

Discussion 5

CS180: Introduction to Algorithms and Complexity

Prof. Mark Burgin

TAs: Yunqi Guo guoyunqi@gmail.com

Ling Ding lingding@cs.ucla.edu

Some slides thanks to Kevin Wayne, ©2005 Pearson-Addison Wesley, used by permission of the publisher.

Outline

Graph

- Testing Bipartiteness (BFS)
- Connectivity in Directed Graphs
- DAGs and Topological Ordering

Greedy Algorithm

- MST

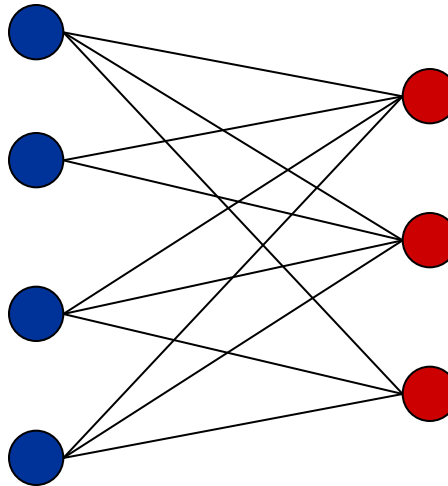
Testing Bipartiteness

Bipartite Graphs

Def. An undirected graph $G = (V, E)$ is **bipartite** if the nodes can be colored red or blue such that every edge has one red and one blue end.

Applications.

- Stable marriage: men = red, women = blue.
- Scheduling: machines = red, jobs = blue.

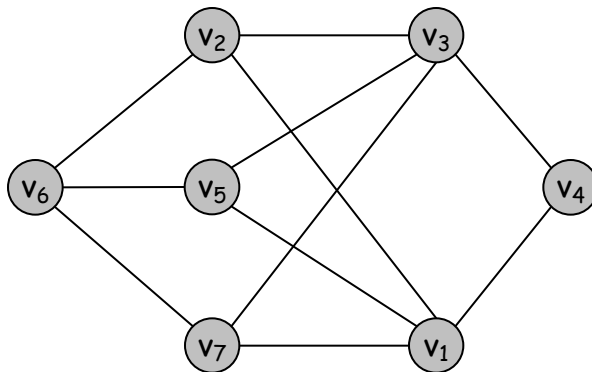


a bipartite graph

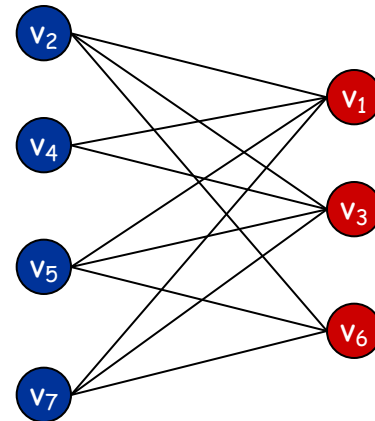
Testing Bipartiteness

Testing bipartiteness. Given a graph G , is it bipartite?

- Many graph problems become:
 - easier if the underlying graph is bipartite (matching)
 - tractable if the underlying graph is bipartite (independent set)
- Before attempting to design an algorithm, we need to understand structure of bipartite graphs.



a bipartite graph G

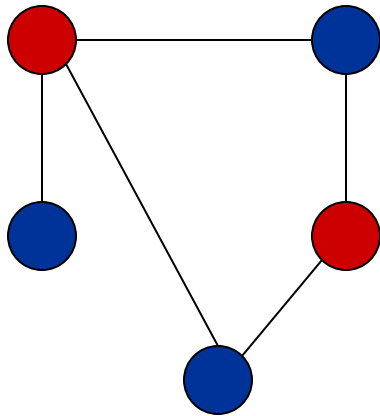


another drawing of G

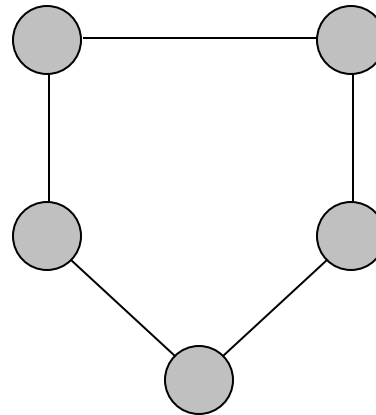
An Obstruction to Bipartiteness

Lemma. If a graph G is bipartite, it cannot contain an odd length cycle.

Pf. Not possible to 2-color the odd cycle, let alone G .



bipartite
(2-colorable)

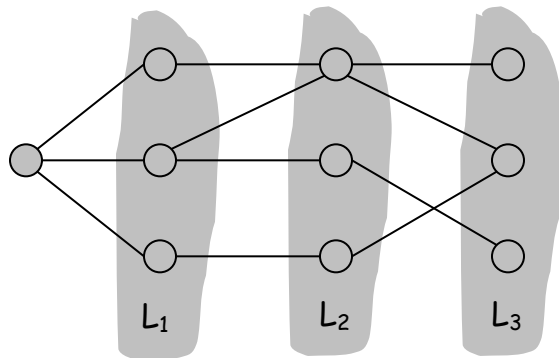


not bipartite
(not 2-colorable)

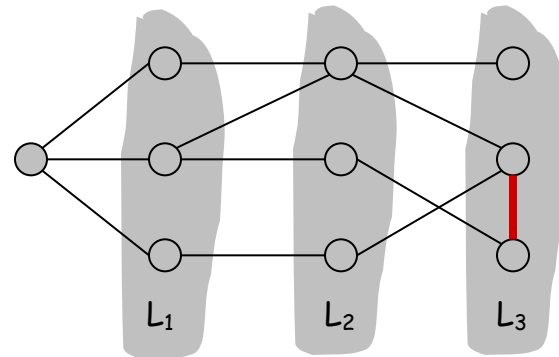
Bipartite Graphs

Lemma. Let G be a connected graph, and let L_0, \dots, L_k be the layers produced by BFS starting at node s . Exactly one of the following holds.

- (i) No edge of G joins two nodes of the same layer, and G is bipartite.
- (ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).



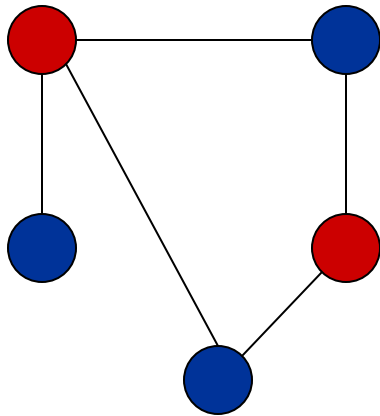
Case (i)



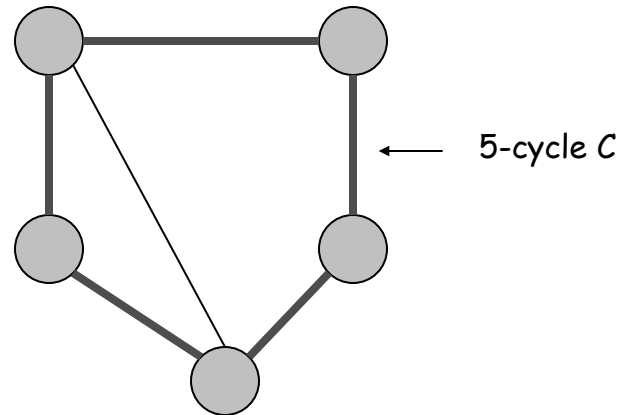
Case (ii)

Obstruction to Bipartiteness

Corollary. A graph G is bipartite iff it contains no odd length cycle.



bipartite
(2-colorable)



not bipartite
(not 2-colorable)

Related Questions

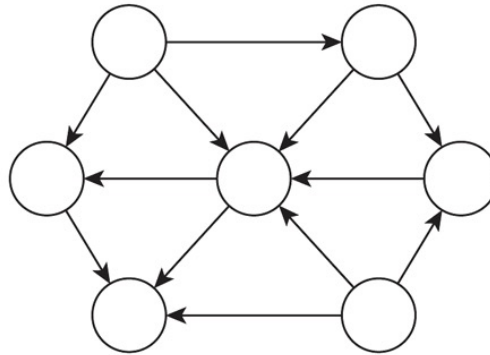
- <https://leetcode.com/problems/is-graph-bipartite/>

Connectivity in Directed Graphs

Directed Graphs

Directed graph. $G = (V, E)$

- Edge (u, v) goes from node u to node v .



Ex. Web graph - hyperlink points from one web page to another.

- Directedness of graph is crucial.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.

Graph Search

Directed reachability. Given a node s , find all nodes reachable from s .

Directed s - t shortest path problem. Given two node s and t , what is the length of the shortest path between s and t ?

Graph search. BFS extends naturally to directed graphs.

Web crawler. Start from web page s . Find all web pages linked from s , either directly or indirectly.

Strong Connectivity

Def. Node u and v are **mutually reachable** if there is a path from u to v and also a path from v to u .

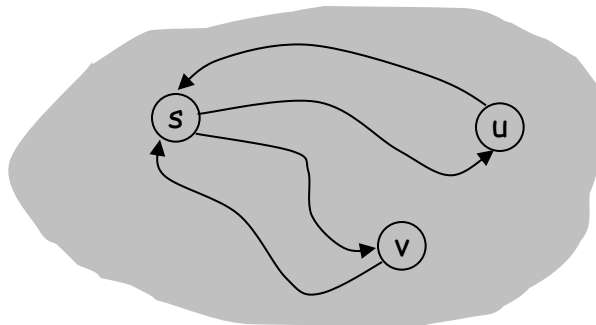
Def. A graph is **strongly connected** if every pair of nodes is mutually reachable.

Lemma. Let s be any node. G is strongly connected iff every node is reachable from s , and s is reachable from every node.

Pf. \Rightarrow Follows from definition.

Pf. \Leftarrow Path from u to v : concatenate u - s path with s - v path.

Path from v to u : concatenate v - s path with s - u path. ■

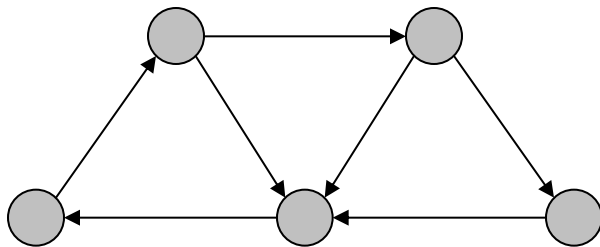


↖
ok if paths overlap

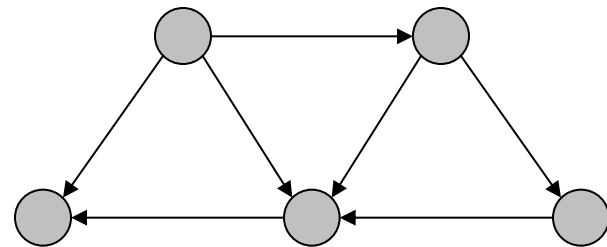
Strong Connectivity: Algorithm

Theorem. Can determine if G is strongly connected in $O(m + n)$ time.
Pf.

