4.1 Interval Partitioning

Proof Technique 2: “Structural”

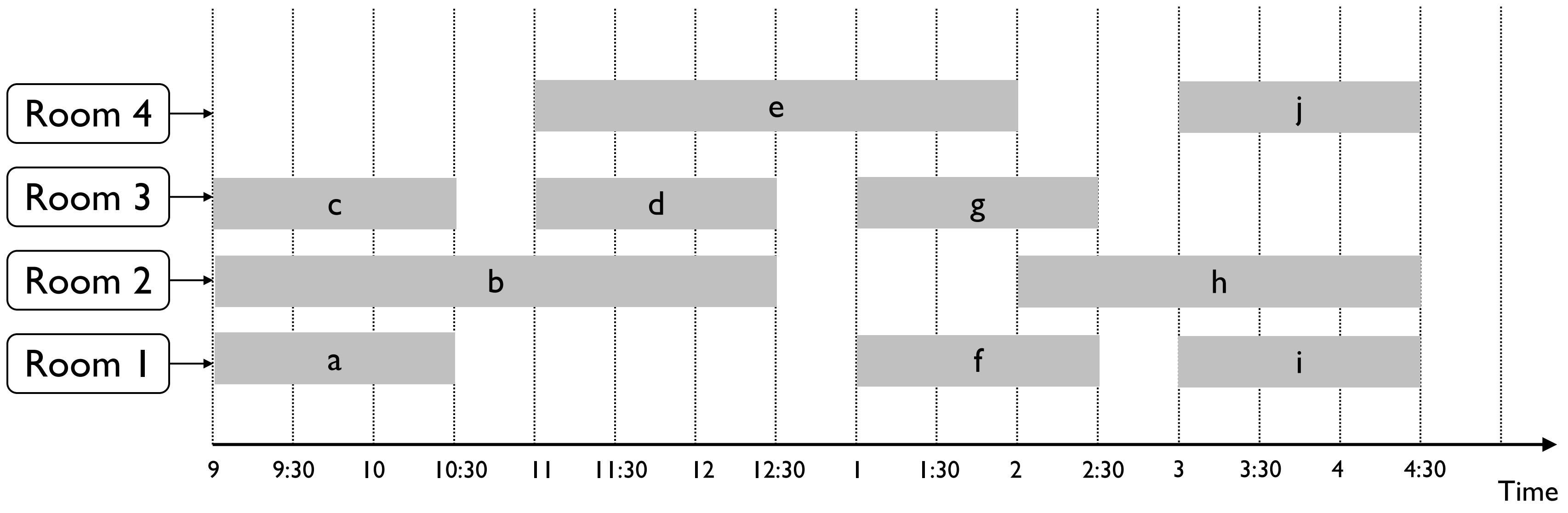
START WED JAN 23

Interval Partitioning

Interval partitioning.

- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

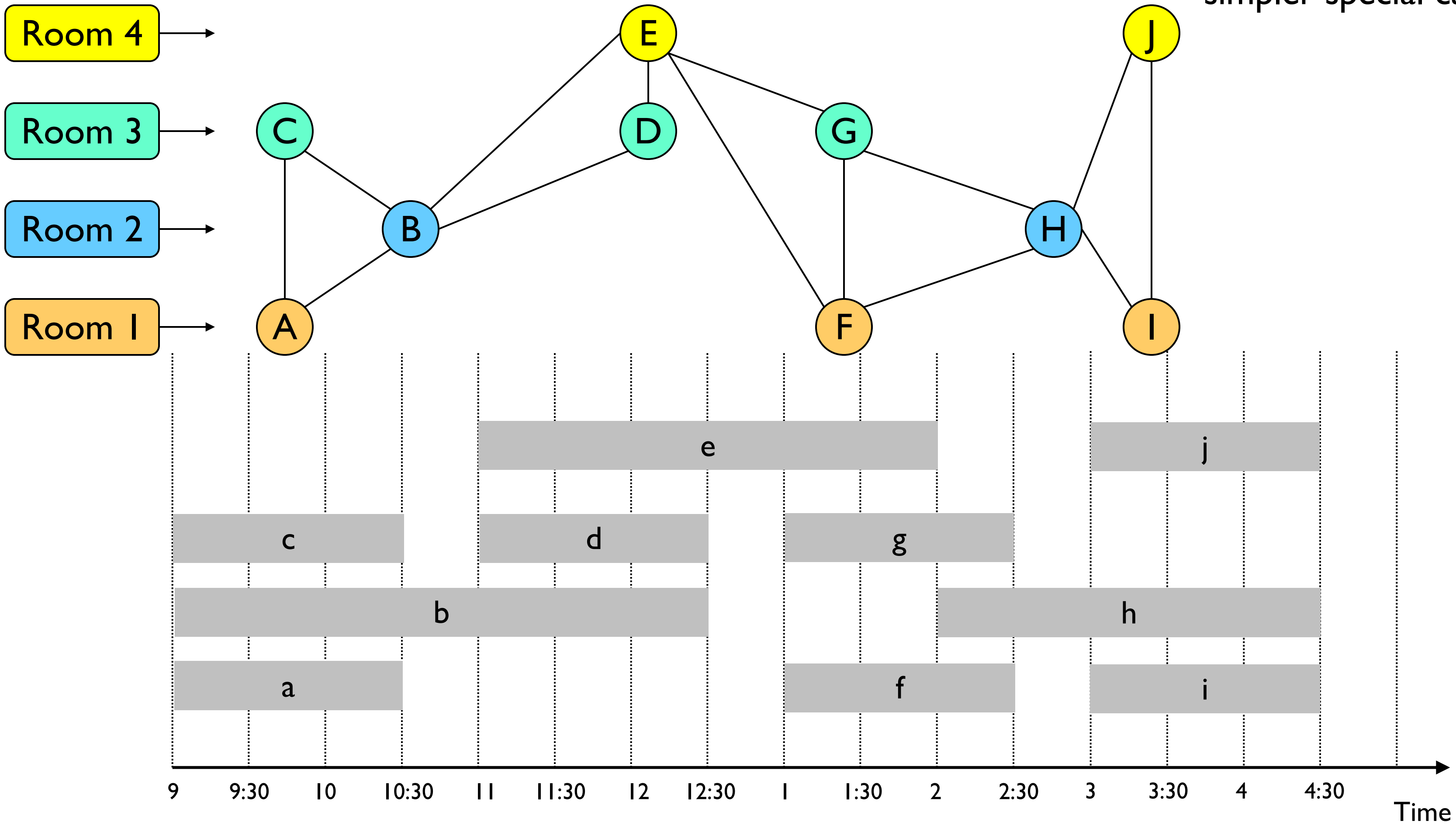
Ex: This schedule uses 4 classrooms to schedule 10 lectures.



Interval Partitioning as Interval Graph Coloring

Vertices = classes;
Edges = conflicting class pairs;
Different colors = different assigned rooms

Note: graph coloring is very hard in general, but graphs corresponding to interval intersections are a much simpler special case.

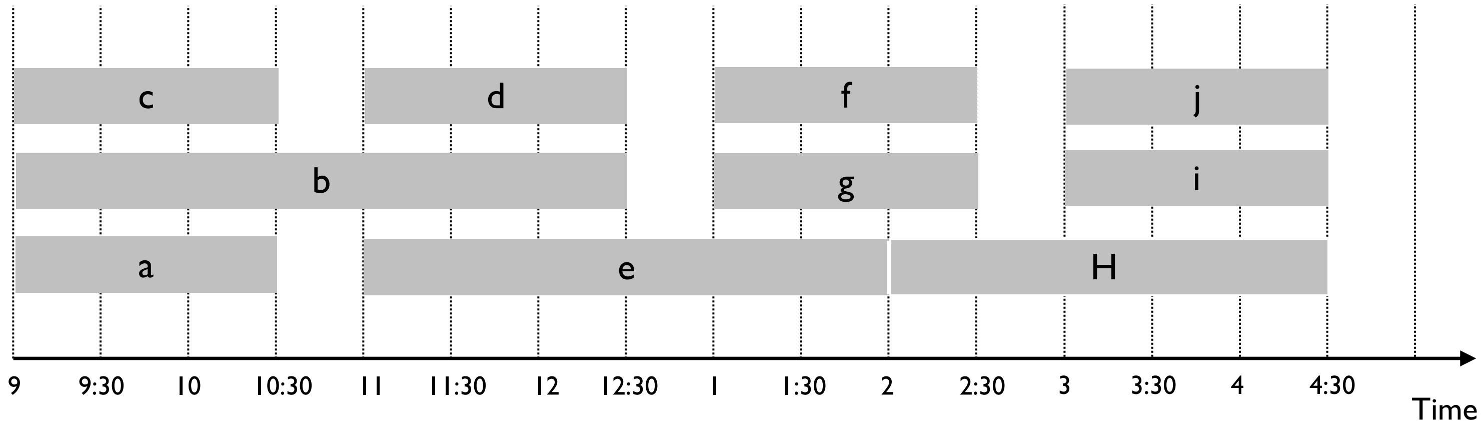


Interval Partitioning

Interval partitioning.

- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Ex: Same classes, but this schedule uses only 3 rooms.



Interval Partitioning: A "Structural" Lower Bound on Optimal Solution

max number of classes/intervals occuring at a specific time. Provides a lower bound for number of rooms we will need.

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

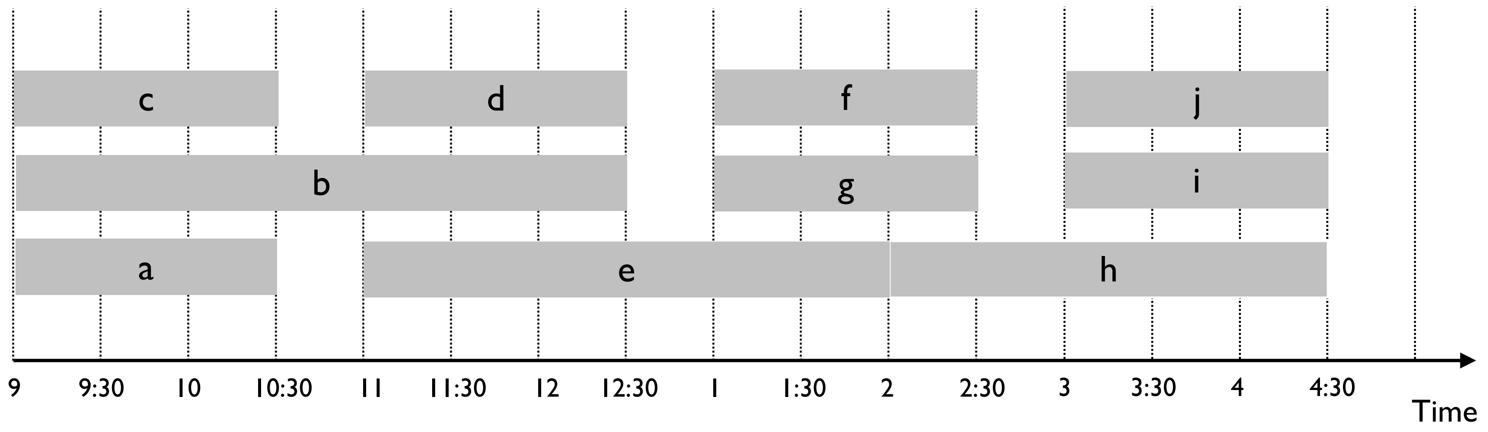
no collisions at ends

Key observation. **Number of classrooms needed $\geq$ depth.**

Ex: Depth of schedule below = 3 $\Rightarrow$ schedule is optimal.

e.g., a, b, c all contain 9:30

Q. Does a schedule equal to depth of intervals always exist?



Interval Partitioning: Earliest Start First Greedy Algorithm

Greedy algorithm. Consider lectures *in increasing order of start time*: assign lecture to any compatible classroom.

```
Sort intervals by start time so $s_1 \leq s_2 \leq \ldots \leq s_n$.
d ← 0   ← number of allocated classrooms

for j = 1 to n {
   if (lect j is compatible with some room k, 1≤k≤d)
      schedule lecture j in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture j in classroom d + 1
      d ← d + 1
}
```

Implementation? Run-time?
Exercises

Interval Partitioning: Greedy Analysis

Observation. Earliest Start First Greedy algorithm never schedules two incompatible lectures in the same classroom.

Theorem. Earliest Start First Greedy algorithm is optimal.
Pf (exploit structural property).

- Let d = number of rooms the greedy algorithm allocates.
- Classroom d is opened because we needed to schedule a job, say j, that is incompatible with all d-1 previously used classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_j.
- Thus, d lectures overlap at time $s_j + \varepsilon$, i.e. depth $\geq$ d
- "Key observation" on earlier slide $\Rightarrow$ all schedules use $\geq$ depth rooms, so d = depth and greedy is optimal

4.2 Scheduling to Minimize Lateness

Proof Technique 3: “Exchange” Arguments

Scheduling to Minimize Lateness

Minimizing lateness problem.

