COMP 3011
DESIGN AND ANALYSIS OF ALGORITHMS
FALL 2024

Graphs and Greedy Algorithms

LI Bo
Department of Computing
The Hong Kong Polytechnic University

COMMENTS

Comment: I think for the difficulties part, you can provide some solid supplementary sources on BB for those who never learn anything about algorithms. And then keep the difficulties in lecture for the advanced students.

- Solid supplementary sources on BB
  1. VIII Appendix: Mathematical Background (Textbook: Introduction to Algorithms)
  2. Self-Reading (Math Foundations) on BB
- Keep the difficulties in lecture for the advanced students
  1. Will Keep the difficulty standard.
  2. Look at the materials from the angles of design and analysis.

COMMENTS

- We have tutorial next Monday (Sep 16, 2024)
- We do not have lectures next Wednesday (Sep 18, 2024)

Happy Mooncake festival!

BIPARTITE GRAPHS

BIPARTITE GRAPHS

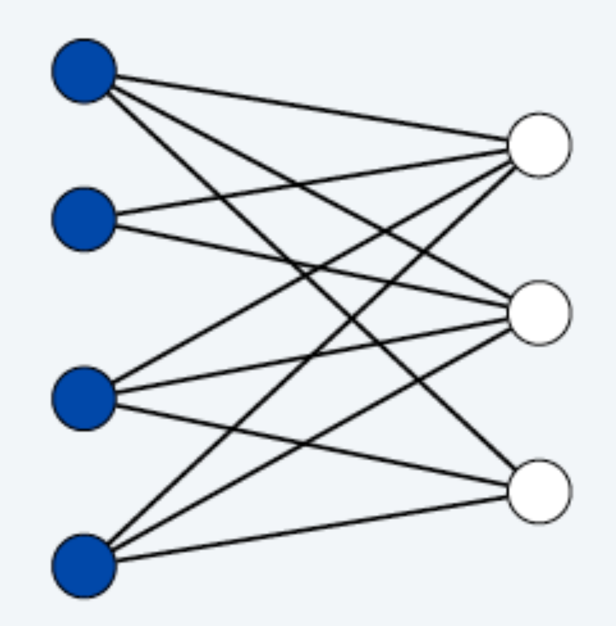
Definition (Bipartite Graphs)

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

Application

- Stable matching: med-school students = blue,

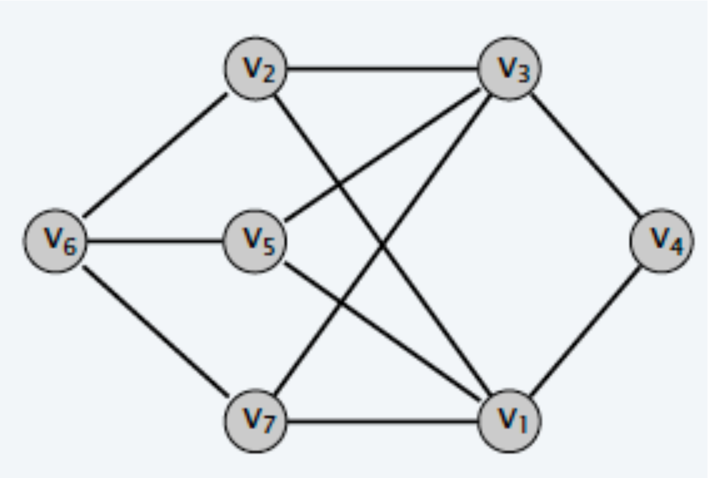
  hospitals = white.

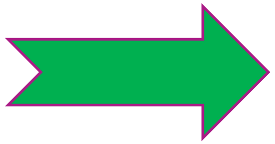


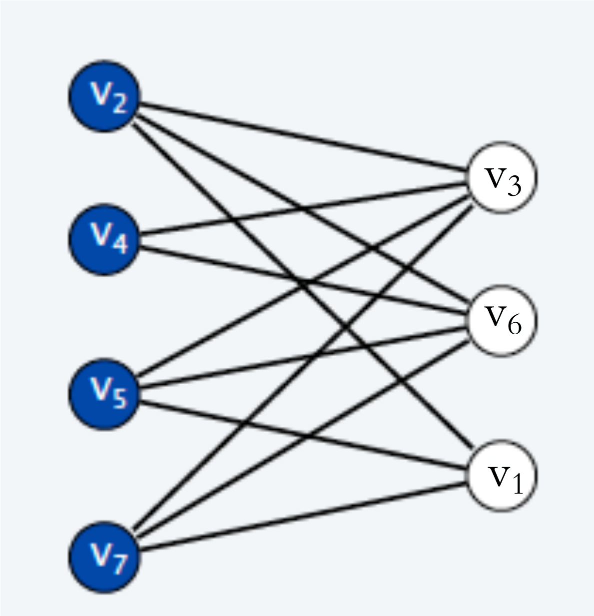
BIPARTITE GRAPHS

- Is this graph bipartite?

Given a graph, how to determine whether it is bipartite or not?





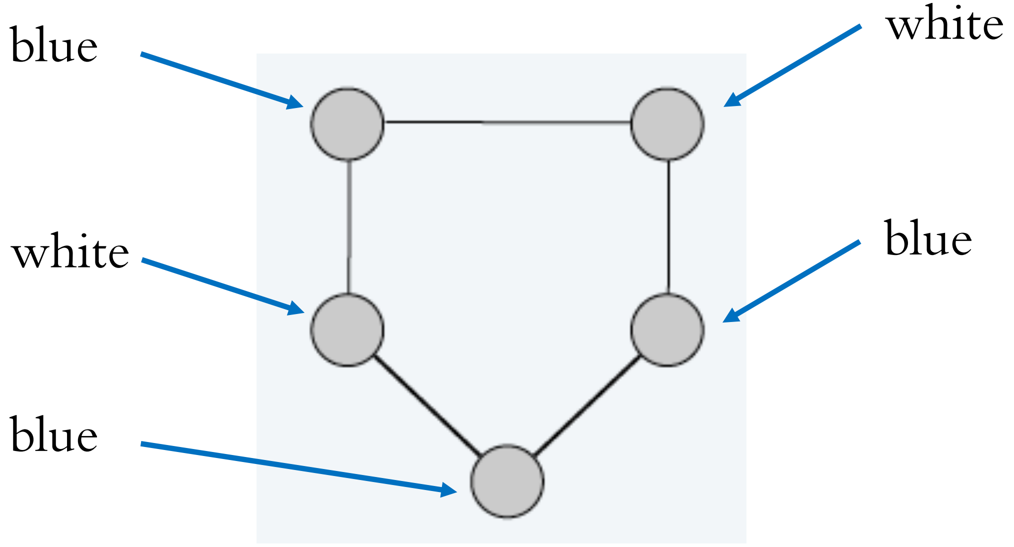


BIPARTITE GRAPHS

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

Proof:

➢ Not possible to 2-color the odd-length cycle, let alone G. ∎



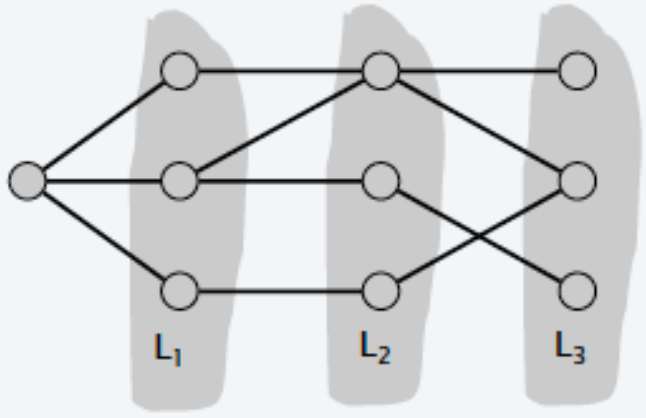
An edge has different colors on the two end points.

BIPARTITE GRAPHS

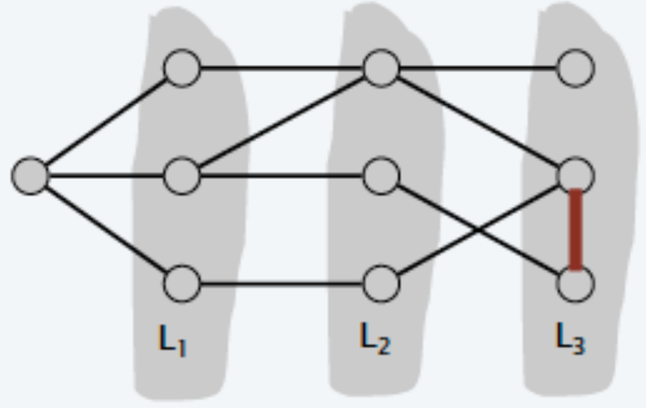
Lemma. Let G be a connected graph, and let $L_0, \dots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

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)

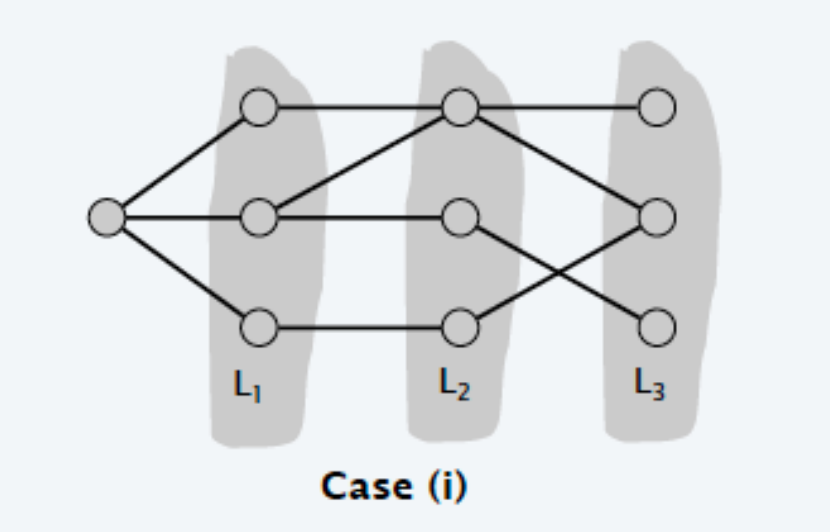


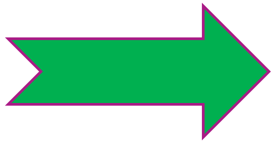
Case (ii)

