Analysis and Design of Algorithms

CS3230

Week 12 (Part 2) + Summary (Part 2)

Graph Algorithms

Diptarka Chakraborty

Ken Sung

Plan for Today

We review

- ❑ Dynamic Programming
- ❑ Greedy Algorithms
- ❑ Approximation Algorithms
- ❑ Randomized Algorithms
- ❑ Intractability and NP-Completeness
- ❑ Amortized Analysis

Plan for Today

We review

- ❑ **Dynamic Programming**
- ❑ Greedy Algorithms
- ❑ Approximation Algorithms
- ❑ Randomized Algorithms
- ❑ Intractability and NP-Completeness
- ❑ Amortized Analysis

Dynamic Programming algorithm paradigm

- Expressing the solution recursively
- Overall there are only polynomial number of subproblems
- But there is a huge overlap among the subproblems. So the recursive algorithm takes exponential time (solving same subproblem multiple times)
- So we compute the recursive solution iteratively in a bottom-up fashion. This avoids wastage of computation and leads to an efficient implementation
- To speed-up the computation one may also use top-down approach using recursion with the help of memoization

Things to remember

- Key to DP is figuring out **optimal substructure**: the optimal solution to your problem can be found from optimal solutions to one or more subproblems.

- Figure out how to index your subproblems. For example, if you have a sequence $a_1, \ldots, a_n$, trying the i'th subproblem being the answer for the sequence from 1 to i is a good place to start. If there are integers in the problem upper bounded by some integer M, you probably need to index over every integer from 1 to M.

- Even if you don't have overlapping subproblems, you can still use DP. But the more overlap you have, the faster DP is compared to recursion.

Shortest path in a graph

Revisiting the problem

Problem Definition

Input: A directed graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ with $\boldsymbol{\omega}: \boldsymbol{E} \rightarrow \boldsymbol{R}$ and a source vertex $\boldsymbol{s} \in \boldsymbol{V}$

Notations:

$\boldsymbol{n} = |\boldsymbol{V}|$, $\boldsymbol{m} = |\boldsymbol{E}|$

$\boldsymbol{\delta}(\boldsymbol{u}, \boldsymbol{v})$: distance from $\boldsymbol{u}$ to $\boldsymbol{v}$.

$\boldsymbol{P}(\boldsymbol{u}, \boldsymbol{v})$: The shortest path from $\boldsymbol{u}$ to $\boldsymbol{v}$.

Aim:

- Compute $\boldsymbol{\delta}(\boldsymbol{s}, \boldsymbol{v})$ for all $\boldsymbol{v} \in \boldsymbol{V} \backslash \{\boldsymbol{s}\}$
- Compute $\boldsymbol{P}(\boldsymbol{s}, \boldsymbol{v})$ for all $\boldsymbol{v} \in \boldsymbol{V} \backslash \{\boldsymbol{s}\}$

If $\boldsymbol{\omega}: \boldsymbol{E} \rightarrow \boldsymbol{R}^{+}$, Dijkstra's algorithm solves the problem in **O**$(\boldsymbol{m} + \boldsymbol{n}$ **log** $\boldsymbol{n})$ time.

Question: How crucial is the non-negative weights for Dijkstra's algorithm ?

Revisiting Dijkstra's algorithm

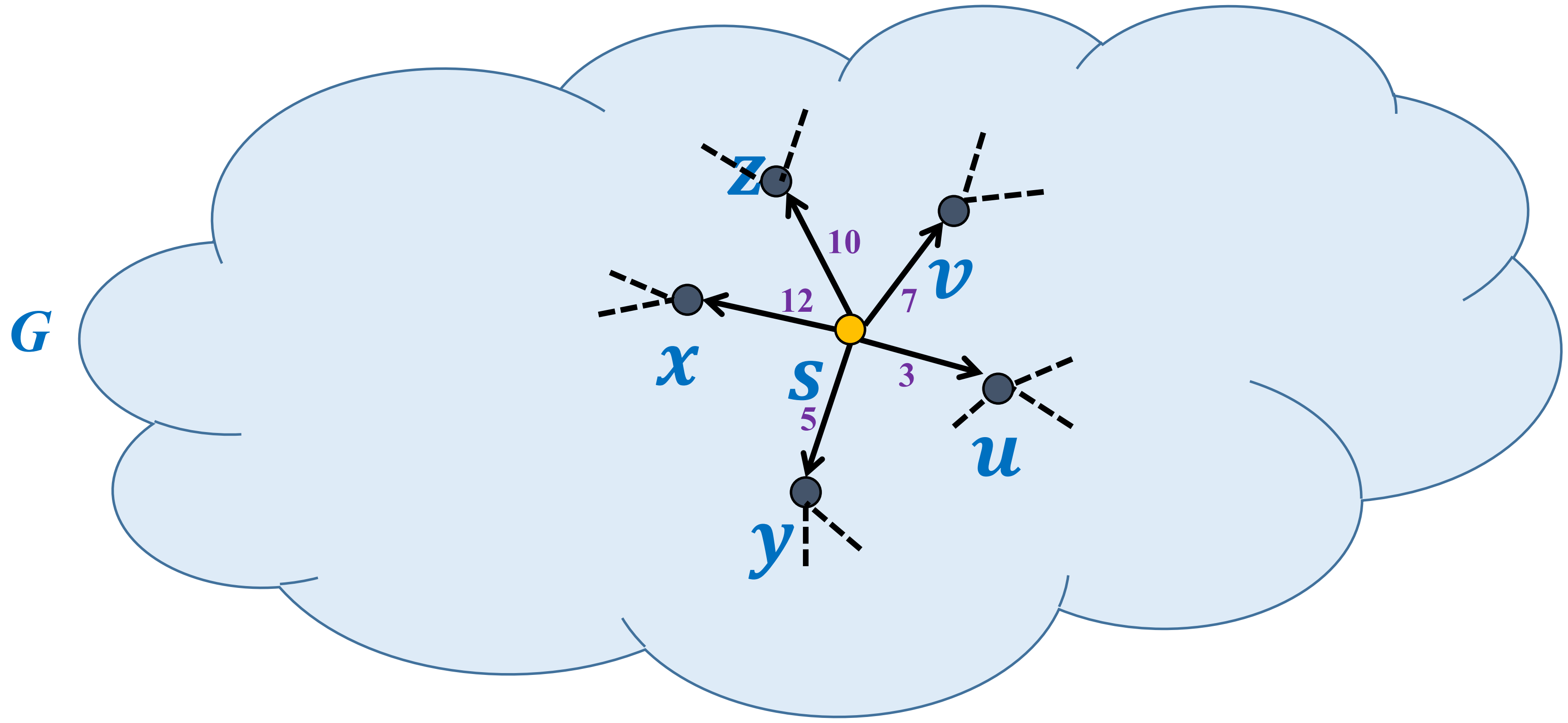
Two crucial facts exploited in **Dijkstra**'s algorithm.

- **Fact** 1: *the nearest neighbor* is also the vertex **nearest** to s.
- **Fact** 2: **Optimal subpath** property

Question:

Which of these two facts get violated if edge weights are potentially negative ?

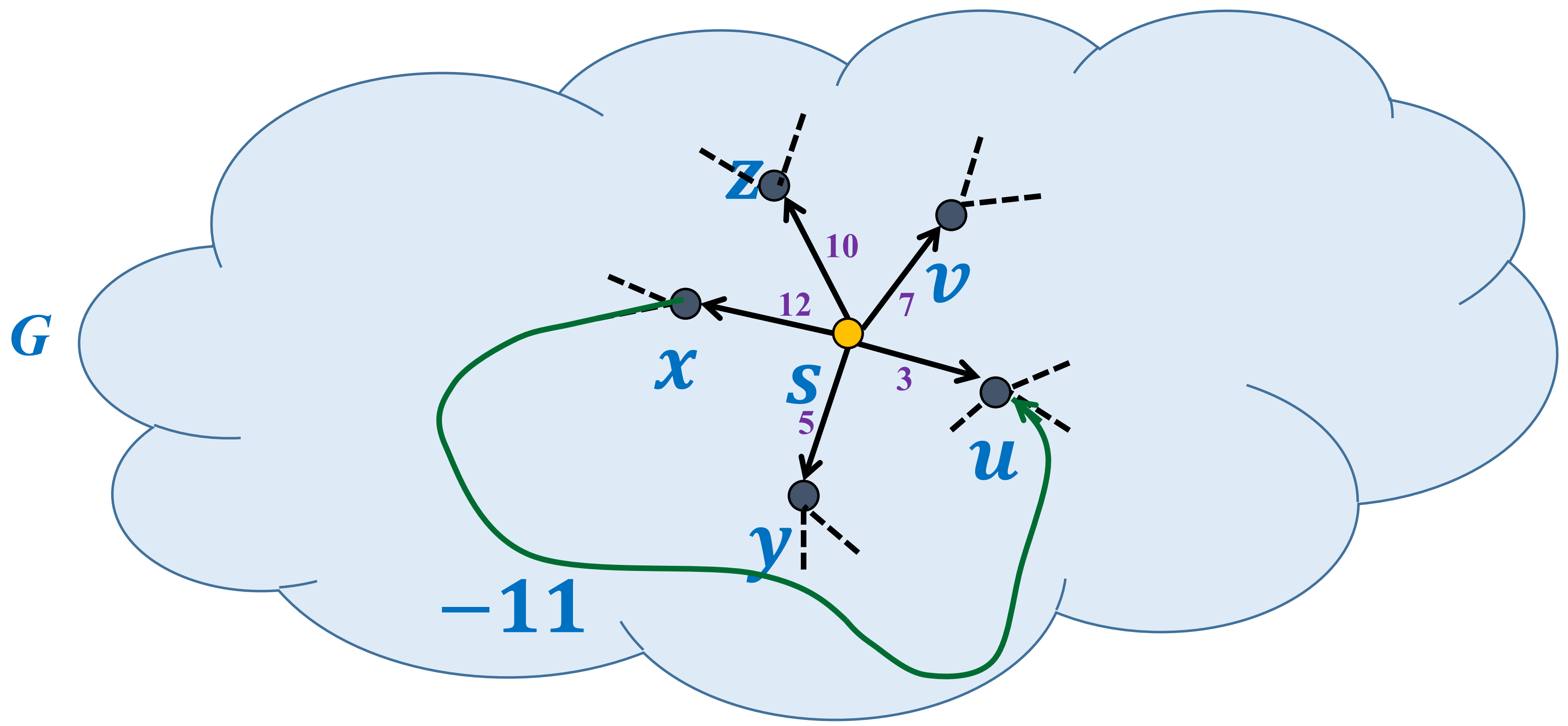
Violation of Fact 1



Question: Can we be certain about the distance from s to any vertex in this picture?

Answer: NO.☹

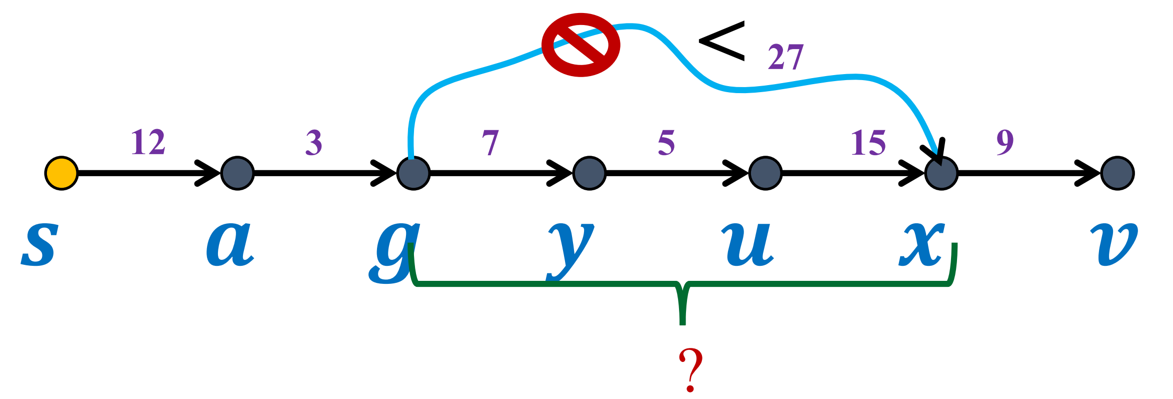
Violation of Fact 1



Violation of Fact 2 Optimal subpath property

Consider any shortest path $P(s, v)$.

$\delta(s, v) = 51$



Lemma 1: Every **subpath** of a shortest path is also a shortest path.

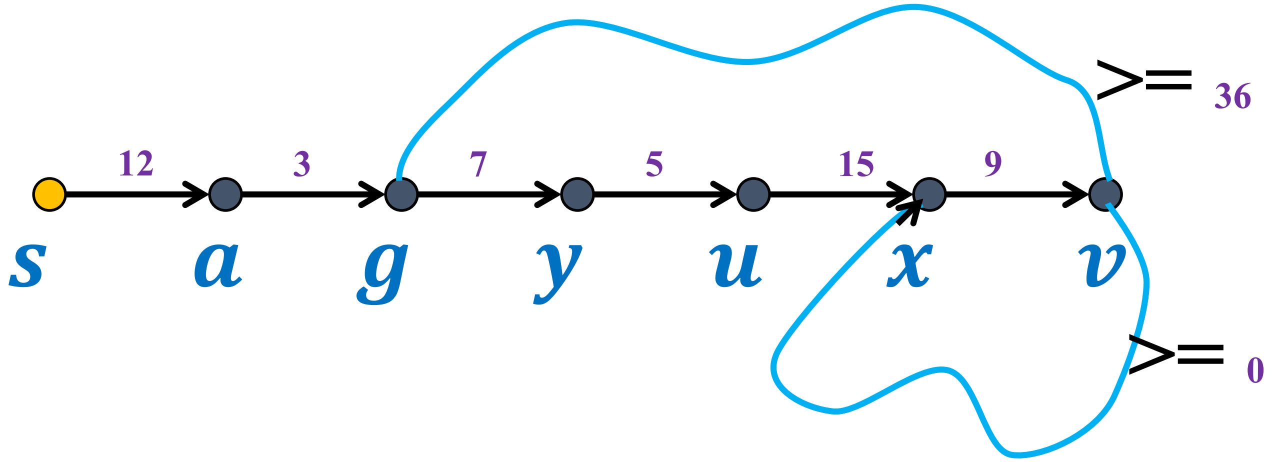
Proof: If not, then replacing the sub-path by the shortest path will give us even shortest path from s to v.

[This proof assumes that the shortest path from s to x can not pass through v. So one must justify the validity of this assumption.]

Violation of Fact 2 Optimal subpath property

Consider any shortest path $\boldsymbol{P}(\boldsymbol{s}, \boldsymbol{v})$.

$\boldsymbol{\delta}(\boldsymbol{s}, \boldsymbol{v})$= 51



If the shortest path $\boldsymbol{P}(\boldsymbol{s}, \boldsymbol{x})$ passes through $\boldsymbol{v}$, then we shall be covering even longer distance to reach $\boldsymbol{x}$ from $\boldsymbol{s}$. Hence the shortest path from $\boldsymbol{s}$ to $\boldsymbol{x}$ can not pass through $\boldsymbol{v}$.

[See the crucial role played by the fact that the edge weights are nonnegative]

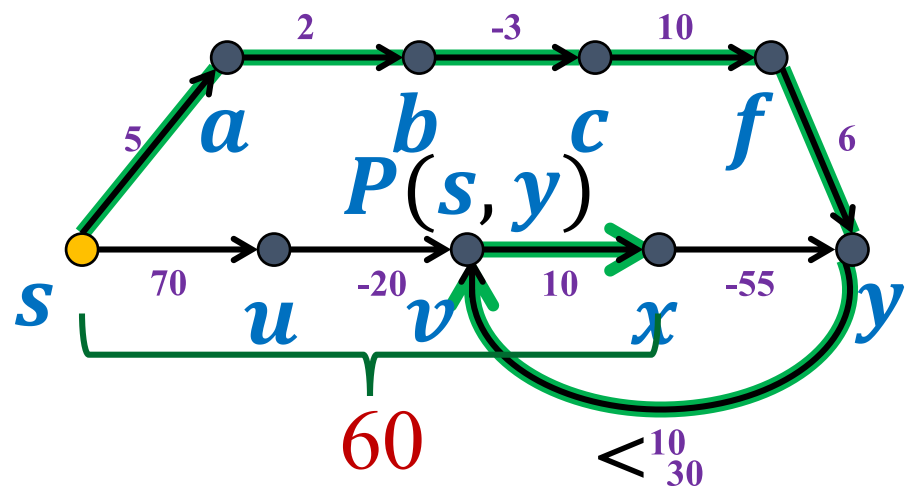
Violating the Optimal substructure property

in graphs having potentially negative edge

Violating the Optimal substructure property

$\delta(s, y)$ = 5

$\delta(s, x)$ = 40

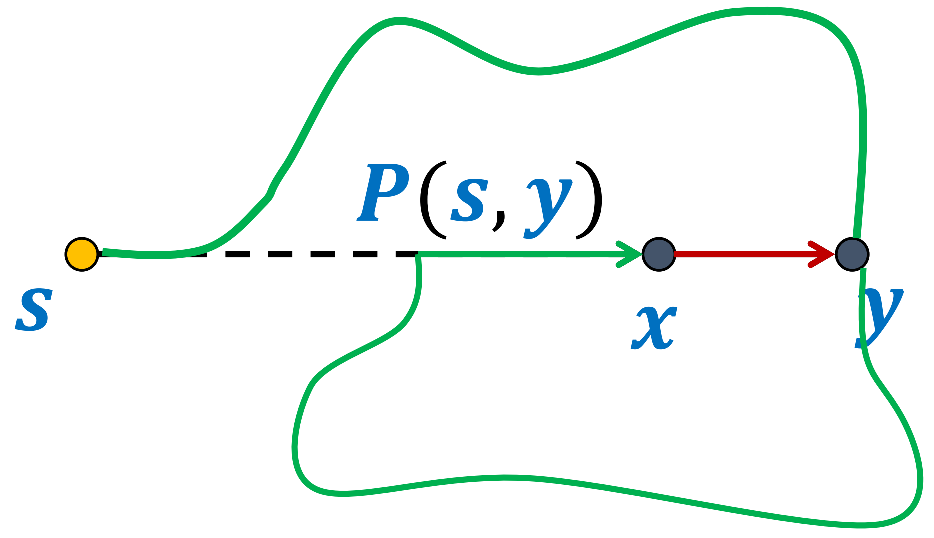


This example illustrates that optimal substructure property gets violated when the edge weights are negative.

But instead of giving up hope by seeing this example, we should have patience, and we should try to find out the **precise condition** under which the optimal substructure property is violated.

