CS240 Algorithm Design and Analysis

Lecture 2

Greedy Algorithms

Quan Li

Fall 2025

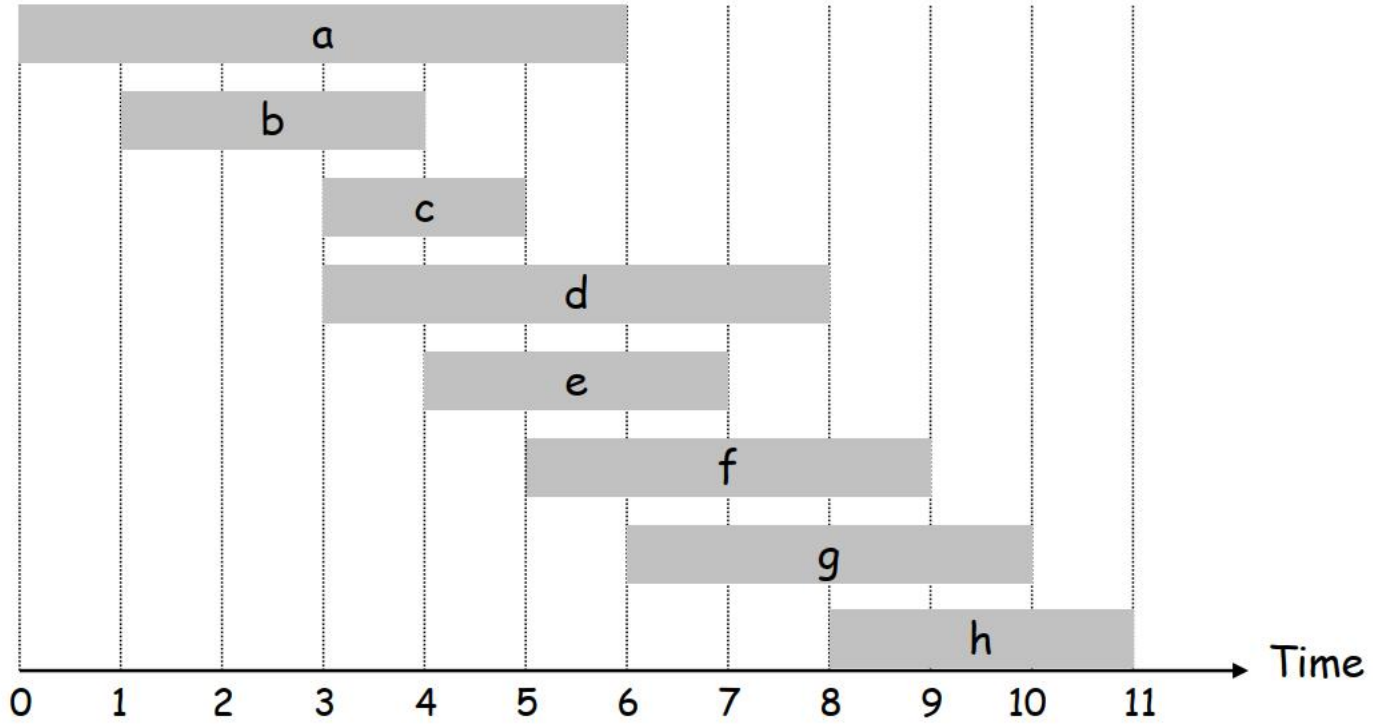
2025.09.18

A greedy algorithm always makes the choice that looks best at the moment and adds it to the current partial solution

Greedy algorithms don't always yield optimal solutions, but when they do, they're usually the simplest and most efficient algorithms available

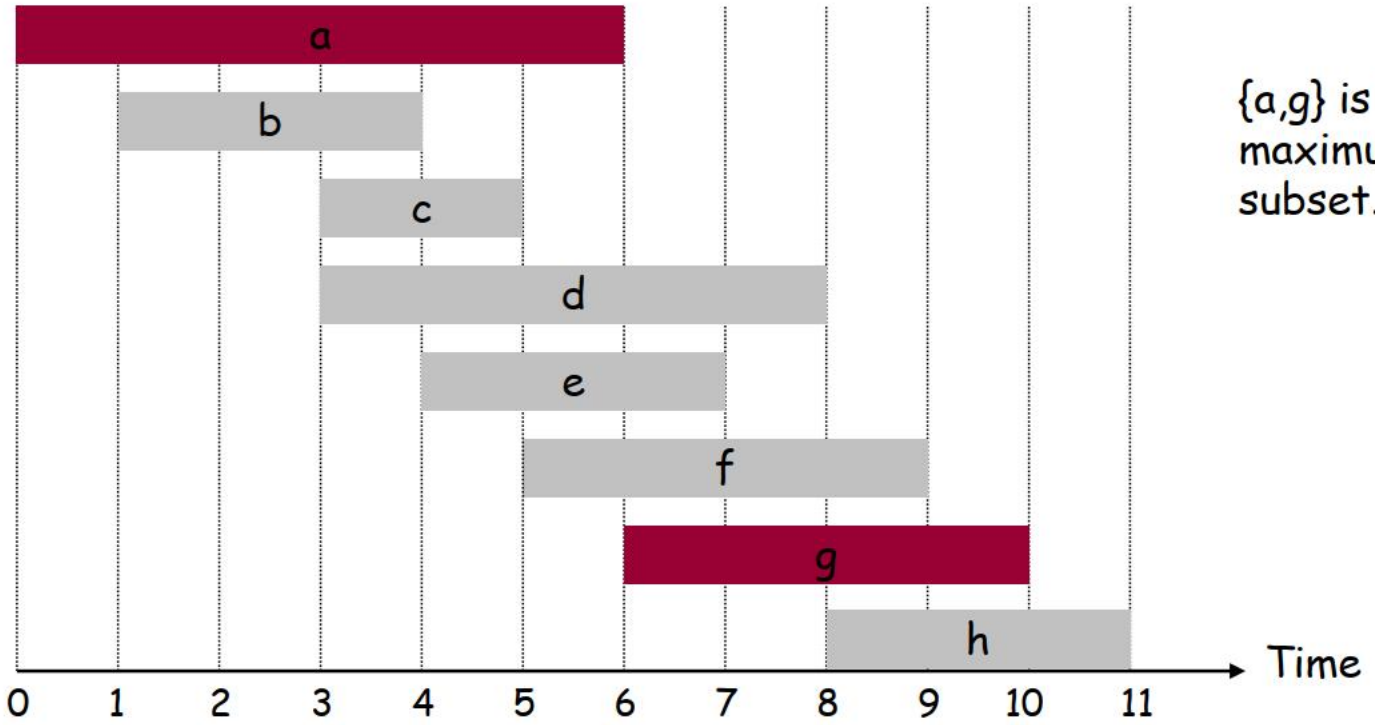
Interval Scheduling

- Interval scheduling
  - Job j starts at s_j and finishes at f_j
  - Two jobs are **compatible** if they don't overlap
  - Goal: find maximum size subset of mutually compatible jobs



Interval Scheduling

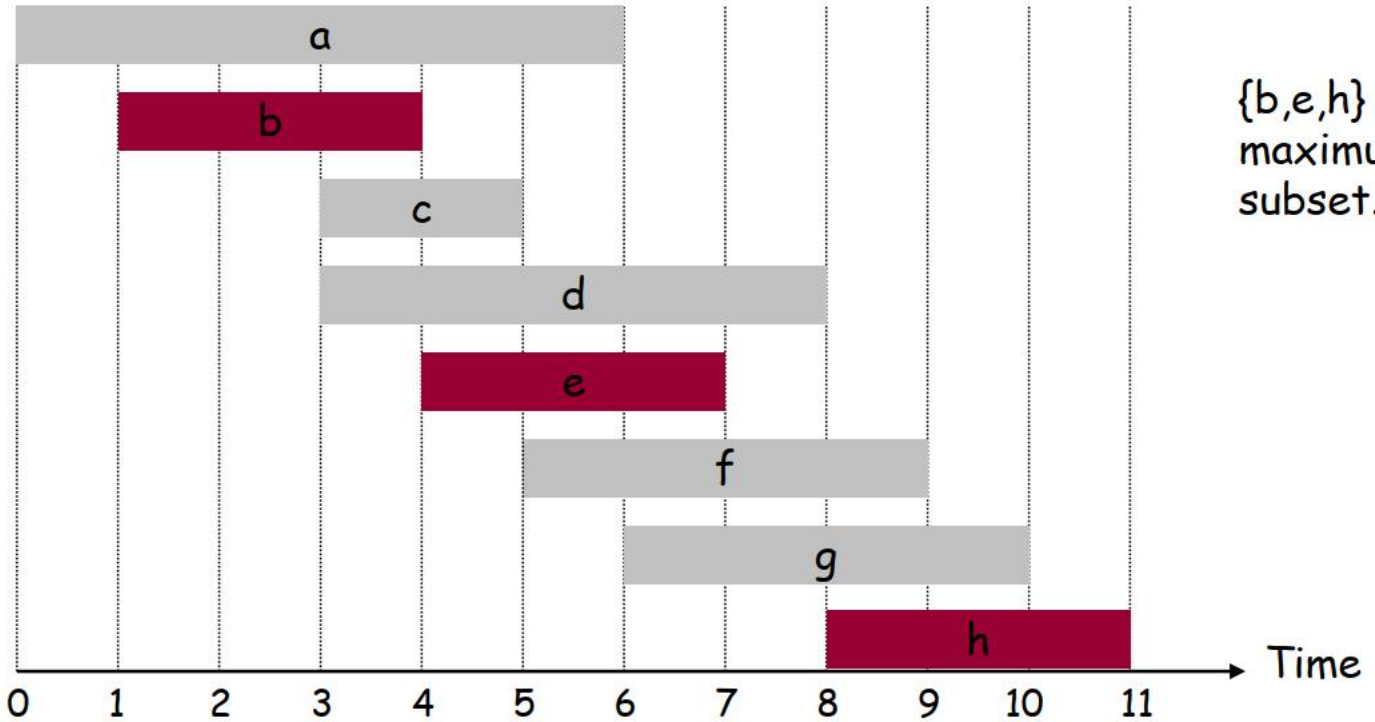
- Interval scheduling
  - Job j starts at s_j and finishes at f_j
  - Two jobs are **compatible** if they don't overlap
  - Goal: find maximum size subset of mutually compatible jobs



{a,g} is NOT a maximum-size subset.

Interval Scheduling

- Interval scheduling
  - Job j starts at s_j and finishes at f_j
  - Two jobs are **compatible** if they don't overlap
  - Goal: find maximum size subset of mutually compatible jobs



{b,e,h} is a maximum-size subset.

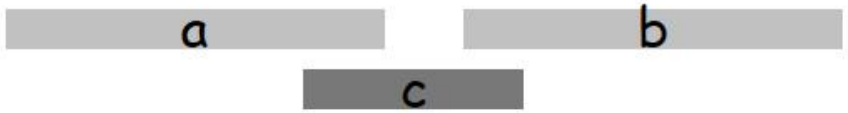
Interval Scheduling: Greedy Algorithms

- Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken
- Usually, many compatible jobs can co-exist
- Need to choose a rule specifying which job to choose next
- Three possible rules are:
  - [Earliest start time]
    - Consider jobs in ascending order of start time s_j
  - [Earliest finish time]
    - Consider jobs in ascending order of finish time f_j
  - [Shortest interval]
    - Consider jobs in ascending order of interval length $f_j - s_j$
  - [Fewest conflicts]
    - For each job, count the number of conflicting jobs c_j. Schedule in ascending order of conflicts c_j

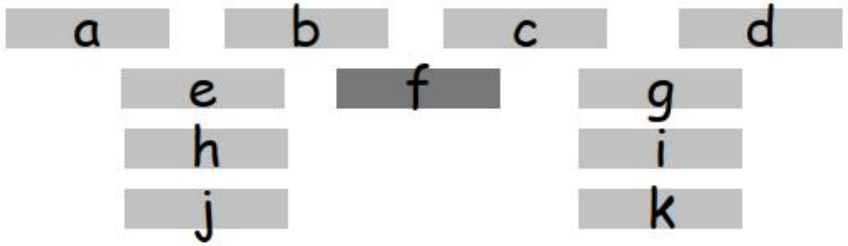
Interval Scheduling: Greedy Algorithms

- Greedy template. Consider jobs in some order. Take a job provided it's compatible with the ones already taken

Order on earliest start time
Chooses {e} instead of {a, b, c, d}



Order on shortest interval
Chooses {c} instead of {a, b}

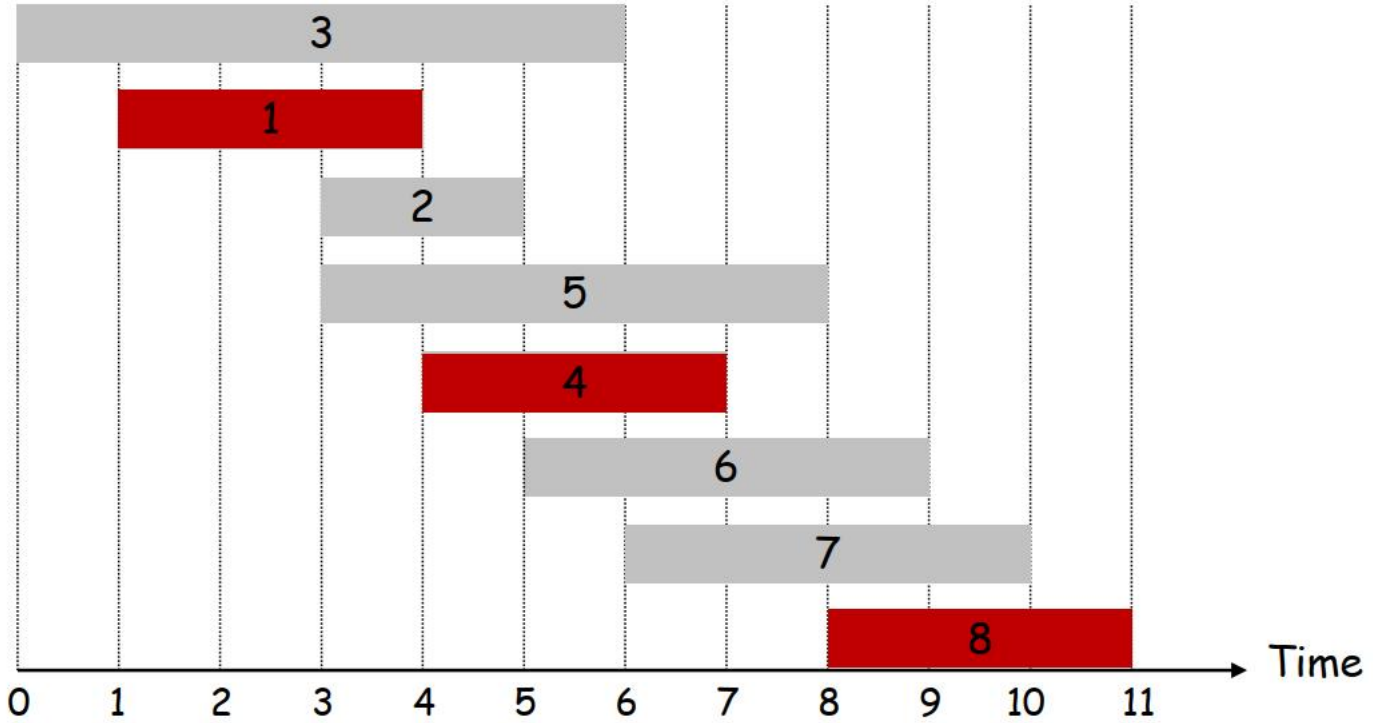


Order on fewest conflicts
Chooses {f} which forces choosing {a, f, d} instead of {a, b, c, d}

Examples above provide counterexamples for the three rules proposed.
For each of them, provides an example input for which the following that rule yields a non-optimal (i.e., non max-size) schedule

Interval Scheduling: Greedy Algorithm

- Greedy algorithm. Consider jobs in increasing order of **finish time.** Take each job provided it's compatible with the ones already taken



