10 – Greedy Algorithms

Greedy algorithms

1. Introduction
2. Basic examples
   - The coin-changing problem
   - The Traveling Salesman Problem
3. Scheduling problems
   - Interval scheduling
   - Scheduling to minimize lateness
4. Discussion: Shortest paths and minimum spanning trees

Greedy algorithms for optimization problems

In each step make the choice that looks best at the moment!

Depending on the problem, the outcome can be:

1. The computed solution is always optimal.

2. The computed solution may not be optimal, but it never differs much from the optimum.

3. The computed solution can be arbitrarily bad.

Basic example: Coin-changing problem

Denominations of coins and banknotes (in €):

500, 200, 100, 50, 20, 10, 5, 2, 1

Observation

Any amount in € can be paid using coins and banknotes of these denominations.

Goal

Pay an amount *n* using the smallest number of coins and banknotes possible.

Coin-changing problem

Greedy algorithm

Repeatedly choose coin/banknote of the largest feasible denomination until the desired amount n is paid.

Example: $n = 487$

500   200   100   50   20   10   5   2   1

Coin-changing problem: formal description

Coin denominations of coins: $n_1, n_2, \ldots, n_k$

$n_1 > n_2 > \ldots > n_k$ and $n_k = 1$

Greedy algorithm

1. $w := n$;
2. **for** i=1 **to** k **do**
3. Pay $m_i := \lfloor w/n_i \rfloor$ coins of denomination n_i;
4. $w := w - m_i \cdot n_i$;
5. **endfor;**

Country 'Absurdia'

Three denominations:

$n_3 = 1$, $n_2 > 1$ arbitrary, $n_1 = 2n_2 + 1$

Example: 41, 20, 1

Amount to pay: $n = 3n_2$ (i.e. $n = 60$)

Optimal method of payment: $3 \times n_2$

Greedy method: $1 \times n_1 + (n_2\text{-}1) \times n_3$

The Traveling Salesman Problem (TSP)

Given: n cities, costs $c(i,j)$ to travel from city i to city j

Goal: Find a cheapest round-trip route that visits each city exactly once and then returns to the starting city.

Formally: Find a permutation p of $\{1, 2, ..., n\}$ such that

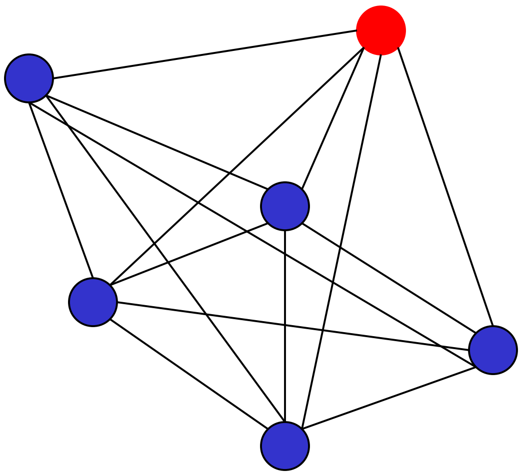
$$c(p(1),p(2)) + \ \ldots \ + c(p(n\text{-}1),p(n)) + c(p(n),p(1))$$

is minimzed.

The Traveling Salesman Problem (TSP)

A greedy algorithm for solving TSP

Starting from city 1, each time go to the nearest city not visited yet. Once all cities have been visited, return to the starting city 1.



The Traveling Salesman Problem (TSP)

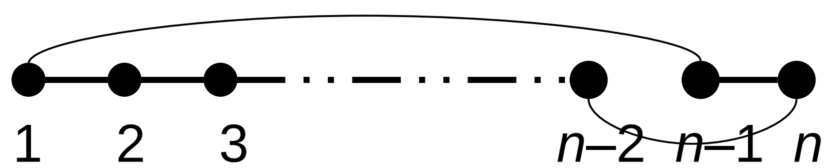
Example

$c(i,i+1) = 1$ for $i = 1, ..., n - 1$

$c(n,1) = M$ for some large number M

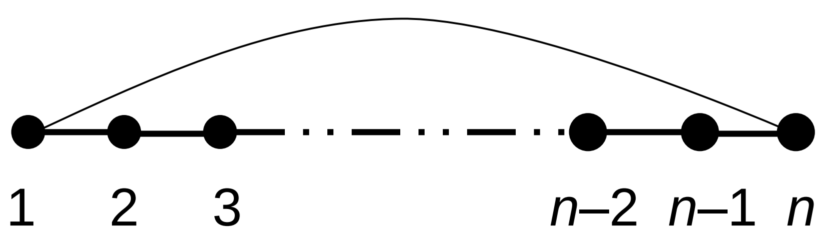
$c(i,j) = 2$ otherwise

Optimal tour:



Cost = $n+2$

Solution of the greedy algorithm:



Cost = $n-1+M$

Interval scheduling

Problem:

Set $S = \{1,\ldots,n\}$ of n requests for a resource, e.g. a lecture hall.

Request i: $[s(i), f(i))$ $s(i)$ = start time $f(i)$ = finish time

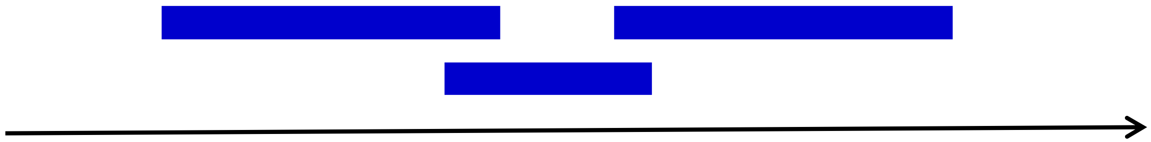
Subset of requests is compatible if no two of them overlap in time.

Goal: Select a maximum-size compatible subset of requests.