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

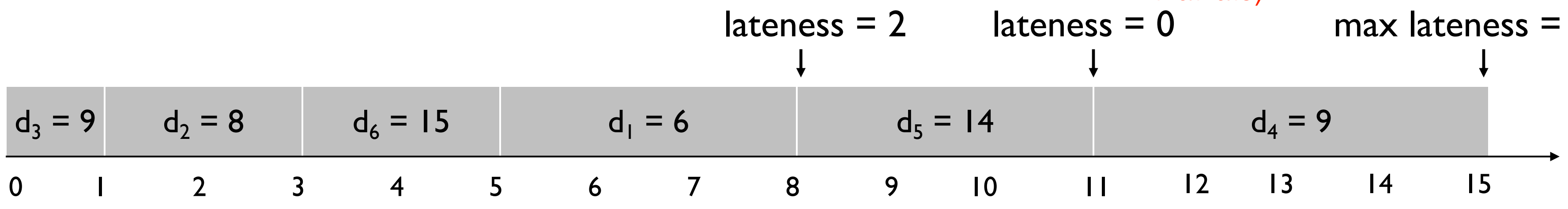
Ex:

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

time to complete job

due date

minimize the maximum lateness of a job (NOT necessarily the total lateness ALL jobs could be late if they all werent that late)



END LECT WED JAN 23 START LECT FRI JAN 25

Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

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

time	1	100
due	1000	100

clearly choosing to do job 1 leads to a late max of 1 while choosing job 2 first has no lateness

☆ [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 job first] Consider in ascending order of processing time t_j.