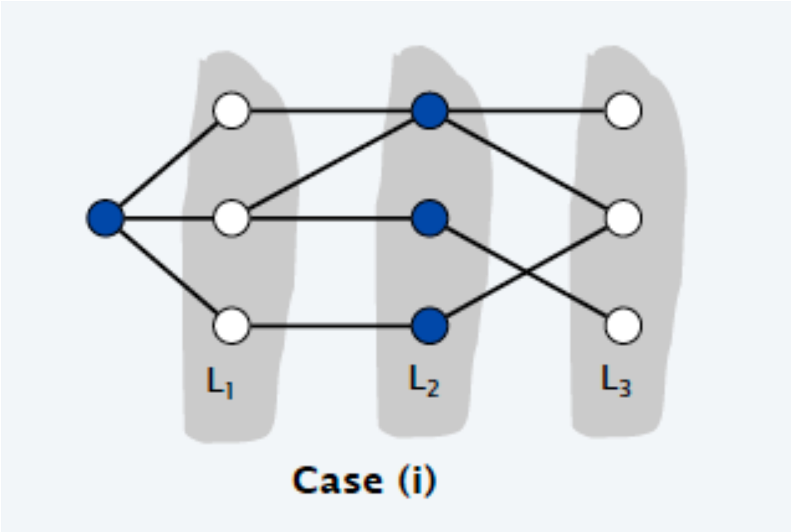
BIPARTITE GRAPHS

Case i:

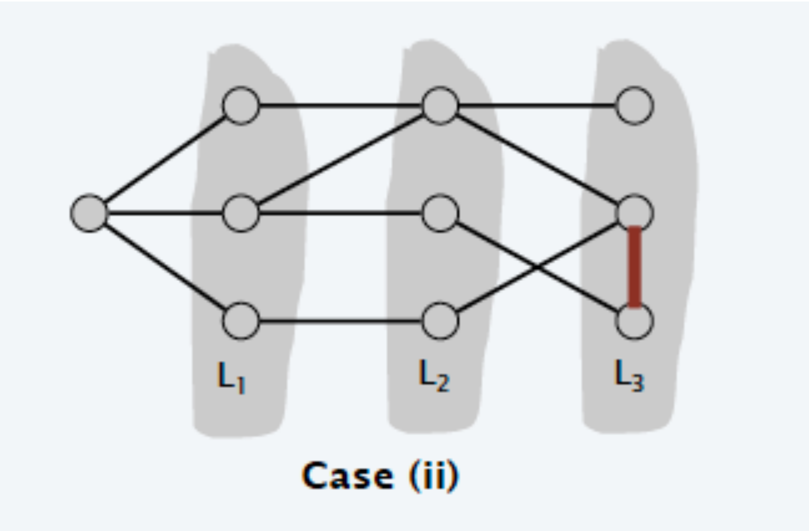
- Suppose no edge joins two nodes in same layer.
- By BFS property, each edge joins two nodes in adjacent levels.
- Bipartition: white = nodes on odd levels, blue = nodes on even levels.





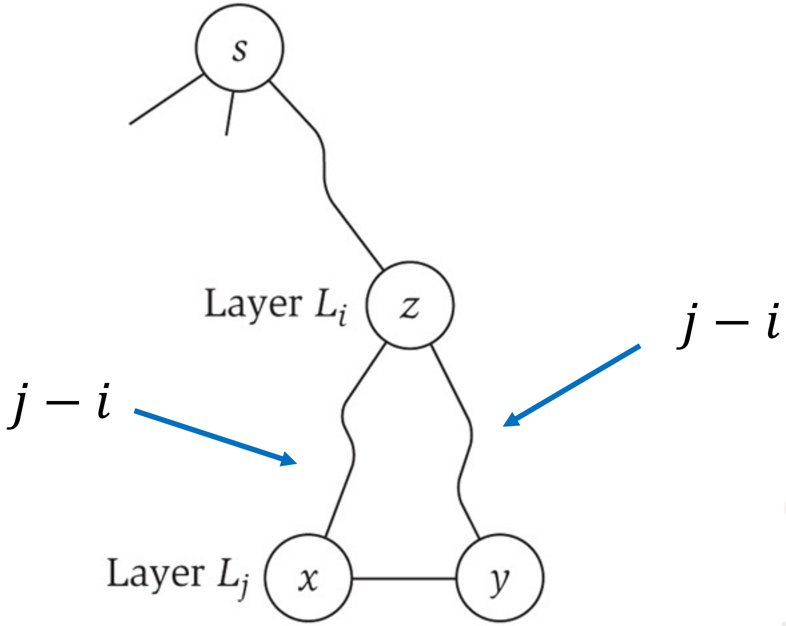


BIPARTITE GRAPHS



Case ii:

- Suppose (x, y) is an edge with x, y in same level L_j.
- Let $z = \text{lca}(x, y) =$ lowest common ancestor.
- Let L_i be level containing z.
- Consider cycle that takes edge from x to y, then path from y to z, then path from z to x.
- Its length is $1 + (j - i) + (j - i) = 2(j - i) + 1$, which is odd. ■



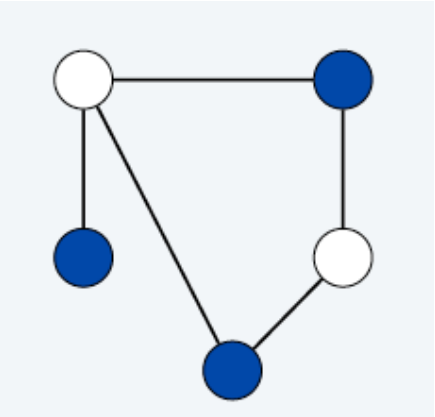
BIPARTITE GRAPHS

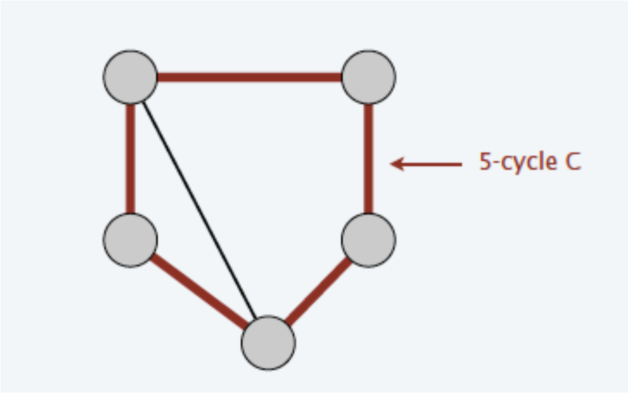
Lemma. Exactly one of the following holds.

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).

Corollary. A graph G is bipartite if and only if it contains no odd-length cycle.

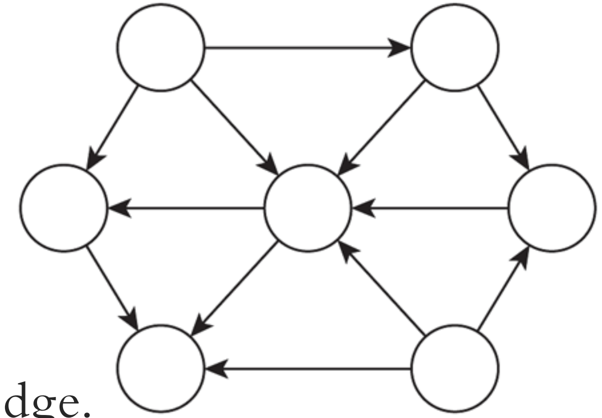




DIRECTED GRAPHS

In-degree: #incoming edges
Out-degree: #outgoing edges

DIRECTED GRAPHS



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

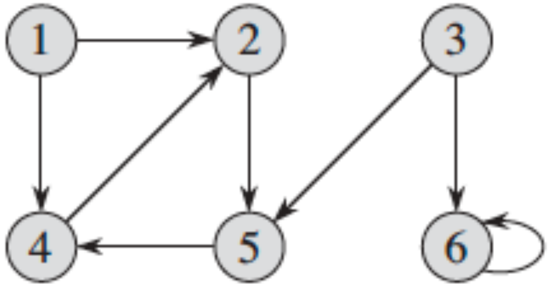
➤ Edge (u, v) leaves node u and enters node v.

Adjacency matrix. n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge.

➤ Space complexity $O(n^2)$.

➤ Checking if (u, v) is an edge takes constant time.

➤ Identifying all edges takes $\Theta(n^2)$ time.

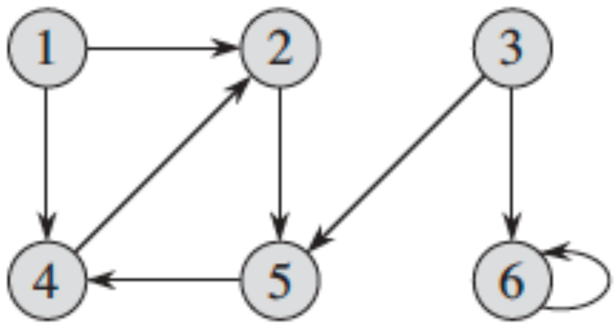


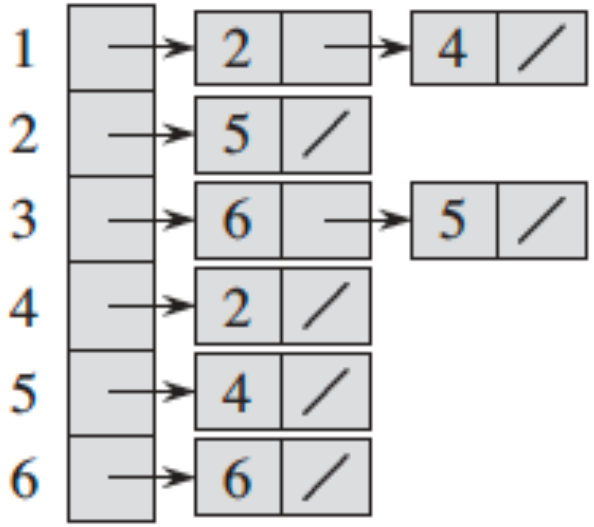
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

DIRECTED GRAPHS

Adjacency lists. Node-indexed array of lists.