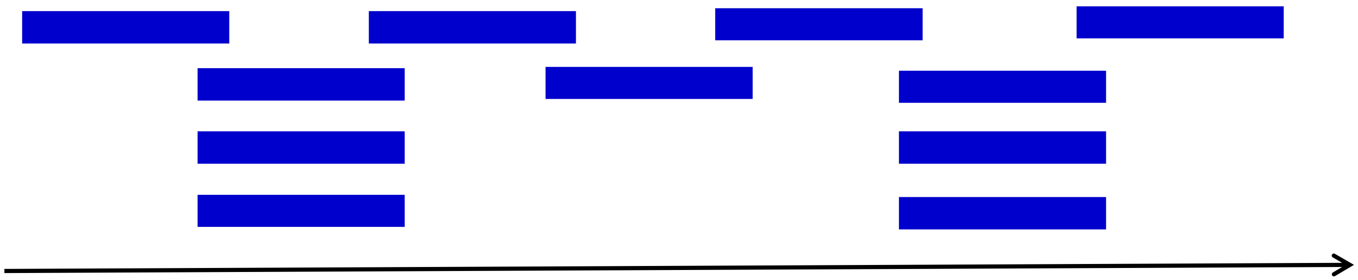
Greedy 1: Always select an available request that starts earliest, i.e. having minimal start time $s(i)$.

Interval scheduling

Greedy 2: Always select an available request that requires the shortest interval in time, i.e. for which *f*(*i*) - *s*(*i*) is as small as possible.



Greedy 3: Always select an available request that has the smallest number of non-compatible requests (interval with the fewest conflicts).



Greedy*

Always choose the request with the earliest finish time
that is compatible with all previously selected requests!

In particular, the request chosen first is the one with
the earliest finish time.

Theorem

Greedy* constructs an optimal solution.

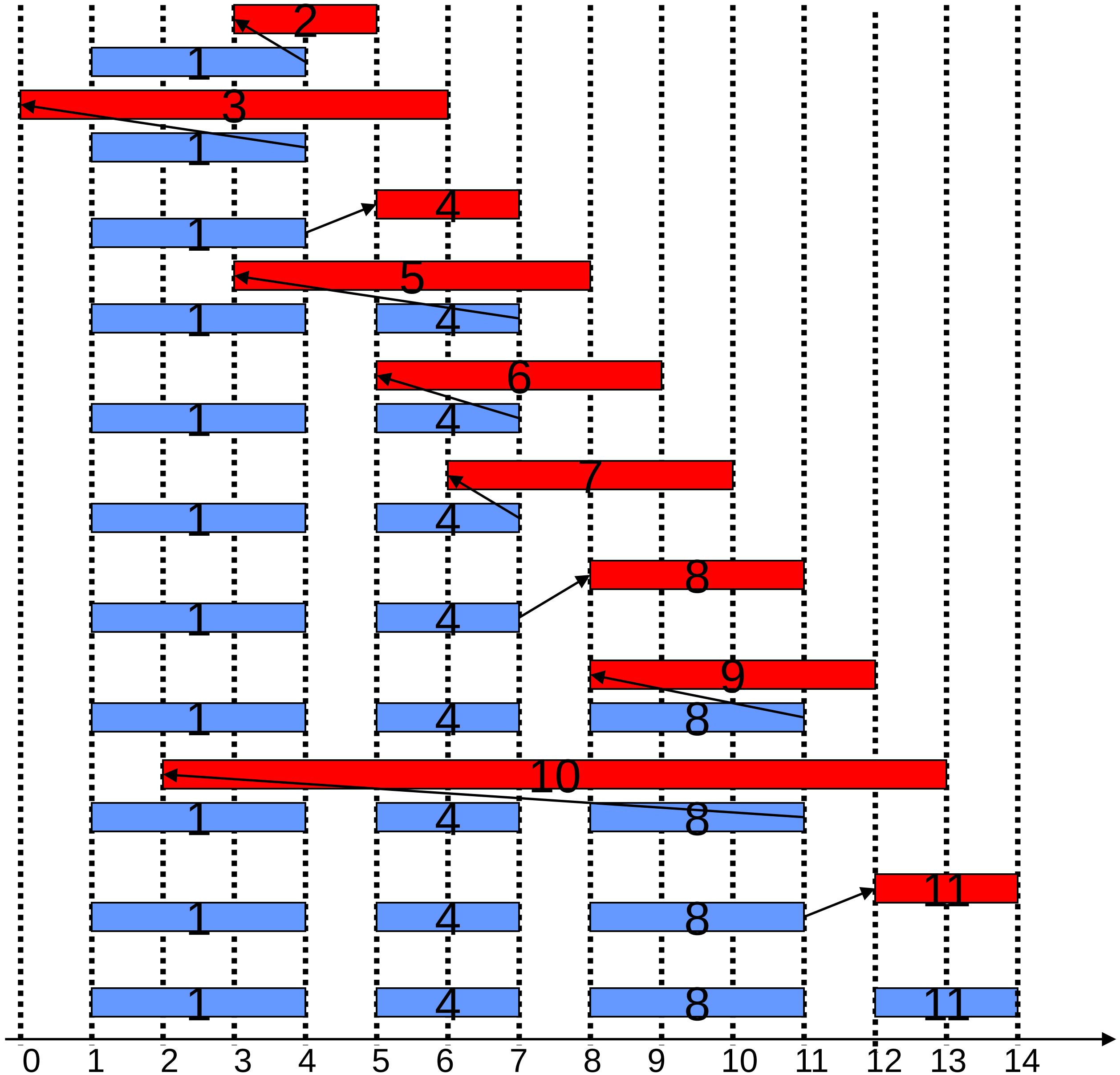
Assumption:

Requests are sorted in non-decreasing order of finish time:

$$f(1) \leq f(2) \leq f(3) \leq \ldots \leq f(n)$$

Greedy*: Example

	$s(i)$	$f(i)$
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



Greedy*

Input: n requests given by intervals $[s(i), f(i))$, $1 \leq i \leq n$, where $f(i) \leq f(i+1)$

Output: A maximum-size compatible subset of requests.