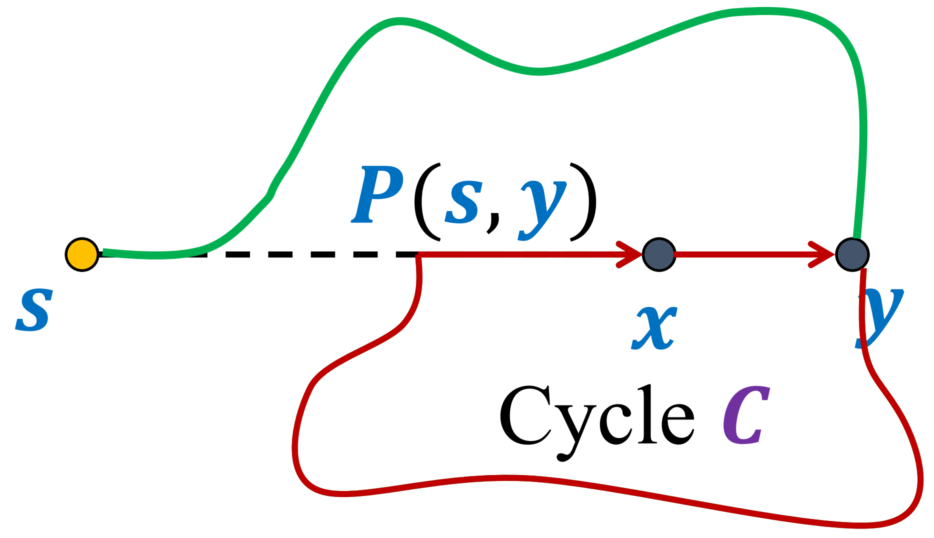
For this purpose, we need to consider a generic example.

Violating the Optimal substructure property



Question: When can the shortest $s \rightsquigarrow x$ path pass through y ?

Violating the Optimal substructure property



Question: When can the shortest $s \rightsquigarrow x$ path pass through y ?

Answer: Weight of the cycle C must be **NEGATIVE**.

Theorem: Given a directed $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ with $\boldsymbol{\omega}: \boldsymbol{E} \rightarrow \boldsymbol{R}$,

all shortest paths in $\boldsymbol{G}$ possess optimal substructure property **if** there is no negative cycle.

Our patience proved helpful.☺

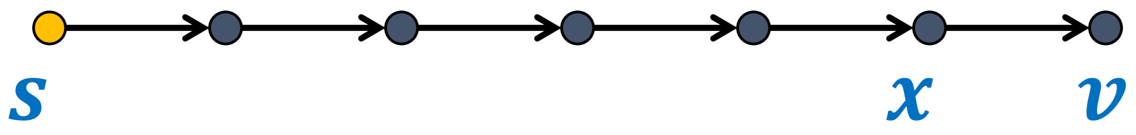
Shortest paths in a graph

Negative weights

BUT No negative cycle

Exploiting the Optimal substructure property

Consider once again a shortest path $\boldsymbol{P(s,v)}$.



Question: If $\boldsymbol{P(s,v)}$ has $\boldsymbol{i}$ edges, and $(\boldsymbol{x},\boldsymbol{v})$ is the last edge on this path, what can we infer about the subpath $\boldsymbol{P(s,x)}$?

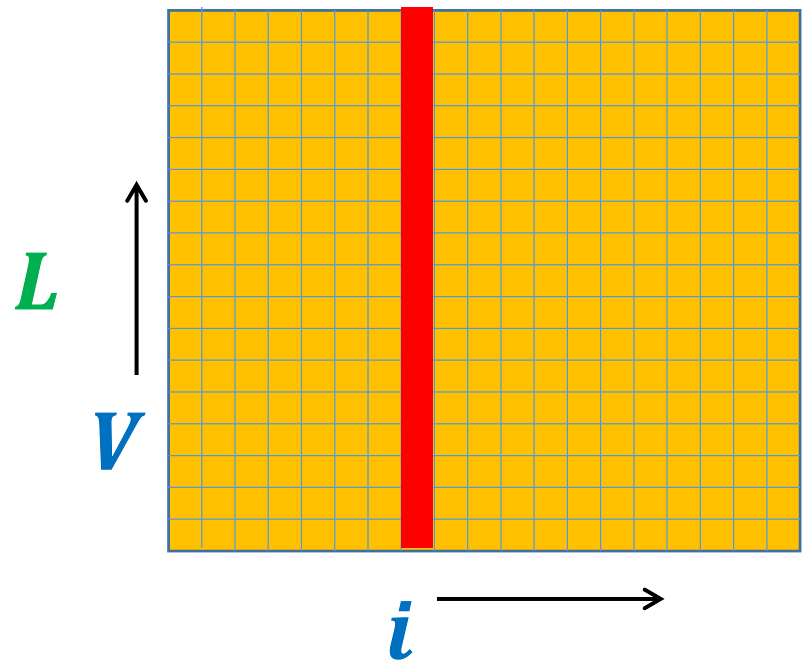
Answer: $\boldsymbol{P(s,x)}$

- is shortest.
- $\boxed{\text{has } \boldsymbol{i-1} \text{ edges.}}$

Exploiting the Optimal substructure property

$L(v, i)$: Weight of the shortest $s \rightsquigarrow v$ having **at most** i edges.

Aim: To compute $L(v, n-1)$ for each v.

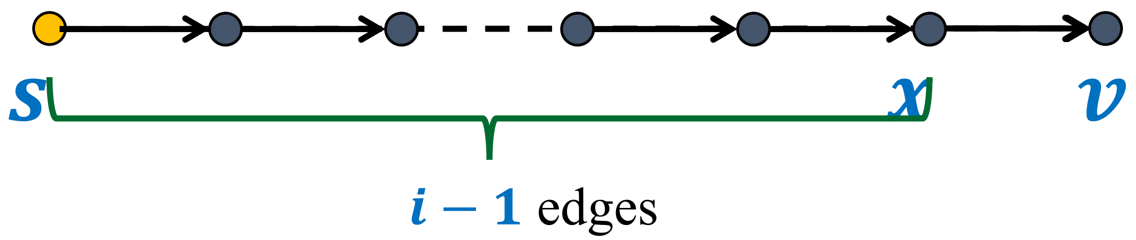


Question: How to find a recursive formulation for $L(v, i)$?

Recursive formulation for $\boldsymbol{L}(\boldsymbol{v}, \boldsymbol{i})$

Recursive Formulation for $L(v,i)$

$L(v,i)$: Weight of the shortest $s \rightsquigarrow v$ having **at most** i edges.



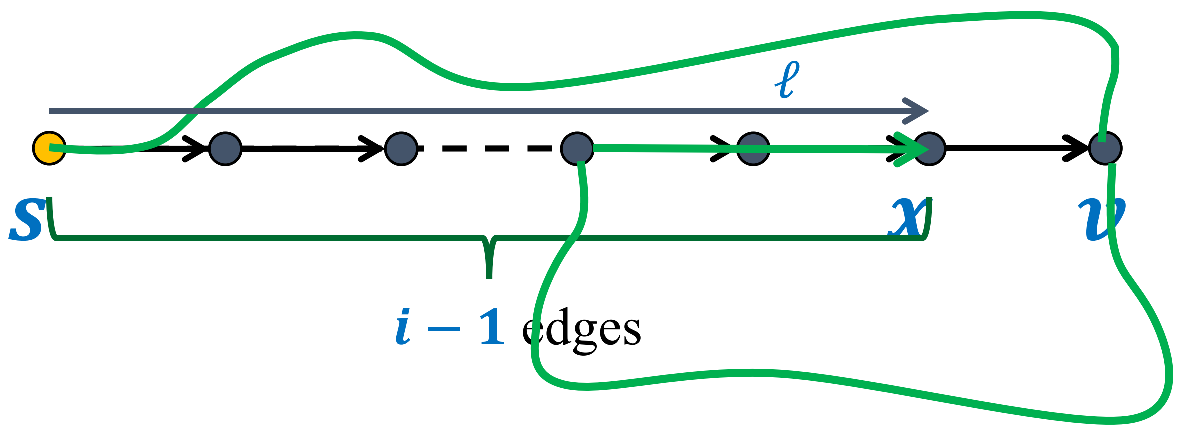
Case 1: If $L(v,i)$ has $< i$ **edges** $L(v,i) = L(v,i-1)$

Case 2: If $L(v,i)$ has exactly i **edges** $L(v,i) = \min_{((x,v)\in E)} \left(L(x,i-1) + \omega(x,v)\right)$

Note : Case 2 assumes optimal sub-structure property of $L(v,i)$.

Though we gave a proof for optimal sub-structure of $P(s,x)$, we need to prove the optimal sub-structure property for $L(v,i)$.

Optimal substructure property for $L(v, i)$

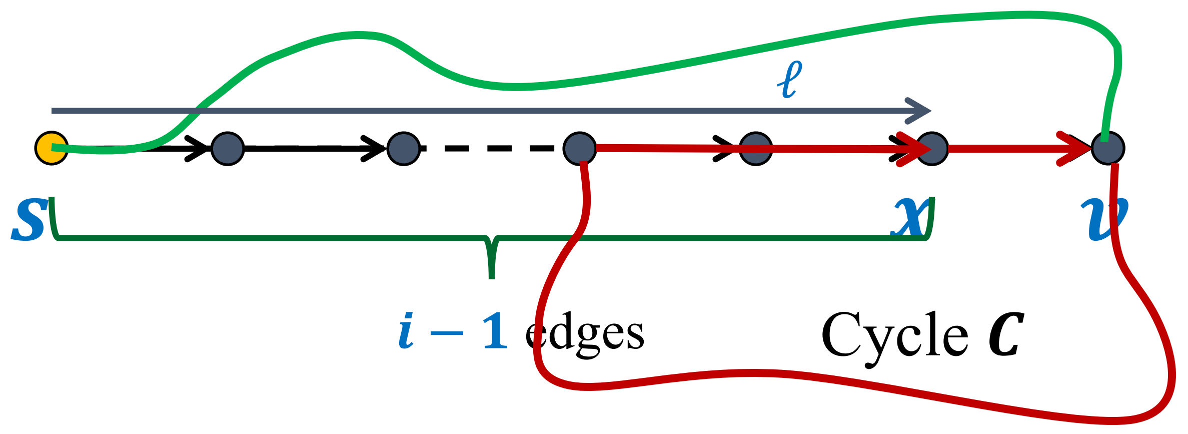


Suppose the path corresponding to $L(x, i-1)$ passes through v.

So $L(x, i-1) \geq \ell$

Question: What can we say about $L(x, i-1)$ in this case ?

Optimal substructure property for $L(v, i)$



Suppose the path corresponding to $L(x, i-1)$ passes through v.

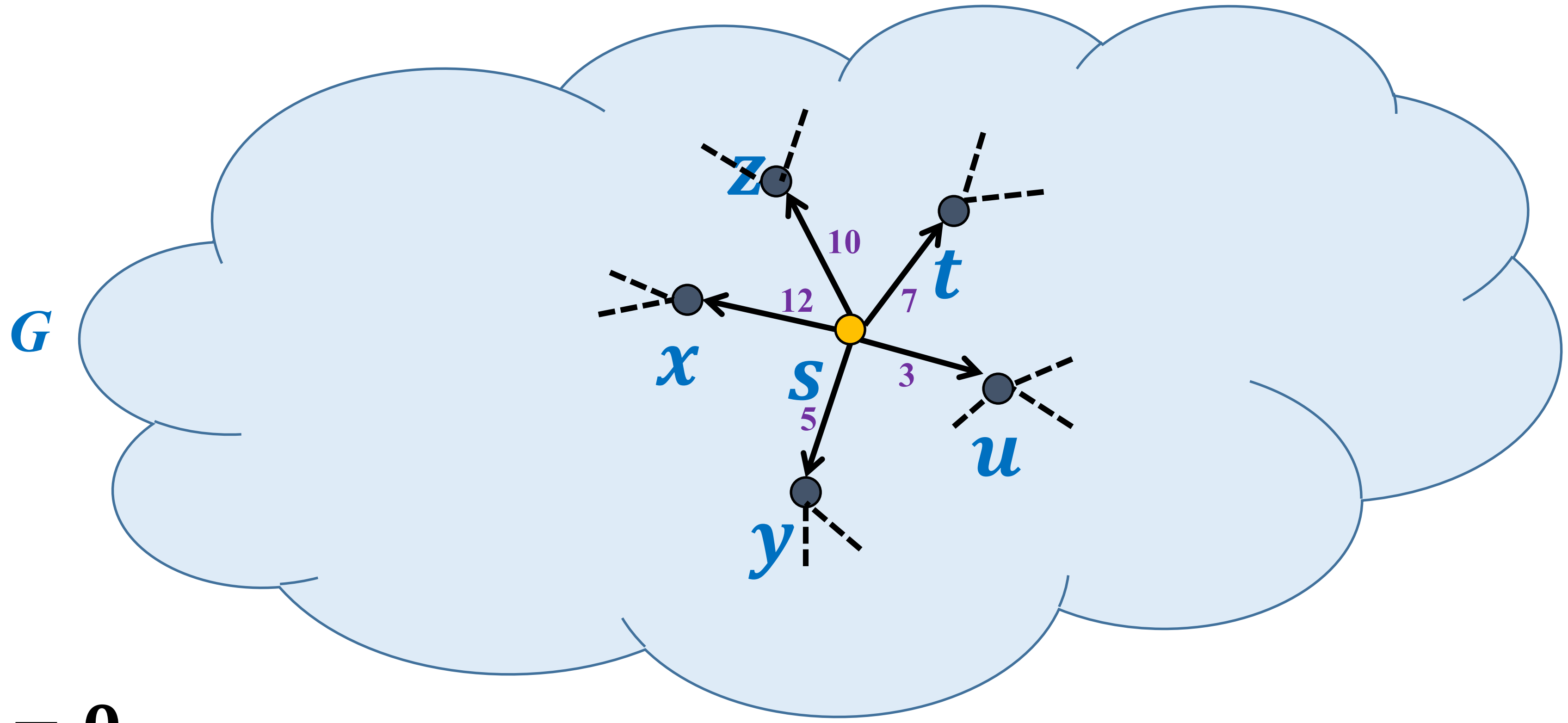
So $L(x, i-1) \geq \ell$

Question: What can we say about $L(x, i-1)$ in this case ?

Answer: $L(x, i-1) \geq \ell + \omega(C)$
$\geq \ell$

A contradiction!

Base case: $L(v, 1)$



$L[s, 1] = 0.$

If $(s, v) \in E$ then $L[v, 1] = \boxed{\omega(s, v)}$

else $L[v, 1] = \boxed{\infty}$

Bellman-Ford Algorithm

For shortest paths in a graph with

Negative weights

BUT No negative cycle

Bellman-Ford's algorithm

```
Bellman-Ford-algo($s$,$G$)
{
  For each $v \in V \backslash \{s\}$ do
        If $(s, v) \in E$  then $L[v, 1] \leftarrow \omega(s, v)$
                           else $L[v, 1] \leftarrow \infty$;
  $L[s, 1] \leftarrow 0$;
  For $i$ = $2$ to $n - 1$ do
  {    For each $v \in V$ do
       {
          $L[v, i] \leftarrow L[v, i - 1]$;
          For each $(x, v) \in E$ do
              $L[v, i] \leftarrow$ min( $L[v, i]$ , $L[x, i - 1] + \omega(x, v)$ )
       }
  }
}
```

$\left.\begin{matrix} \\ \\ \end{matrix}\right\}$ Initializing $L[*, 1]$

$\left.\begin{matrix} \\ \\ \end{matrix}\right\}$ Computing $L[v, i]$

Bellman-Ford's algorithm

Lemma: $L[v, i]$ stores the shortest path from s to v having at most i edges.
Proof: Just based on the recurrence we derived for $L(v, i)$ a few slides ago.

Theorem: Given a directed $G = (V, E)$ with $\omega: E \rightarrow R$ and $s \in V$,
if there is no negative cycle, then we can compute shortest paths from s

1. in **O**(mn) time
2. using **O**(n^2) space

Exercise:
How to reduce space to **O**(n) ?
How to extract shortest path ?
Modify the algorithm to detect a negative weight cycle (if present).

Plan for Today

We review

- ❑ **Dynamic Programming**
- ❑ **Greedy Algorithms**
- ❑ Approximation Algorithms
- ❑ Randomized Algorithms
- ❑ Intractability and NP-Completeness
- ❑ Amortized Analysis

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.

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.

Minimum spanning trees

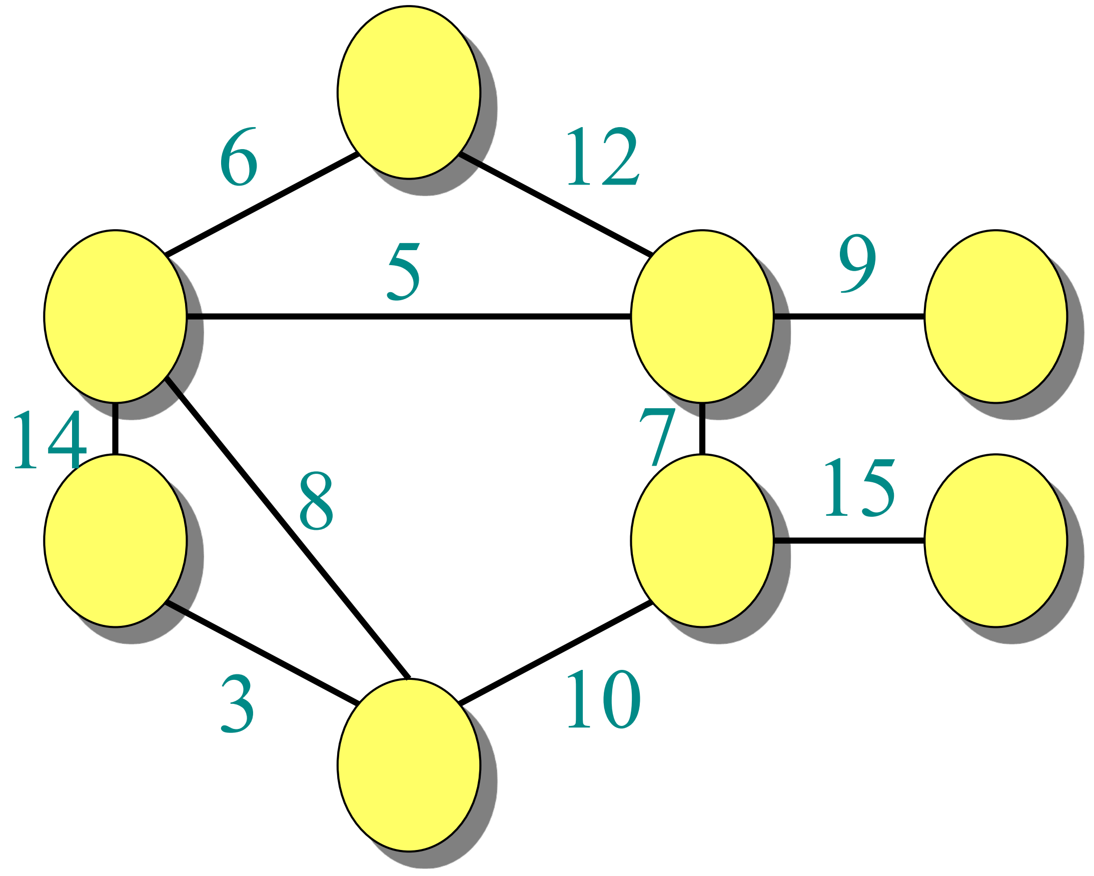
Input: A connected, undirected graph $G = (V, E)$ with weight function $w : E \rightarrow \mathbb{R}$.

- For simplicity, assume that all edge weights are distinct.