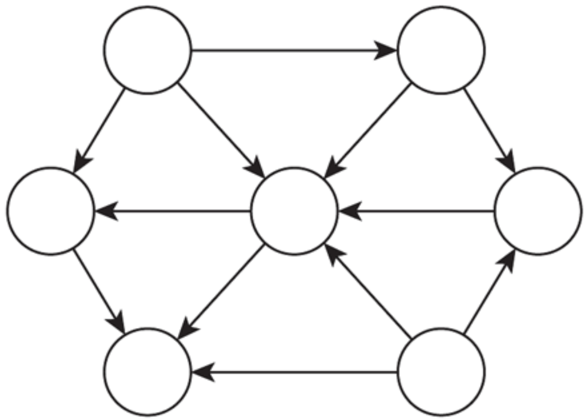
- Space complexity is $\Theta(m+n)$.
- Checking if (u, v) is an edge takes $O(\text{degree}(u))$ time.
- Identifying all edges takes $\Theta(m+n)$ time.





DIRECTED GRAPHS

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



Directed reachability.

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

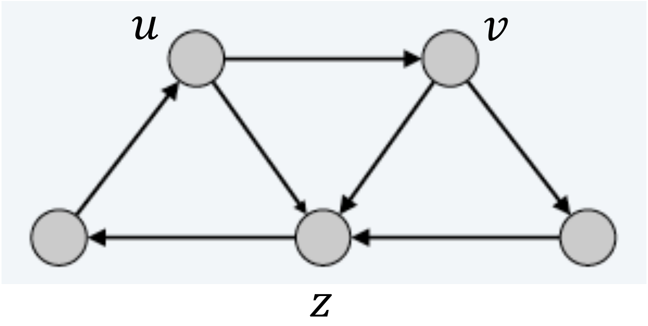
Directed $\boldsymbol{s-t}$ shortest path problem.

Given two nodes s and t, what is the length of a shortest path from s to t?

➢ ***Graph search***: BFS extends naturally to directed graphs.

Connectivity?

DIRECTED GRAPHS



Definition.

- Nodes u and v are ***mutually reachable*** if there is both a path from u to v and also a path from v to u.
- A graph is ***strongly connected*** if every pair of nodes is mutually reachable.

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

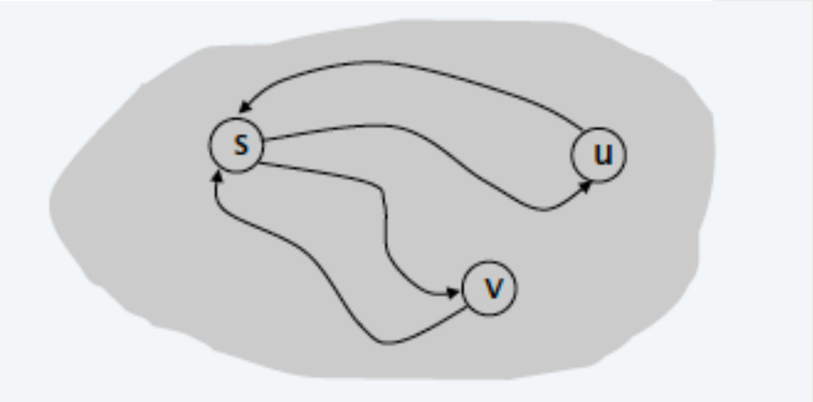
Proof.

$\Rightarrow$ Follows fro

$\Leftarrow$ Path from u ... with $s \rightsquigarrow v$ path.

Path from v ... with $s \rightsquigarrow u$ path. ■

Given a graph, how to determine whether it is strongly connected or not?



DIRECTED GRAPHS

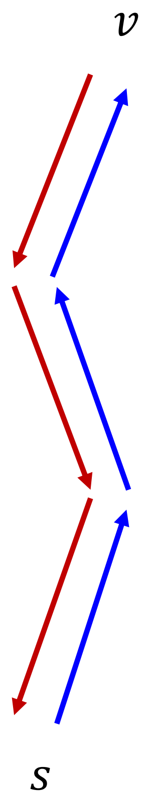
Theorem. Can determine if G is strongly connected in $O(m+n)$ time.

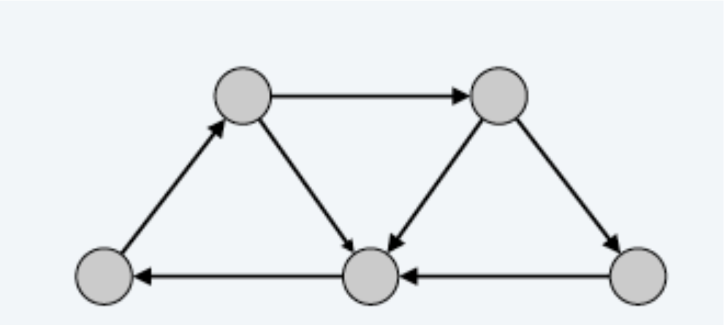
Proof.

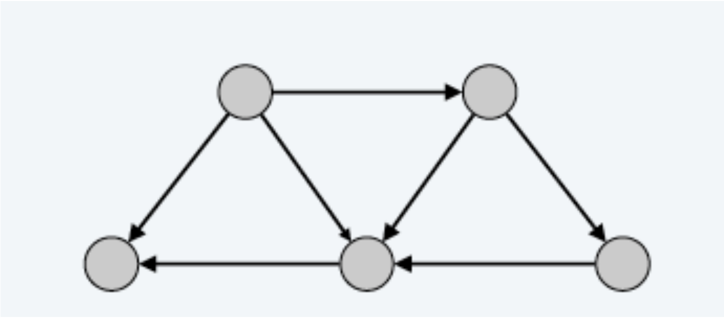
- Pick any node s.
- Run BFS from s in G.

reverse orientation of every edge in G

- Run BFS from s in G^{reverse}.
- Return true if and only if all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma. ■







DIRECTED ACYCLIC GRAPHS

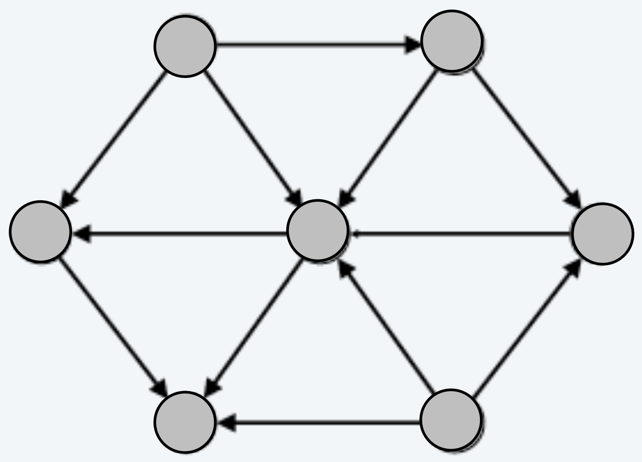
DIRECTED ACYCLIC GRAPHS

If the graph contains a cycle, then no linear ordering is possible.

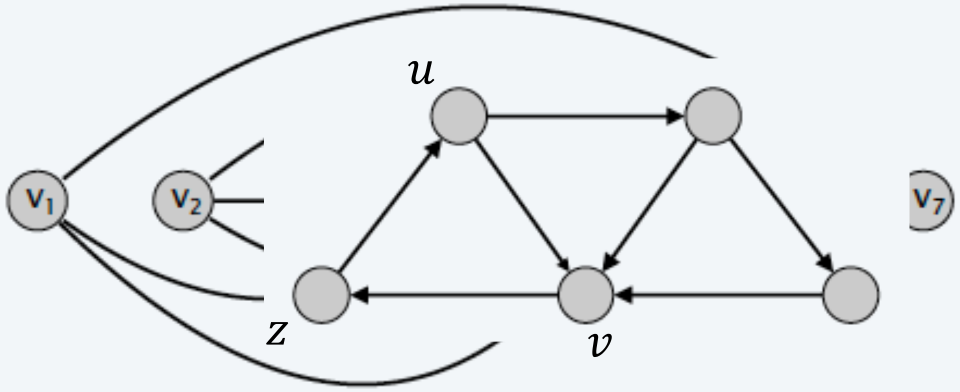
Definitions

An ordering of the nodes so that all edges point "forward".

- A directed acyclic graphs (DAG) is a directed graph that contains no directed cycles.
- 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

DIRECTED ACYCLIC GRAPHS

$v \longrightarrow u$

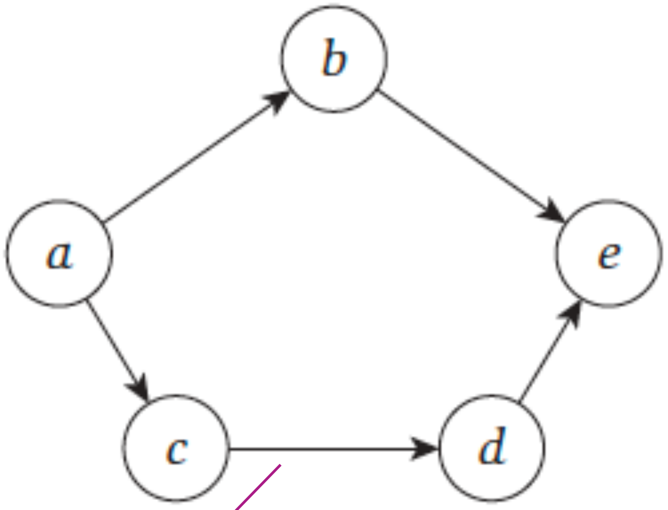
- How many topological orderings does the following graph have?
- An ordering of the nodes so that all edges point "forward".