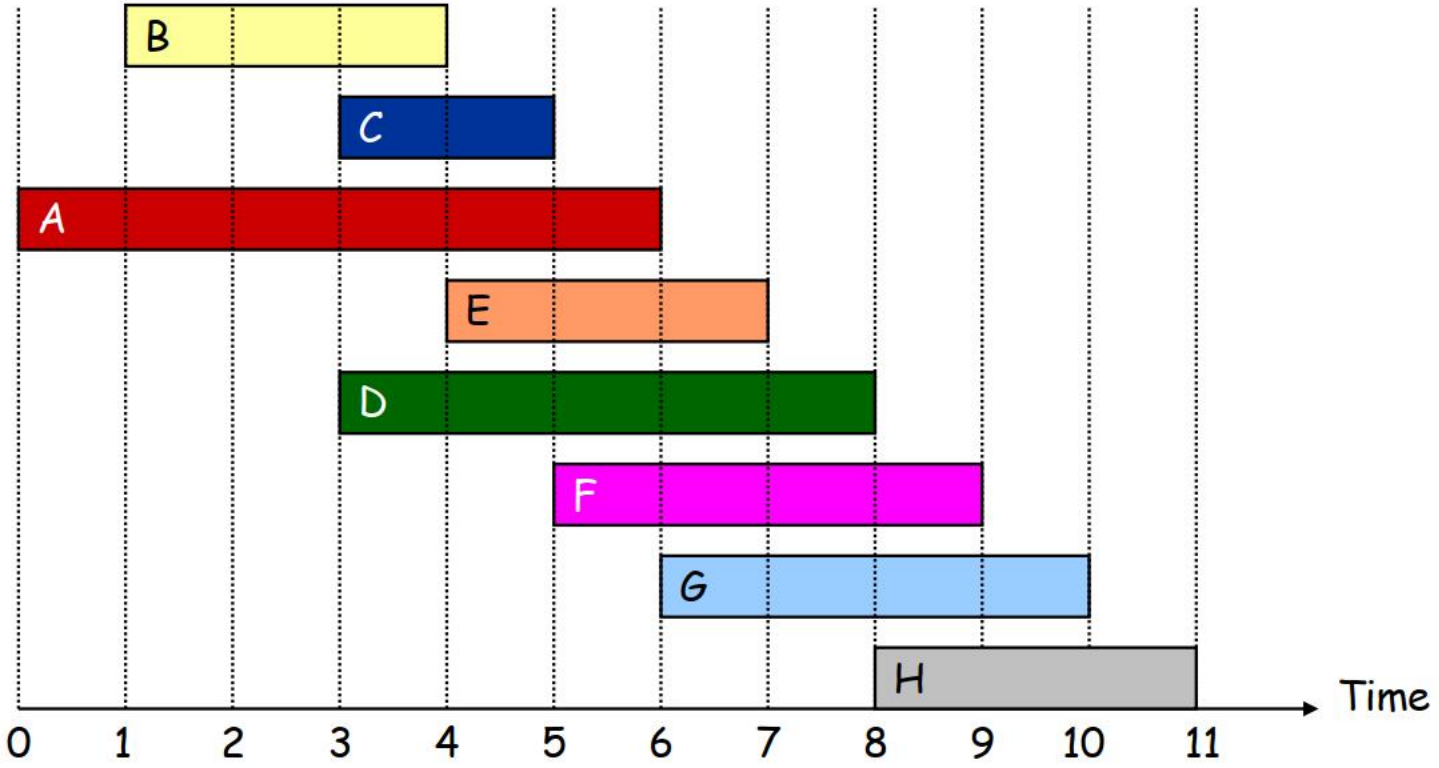
Interval Scheduling: Greedy Algorithm

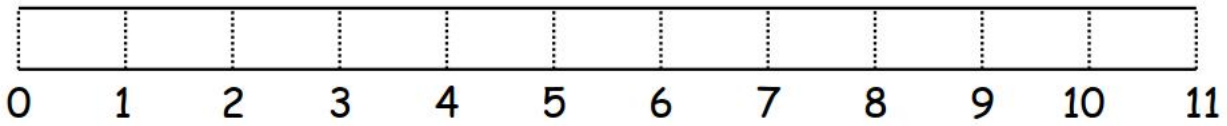
- Greedy algorithm. Consider jobs in increasing order of **finish time**. Take each job provided it's compatible with the ones already taken

```
Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n
A ← ∅,  last ← 0
for j ← 1 to n
    if s_j ≥ last then A ← A ∪ {j},  last ← f_j
return A
```

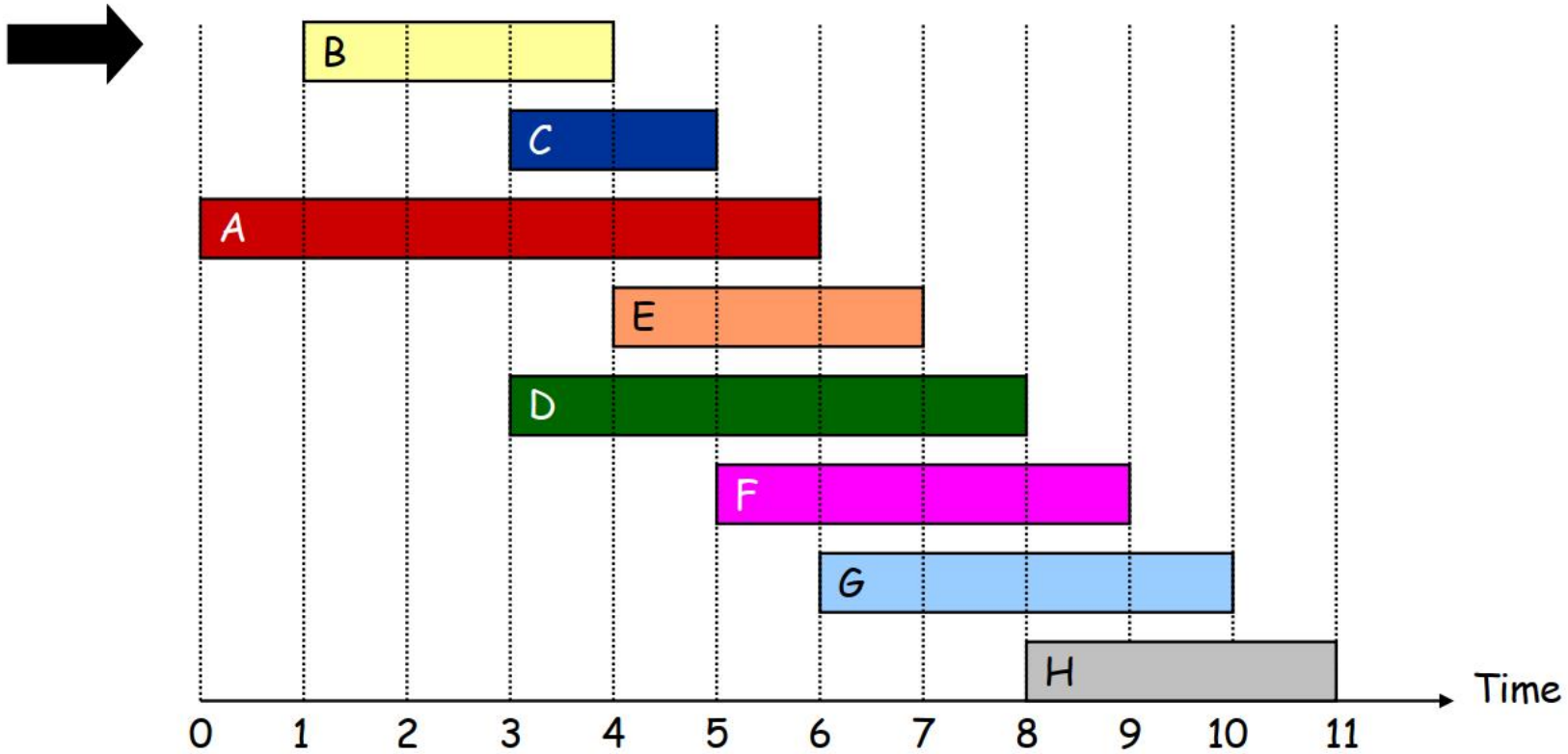
- **Running time dominated by cost of sorting: Θ(nlogn)**
- Remember the finish time of the last job (j^*) added to A
- Job j is compatible with A iff s_j >= last (i.e., f_{j^*})
- **Remember:** Correctness (optimality) of greedy algorithms is usually not obvious.
- **Need to prove**!