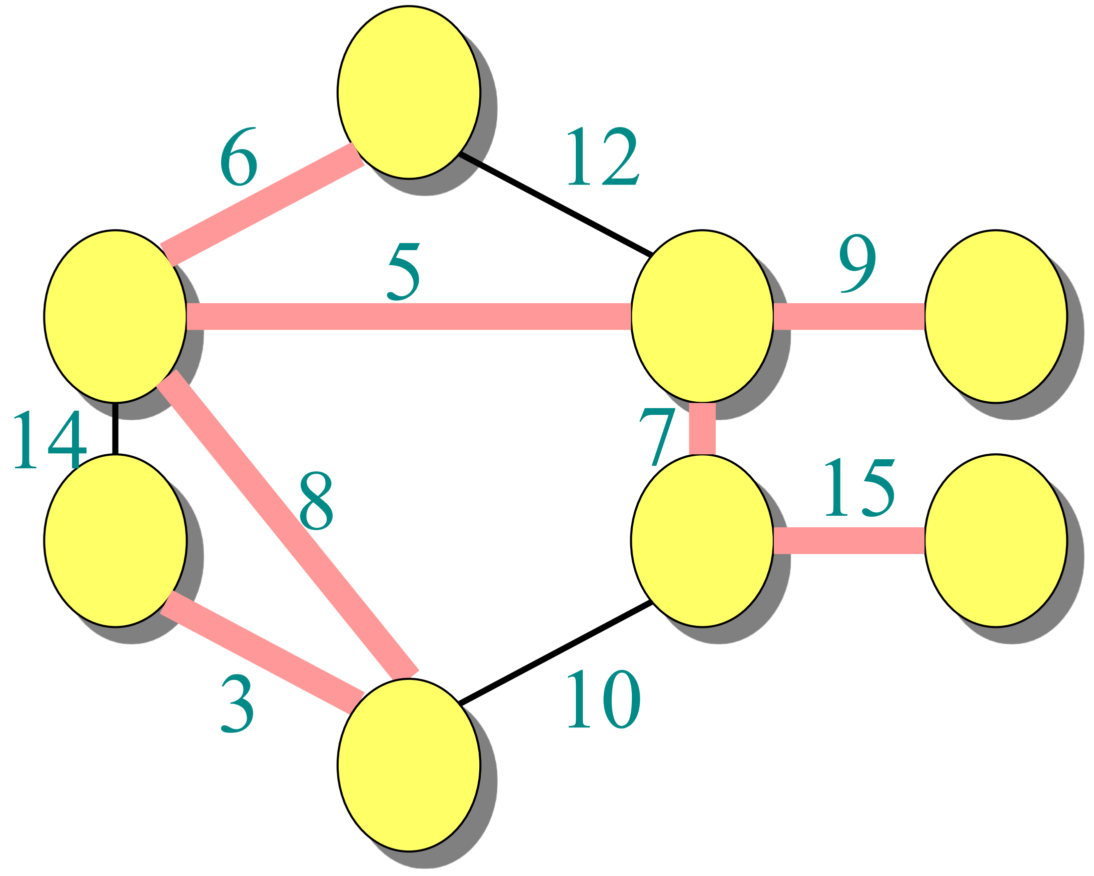
Output: A ***spanning tree*** T — a tree that connects all vertices — of minimum weight:

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

Example of MST



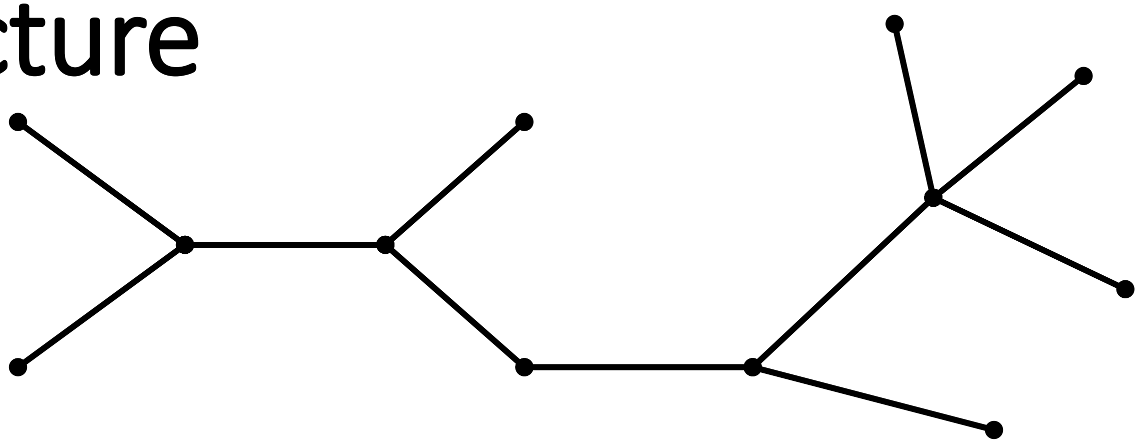
Example of MST



Optimal substructure

MST T:

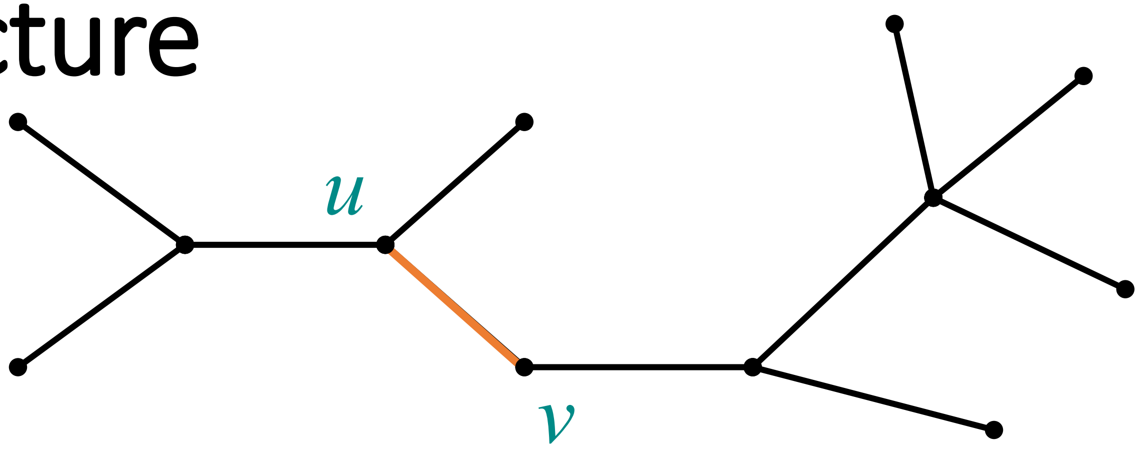
(Other edges of G are not shown.)



Optimal substructure

MST T:

(Other edges of G are not shown.)

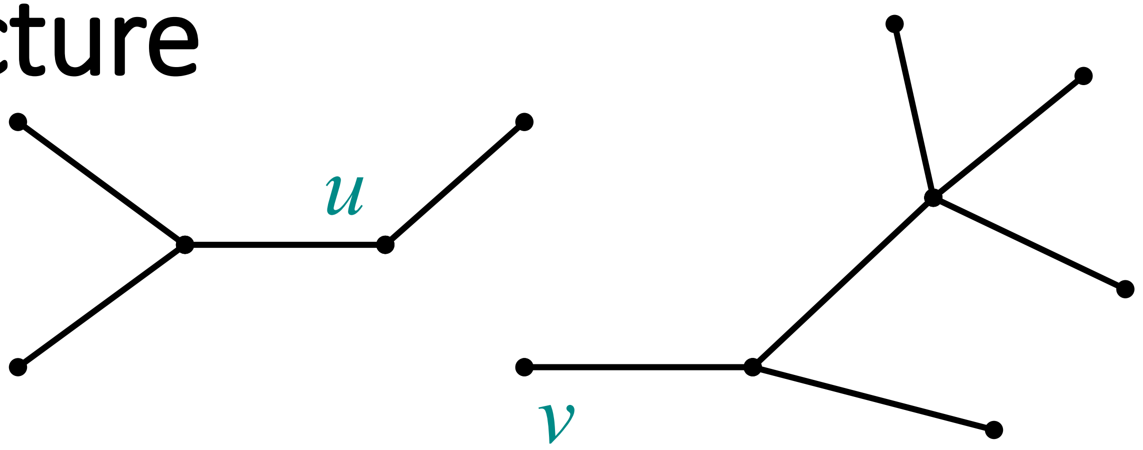


Remove any edge $(u, v) \in T$.

Optimal substructure

MST T:

(Other edges of G are not shown.)

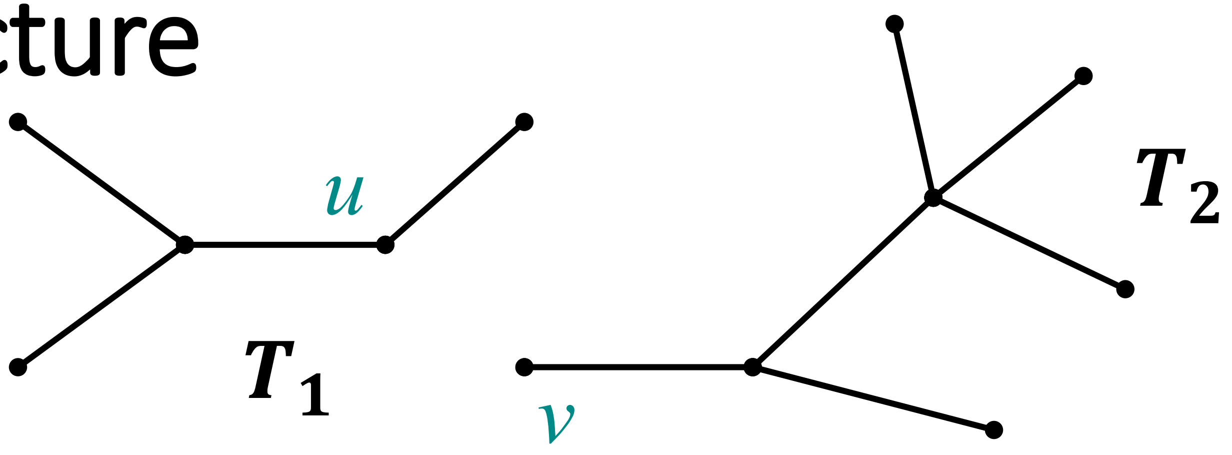


Remove any edge $(u, v) \in T$.

Optimal substructure

MST T:

(Other edges of G are not shown.)

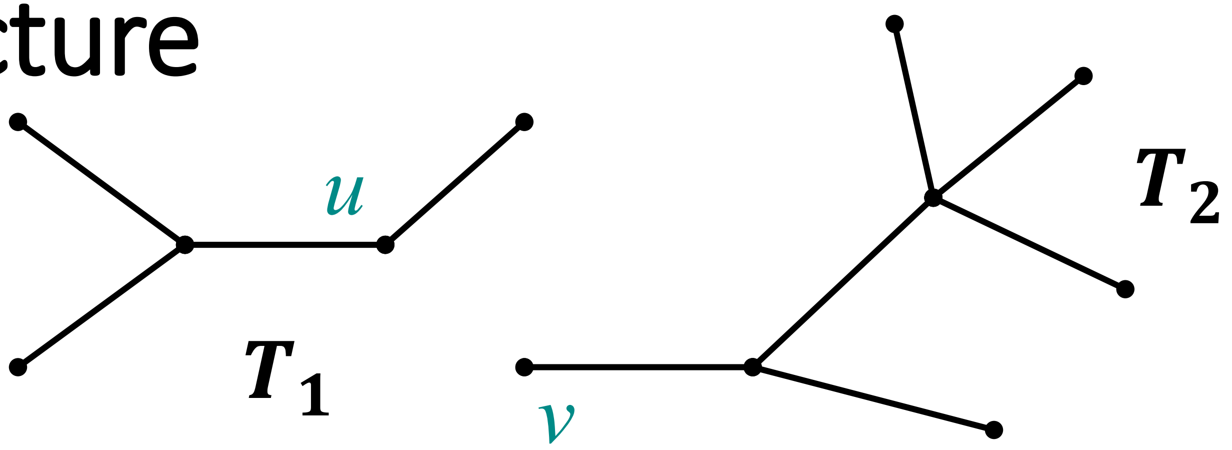


Remove any edge $(u, v) \in T$. Then, T is partitioned into two subtrees T_1 and T_2.

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:
$V_1 =$ 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.

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

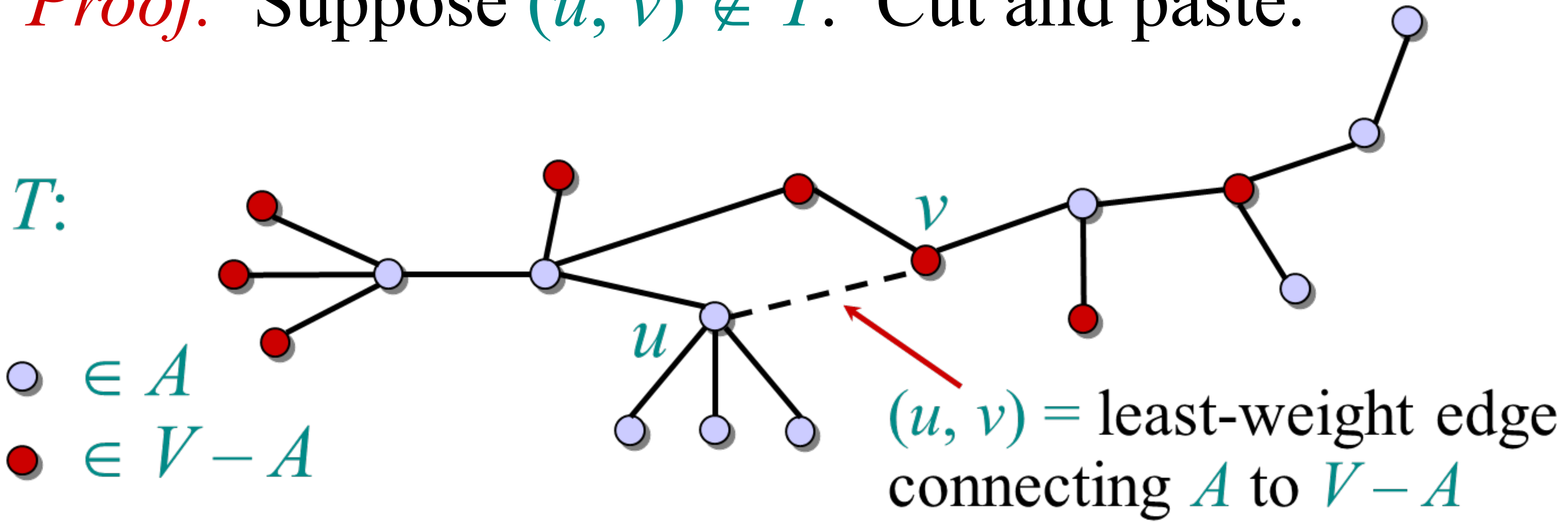
A locally optimal choice is globally optimal.

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?

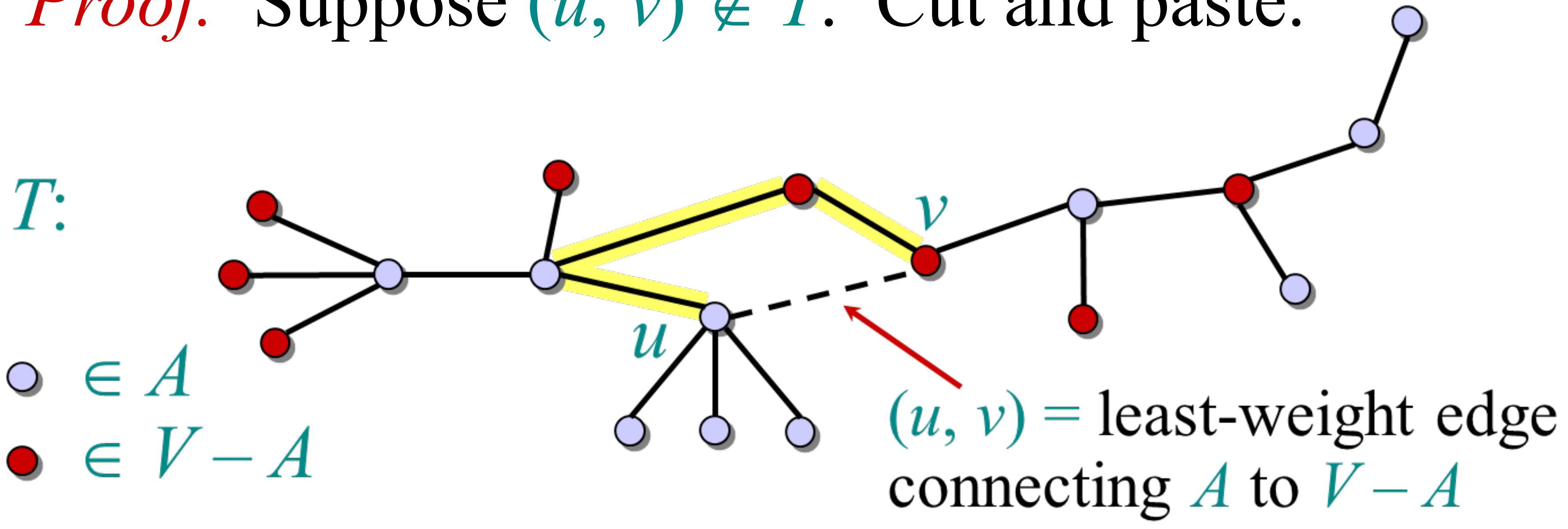
Proof. Suppose $(u, v) \notin T$. Cut and paste.

T:



Greedy-choice property: Why?

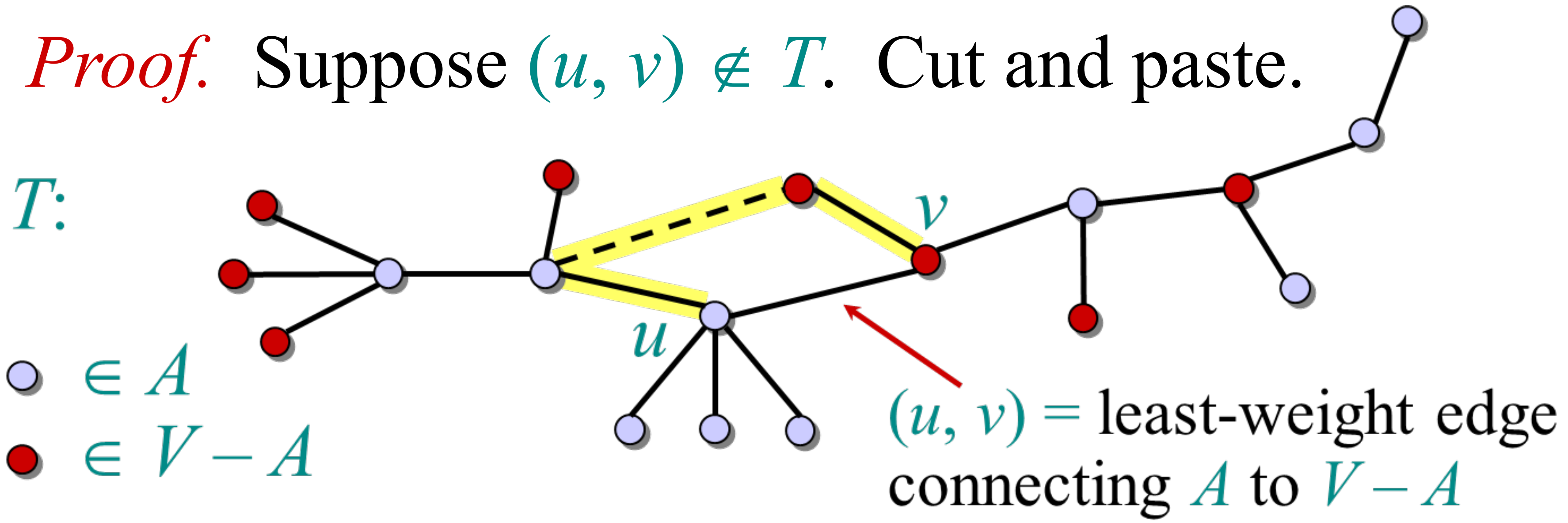
Proof. Suppose $(u, v) \notin T$. Cut and paste.