"a" must be first and "e" must be last

Exhaustive Search: there will be 5*4*3*2 = 120 possibilities.

Observations:

- The first node must be one that has no edge coming into it.
- The last node must be one that has no edge leaving it.



a, b, c, d, e

a, c, b, d, e

a, c, d, b, e

"c" must be before "d"

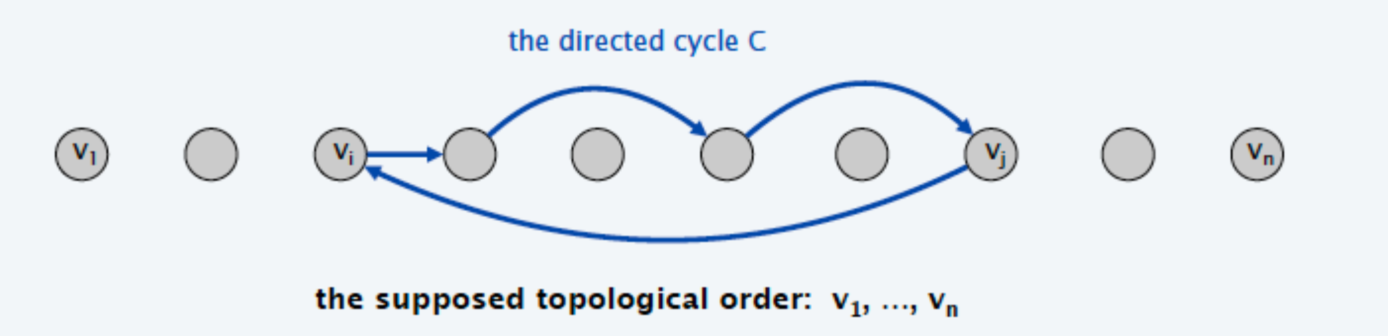
DIRECTED ACYCLIC GRAPHS

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

Proof [by contradiction]

- Suppose that G has a topological order $v_1, v_2, \ldots, v_n$ and that G also has a directed cycle C.
- 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 $v_1, v_2, \ldots, v_n$is a topological order, we must have $j < i$, a contradiction. ■



DIRECTED ACYCLIC GRAPHS

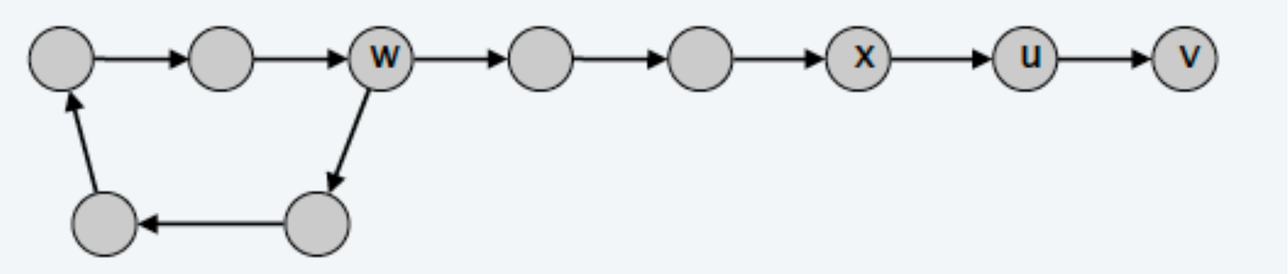
- Question: Does every DAG have a topological ordering?
- Question: If so, how do we compute one?

DIRECTED ACYCLIC GRAPHS

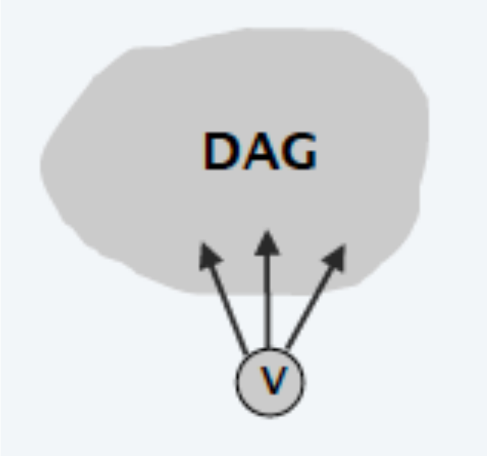
Lemma. If G is a DAG, then G has a node with no entering edges.

Proof.[by contradiction]

- Suppose that G is a DAG and every node has at least one entering edge.
- Pick any node v, and begin following edges backward from v.
- Since v has at least one entering edge (u, v) we can walk backward to u.
- Then, since u has at least one entering 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.



DIRECTED ACYCLIC GRAPHS



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

Proof. [by induction on n]

- Base case: true if $n = 1$.
- Inductive hypothesis: Assume true for $k < n$ nodes.
- Given DAG on $n > 1$ nodes, find a node v with no entering 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 entering edges. ■

DIRECTED ACYCLIC GRAPHS

```
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
```


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

Proof:

Maintain the following information:

- count(w) = remaining number of incoming edges to node w
- S = set of remaining nodes with no incoming edges

Initialization: $O(m+n)$ via single scan through graph.

DIRECTED ACYCLIC GRAPHS

```
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
```


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 ALGORITHMS

INTERVAL SCHEDULING PROBLEM

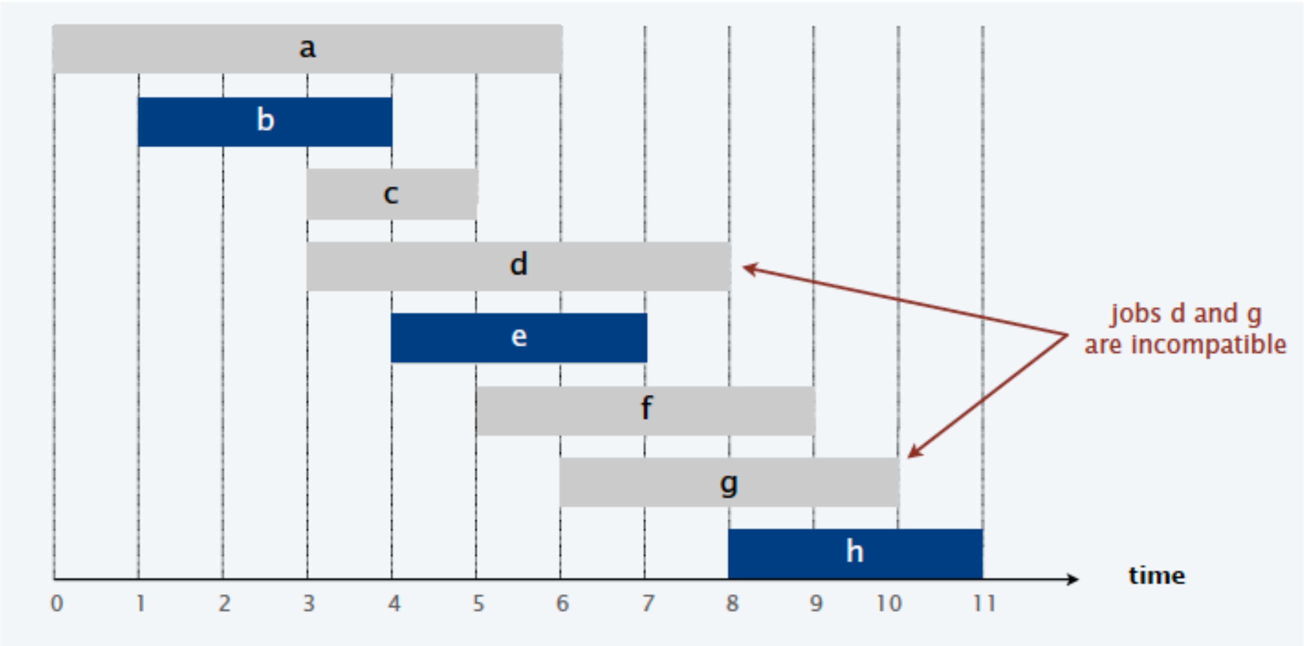
INTERVAL SCHEDULING PROBLEM

Given a set of jobs $J = \{1, 2, \ldots, n\}$

➢ Job j starts at s_j and finishes at $f_j \geq s_j$.

➢ Two jobs (open intervals) are compatible if they don't overlap.

Goal: find maximum subset of mutually compatible jobs.



Intuition: shorter is better

INTERVAL SCHEDULING PROBLEM

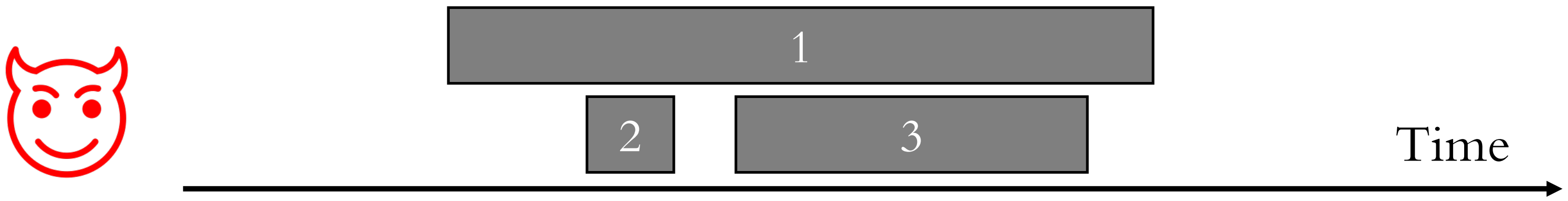
Idea 1:

➢ Repeatedly pick shortest compatible, unscheduled job (i.e. that does not conflict with any scheduled job).

Idea 2:

Intuition: earlier is better

➢ Repeatedly pick compatible job with earliest starting time.



GREEDY ALGORITHM

- Repeatedly pick an item until no more feasible choices.
- Among all feasible choices, we always pick the one that minimizes (or maximizes) ***some property***.
  - length, starting time, …
- Such algorithms are called ***greedy***.

- Greedy algorithms may not be optimal.
- But maybe we have been using the wrong property!

