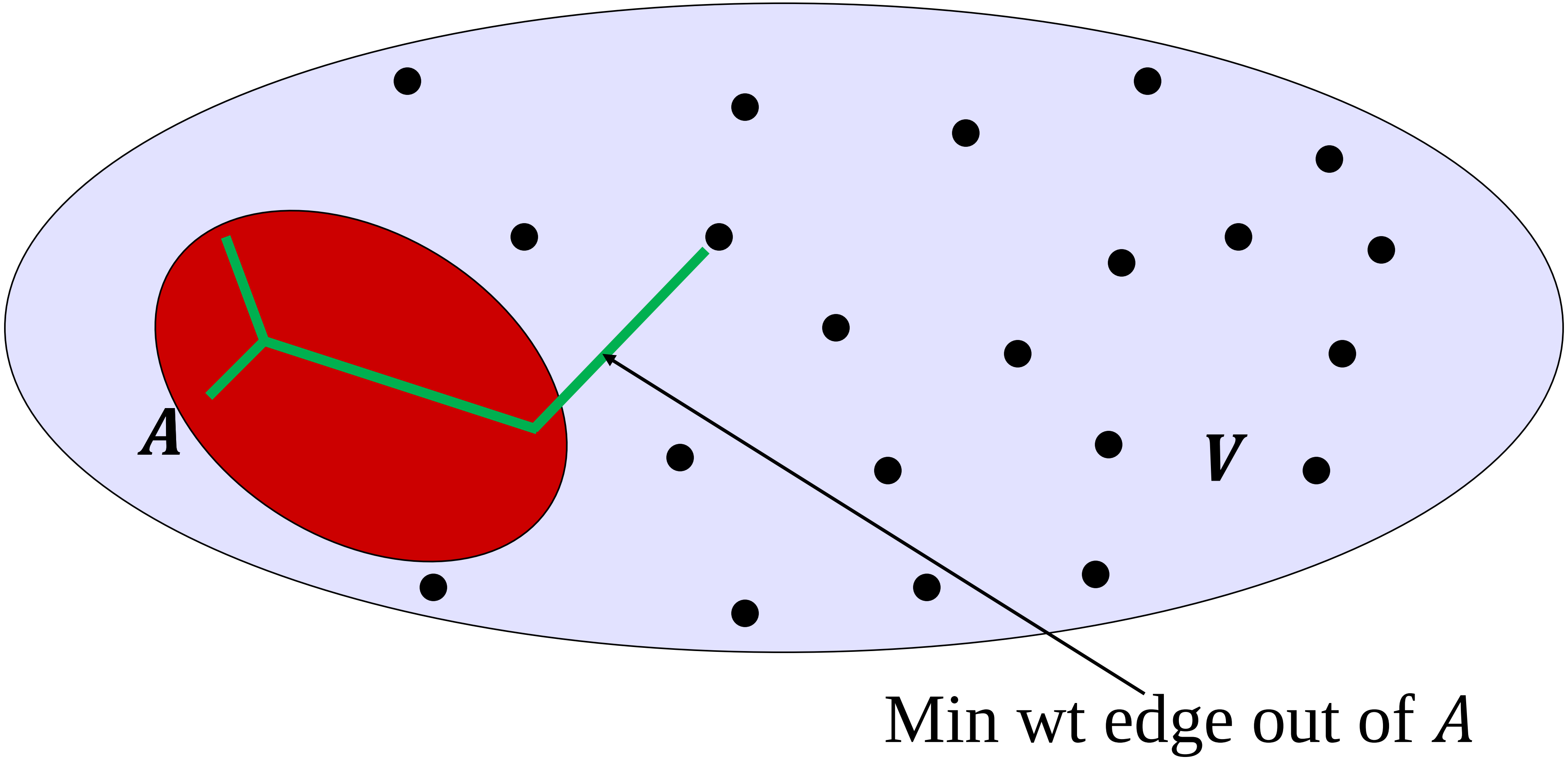
Min wt edge out of A

Greedy Algorithm

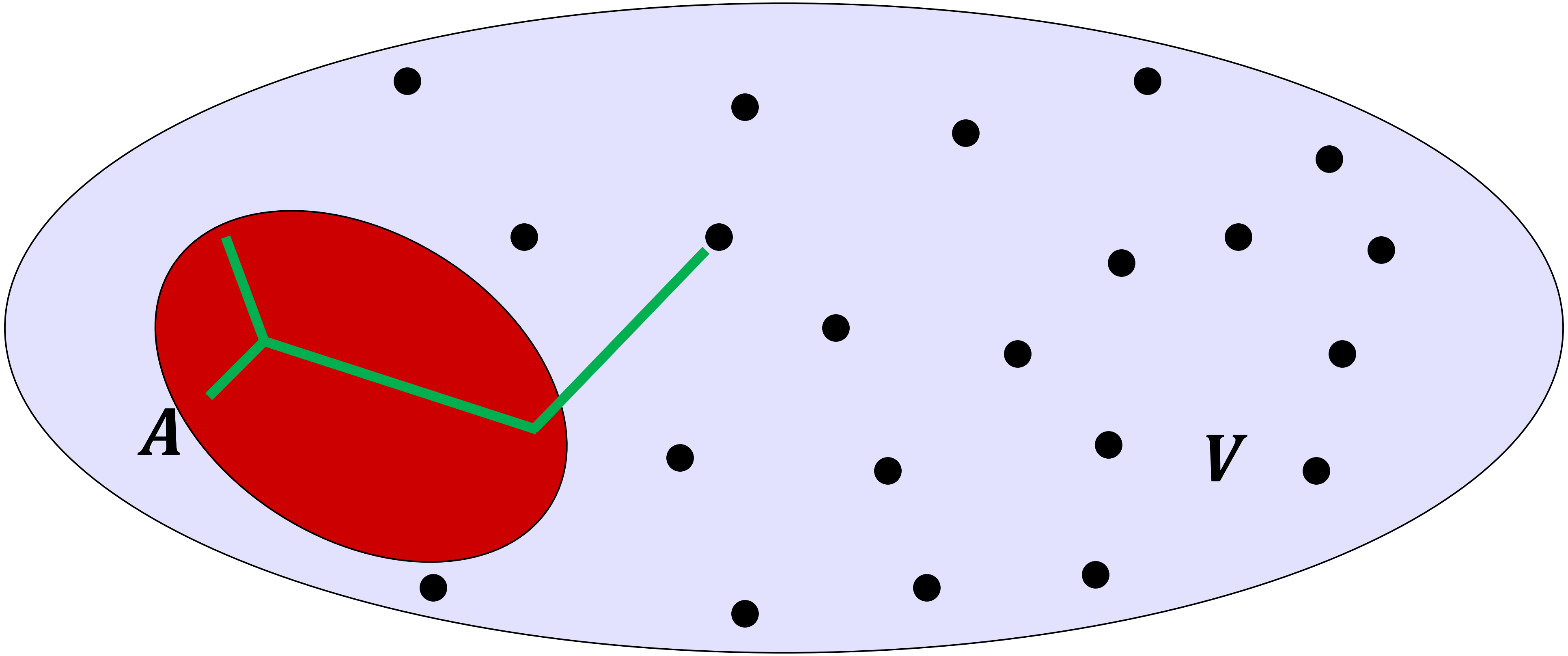


Min wt edge out of A

Greedy Algorithm

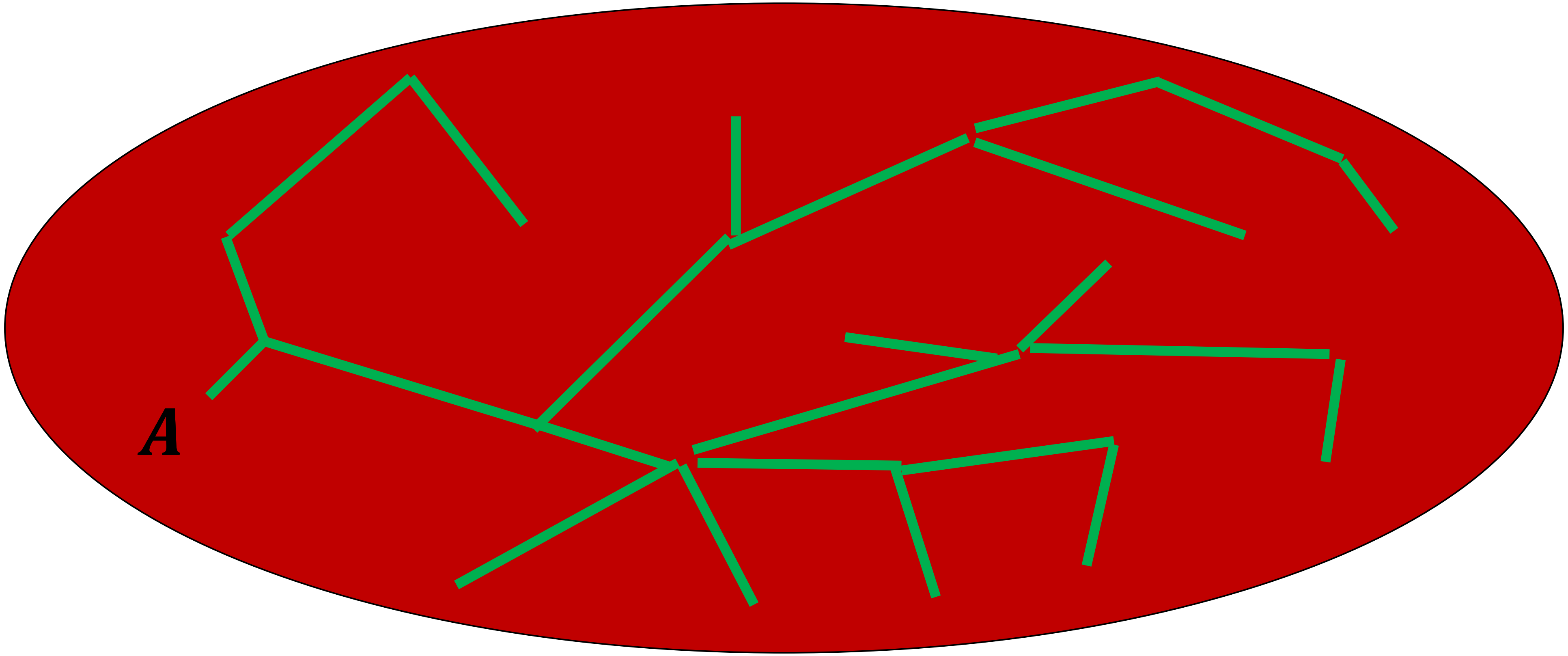


Greedy Algorithm