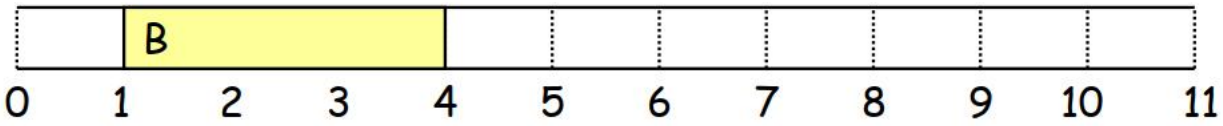
Interval Scheduling



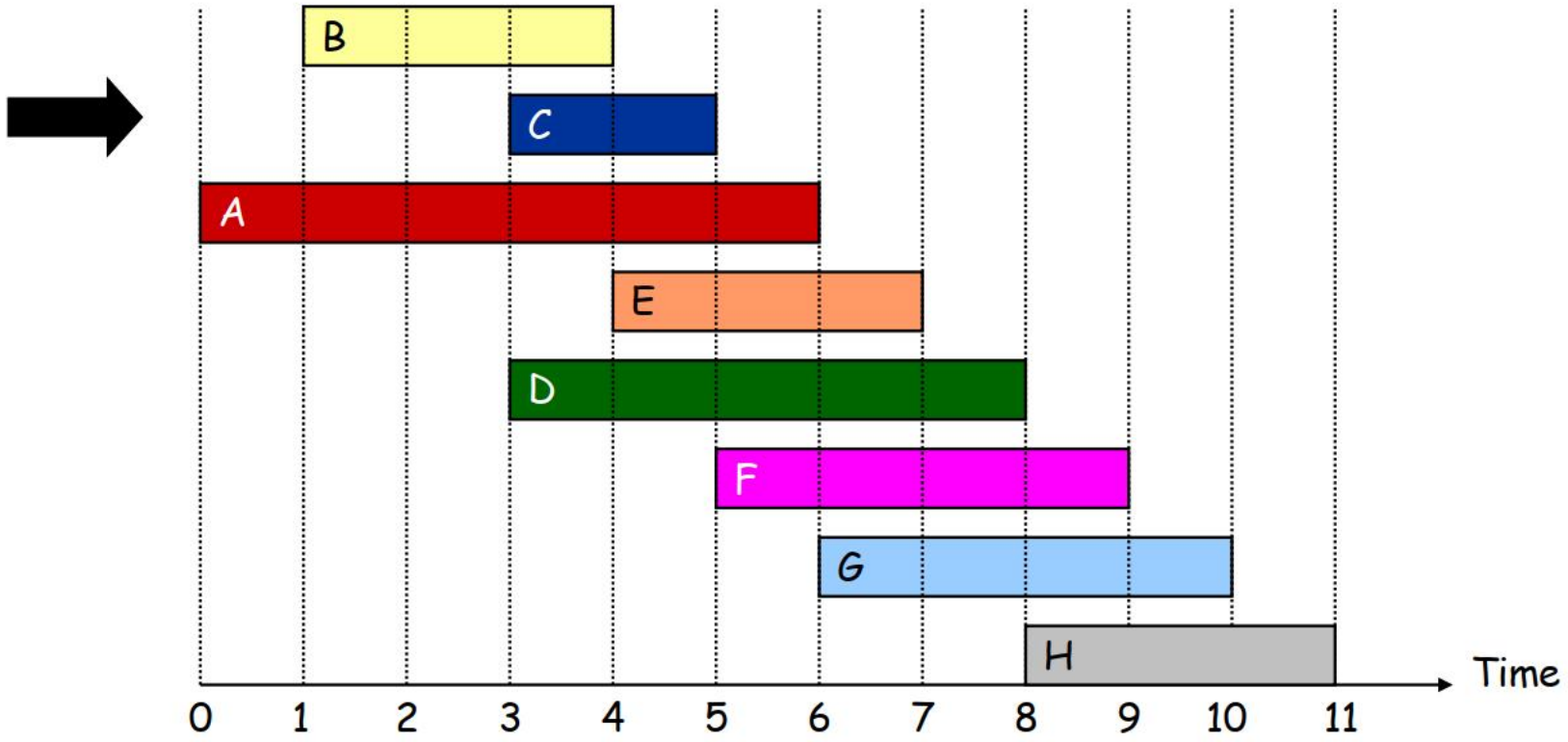


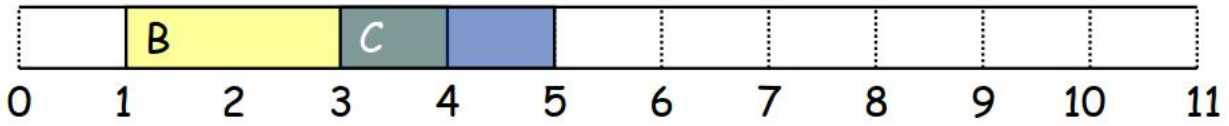
Interval Scheduling



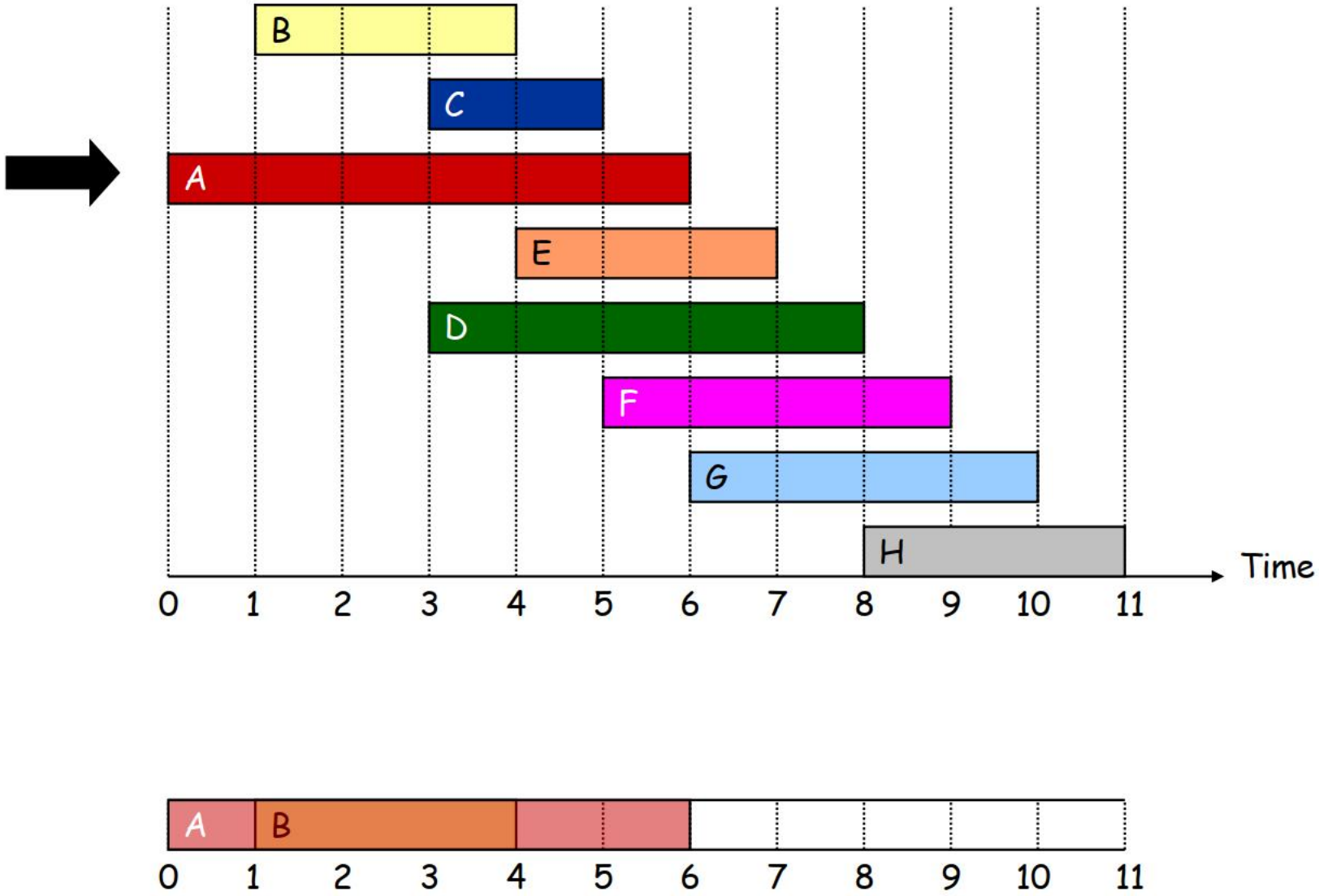


Interval Scheduling

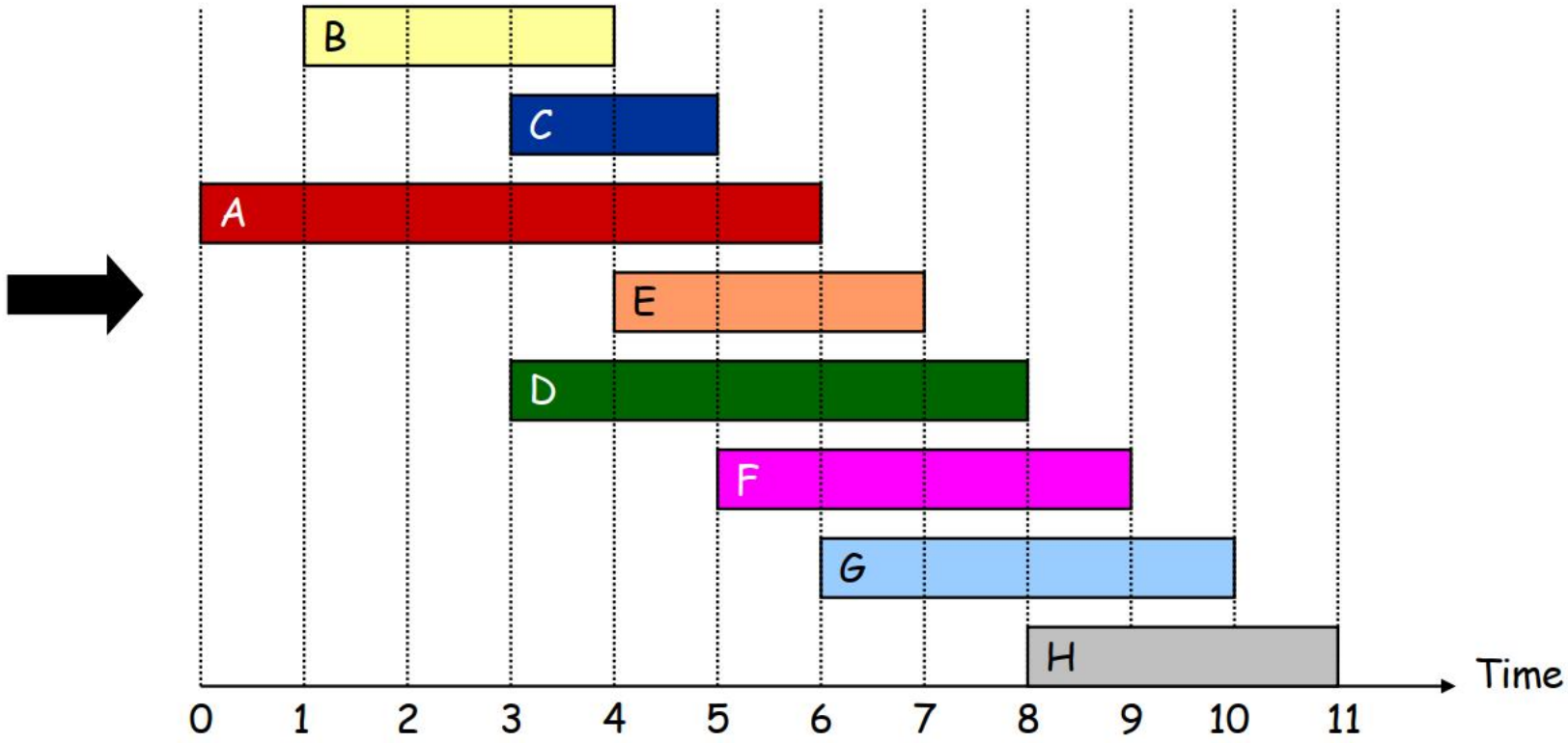


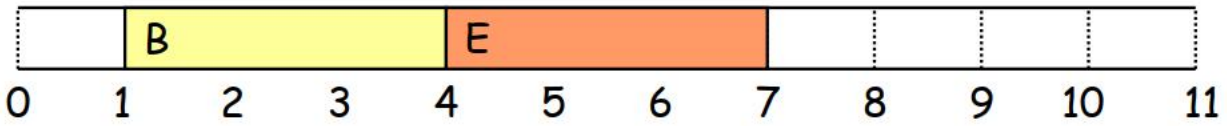


Interval Scheduling

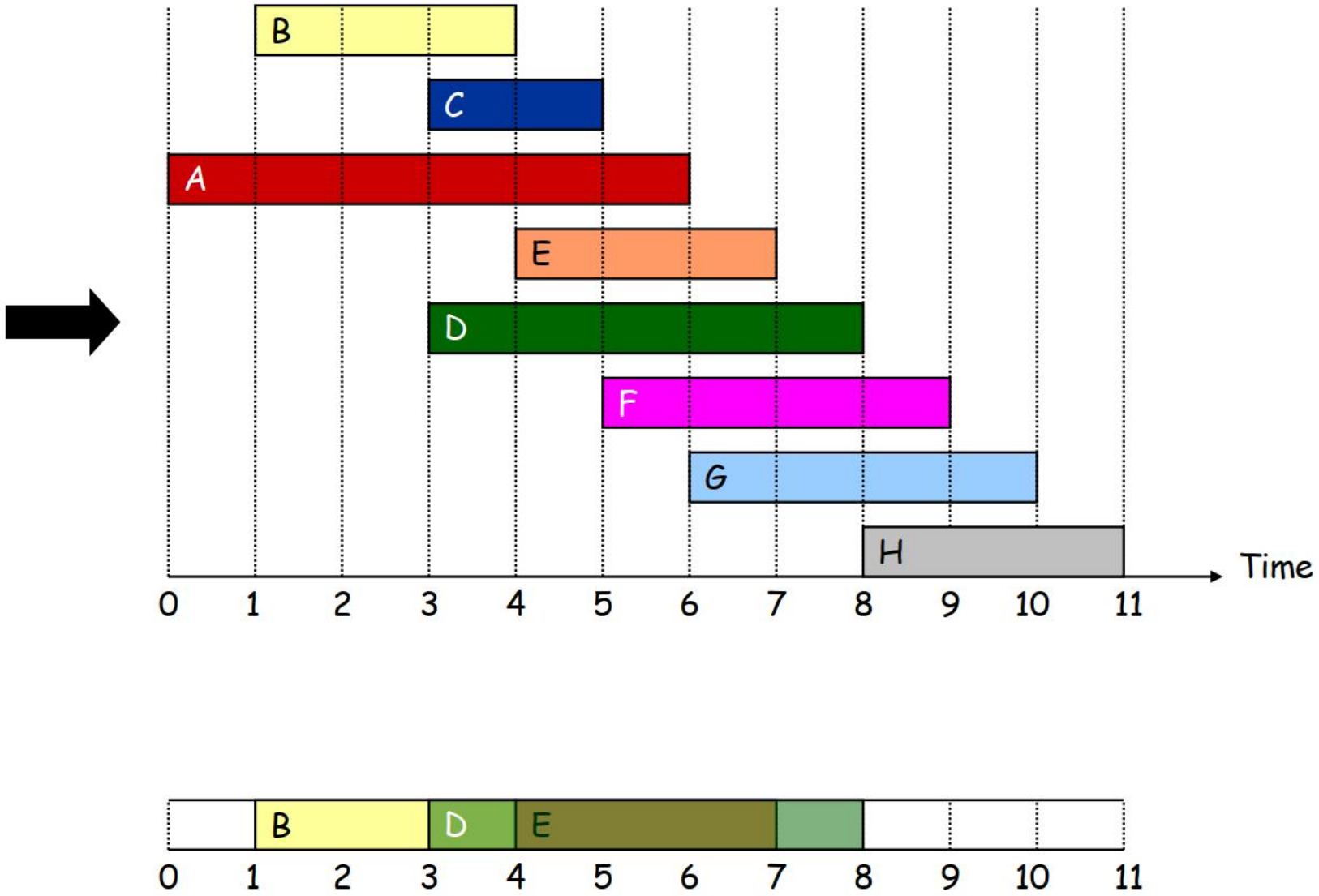


Interval Scheduling

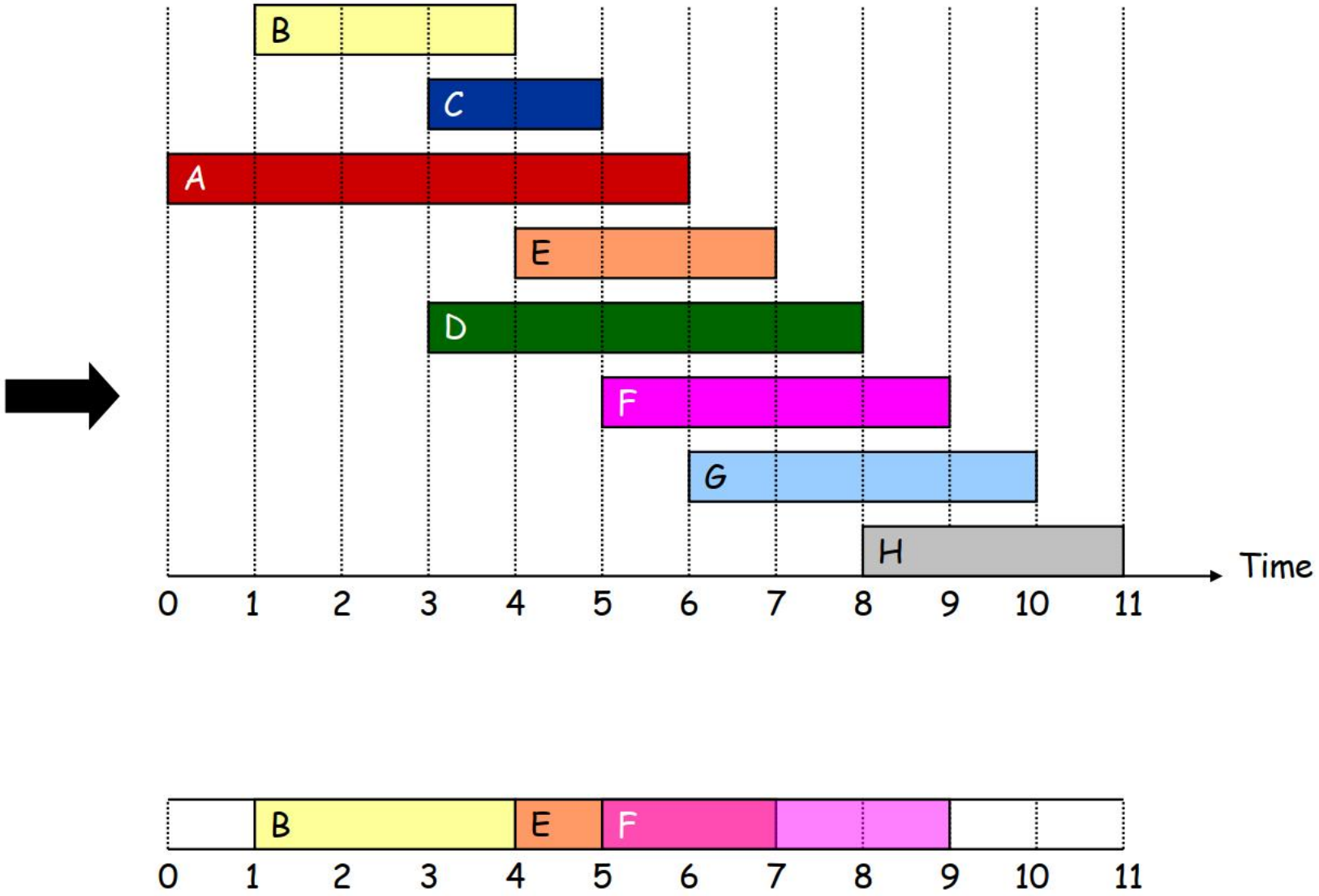




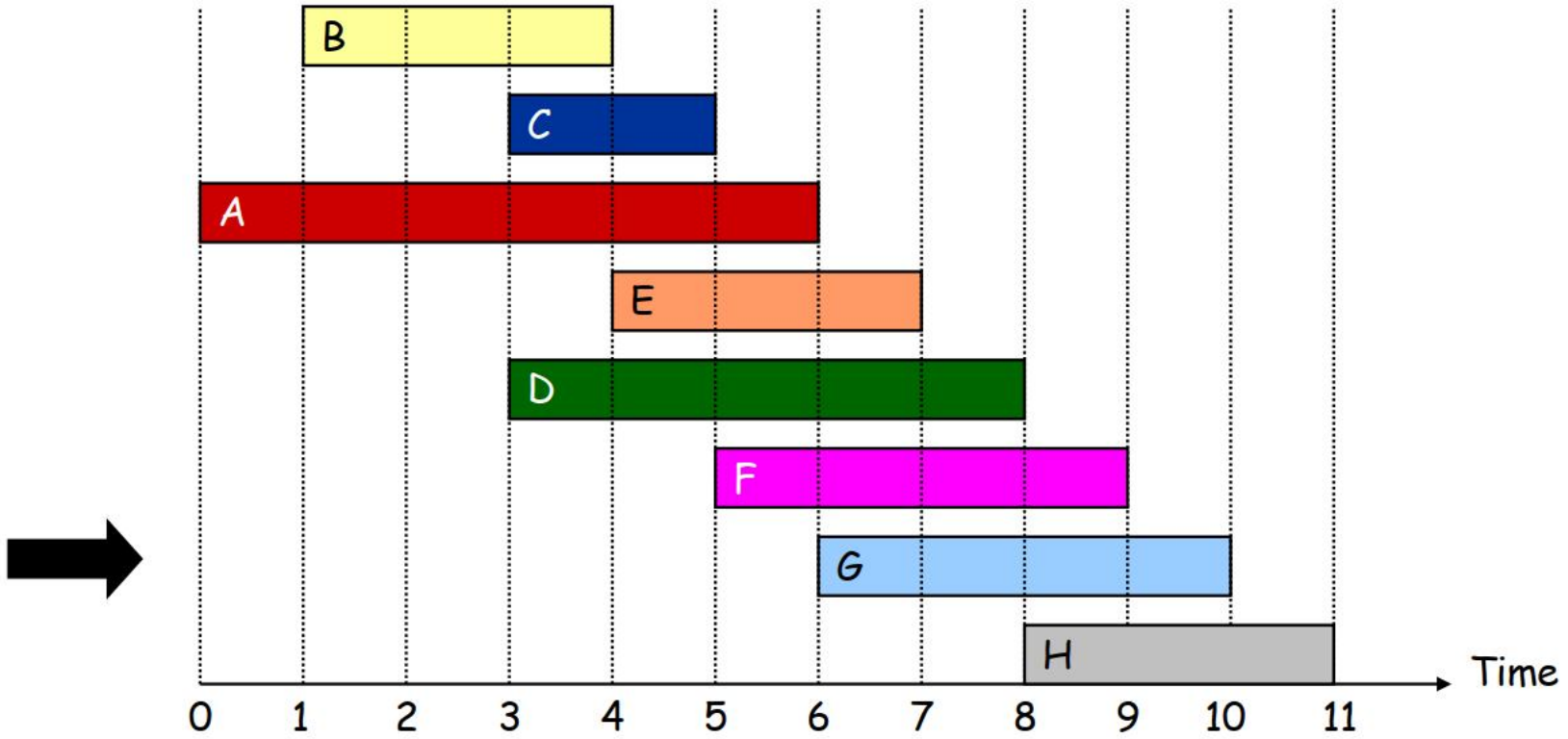
Interval Scheduling



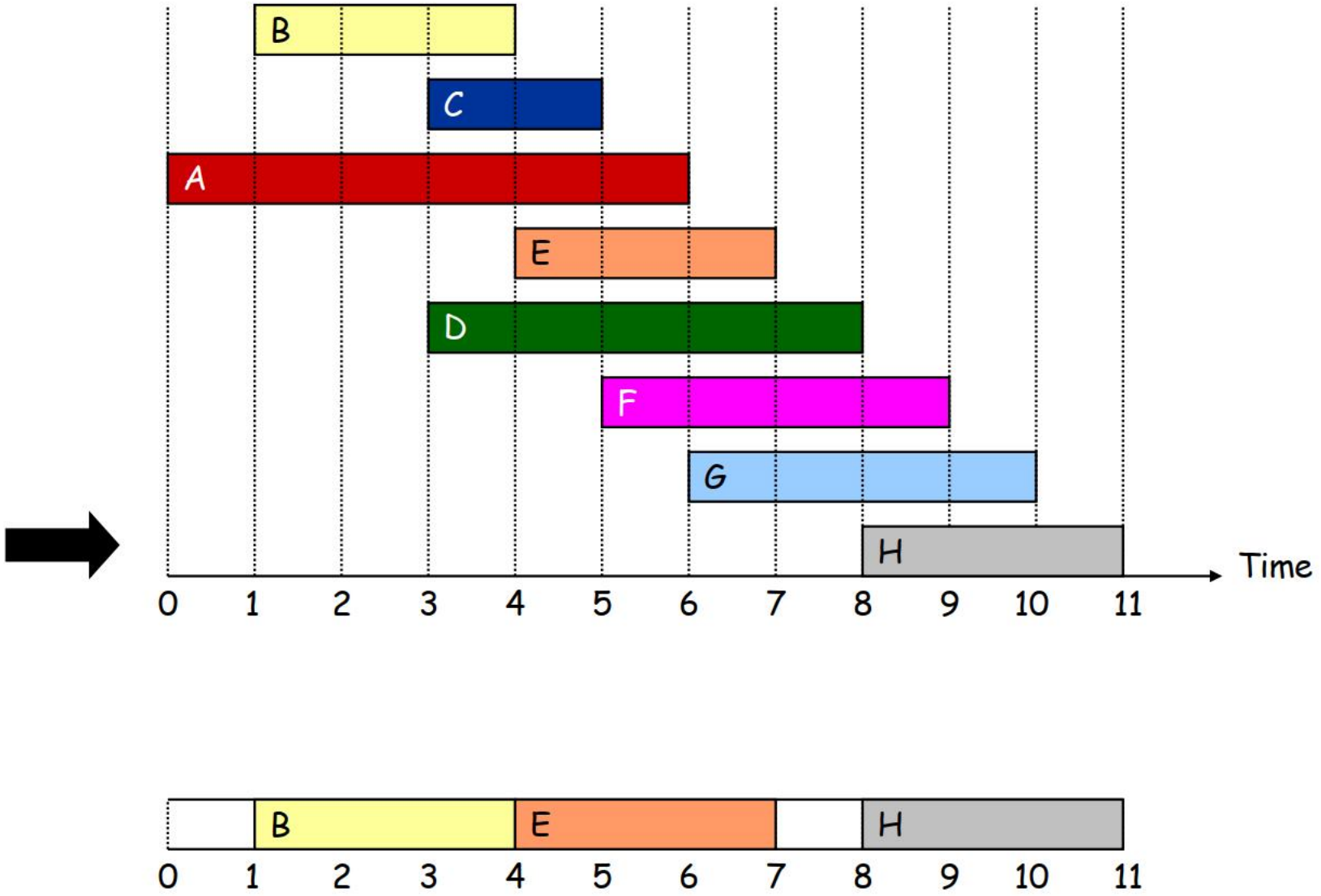
Interval Scheduling



Interval Scheduling



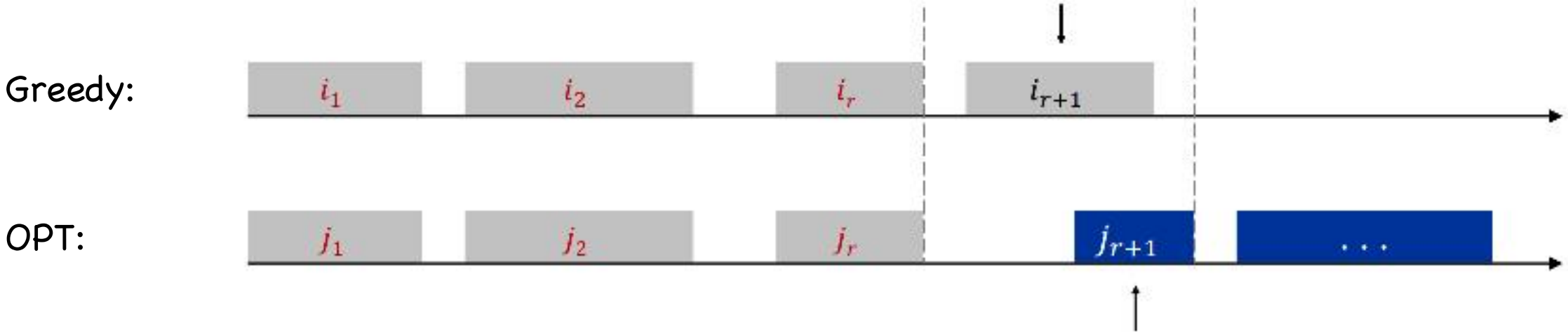
Interval Scheduling



Interval Scheduling: Correctness

- Theorem. Greedy algorithm is optimal.
- Proof (by contradiction)
  - Assume Greedy is different from OPT. Let's see what's different.
  - Let $i_1, i_2, \ldots, i_k$ denote the set of jobs selected by greedy.
  - Let $j_1, j_2, \ldots, j_m$ denote set of jobs in the optimal solution.
  - Find largest possible value of r such that $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$

By definition of Greedy, job i_{r+1} finishes before j_{r+1}

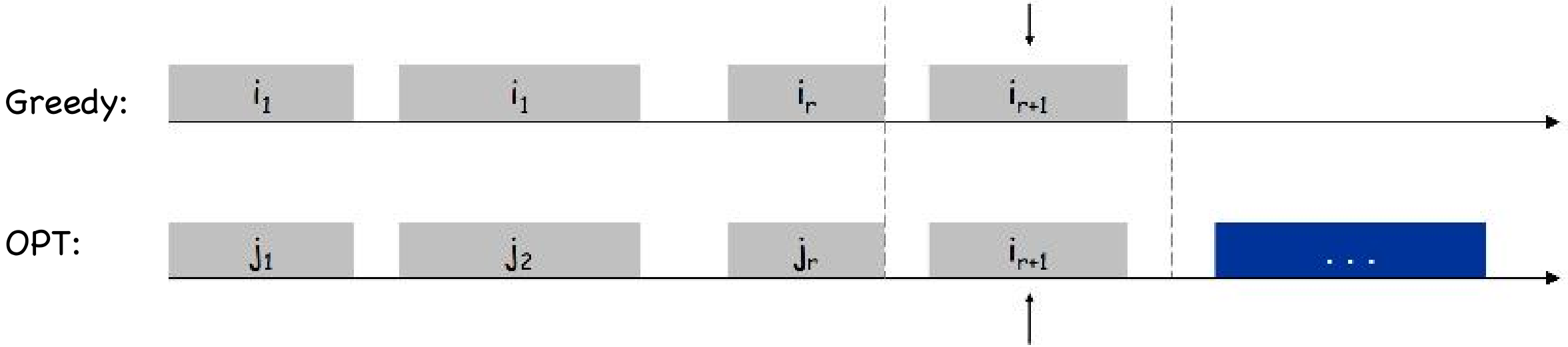


Why not replace job j_{r+1} with job i_{r+1}

Interval Scheduling: Analysis

- Theorem. Greedy algorithm is optimal.
- Pf. (by contradiction)
  - Assume greedy is not optimal, and let's see what happens
  - Let $i_1, i_2, \ldots, i_k$ denote the set of jobs selected by greedy.
  - Let $j_1, j_2, \ldots, j_k$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ **for the largest possible value of r.**

By definition of Greedy, job i_{r+1} finishes before j_{r+1}



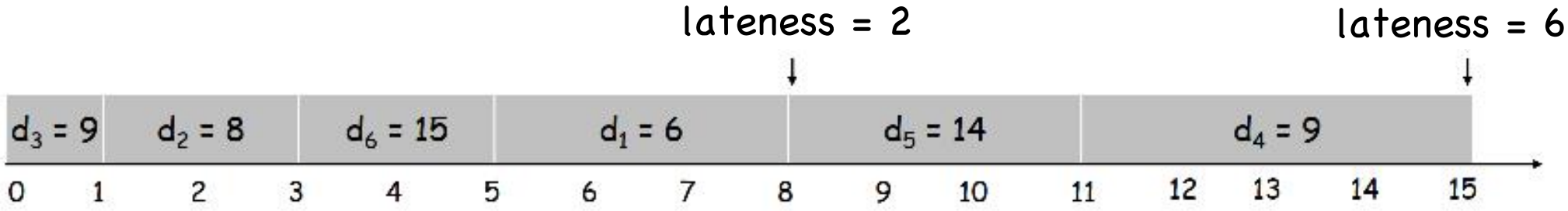
Solution still feasible and optimal, but contradicts maximality of r

Scheduling to Minimize Lateness

Scheduling to Minimizing Lateness

- **Minimizing lateness problem**
  - Single resource processes one job at a time
  - Job j requires t_j units of processing time and is due at time d_j
  - If j starts at time s_j, it finishes at time $f_j = s_j + t_j$
  - Lateness: $l_j = \max\{ 0, f_j - d_j \}$
  - Goal: schedule all jobs to minimize **maximum** lateness $L = \max l_j$