	1	2
t_j	1	10
d_j	100	10

counterexample

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

	1	2
t_j	1	10
d_j	2	10

counterexample

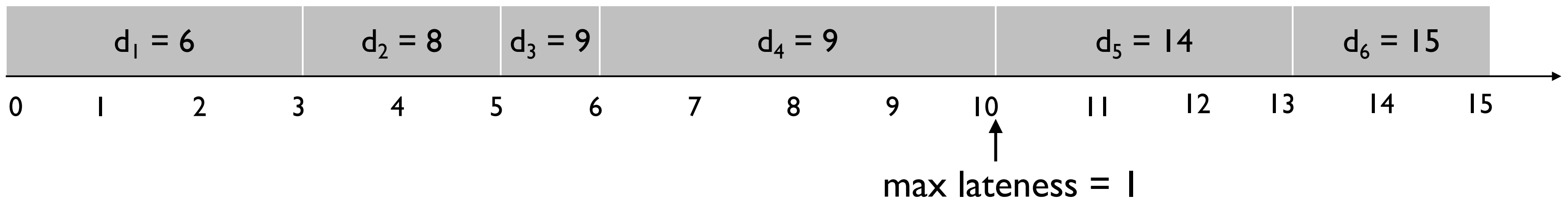
Minimizing Lateness: Greedy Algorithm

Greedy algorithm. Earliest deadline first.

```
Sort n jobs by deadline so that $d_1 \le d_2 \le … \le d_n$

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



Proof Strategy

A *schedule* is an ordered list of jobs

Suppose S_1 is any schedule & let G be the/a the greedy algorithm's schedule

To show: Lateness(S_1) ≥ Lateness(G)

Idea: find simple changes that successively transform S_1 into other schedules increasingly like G, each better (or at least no worse) than the last, until we reach G. I.e.

Lateness(S_1) ≥ Lateness(S_2) ≥ Lateness(S_3) ≥ … ≥ Lateness(G)

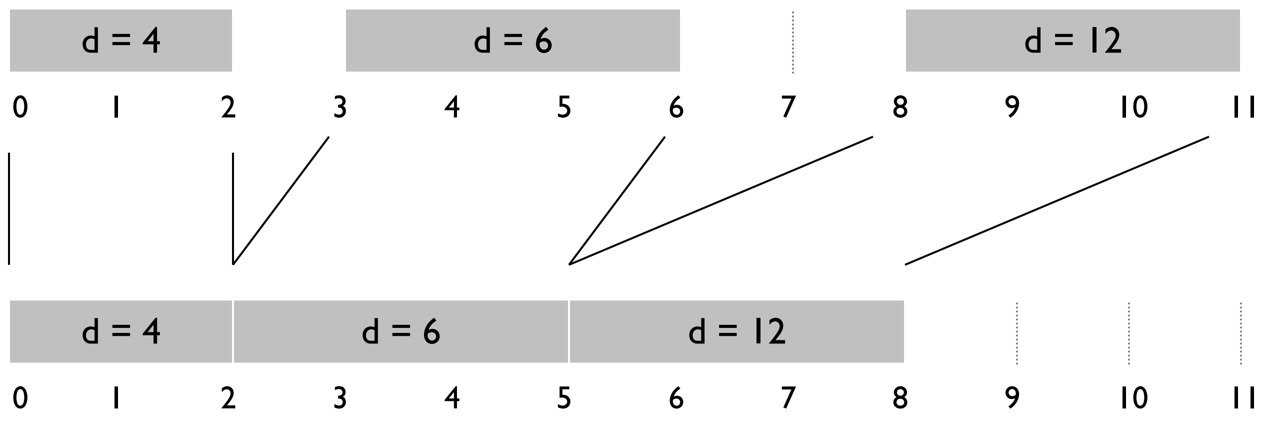
If it works for *any* S_1, it will work for an *optimal* S_1, so G is optimal

HOW?: *exchange* pairs of jobs

Minimizing Lateness: No Idle Time

Notes:

1. There is an optimal schedule with no idle time.

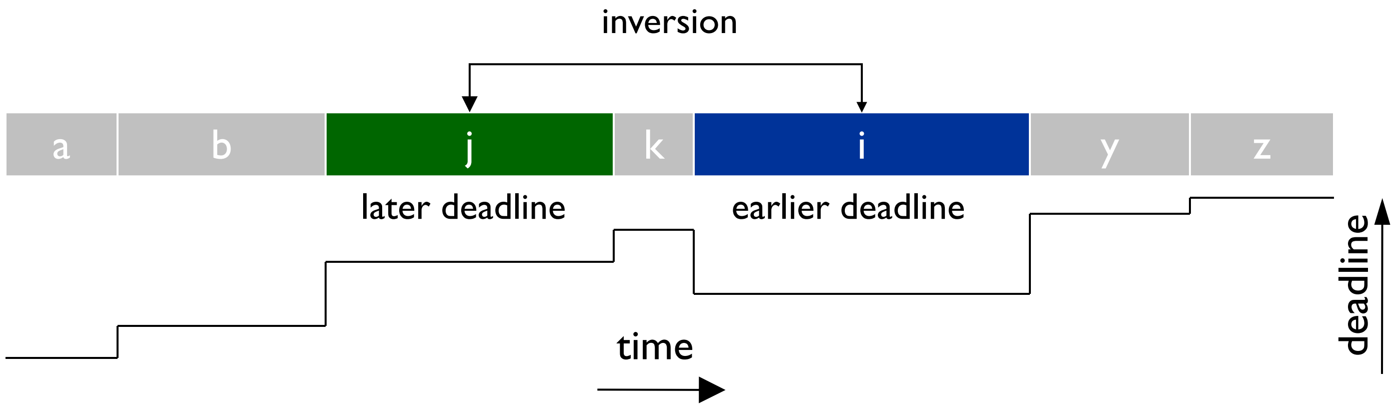


2. The greedy schedule has no idle time.

Minimizing Lateness: Inversions

i.e. a pair of jobs where you did job j before i even though j has the later deadline

Def. An *inversion* in schedule S is a pair of jobs i and j s.t.: deadline i < deadline j but j scheduled before i.

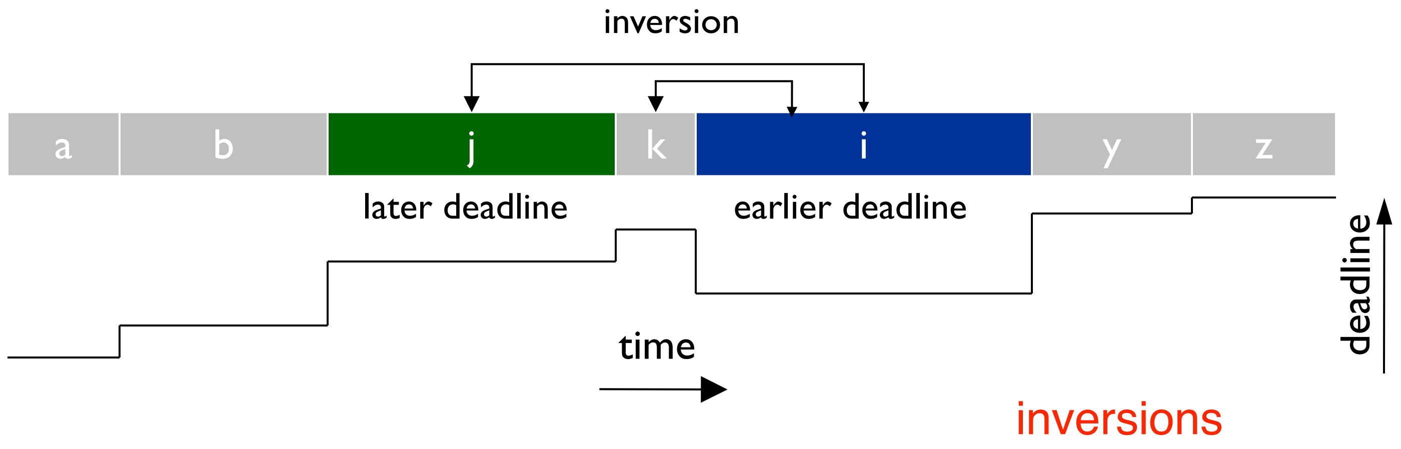


- Greedy schedule has no inversions.
- Claim: If a schedule has an inversion, it has an adjacent inversion, i.e., a pair of inverted jobs scheduled consecutively.