1. $A := \{1\}$;
2. $last := 1$; /* $last$ is the request added most recently */
3. **for** $i = 2$ **to** n **do**
4. **if** $s(i) \geq f(last)$ **then**
5. $A := A \cup \{i\}$;
6. $last := i$;
7. **endif;**
8. **endfor;**
9. **return** A;

Running time: $O(n)$

Analysis of Greedy*

Lemma: Set A is a compatible set of requests.

Proof: Requests are added to A in order of increasing finish times. A request i added to A does not overlap with the last request added to A, and hence with no request contained in A.

Let O be an optimal set of intervals.

We will prove $|A| = |O|$.

Specifically, we will compare a partial solution of Greedy* to an initial segment of O, and show that Greedy* does at least as good. Intuitively, Greedy* always „stays ahead".

Analysis of Greedy*

Let $A = \{i_1,\ldots,i_k\}$. Requests $i_1 < \ldots < i_k$ were added in this order.

Let $O = \{j_1,\ldots,j_m\}$. Requests $j_1 < \ldots < j_m$ are compatible.

We will prove that $k = m$.

Intuition of Greedy*: Resource becomes available as soon as possible.

Lemma: For $r = 1,\ldots,k$, there holds $f(i_r) \leq f(j_r)$.

Proof: Induction on r.

r=1: Greedy selects request 1, which has the earliest finish time among all requests.

Assume that the lemma holds for r-1. For the induction step we consider integer r.

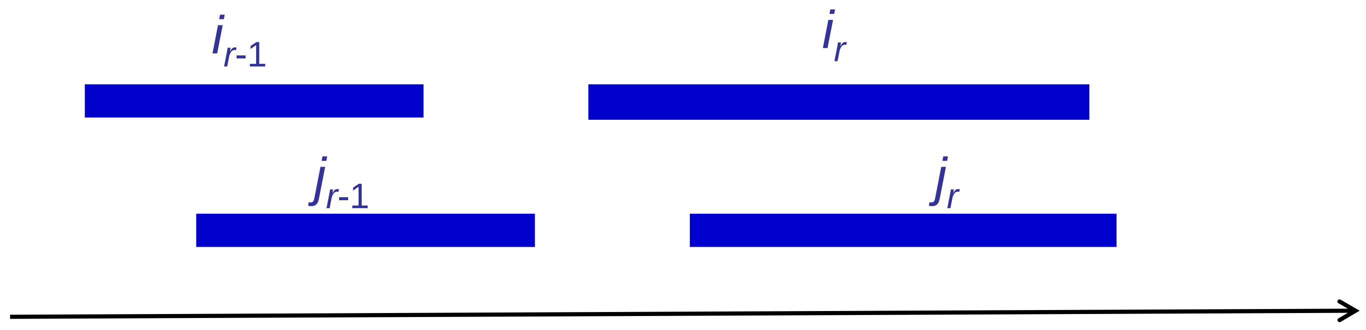
Analysis of Greedy*

There holds $f(i_{r-1}) \leq f(j_{r-1})$.

Also $f(j_{r-1}) \leq s(j_r)$, which implies $f(i_{r-1}) \leq s(j_r)$, and Greedy* could have added request j_r to A.

The general situation is depicted in the figure below.

If $i_r \neq j_r$, then $f(i_r) \leq f(j_r)$ because Greedy* considers requests in order of non-decreasing finish times.



Analysis of Greedy*

Theorem: Greedy* returns an optimal set A.

Proof: Suppose that A is not an optimal set. Then $m > k$.

Using the above lemma with $r=k$ we get $f(i_k) \leq f(j_k)$.

Request j_{k+1} in O satisfies $f(j_k) \leq s(j_{k+1})$ and thus $f(i_k) \leq f(j_k) \leq s(j_{k+1})$.

Hence Greedy* would have added request j_{k+1} or some other request to A.

Extensions:

Weighted problem: Request i has a value v_i. Maximize the total value of the selected requests.

Online setting: Requests arrive one by one. A scheduler has to accept/reject requests without knowledge of any future requests.

Interval Partitioning Problem

Many identical resources are available. Schedule all the requests using as few resources as possible.

Problem:

Set $S = \{1,\dots,n\}$ of n requests. Pool of identical resources.