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Minimizing lateness: Greedy Algorithms

- Greedy template: Consider jobs in some order
- [Shortest processing time first] Consider jobs in ascending order of processing time t_j
- [Earliest deadline first] Consider jobs in ascending order of deadline d_j
- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$

Minimizing lateness: Greedy Algorithms

- Greedy template: Consider jobs in some order

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j

	1	2
t_j	1	10
d_j	100	10

counterexample

- [Earliest deadline first] Consider jobs in ascending order of deadline d_j

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$

	1	2
t_j	1	10
d_j	2	10

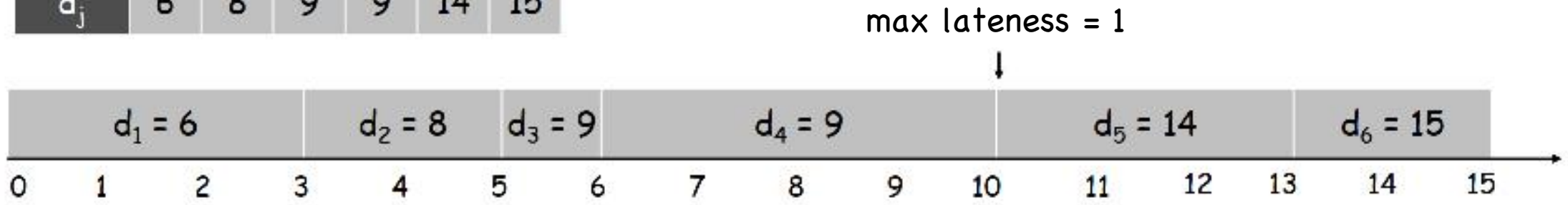
counterexample

Minimizing Lateness: Greedy Algorithm

- **Greedy algorithm. Earliest deadline first**

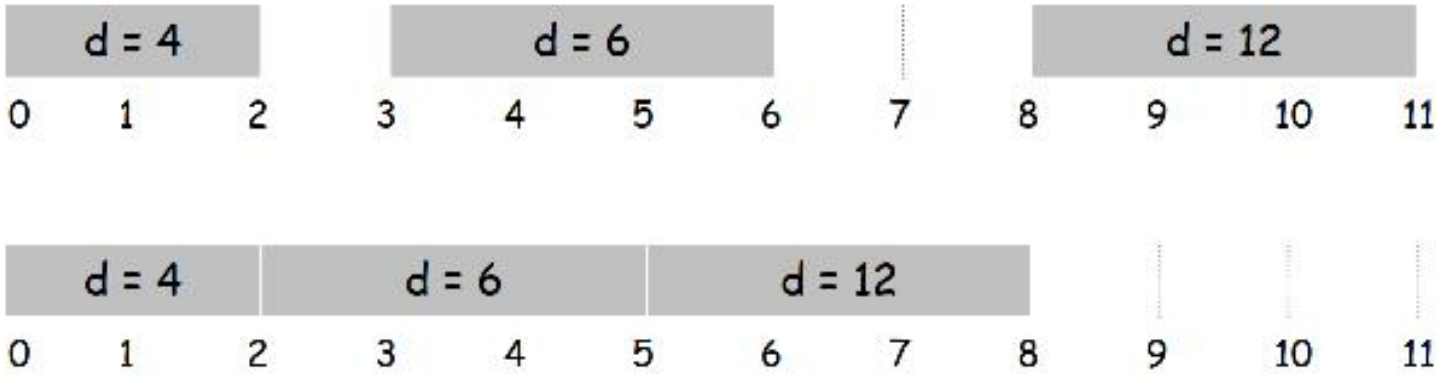
```
Sort n jobs by deadline so that d_1 ≤ d_2 ≤ … ≤ d_n

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + t_j]
   s_j ← t, f_j ← t + t_j
   t ← t + t_j
output intervals [s_j, f_j]
```

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Minimizing Lateness: No Idle Time

- **Observation**. There exists an optimal schedule with no **idle time**



- **Observation.** The greedy schedule has no idle time

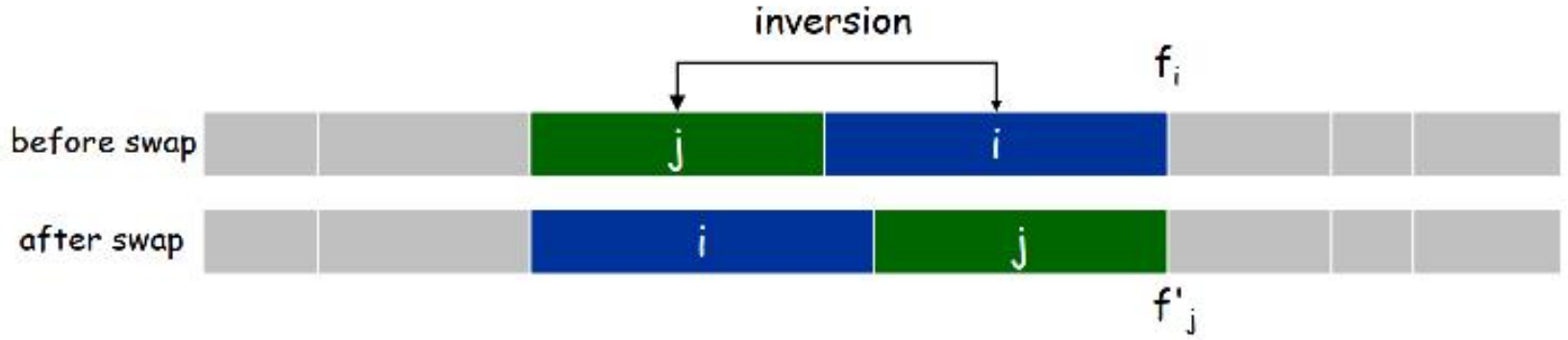
Minimizing Lateness: Inversions

- **Def.** An **inversion** in schedule S is a pair of jobs i and j such that: i < j but j scheduled before i

- Observation. Greedy schedule has no inversions
- Observation. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively

Minimizing Lateness: Inversions

- **Def.** An **inversion** in schedule S is a pair of jobs i and j such that: i < j but j scheduled before i



- **Claim.** Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness
- **Pf.** Let l be the lateness before the swap, and let l′ be it afterwards
- $l'_k = l_k$ for all $k \neq i, j$
- $l'_i \leq l_i$
- If job j is late:

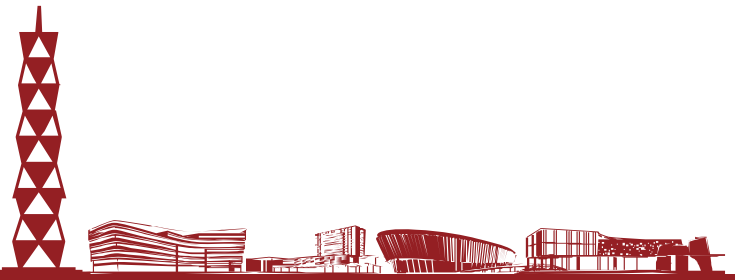
$$\begin{aligned} \ell'_j &= f'_j - d_j && \text{(definition)} \\ &= f_i - d_j && (j \text{ finishes at time } f_i) \\ &\leq f_i - d_i && (i < j) \\ &\leq \ell_i && \text{(definition)} \end{aligned}$$