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

Greedy-choice property: Why?

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



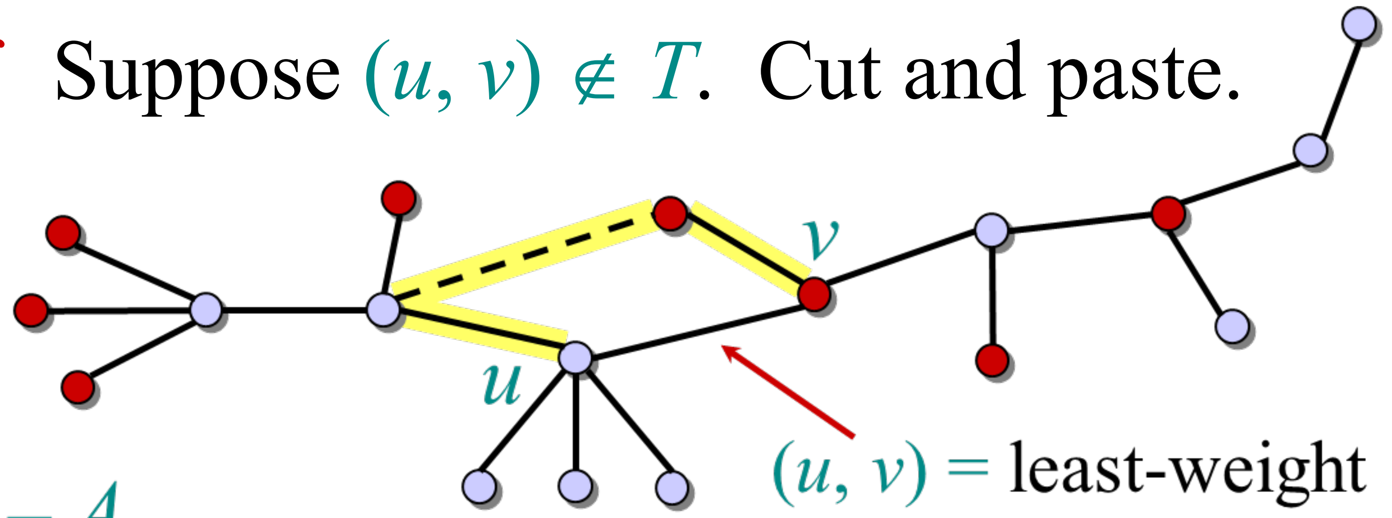
(u, v) = least-weight edge connecting A to $V - A$

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.

T:



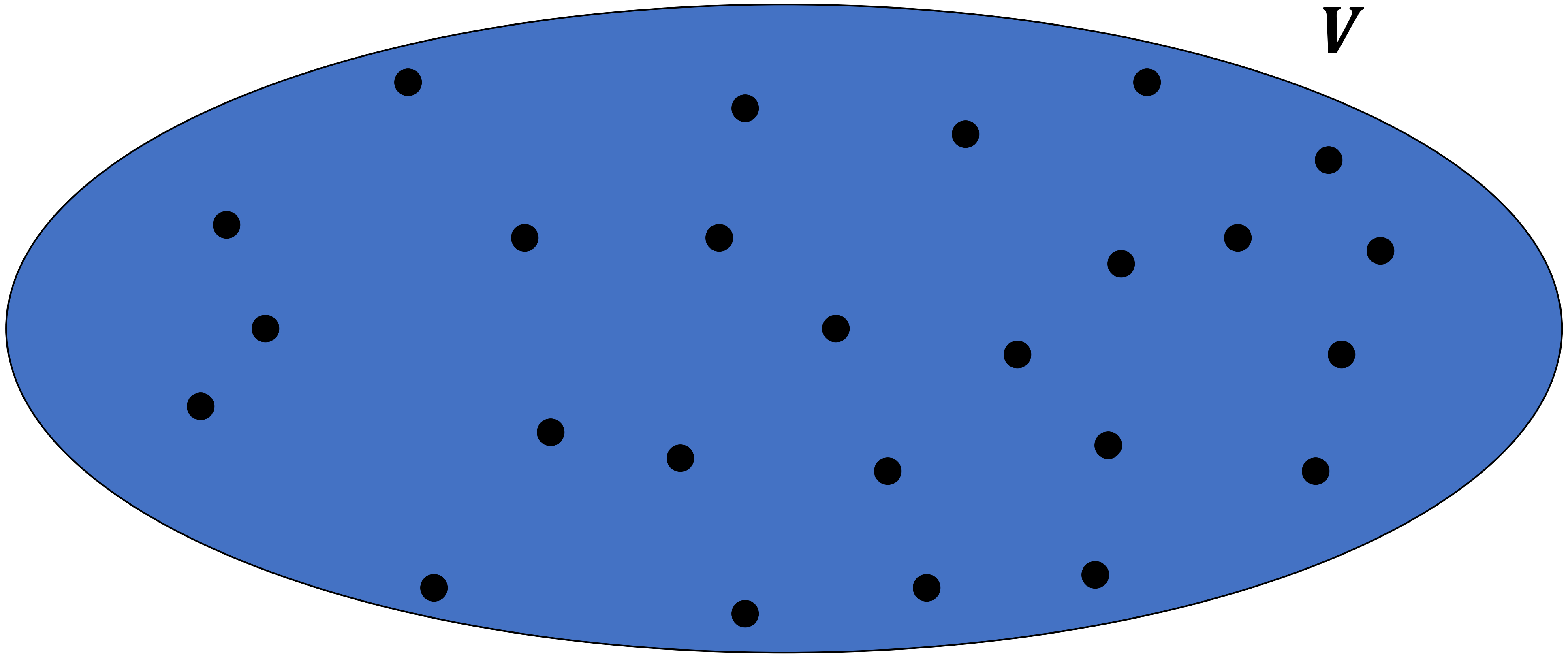
● $\in A$
● $\in V - A$

(u, v) = least-weight edge connecting A to $V - A$

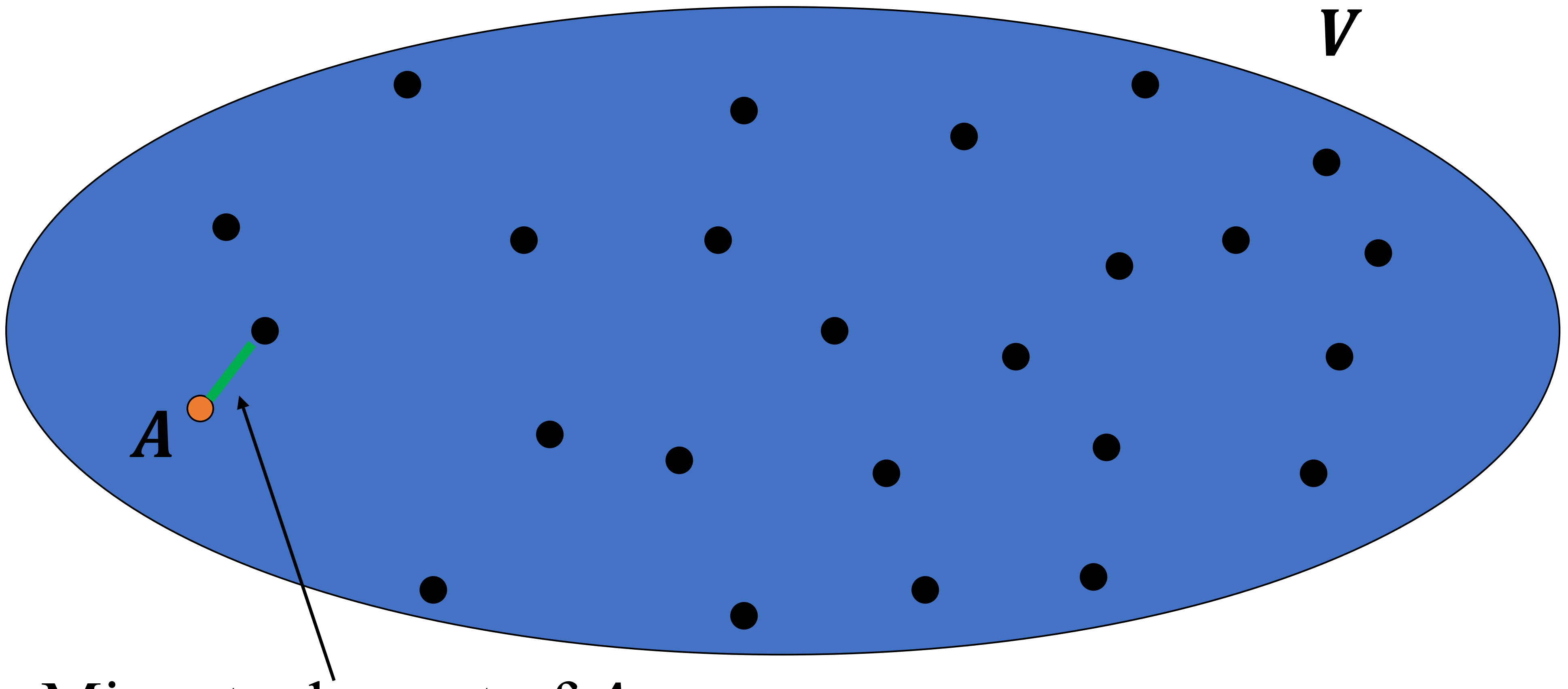
Swap (u, v) with the first edge on this path that connects a vertex in A to a vertex in $V - A$.

A lighter weight spanning tree than T results.