INTERVAL SCHEDULING PROBLEM

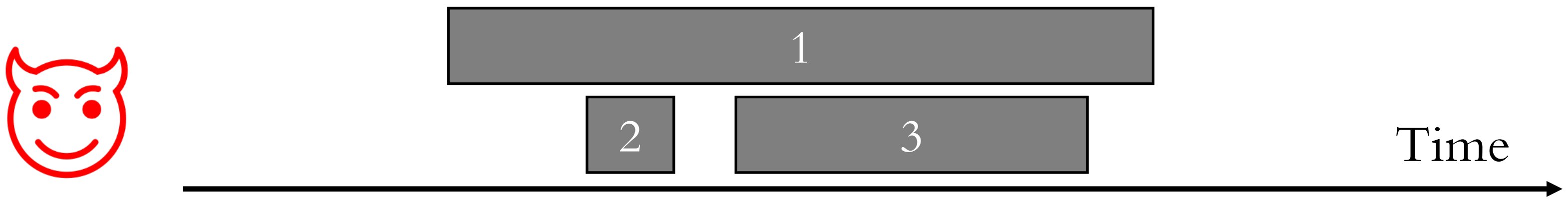
What about earliest-finish-time-first?

Idea 1:

- Repeatedly pick shortest compatible, unscheduled job (i.e. that does not conflict with any scheduled job).

Idea 2:

- Repeatedly pick compatible job with earliest starting time.



EARLIEST-FINISH-TIME-FIRST ALGORITHM

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

$S \leftarrow \varnothing$. ⟵ set of jobs selected

FOR $j = 1$ TO n

  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

RETURN S.

Proposition. Can implement earliest-finish-time first in $O(n \log n)$ time.

EARLIEST-FINISH-TIME-FIRST ALGORITHM

Theorem. The earliest-finish-time-first algorithm is optimal.

Proof. [by contradiction]

Switching j_{r+1} by i_{r+1} in O: Still ***feasible*** and ***optimal***!

- Assume Greedy is not optimal.
- Let $A = \{i_1, i_2, \ldots, i_k\}$ be set of jobs selected by Greedy.
- Let $O = \{j_1, j_2, \ldots, j_m\}$ be set of jobs in an optimal solution. Then $m > k$.
- Let $r + 1$ be first index such that $i_{r+1} \neq j_{r+1}$.

such a job exists ➡ $f_{i_{r+1}} \leq f_{j_{r+1}}$

$i_1 = j_1$ $\quad$ $i_2 = j_2$ $\quad$ $i_r = j_r$ $\quad$ $i_{r+1} \neq j_{r+1}$

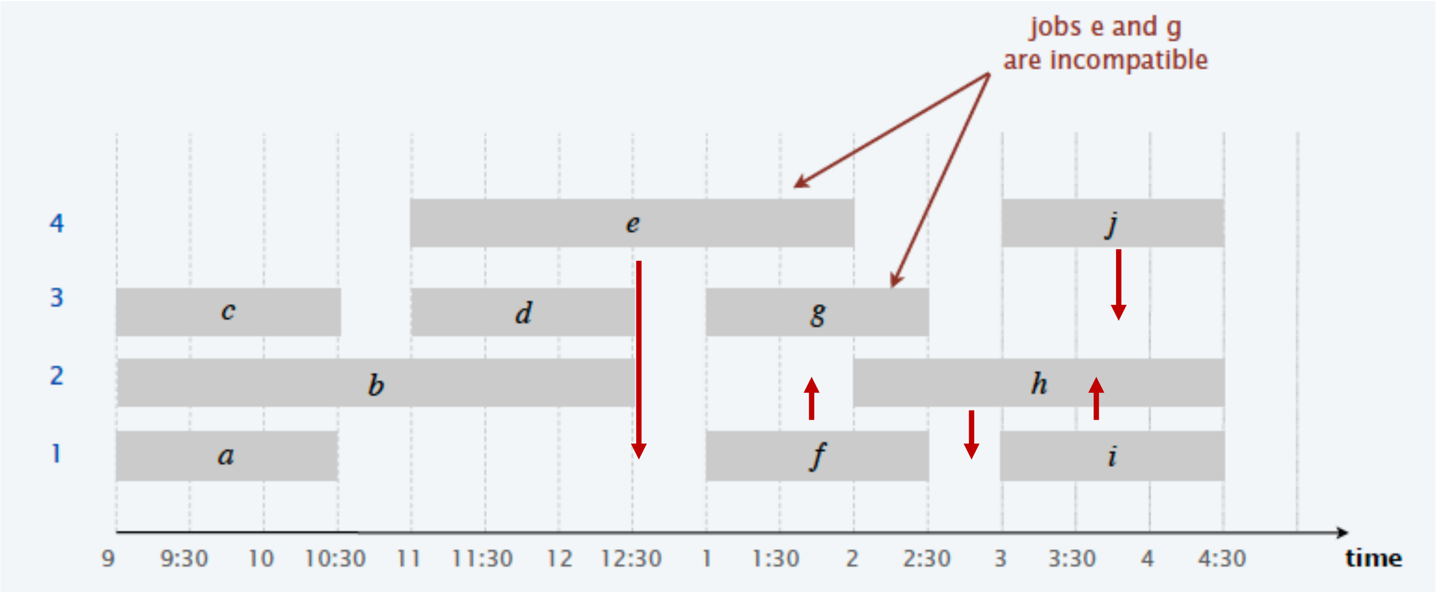
INTERVAL PARTITIONING

INTERVAL PARTITIONING

Given a set of lectures (jobs) $L = \{1, 2, \dots, n\}$;

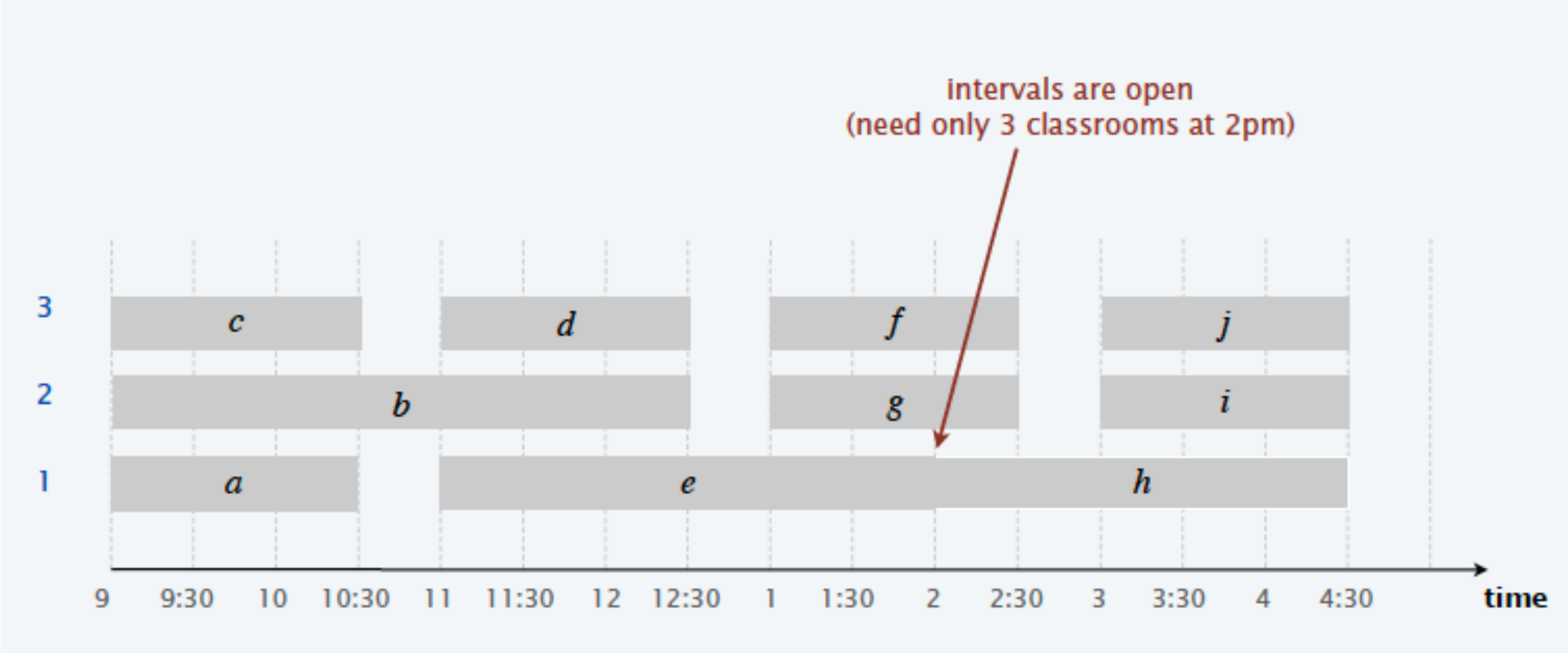
- Lecture j starts at s_j and finishes at $f_j \geq s_j$.
- Two lectures are compatible if they don't overlap.

Goal: find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room



INTERVAL PARTITIONING

- Optimal is 3 classrooms.