Minimizing Lateness: Analysis of Greedy Algorithm

- **Theorem.** Greedy schedule S is optimal
- **Pf.** Define S^* to be an optimal schedule that has the fewest number of inversions, and let's see what happens
  - Can assume S^* has no idle time
  - If S^* has no inversions, then $S = S^*$
  - If S^* has an inversion, let i – j be an adjacent inversion
    - Swapping i and j does not increase the maximum lateness and strictly decreases the number of inversions
    - This contradicts definition of S^*

Greedy Analysis Strategies

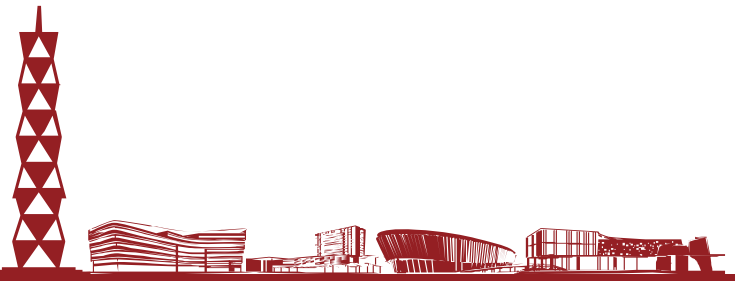
- **Greedy algorithm stays ahead.** Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's
- **Exchange argument.** Gradually transform any solution to the one found by the greedy algorithm without hurting its quality



Minimum Spanning Tree

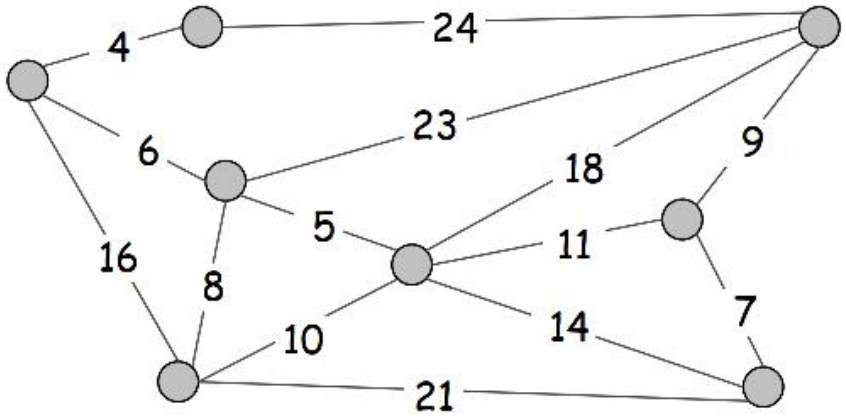
Minimum Spanning Trees

- **Definition**
- Prim's Algorithm
- The Cut Lemma
  - Correctness of Prim's Algorithm
  - Uniqueness of MSTs (under distinct weight assumption)
- Kruskal's Algorithm
  - Basic Idea
  - Correctness of Kruskal's Algorithm

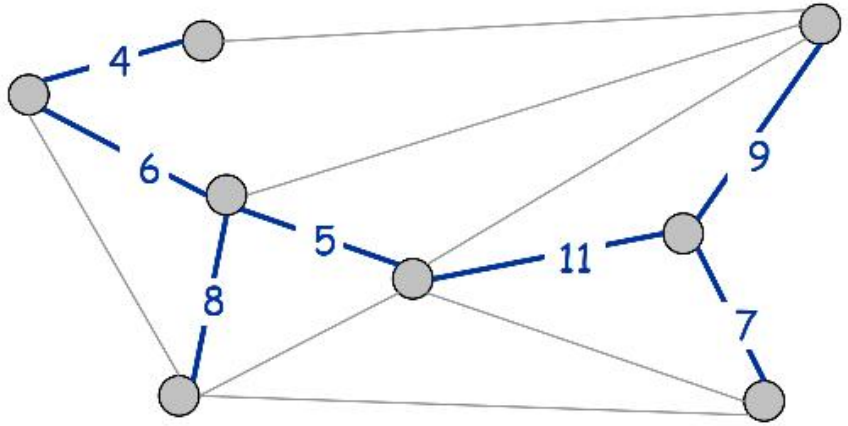


Minimum Spanning Trees

- **Minimum spanning tree.** Given a connected undirected graph G = (V, E) with real-valued edge weights w(e), an MST is a subset of the edges T⊆E such that T is a tree that connects all nodes whose sum of edge weights is minimized.



$G = (V, E)$

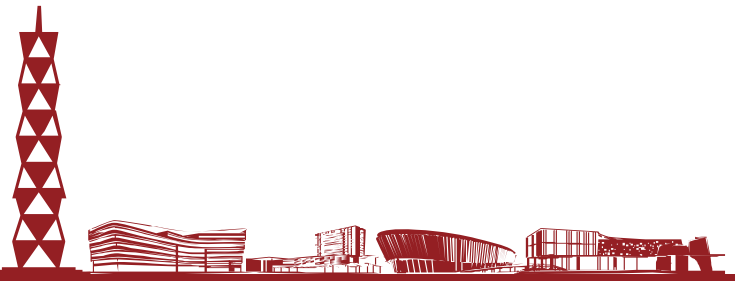


$T, \Sigma_{e \in T}\, w(e) = 50$

- Applications: telephone networks, electrical and hydraulic systems, TV cables, computer networks, road systems

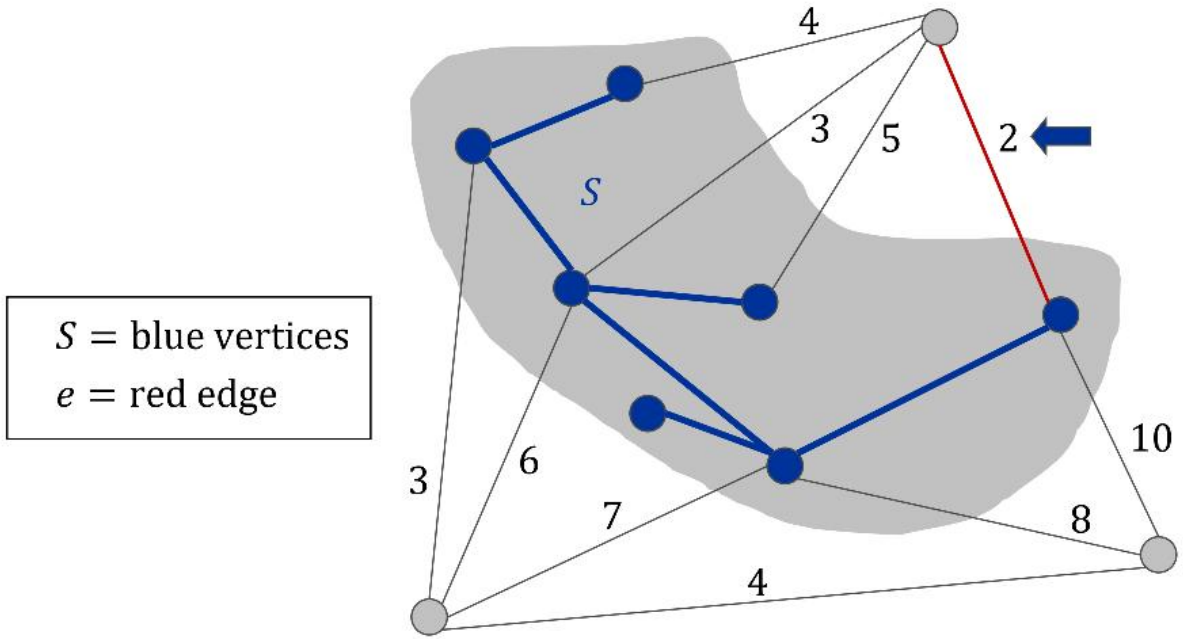
Minimum Spanning Trees

- Definition
- **Prim's Algorithm**
- The Cut Lemma
  - Correctness of Prim's Algorithm
  - Uniqueness of MSTs (under distinct weight assumption)
- Kruskal's Algorithm
  - Basic Idea
  - Correctness of Kruskal's Algorithm



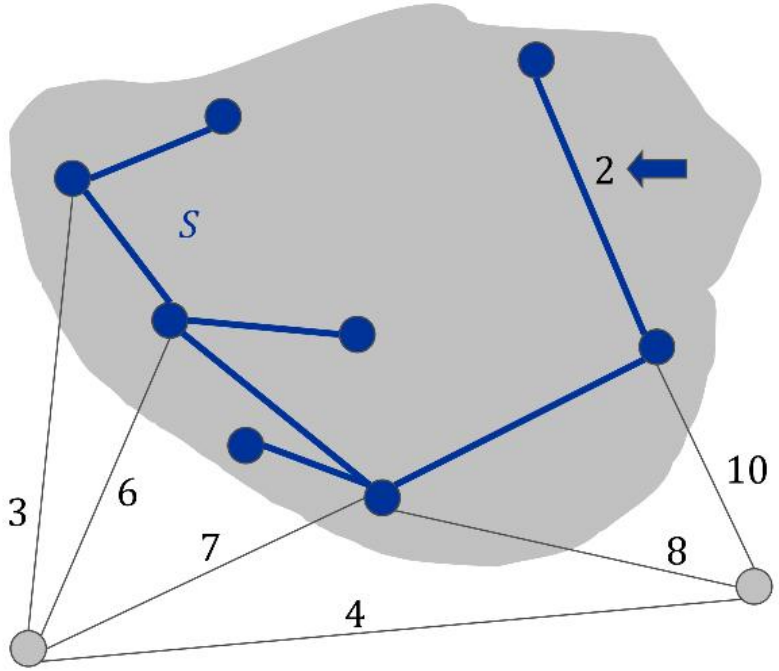
Prim's Algorithm: Idea

- **Prim's algorithm**
  - Initialize S = {any one node}
  - Add min cost edge e = (u, v) with u $\in$ S and v $\in$ V − S to T
  - Add v to S
  - Repeat until S = V



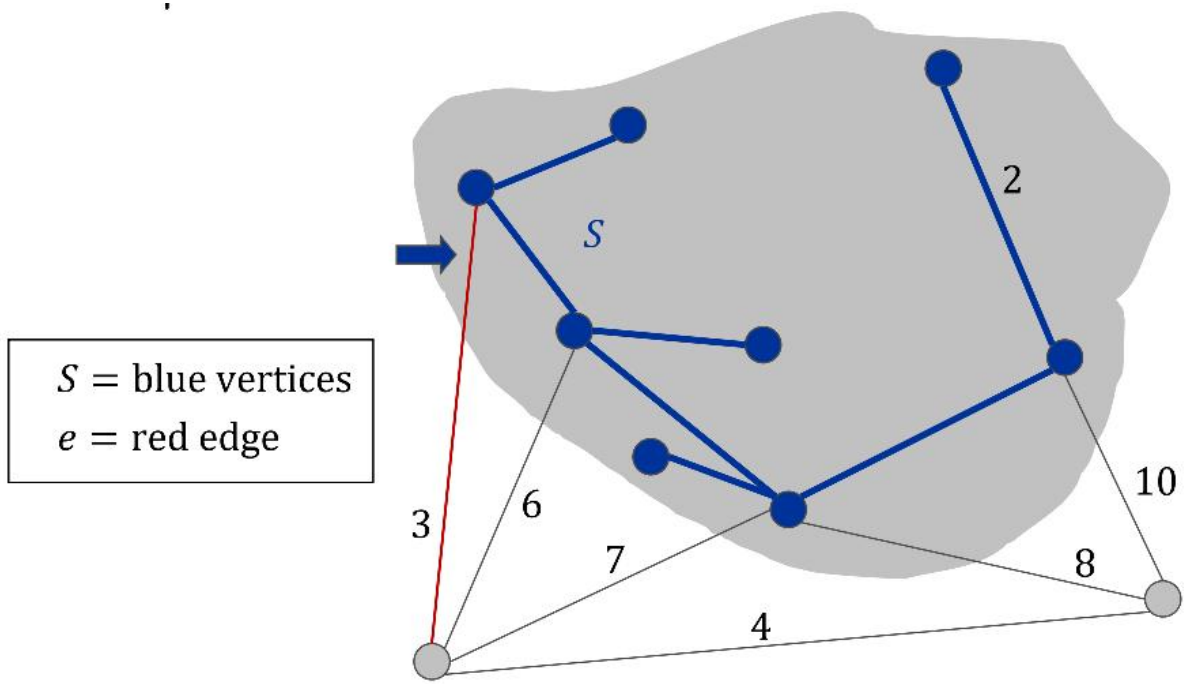
Prim's Algorithm: Idea

- **Prim's algorithm**
  - Initialize S = {any one node}
  - Add min cost edge e = (u, v) with u $\in$ S and v $\in$ V − S to T
  - Add v to S
  - Repeat until S = V



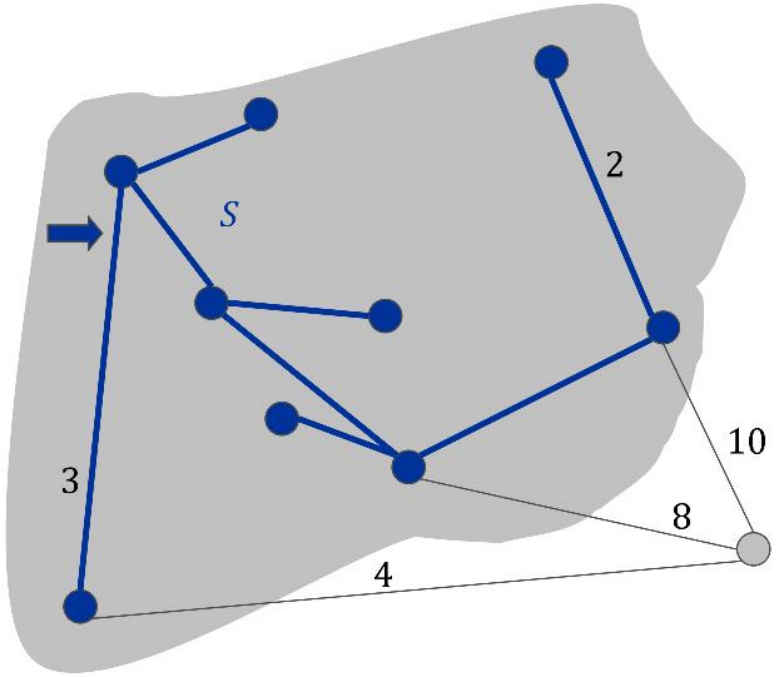
Prim's Algorithm: Idea

- **Prim's algorithm**
  - Initialize S = {any one node}
  - Add min cost edge e = (u, v) with u $\in$ S and v $\in$ V − S to T
  - Add v to S
  - Repeat until S = V



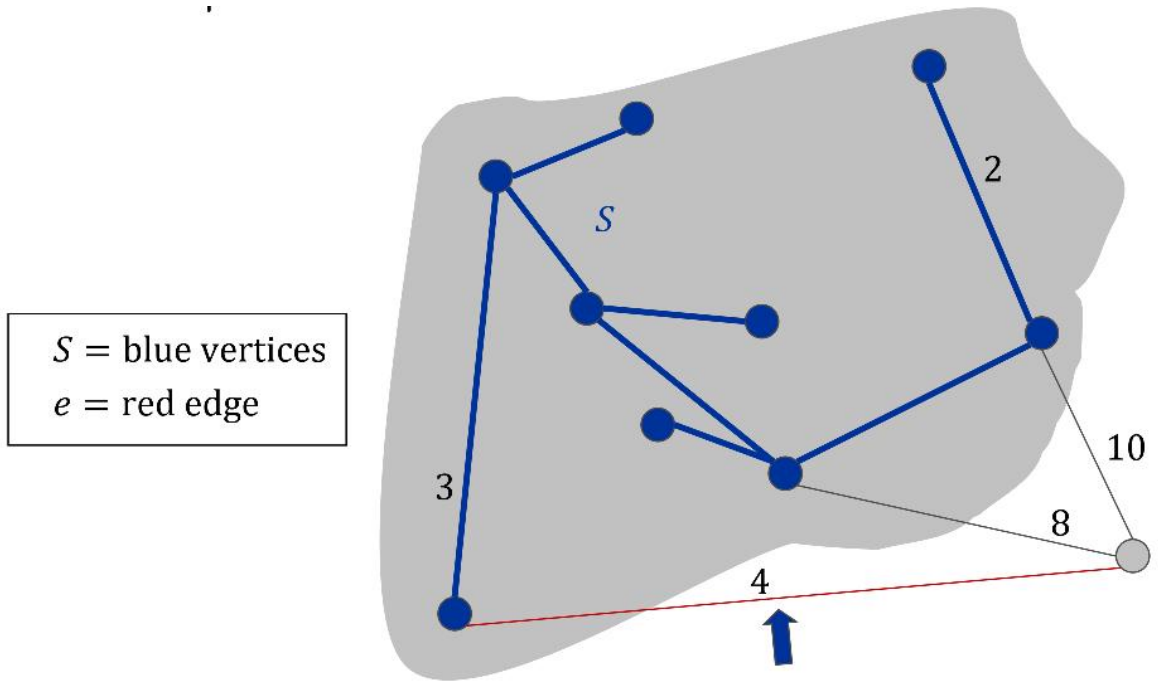
Prim's Algorithm: Idea

- **Prim's algorithm**
  - Initialize S = {any one node}
  - Add min cost edge e = (u, v) with u $\in$ S and v $\in$ V − S to T
  - Add v to S
  - Repeat until S = V



