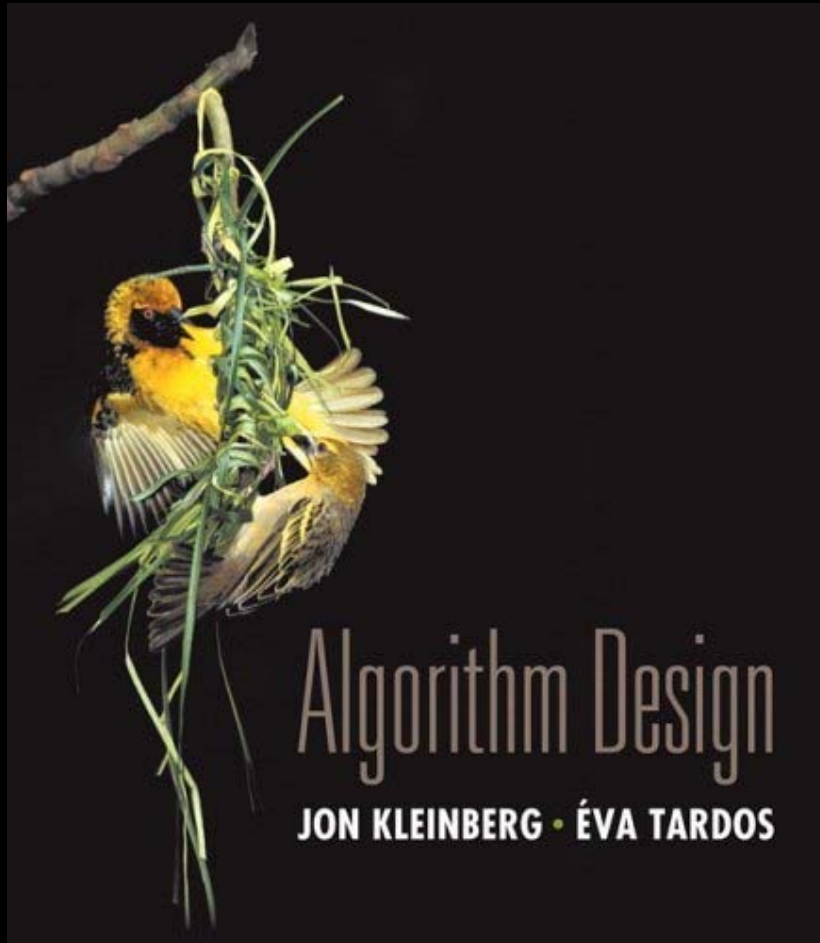




Chapter 4

Greedy Algorithms



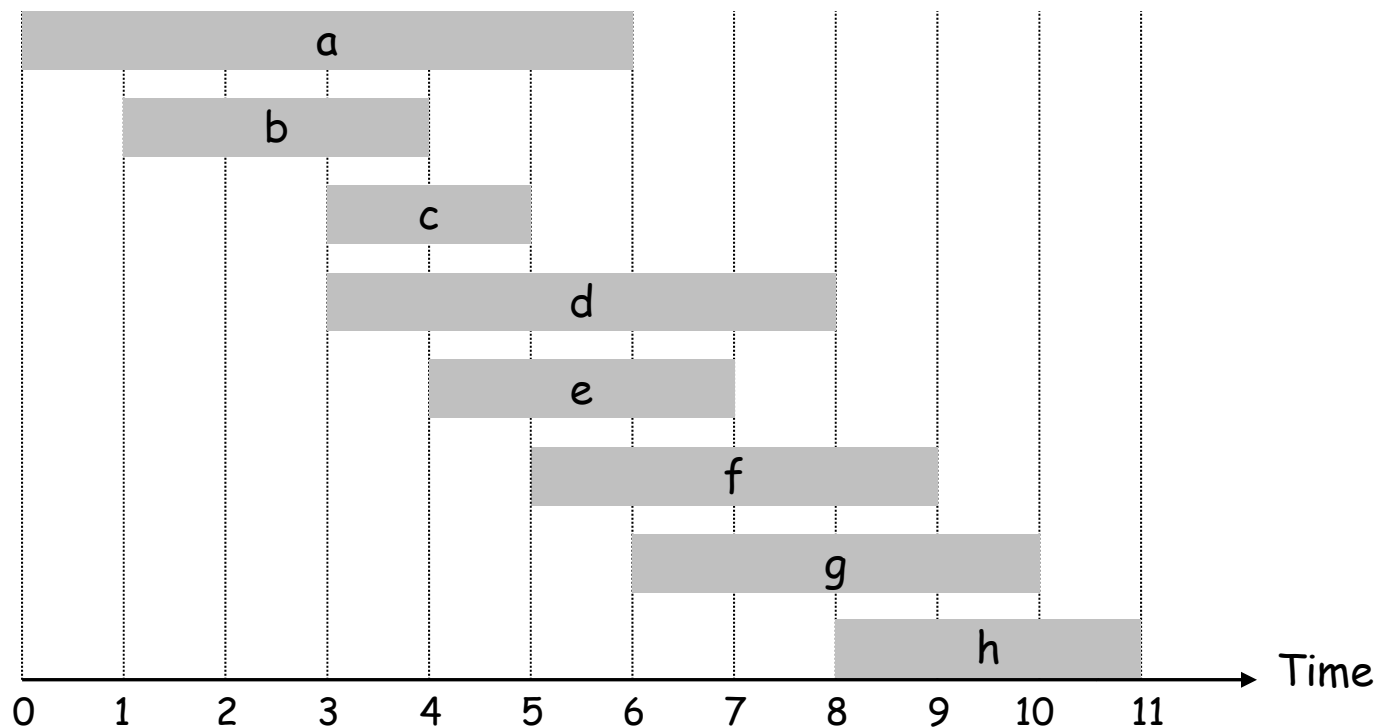
Slides by Kevin Wayne.
Copyright © 2005 Pearson-Addison Wesley.
All rights reserved.
Updated by T. K. Prasad

4.1 Interval Scheduling

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 subset of mutually compatible jobs.



Interval Scheduling: Greedy Algorithms

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

- **[Earliest start time]**

Consider jobs in ascending order of start time s_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 .

- **[Earliest finish time]**

Consider jobs in ascending order of finish time f_j .

Interval Scheduling: Greedy Algorithms

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



breaks earliest start time



breaks shortest interval



breaks 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 $f_1 \leq f_2 \leq \dots \leq f_n$.

↙ jobs selected
 $A \leftarrow \phi$
for $j = 1$ to n {
 if (job j compatible with A)
 $A \leftarrow A \cup \{j\}$
}
return A



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

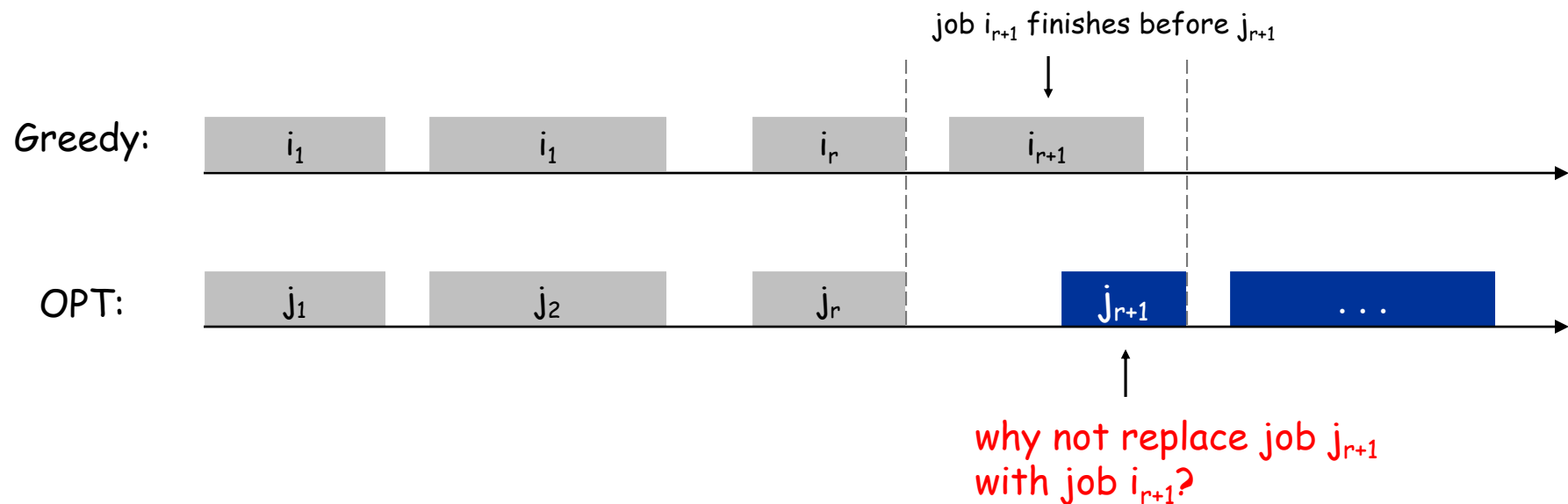
- Remember job j^* that was added last to A .
- Job j is compatible with A if $s_j \geq f_{j^*}$.

Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal.
- Let $i_1, i_2, \dots, i_k$ denote set of jobs selected by greedy algorithm.
- 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 .