- Pick any node s .
- Run BFS from s in G .
- Run BFS from s in G^{rev} . ← reverse orientation of every edge in G
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma. ▪



strongly connected



not strongly connected

Related Questions

- <https://leetcode.com/problems/number-of-operations-to-make-network-connected/>
- <https://leetcode.com/problems/word-ladder-ii/>

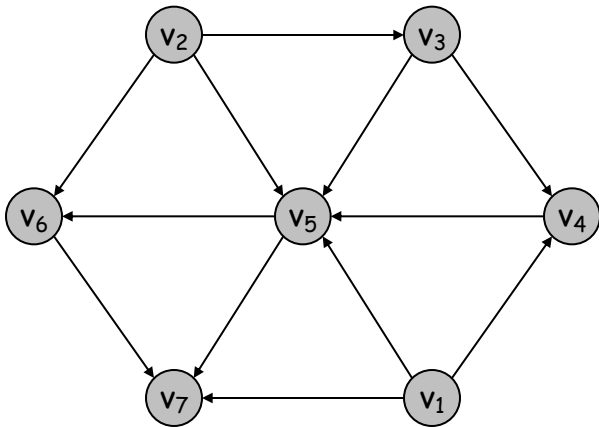
DAGs and Topological Ordering

Directed Acyclic Graphs

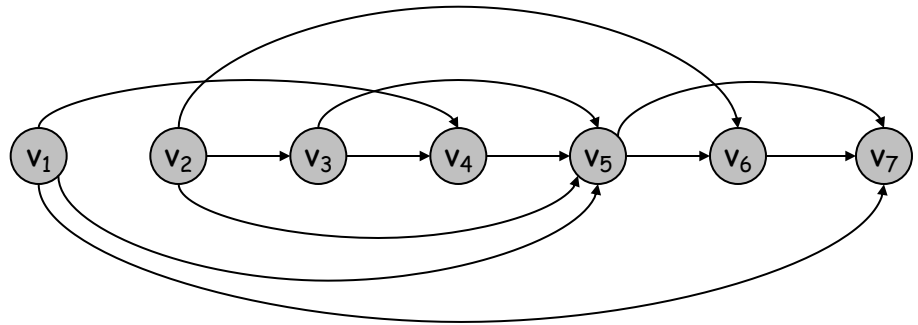
Def. An **DAG** is a directed graph that contains no directed cycles.

Ex. Precedence constraints: edge (v_i, v_j) means v_i must precede v_j .

Def. A **topological order** of a directed graph $G = (V, E)$ is an ordering of its nodes as v_1, v_2, \dots, v_n so that for every edge (v_i, v_j) we have $i < j$.



a DAG



a topological ordering

Precedence Constraints

Precedence constraints. Edge (v_i, v_j) means task v_i must occur before v_j .

Applications.

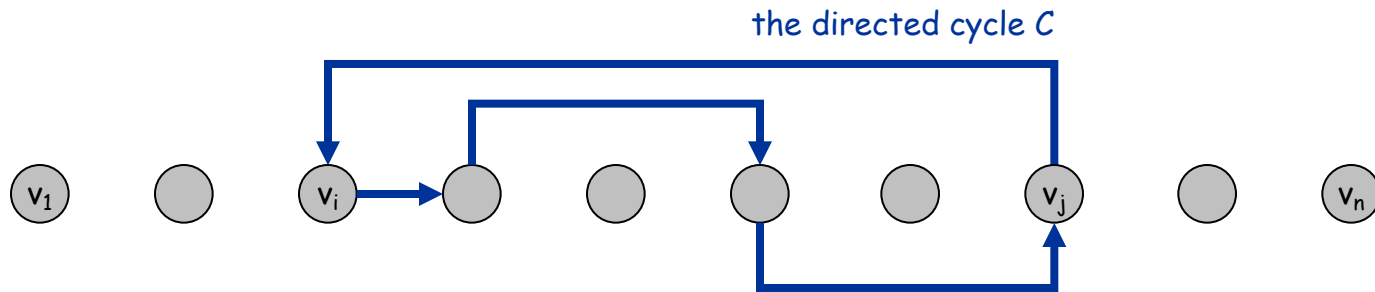
- Course prerequisite graph: course v_i must be taken before v_j .
- Compilation: module v_i must be compiled before v_j . Pipeline of computing jobs: output of job v_i needed to determine input of job v_j .

Directed Acyclic Graphs

Lemma. If G has a topological order, then G is a DAG.

Pf. (by contradiction)

- Suppose that G has a topological order v_1, \dots, v_n and that G also has a directed cycle C . Let's see what happens.
- Let v_i be the lowest-indexed node in C , and let v_j be the node just before v_i ; thus (v_j, v_i) is an edge.
- By our choice of i , we have $i < j$.
- On the other hand, since (v_j, v_i) is an edge and v_1, \dots, v_n is a topological order, we must have $j < i$, a contradiction. ■



the supposed topological order: v_1, \dots, v_n

Directed Acyclic Graphs

Lemma. If G has a topological order, then G is a DAG.

Q. Does every DAG have a topological ordering?

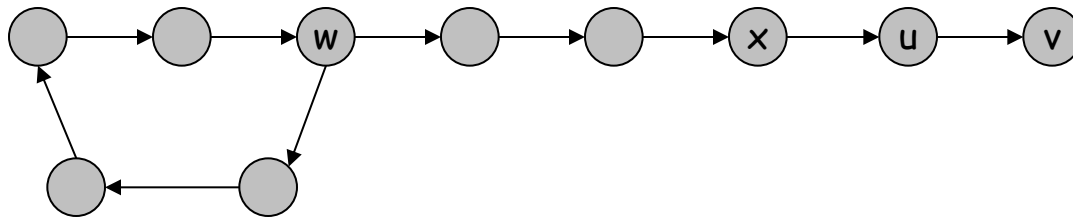
Q. If so, how do we compute one?

Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a node with no incoming edges.

Pf. (by contradiction)

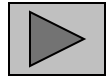
- Suppose that G is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node v , and begin following edges backward from v . Since v has at least one incoming edge (u, v) we can walk backward to u .
- Then, since u has at least one incoming edge (x, u) , we can walk backward to x .
- Repeat until we visit a node, say w , twice.
- Let C denote the sequence of nodes encountered between successive visits to w . C is a cycle. ▪



Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a topological ordering.

Pf. (by induction on n)



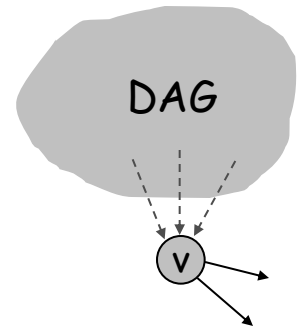
- Base case: true if $n = 1$.
- Given DAG on $n > 1$ nodes, find a node v with no incoming edges.
- $G - \{v\}$ is a DAG, since deleting v cannot create cycles.
- By inductive hypothesis, $G - \{v\}$ has a topological ordering.
- Place v first in topological ordering; then append nodes of $G - \{v\}$
- in topological order. This is valid since v has no incoming edges. ▪

To compute a topological ordering of G :

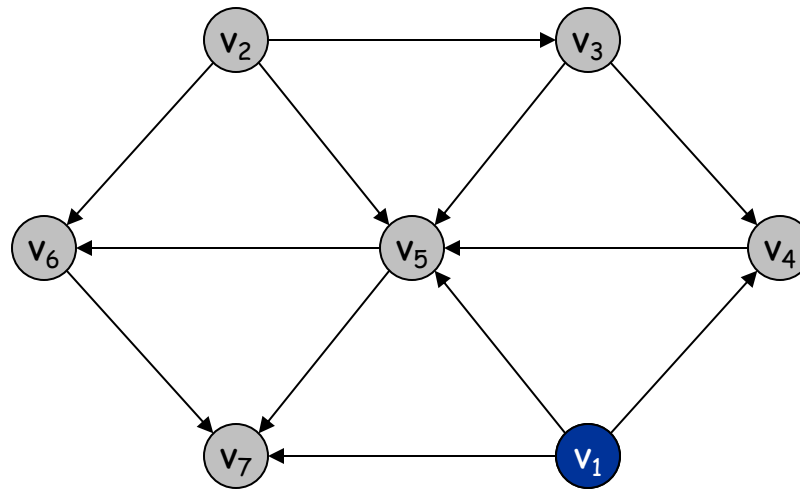
Find a node v with no incoming edges and order it first

Delete v from G

Recursively compute a topological ordering of $G - \{v\}$
and append this order after v

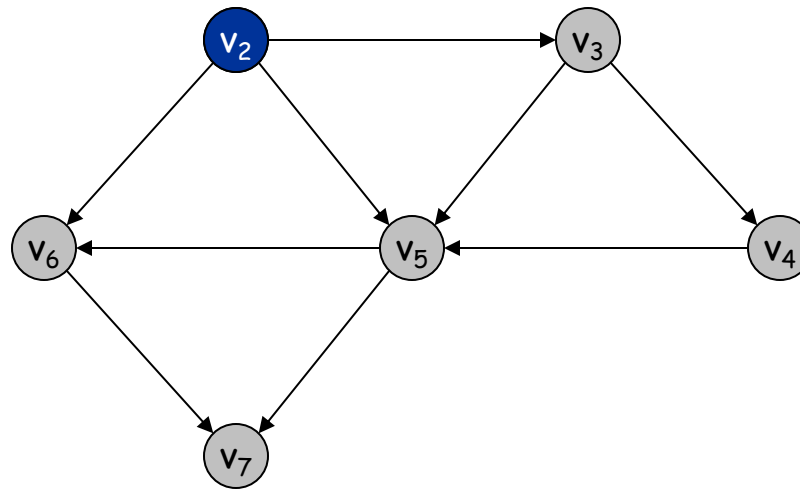


Topological Ordering Algorithm: Example



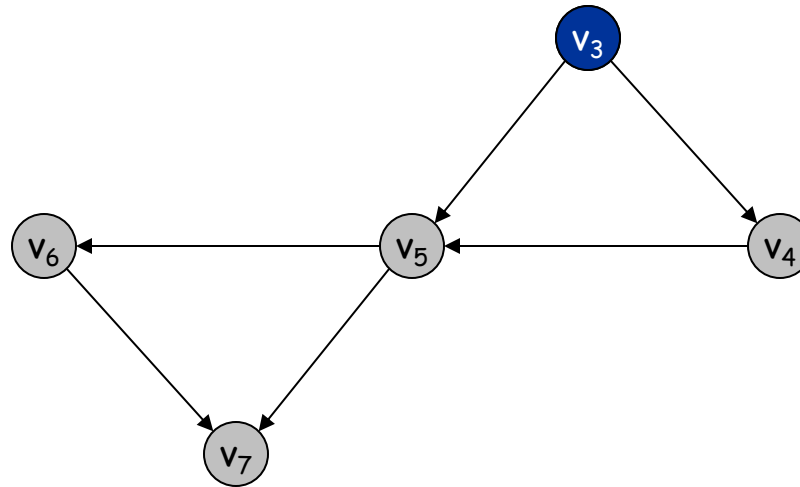
Topological order:

Topological Ordering Algorithm: Example



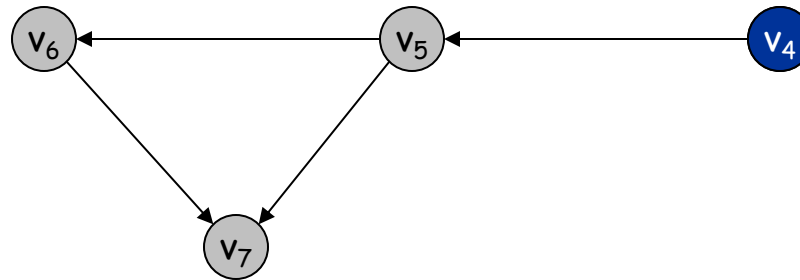
Topological order: v_1

Topological Ordering Algorithm: Example



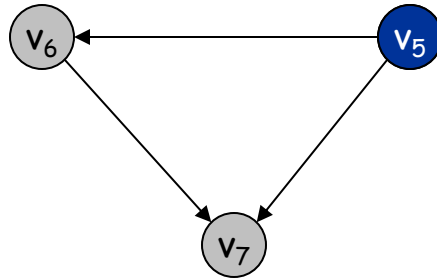
Topological order: v_1, v_2

Topological Ordering Algorithm: Example



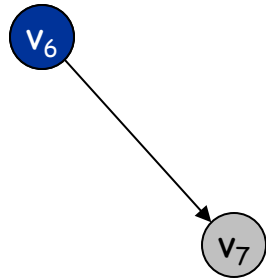
Topological order: v_1, v_2, v_3

Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

Topological Ordering Algorithm: Example



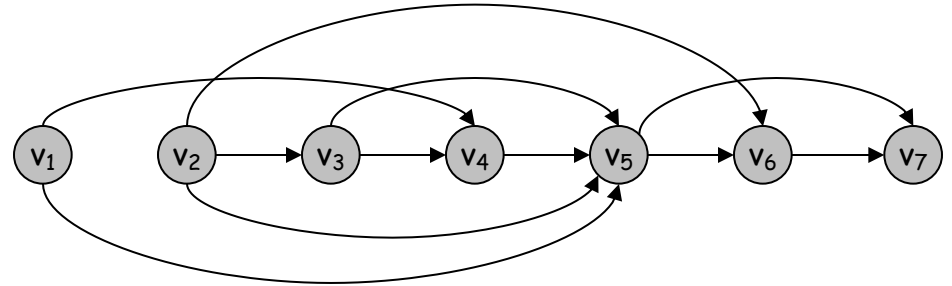
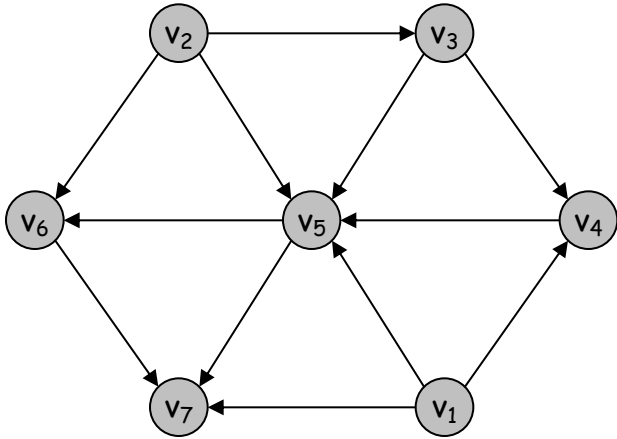
Topological order: v_1, v_2, v_3, v_4, v_5

Topological Ordering Algorithm: Example



Topological order: $v_1, v_2, v_3, v_4, v_5, v_6$

Topological Ordering Algorithm: Example



Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$.

Topological Sorting Algorithm: Running Time

Theorem. Algorithm finds a topological order in $O(m + n)$ time.

Pf.

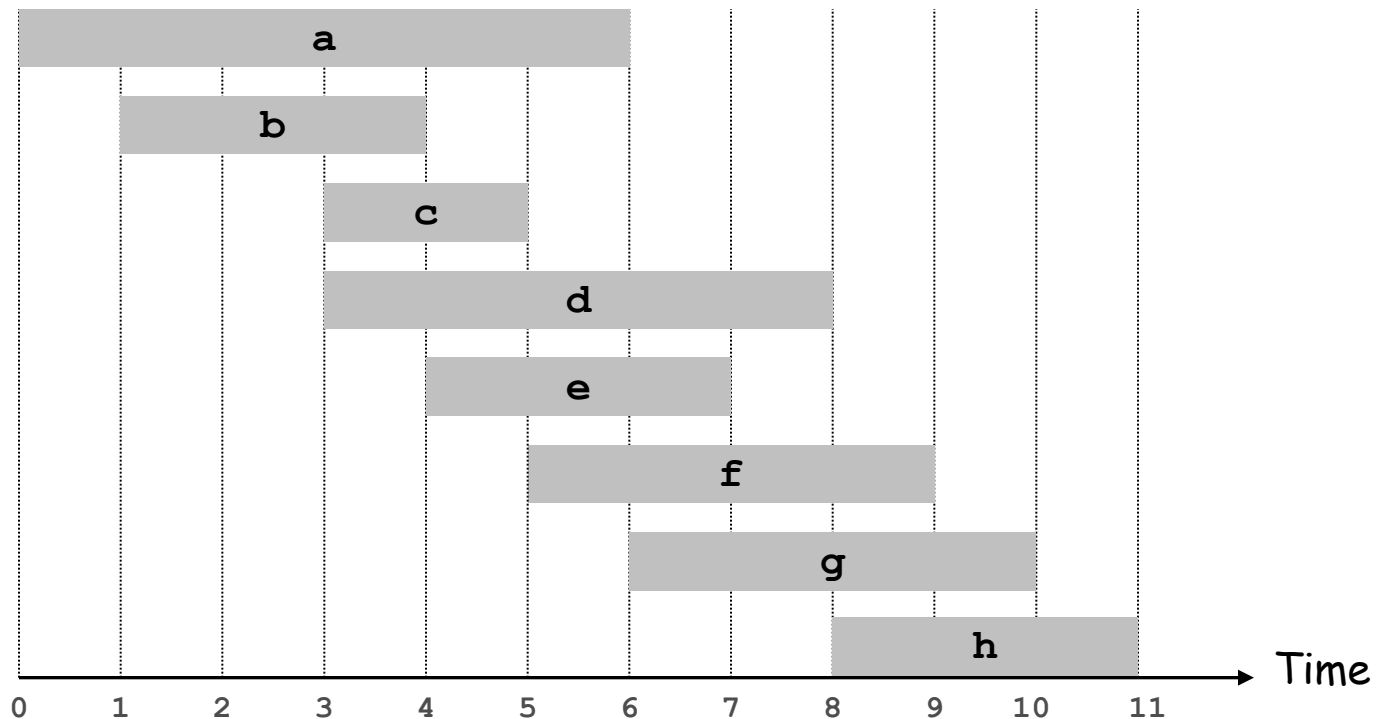
- Maintain the following information:
 - `count[w]` = remaining number of incoming edges
 - S = set of remaining nodes with no incoming edges
- Initialization: $O(m + n)$ via single scan through graph.
- Update: to delete v
 - remove v from S
 - decrement `count[w]` for all edges from v to w , and add w to S if `count[w]` hits 0
 - this is $O(1)$ per edge ▪

Greedy algorithm

Interval Scheduling

Interval scheduling.

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



Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some natural order.

Take each job provided it's compatible with the ones already taken.

- [Earliest start time] Consider jobs in ascending order of s_j .
- [Earliest finish time] Consider jobs in ascending order of f_j .
- [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.
- [Fewest conflicts] For each job j , count the number of conflicting jobs c_j . Schedule in ascending order of c_j .

Interval Scheduling: Greedy Algorithms

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



counterexample for earliest start time



counterexample for shortest interval

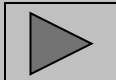


counterexample for fewest conflicts

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.

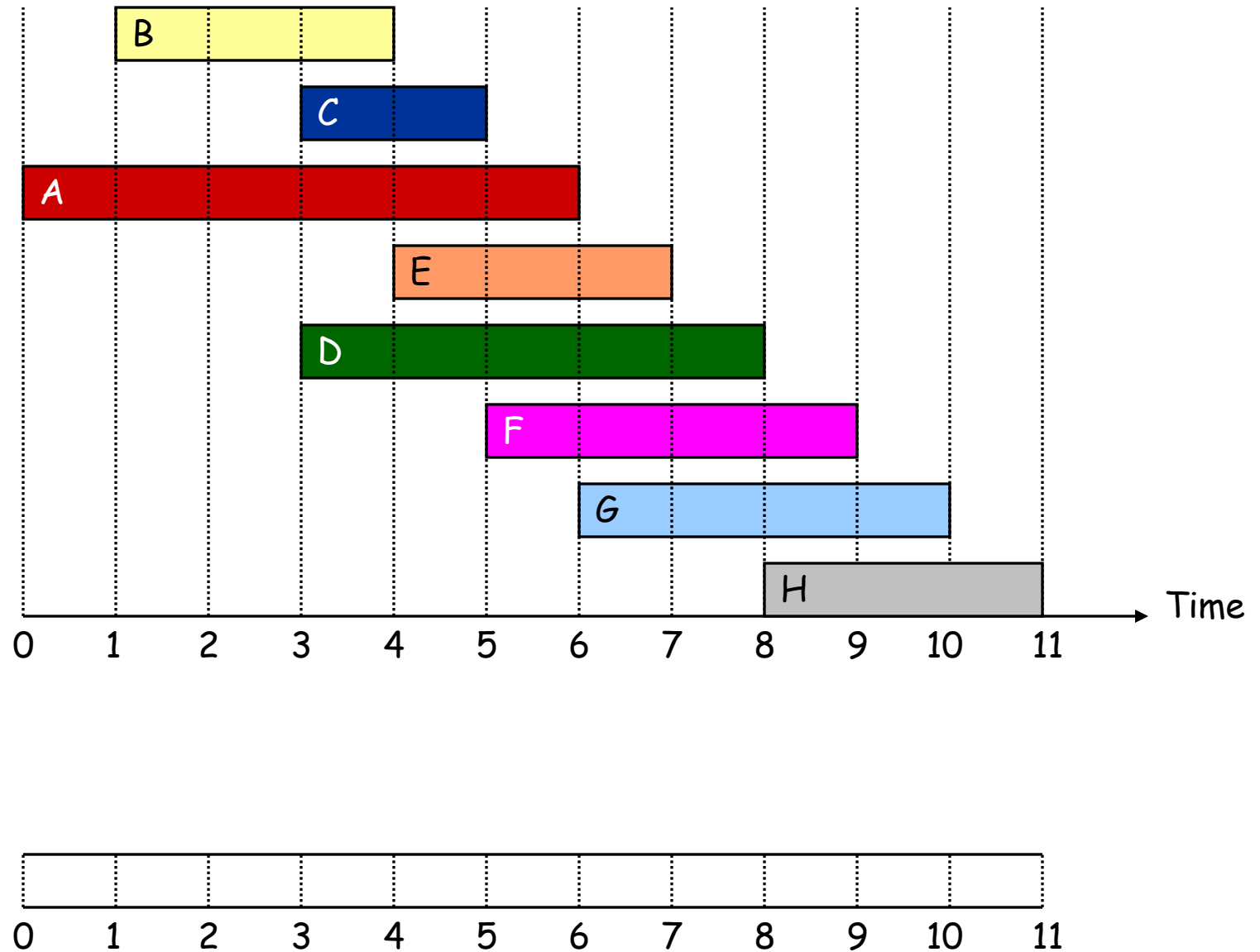
```
Sort jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .  
    ↙ set of jobs selected  
A ←  $\phi$   
for j = 1 to n {  
    if (job j compatible with A)  
        A ← A  $\cup$  {j}  
}  
return A
```