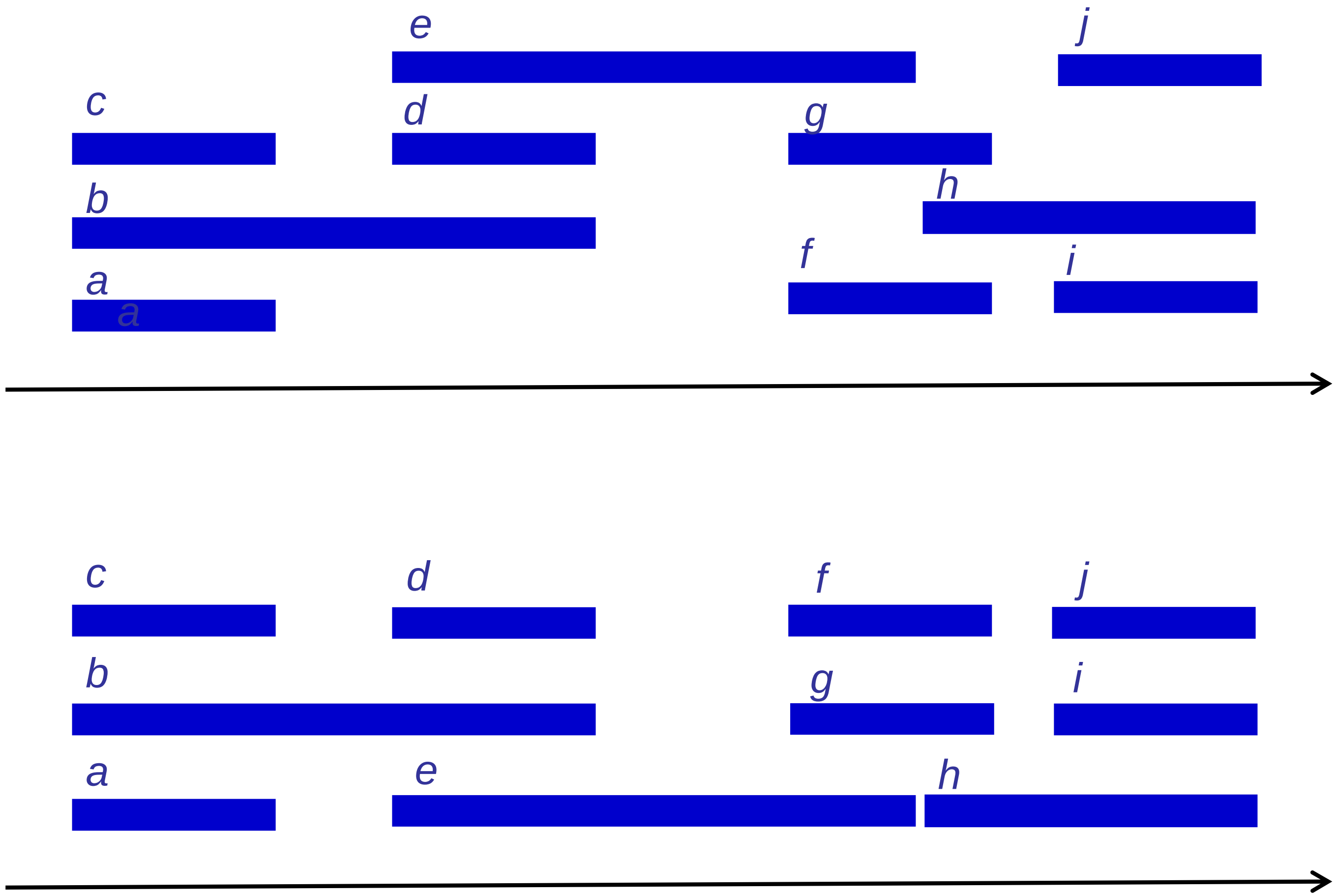
Request i: $[s(i), f(i))$ $s(i)$ = start time $f(i)$ = finish time.

Goal: Schedule all the requests feasibly so as to minimize the number of required resources.

Applications:

- Schedule requests for a classroom using as few classrooms as possible.
- Schedule jobs that need to be processed for a specific period of time on a small set of machines.
- Route requests that need to be allocated bandwidth on a fiber-optic cable.

Interval Partitioning: Example



Interval Partitioning Problem

Depth of a set of intervals: maximum number of intervals that pass over any single point on the time-line.

Lemma: In any instance of Interval Partitioning, the number of resources needed is a least the depth of the set of intervals.

Proof: Suppose that intervals $I_1,\ldots,I_d$ all pass over a common point on the time-line. Then they must be scheduled on different resources.

- Does there exist a polynomial time algorithm for Interval Partitioning?
- Is there always a schedule using a number of resources equal to the depth?

Greedy

Let d be the depth of the set of intervals.

Process intervals in order of non-decreasing start times. Each interval is assigned a label, where labels come of the set of numbers $\{1,\dots,d\}$. Overlapping intervals are labeled with different numbers.

Each number can be interpreted as the name of a resource. The label of an interval indicates to which resource the interval is assigned.

Greedy

Input: n requests/intervals $I_i = [s(i), f(i))$, $1 \le i \le n$, where $s(i) \le s(i+1)$

Output: A labeling of the intervals with numbers from $\{1,2,\ldots,d\}$. Overlapping intervals are labeled with different numbers.

```
for i = 1 to n do
    L := {1,…,d};
    for j = 1 to i-1 do
        if I_j overlaps with I_i then
            Remove the label of I_j from L;
        endif;
    endfor;
    if L ≠ ∅ then
        Assign to I_i any label from L;
    else
        Leave I_i unlabeled;
    endif;
endfor;
```

Running time: $O(n^2)$

Analysis

Lemma: The Greedy algorithm assigns to every interval a label. No two overlapping interals receive the same label.

Proof: We first argue that each interval is assigned a label.

Consider interval I_i and suppose that there exist exactly t intervals among $I_1,\ldots I_{i-1}$ that overlap with I_i. These t intervals, together with I_i, pass over a common point on the time-line. Hence $t+1 \leq d$, and $t \leq d-1$. Therefore, in line 8 of Greedy, L is non-empty.

Line 5 ensures that overlapping intevals do not receive the same label.

Theorem: Greedy schedules every interval on a resource, using a number of resources equal to the depth of the set of intervals. This is the optimal number of resources needed.

Scheduling to minimize lateness

Schedule n requests/jobs on a single resource so as to minimize the maximum lateness.

Problem:

n jobs $J_1,..,J_n$ that are available at a common start time s=0.

Job J_i has a length t_i and a deadline d_i, $1 \leq i \leq n$.

1 resource.