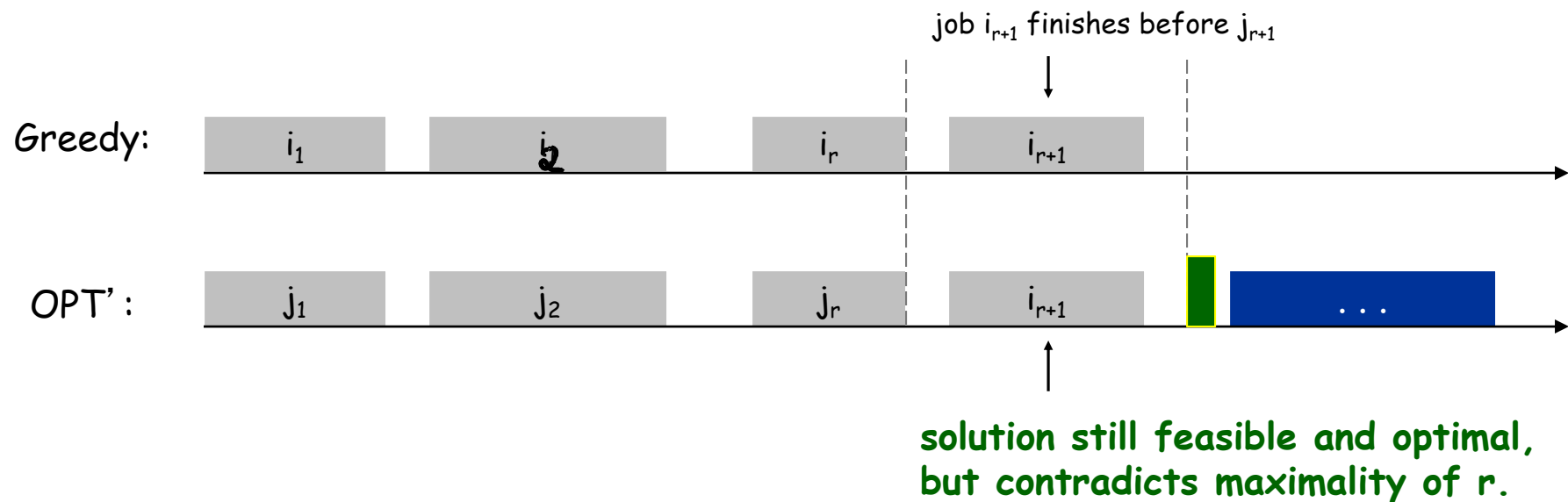


Interval Scheduling: Analysis

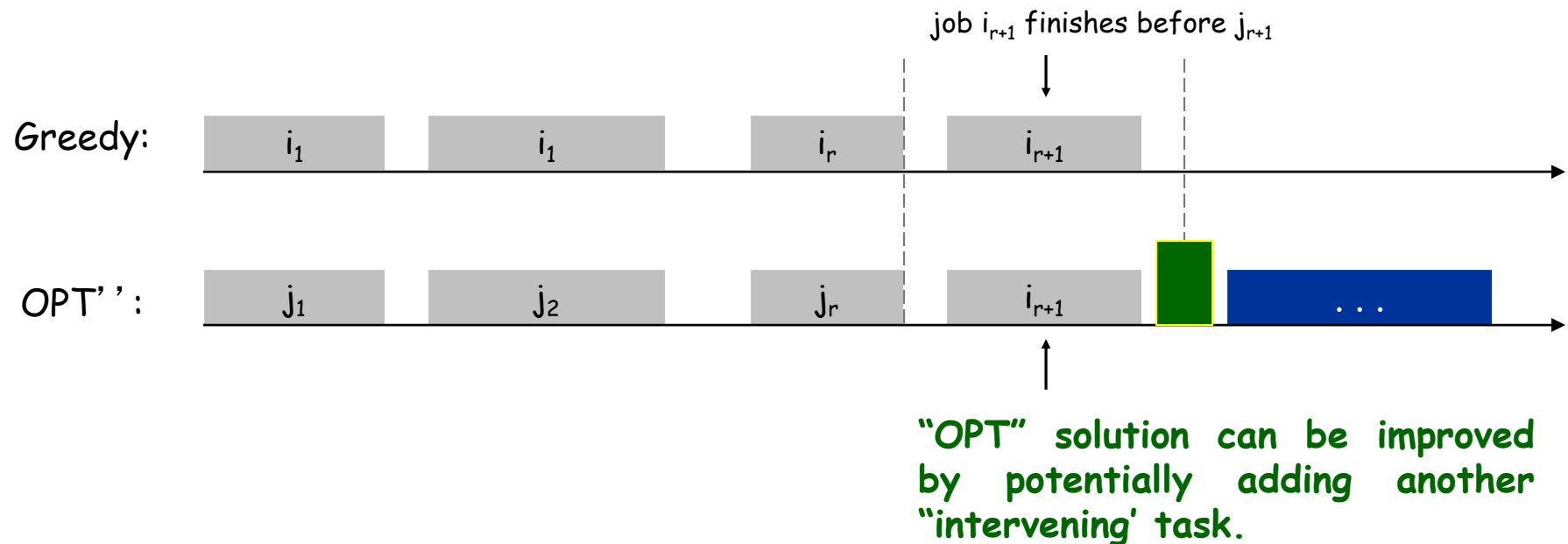
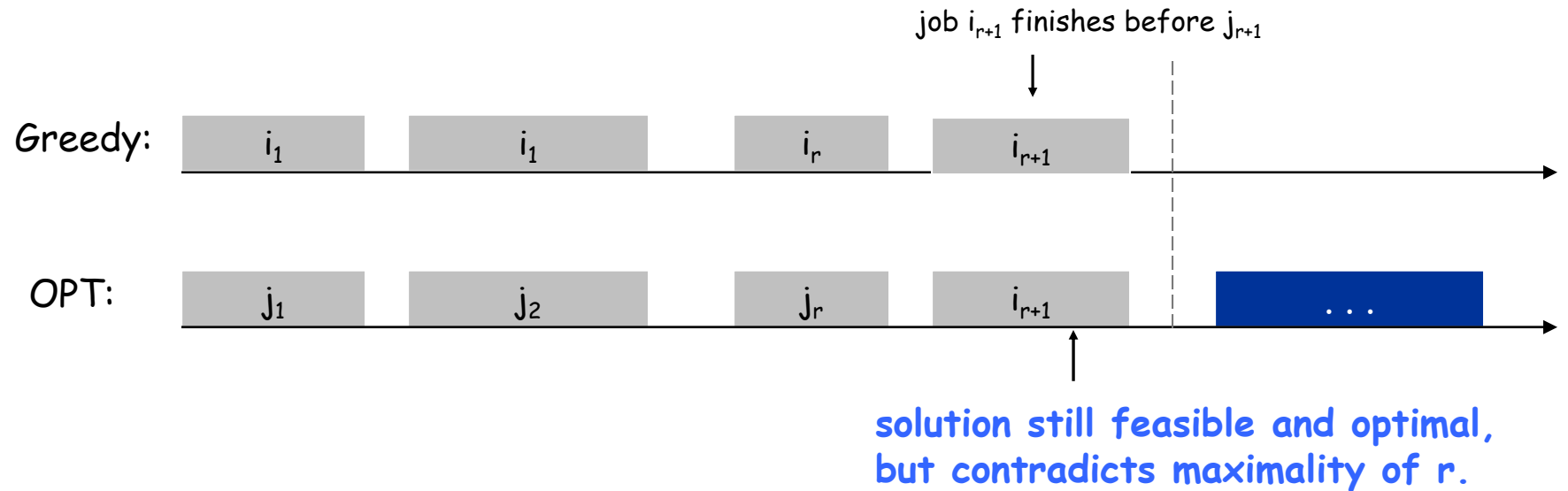
Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal.
- 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



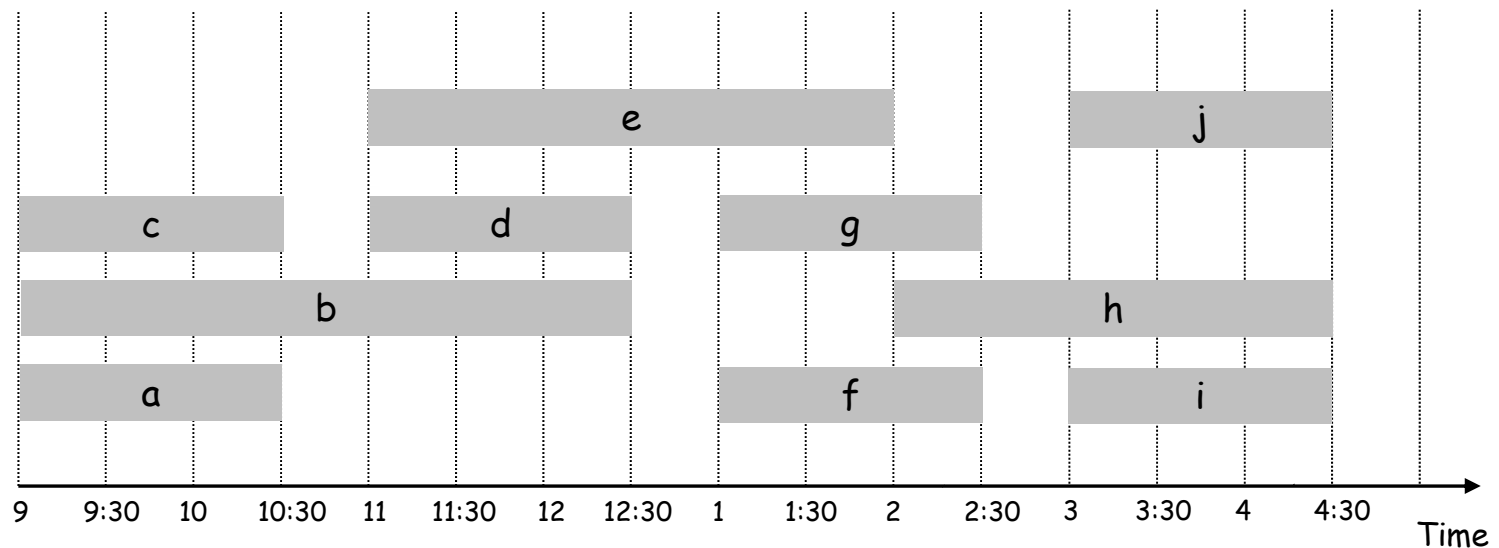
4.1 Interval Partitioning

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.

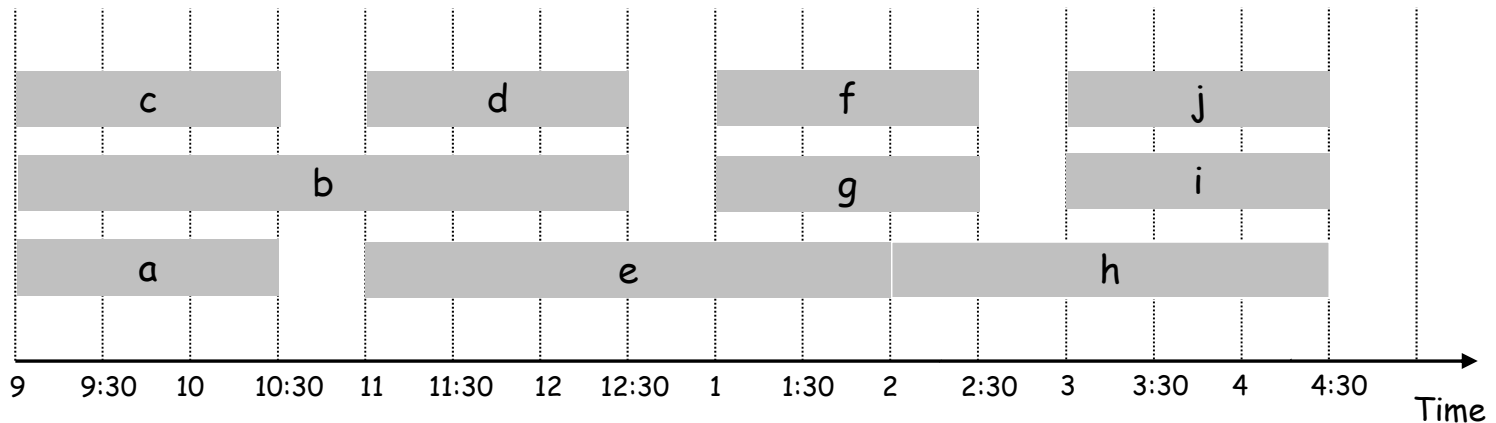


Interval Partitioning

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 only 3 classrooms.



Interval Partitioning: Lower Bound on Optimal Solution

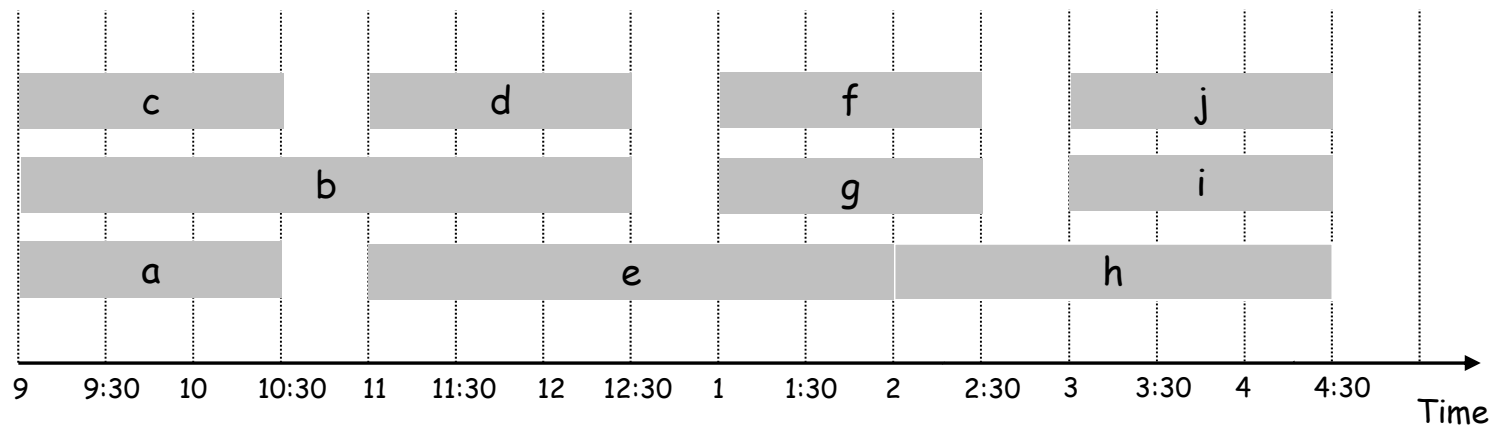
Def. The **depth** of a set of open intervals is the maximum number of intervals that contain any given time.

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

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

↑
a, b, c all contain 9:30

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



Interval Partitioning: Greedy Algorithm

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

```
Sort intervals by starting time so that  $s_1 \leq s_2 \leq \dots \leq s_n$ .  
 $d \leftarrow 0$   $\leftarrow$  number of allocated classrooms  
  
for  $j = 1$  to  $n$  {  
    if (lecture  $j$  is compatible with some classroom  $k$ )  
        schedule lecture  $j$  in the classroom  $k_{\text{best}}$   
        with lowest finish time  
    else  
        allocate a new classroom  $d + 1$   
        schedule lecture  $j$  in classroom  $d + 1$   
         $d \leftarrow d + 1$   
}
```

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

- For each classroom k , maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.

Interval Partitioning: Greedy Analysis

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

Theorem. Greedy algorithm is optimal.

Pf.

- Let d = number of classrooms that the greedy algorithm allocates.
- Classroom d is opened because we needed to schedule a job, say j , that is incompatible with all $d-1$ other classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_j .
- Thus, we have d lectures overlapping at time $s_j + \epsilon$.
- **Key observation $\Rightarrow$**
all schedules use $\geq d$ classrooms. ▪

Interval Partitioning: Greedy Algorithm

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

```
Sort intervals by starting time so that  $s_1 \leq s_2 \leq \dots \leq s_n$ .  
 $d \leftarrow 0$   $\leftarrow$  number of allocated classrooms  
  
for  $j = 1$  to  $n$  {  
    if (lecture  $j$  is compatible with some classroom  $k$ )  
        schedule lecture  $j$  in classroom  $k$   
    else  
        allocate a new classroom  $d + 1$   
        schedule lecture  $j$  in classroom  $d + 1$   
         $d \leftarrow d + 1$   
}
```

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

- For each classroom k , maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.

Interval partitioning: earliest-start-time-first algorithm

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

Pf. Store classrooms in a **priority queue** (key = finish time of its last lecture).

- To determine whether lecture j is compatible with some classroom, compare s_j to key of min classroom k in priority queue.
- To add lecture j to classroom k , increase key of classroom k to f_j .
- Total number of priority queue operations is $O(n)$.
- Sorting by start time takes $O(n \log n)$ time. ■

Remark. This implementation chooses the classroom k whose finish time of its last lecture is the **earliest**.