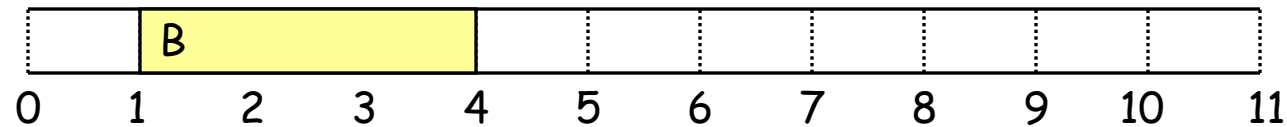
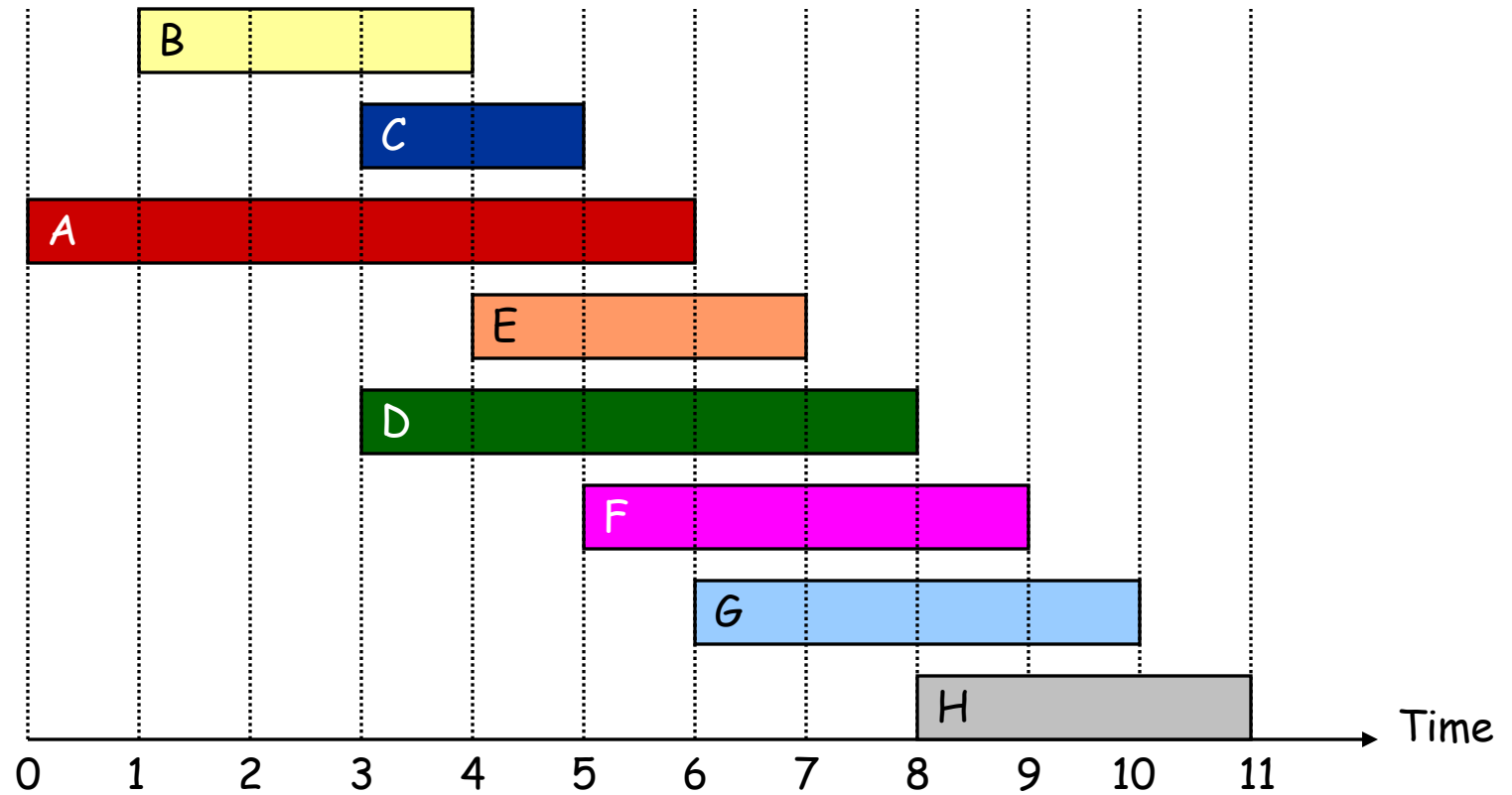
Implementation. $O(n \log n)$.

- Remember job j^* that was added last to A.
- Job j is compatible with A if $s_j \geq f_{j^*}$.