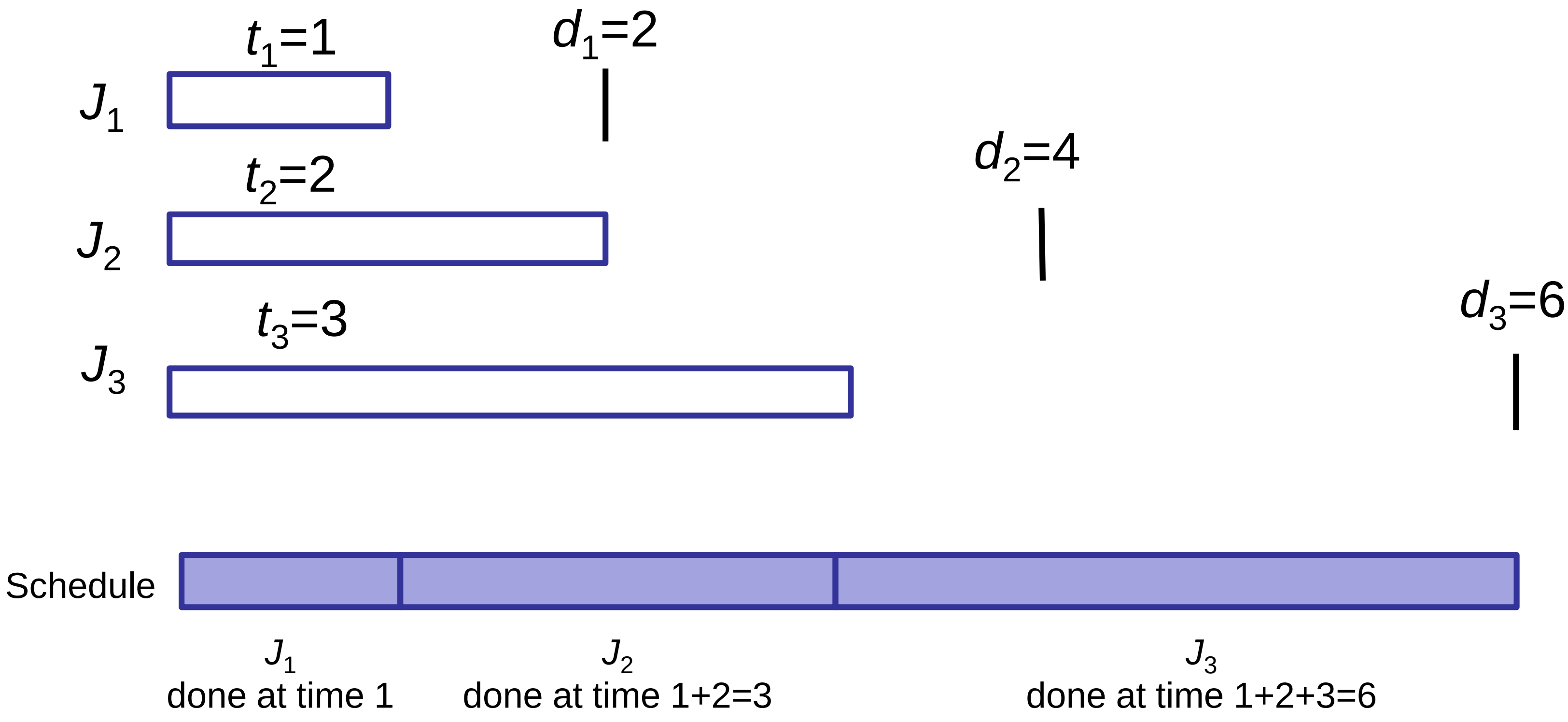
Job J_i must be assigned an interval $[s(i),f(i))$ with $f(i) = s(i) + t_i$.

Lateness of J_i is $l_i = \max\{0, f(i) - d_i\}$.

Different jobs must be assigned non-overlapping intervals.

Goal: Construct a schedule that minimizes $L = \max_{1 \leq i \leq n} l_i$.

Example



t_1=1
d_1=2
J_1
t_2=2
d_2=4
J_2
t_3=3
d_3=6
J_3
Schedule
J_1
done at time 1
J_2
done at time 1+2=3
J_3
done at time 1+2+3=6

Greedy algorithms

Greedy 1: Schedule jobs in order of increasing length.

Not optimal. J_1: t_1=1 d_1=100 J_2: t_2=10 d_2=10

Greedy 2: Schedule jobs in order of increasing slack time d_i - t_i.

Not optimal. J_1: t_1=1 d_1=2 J_2: t_2=10 d_2=10

Earliest Deadline First (EDF): Schedule jobs in order of increasing deadlines.

1. Sort/number the jobs $J_1,\ldots,J_n$ such that $d_1 \le \ldots \le d_n$.
2. f := 0;
3. **for** i = 1 **to** n **do**
4. Assign J_i to the time interval from $s(i) := f$ to $f(i) := f + t_i$;
5. $f := f + t_i$;
6. **endfor;**

Analysis EDF

Idle time: Gap time in the schedule when the machine is not working, yet there are jobs left.

EDF constructs a schedule with no idle time.

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

A = schedule constructed by EDF

O = optimal schedule

Idea: Repeatedly modify O so that it is eventually identical to A. In each step optimality is preserved.

Inversion in a schedule: Pair of jobs J_i and J_j such that J_i is scheduled before J_j but $d_j < d_i$.

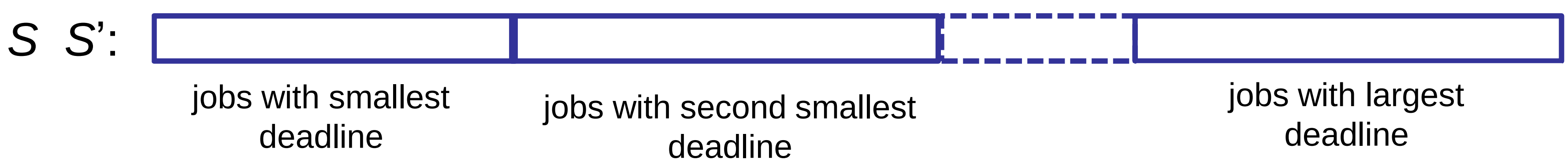
Analysis EDF

Lemma 1: All schedules with no inversions and no idle time have the same maximum lateness.