...

Greedy Algorithm



Prim's algorithm

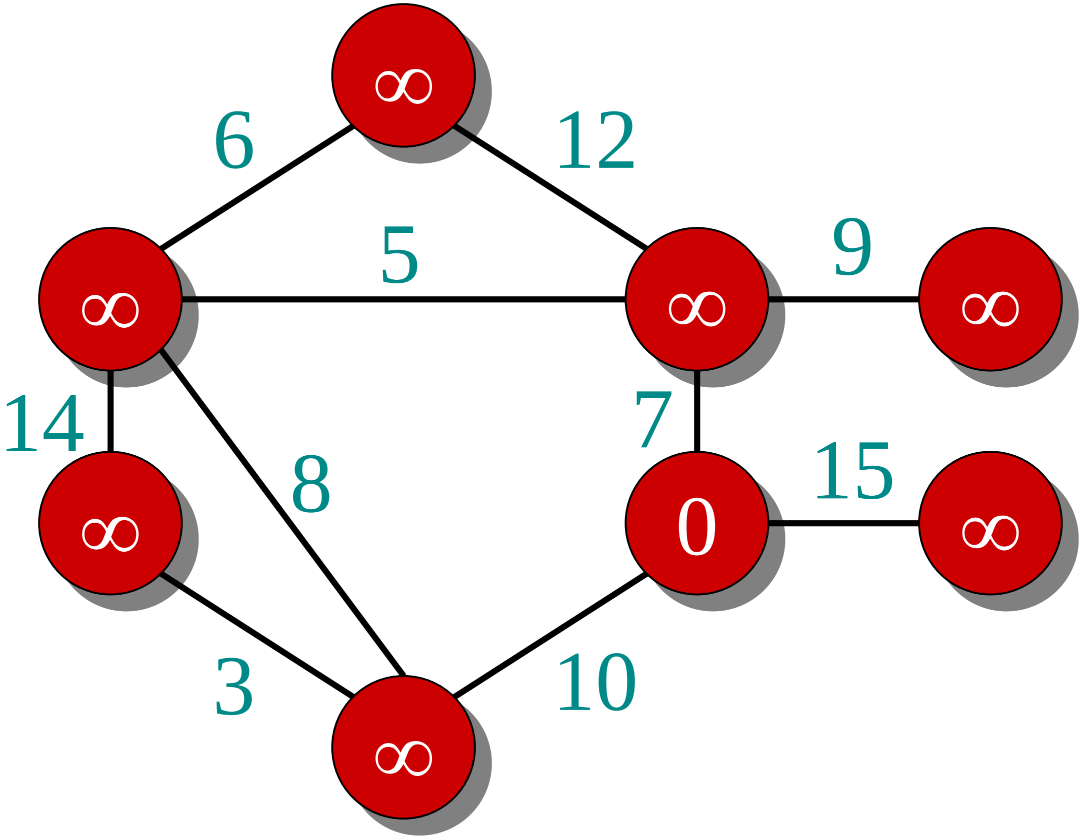
IDEA: Maintain $V - A$ as a priority queue Q. Key each vertex in Q with the weight of the least-weight edge connecting it to a vertex in A.