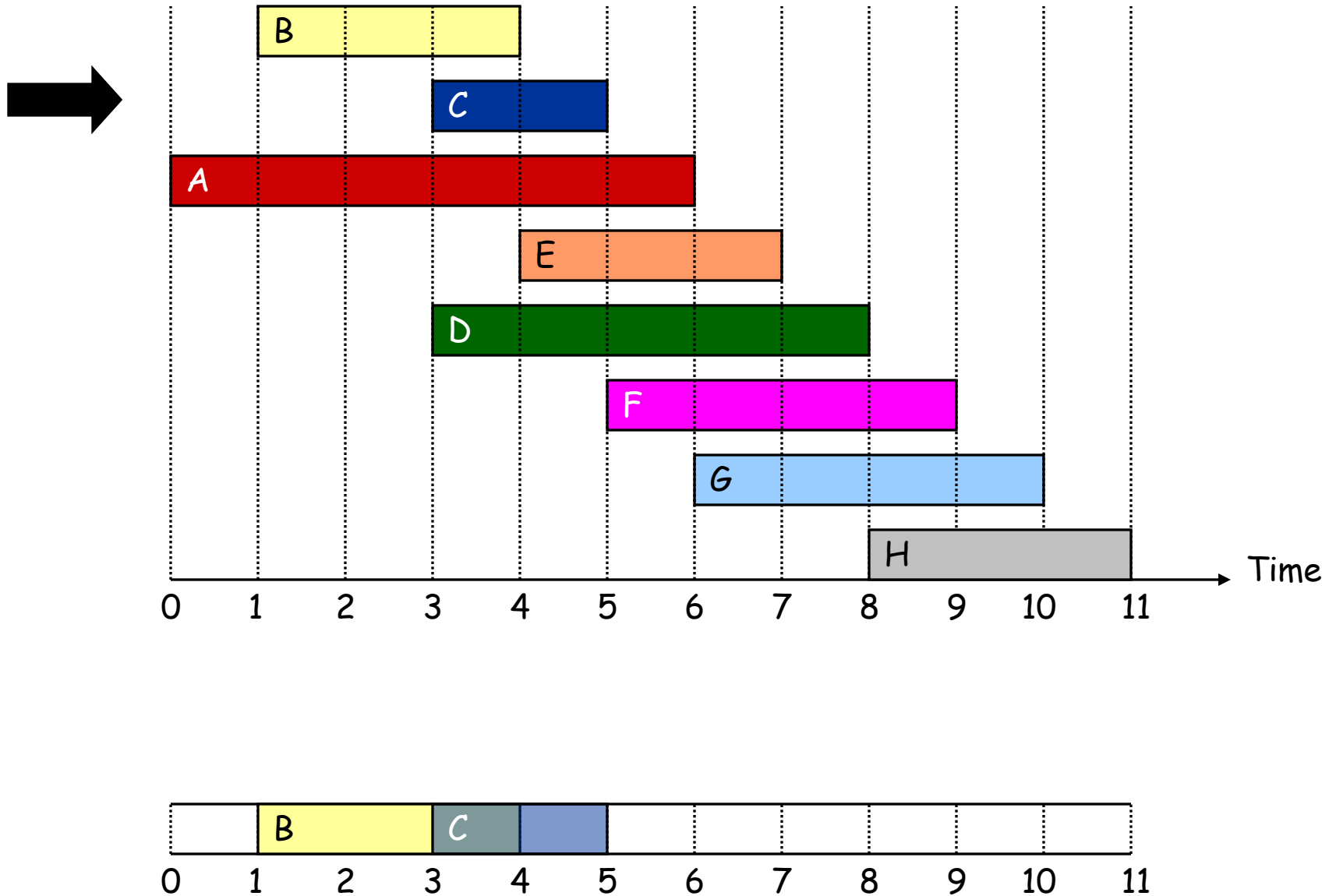
Interval Scheduling



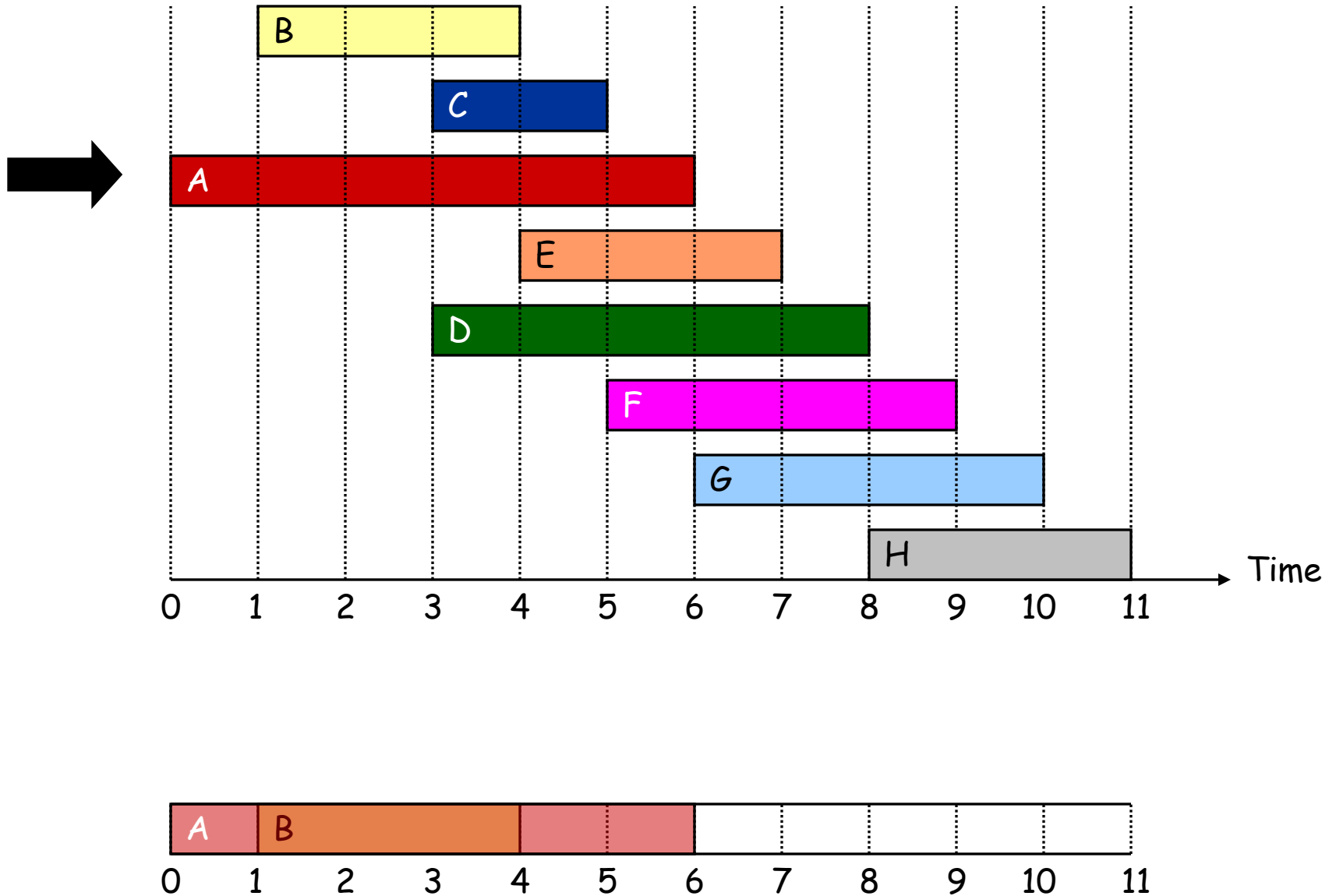
Interval Scheduling



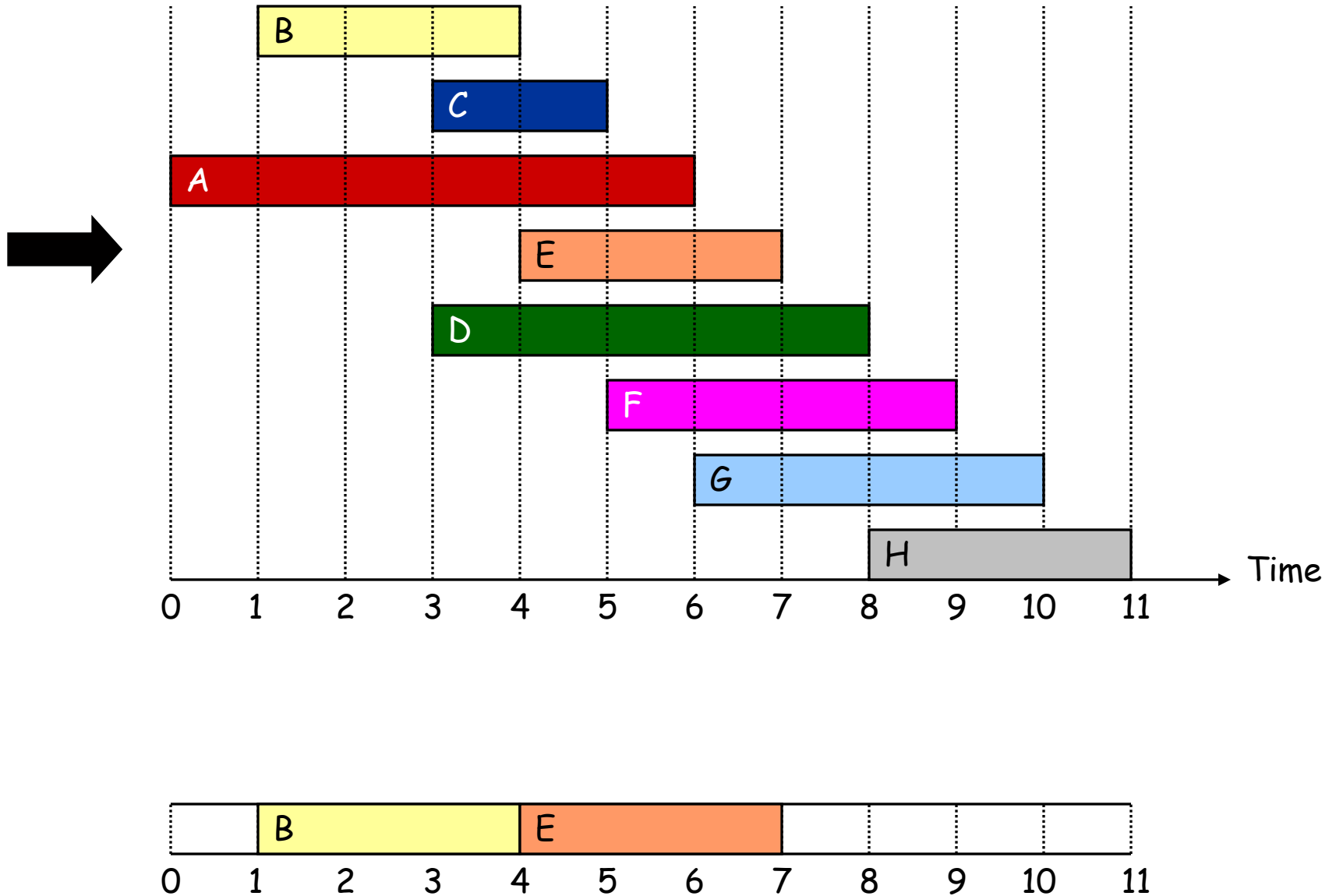
Interval Scheduling



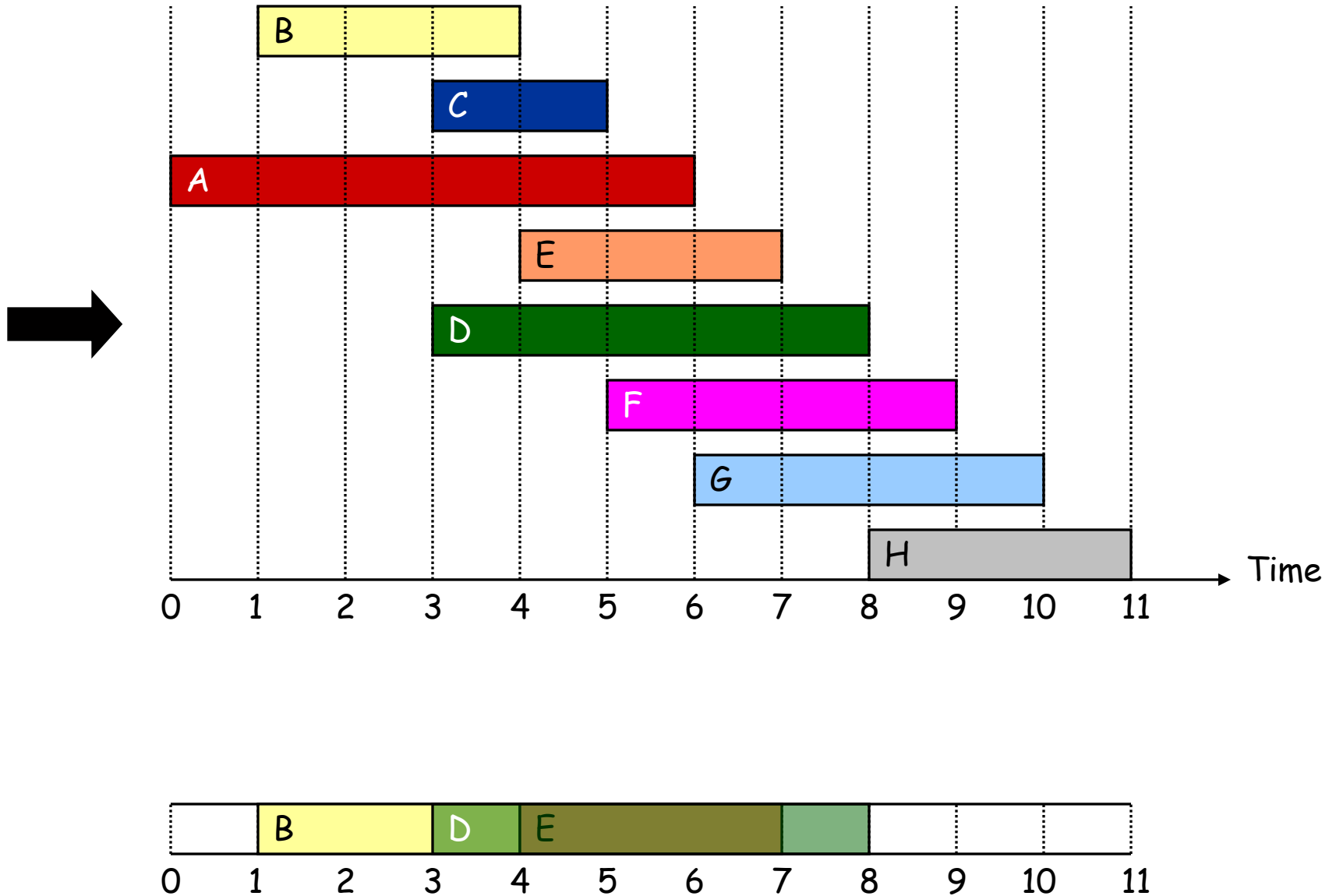
Interval Scheduling



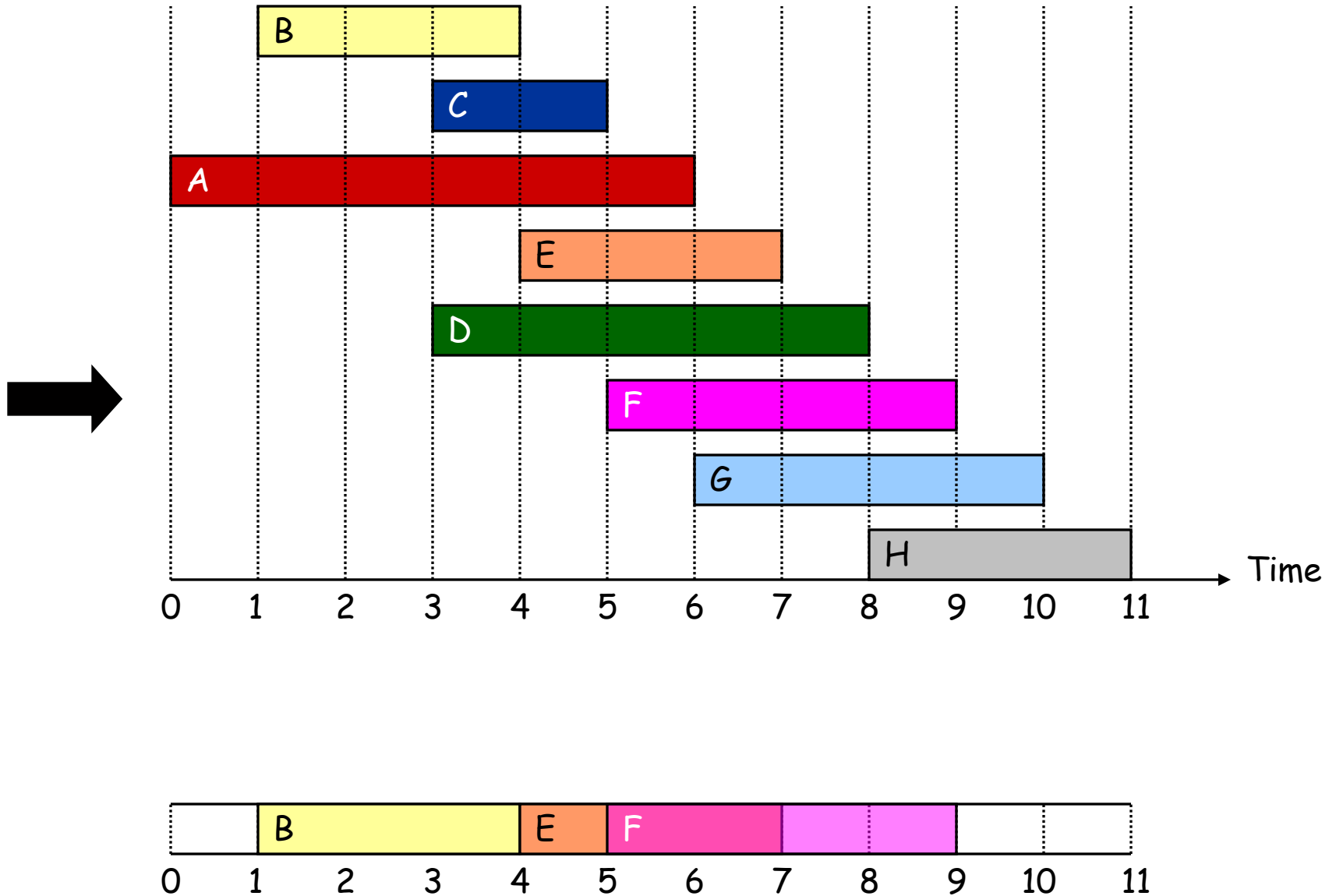
Interval Scheduling



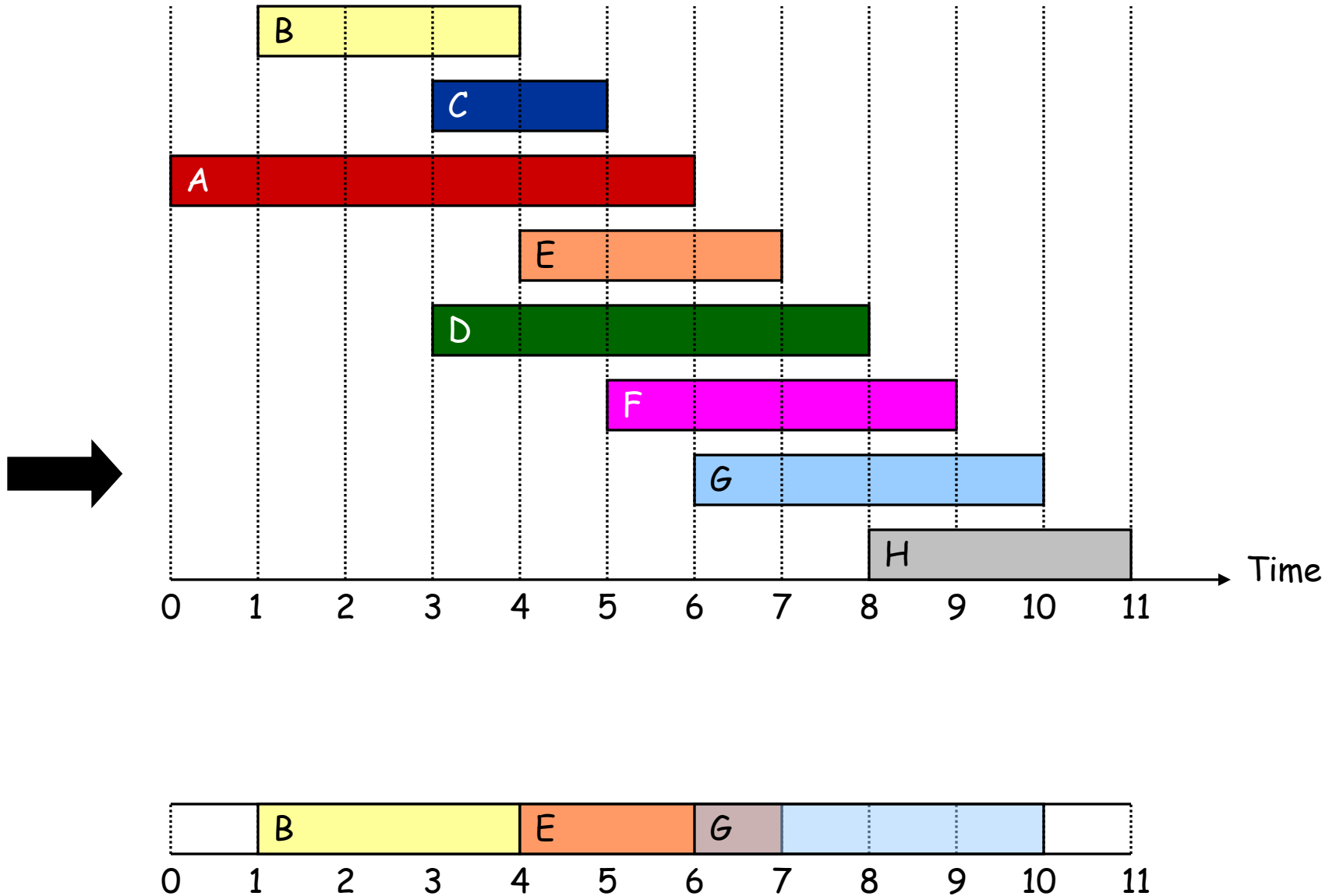
Interval Scheduling