4.2 Scheduling to Minimize Lateness

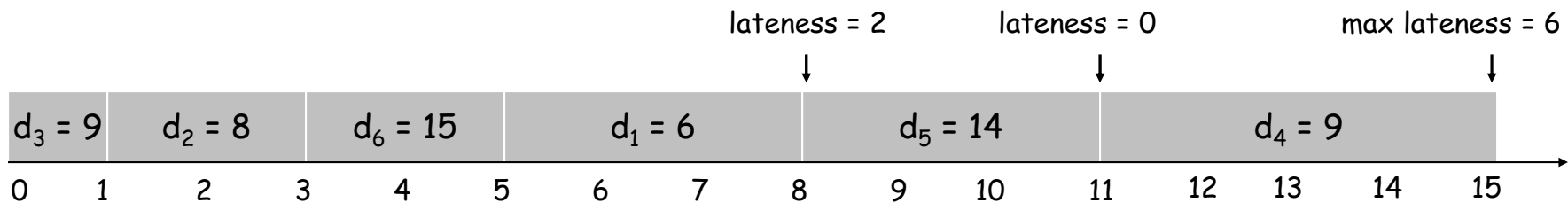
Scheduling to Minimize Lateness

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: $\ell_j = \max \{ 0, f_j - d_j \}$.
- Goal: schedule all jobs to minimize maximum lateness $L = \max \ell_j$.

Ex:

	1	2	3	4	5	6
$\dagger_j$	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 .
- [**Smallest slack**] Consider jobs in ascending order of slack $d_j - t_j$.
- [**Earliest deadline first**] Consider jobs in ascending order of deadline d_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

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

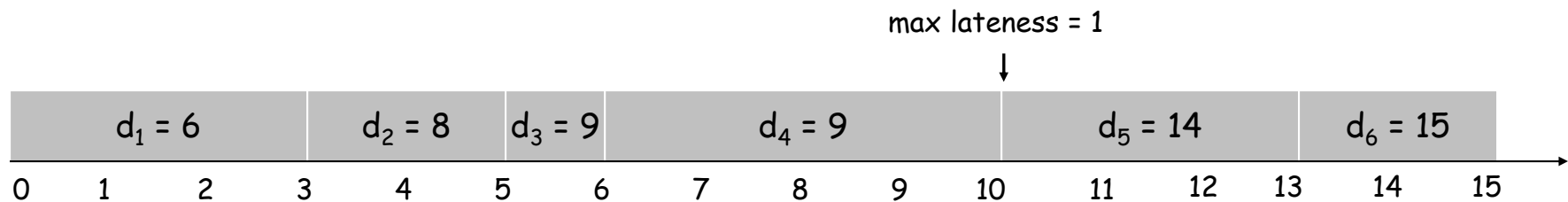
	1	2
t_j	1	10
d_j	2	10

counterexample

Minimizing Lateness: Greedy Algorithm

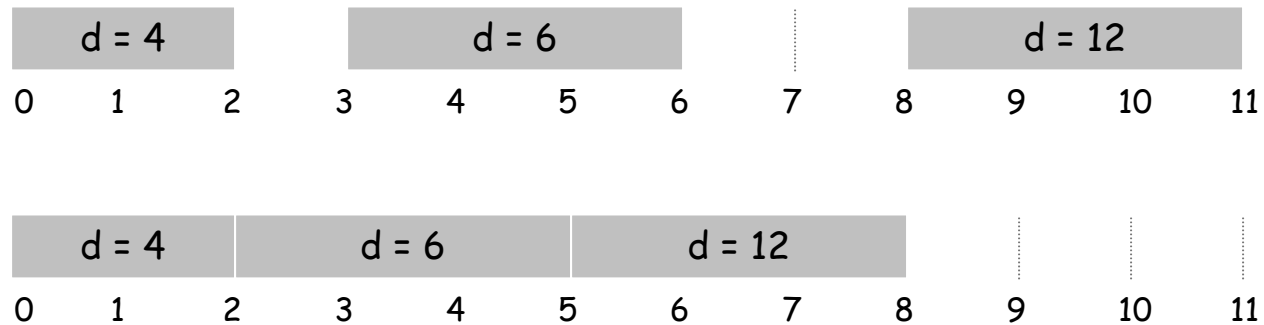
Greedy algorithm. Earliest deadline first.

```
Sort n jobs by deadline so that  $d_1 \leq d_2 \leq \dots \leq d_n$   
  
 $t \leftarrow 0$   
for  $j = 1$  to  $n$   
    Assign job  $j$  to interval  $[t, t + t_j]$   
     $s_j \leftarrow t, f_j \leftarrow t + t_j$   
     $t \leftarrow t + t_j$   
output intervals  $[s_j, f_j]$ 
```



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.

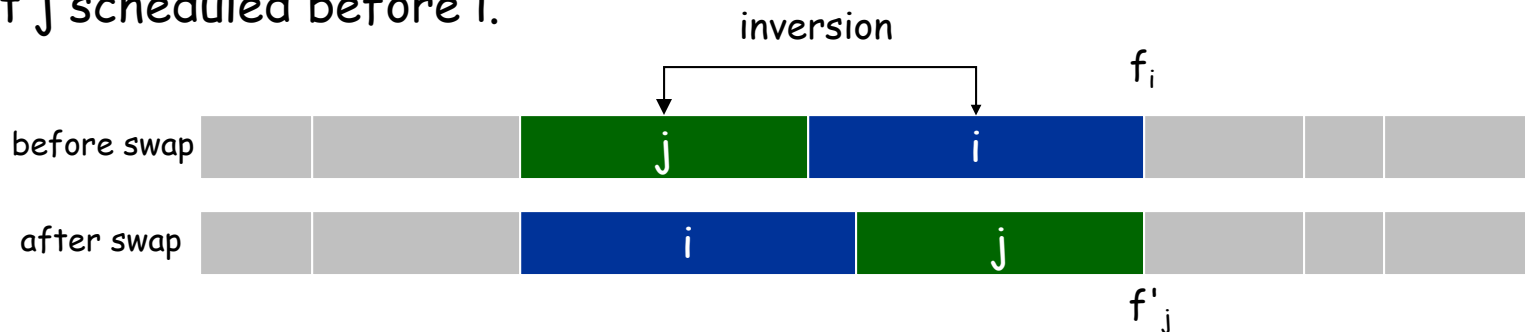
Observation. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

Example:

3			...			2
3	4	5	...	8	...	<u>8</u> 2
3	4	4	...	<u>8</u> 6	...	1 2

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 and ℓ' be the lateness before and after the swap resp.

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

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

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.

- 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^* ▪

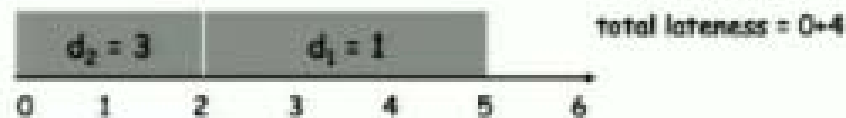
Variant: Scheduling to Minimizing *Total* Lateness

Minimizing total 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: $\ell_j = \max \{ 0, f_j - d_j \}$.
- Goal: schedule all jobs to **minimize total lateness** $L = \sum_j \ell_j$.

Ex:

	1	2	job number
t_j	3	2	time required
d_j	1	3	deadline



No polynomial algorithm can
compute optimal schedule
(unless $P=NP$)
(see Master's course Advanced Algorithms)

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.

4.3 Optimal Caching

Optimal Offline Caching

Caching.

- Cache with capacity to store k items.
- Sequence of m item requests $d_1, d_2, \dots, d_m$.
- **Cache hit:** item already in cache when requested.
- **Cache miss:** item not already in cache when requested: must bring requested item into cache, and evict some existing item, if full.

Goal. Eviction schedule that minimizes number of cache misses.

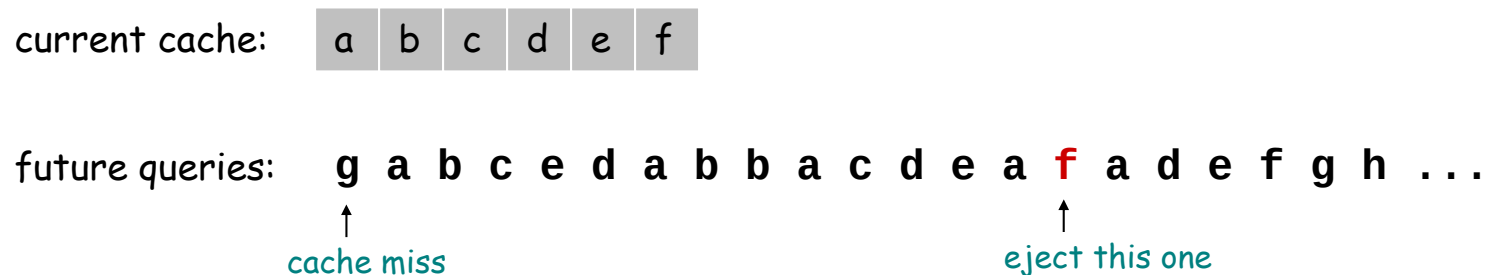
Ex: $k = 2$, initial cache = ab ,
requests: a, b, c, b, c, a, a, b .

Optimal eviction schedule: 2 cache misses.

a	a	b
b	a	b
c	c	b
b	c	b
c	c	b
a	a	b
a	a	b
b	a	b
requests	cache	

Optimal Offline Caching: Farthest-In-Future

Farthest-in-future. Evict item in the cache that is not requested until farthest in the future.



Theorem. [Bellady, 1960s] FF is optimal eviction schedule.

Pf. Algorithm and theorem are intuitive; proof is subtle.

Reduced Eviction Schedules

Def. A **reduced** schedule is a schedule that inserts an item into the cache only in a step in which that item is requested.

Intuition. Can transform an unreduced schedule into a reduced one with no more cache misses.

a	a	b	c
a	a	x	c
c	a	d	c
d	a	d	b
a	a	c	b
b	a	x	b
c	a	c	b
a	a	b	c
a	a	b	c

an unreduced schedule

a	a	b	c
a	a	b	c
c	a	b	c
d	a	d	c
a	a	d	c
b	a	d	b
c	a	c	b
a	a	c	b
a	a	c	b

a reduced schedule

Multiple Optimal Schedules

a	a	b	c
b	a	b	c
c	a	b	c
d	a	d	c
a	a	d	c
d	a	d	c
e	a	d	e
a	a	d	e
d	a	d	e
b	a	d	b
c	a	d	c

Another optimal schedule
(2+2 cache misses)

a	a	b	c
b	a	b	c
c	a	b	c
d	a	b	d
a	a	b	d
d	a	b	d
e	a	e	d
a	a	e	d
d	a	e	d
b	a	b	d
c	a	c	d

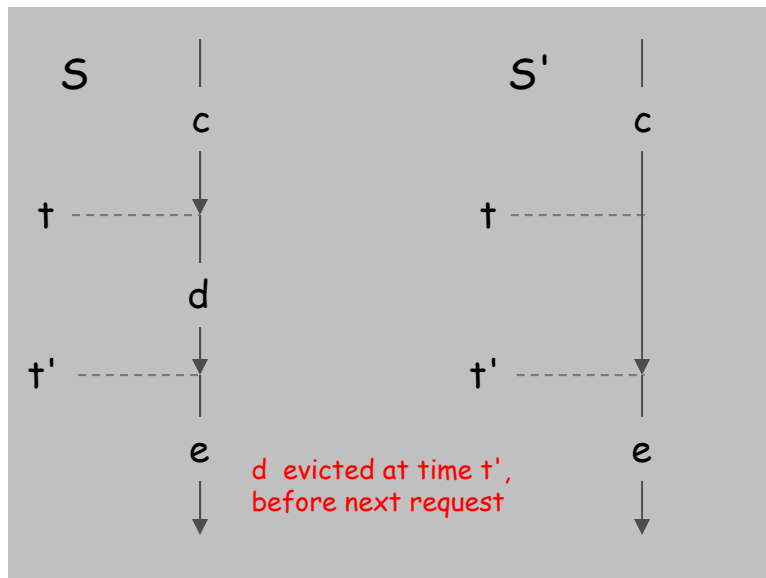
Belady Schedule
(Farthest in Future)

Reduced Eviction Schedules

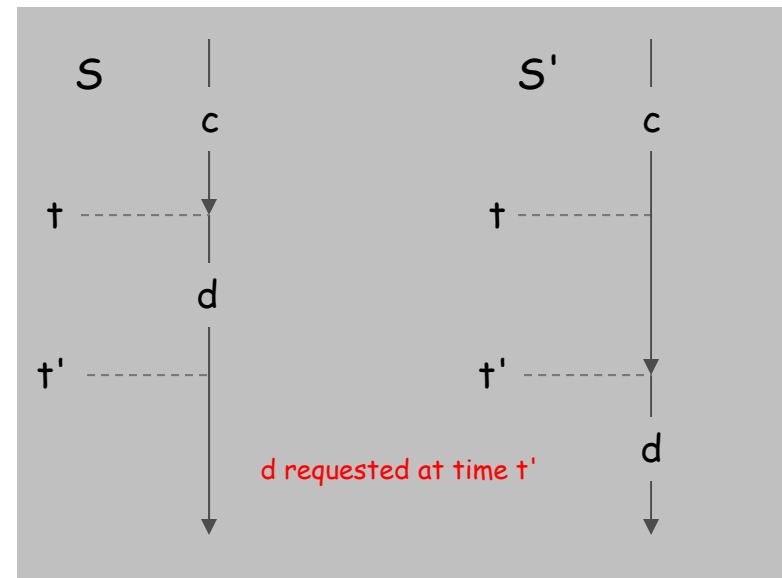
Claim. Given any unreduced schedule S , it can be transformed into a reduced schedule S' with no more cache misses.