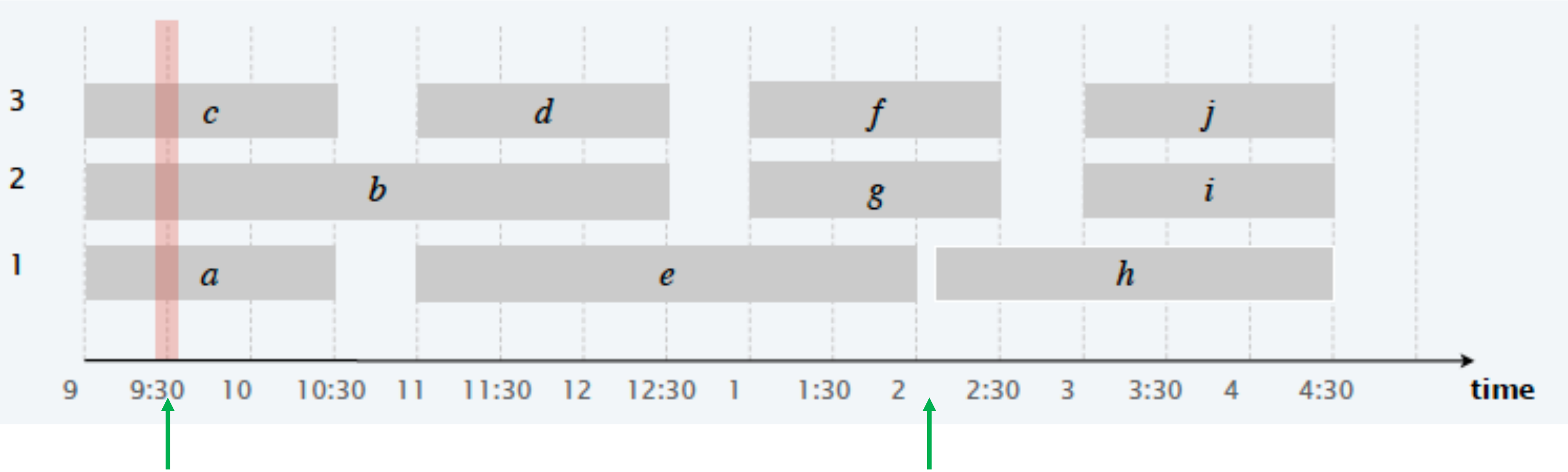


INTERVAL PARTITIONING

Definition. The depth of a set of open intervals is the maximum number of intervals that contain any given point.

Key observation. #rooms needed ≥ depth.

Is depth enough???

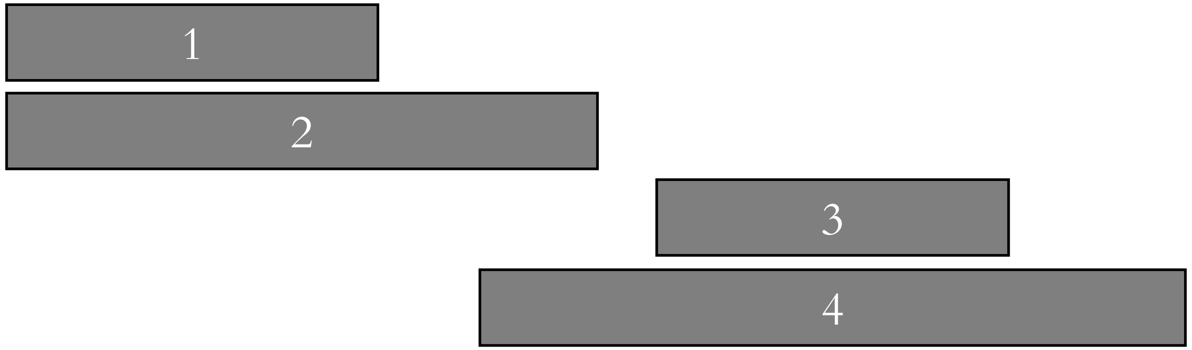


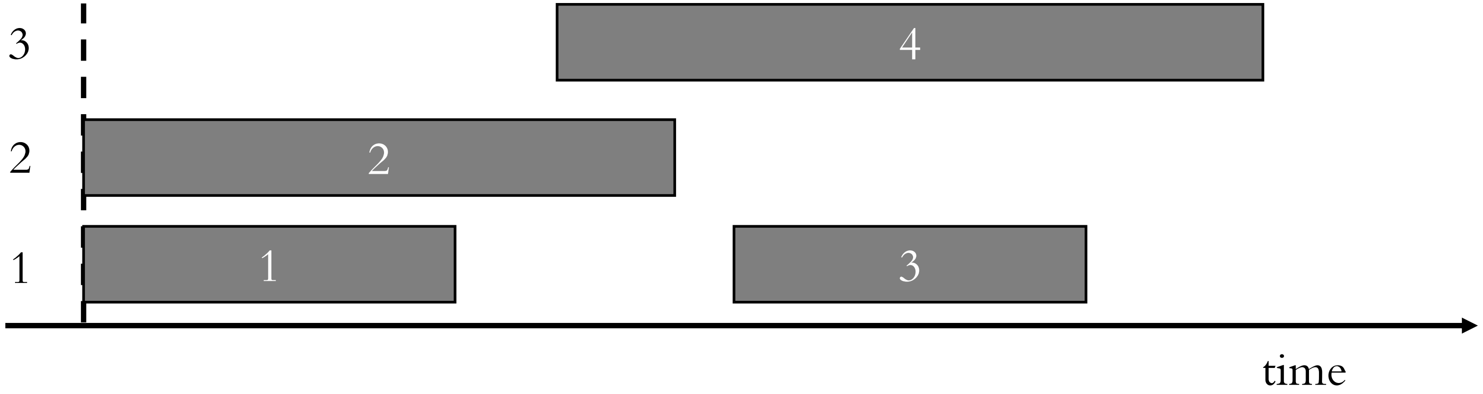
3 classrooms are needed

2 classrooms are needed

INTERVAL PARTITIONING

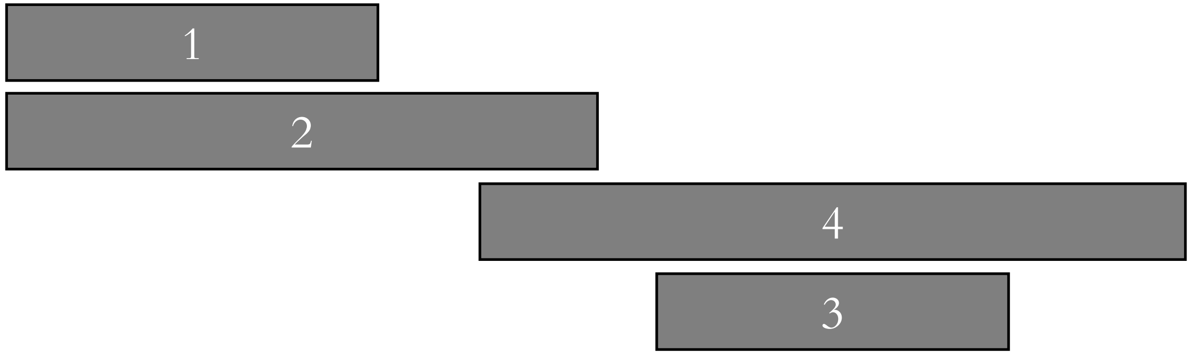
Can we do earliest-finish-time-first?

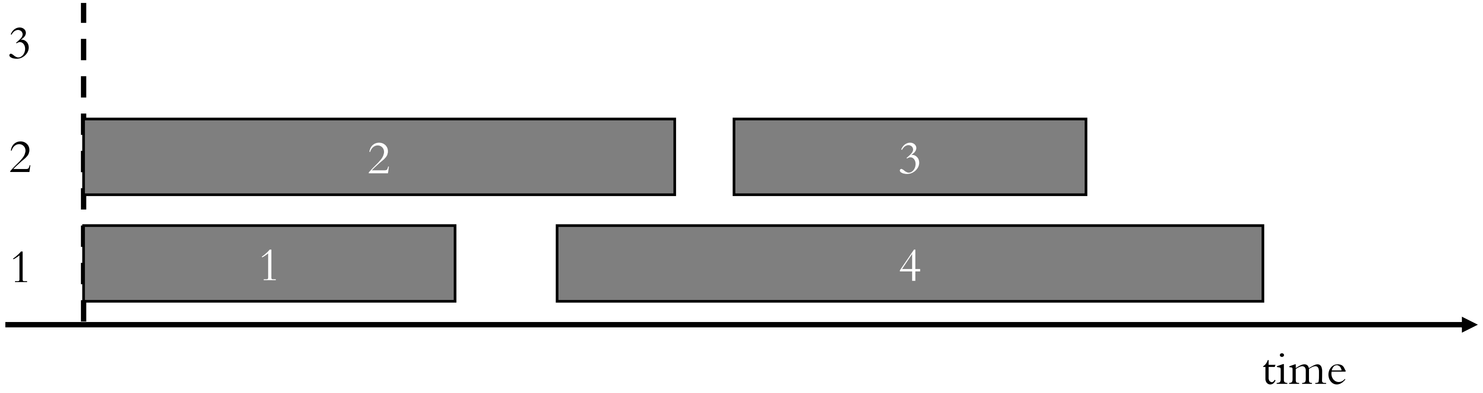




INTERVAL PARTITIONING

Can we do earliest-start-time-first?





INTERVAL PARTITIONING: EARLIEST-START-TIME-FIRST ALGORITHM

EARLIEST-START-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT lectures by start times and renumber so that $s_1 \leq s_2 \leq \ldots \leq s_n$.

$d \leftarrow 0$. ⟵ number of allocated classrooms

FOR $j = 1$ TO n

    IF (lecture j is compatible with some classroom)

        Schedule lecture j in any such classroom k.

    ELSE

        Allocate a new classroom $d + 1$.

        Schedule lecture j in classroom $d + 1$.

        $d \leftarrow d + 1$.

RETURN schedule.

***Lemma*.**

The earliest-start-time-first algorithm can be implemented in $O(n \log n)$ time.

***Lemma*.**

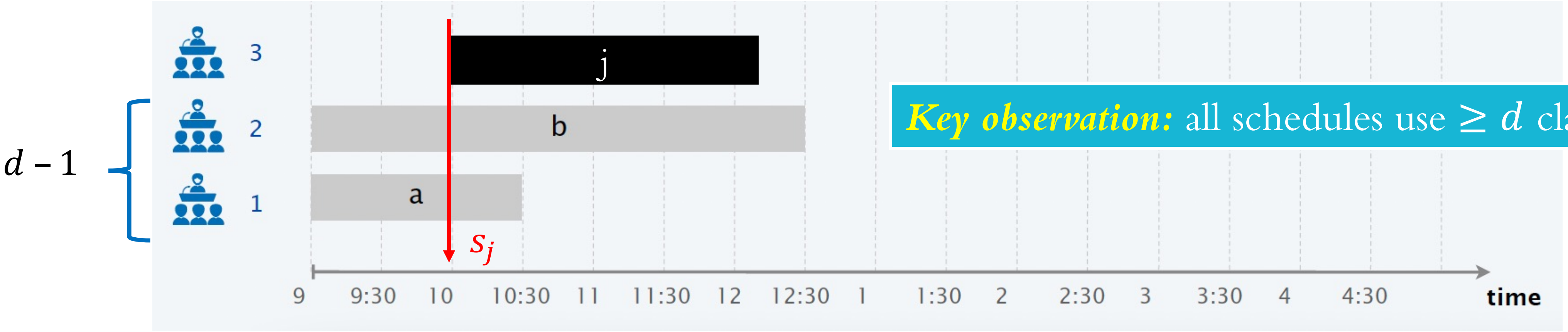
The earliest-start-time first algorithm never schedules two incompatible lectures in the same classroom.