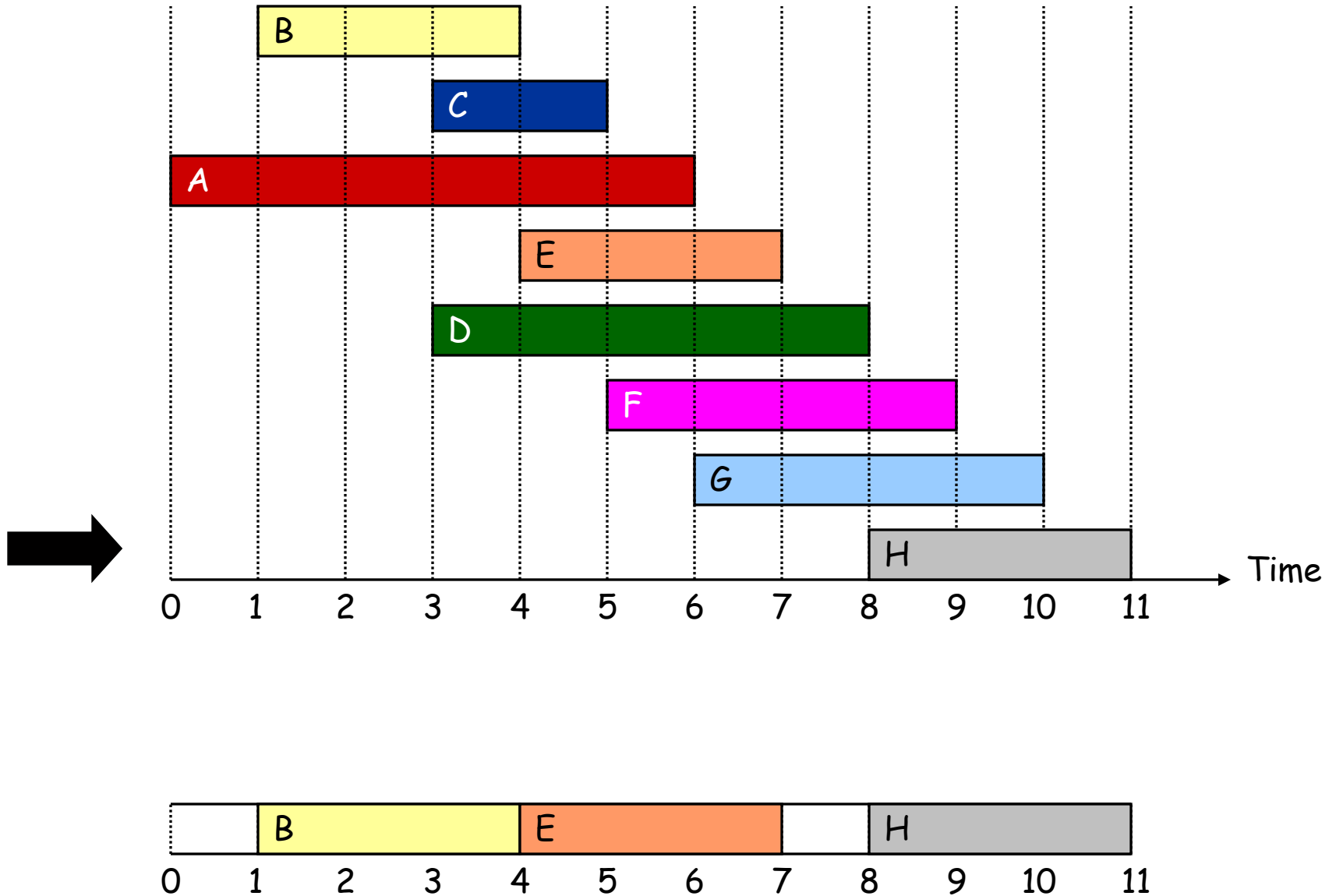
Interval Scheduling



Interval Scheduling



Interval Scheduling

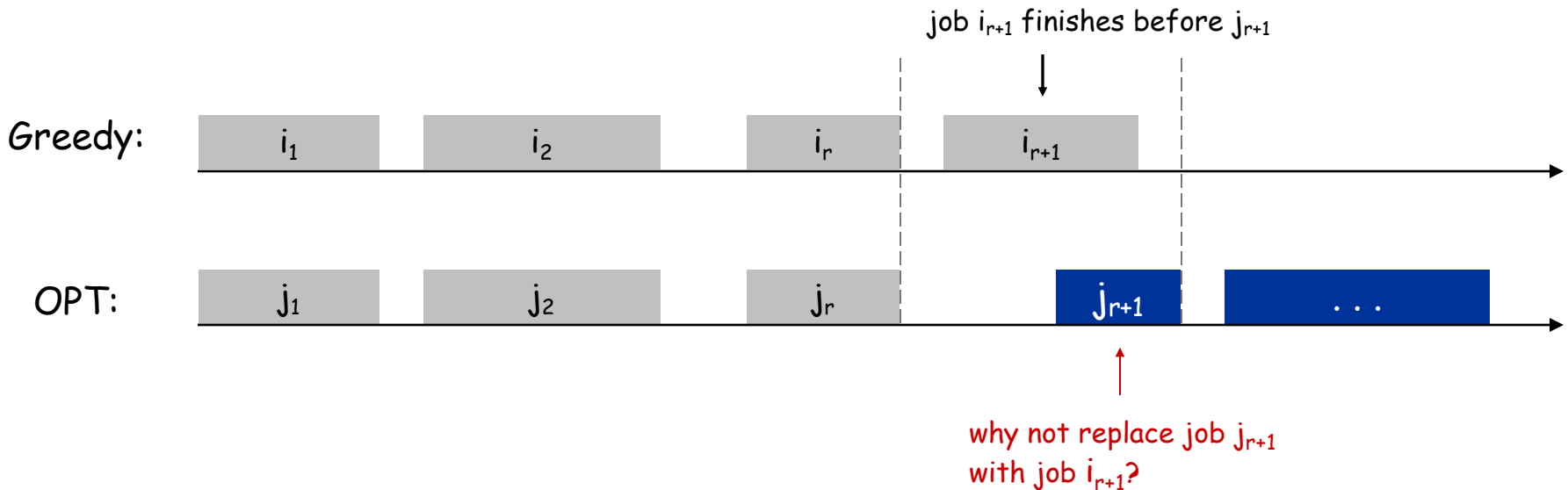


Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let i_1, i_2, \dots, i_k denote set of jobs selected by greedy.
- Let j_1, j_2, \dots, j_m denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \dots, i_r = j_r$ for the largest possible value of r .