Proof: Let *S* and *S*‘ be two different schedules that have neither inversions nor idle time. The schedules only differ in the order in which jobs with identical deadlines are scheduled.

Consider jobs with a common deadline *d*. They are scheduled consecutively after all jobs with earlier deadlines and before all jobs with later deadlines.

Among the jobs with deadline *d*, the one scheduled last has the greatest lateness. This lateness does not depend on the order of the jobs.



Analysis EDF

Lemma 2: There exists an optimal schedule that has no inversions and no idle time.

Proof: By the above observation there exists an optimal schedule O with no idle time.

Statement (a): If O has an inversion, then there exist two jobs J_j and J_k such that J_k is scheduled immediately after J_j and $d_k < d_j$.

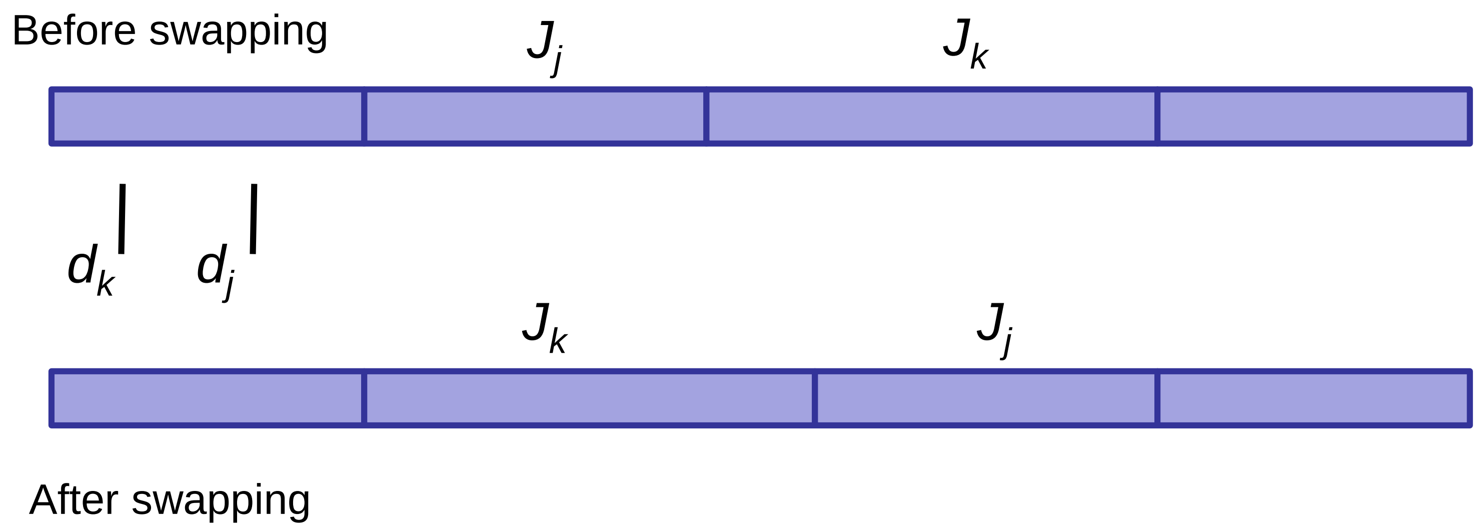
For the proof of the statement, consider an inversion where job J_a is scheduled sometime before J_b and $d_b < d_a$. In the schedule, starting at J_a traverse the subsequent jobs until reaching a point where the deadline encountered decreases for the first time.

Now suppose that O has at least one inversion and let J_j and J_k be two jobs as specified in Statement (a). Swap the two jobs and let O‘ be the new schedule. We argue that the maximum lateness does not increase.

Analysis EDF

Obviously, only the lateness of J_j can increase. The new lateness l_j' satisfies $l_j' = \max\{0, f(k) - d_j\} \leq \max\{0, f(k) - d_k\} = l_k \leq L$, where l_k denotes the lateness of J_k in O and L is the maximum lateness of this former schedule.

Thus the swap perserves optimality of the schedule. After at most $\binom{n}{2}$ swaps we obtain a schedule with the properties of the lemma.



Analysis EDF

Theorem: The schedule constructed by EDF has an optimal maximum lateness.