Pf. (by induction on number of unreduced items) doesn't enter cache at requested time

- Suppose S brings d into the cache at time t , without a request.
- Let c be the item S evicts when it brings d into the cache.
- **Case 1:** d evicted at time t' , before next request for d .
- **Case 2:** d requested at time t' before d is evicted.



Case 1



Case 2

Farthest-In-Future: Analysis

Theorem. FF is an optimal eviction algorithm.

Pf. (by induction on number of requests j)

Invariant: There exists an optimal reduced schedule S that makes the same eviction schedule as S_{FF} through the first j requests.

Let S be reduced schedule that satisfies invariant through j requests.

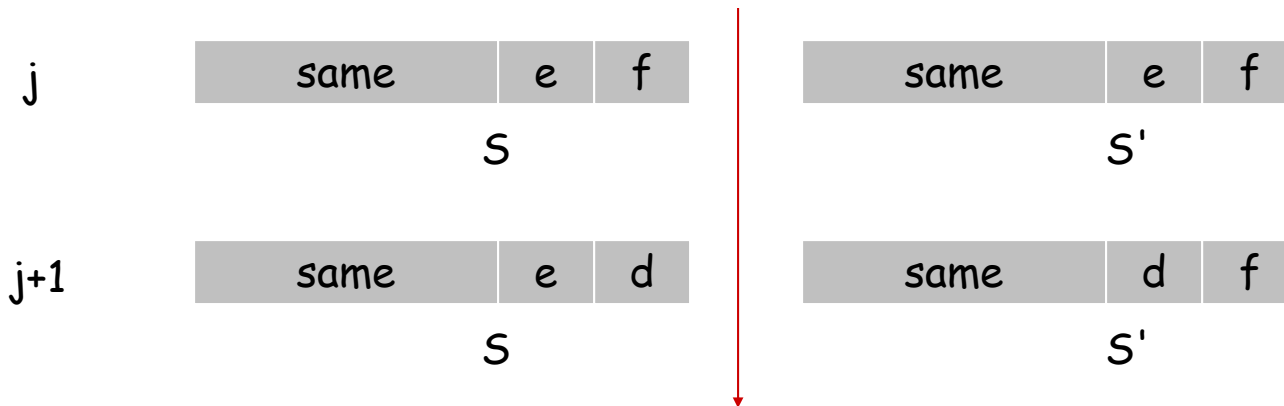
We produce S' that satisfies invariant after $j+1$ requests.

- Consider $(j+1)^{st}$ request $d = d_{j+1}$.
- Since S and S_{FF} have agreed up until now, they have the same cache contents before request $j+1$.
- **Case 1:** (d is already in the cache). $S' = S$ satisfies **invariant**.
- **Case 2:** (d is not in the cache, and S and S_{FF} evict the same element).
 $S' = S$ satisfies **invariant**.

Farthest-In-Future: Analysis

Pf. (continued)

- **Case 3:** (d is not in the cache; S_{FF} evicts e ; S evicts $f \neq e$).
 - begin construction of S' from S by evicting e instead of f .

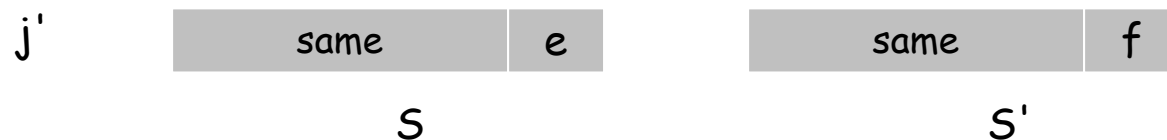


- now S' agrees with S_{FF} on first $j+1$ requests; **we show that having element f in cache is no worse than having element e .**

Farthest-In-Future: Analysis

Let j' be the **first** time after $j+1$ that S and S' take different actions, and let g be item requested at time j' .

↑
must involve e or f (or both)



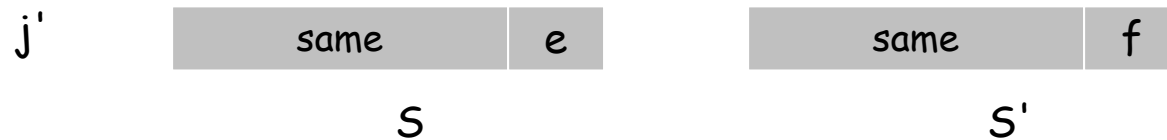
- **Case 3a:** $g = e$. Can't happen with Farthest-In-Future since there must be a request for f before e . (See S_{FF} definition)
- **Case 3b:** $g = f$. Element f can't be in cache of S , so let e' be the element that S evicts (to get room for f).
 - if $e' = e$, S' accesses f from cache; now S and S' have same cache. (S does worse now.)
 - if $e' \neq e$, S' evicts e' and brings e into the cache; now S and S' have the same cache. (Both S and S' have e and f now.)

Note: S' is no longer reduced, but can be transformed into a reduced schedule that agrees with S_{FF} through step $j+1$

Farthest-In-Future: Analysis

Let j' be the **first** time after $j+1$ that S and S' take a different action, and let g be item requested at time j' .

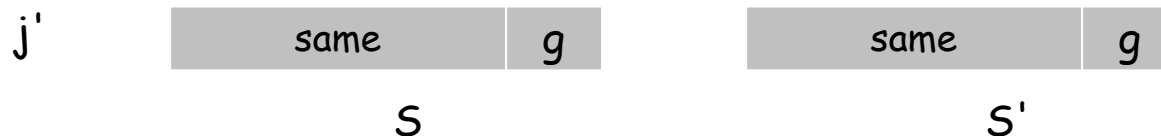
↑
must involve e or f (or both)



otherwise S' would take the same action

- **Case 3c:** $g \neq e, f$. S must evict e .

Make S' evict f ; now S and S' have the same cache. ▪



Caching Perspective

Online vs. offline algorithms.

- **Offline:** full sequence of requests is known a priori.
- **Online (reality):** requests are not known in advance.
- Caching is among the most fundamental online problems in CS.

FIFO. Evict page brought in least recently.

LIFO. Evict page brought in most recently.

LRU. Evict page whose most recent access was earliest.

↑
FF with direction of time reversed!

Theorem. FF is optimal offline eviction algorithm.

- Provides basis for understanding and analyzing online algorithms.
- LRU is k -competitive. [Section 13.8]
- LIFO is arbitrarily bad.

Belady's Optimal Algorithm

Let page reference stream, $R = 2031203120316457$

Frame	2	0	3	1	2	0	3	1	2	0	3	1	6	4	5	7
0	<u>2</u>	2	2	2	2	2	2	2	2	<u>0</u>	0	0	0	<u>4</u>	4	4
1		<u>0</u>	0	0	0	0	<u>3</u>	3	3	3	3	3	<u>6</u>	6	6	<u>7</u>
2			<u>3</u>	<u>1</u>	1	1	1	1	1	1	1	1	1	1	<u>5</u>	5

10 page faults

- Perfect knowledge of $R \Rightarrow$ perfect performance
- Impossible to implement

Belady's Anomaly

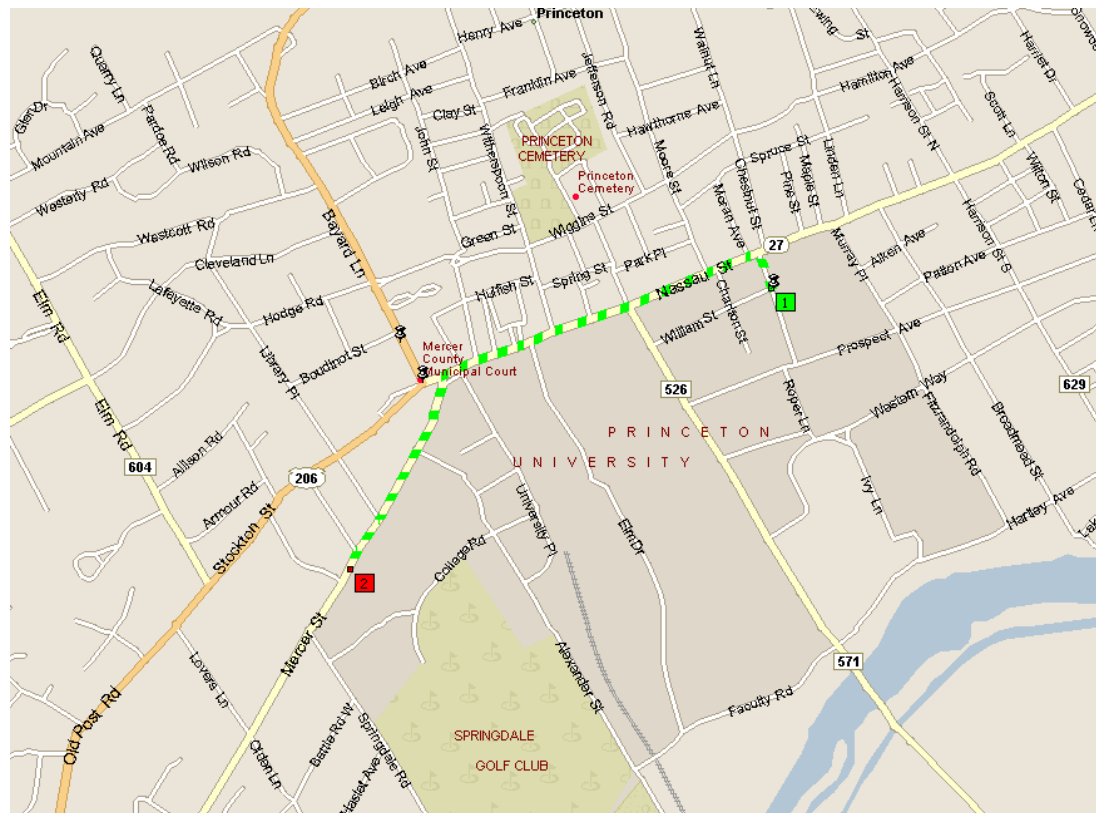
Let page reference stream, $R = 012301401234$

Frame	0	1	2	3	0	1	4	0	1	2	3	4
0	<u>0</u>	0	0	<u>3</u>	3	3	<u>4</u>	4	4	4	4	4
1		<u>1</u>	1	1	<u>0</u>	0	0	0	0	<u>2</u>	2	2
2			<u>2</u>	2	2	<u>1</u>	1	1	1	1	<u>3</u>	3

Frame	0	1	2	3	0	1	4	0	1	2	3	4
0	<u>0</u>	0	0	0	0	0	<u>4</u>	4	4	4	<u>3</u>	3
1		<u>1</u>	1	1	1	1	1	<u>0</u>	0	0	0	<u>4</u>
2			<u>2</u>	2	2	2	2	2	<u>1</u>	1	1	1
3				<u>3</u>	3	3	3	3	3	<u>2</u>	2	2

- FIFO with $m = 3$ has 9 faults
- FIFO with $m = 4$ has 10 faults

4.4 Shortest Paths in a Graph



shortest path from Princeton CS department to Einstein's house

Car navigation



Shortest path applications

- PERT/CPM.
- Map routing.
- Seam carving.
- Robot navigation.
- Texture mapping.
- Typesetting in LaTeX.
- Urban traffic planning.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Optimal truck routing through given traffic congestion pattern.

Reference: Network Flows: Theory, Algorithms, and Applications, R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Prentice Hall, 1993.

Shortest Path Problem

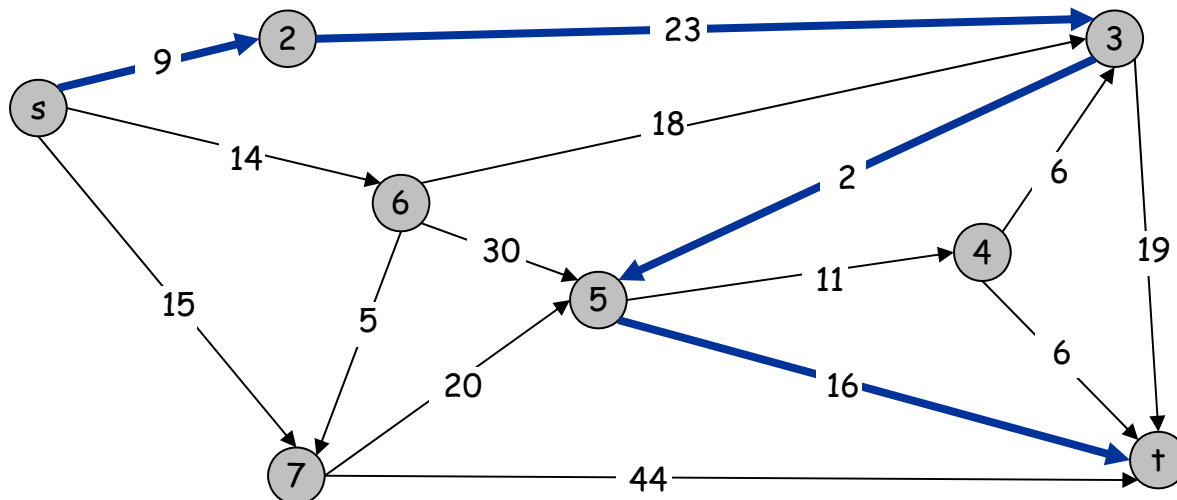
Shortest path network.

- 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 .