INTERVAL PARTITIONING: EARLIEST-START-TIME-FIRST ALGORITHM

Theorem. Earliest-start-time-first algorithm uses #depth rooms and thus is optimal.

- Let d = number of classrooms that the algorithm allocates.
- Classroom d is opened because we needed to schedule a lecture, say j, that is incompatible with a lecture in each of $d-1$ other classrooms.
- Thus, these d lectures each end after s_j.
- Since we sorted by start time, each of these incompatible lectures start no later than s_j. ■

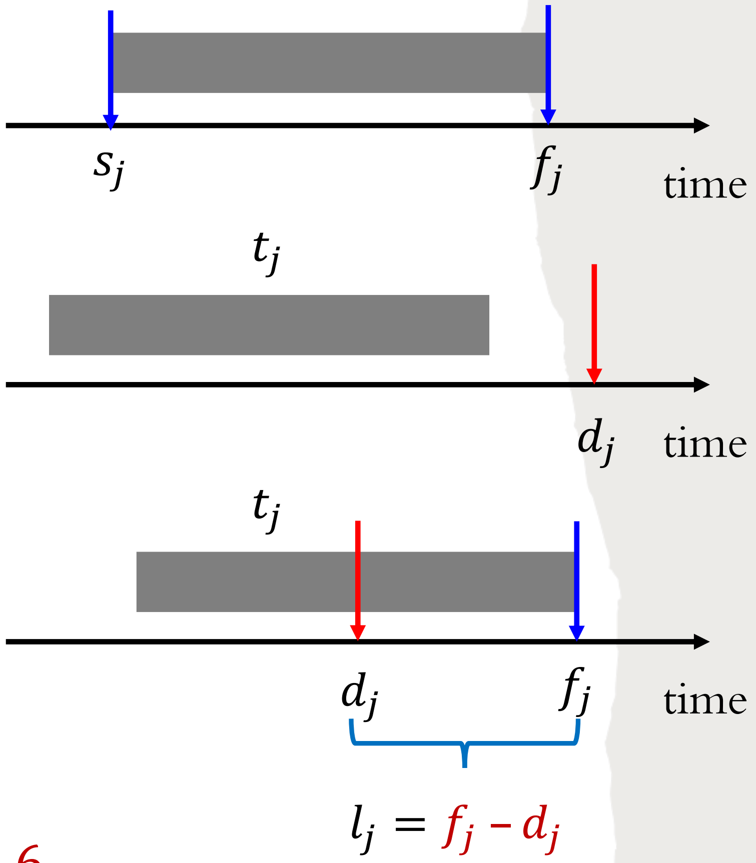
The d lectures are incompatible.



Key observation: all schedules use $\geq d$ classrooms.

SCHEDULING TO MINIMIZING LATENESS

SCHEDULING TO MINIMIZING LATENESS

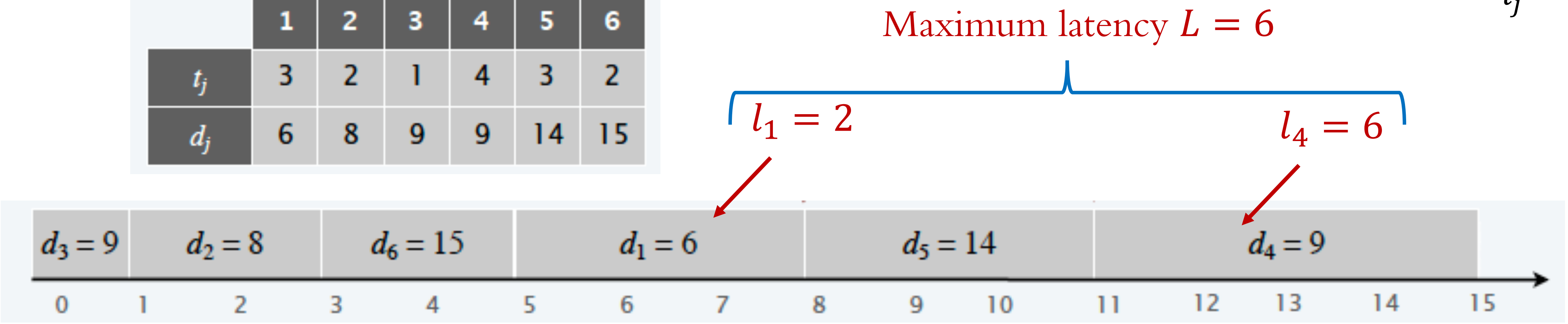


Single resource processes one job at a time.

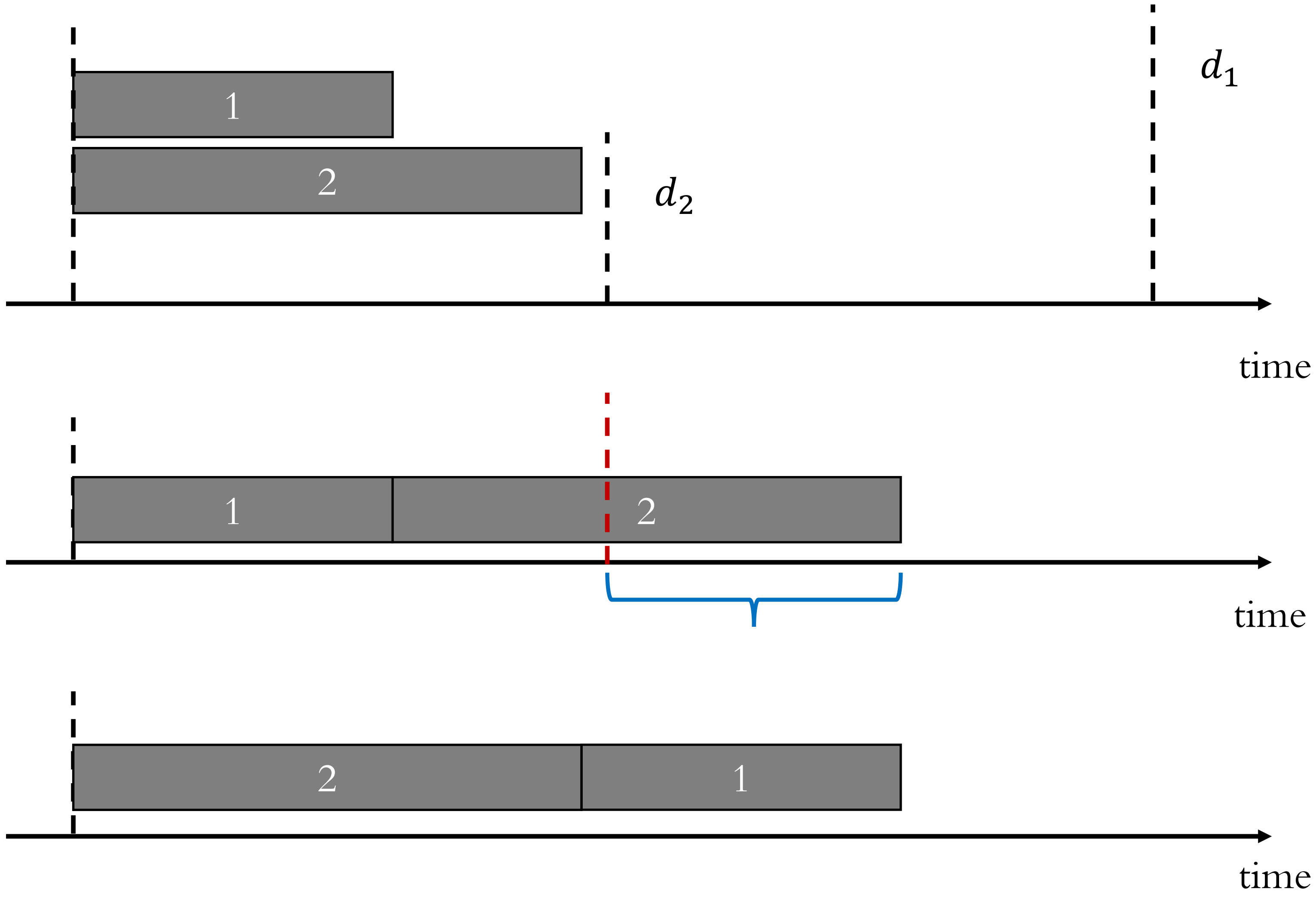
- Job j requires t_j units of processing time and is due at time d_j.
- If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
- Lateness: $l_j = \max\{0, f_j - d_j\}$.

Goal: schedule all jobs to minimize maximum lateness $L = \max_j l_j$.

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



SCHEDULING TO MINIMIZING LATENESS



SCHEDULING TO MINIMIZING LATENESS

EARLIEST-DEADLINE-FIRST $(n, t_1, t_2, \ldots, t_n, d_1, d_2, \ldots, d_n)$

SORT jobs by due times and renumber so that $d_1 \le d_2 \le \ldots \le d_n$.

$t \leftarrow 0$.

FOR $j = 1$ TO n ← Process the ordered jobs one by one (immediately)

  Assign job j to interval $[t, t + t_j]$.

  $s_j \leftarrow t$; $f_j \leftarrow t + t_j$.