Greedy Algorithm

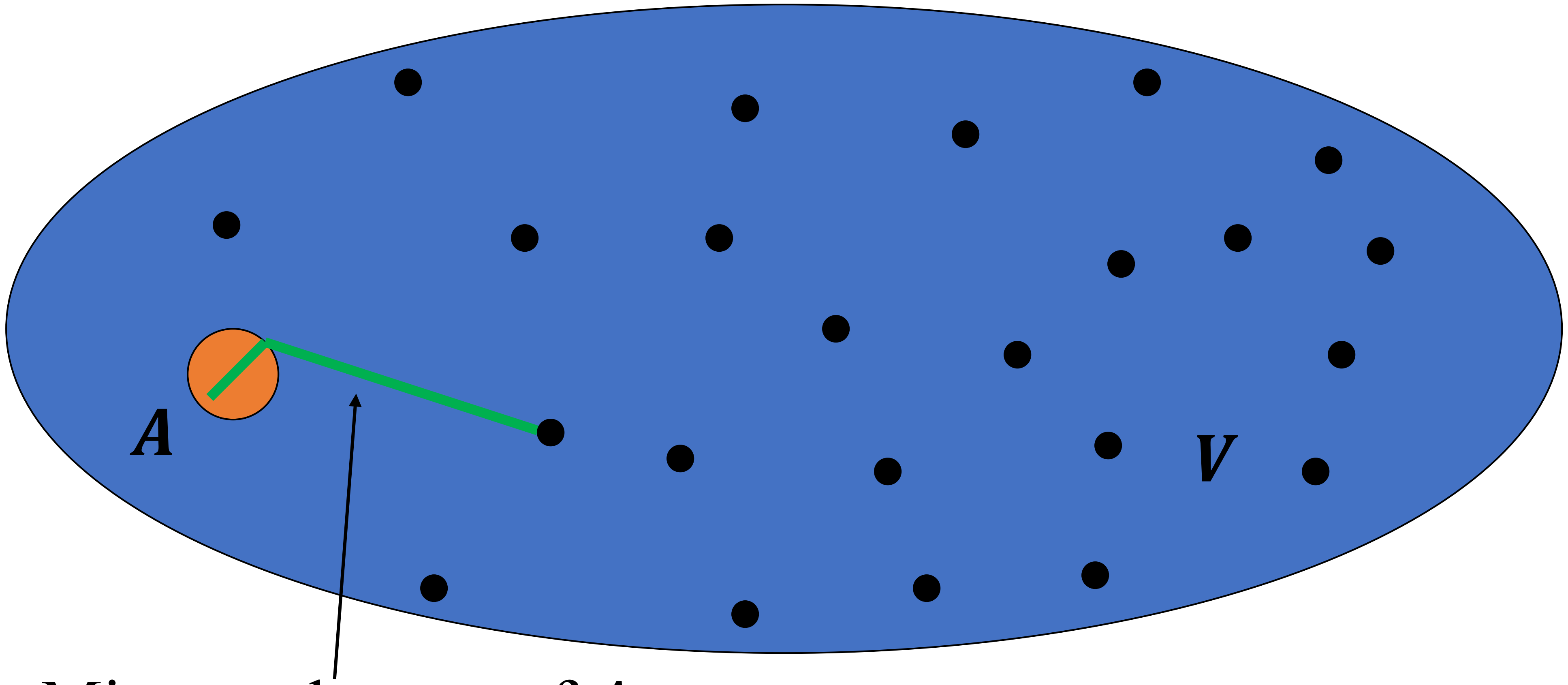


Greedy Algorithm



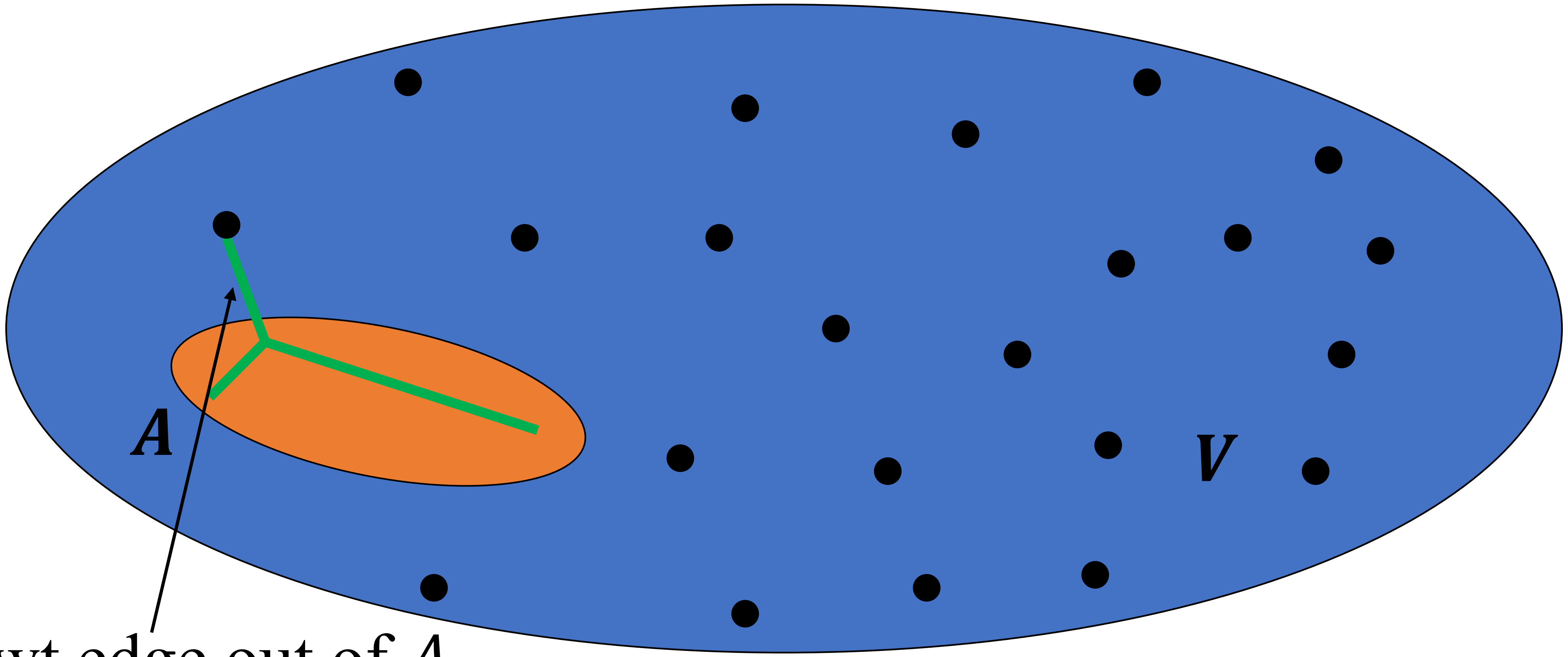
Min wt edge out of A

Greedy Algorithm



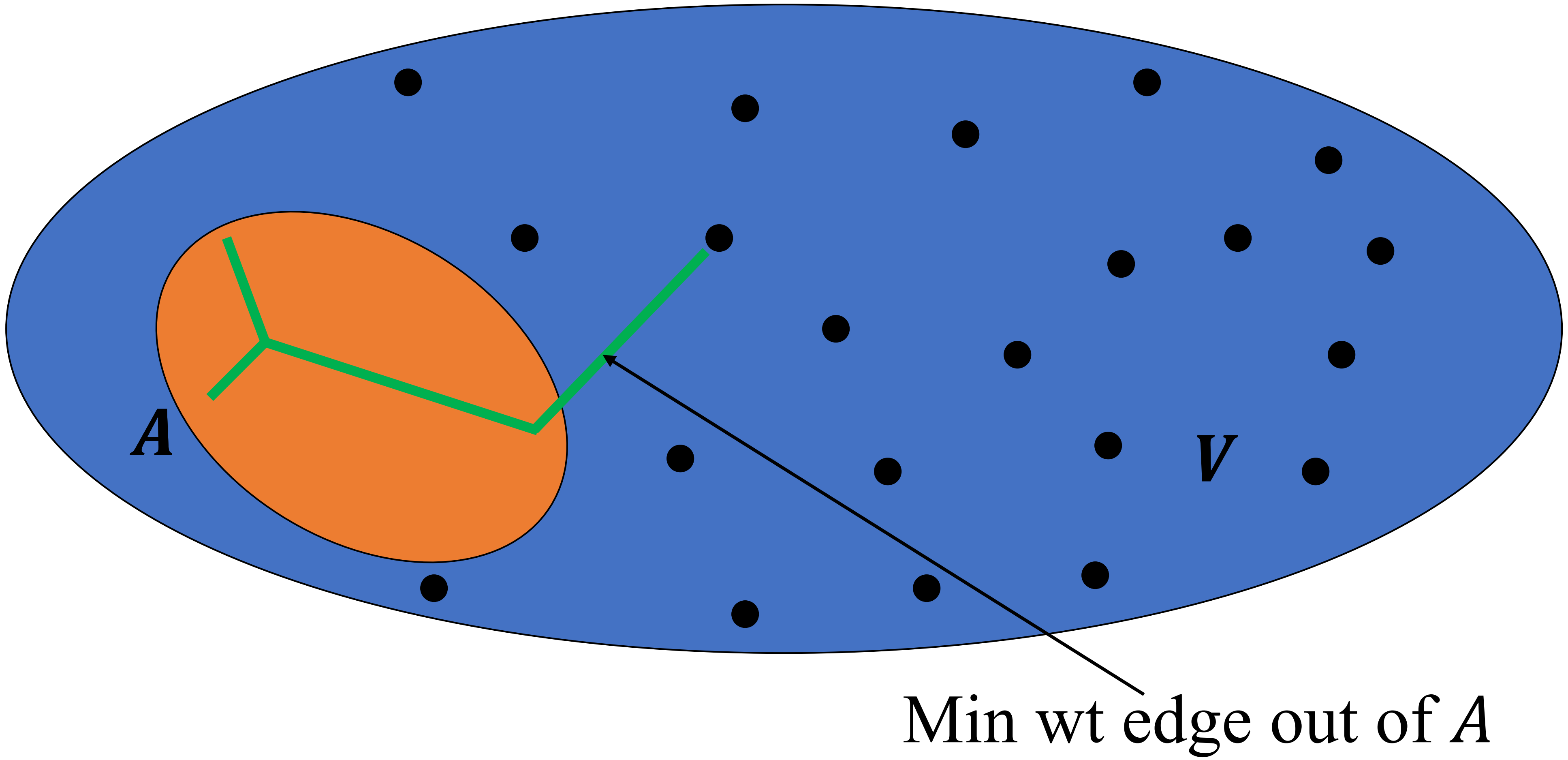
Min wt edge out of A

Greedy Algorithm

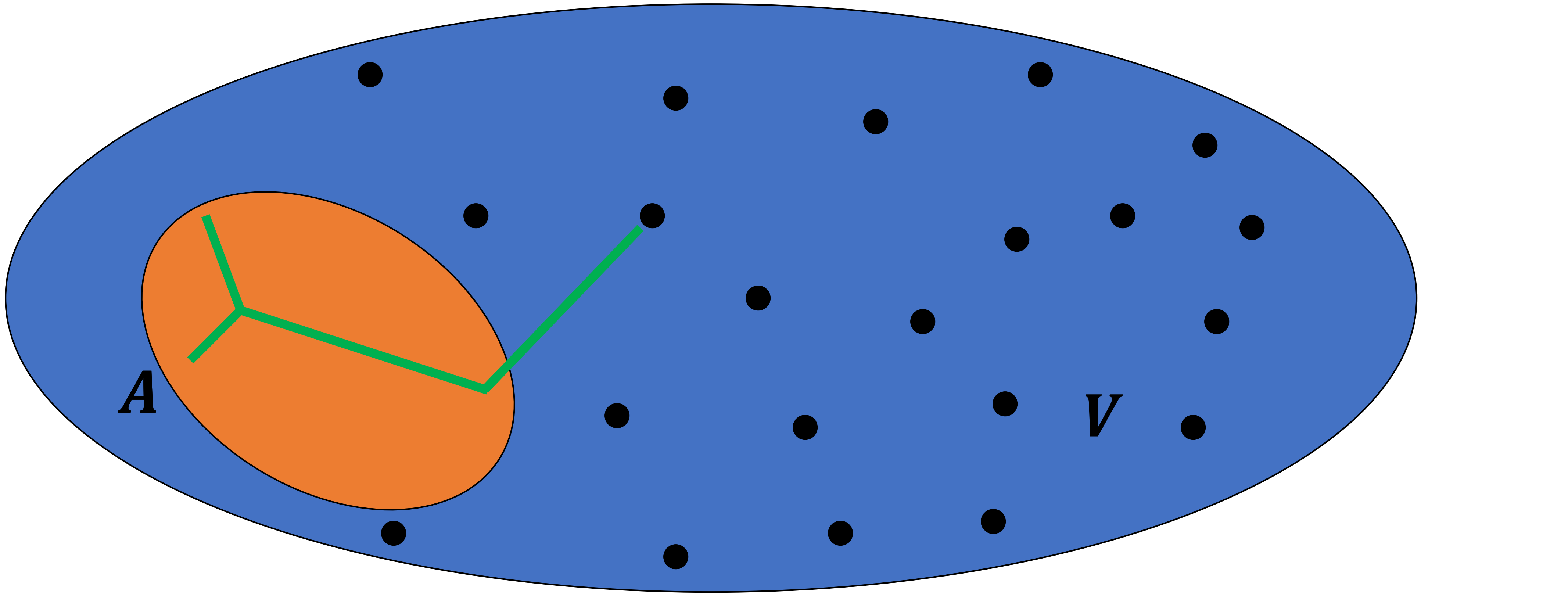


Min wt edge out of A

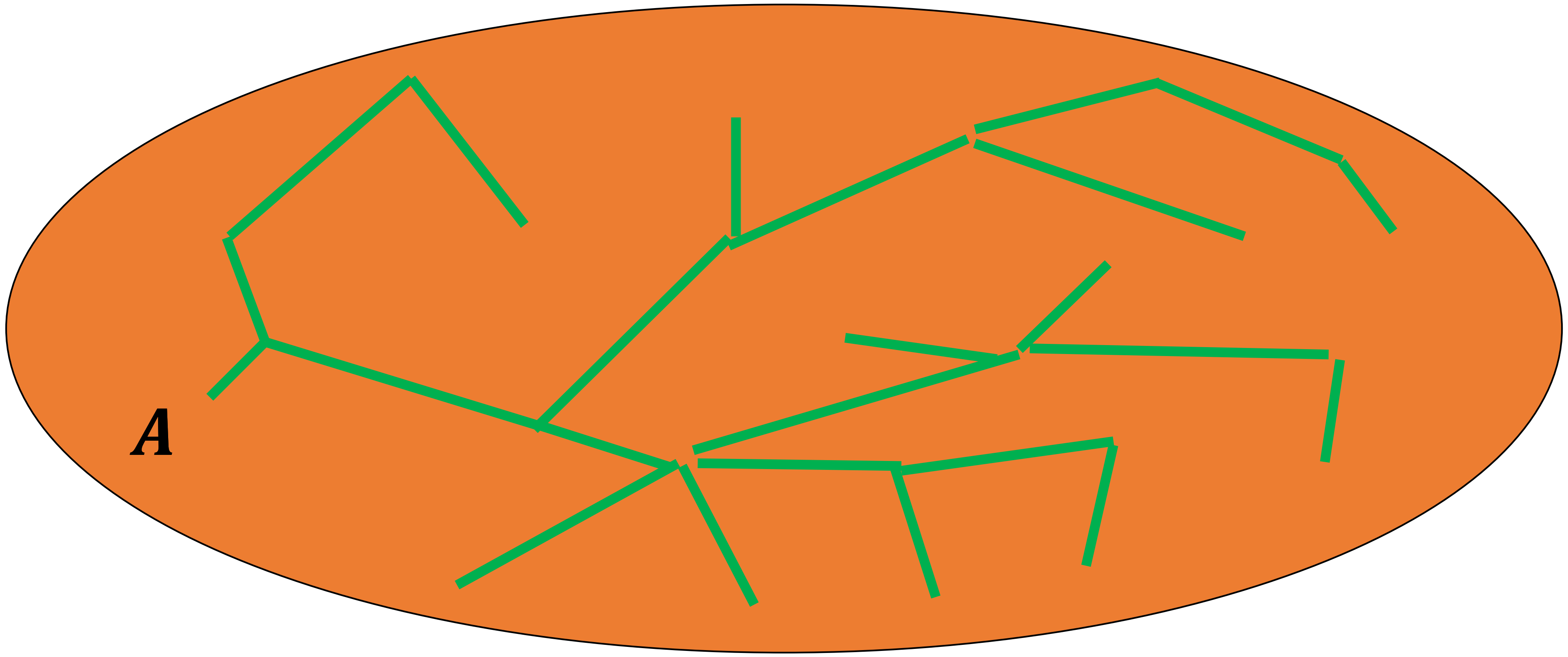
Greedy Algorithm



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.