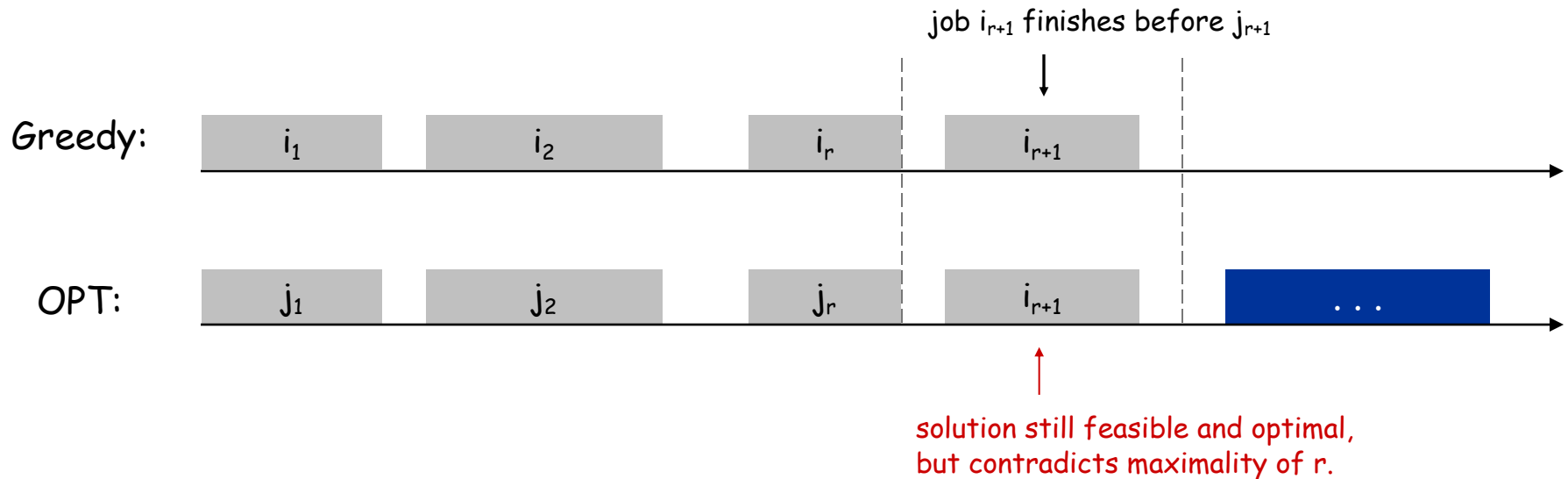


Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let i_1, i_2, \dots, i_k denote set of jobs selected by greedy.
- Let j_1, j_2, \dots, j_m denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \dots, i_r = j_r$ for the largest possible value of r .



Related questions

- <https://leetcode.com/problems/jump-game/>

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.

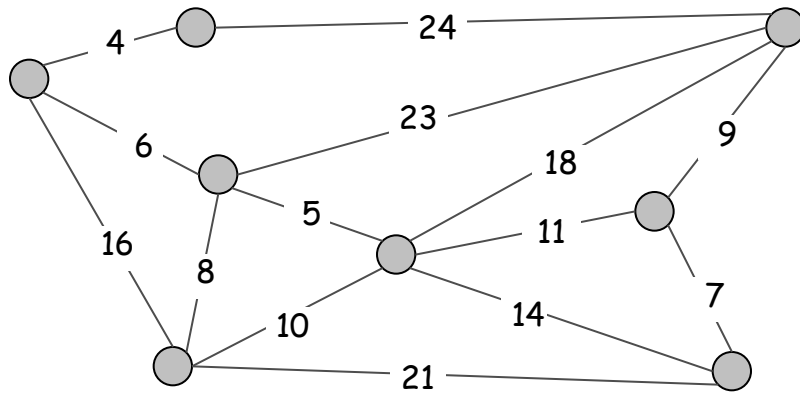
Structural. Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

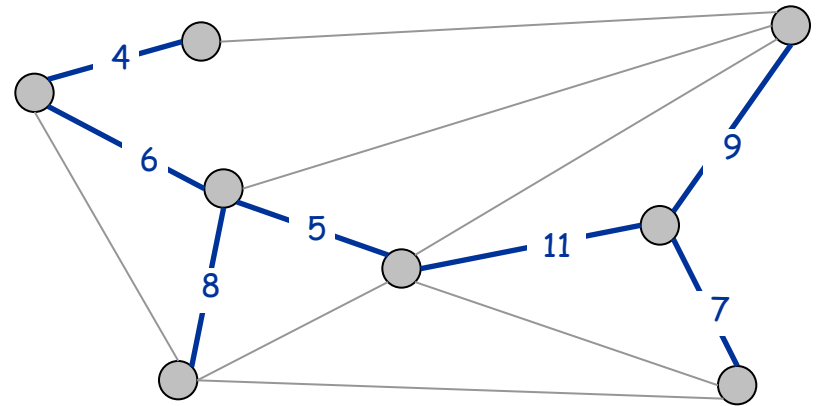
Other greedy algorithms. Kruskal, Prim, Dijkstra, Huffman, ...

Minimum Spanning Tree

Minimum spanning tree. Given a connected graph $G = (V, E)$ with real-valued edge weights c_e , an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree whose sum of edge weights is minimized.



$G = (V, E)$



$T, \sum_{e \in T} c_e = 50$

Cayley's Theorem. There are n^{n-2} spanning trees of K_n .

↑
can't solve by brute force

Applications

MST is fundamental problem with diverse applications.

- Network design.
 - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems.
 - traveling salesperson problem, Steiner tree
- Indirect applications.
 - max bottleneck paths
 - LDPC codes for error correction
 - image registration with Renyi entropy
 - learning salient features for real-time face verification
 - reducing data storage in sequencing amino acids in a protein
 - model locality of particle interactions in turbulent fluid flows
 - autoconfig protocol for Ethernet bridging to avoid cycles in a network

Greedy Algorithms

Kruskal's algorithm. Start with $T = \emptyset$. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.

Reverse-Delete algorithm. Start with $T = E$. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T .

Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T .

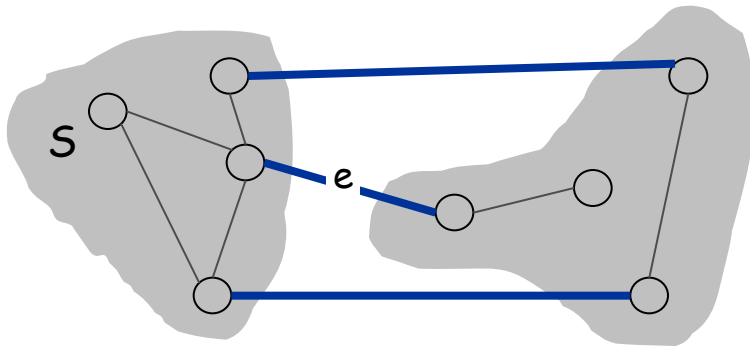
Remark. All three algorithms produce an MST.

Greedy Algorithms

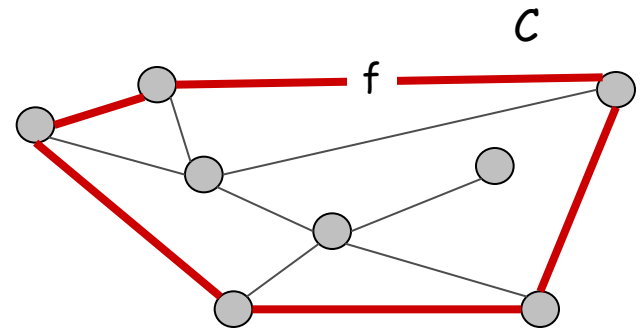
Simplifying assumption. All edge costs c_e are distinct.

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

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C . Then the MST does not contain f .



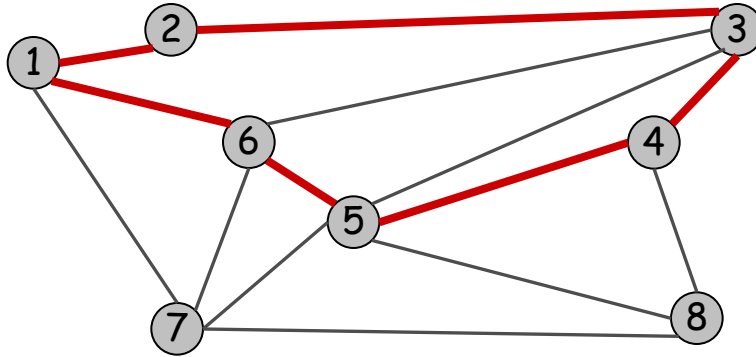
e is in the MST



f is not in the MST

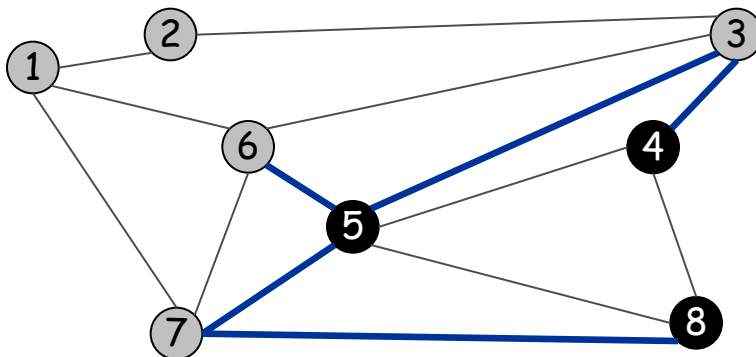
Cycles and Cuts

Cycle. Set of edges the form $a-b, b-c, c-d, \dots, y-z, z-a$.



Cycle $C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1$

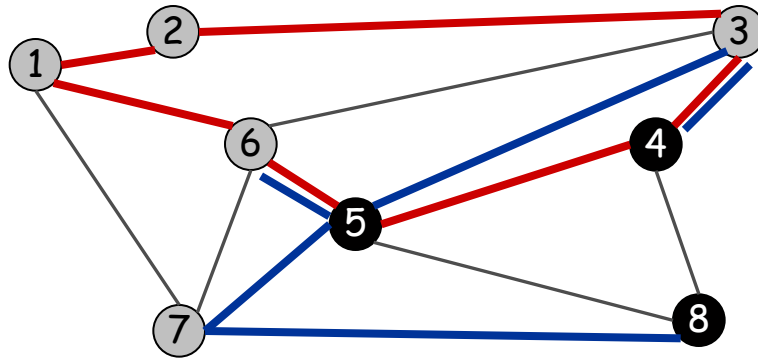
Cutset. A cut is a subset of nodes S . The corresponding cutset D is the subset of edges with exactly one endpoint in S .



Cut $S = \{4, 5, 8\}$
Cutset $D = 5-6, 5-7, 3-4, 3-5, 7-8$

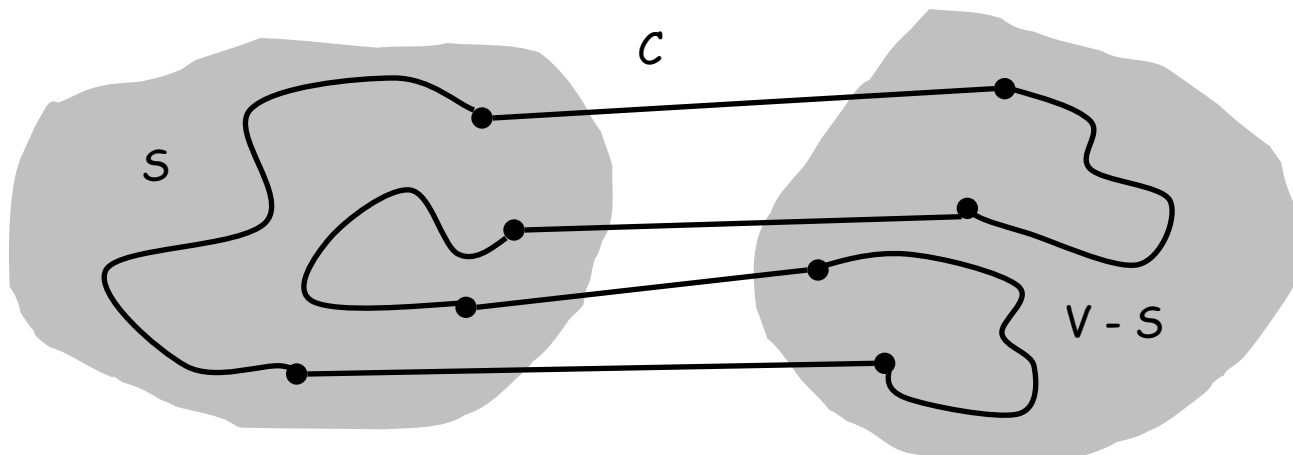
Cycle-Cut Intersection

Claim. A cycle and a cutset intersect in an even number of edges.