Prim's Algorithm: Idea

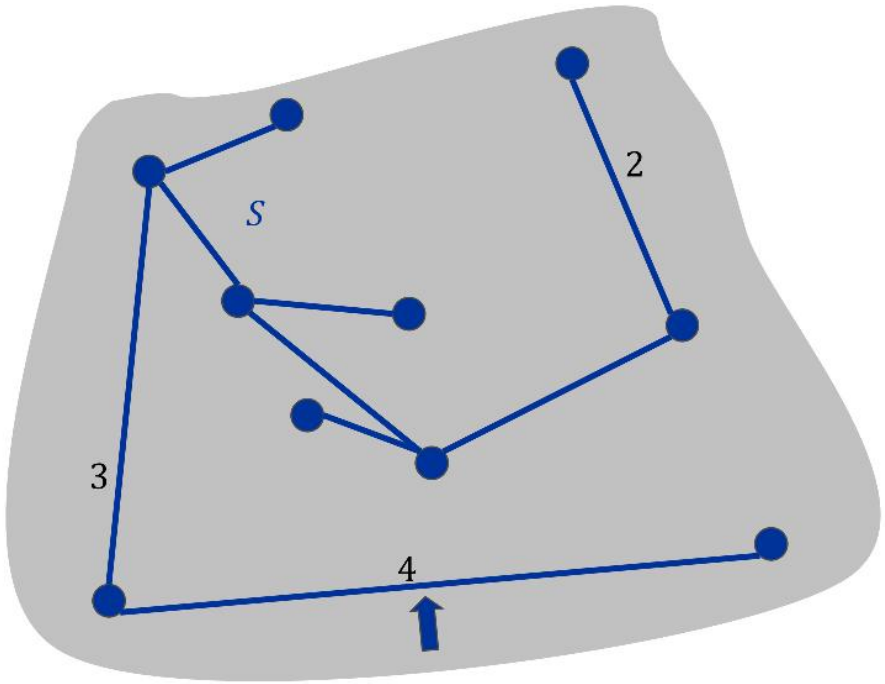
- **Prim's algorithm**
  - Initialize S = {any one node}
  - Add min cost edge e = (u, v) with u $\in$ S and v $\in$ V − S to T
  - Add v to S
  - Repeat until S = V



Prim's Algorithm: Idea

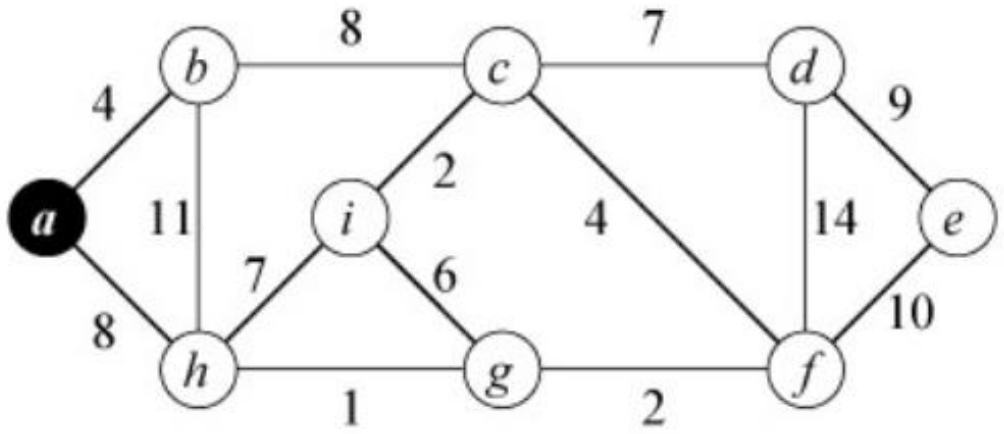
- **Prim's algorithm**
  - Initialize S = {any one node}
  - Add min cost edge e = (u, v) with u $\in$ S and v $\in$ V − S to T
  - Add v to S
  - Repeat until S = V

S = blue vertices
= V
form a tree

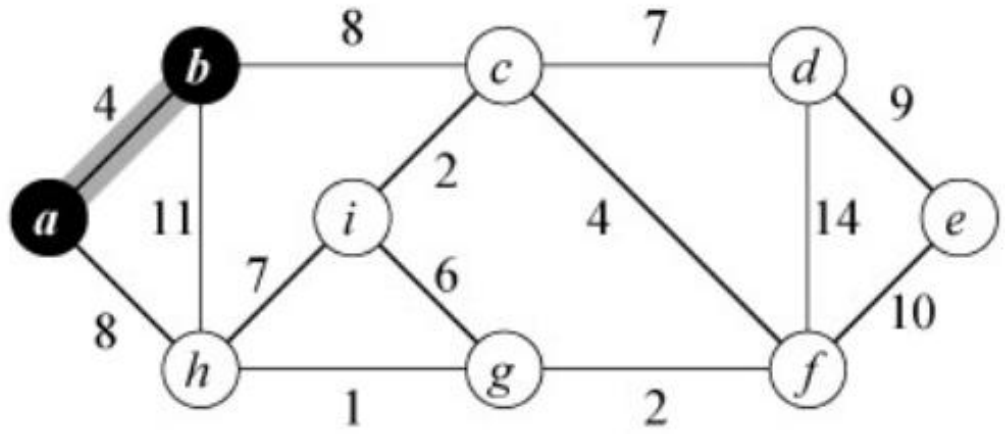


Prim's Algorithm: Example

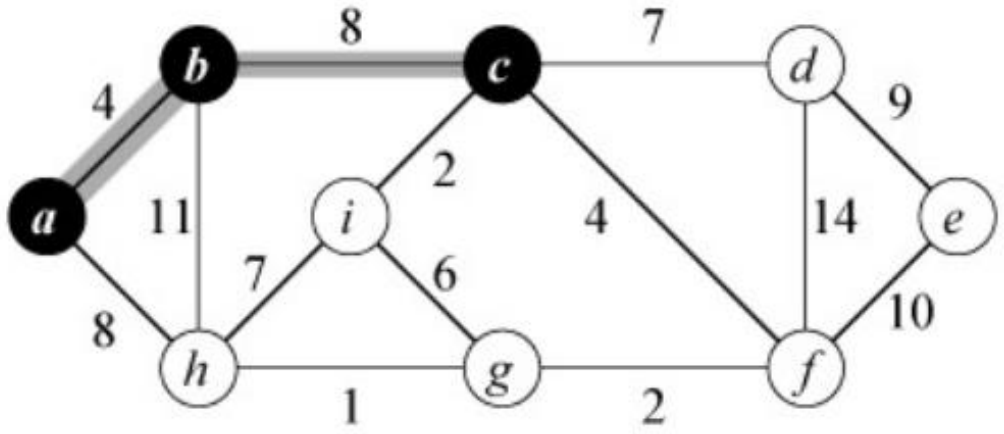
(a)



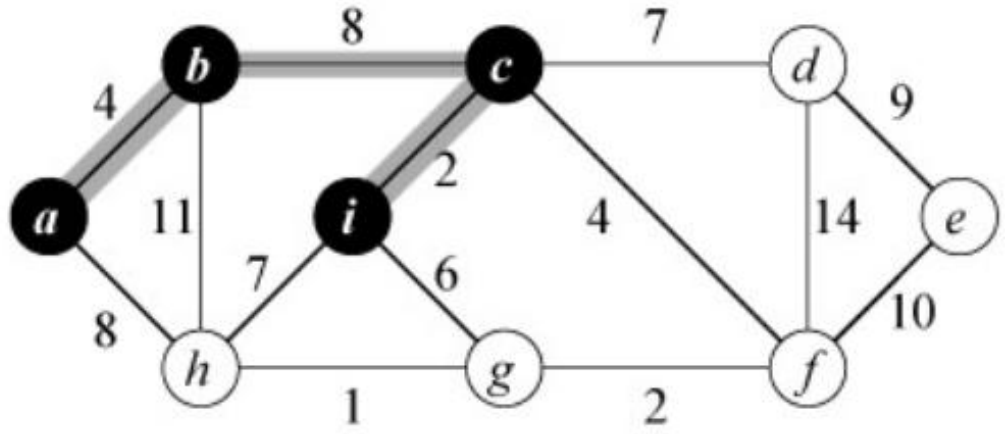
(b)



(c)

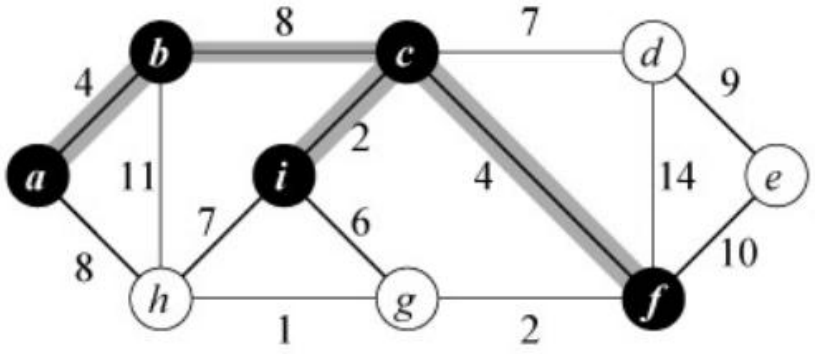


(d)

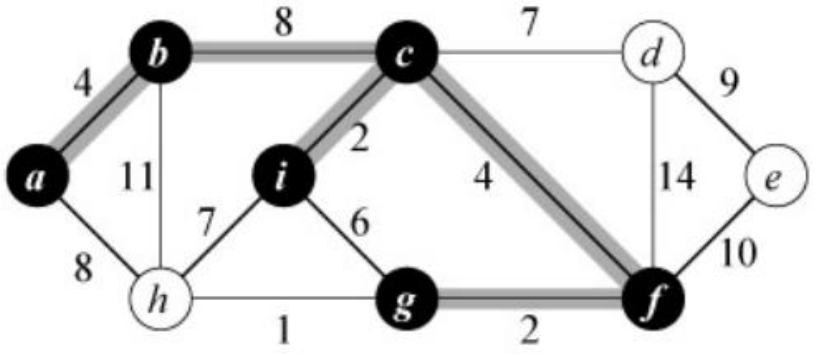


Prim's Algorithm: Example (continued)

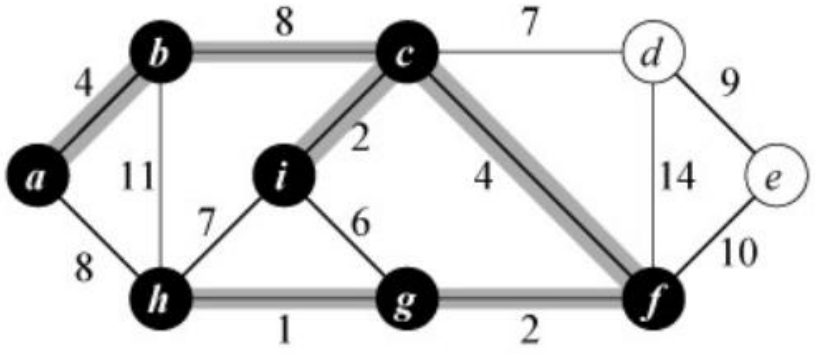
(e)



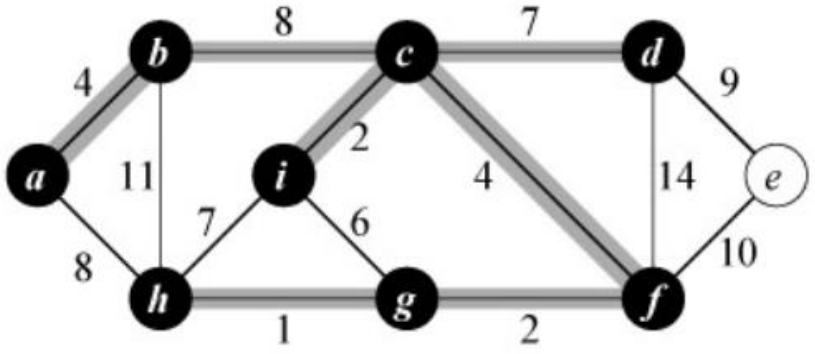
(f)



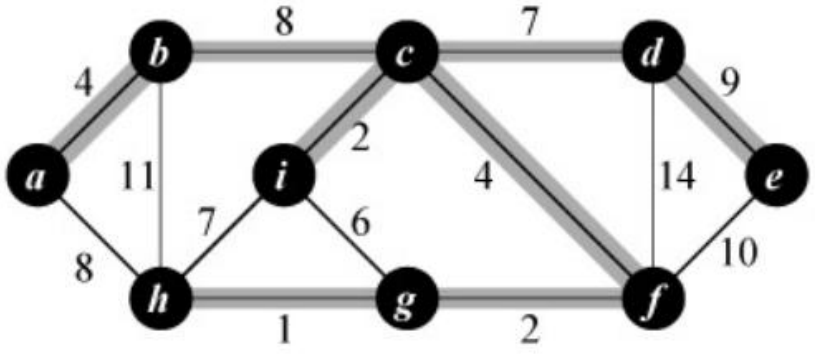
(g)



(h)

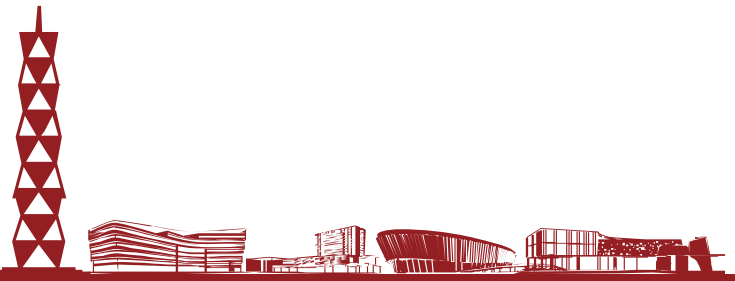


(i)



Minimum Spanning Trees

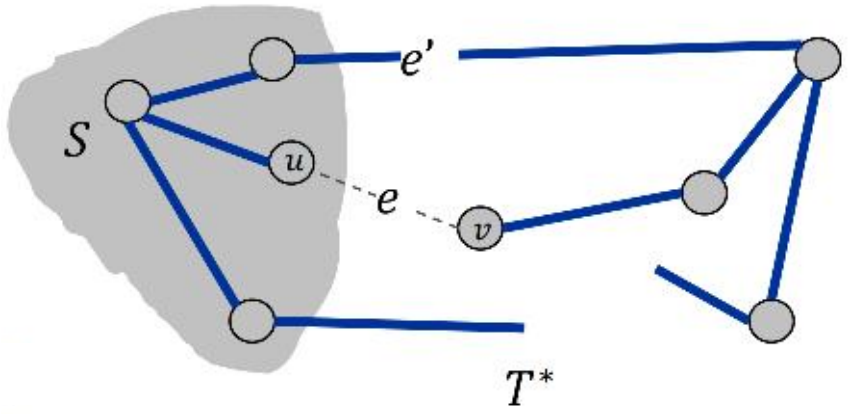
- Definition
- Prim's Algorithm
- **The Cut Lemma**
  - Correctness of Prim's Algorithm
  - Uniqueness of MSTs (under distinct weight assumption)
- Kruskal's Algorithm
  - Basic Idea
  - Correctness of Kruskal's Algorithm



Cut Lemma

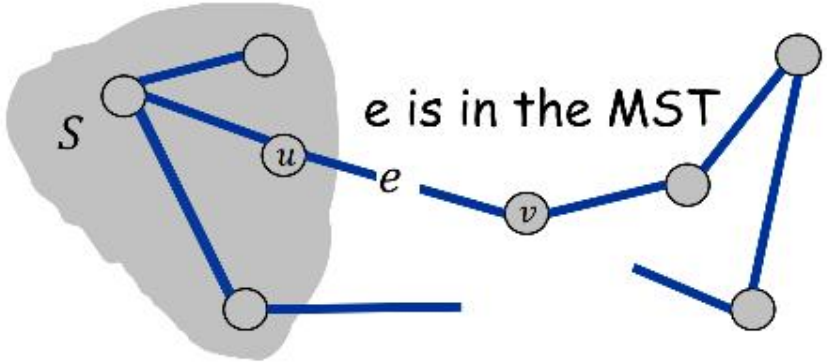
- Simplifying assumption. **All edge weights are distinct**

Cut lemma. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then every MST must contain e



Pf of Cut Lemma (exchange argument)

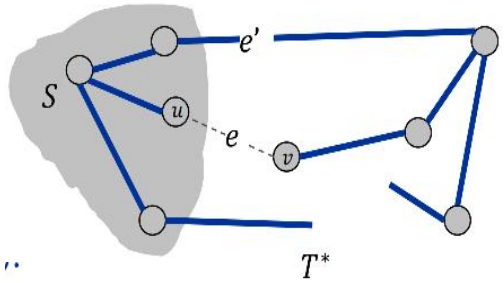
- Let T* be any MST
- Let e = (u, v) and suppose $e \notin T^*$
- There is a path in T* that connects u to v, **which must cross cut separating S from V – S** using some other edge $e' \in T^*$ with w(e') > w(e)
- If we replace e' in T* with e, then T* is still a spanning tree, but the total cost will be lowered, **contradicting fact that T* is an MST**

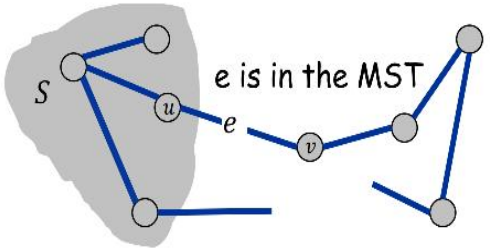


=> e is in every MST!