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

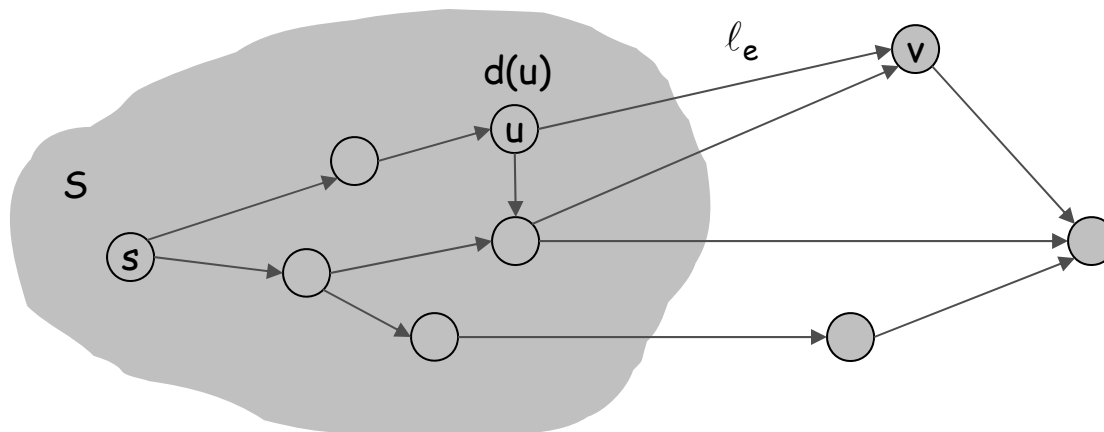
Dijkstra's algorithm.

- 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 (i.e., $v \notin S$) that minimizes

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

add v to S , and set $d(v) = \pi(v)$.

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



Dijkstra's Algorithm

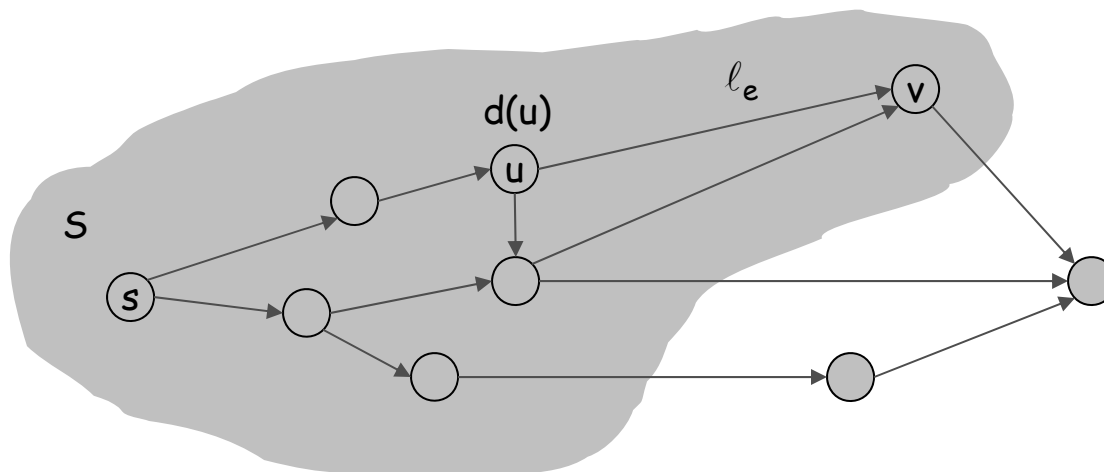
Dijkstra's algorithm.

- 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 $d(v) = \pi(v)$.

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



Dijkstra's Algorithm: Proof of Correctness

Invariant. For each node $u \in S$, $d(u)$ is the length of the shortest s - u path.

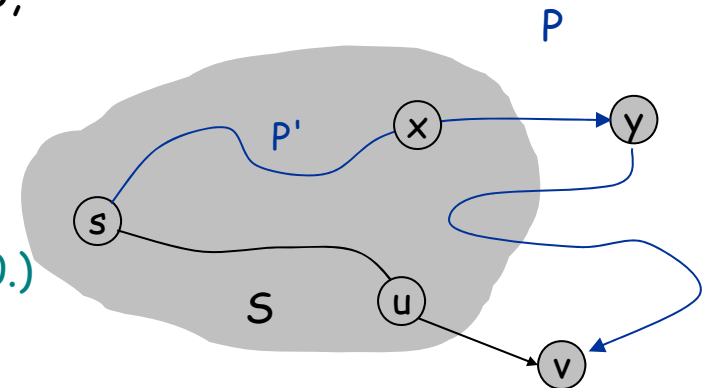
Pf. (by induction on $|S|$)

Base case: $|S| = 1$ is trivial.

Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be the next node added to S , and let u - v be the chosen edge.
- The shortest s - u path plus (u, v) is an s - v path of length $\pi(v)$.
- Consider any s - v path P . **Show that it is no shorter than $\pi(v)$.**
- Let x - y be the first edge in P that leaves S , and let P' be the subpath to x .
- P is already too long as soon as it leaves S and path y - v has positive weight.**

(Boundary Case: P has same length with $\text{len}(y-v) = 0$.)



$$\ell(P) \geq \ell(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq \pi(y) \geq \pi(v)$$

↑
nonnegative
weights (lengths)

↑
inductive
hypothesis

↑
defn of $\pi(y)$

↑
Dijkstra's algorithm
chose v instead of y

Dijkstra's algorithm: efficient implementation

Critical optimization 1. For each unexplored node v , explicitly maintain $\pi(v)$ instead of computing directly from formula:



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

- For each $v \notin S$, $\pi(v)$ can only decrease (because S only increases).
- More specifically, suppose u is added to S and there is an edge (u, v) leaving u . Then, it suffices to update:

$$\pi(v) = \min \{ \pi(v), d(u) + \ell(u, v) \}$$

Critical optimization 2. Use a **priority queue** to choose the unexplored node that minimizes $\pi(v)$.

Dijkstra's algorithm: efficient implementation

Implementation.

- Algorithm stores $d(v)$ for each explored node v .
- Priority queue stores $\pi(v)$ for each unexplored node v .
- Recall: $d(u) = \pi(u)$ when u is deleted from priority queue.

DIJKSTRA(V, E, s)

Create an empty priority queue.

FOR EACH $v \neq s$: $d(v) \leftarrow \infty$; $d(s) \leftarrow 0$.

FOR EACH $v \in V$: *insert* v with key $d(v)$ into priority queue.

WHILE (the priority queue *is not empty*)

$u \leftarrow$ *delete-min* from priority queue.

 FOR EACH edge $(u, v) \in E$ leaving u :

 IF $d(v) > d(u) + \ell(u, v)$

decrease-key of v to $d(u) + \ell(u, v)$ in priority queue.

$d(v) \leftarrow d(u) + \ell(u, v)$.

Dijkstra's algorithm: which priority queue?

Performance. Depends on PQ: n insert, n delete-min, m decrease-key.

- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci/Brodal best in theory, but not worth implementing.

PQ implementation	insert	delete-min	decrease-key	total
unordered array	$O(1)$	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m \log n)$
d-way heap (Johnson 1975)	$O(d \log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	$O(m \log_{m/n} n)$
Fibonacci heap (Fredman-Tarjan 1984)	$O(1)$	$O(\log n)^\dagger$	$O(1)^\dagger$	$O(m + n \log n)$
Brodal queue (Brodal 1996)	$O(1)$	$O(\log n)$	$O(1)$	$O(m + n \log n)$

† amortized

Review of Dijkstra (Non-negative Edge Weights)

Problem: Given a directed graph $G = (V, E)$ with edge-weight function $w : E \rightarrow \mathbb{R}^+$, and a node s , find the shortest-path weight $\delta(s, v)$ (and a corresponding shortest path) from s to each v in V .

Greedy iterative approach

1. maintain a set S of vertices whose shortest-path distances from s are known.
2. at each step add to S the vertex $v \in V - S$ whose distance estimate from s is minimal.
3. update distance estimates of vertices adjacent to v .

Dijkstra's algorithm

$d[s] \leftarrow 0$

for each $v \in V - \{s\}$

do $d[v] \leftarrow \infty$

$S \leftarrow \emptyset$

$Q \leftarrow V$

while $Q \neq \emptyset$

(Q min-priority queue for $V - S$)

do $u \leftarrow \text{DELETE-MIN}(Q)$

$S \leftarrow S \cup \{u\}$

for each $v \in \text{Adj}[u]$

do if $d[v] > d[u] + w(u, v)$

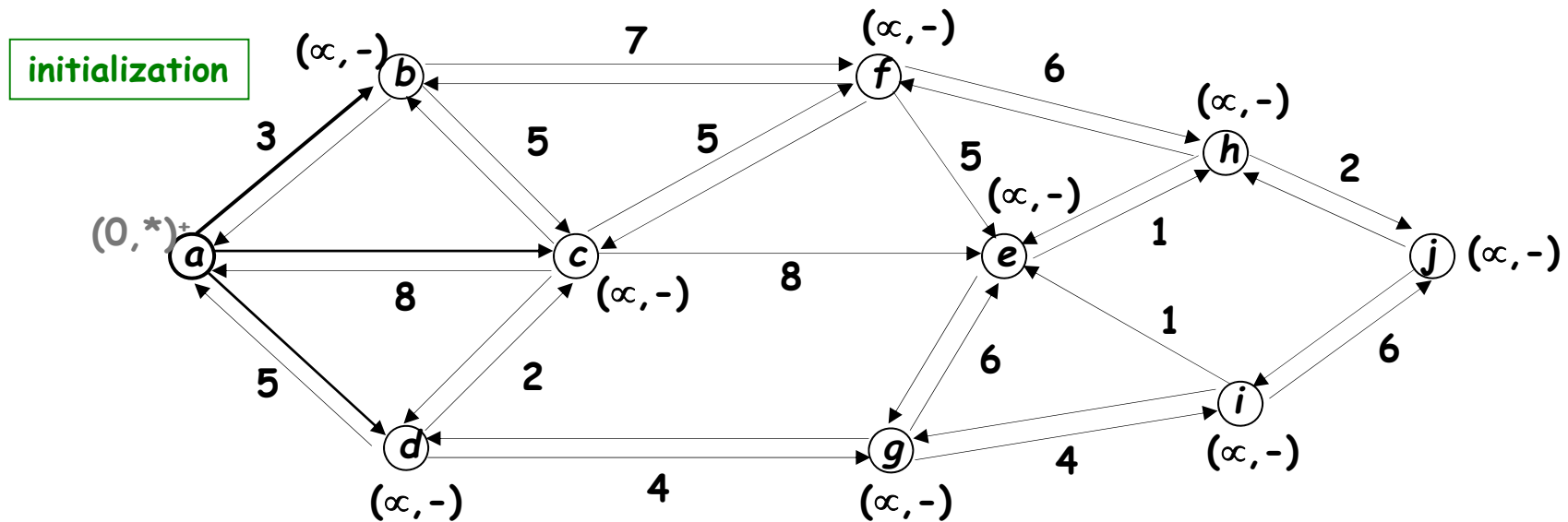
then $d[v] \leftarrow d[u] + w(u, v)$

(Implicit **DECREASE-KEY**)

initialization

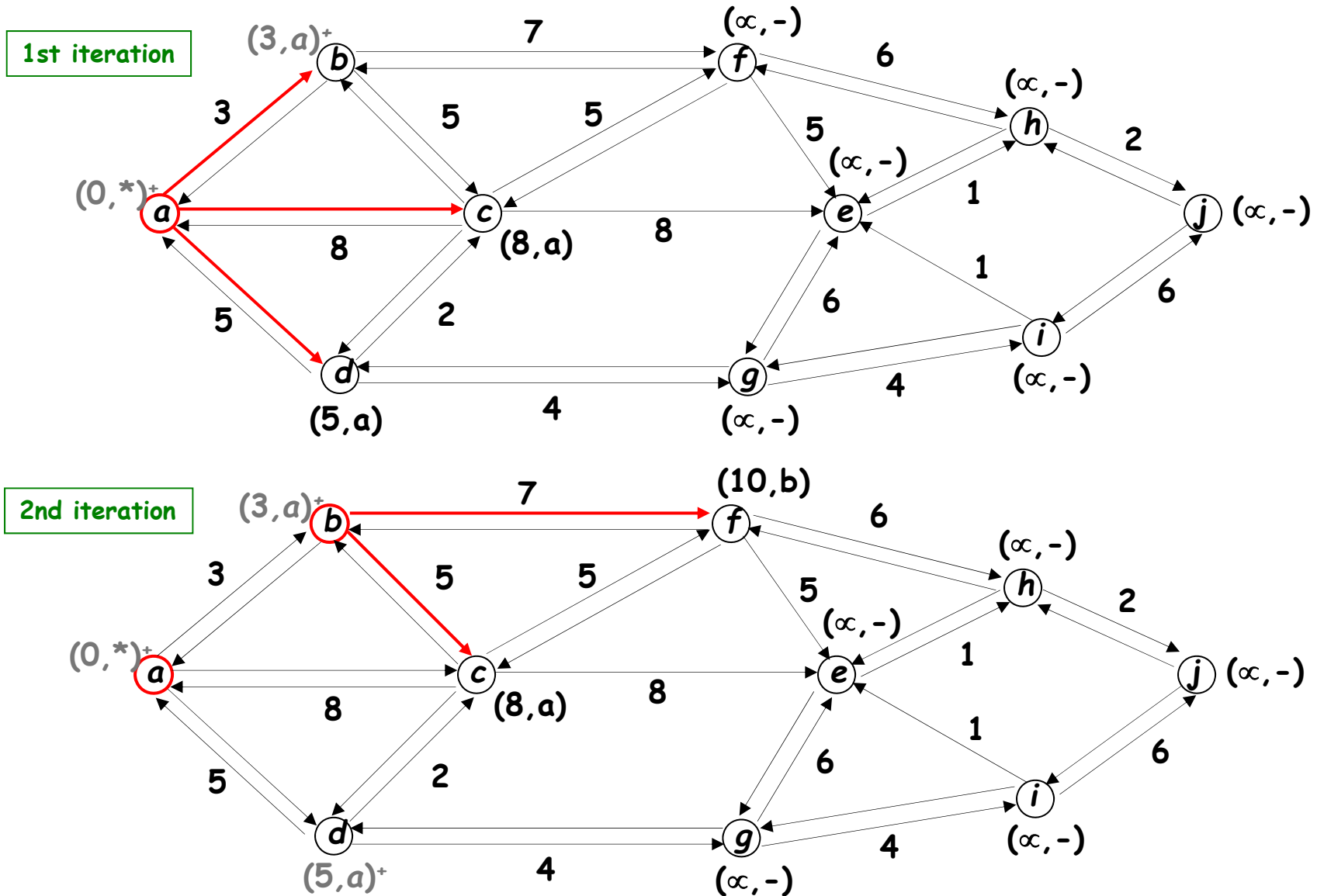
relaxation steps

Dijkstra: Example (Min-cost, Min-Path)