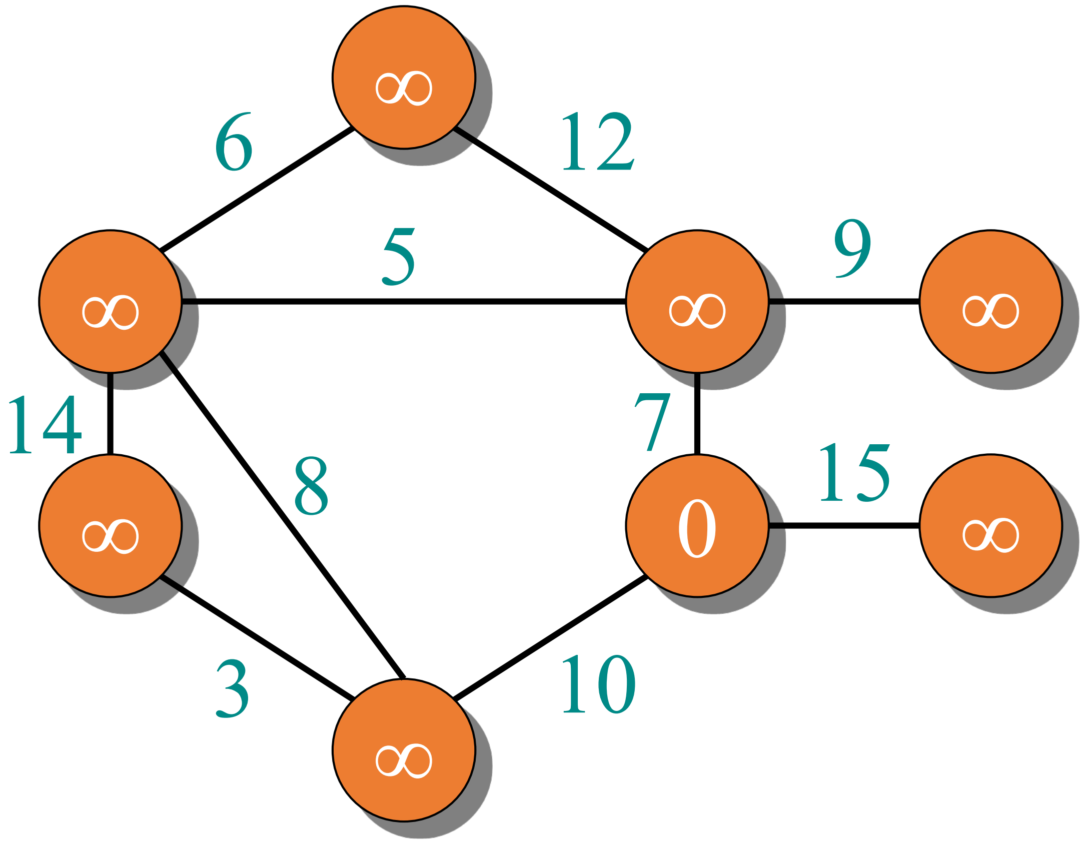
```
$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN($Q$)
        for each $v \in Adj[u]$
            do if $v \in Q$ and $w(u, v) < key[v]$
                then $key[v] \leftarrow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

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

Example of Prim's algorithm

○ $\in A$

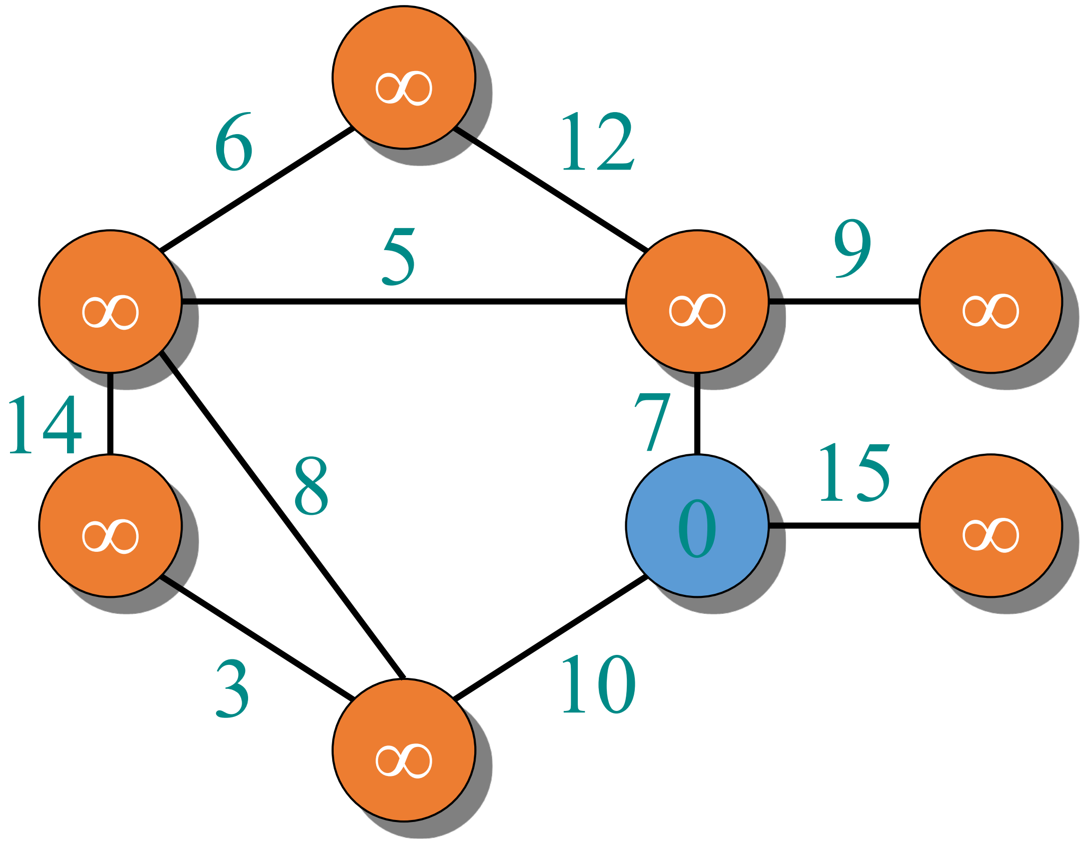
● $\in V - A$



Example of Prim's algorithm

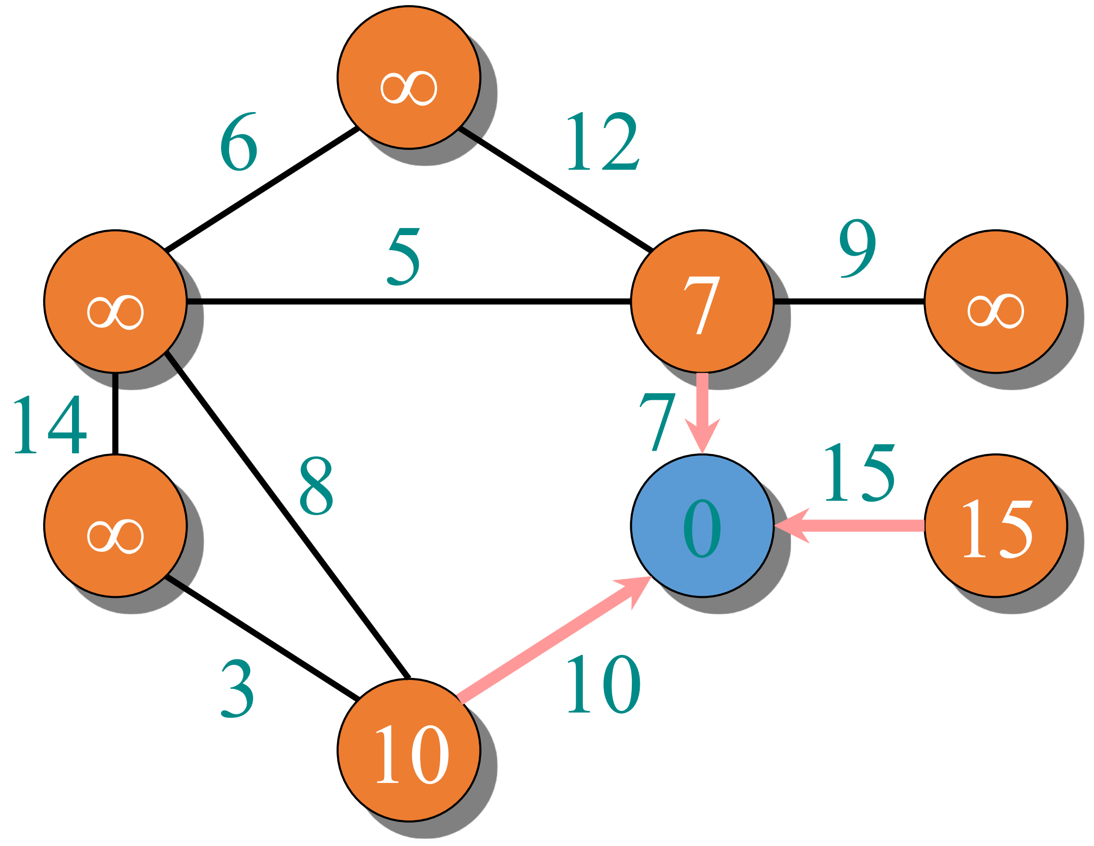
○ $\in A$

● $\in V - A$



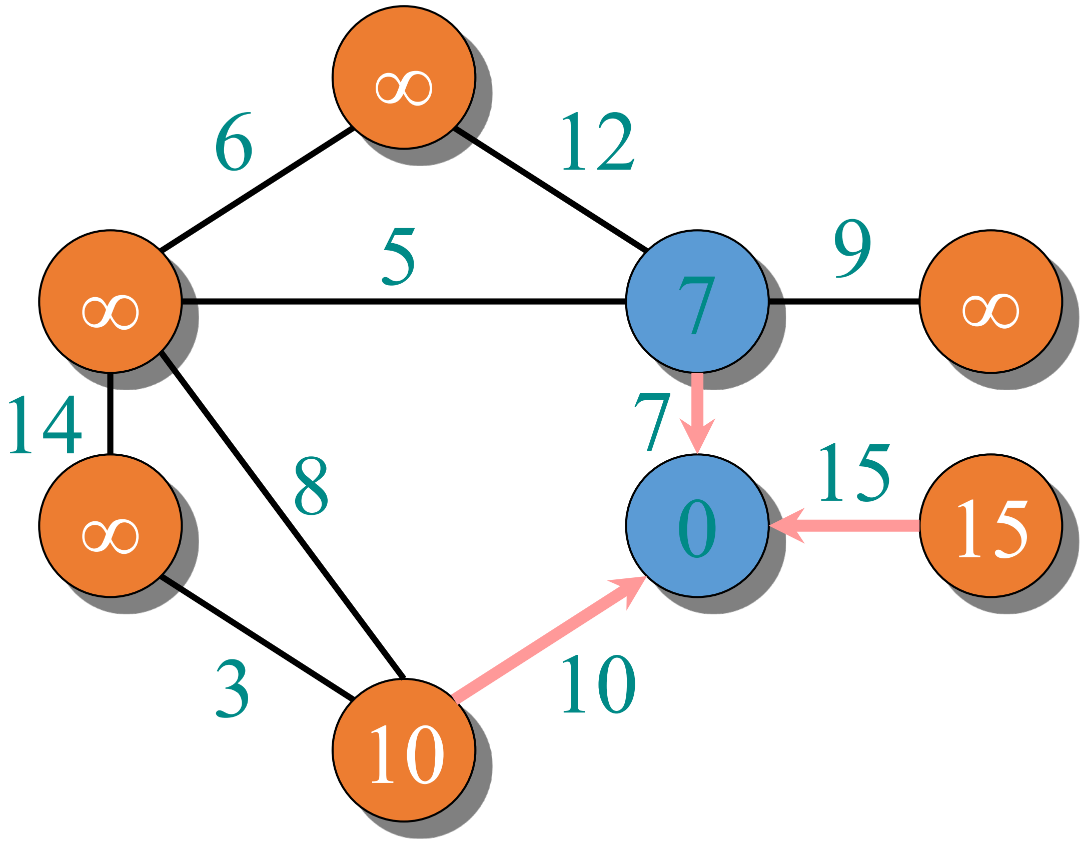
Example of Prim's algorithm

● $\in A$
● $\in V - A$



Example of Prim's algorithm

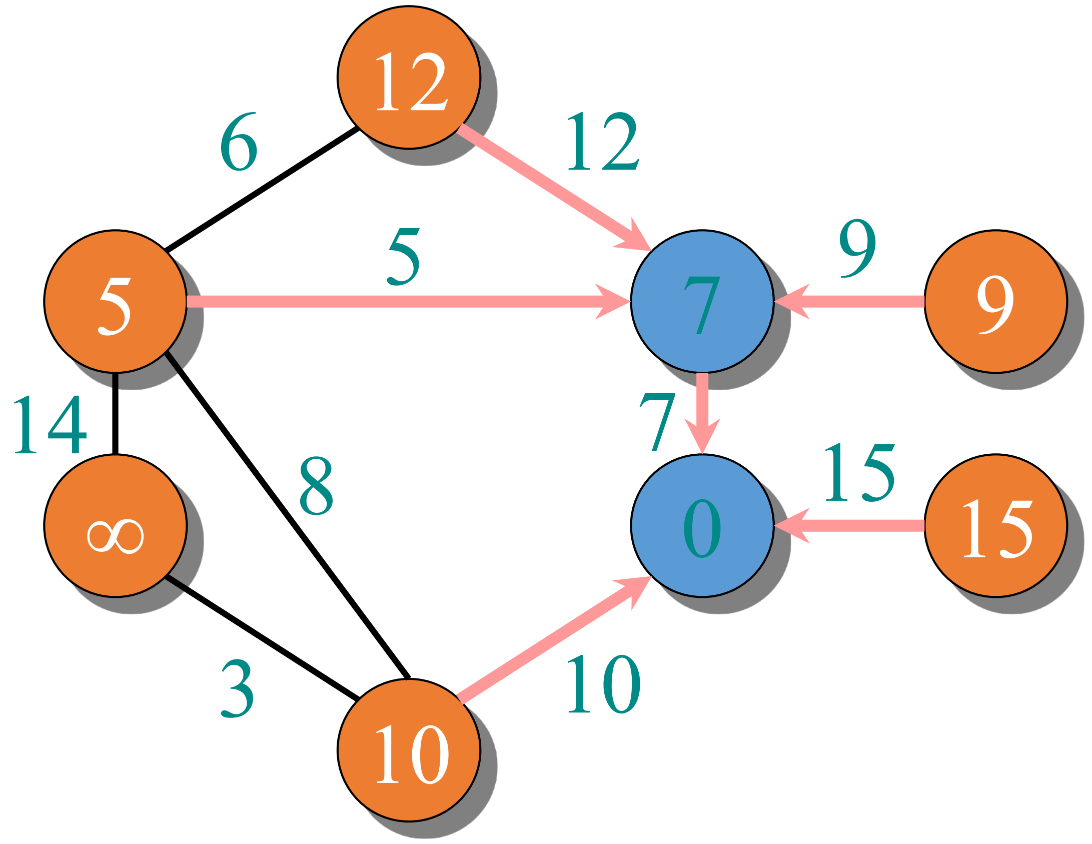
- ○ $\in A$
- ● $\in V - A$



Example of Prim's algorithm

○ $\in A$

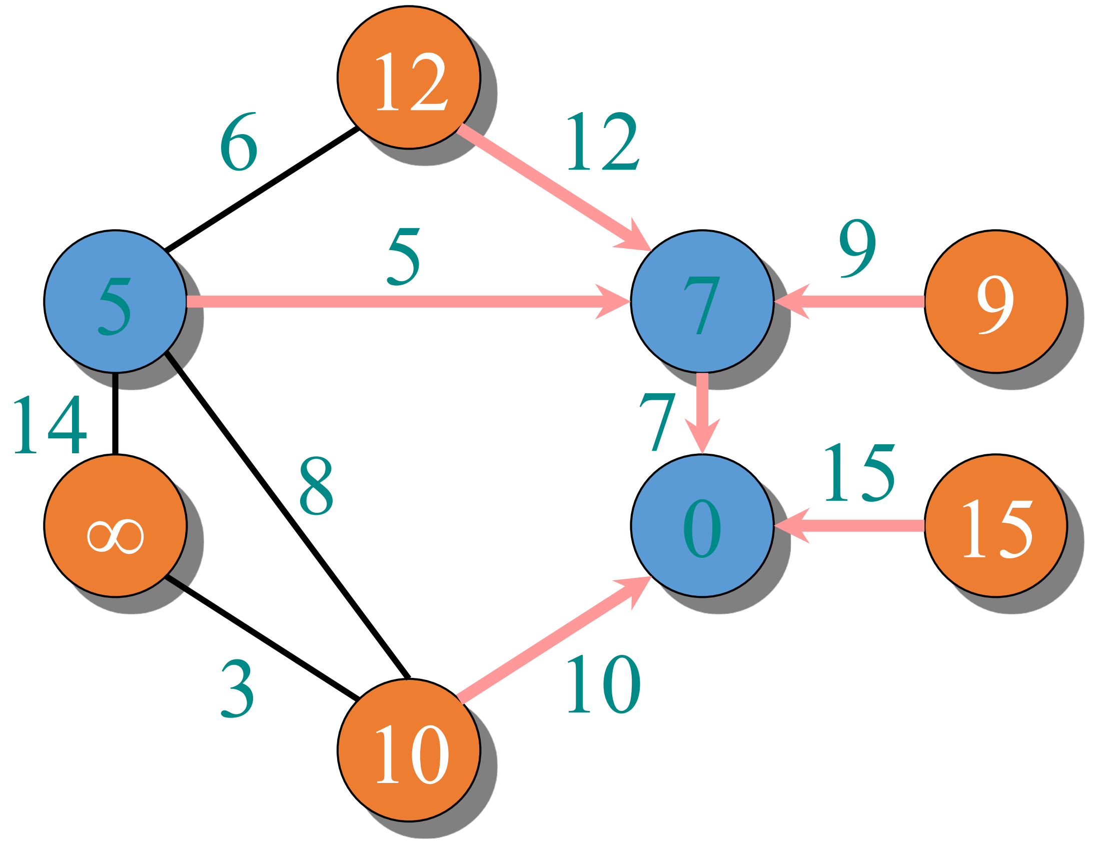