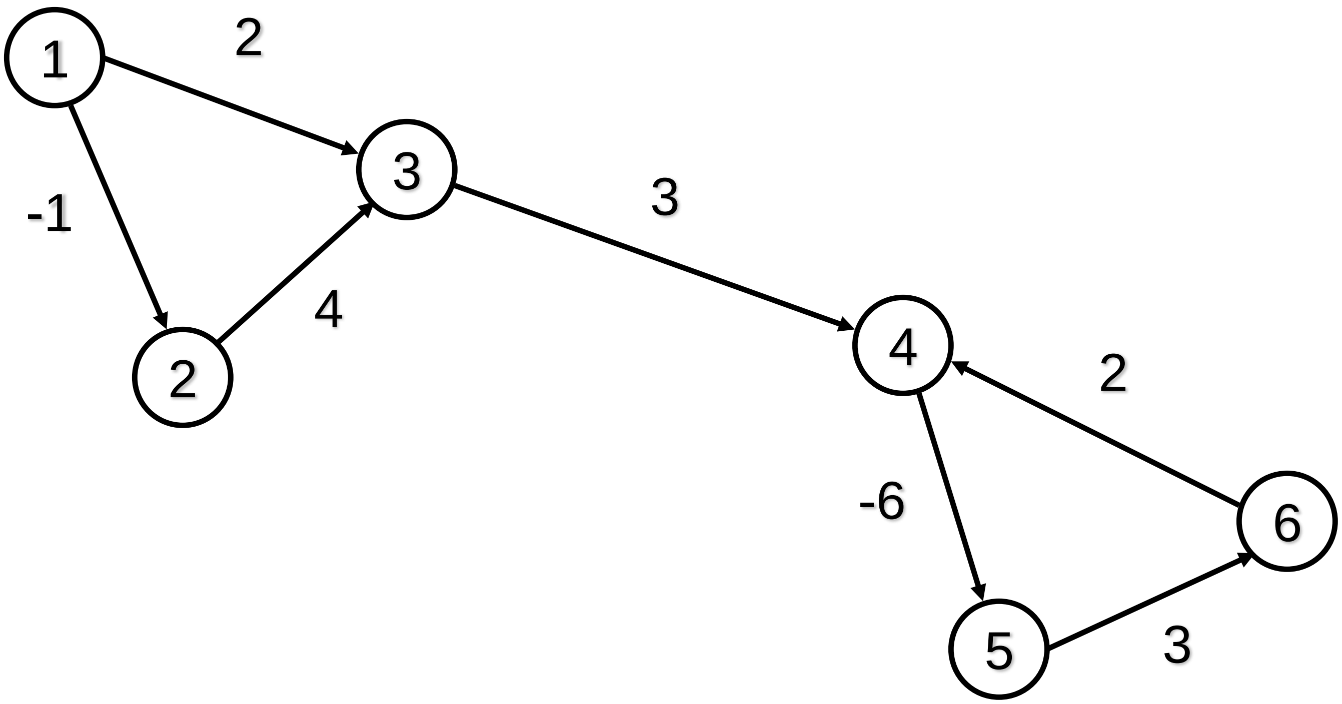
Proof: By Lemma 2 there exists an optimal schedule that has no inversions and no idle time. By Lemma 1 all schedules with these two properties have the same maximum lateness. Hence the schedule by EDF is optimal.

Extension: Assume that each job J_i, additonally, has a release time r_i. The analysis of EDF crucially uses of the fact that all jobs are available at a common start time.

Discussion: Shortest-paths problem

Directed graph $G = (V, E)$

Cost function $c : E \rightarrow \mathbb{R}$



Distance between two vertices

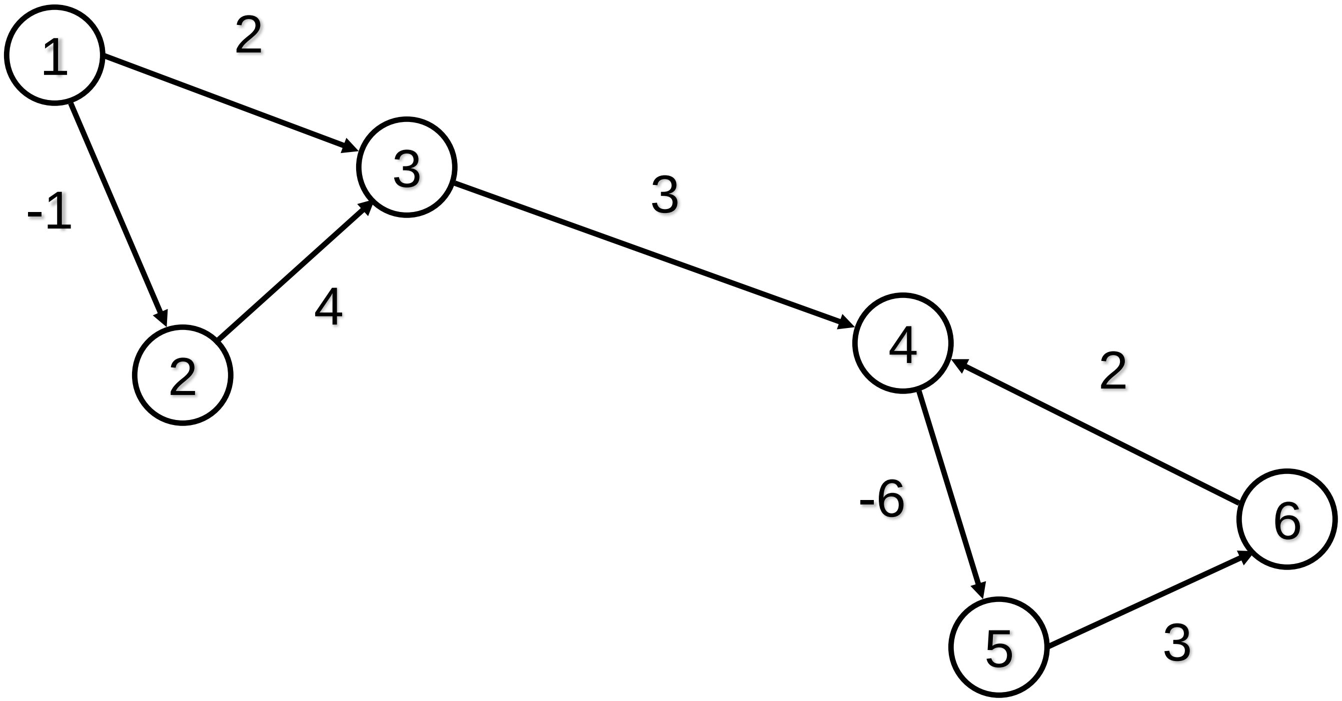
Cost/length of a path $P = v_0, v_1, \ldots, v_l$ from u to v :

$$c(P) = \sum_{i=0}^{l-1} c(v_i, v_{i+1})$$

Distance between u and v (not always defined):