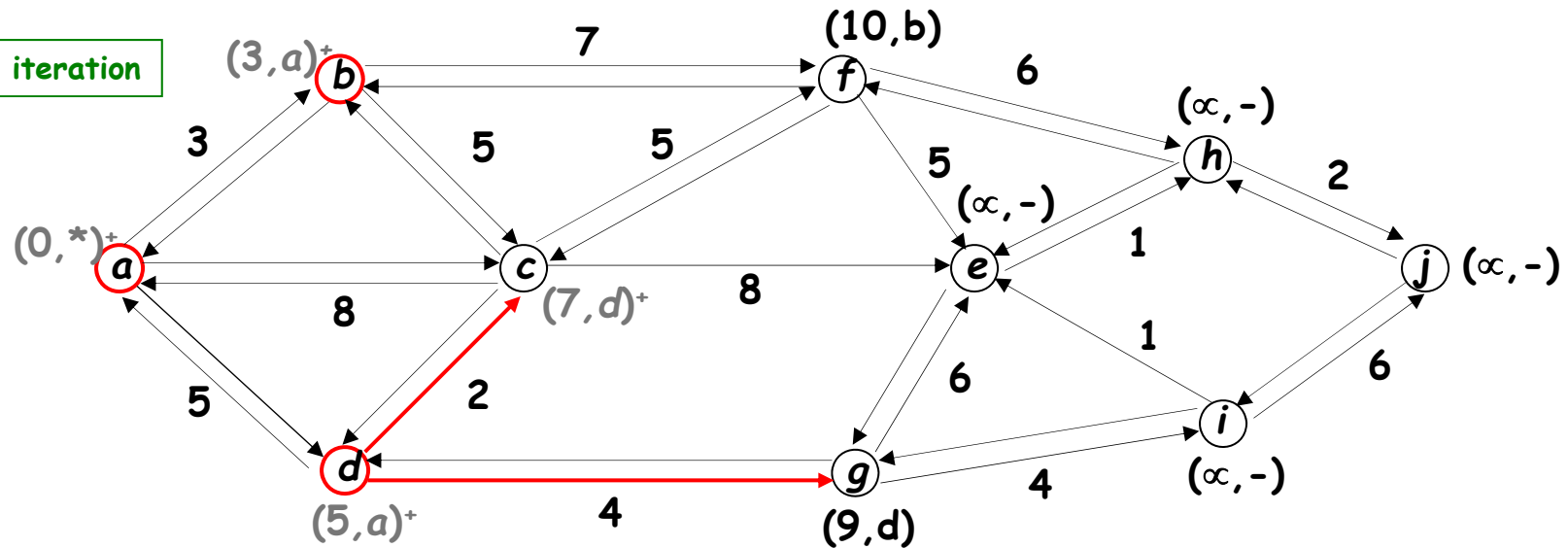
$Q = V, a = \text{DELETE-MIN}(Q)$

Dijkstra: Example

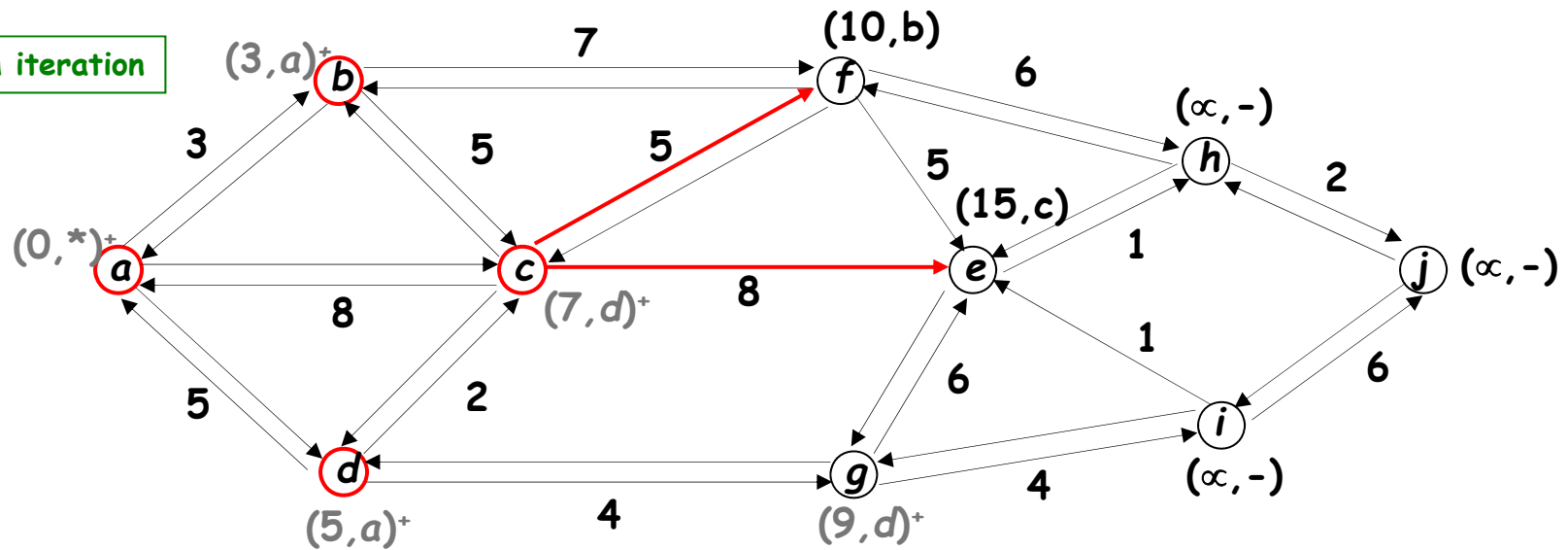


Dijkstra: Example

3rd iteration

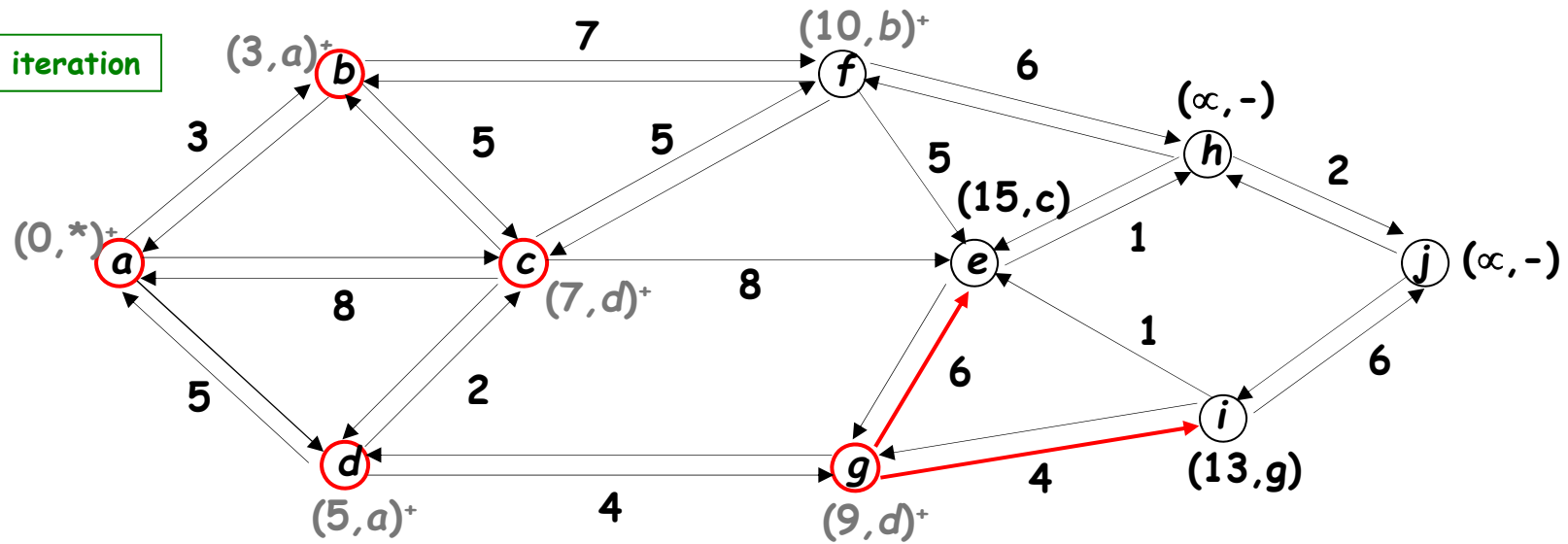


4th iteration

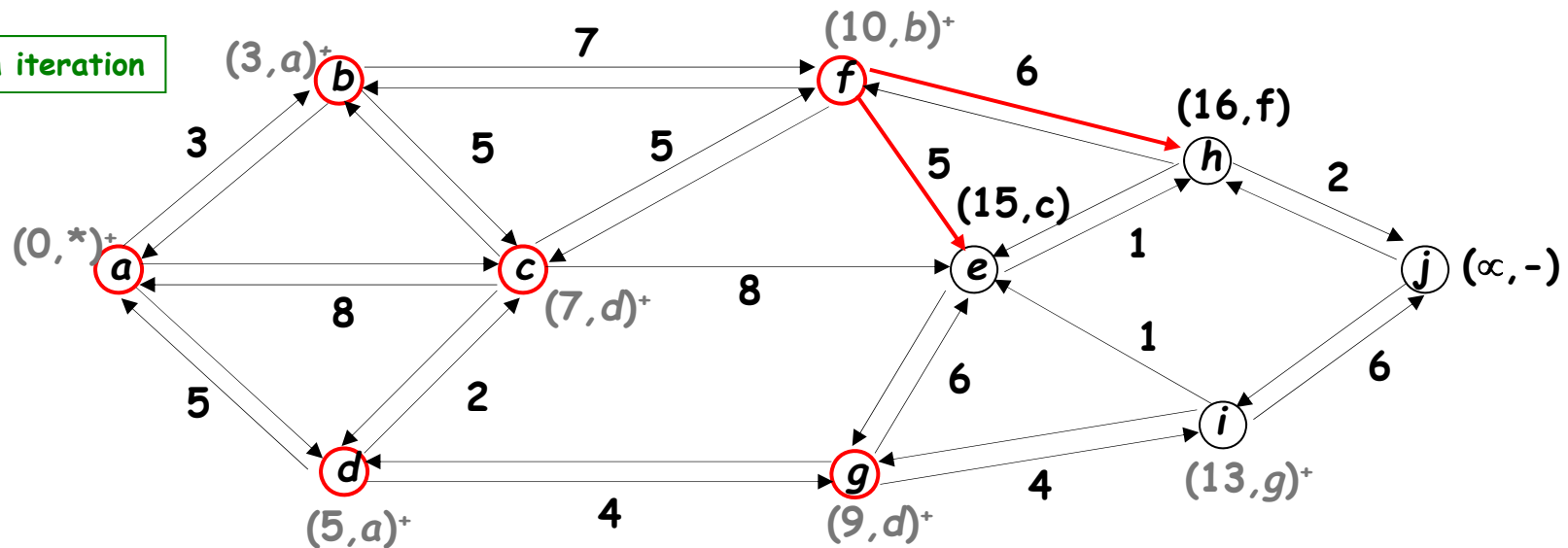


Dijkstra: Example

5th iteration

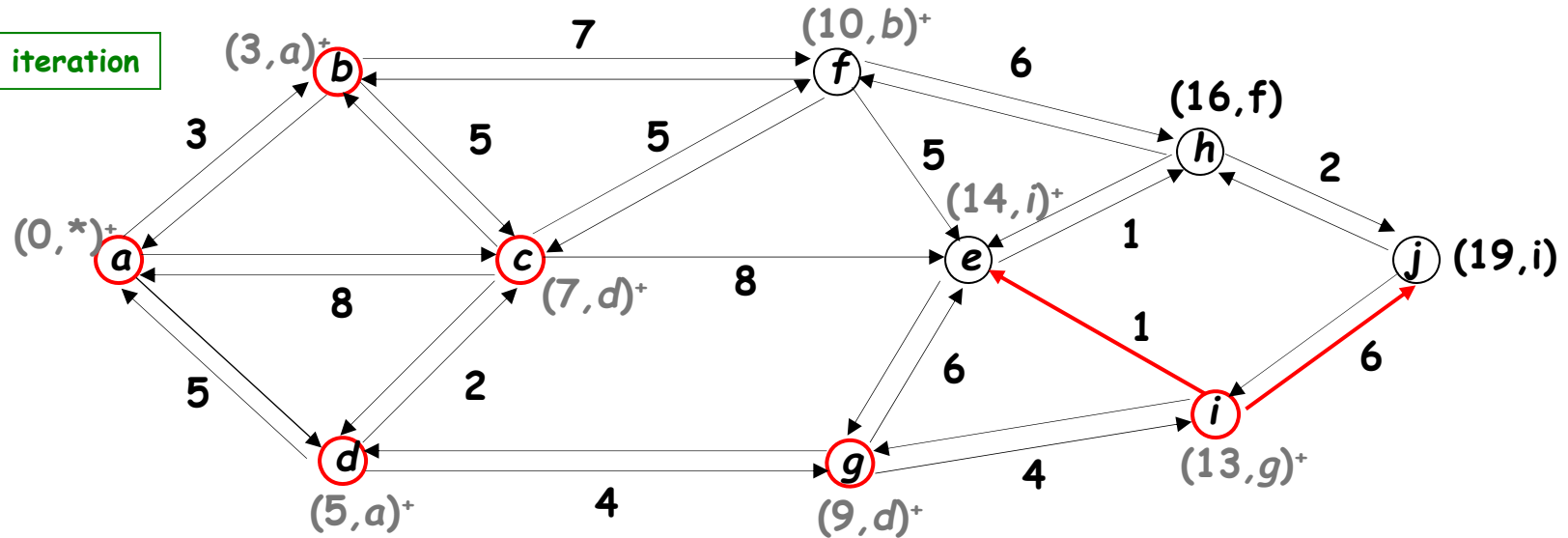


6th iteration

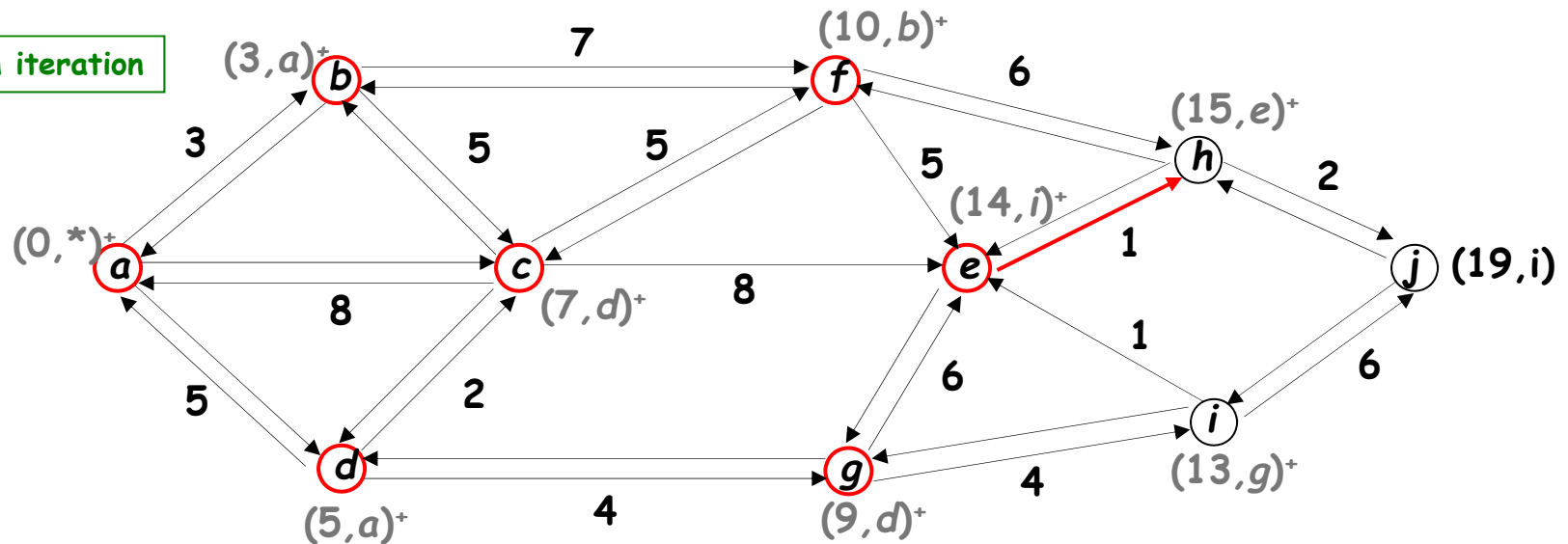


Dijkstra: Example

7th iteration

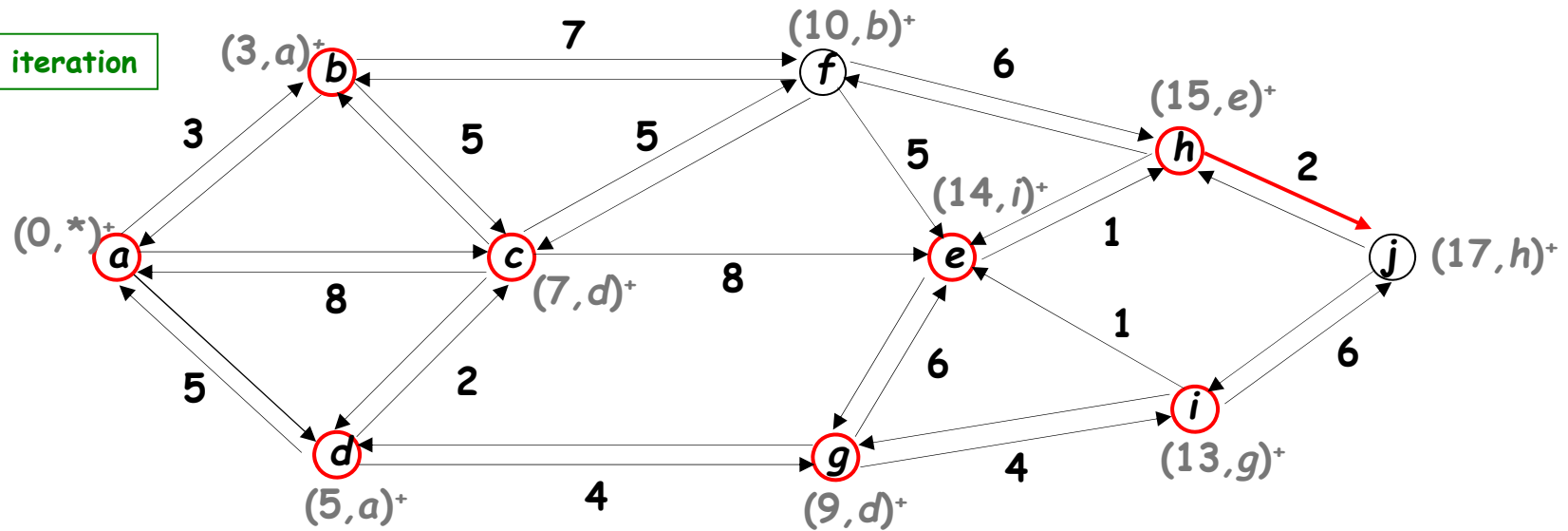


8th iteration

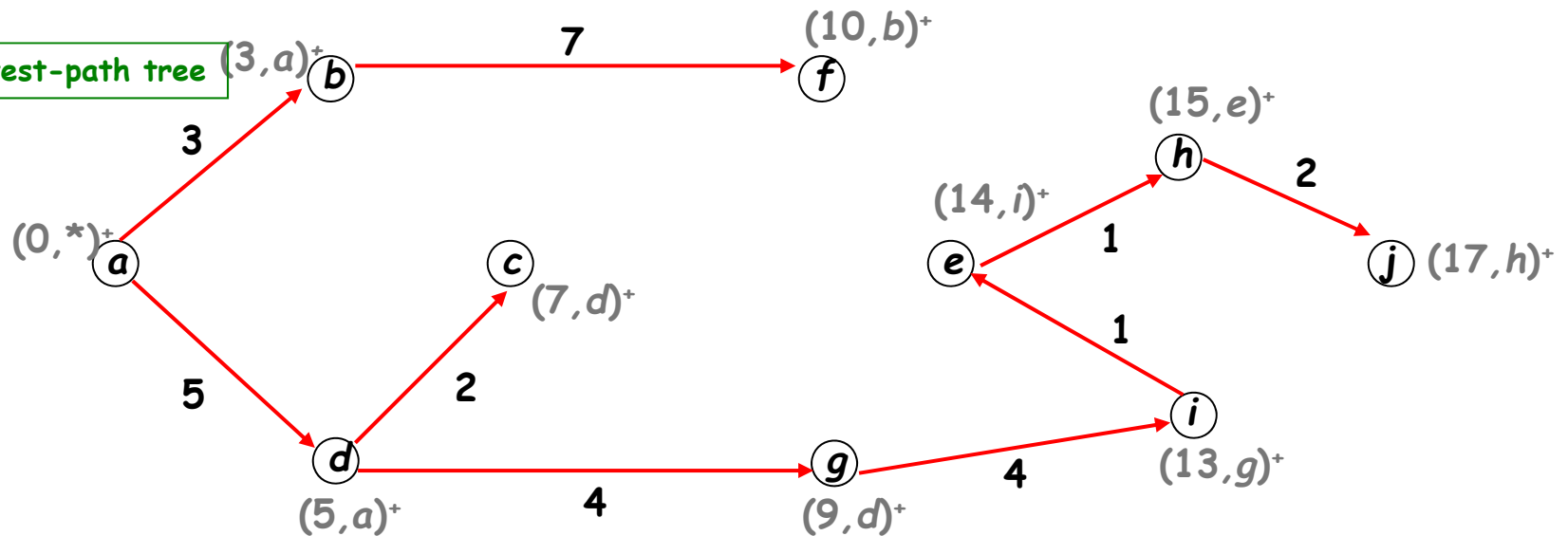


Dijkstra: Example

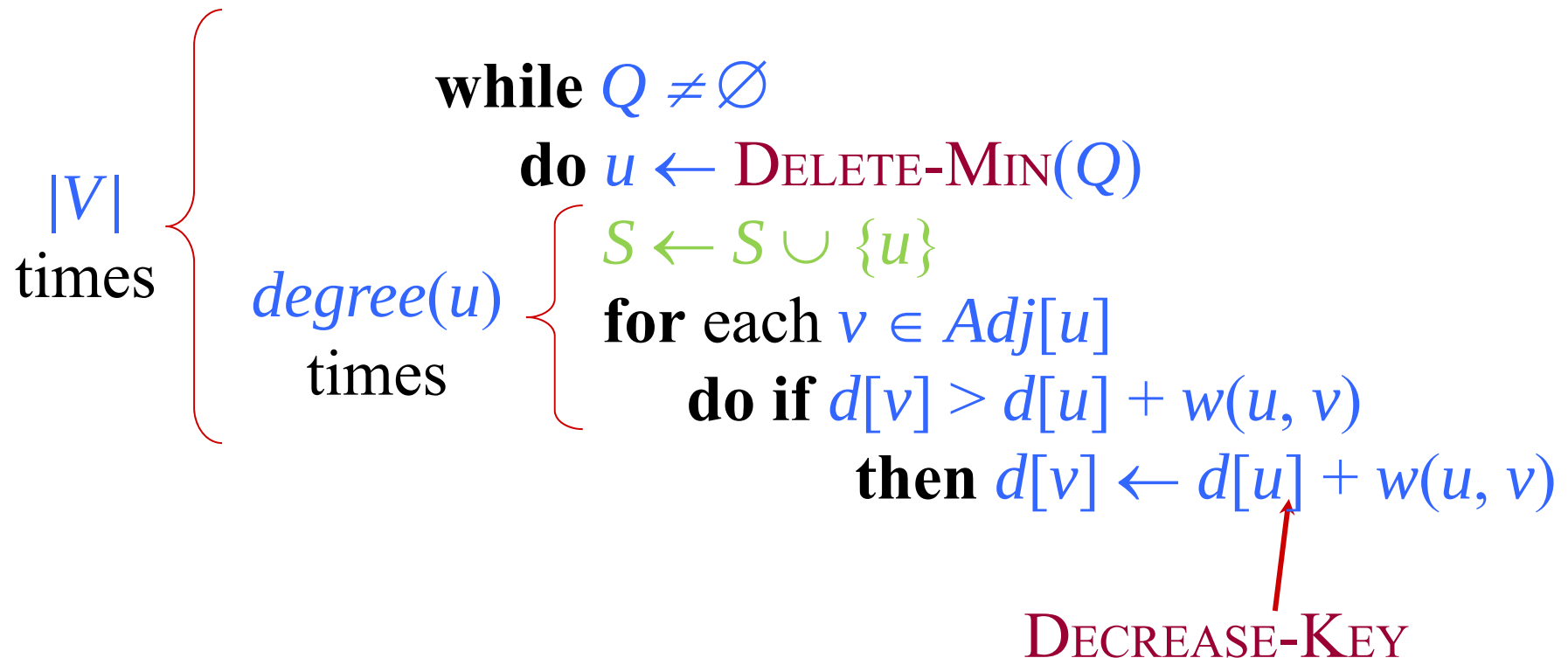
9th iteration



Shortest-path tree



Complexity Analysis of Dijkstra



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

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

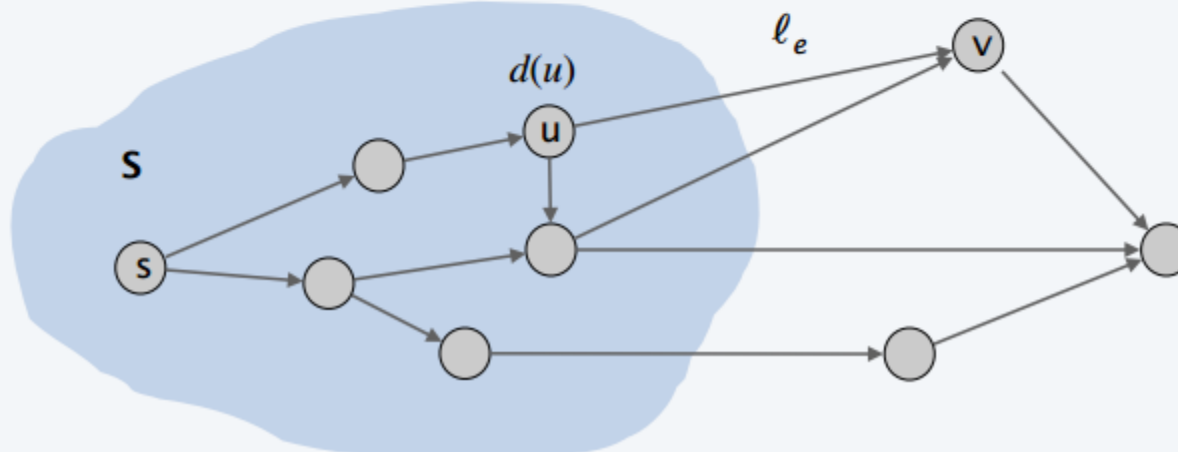
Q	$T_{\text{DELETE-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\lg n)$	$O(\lg n)$	$O(m \lg n)$
Fibonacci heap	$O(\lg n)$ amortized	$O(1)$ amortized	$O(m + n \lg n)$ worst case

Extensions of Dijkstra's algorithm

Dijkstra's algorithm and proof extend to several related problems:

- Shortest paths in undirected graphs: $d(v) \leq d(u) + \ell(u, v)$.
- Maximum capacity paths: $d(v) \geq \min \{ \pi(u), c(u, v) \}$.
- Maximum reliability paths: $d(v) \geq d(u) \times \gamma(u, v)$.
- ...

Key algebraic structure. Closed semiring (tropical, bottleneck, Viterbi).

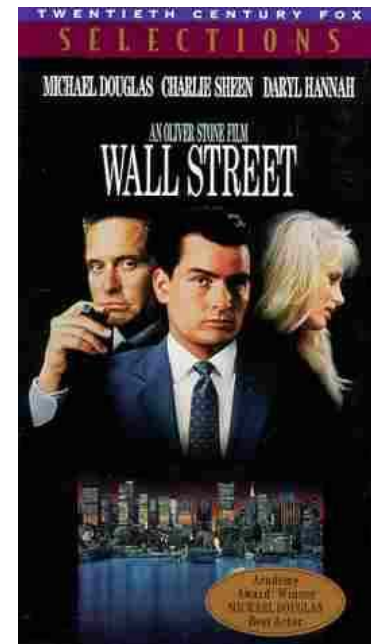


Extra Slides

Coin Changing

Greed is good. Greed is right. Greed works. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit.

- *Gordon Gecko (Michael Douglas)*



Coin Changing

Goal. Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins.

Ex: 34¢.



Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Ex: \$2.89.



Coin-Changing: Greedy Algorithm

Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Sort coins denominations by value: $c_1 < c_2 < \dots < c_n$.

coins selected
↓
 $S \leftarrow \phi$
while ($x \neq 0$) {
 let k be largest integer such that $c_k \leq x$
 if ($k = 0$)
 return "no solution found"
 $x \leftarrow x - c_k$
 $S \leftarrow S \cup \{k\}$
}
return S

Q. Is cashier's algorithm optimal?

Coin-Changing: Analysis of Greedy Algorithm

Theorem. Greedy is optimal for U.S. coinage: 1, 5, 10, 25, 100.

Pf. (by induction on x)

- Consider optimal way to change $c_k \leq x < c_{k+1}$: greedy takes coin k .
- We claim that any optimal solution must also take coin k .
 - if not, it needs enough coins of type $c_1, \dots, c_{k-1}$ to add up to x
 - table below indicates no optimal solution can do this
- Problem reduces to coin-changing $x - c_k$ cents, which, by induction, is optimally solved by greedy algorithm. ▪

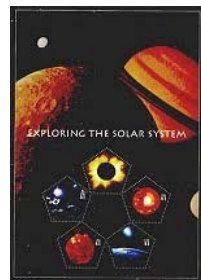
k	c_k	All optimal solutions must satisfy	Max value of coins 1, 2, ..., $k-1$ in any OPT
1	1	$P \leq 4$	-
2	5	$N \leq 1$	4
3	10	$N + D \leq 2$	$4 + 5 = 9$
4	25	$Q \leq 3$	$20 + 4 = 24$
5	100	no limit	$75 + 24 = 99$

Coin-Changing: Analysis of Greedy Algorithm

Observation. Greedy algorithm is sub-optimal for US postal denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

Counterexample. 140¢.

- Greedy: 100, 34, 1, 1, 1, 1, 1, 1.
- Optimal: 70, 70.



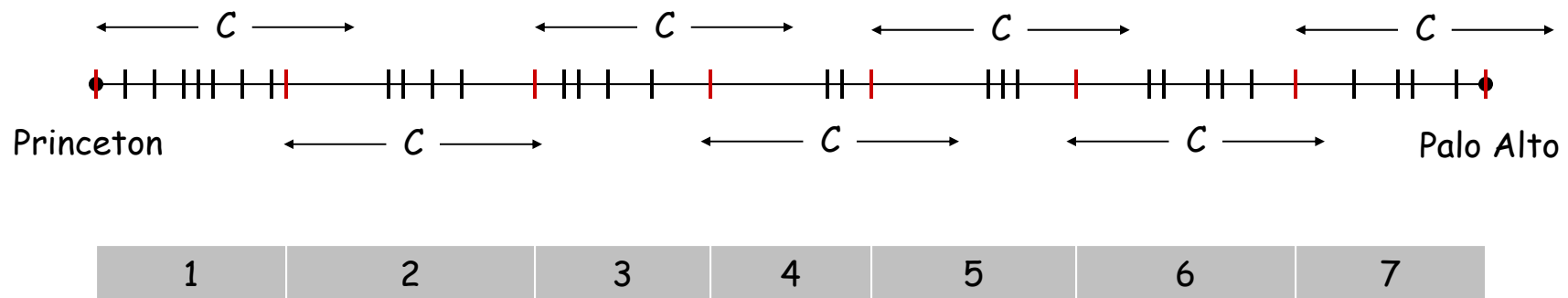
Selecting Breakpoints

Selecting Breakpoints

Selecting breakpoints.

- Road trip from Princeton to Palo Alto along fixed route.
- Refueling stations at certain points along the way.
- Fuel capacity = C .
- **Goal:** makes as few refueling stops as possible.

Greedy algorithm. Go as far as you can before refueling.



Selecting Breakpoints: Greedy Algorithm