Cycle $C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1$
Cutset $D = 3-4, 3-5, 5-6, 5-7, 7-8$
Intersection = $3-4, 5-6$

Pf. (by picture)



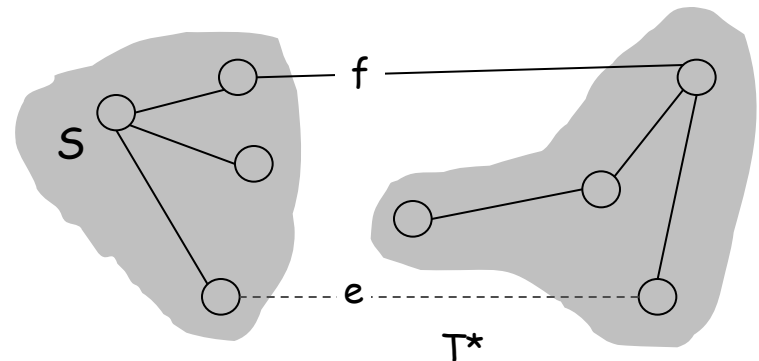
Greedy Algorithms

Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S . Then the MST T^* contains e .

Pf. (exchange argument)

- Suppose e does not belong to T^* , and let's see what happens.
- Adding e to T^* creates a cycle C in T^* .
- Edge e is both in the cycle C and in the cutset D corresponding to S
 \Rightarrow there exists another edge, say f , that is in both C and D .
- $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
- Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$.
- This is a contradiction. ▪



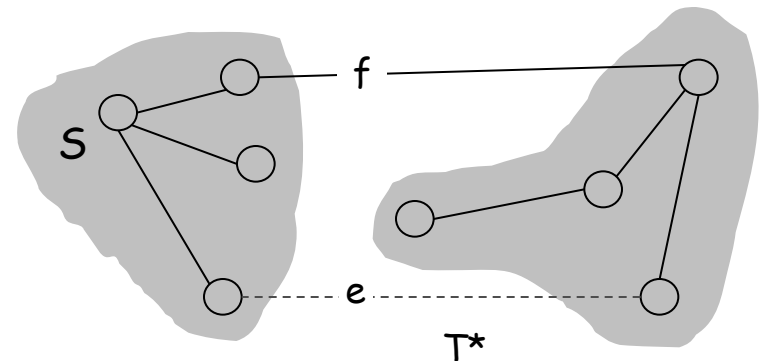
Greedy Algorithms

Simplifying assumption. All edge costs c_e are distinct.

Cycle property. Let C be any cycle in G , and let f be the max cost edge belonging to C . Then the MST T^* does not contain f .

Pf. (exchange argument)

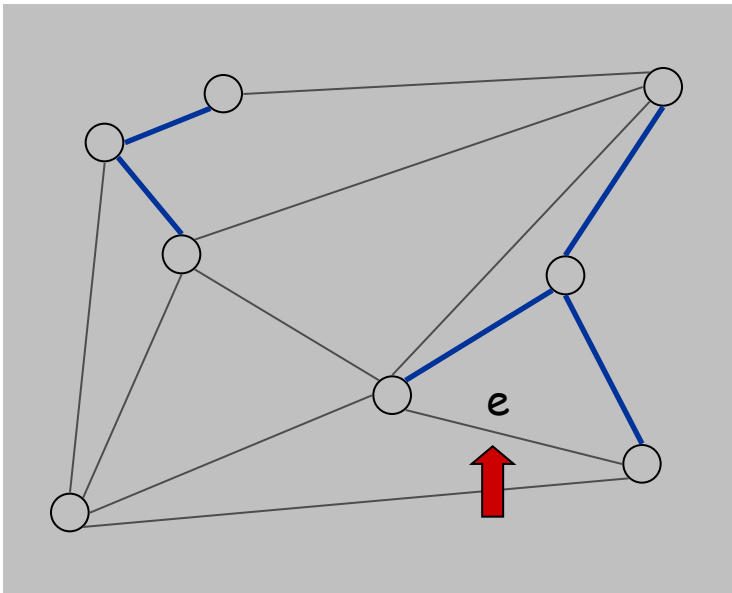
- Suppose f belongs to T^* , and let's see what happens.
- Deleting f from T^* creates a cut S in T^* .
- Edge f is both in the cycle C and in the cutset D corresponding to S
 \Rightarrow there exists another edge, say e , that is in both C and D .
- $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
- Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$.
- This is a contradiction. ■



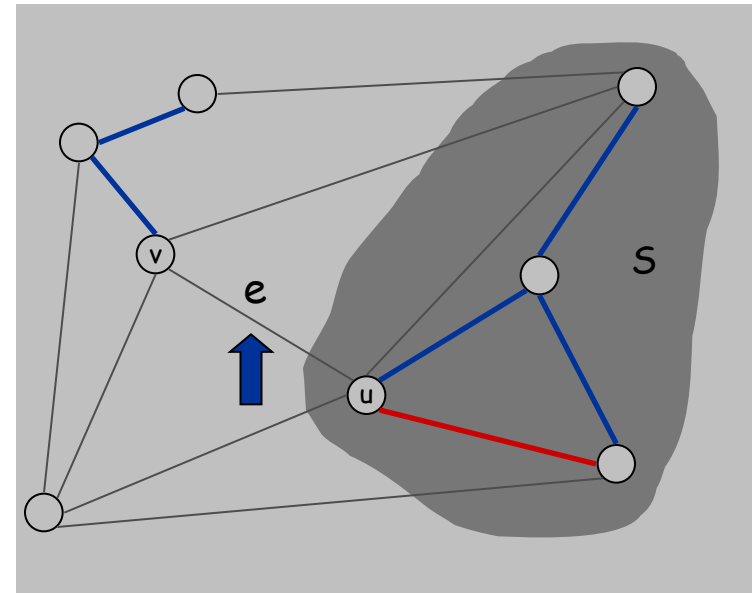
Kruskal's Algorithm: Proof of Correctness

Kruskal's algorithm. [Kruskal, 1956]

- Consider edges in ascending order of weight.
- Case 1: If adding e to T creates a cycle, discard e according to cycle property.
- Case 2: Otherwise, insert $e = (u, v)$ into T according to cut property where S = set of nodes in u 's connected component.



Case 1



Case 2

Implementation: Kruskal's Algorithm

Implementation. Use the **union-find** data structure.

- Build set T of edges in the MST.
- Maintain set for each connected component.
- $O(m \log n)$ for sorting and $O(m \underbrace{\alpha(m, n)}_{\text{essentially a constant}})$ for union-find.

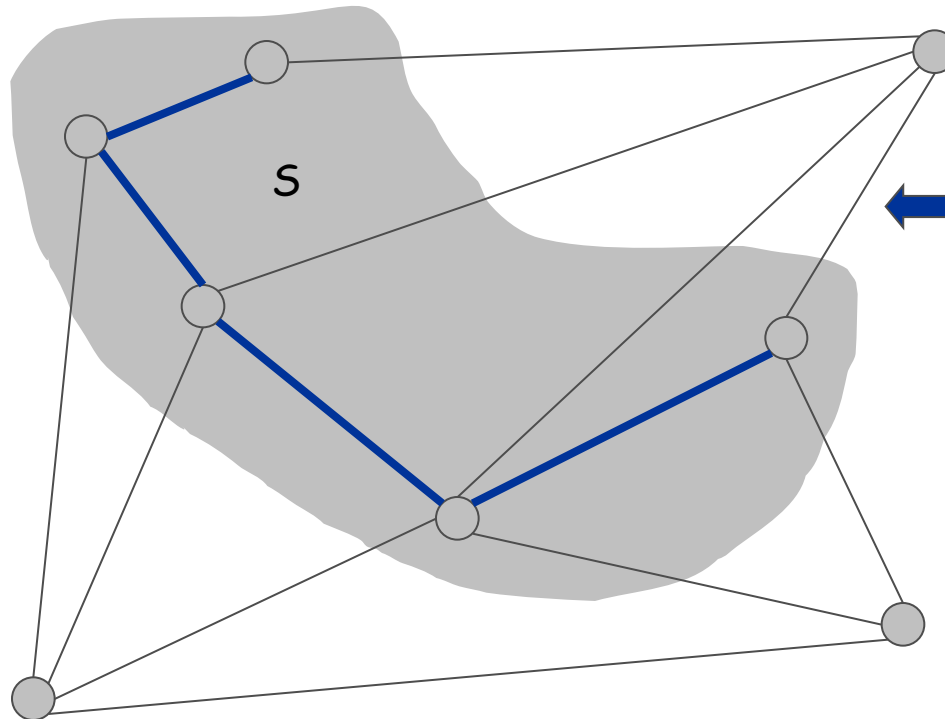
\swarrow $m \leq n^2 \Rightarrow \log m$ is $O(\log n)$ $\underbrace{\hspace{1.5cm}}$ essentially a constant

```
Kruskal(G, c) {  
    Sort edges weights so that  $c_1 \leq c_2 \leq \dots \leq c_m$ .  
     $T \leftarrow \phi$   
  
    foreach ( $u \in V$ ) make a set containing singleton  $u$   
  
    for  $i = 1$  to  $m$       are  $u$  and  $v$  in different connected components?  
         $(u, v) = e_i$        $\swarrow$   
        if ( $u$  and  $v$  are in different sets) {  
             $T \leftarrow T \cup \{e_i\}$   
            merge the sets containing  $u$  and  $v$   
        }  
         $\swarrow$  merge two components  
    return  $T$   
}
```

Prim's Algorithm: Proof of Correctness

Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]

- Initialize S = any node.
- Apply cut property to S .
- Add min cost edge in cutset corresponding to S to T , and add one new explored node u to S .



Implementation: Prim's Algorithm

Implementation. Use a priority queue ala Dijkstra.

- Maintain set of explored nodes S .
- For each unexplored node v , maintain attachment cost $a[v]$ = cost of cheapest edge v to a node in S .
- $O(n^2)$ with an array; $O(m \log n)$ with a binary heap.

```
Prim(G, c) {  
    foreach ( $v \in V$ )  $a[v] \leftarrow \infty$   
    Initialize an empty priority queue  $Q$   
    foreach ( $v \in V$ ) insert  $v$  onto  $Q$   
    Initialize set of explored nodes  $S \leftarrow \phi$   
  
    while ( $Q$  is not empty) {  
         $u \leftarrow$  delete min element from  $Q$   
         $S \leftarrow S \cup \{u\}$   
        foreach (edge  $e = (u, v)$  incident to  $u$ )  
            if ( $(v \notin S)$  and ( $c_e < a[v]$ ))  
                decrease priority  $a[v]$  to  $c_e$   
    }  
}
```

MST Algorithms: Theory

Deterministic comparison based algorithms.

- $O(m \log n)$ [Jarník, Prim, Dijkstra, Kruskal, Boruvka]
- $O(m \log \log n)$. [Cheriton-Tarjan 1976, Yao 1975]
- $O(m \beta(m, n))$. [Fredman-Tarjan 1987]
- $O(m \log \beta(m, n))$. [Gabow-Galil-Spencer-Tarjan 1986]
- $O(m \alpha(m, n))$. [Chazelle 2000]

Holy grail. $O(m)$.

Notable.

- $O(m)$ randomized. [Karger-Klein-Tarjan 1995]
- $O(m)$ verification. [Dixon-Rauch-Tarjan 1992]

Euclidean.

- 2-d: $O(n \log n)$. compute MST of edges in Delaunay
- k-d: $O(k n^2)$. dense Prim

Related Question

- <https://leetcode.com/problems/min-cost-to-connect-all-points/>