● $\in V - A$



Example of Prim's algorithm

○ $\in A$

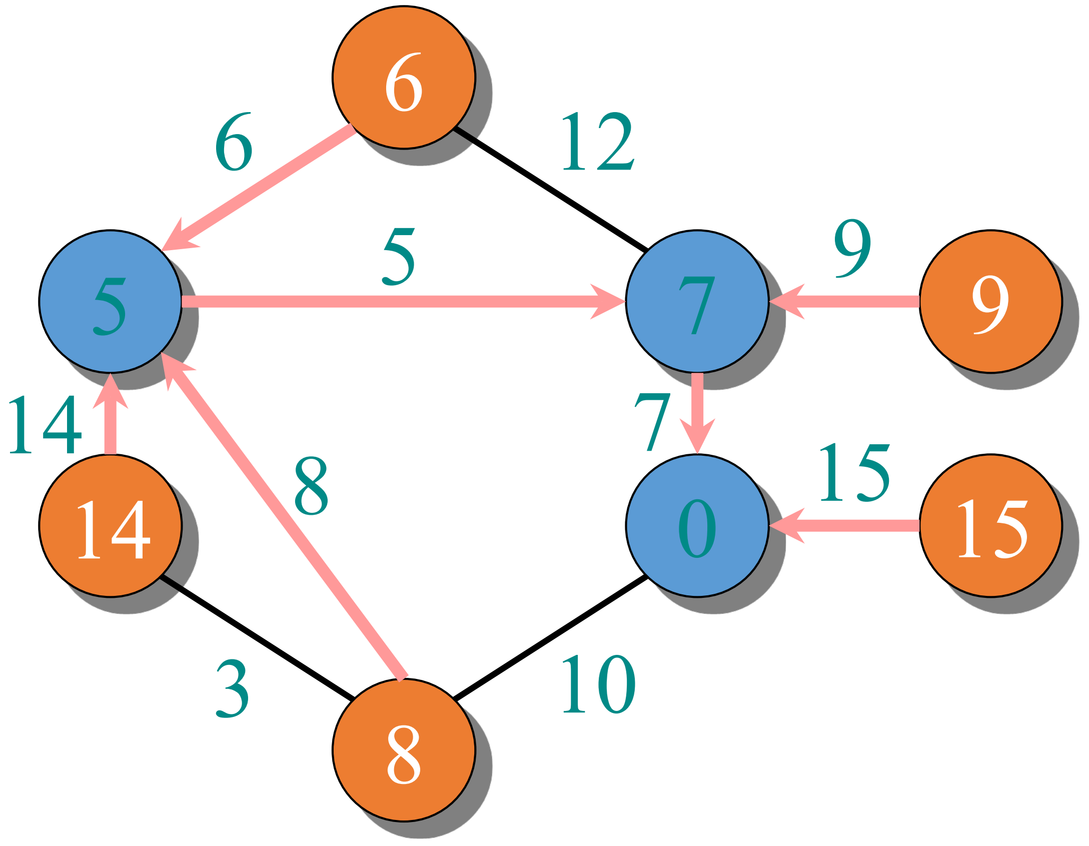
● $\in V - A$



Example of Prim's algorithm

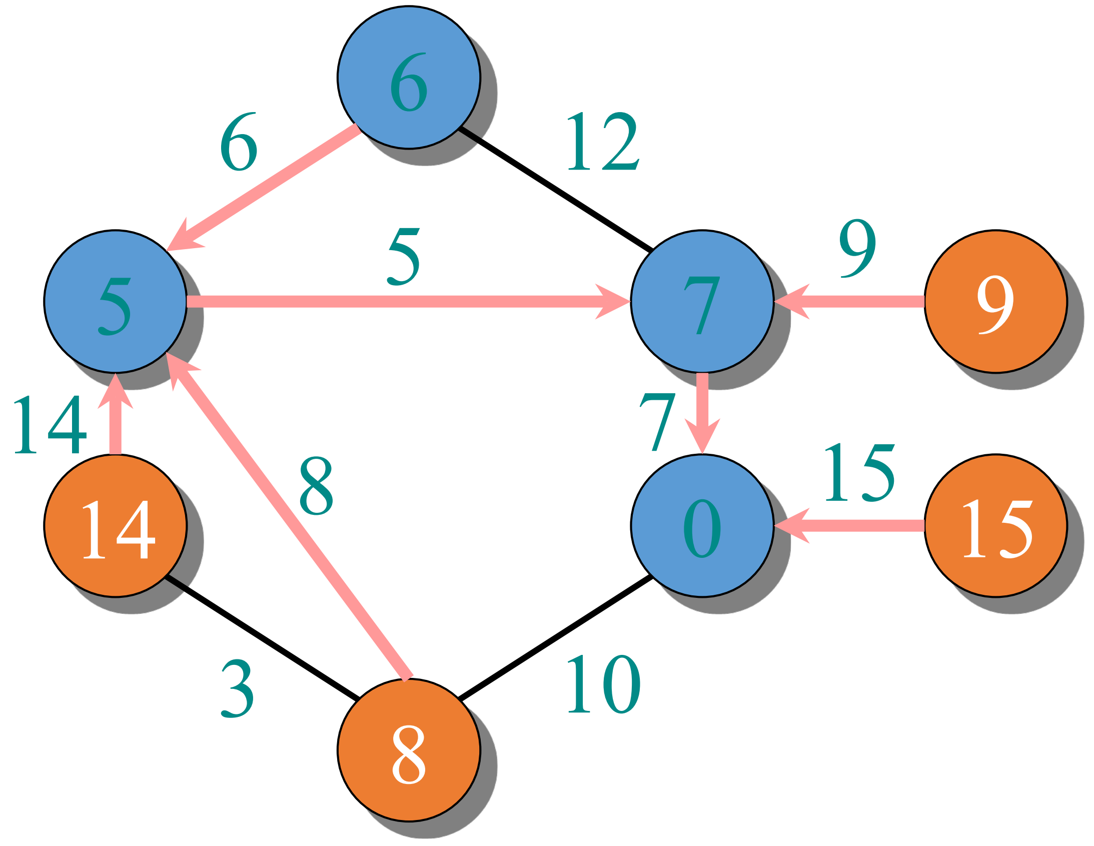
○ $\in A$

● $\in V - A$



Example of Prim's algorithm

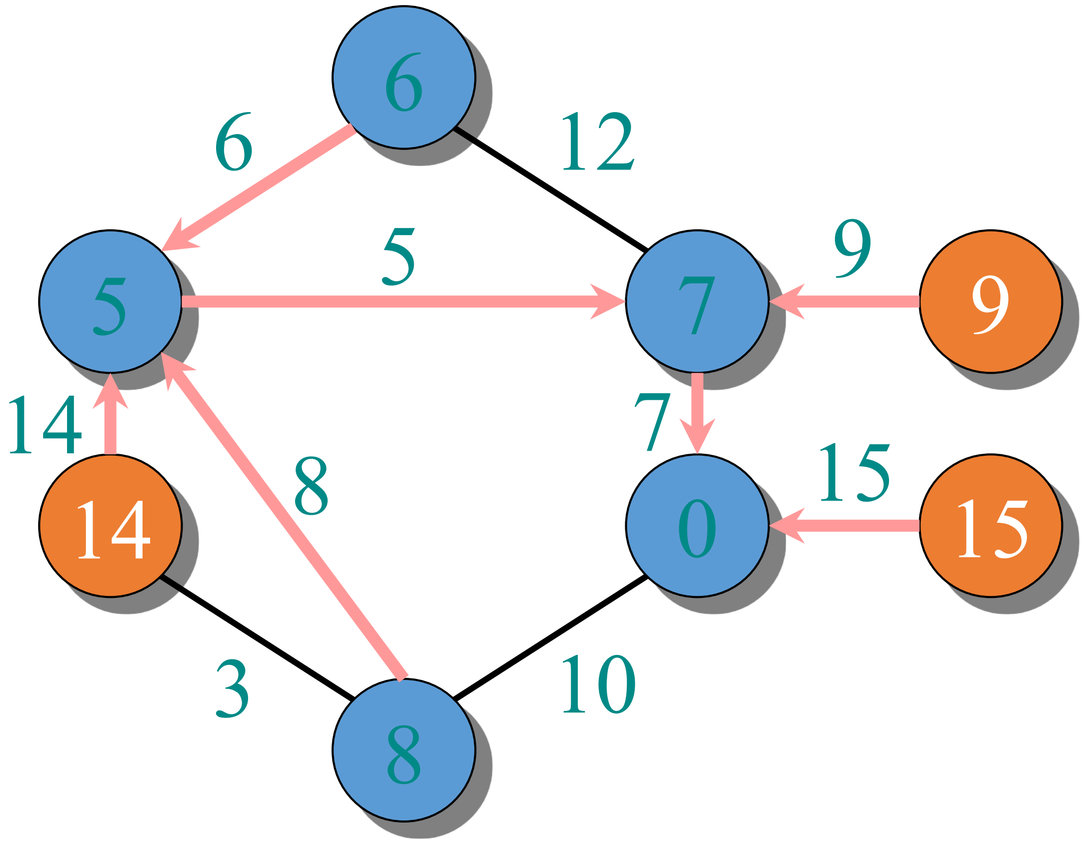
- ○ $\in A$
- ● $\in V - A$



Example of Prim's algorithm

○ $\in A$

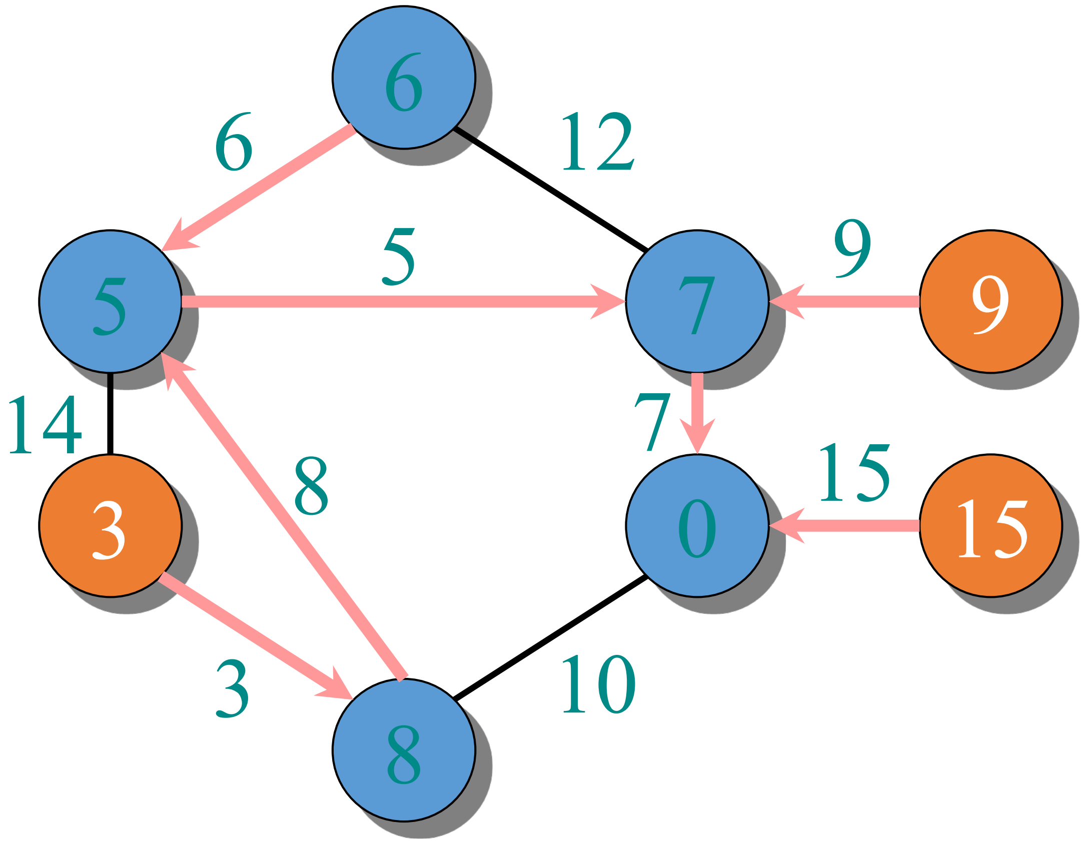
● $\in V - A$



Example of Prim's algorithm

○ $\in A$

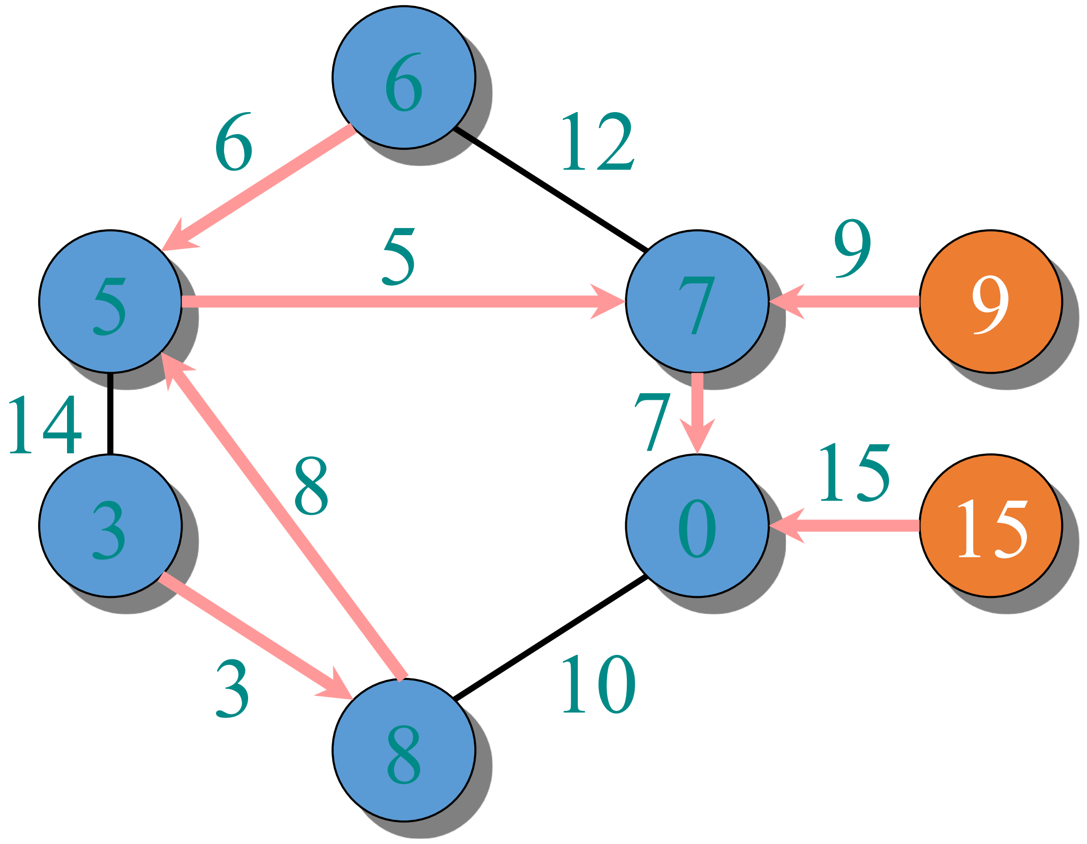
● $\in V - A$



Example of Prim's algorithm

○ $\in A$

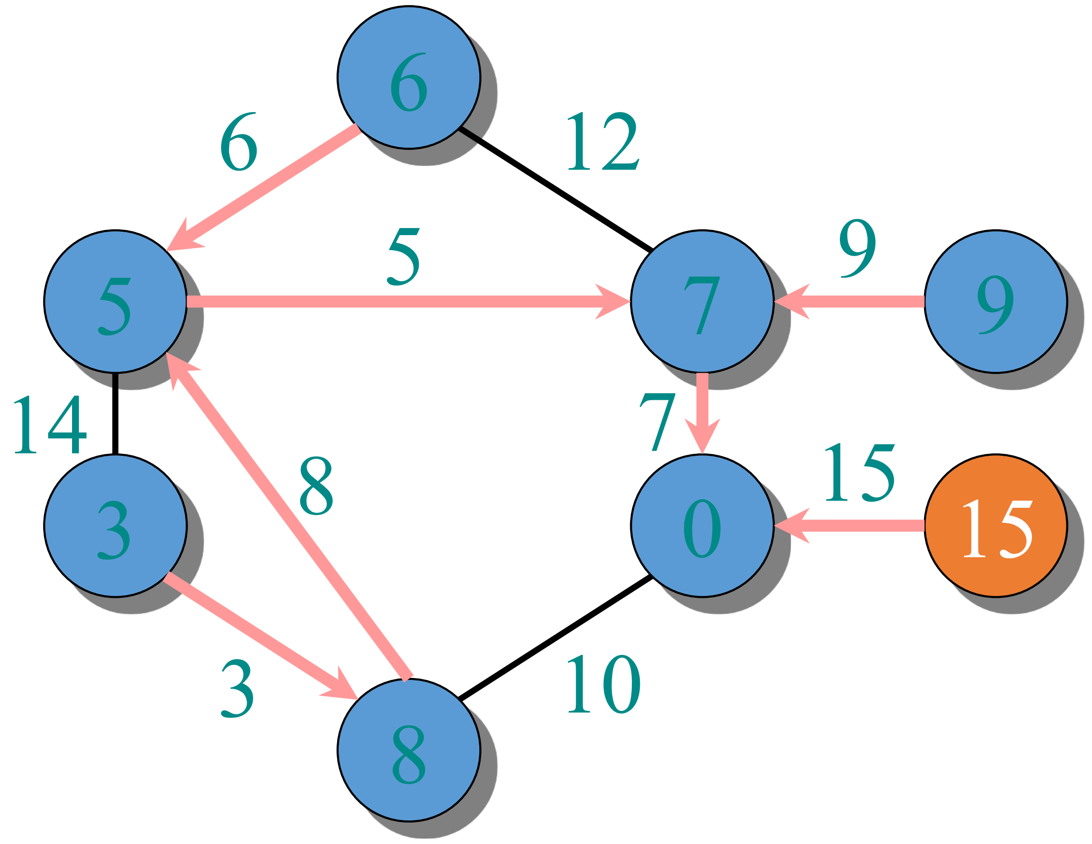
● $\in V - A$



Example of Prim's algorithm

○ $\in A$

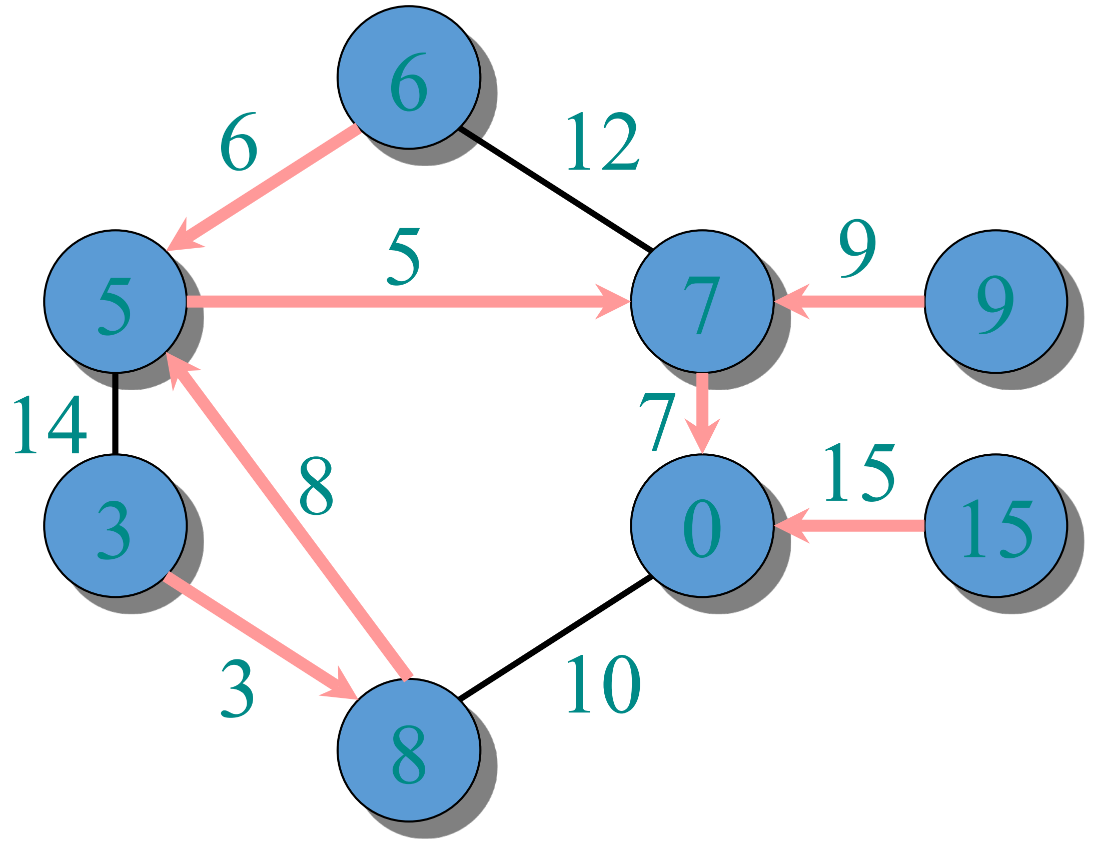
● $\in V - A$



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

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