Truck driver's algorithm.

Sort breakpoints so that: $0 = b_0 < b_1 < b_2 < \dots < b_n = L$

$S \leftarrow \{0\}$ $\leftarrow$ breakpoints selected

$x \leftarrow 0$ $\leftarrow$ current location

```
while (x  $\neq$   $b_n$ )  
    let p be largest integer such that  $b_p \leq x + C$   
    if ( $b_p = x$ )  
        return "no solution"  
    x  $\leftarrow$   $b_p$   
    S  $\leftarrow$  S  $\cup$  {p}  
return S
```

Implementation. $O(n \log n)$

- Use **binary search** to select each breakpoint p.

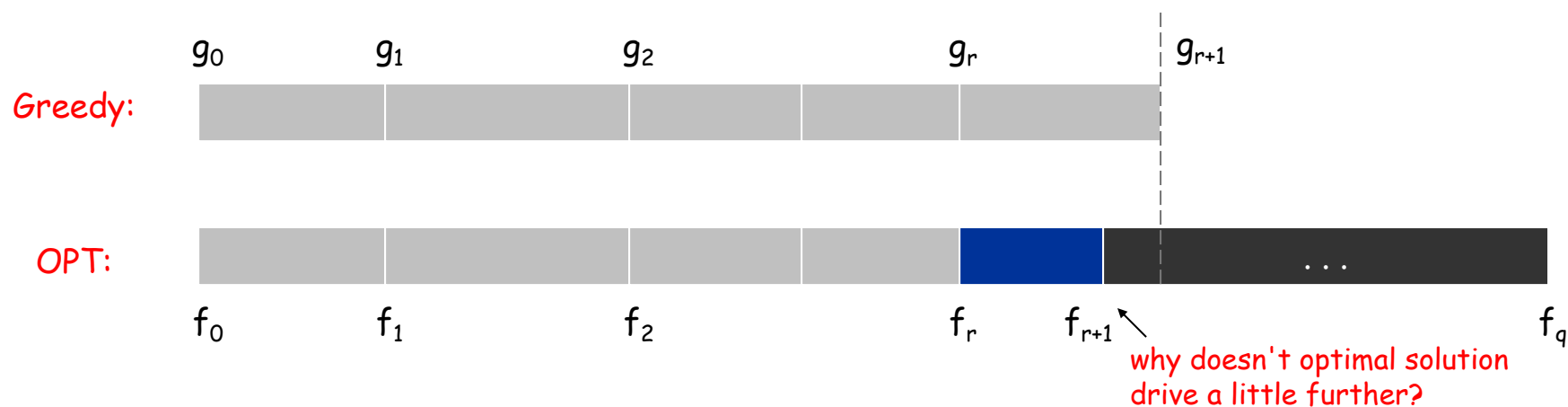
BUG:
Use
REPEAT-UNTIL
loop

Selecting Breakpoints: Correctness

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy algorithm is not optimal.
- Let $0 = g_0 < g_1 < \dots < g_p = L$ denote set of breakpoints chosen by it.
- Let $0 = f_0 < f_1 < \dots < f_q = L$ denote set of breakpoints in an optimal solution with $f_0 = g_0, f_1 = g_1, \dots, f_r = g_r$ for largest possible value of r .
- Note: $g_{r+1} > f_{r+1}$ by greedy choice of algorithm.

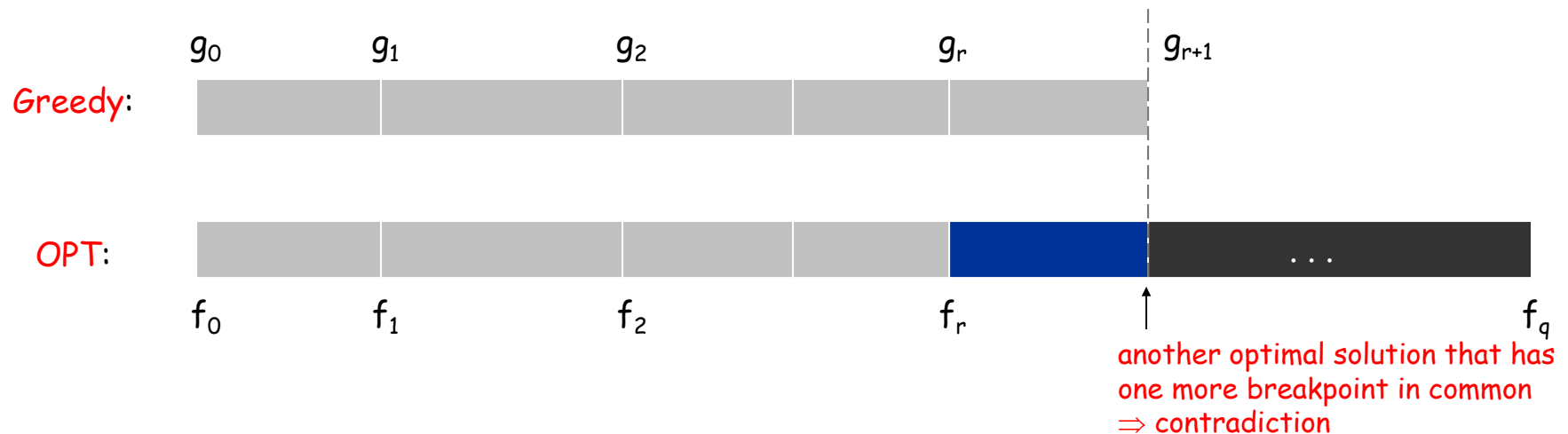


Selecting Breakpoints: Correctness

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy algorithm is not optimal.
- Let $0 = g_0 < g_1 < \dots < g_p = L$ denote set of breakpoints chosen by it.
- Let $0 = f_0 < f_1 < \dots < f_q = L$ denote set of breakpoints in an optimal solution with $f_0 = g_0, f_1 = g_1, \dots, f_r = g_r$ for largest possible value of r .
- Note: $g_{r+1} > f_{r+1}$ by greedy choice of algorithm.



Edsger W. Dijkstra

The question of whether computers can think is like the question of whether submarines can swim.

Do only what only you can do.

In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.

The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.

APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.