$dist(u,v) = \inf \{ c(P) \mid P \text{ is a path from } u \text{ to } v \}$

Example



dist(1,2) = -1

dist(1,3) = 2

dist(3,1) = ∞

dist(3,4) = -∞

Single-source shortest paths problem

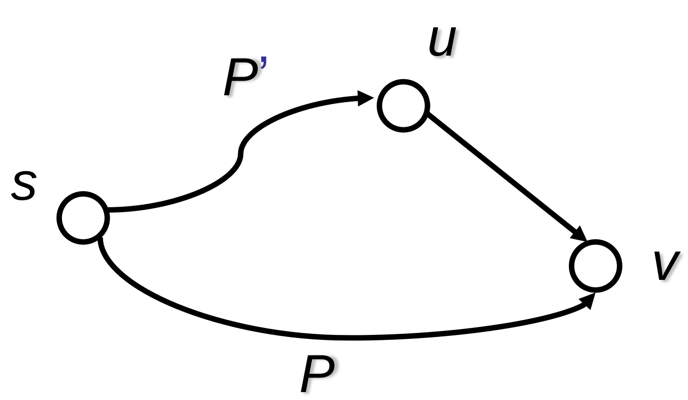
Input: Network $G = (V, E, c)$, $c : E \to \mathbb{R}$, vertex s

Output: $dist(s,v)$ for all $v \in V$

Observation: The function *dist* satisfies the triangle inequality.

For any edge $(u,v) \in E$:

$$dist(s,v) \le dist(s,u) + c(u,v)$$



P = shortest path from s to v

P' = shortest path from s to u

Greedy approach to an algorithm

1. Overestimate the function *dist*

$$dist(s,v) = \begin{cases} 0 & \text{if } \; v = s \\ \infty & \text{if } \; v \neq s \end{cases}$$

2. While there exists an edge $e = (u,v)$ with

$$dist(s,v) > dist(s,u) + c(u,v)$$

set $\quad dist(s,v) \leftarrow dist(s,u) + c(u,v)$

Generic algorithm

1. DIST[s] $\leftarrow 0$;
2. **for all** $v \in V \setminus \{s\}$ **do** DIST[v] $\leftarrow \infty$ **endfor;**
3. **while** $\exists\, e = (u,v) \in E$ with DIST[v] > DIST[u] + $c(u,v)$ **do**
4. Choose such an edge $e = (u,v)$;
5. DIST[v] $\leftarrow$ DIST[u] + $c(u,v)$;
6. **endwhile;**

Questions:

1. How can we efficiently check in line 3 if the triangle inequality is violated?
2. Which edge shall we choose in line 4?

Solution

Maintain a set U of all those vertices that might have an outgoing edge violating the triangle inequality.

- Initialize $U = \{s\}$
- Add vertex v to U whenever DIST[v] decreases.

1. Check if the triangle inequality is violated: $U \neq \emptyset$?
2. Choose a vertex from U and restore the triangle inequality for all outgoing edges (edge relaxation).

Refined Greedy algorithm

```
DIST[s] ← 0;
for all v ∈ V \ {s} do DIST[v] ← ∞ endfor;
U ← {s};
while U ≠ ∅ do
    Choose a vertex u ∈ U and delete it from U;
    for all e = (u,v) ∈ E do
        if DIST[v] > DIST[u] + c(u,v) then
            DIST[v] ← DIST[u] + c(u,v);
            U ← U ∪ {v};
        endif;
    endfor;
endwhile;
```

Efficient implementations

- Non-negative networks (only non-negative edge costs)
  U is a priority queue. Dijkstra‘s algorithm. $O(m + n \log n)$

- Networks without negative-cost cycles
  U is a queue. Bellman-Ford algorithm. $O(n \cdot m)$

- Acyclic networks
  U is a topological sorting of V. $O(n + m)$

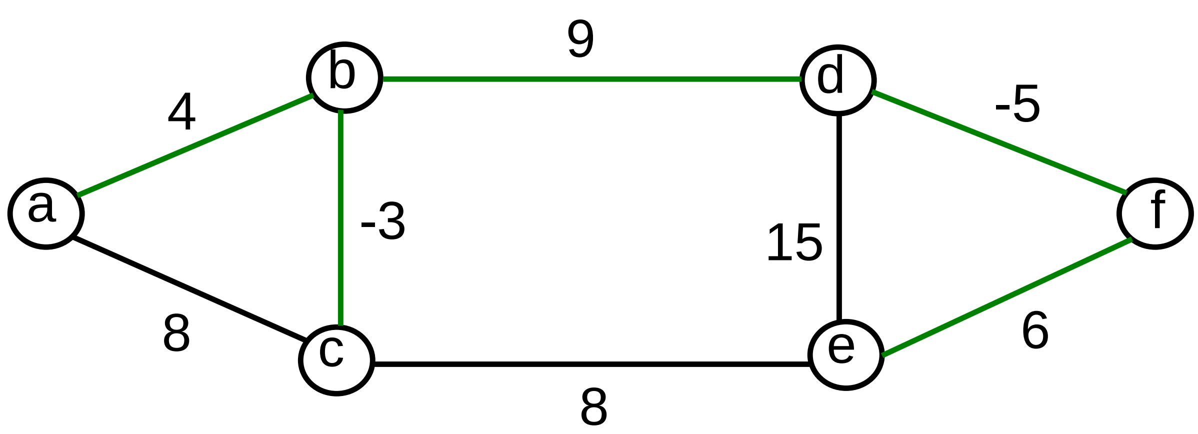
$n = |V| \qquad m = |E|$

Discussion: Minimum spanning trees

$G = (V, E)$ undirected graph $\qquad$ $w: E \to R$ weight function

Minimum spanning tree: Tree $T \subseteq E$ (connected, acyclic subgraph) that connects all vertices in V and whose total weight $w(T)$ is minimum.

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$



Greedy algorithms

- Kruskal's algorithm: Grow a forest. Initially each tree consists of a single vertex. In each step add a minimum-weight edge that connects different trees.

- Prim's algorithm: Grow a single tree. In each step add a minimum-weight edge maintaining the tree structure.