  (Pf: If j & i aren't consecutive, then look at the job k scheduled right after j. If $d_k < d_j$, then (j,k) is a consecutive inversion; if not, then (k,i) is an inversion, & nearer to each other - repeat.)

Minimizing Lateness: Inversions

Def. An *inversion* in schedule S is a pair of jobs i and j s.t.: deadline i < deadline j but j scheduled before i.

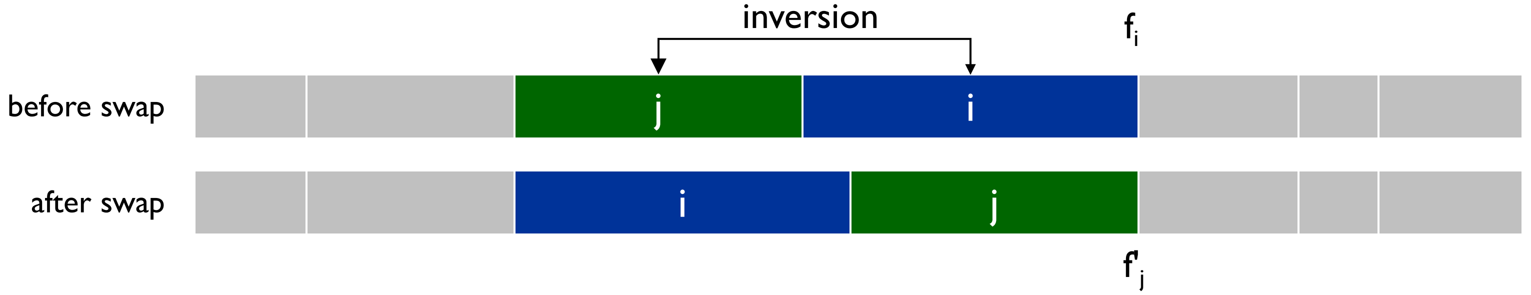


- Claim: Swapping an *adjacent* inversion reduces tot # invs by 1 (exactly)

Pf: Let i,j be an adjacent inversion. For any pair (p,q), inversion status of (p,q) is unchanged by i↔j swap unless {p, q} = {i, j}, and the i,j inversion is removed by that swap.

Minimizing Lateness: Inversions

Def. An *inversion* in schedule S is a pair of jobs i and j s.t.: deadline i < j but j scheduled before i.



(j had later deadline, so is less tardy than i was)

Claim. *Swapping* two adjacent, inverted jobs does not increase the max lateness.

swapping i forward reduces its lateness, and since j has a later deadline than i, moving it to where i was (it will finish at same time i used to finish) will not make it as late as i was

Pf. Let ℓ / ℓ' be the lateness before / after swap, resp.

- $\ell'_k = \ell_k$ for all $k \neq i, j$
- $\ell'_i \leq \ell_i$
- If job j is now late:

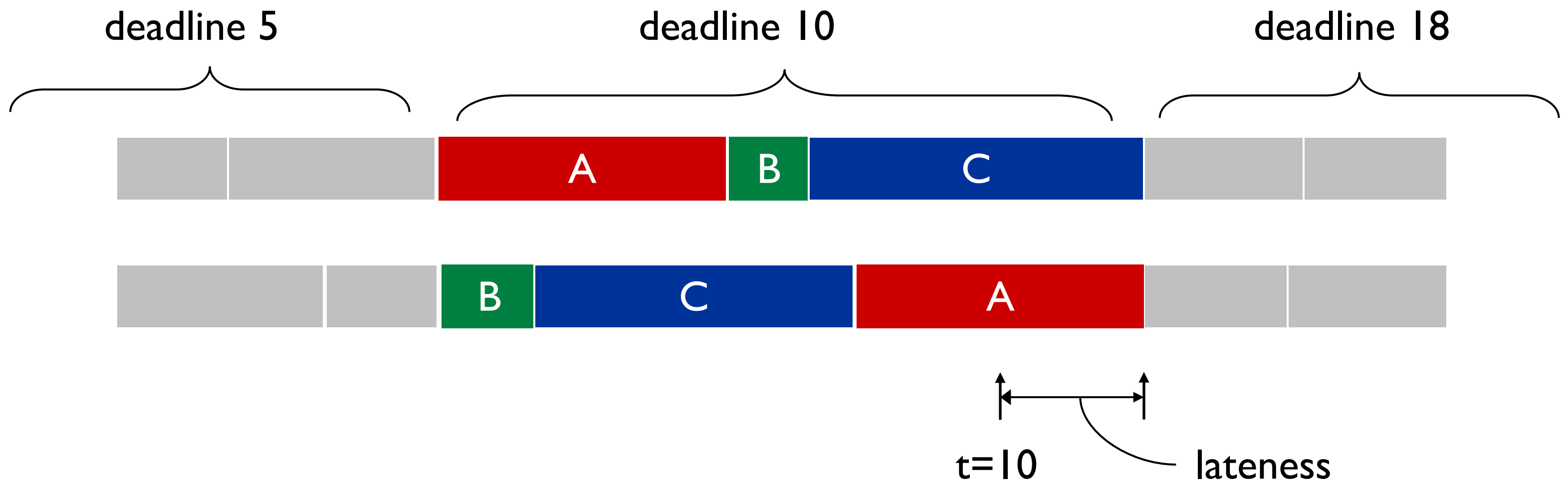
$$\begin{aligned} \ell'_j &= f'_j - d_j && \text{(definition)} \\ &= f_i - d_j && (j \text{ finishes at time } f_i) \\ &\leq f_i - d_i && (d_i \leq d_j) \\ &= \ell_i && \text{(definition)} \end{aligned}$$

only j moves later, but it's no later than i was, so max not increased