Cut Lemma: Deeper

Pf of Cut Lemma (exchange argument)





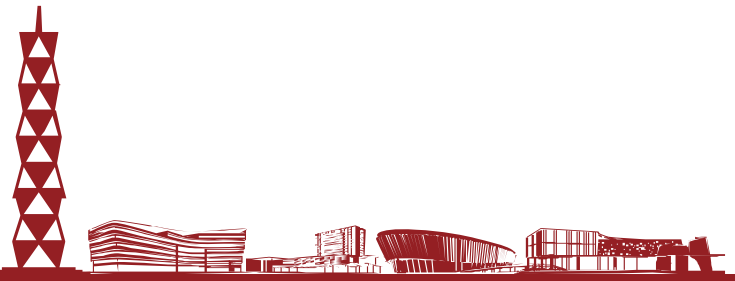
- Let T* be any MST
- Let e = (u, v) and suppose e ∉ T*
- There is a path in T* that connects u to v, **which must cross cut separating S from V – S** using some other edge e′ ∈ T* with w(e′) > w(e)
- If we replace e′ in T* with e, then T* is still a spanning tree, but the total cost will be lowered, **contradicting fact that T* is an MST**
- **Why is the above statement correct?**
- Adding e to T* creates a cycle that contains both e & the unique path in T* connecting u to v
- Removing ANY edge from the cycle will keep the graph connected (and keep it a tree)
- In particular, T* ∪ {e} – {e′}, is a tree!

=> w(T* ∪ {e} – {e′}) = w(T*) + w(e) – w(e′) < w(T*) contradicting fact that T* was supposed to be a MINIMUM Spanning Tree

Note: If there are edges with equal weights, then the MST may not be unique

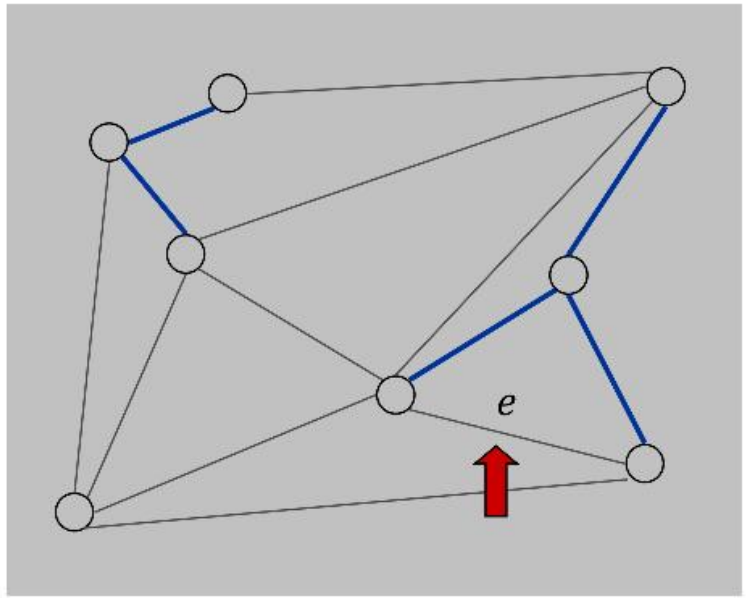
Minimum Spanning Trees

- Definition
- Prim's Algorithm
- The Cut Lemma
  - Correctness of Prim's Algorithm
  - Uniqueness of MSTs (under distinct weight assumption)
- **Kruskal's Algorithm**
  - **Basic Idea**
  - Correctness of Kruskal's Algorithm

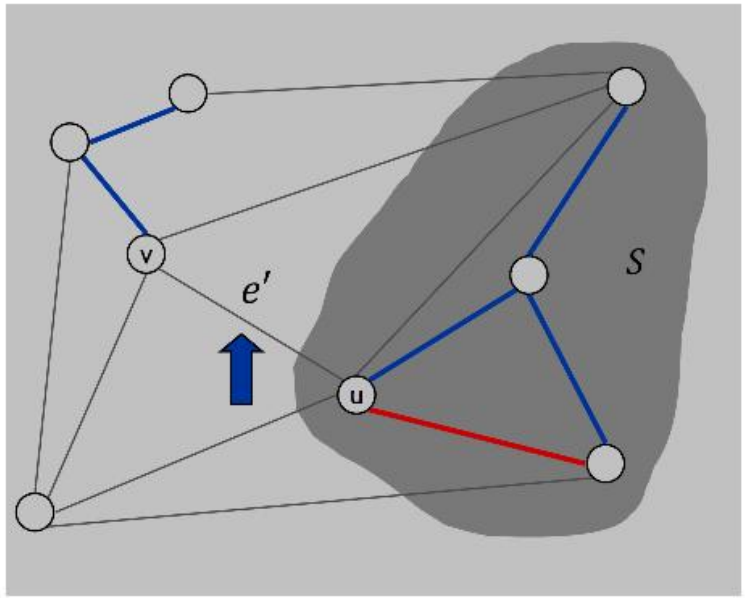


Kruskal's Algorithm: Idea

- **Kruskal's algorithm**
  - Starts with an empty tree T
  - Consider edges in increasing order of weight
  - Case 1: If adding e to T creates a cycle, discard e
  - Case 2: Otherwise, insert e = (u, v) into T according to cut lemma



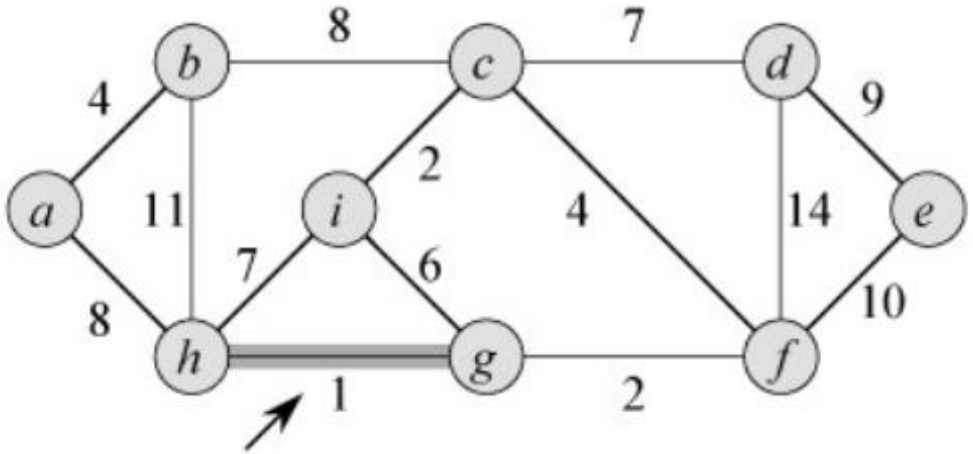
Case 1
e creates cycle.
Don't add e to T



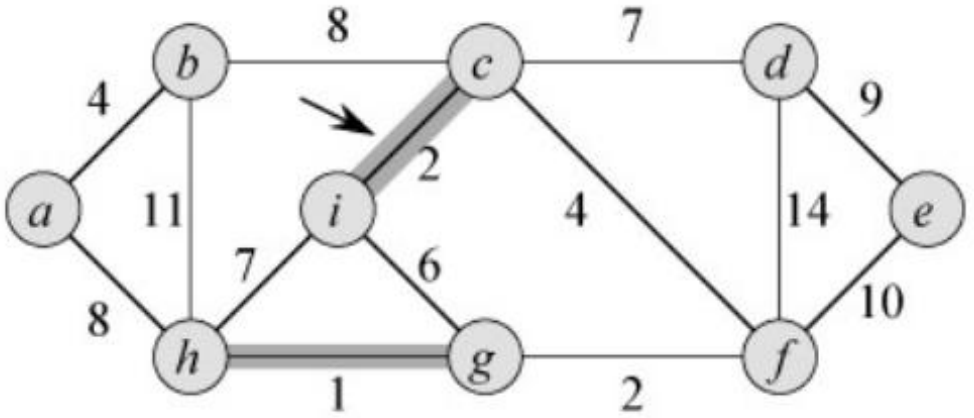
Case 2
e' doesn't create cycle
Add e' to T

Kruskal's Algorithm: Example

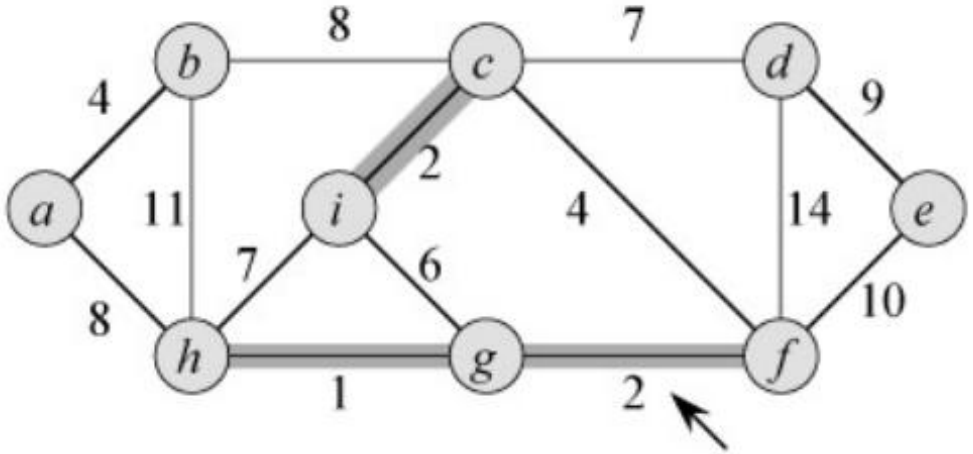
(a)



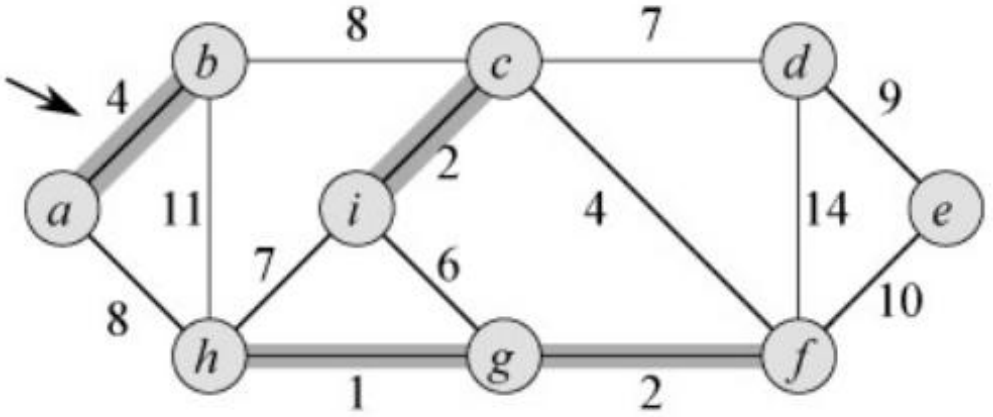
(b)



(c)

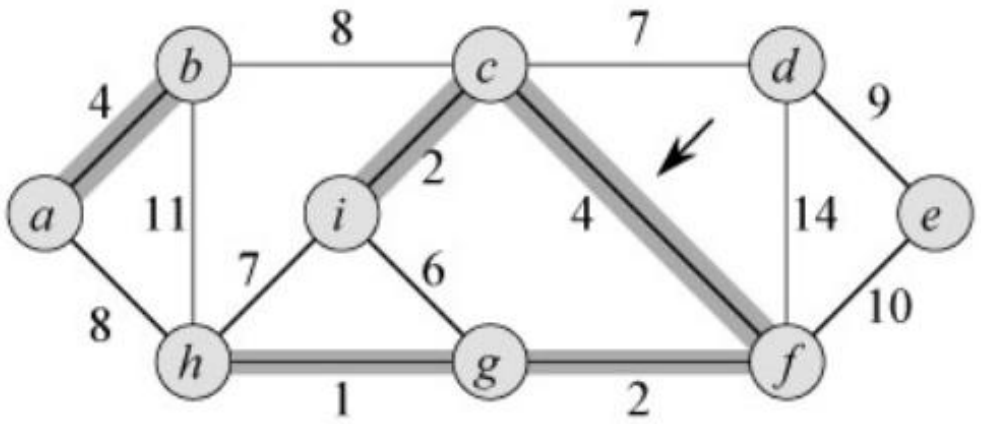


(d)

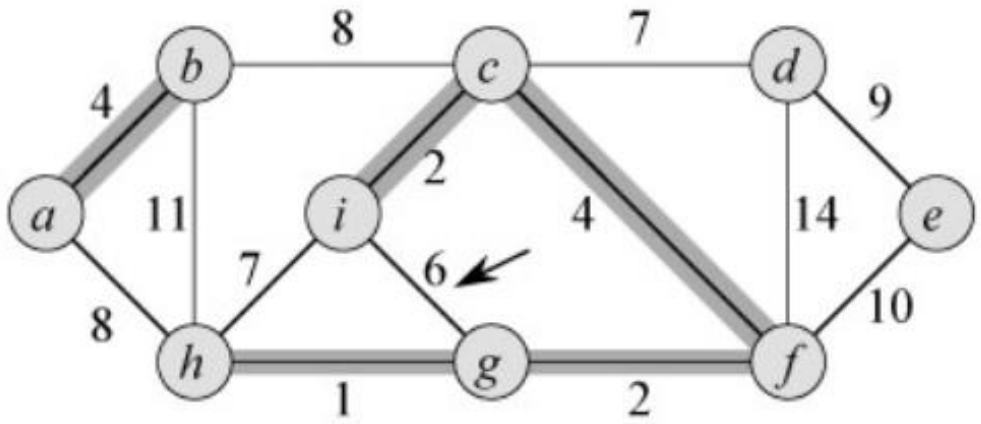


Kruskal's Algorithm: Example (continued)