$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$
while $Q \neq \emptyset$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    for each $v \in Adj[u]$
      do if $v \in Q$ and $w(u, v) < key[v]$
        then $key[v] \leftarrow w(u, v)$ ⊳ DECREASE-KEY
          $\pi[v] \leftarrow u$

At the end, $\{(v, \pi[v])\}$ forms the MST.

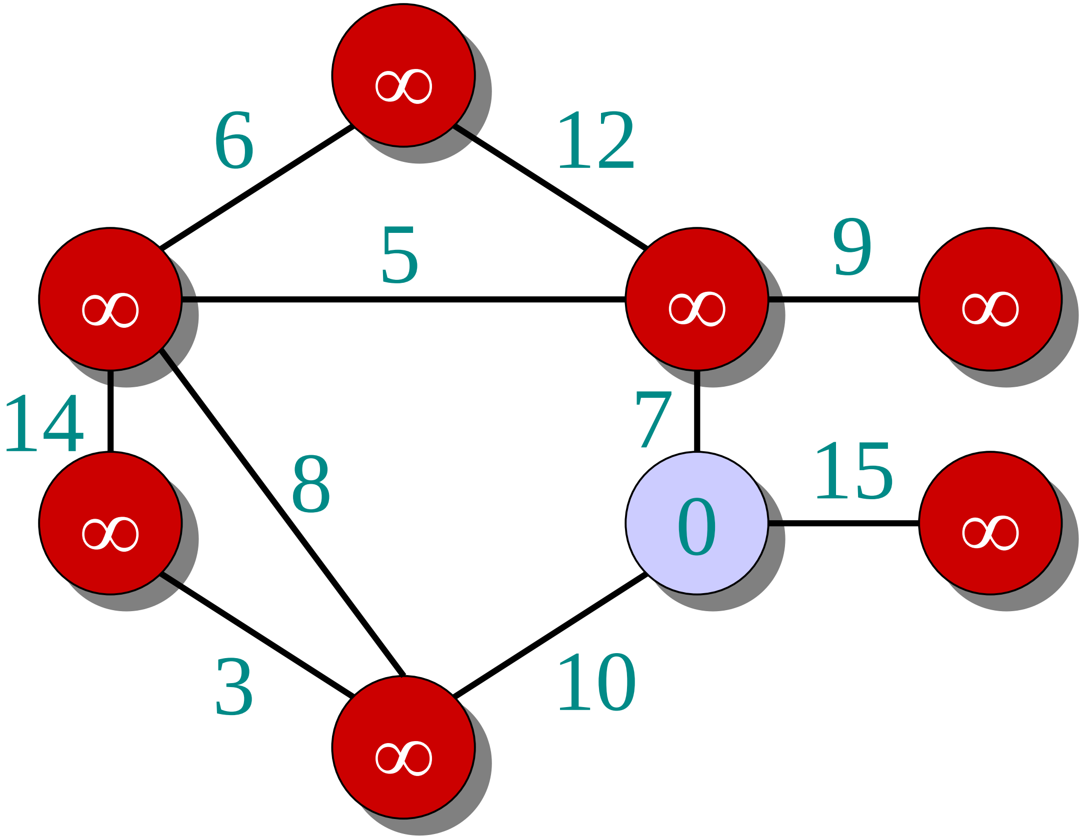
Example of Prim's algorithm

- $\in A$
- $\in V - A$



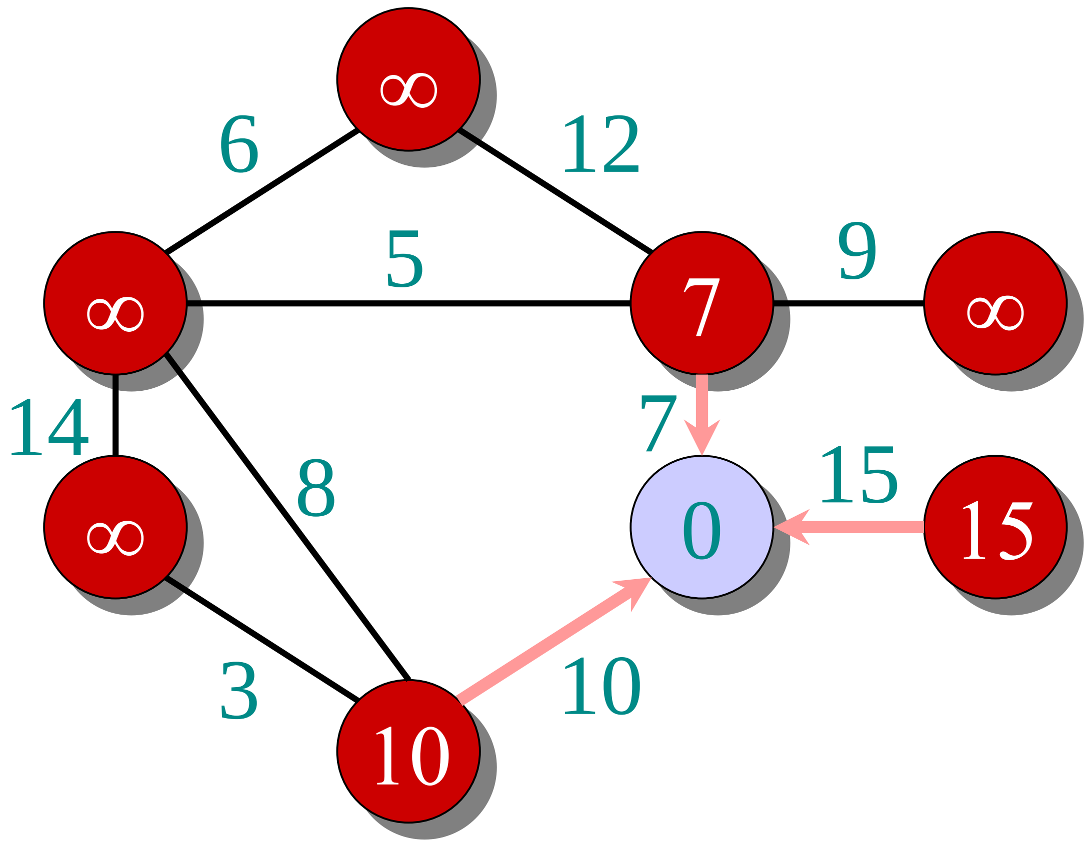
Example of Prim's algorithm

- $\in A$
- $\in V - A$



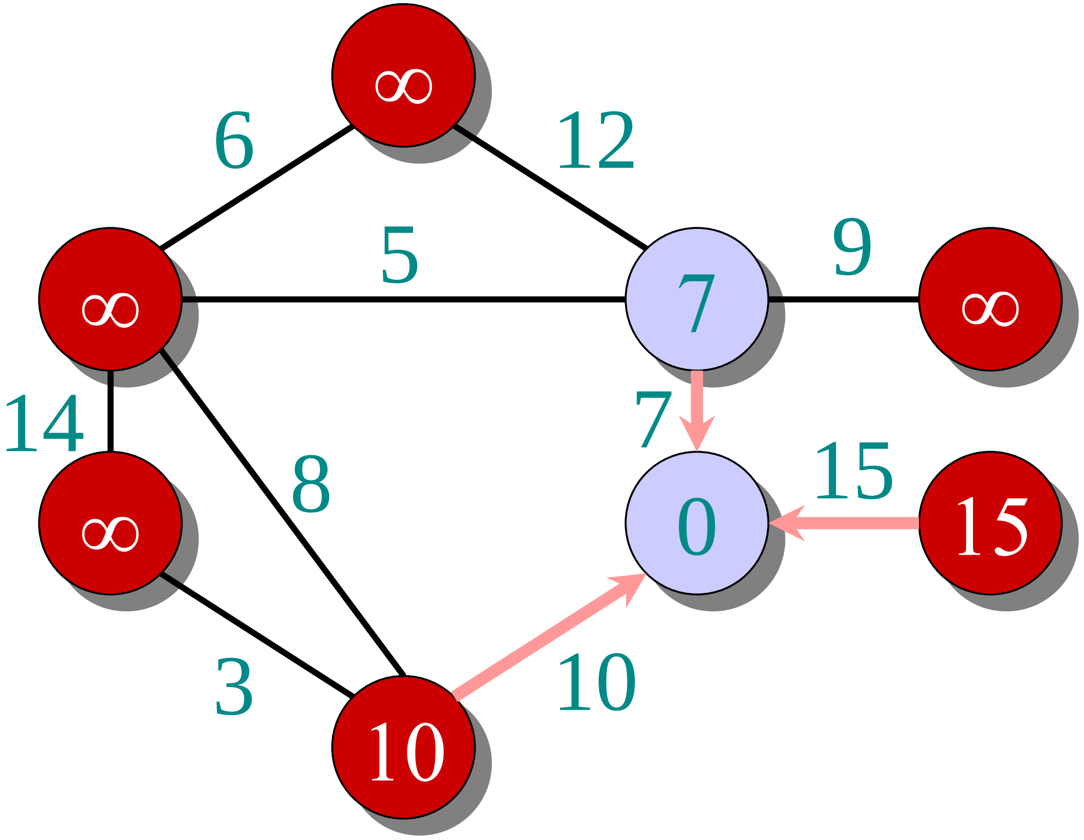
Example of Prim's algorithm

- $\in A$
- $\in V - A$

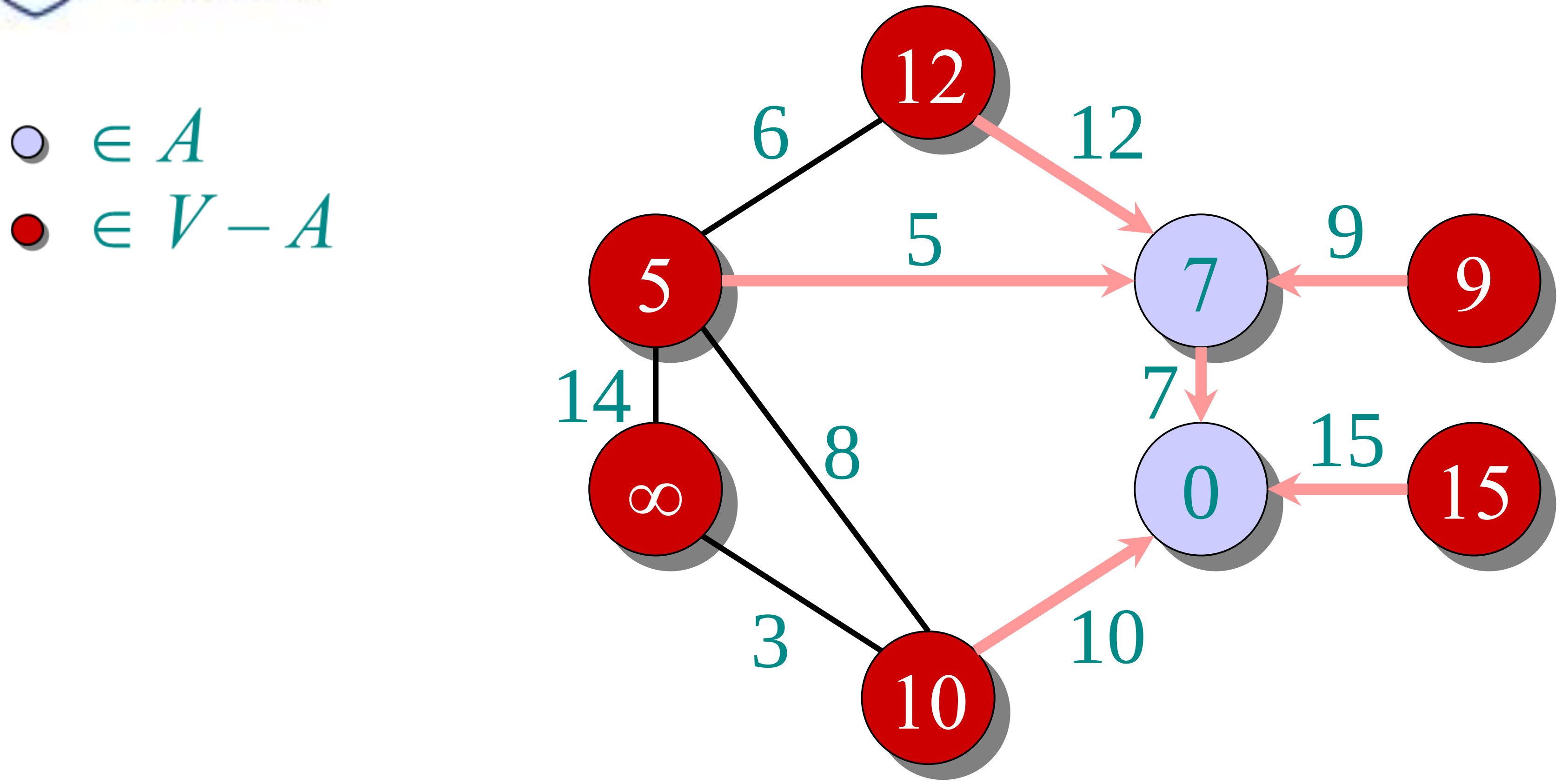


Example of Prim's algorithm

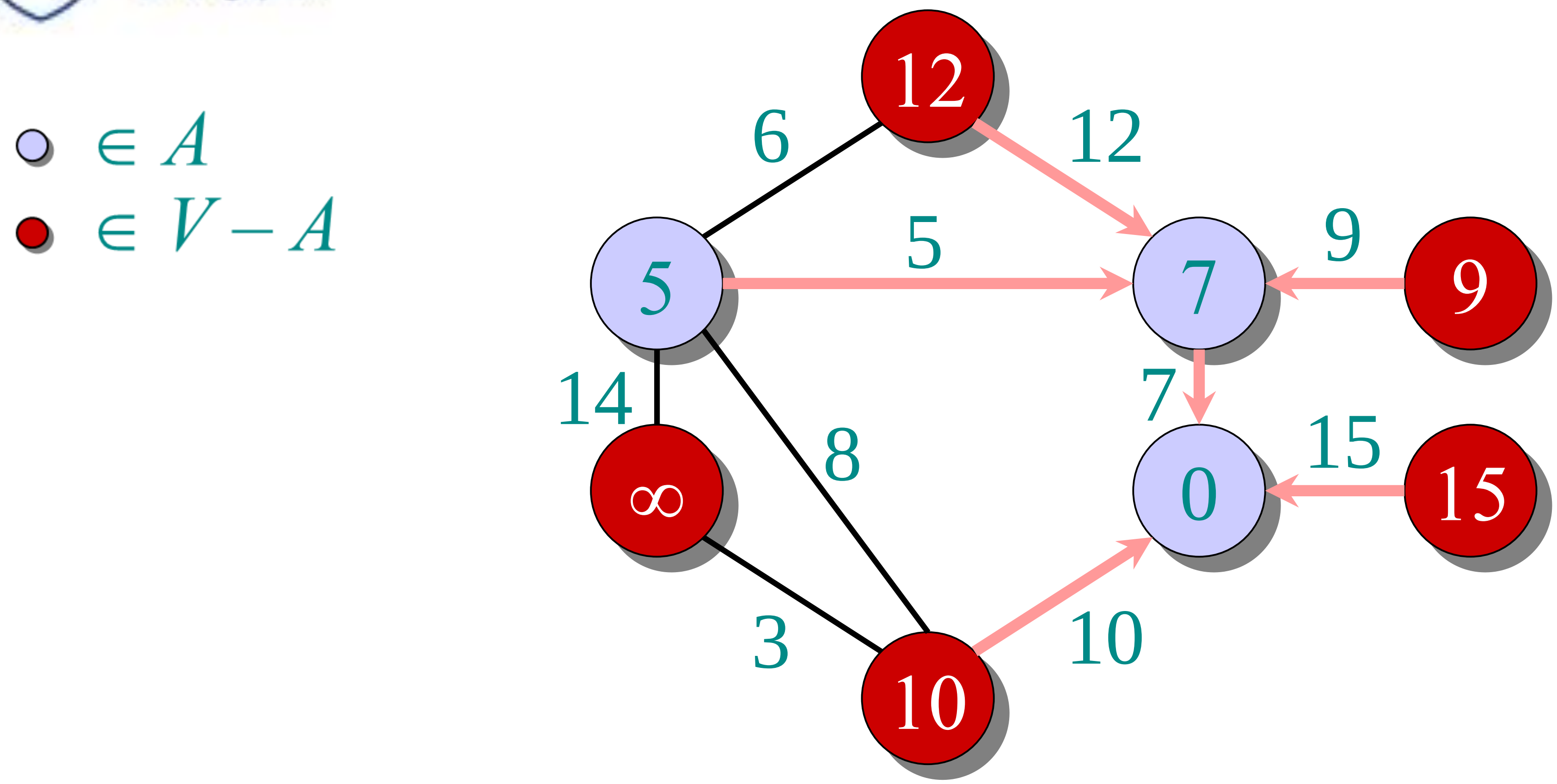