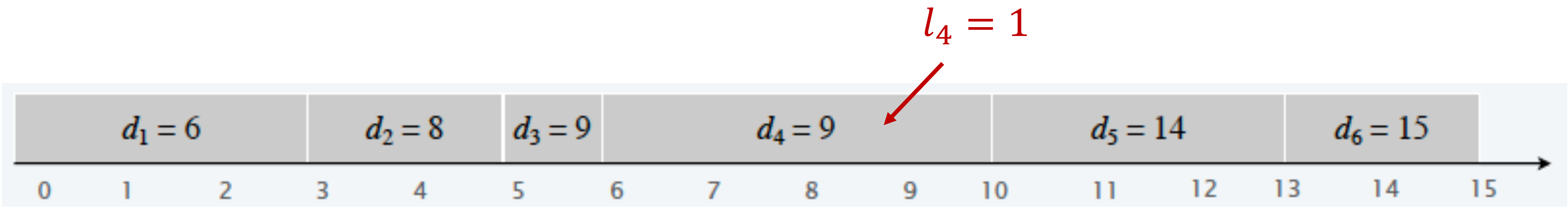
  $t \leftarrow t + t_j$.

RETURN intervals $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$.

$\Downarrow$

t

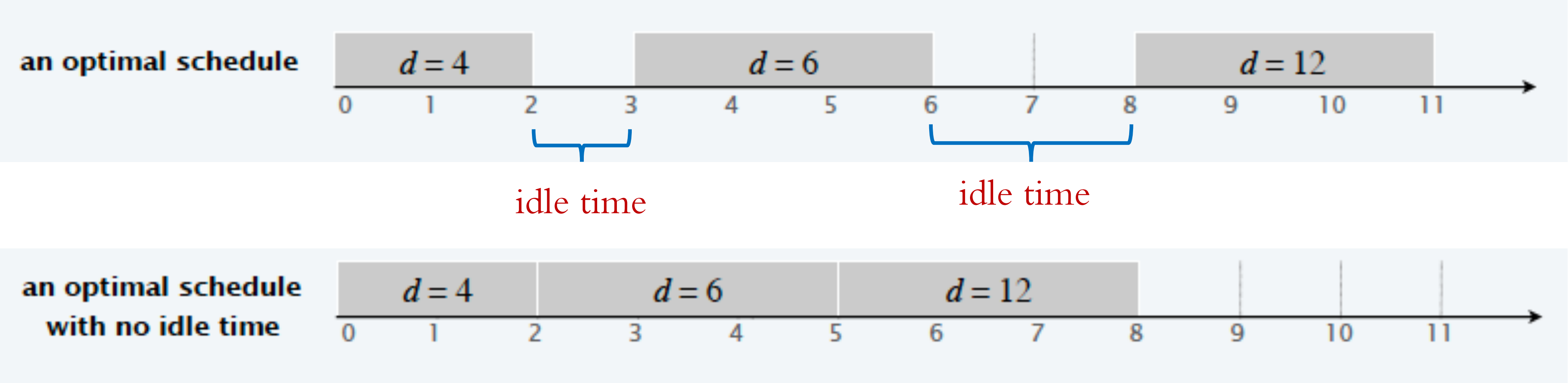
	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



SCHEDULING TO MINIMIZING LATENESS

Properties for optimal schedules.

Observation 1. There exists an optimal schedule with no idle time.

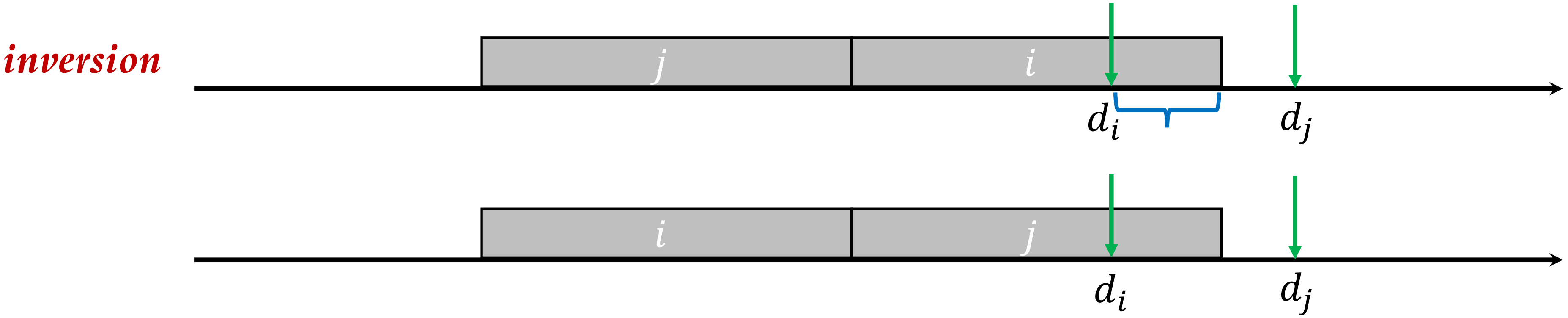


Observation 2. The earliest-deadline-first schedule has no idle time.

SCHEDULING TO MINIMIZING LATENESS

or $i < j$ for ordered jobs

Definition. Given a schedule S, an inversion is a pair of jobs i and j such that: $d_i < d_j$ but j is scheduled before i.



swap makes the schedule better!

Observation 3. The earliest-deadline-first schedule is the ***unique*** idle-free schedule with no inversions.

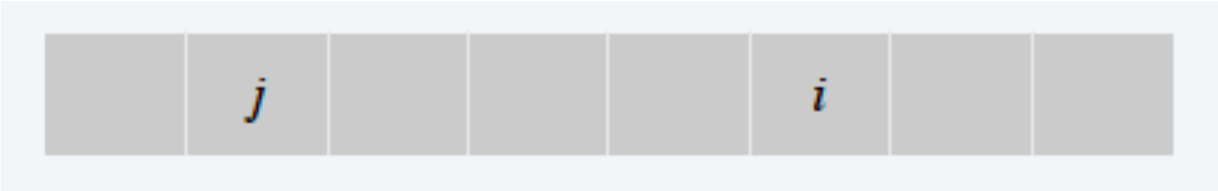
SCHEDULING TO MINIMIZING LATENESS

Observation 4. If an idle-free schedule has an inversion, then it has an adjacent inversion.

two inverted jobs scheduled consecutively

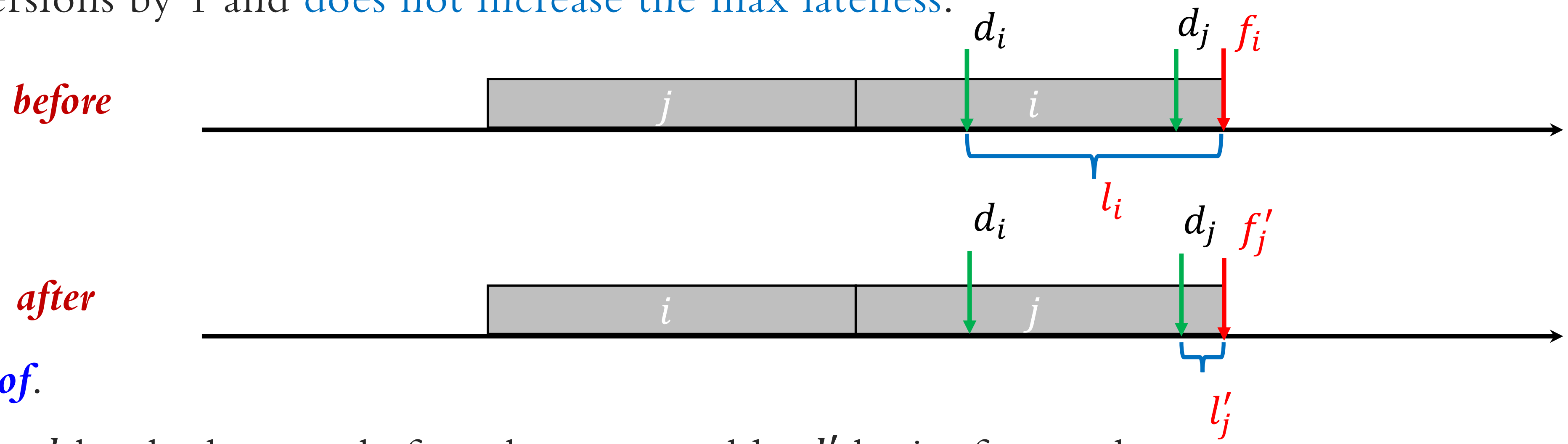
Proof.

- Let i–j be a closest inversion. $d_j > d_i$
- Let k be element immediately to the right of j.
  - Case 1: $d_j > d_k$. Then $j - k$ is an adjacent inversion.
  - Case 2. $d_j < d_k$. Then $i - k$ is a closer inversion. ■



SCHEDULING TO MINIMIZING LATENESS

Key Claim. Exchanging two adjacent, inverted jobs i and j reduces the number of inversions by 1 and does not increase the max lateness.



Proof.

- Let l be the lateness before the swap, and let l' be it afterwards.
- $l_k' = l_k$ for all $k \neq i, j$.
- $l_i' \leq l_i$

$f_j' = f_i$ $\qquad$ $i < j$: $d_i \leq d_j$

- If job j is late, $l_j' = f_j' - d_j = f_i - d_j \leq f_i - d_i \leq l_i$. ■

SCHEDULING TO MINIMIZING LATENESS

Theorem. The earliest-deadline-first schedule S is optimal.

Proof. [by contradiction]

- Define S^* to be an optimal schedule with the fewest inversions.
- Can assume S^* has no idle time. → Observation 1
- Case 1: S^* has no inversions. Then $S = S^*$. → Observation 3
- Case 2: S^* has an inversion.
  - Let $i - j$ be an adjacent inversion → Observation 4
  - Exchanging jobs i and j decreases the number of inversions by 1 without increasing the max lateness → Key Claim
  - Contradicts "fewest inversions" part of the 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.
- [Interval scheduling]

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.
- [Interval partitioning]

Exchange argument.

- Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.
- [Minimizing lateness, Interval scheduling]

Thank You!