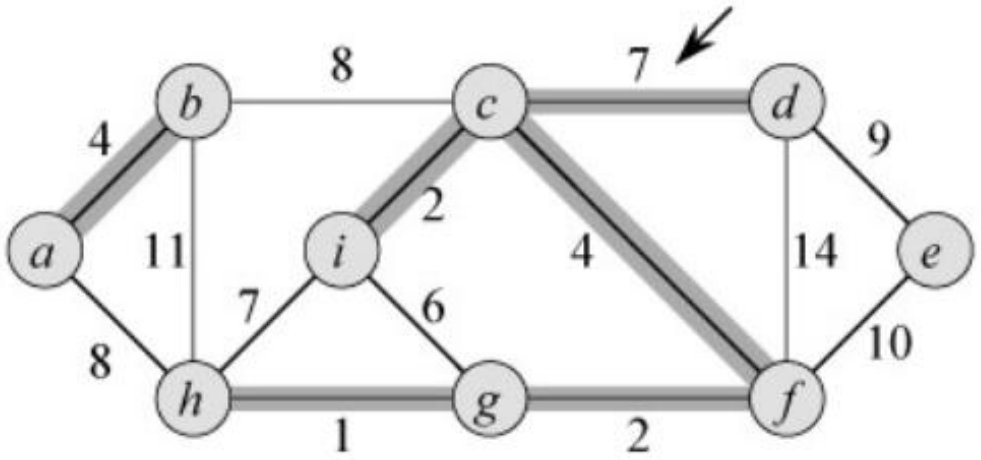
(e)



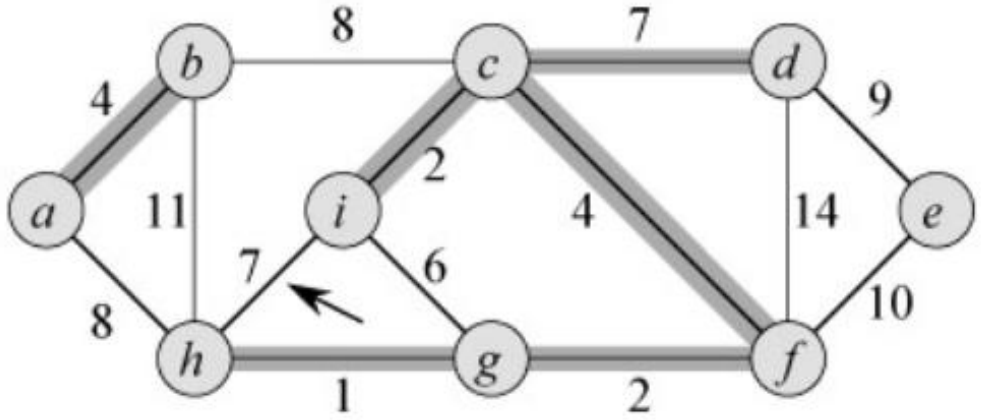
(f)



(g)

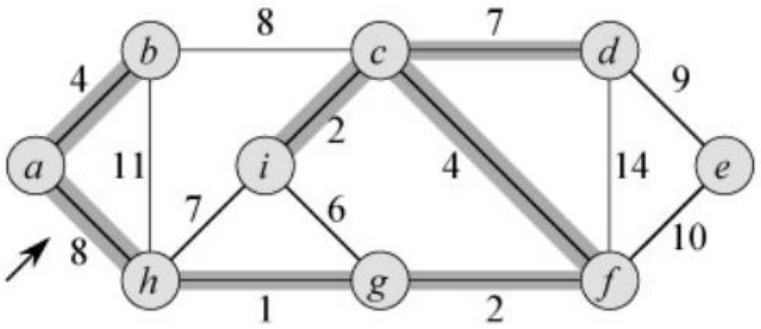


(h)

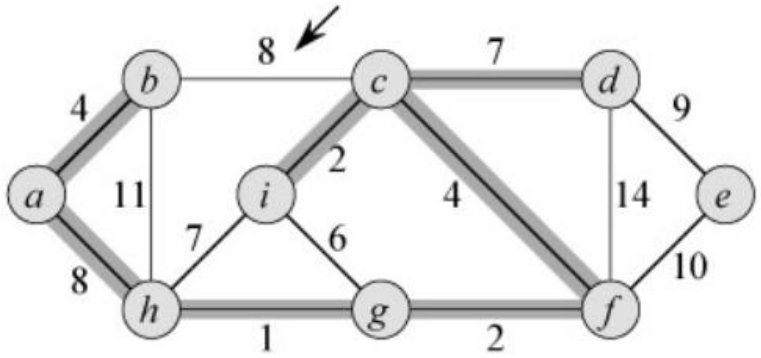


Kruskal's Algorithm: Example (continued)

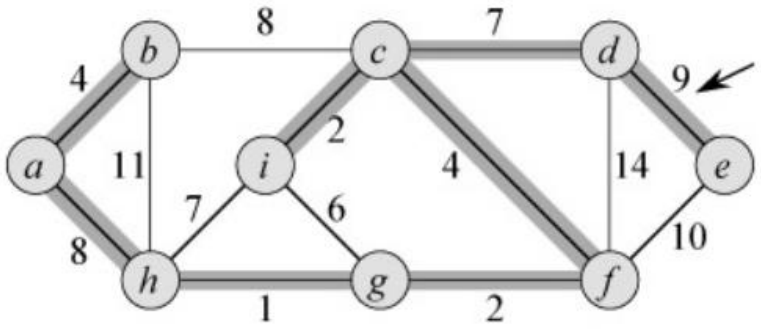
(i)



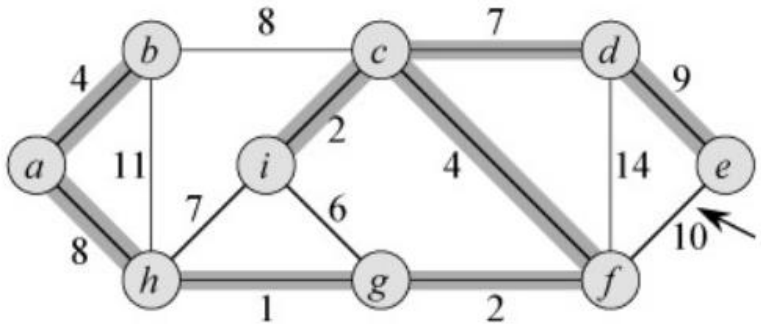
(j)



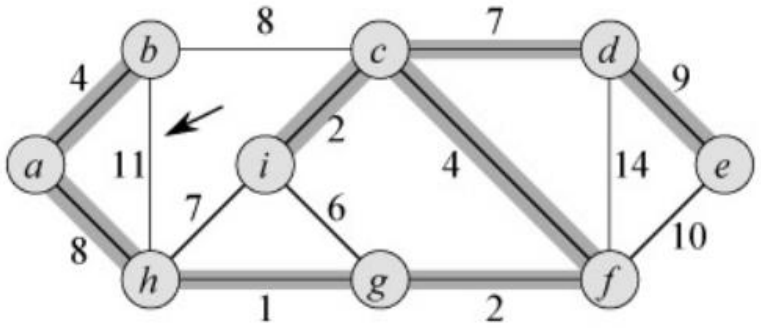
(k)



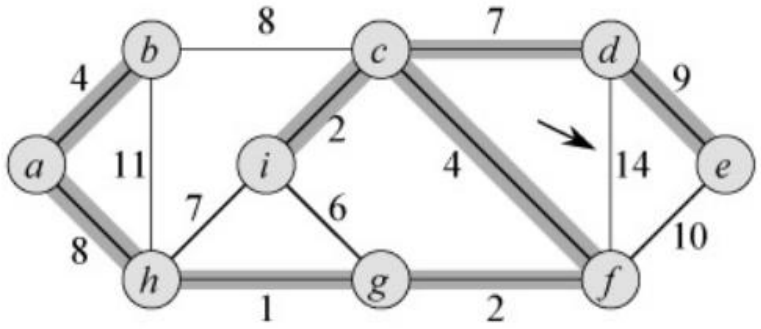
(l)



(m)

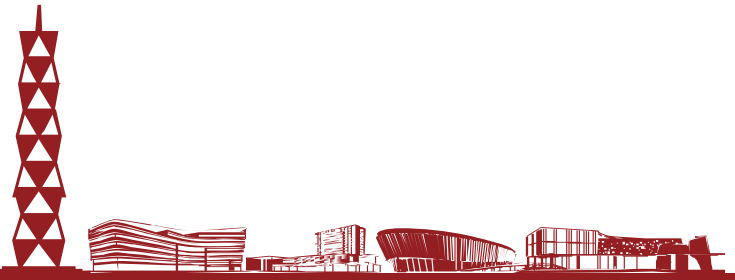


(n)



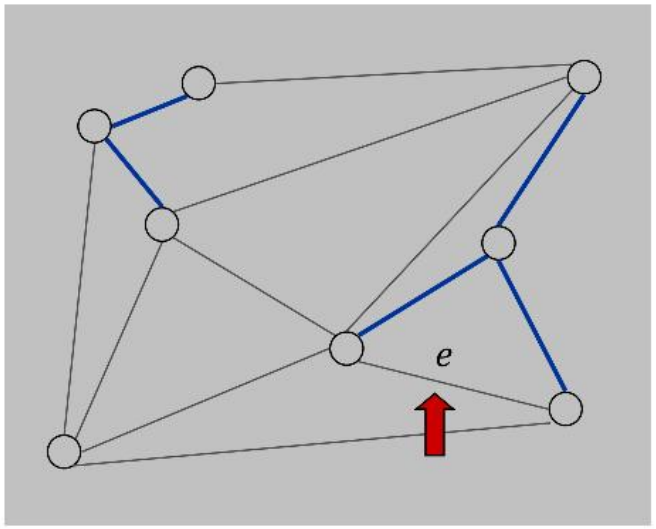
Minimum Spanning Trees

- Definition
- Prim's Algorithm
- The Cut Lemma
  - Correctness of Prim's Algorithm
  - Uniqueness of MSTs (under distinct weight assumption)
- **Kruskal's Algorithm**
  - Basic Idea
  - **Correctness of Kruskal's Algorithm**



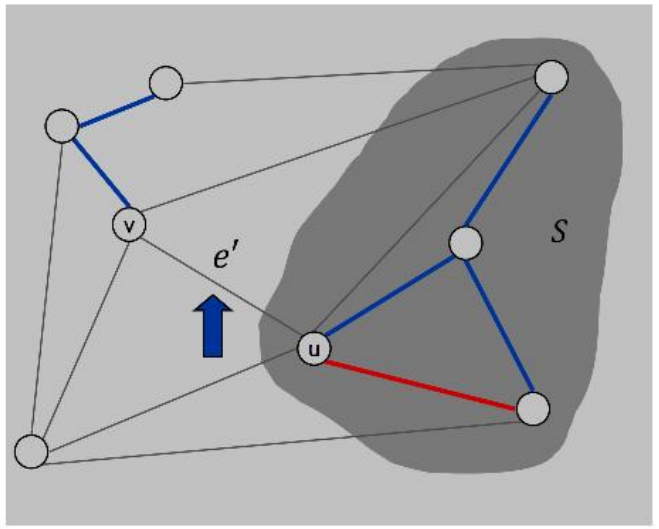
Proof of Correctness of Kruskal's Algorithm

- **Kruskal's algorithm**
  - Starts with an empty tree T, consider edges in increasing order of weight
  - Case 1: If adding e to T creates a cycle, discard e
  - Case 2: Otherwise, insert e = (u, v) into T according to cut lemma



Case 1

e creates cycle. Don't add e to T



Case 2

e` doesn't create cycle. Add e' to T

Simplifying assumption. All edge weights are distinct.

Given: Cut lemma. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then every MST must contain e

Proof of Correctness of Kruskal's Algorithm

- Three steps
  1. Will prove that Kruskal's algorithm always produces a tree,
     1. Will prove that it produces an acyclic graph
     2. Will prove that it produces a connected graph

     ⇒ Kruskal's algorithm produces a tree T

  2. Will prove that every edge chosen by Kruskal's algorithm is in every MST
  3. => tree T produced by Kruskal's algorithm is a MST

     This follows directly from (1) and (2).

     From (1), it produces a tree T; T has |v| - 1 edges

     From (2), every edge in T is in every MST

     But, every MST contains exactly |v| - 1 edges so every MST must be T

Uses same simplifying assumption as in the proof of Prim's algorithm that **all edges have different weights.** Also assumes that **input graph is connected**; otherwise result could never be a tree

Proof of Correctness of Kruskal's Algorithm

- Three steps

1. Will prove that Kruskal's algorithm always produces a tree,
   a) **Prove that it produces an acyclic graph**
   b) Prove that it produces a connected graph

   => Kruskal's algorithm produces a tree T

---

a) Prove that it produces an acyclic graph

Proof:
Kruskal's algorithm starts with T = ∅ (the empty set) and only adds edge e if T ∪ {e} is acyclic

⇒ T remains acyclic through the entire algorithm
⇒ Final T output by the algorithm is acyclic

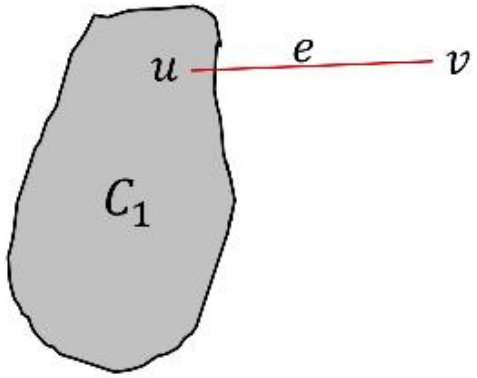
Proof of Correctness of Kruskal's Algorithm

- Three steps

1. Will prove that Kruskal's algorithm always produces a tree,
   a) Prove that it produces an acyclic graph
   b) Prove that it produces a connected graph
   => Kruskal's algorithm produces a tree T

---

a) Prove that it produces a connected graph



Suppose T is not connected.
Let C_1 be a connected component of T

$\Rightarrow$ There must exist some $u \in C_1$ with $e = (u, v)$ in G but $v \notin C_1$, because otherwise G is not connected
But adding e to T would not cause a cycle with any edges in T
=> Kruskal's algorithm would have added e to T, so $e \in T$
$\Rightarrow v \in C_1$
$\Rightarrow$ Contradiction => **T is connected**

Proof of Correctness of Kruskal's Algorithm

2. Show that every edge chosen by Kruskal's algorithm is in every MST

Given: Cut lemma. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then every MST must contain e.

Proof of (2): Let e = (u, v) be in T.

Plan: Will display S s.t. e is min-cost edge with exactly one endpoint in S.

---

a) Consider the set of edges in T right before e was added

Let S = C_u be the vertices in the connected component containing u at that time

b) Since e was added to T, it doesn't create a cycle, so v ∉ S.

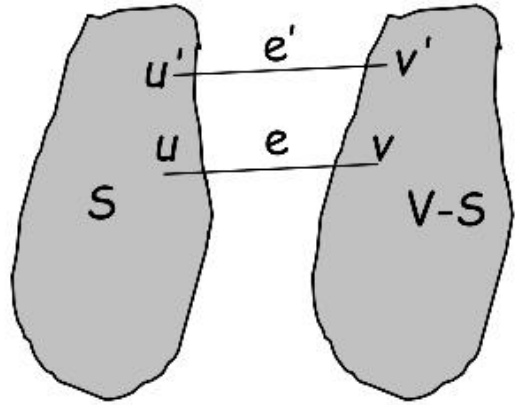
⇒ e has exactly one endpoint in S.

c) Suppose there exists e′ = (u′, v′) with w(e′) < w(e) s.t.

e′ also has exactly one endpoint in S, i.e., u′ ∈ S, v′ ∉ S.

e′ would have been processed by Kruskal's algorithm before e.

But e′ would not have caused a cycle, so e′ would have been added to T. But then u′, v′ would be in the SAME component at the time e was being processed so v′ ∈ S, contradicting v′ ∉ S.



=> e is min-cost edge with exactly one endpoint in S

Proof of Correctness of Kruskal's Algorithm

Three steps completed:

1) Proved that Kruskal's algorithm always produces a tree,

a) Proved that it produces an acyclic graph

b) Proved that it produces a connected graph

$\Rightarrow$ Kruskal's algorithm produces a tree T

2) Proved that every edge chosen by Kruskal's algorithm is in every MST

3) => tree T produced by Kruskal's algorithm is a MST

This follows directly from (1) and (2)

From (1), it produces a tree T; T has |V| - 1 edges

From (2), every edge in T is in every MST.

But, EVERY MST contains exactly |V| - 1 edges so every MST must be T

Next Time:
Greedy Algorithms (Cont.)
&
Divide and Conquer