- $\in A$
- $\in V - A$



Example of Prim's algorithm

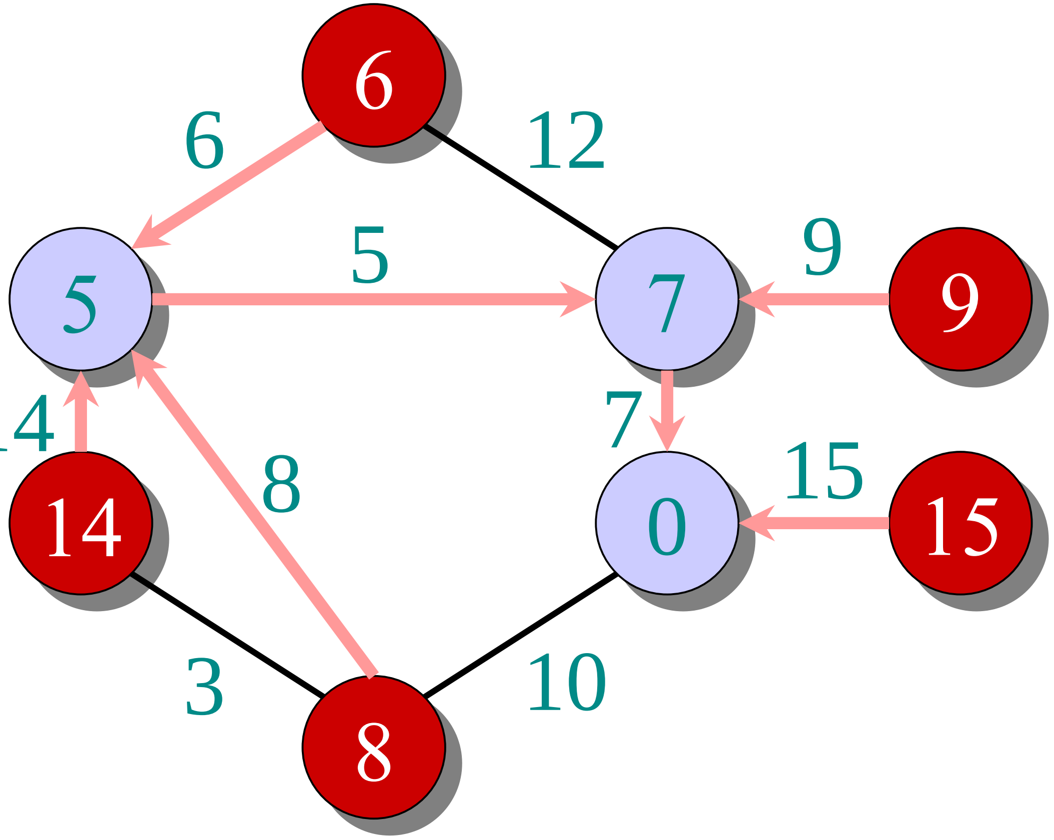


Example of Prim's algorithm



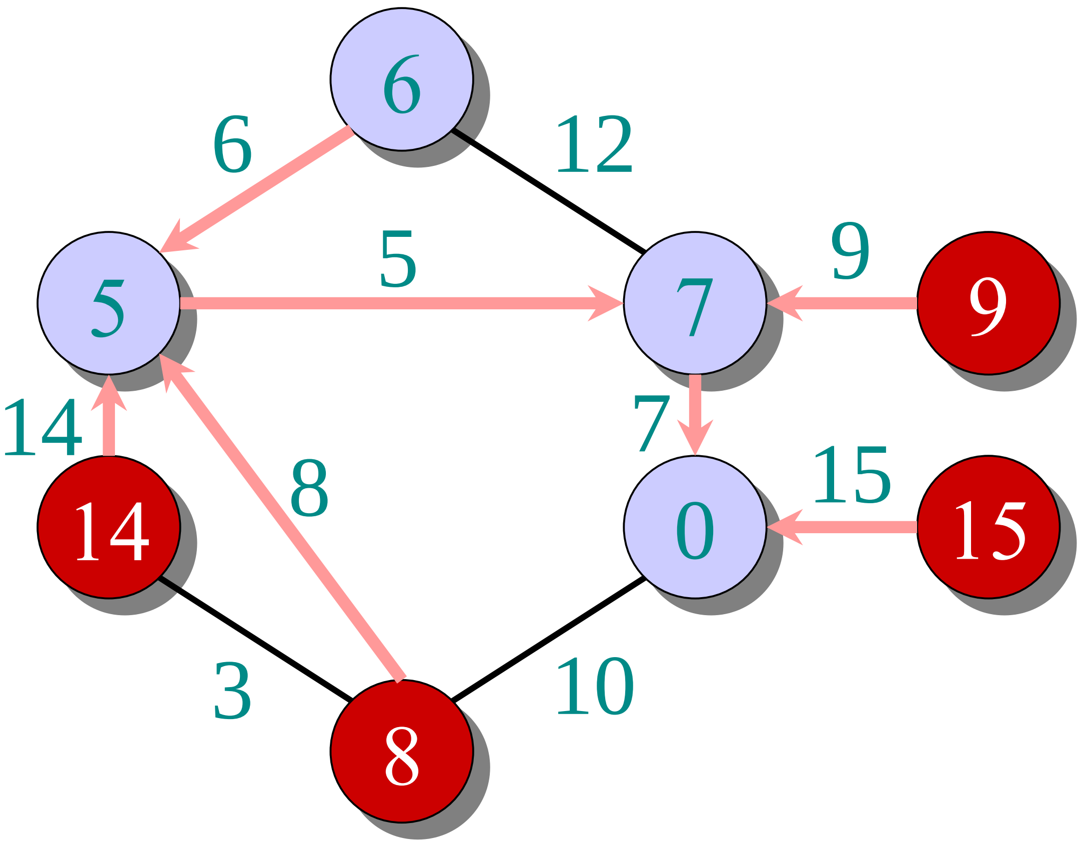
Example of Prim's algorithm

- $\in A$
- $\in V - A$



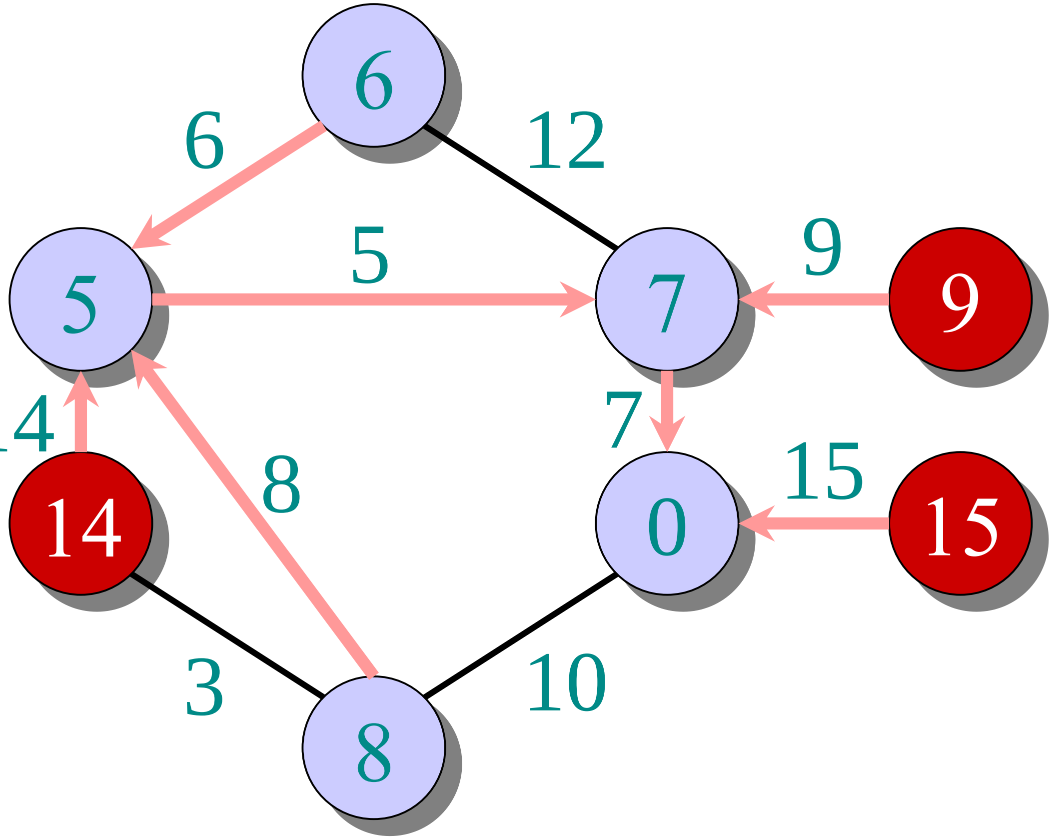
Example of Prim's algorithm

- $\in A$
- $\in V - A$



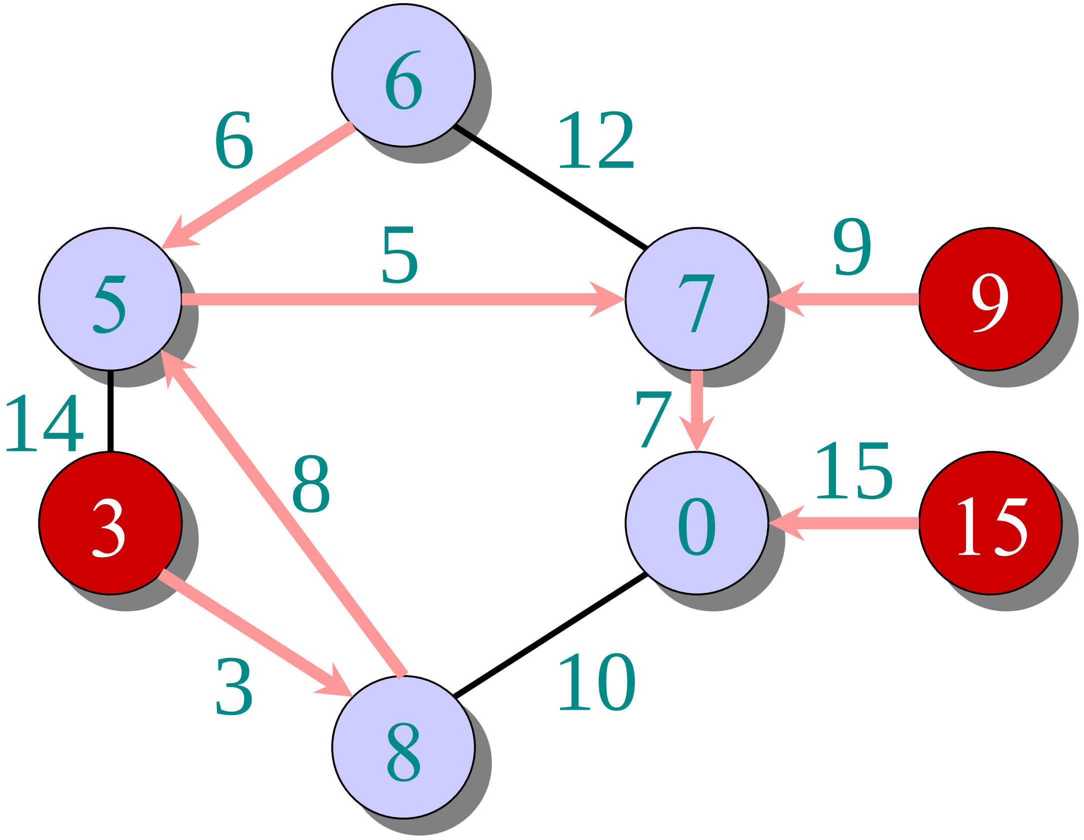
Example of Prim's algorithm

- $\in A$
- $\in V - A$



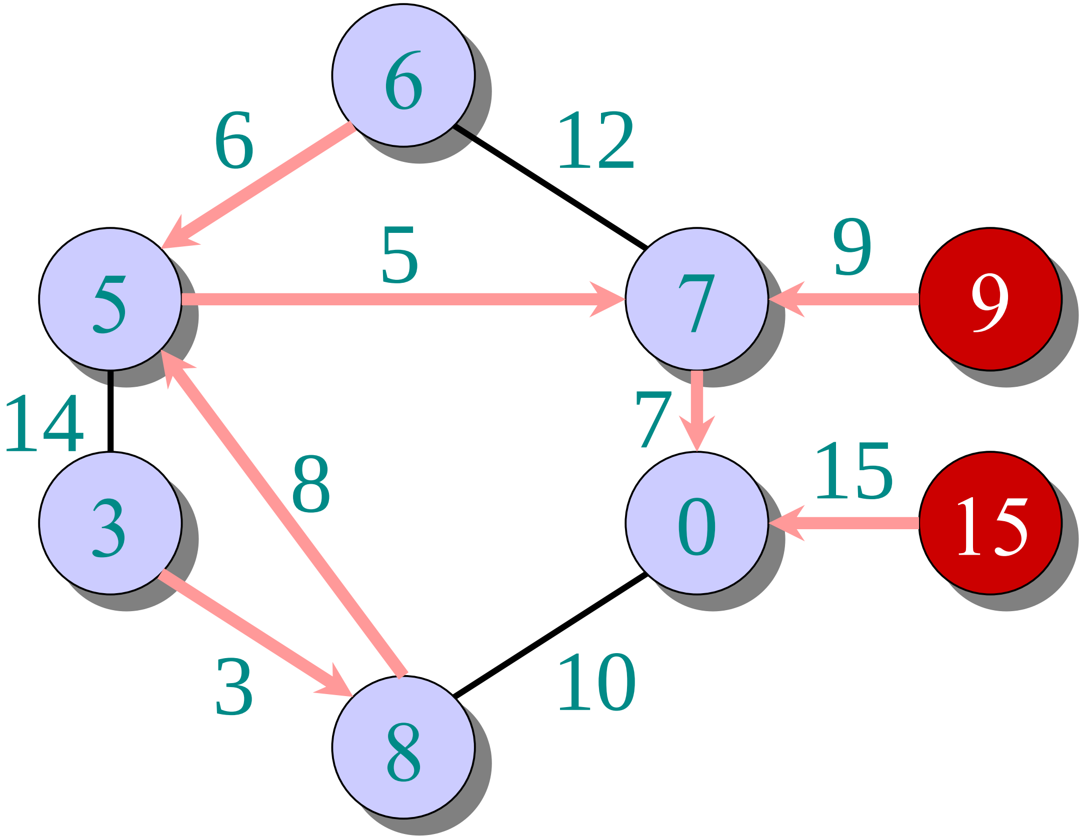
Example of Prim's algorithm

- $\in A$
- $\in V - A$



Example of Prim's algorithm

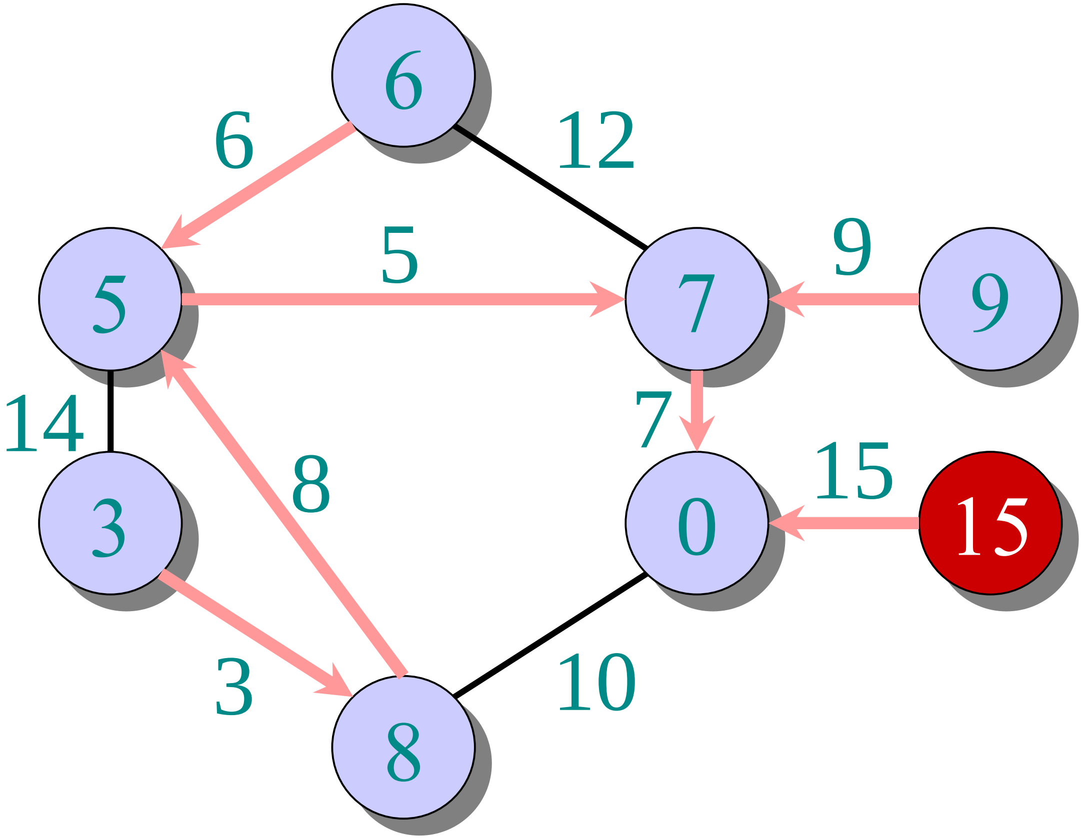
- $\in A$
- $\in V - A$



Example of Prim's algorithm

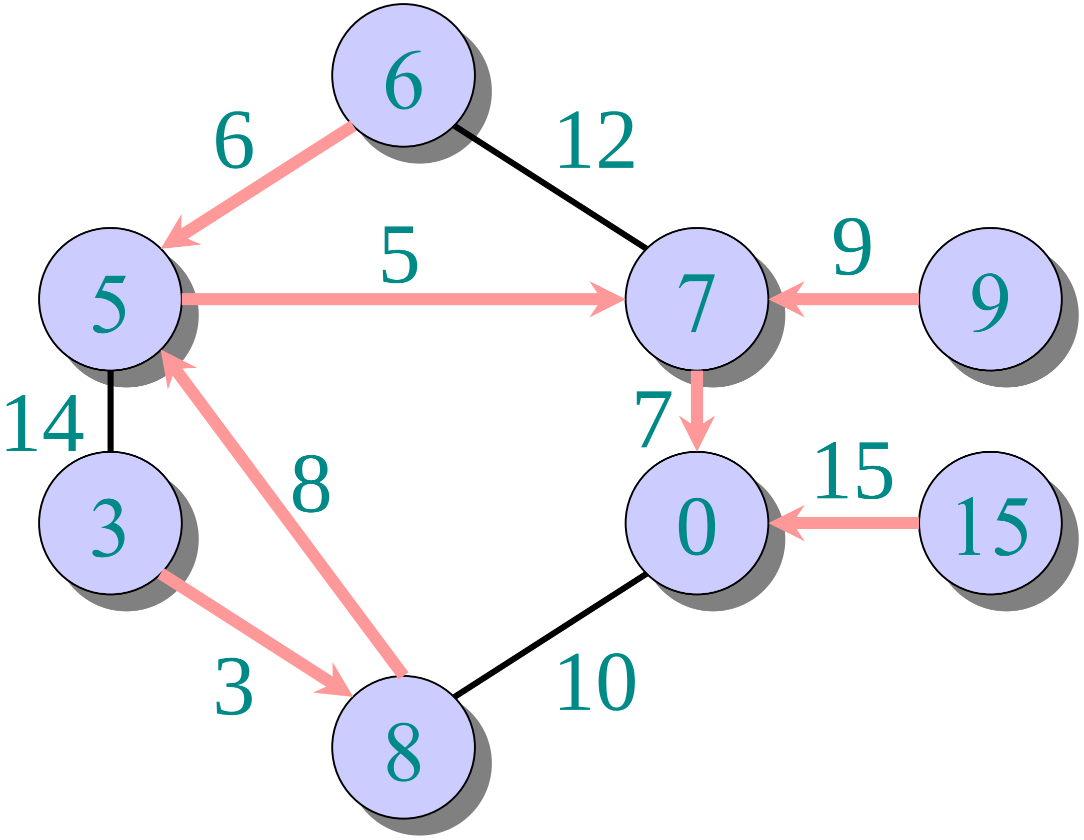
○ $\in A$

● $\in V - A$