```
$\Theta(n)$ 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

```
$\Theta(n)$ 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

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

$\Theta(m)$ implicit DECREASE-KEY's.

Analysis of Prim

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

$\Theta(m)$ implicit DECREASE-KEY's.

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

Analysis of Prim (continued)

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

Analysis of Prim (continued)

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

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

Analysis of Prim (continued)

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

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

Analysis of Prim (continued)

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

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

Analysis of Prim (continued)

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

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\lg n)$	$O(\lg n)$	$O(m \lg n)$
Fibonacci heap	$O(\lg n)$ amortized	$O(1)$ amortized	$O(m + n \lg n)$ 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 $O(m \log n)$, 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 $O(m \log n)$, though Boruvka's algorithm is usually the method of choice in practice.

Best to date:

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

Question 1

Your friend proposes a new divide-and-conquer algorithm for MST. Given a graph $G = (V, E)$, divide the vertex set into two (nearly) equal halves V_1 and V_2. Recursively find an MST T_1 for the graph induced on V_1 and an MST T_2 for the graph induced on V_2. Let e be the minimum weight edge with one endpoint in V_1 and the other in V_2. Return $T_1 \cup T_2 \cup \{e\}$.

Why does this new algorithm not work?

Question 1: Solution

By the greedy-choice property, e will be in the MST. Does optimal substructure mean that the MST restricted to V_1 is T_1 and the MST restricted to V_2 is T_2? No!

In optimal substructure, we assumed that there was one edge in MST between V_1 and V_2. But here, there can be many!

Plan for Today

We review

- ❑ **Dynamic Programming**
- ❑ **Greedy Algorithms**
- ❑ **Approximation Algorithms**
- ❑ Randomized Algorithms
- ❑ Intractability and NP-Completeness
- ❑ Amortized Analysis

Approximate Solutions

- For optimization problems sometimes it is "OK" to find a solution that is *nearly optimal* in its cost.

- Recall that an optimization problem looks like:

$$\max/\min C(x)$$

such that x satisfies some constraints

Approximation Ratio

Let C^* be the optimal cost and C be the cost of the solution given by an approximation algorithm A.

An approximation algorithm A has an **approximation ratio** of $\rho(n)$ if:

$$\frac{C}{C^*} \leq \rho(n) \qquad \text{(for minimization)} \qquad \boxed{\geq 1}$$

$$\frac{C}{C^*} \geq \rho(n) \qquad \text{(for maximization)} \qquad \boxed{\leq 1}$$

Two general approaches

- **Analyze a heuristic**: A “heuristic” is a procedure that does not always produce the optimal answer. Sometimes, we can show that it’s not too bad though.

- **Solve an LP relaxation**: Many problems can be reduced to Integer Programming. Solve the Linear Programming version instead.

Metric TSP

Recall the TSP problem: Given an undirected complete graph with non-negative edge cost $w(.)$, find a TSP tour with minimum cost.

Question 6 of Practice Problem Set 4 implies that this problem cannot be approximated within ratio $\alpha(n)$ for any positive function $\alpha(.)$ unless **P=NP**.

Now study the above problem under the assumption that the edge cost follows triangle inequality, i.e., for any three vertices u, v, x, $w\big((u, v)\big) \leq w\big((u, x)\big) + w((x, v))$. Under this assumption the TSP problem is known as **Metric TSP**.

Question 2

Consider the following algorithm for Metric TSP.

1. Find an MST of G
2. Double every edge of that MST to obtain an Eulerian graph
3. Find an Eulerian tour (a tour that visits every edge exactly once) T on this graph
4. Output the tour that visits vertices of G in the order of their first appearance in T

Show that the approximation ratio of the above algorithm is 2.

Question 2: Solution

- Cost of the MST is at most OPT (minimum cost of a TSP tour)
- Cost of T is at most **twice** the cost of the MST (by construction in step 2)
- Due to triangle inequality cost of output tour is at most cost of T
- So the approximation ratio is 2

Randomized Approximation

Let C^* be the optimal cost and C be the **expected** cost of the solution given by a **randomized** approximation algorithm A.

A randomized approximation algorithm A has an **approximation ratio** of $\rho(n)$ if:

$$\frac{C}{C^*} \leq \rho(n) \qquad \text{(for minimization)}$$

$$\frac{C}{C^*} \geq \rho(n) \qquad \text{(for maximization)}$$

Plan for Today

We review

- ❑ **Dynamic Programming**
- ❑ **Greedy Algorithms**
- ❑ **Approximation Algorithms**
- ❑ **Randomized Algorithms**
- ❑ Intractability and NP-Completeness
- ❑ Amortized Analysis

Things to remember

- Union bound in probability: $\Pr[A \cup B] \leq \Pr[A] + \Pr[B]$

- Linearity of expectations: $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$ even if X and Y are not independent random variables.

- Indicator variables are very, very helpful! Use them wherever possible.

Question 3

Let $G = (V, E)$ be an undirected graph with $\boldsymbol{n}$ vertices and $\boldsymbol{m}$ edges.
Show that the graph G has an independent set of size at least $\boldsymbol{\frac{n^2}{4m}}$.

To show the above use the following randomized algorithm and compute the **expected size of the set output** by the algorithm.

1. Delete each vertex (together with all its incident edges) independently with probability $1 - \frac{1}{d}$, where $d = \frac{2m}{n}$ is the average degree of the graph.
2. For each remaining edge, remove it and one of its adjacent vertex.

Question 3: Solution

- Let X be the set of vertices survived after step 1.

$$E[|X|] = \frac{n}{d}$$

- Let Y be the set of edges survived after step 1.

$$E[|Y|] = \frac{nd}{2}\left(\frac{1}{d}\right)^2 = \frac{n}{2d}$$

- The second step removes all the remaining edges and at most $|Y|$ many vertices.

$$E[size\ of\ output\ set] = \frac{n}{d} - \frac{n}{2d} = \frac{n}{2d} = \frac{n^2}{4m}$$

What you have just seen is called **Probabilistic Method**

Plan for Today

We review

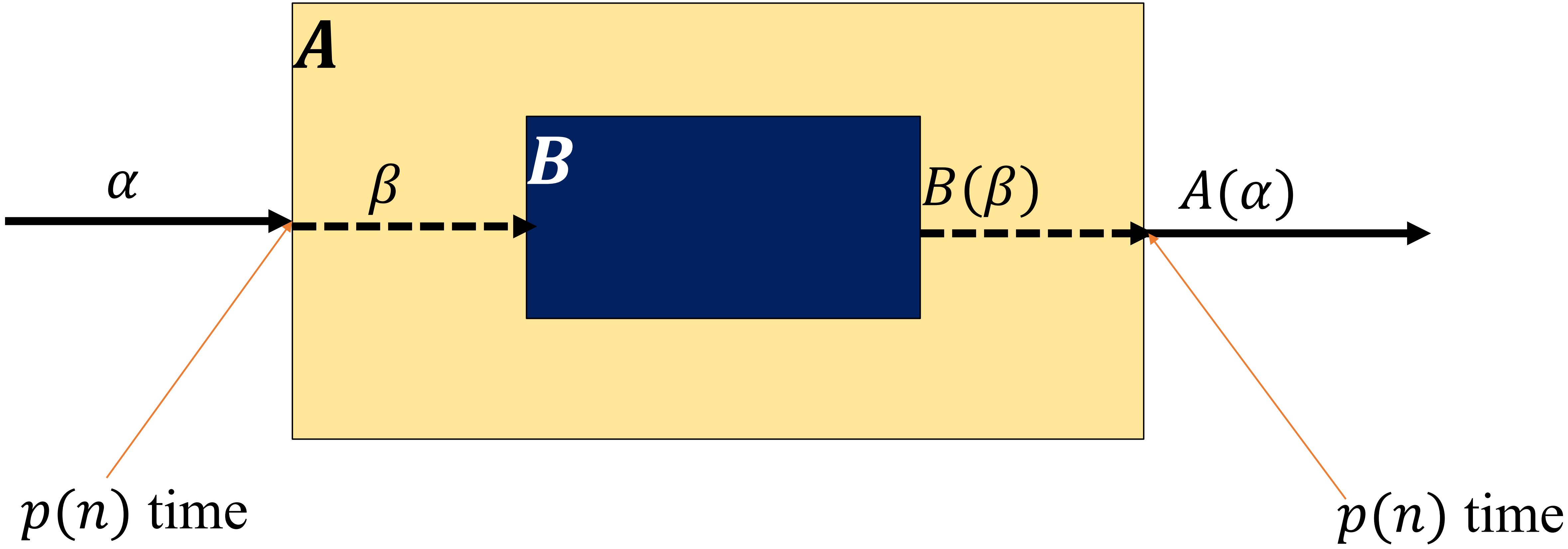
- ❑ **Dynamic Programming**
- ❑ **Greedy Algorithms**
- ❑ **Approximation Algorithms**
- ❑ **Randomized Algorithms**
- ❑ **Intractability and NP-Completeness**
- ❑ Amortized Analysis

Problem Reduction

- Reductions are a basic tool in algorithm design: using an algorithm for one problem to solve another.

- If you have a poly time reduction from A to B and you also have a poly time algorithm for B, then you get a poly time algorithm for A.

$\boldsymbol{p(n)}$-time Reduction



Poly-time Reduction

$$A \leq_P B$$

$$\Big\downarrow$$

If B has a polynomial time algorithm, then so does A!

Poly-time Reduction

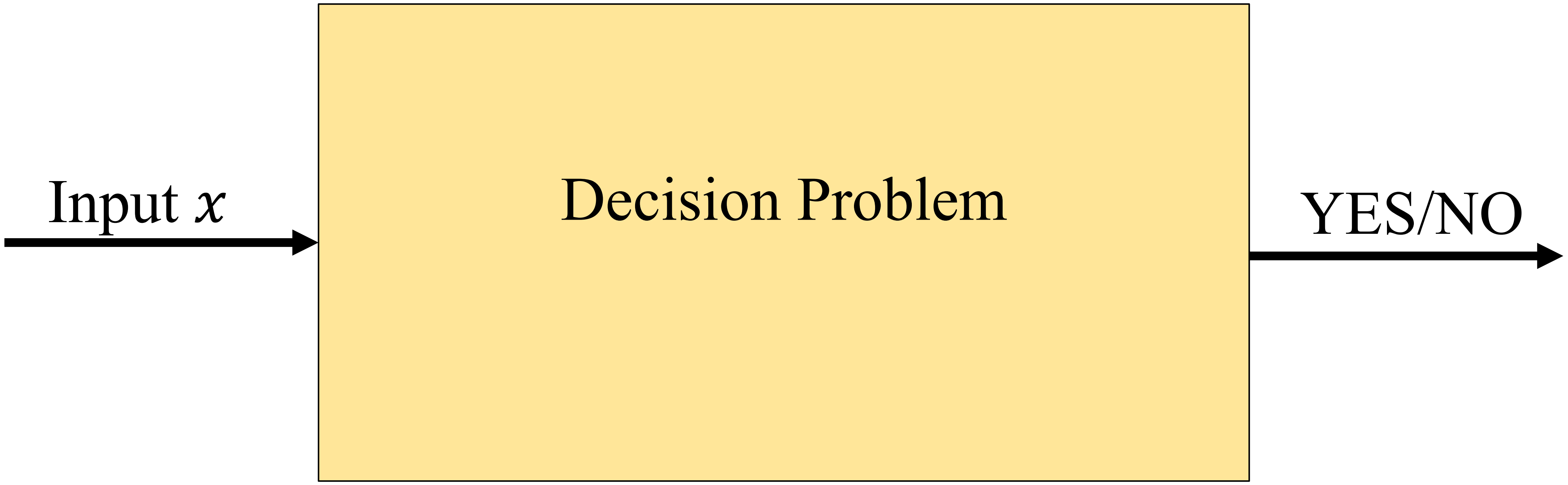
$$A \leq_P B$$

$$\Downarrow$$

If A is “hard”, then so is B!

Decision Problems

A **decision problem** is a function that maps an instance space I to the solution set $\{\text{YES, NO}\}$.



Reductions between Decision Problems (Karp Reduction)

Given two decision problems A and B, a **polynomial time reduction** from A to B, denoted $A \leq_P B$, is a transformation from instances α of A to instances β of B such that:

1. α is a YES-instance for A **<u>if and only if</u>** β is a YES-instance for B.
2. The transformation takes polynomial time in the size of α.

NP class

Yes instance

No instance

X : any decision problem

I : any (input) instance of X

Efficient certifier for X :

A polynomial time algorithm A with output {yes,no}

- **Input** : (I, s)

Proposed solution

- **Behavior**: There is a polynomial function p such that I is yes-instance of X **if and only if** there exists a string s with $|s| \leq p(|I|)$ such that A outputs yes on input (I, s).

NP class

Definition (**NP**):

The set of all decision problems which have **efficient certifier**.

NP : “Non-deterministic polynomial time”

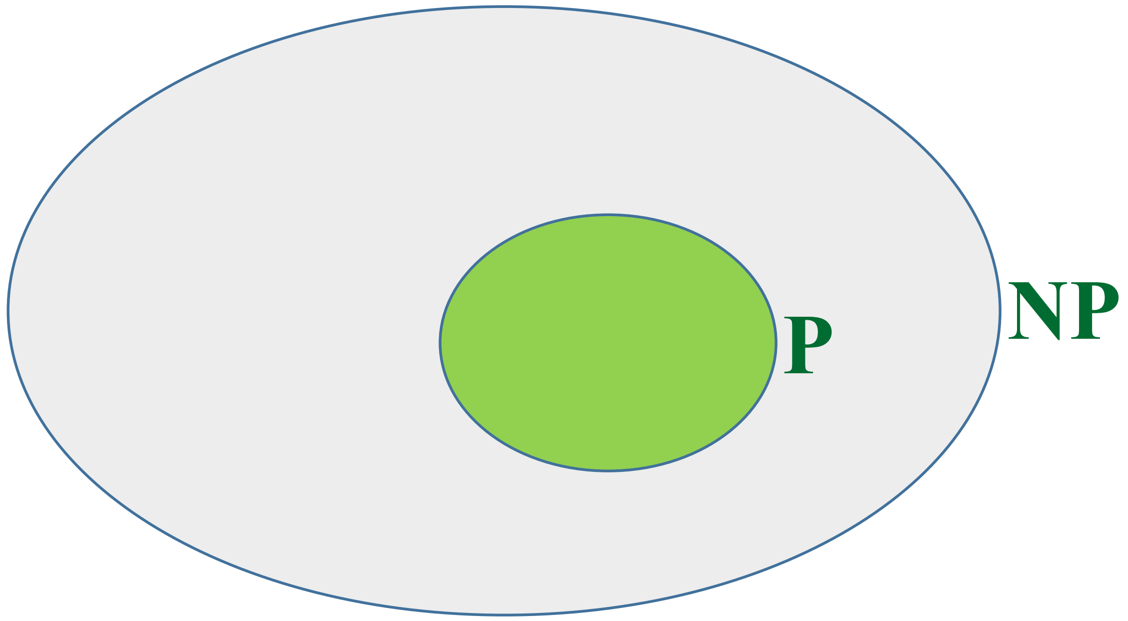
Definition (**P**):

The set of all decision problems which have **efficient** (poly-time) algorithm.

Is there any Relation between **P** and **NP** ?

NP versus P

Is P = NP ?



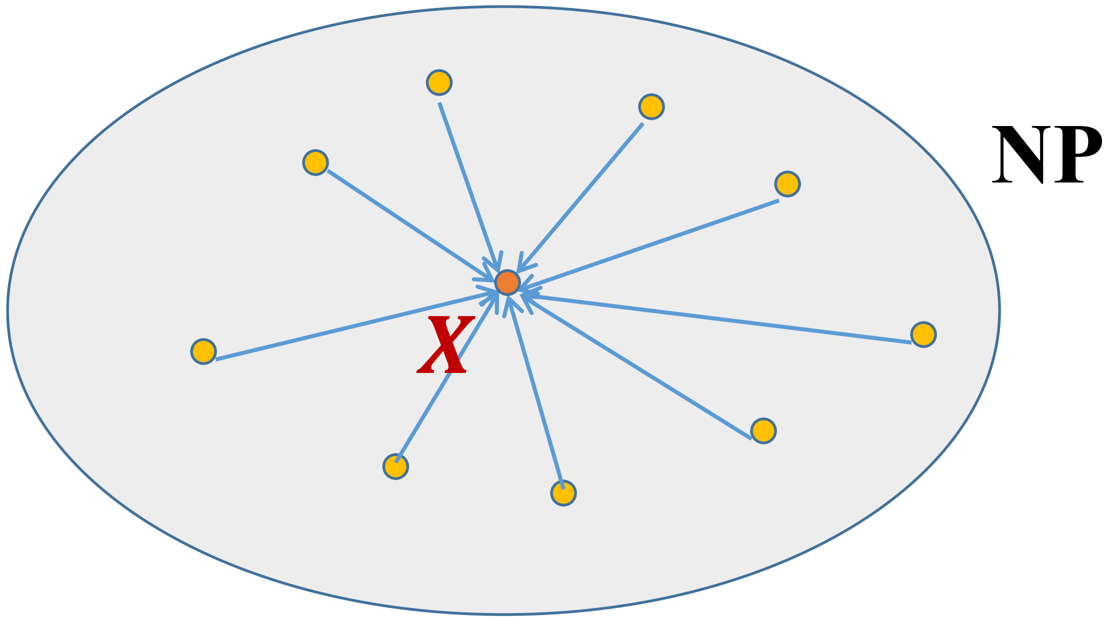
Verifying a <u>proposed solution</u> versus **finding a <u>solution</u>**

NP-complete

If X is not known to be in **NP,** then we say X is just **NP-hard**

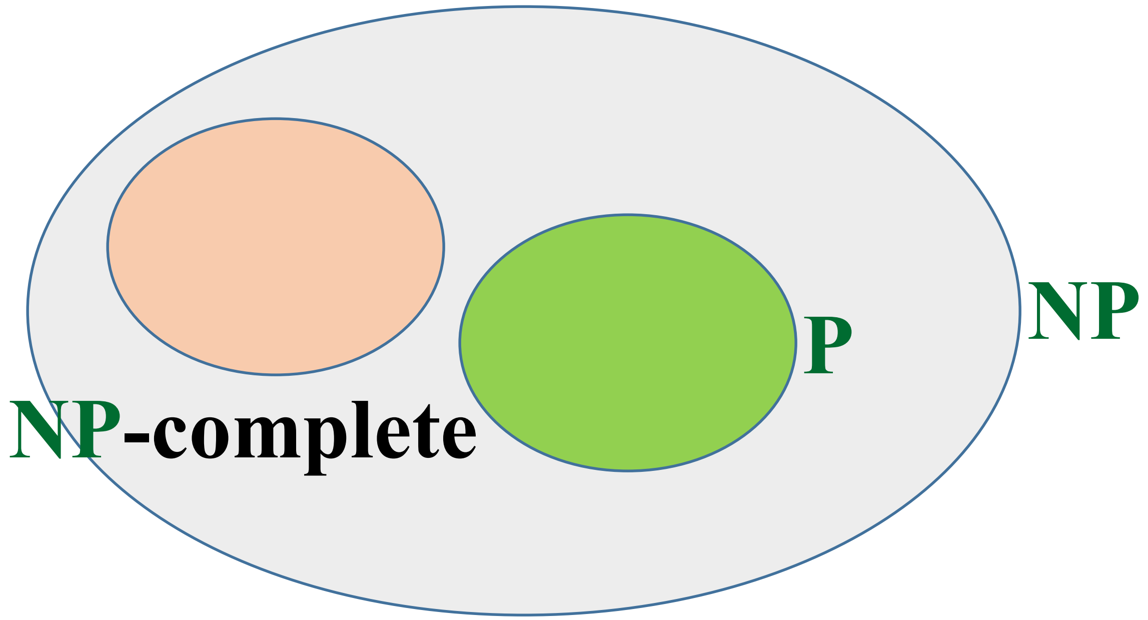
- A problem X in **NP** class is **NP-complete** if for every $A \in$ **NP**

$$A \leq_P X$$



NP versus P

Is P = NP ?



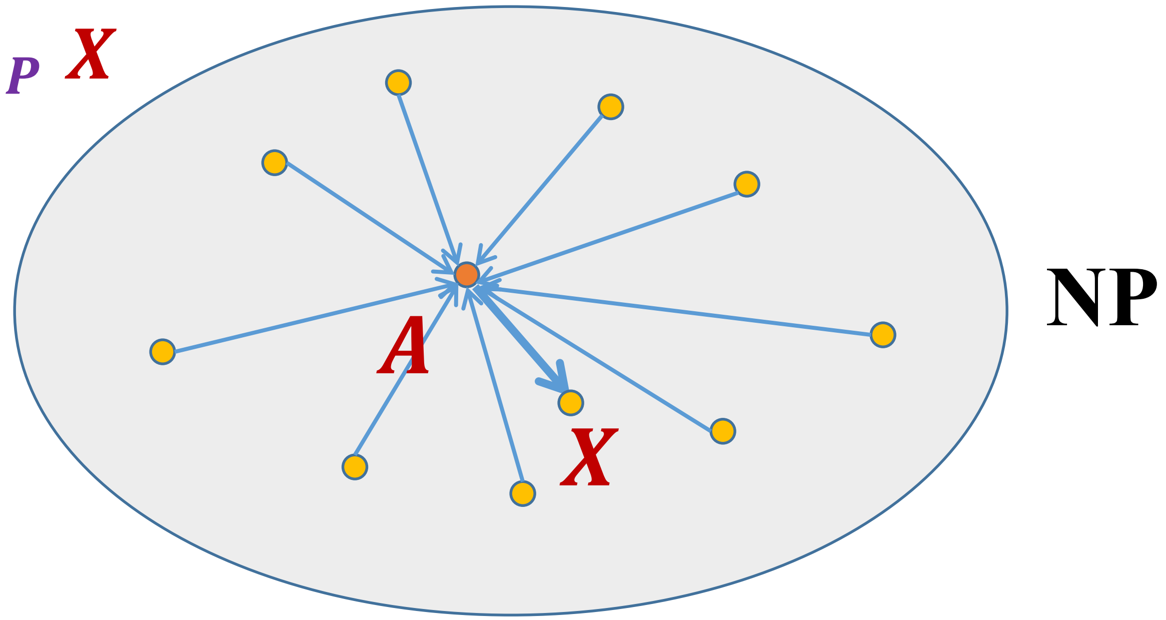
If any **NP**-complete problem is solved in polynomial time

➔ P = NP

How to show a problem to be NP-complete ?

Let X be a problem which we wish to show to be **NP**-complete

1. Show that X ϵ **NP**
2. Pick a problem A which is already known to be **NP**-complete
3. Show that $A \leq_P X$



Question 4

In the MIN-FEEDBACK-VERTEX-SET (MFVS) problem: given a directed graph G and a non-negative integer k, is it possible to remove at most k vertices (along with their incident edges) such that the resulting graph has no directed cycles?

Show that the following is a valid poly time reduction from VERTEX-COVER to MFVS. Given an undirected graph G and number k (instance of Vertex-Cover), output a directed graph G' and number k' where:

- $V(G') = V(G)$ and directed edges $(u, v), (v, u) \in E(G')$ for any undirected edge $\{u, v\}$ in $E(G)$.
- $k' = k$

Question 4: Solution

- Clearly, poly time

- If there is a vertex cover of size $\leq k$, then it contains either u or v of any edge $\{u, v\} \in E(G)$. Any cycle in G' corresponds to some set of edges in G, so it's also covered by the vertex cover.

- If there a MFVS of size $\leq k$, then it must cover each of the cycles of length 2 formed by the edges of G, hence it must be a vertex cover of G as well.

Plan for Today

We review

- ❑ **Dynamic Programming**
- ❑ **Greedy Algorithms**
- ❑ **Approximation Algorithms**
- ❑ **Randomized Algorithms**
- ❑ **Intractability and NP-Completeness**
- ❑ **Amortized Analysis**

Types of amortized analyses

- Three common amortization arguments:
  - *Aggregate* method
  - *Accounting* method
  - *Potential* method

- The aggregate method, though simple, lacks the precision of the other two methods. In particular, the accounting and potential methods allow a specific ***amortized cost*** to be allocated to each operation.

Accounting (Banker's) method

- Charge *i* th operation a fictitious ***amortized cost*** *c(i)*, assuming $1 pays for 1 unit of work (time).
- This fee is consumed to perform the operation.
- Any amount not immediately consumed is stored in the ***bank*** for use by subsequent operations.
- The idea is *to impose an extra charge on inexpensive operations* and use it to pay for expensive operations later on.
- The bank balance ***must not go negative***!
- We must ensure that $\sum_{i=1}^{n} t(i) \leq \sum_{i=1}^{n} c(i)$ for all *n*.
- Thus, **the total amortized costs provide an upper bound on the total true costs.**

Potential method

ϕ: potential function associated with the algorithm/data-structure

$\phi(i)$: Potential at the end of ith operation

Important conditions to be fulfilled by ϕ

$\phi(0) = 0$

$\phi(i) \geq 0$ for all i

Amortized cost of ith operation $\overset{\text{def}}{=}$ Actual cost of ith operation $+ \Big(\overbrace{\phi(i) - \phi(i-1)}^{\Delta\phi_i \,=\, \text{Potential difference}} \Big)$

Amortized cost of n operations $\geq$ Actual cost of n operations

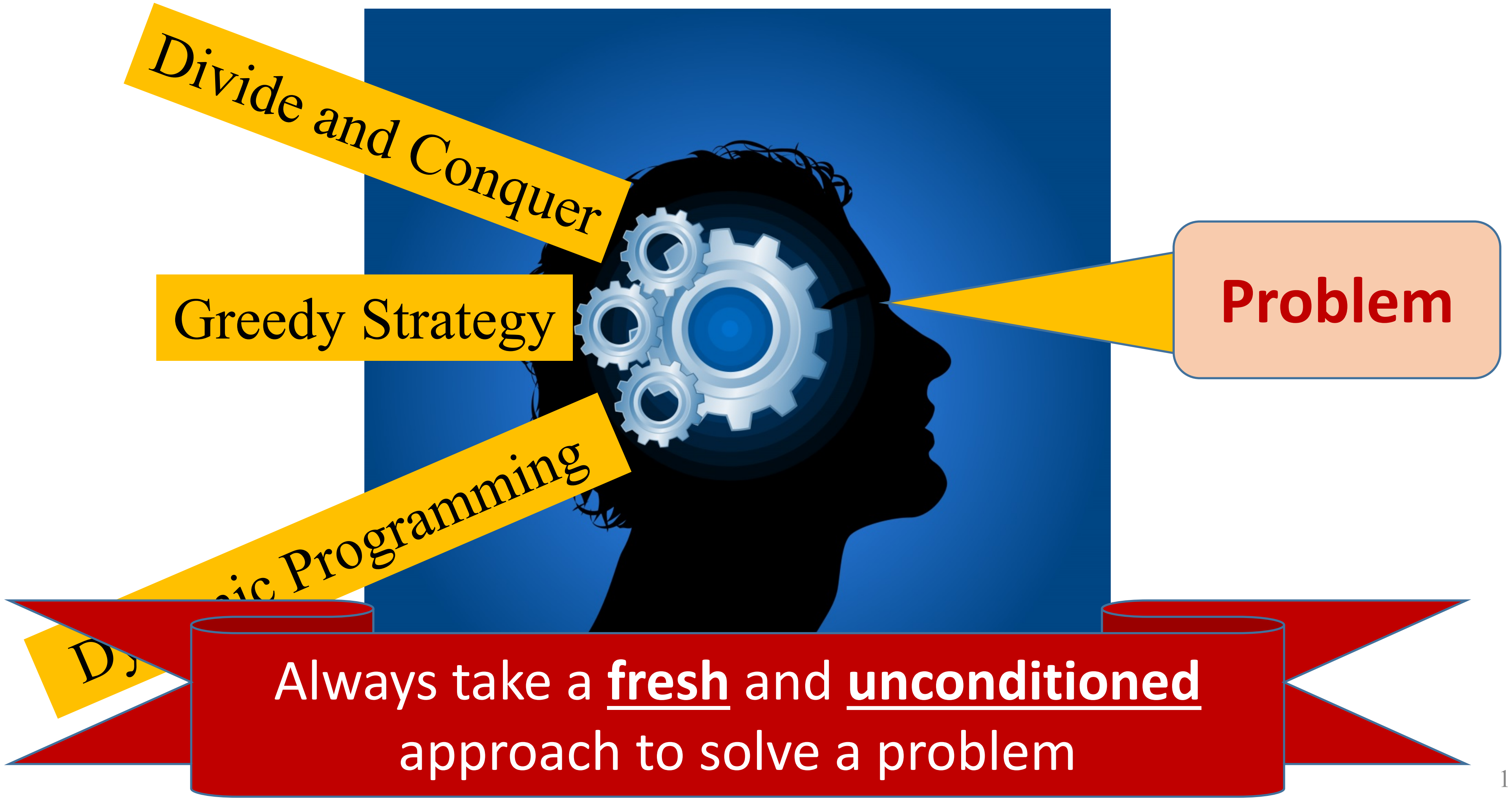
If we want to show that ***actual cost*** of n operations is O(g(n)) then it suffices to show that ***amortized cost*** of n operations is O(g(n))

Potential method - recipe

- Try to select a suitable $\boldsymbol{\phi}$, so that for the costly operation, $\Delta\boldsymbol{\phi}_i$ is negative to such an extent that it *nullifies* or *reduces* the effect of actual cost.
- **Question**: How to find such a suitable potential function $\boldsymbol{\phi}$?

Try to view carefully the costly operation and see if there is some quantity that is "**decreasing**" during the operation.

Designing an algorithm



Always take a **fresh** and **unconditioned** approach to solve a problem

Designing an algorithm

- Fresh and unconditioned approach
- Working on examples and learning from them
- Perseverance
- Theoretical formulation for a better and clear understanding

Some Courses for Interested Students

- Randomized Algorithms
- Optimization Algorithms/ Approximation Algorithms
- Algorithms at Scale
- Advanced Algorithms
- Parallel and Distributed Algorithms
- Computational Complexity

The Course Ends Here

Thanks a lot for

- Giving me an opportunity to share the joy of algorithm design & analysis.
- For sparing your (those who attended the classes) precious time.

I look forward to your feedback/criticism in the review. I shall try my best to improve myself as an instructor of algorithms. ☺

Special Thanks

- Wei Liang
- Eldon Chung
- Govind Venugopalan
- Tran Tan Phat
- Joshua Casey Darian
- Le Quang Tuan
- Zhang Dongping

Acknowledgement

- The slides are modified from
  - The slides from Prof. Kevin Wayne
  - The slides from Prof. Surender Baswana
  - The slides from Prof. Erik D. Demaine and Prof. Charles E. Leiserson
  - The slides from Prof. Arnab Bhattacharya and Prof. Wing-Kin Sung