Minimizing Lateness: No Inversions

Claim. All idle-free, inversion-free schedules S have the same max lateness.

Pf. If S has no inversions, then deadlines of scheduled jobs are monotonically nondecreasing (i.e., increase or stay the same) as we walk through the schedule from left to right. Two such schedules can differ only in the order of jobs with the same deadlines. Within a group of jobs with the same deadline, the max lateness is the lateness of the last job in the group - order within the group doesn't matter.



Minimizing Lateness: Correctness of Greedy Algorithm

Theorem. Greedy schedule G is optimal

Pf. Let S* be an optimal schedule with the fewest number of inversions among all optimal schedules

Can assume S* has no idle time.

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*

So, S* has no inversions. Hence Lateness(G) = Lateness(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. (Part of the cleverness is deciding what’s “good.”)

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. (Cleverness here is usually in finding a useful structural characteristic.)

Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality. (Cleverness usually in choosing which pair to swap.)

(In all 3 cases, proving these claims may require cleverness, too.)

4.4 Shortest Paths in a Graph

You've seen this in prerequisite courses, so this section and next two on min spanning tree are review. I won't lecture on them, but you should review the material. Both, but especially shortest paths, are common problems, having many applications. (And, hint, hint, *very* frequent fodder for job interview questions…)

Shortest Path Problem

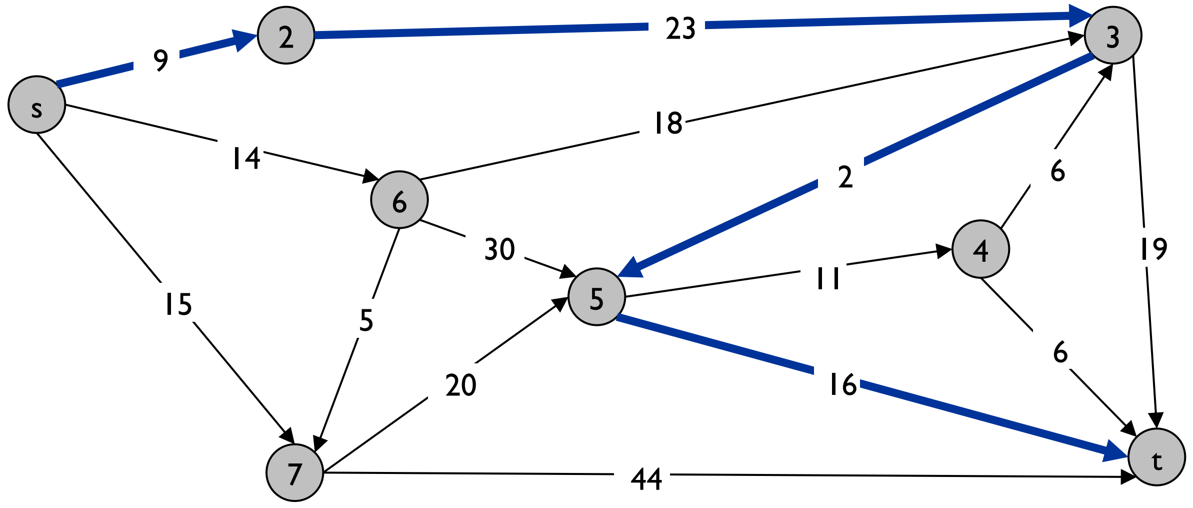
Shortest path network.

- Directed graph G = (V, E).
- Source s, destination t.
- Length ℓ_e = length of edge e.

Shortest path problem: find shortest directed path from s to t.

$\uparrow$

cost of path = sum of edge costs in path



Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 48.

Dijkstra's Algorithm

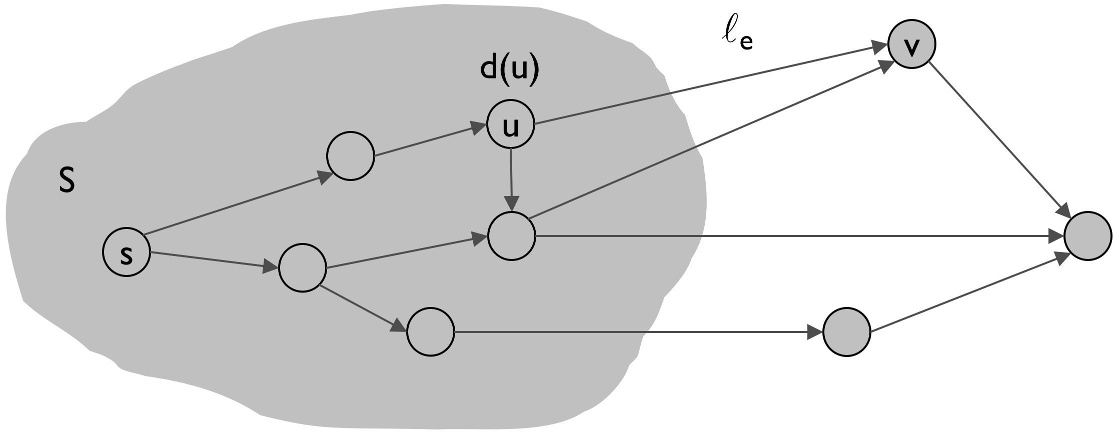
Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e ,$$

add v to S, and set d(v) = π(v).

shortest path to some u in explored part, followed by a single edge (u, v)



Dijkstra's Algorithm

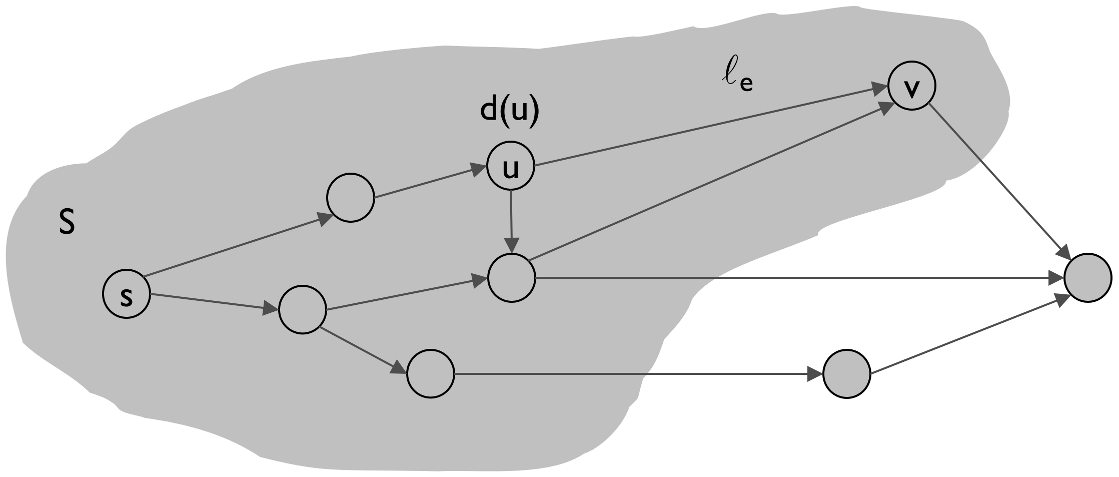
Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v)\,:\, u \in S} d(u) + \ell_e ,$$

  add v to S, and set d(v) = π(v).

shortest path to some u in explored part, followed by a single edge (u, v)



Summary

"Greedy" algorithms are natural, often intuitive, tend to be simple and efficient

But seductive – often incorrect!

E.g., "Change making," depending on the available denominations

So, we look at a few examples, each useful in its own right, but emphasize *correctness,* and various approaches to reasoning about these algorithms

Interval Scheduling – greedy stays ahead

Interval Partitioning – greedy matches structural lower bound

Minimizing Lateness – exchange arguments

Next: Huffman codes and another exchange argument

Also: This is a good time to review shortest paths and min spanning trees