Example of Prim's algorithm

- $\in A$
- $\in V - A$



Analysis of Prim

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

Analysis of Prim

$$\left.\begin{array}{r}\Theta(V)\\ \text{total}\end{array}\right\}\begin{array}{l} Q \leftarrow V \\ key[v] \leftarrow \infty \text{ for all } v \in V \\ key[s] \leftarrow 0 \text{ for some arbitrary } s \in V \end{array}$$

while $Q \neq \emptyset$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    for each $v \in Adj[u]$
      do if $v \in Q$ and $w(u, v) < key[v]$
        then $key[v] \leftarrow w(u, v)$
          $\pi[v] \leftarrow u$

Analysis of Prim

$\Theta(V)$ total

$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$
while $Q \neq \emptyset$

$|V|$ times

do $u \leftarrow$ EXTRACT-MIN(Q)
  for each $v \in Adj[u]$
    do if $v \in Q$ and $w(u, v) < key[v]$
      then $key[v] \leftarrow w(u, v)$
        $\pi[v] \leftarrow u$

Analysis of Prim

$\Theta(V)$ total:

$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$

while $Q \neq \emptyset$

$|V|$ times:

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 $key[v] \leftarrow w(u, v)$
$\pi[v] \leftarrow u$

Analysis of Prim

$\Theta(V)$ total:

$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$

while $Q \neq \emptyset$

$|V|$ times:

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 $key[v] \leftarrow w(u, v)$
$\pi[v] \leftarrow u$

Handshaking Lemma $\Rightarrow \Theta(E)$ implicit DECREASE-KEY's.

Analysis of Prim

$\Theta(V)$ total:

$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$

while $Q \neq \emptyset$

$|V|$ times:

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 $key[v] \leftarrow w(u, v)$
$\pi[v] \leftarrow u$

Handshaking Lemma $\Rightarrow \Theta(E)$ implicit DECREASE-KEY's.

Time $= \Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

Analysis of Prim (continued)

Time $= \Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

Analysis of Prim (continued)

$$\text{Time} = \Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total

Analysis of Prim (continued)

$$\text{Time} = \Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$

Analysis of Prim (continued)

Time = $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$

Analysis of Prim (continued)

Time = $\Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$O(1)$ amortized	$O(E + V \lg V)$ worst case

Other greedy algorithms for MST

- Boruvka’s algorithm
- Kruskal’s algorithm
  - Use the same greedy-choice property we used
  - Both run in time $O(E \log V)$, though Boruvka’s algorithm is usually the method of choice in practice.

Other greedy algorithms for MST

- Boruvka’s algorithm
- Kruskal’s algorithm
    - Use the same greedy-choice property we used
    - Both run in time $O(E \log V)$, though Boruvka’s algorithm is usually the method of choice in practice.

Best to date:

- Karger, Klein, and Tarjan [1993].
- Randomized algorithm.
- $O(V + E)$ expected time.