Algorithm Design
Greedy Algorithms

Prof. Dr. Brahim Hnich

March 25, 2013
Introduction
Outline

- Introduction
- Interval scheduling
Outline

▶ Introduction
▶ Interval scheduling
▶ Scheduling all intervals
Outline

- Introduction
- Interval scheduling
- Scheduling all intervals
- Scheduling to minimize lateness
Greed is good. Greed is right. Greed works. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit. - Gordon Gecko (Michael Douglas)
We investigate the pros and cons of short-sighted greed in the design of algorithms.
Introduction

- We investigate the pros and cons of short-sighted greed in the design of algorithms
- We approach a number of computational problems with a recurring set of questions
Introduction

- We investigate the pros and cons of short-sighted greed in the design of algorithms
- We approach a number of computational problems with a recurring set of questions
  1. Is greed good?
Introduction

- We investigate the pros and cons of short-sighted greed in the design of algorithms
- We approach a number of computational problems with a recurring set of questions
  1. Is greed good?
  2. Does greed work?
Greedy algorithm: An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion.
**Greedy algorithm**: An algorithm is *greedy* if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion. Can often design many different greedy for the same problem.
Greedy algorithm: An algorithm is *greedy* if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion. Can often design many different greedy for the same problem. Each algorithm locally and incrementally optimizes some different measure on its way to a solution.
Introduction

▶ When a greedy algorithm succeeds in solving a problem, then
Introduction

- When a greedy algorithm succeeds in solving a problem, then
  - Something is interesting and useful about the structure of the problem
When a greedy algorithm succeeds in solving a problem, then

- Something is interesting and useful about the structure of the problem
- There is a local decision rule that one can use to construct optimal solutions
Interval scheduling

- **Input.** Set of jobs with start times and finish times.
- **Goal.** Find maximum cardinality subset of mutually compatible jobs. I.e., jobs that do not overlap.
Set of requests: \( \{1, \ldots, n\} \)
Set of requests: \( \{1, \ldots, n\} \)

the \( i^{th} \) request is an interval of time \([s(i), f(i)]\)
Interval scheduling: notation

- Set of requests: \( \{1, \ldots, n\} \)
- The \( i^{th} \) request is an interval of time \([s(i), f(i)]\)
- A subset of requests is \textit{compatible} if no two of them overlap in time
Set of requests: \( \{1, \ldots, n\} \)

The \( i^{th} \) request is an interval of time \([s(i), f(i)]\)

A subset of requests is *compatible* if no two of them overlap in time.

Compatible sets of maximum size are called *optimal*.
Interval scheduling: Designing a greedy algorithm

- Basic idea
Interval scheduling: Designing a greedy algorithm

- Basic idea
  1. Use a simple rule to select a first request $i_1$
Interval scheduling: Designing a greedy algorithm

- Basic idea
  1. Use a simple rule to select a first request $i_1$
  2. Reject all requests that are not compatible with $i_1$
Basic idea
1. Use a simple rule to select a first request $i_1$
2. Reject all requests that are not compatible with $i_1$
3. Select the next request $i_2$
Interval scheduling: Designing a greedy algorithm

Basic idea
1. Use a simple rule to select a first request $i_1$
2. Reject all requests that are not compatible with $i_1$
3. Select the next request $i_2$
4. Reject all requests that are not compatible with $i_2$
Basic idea

1. Use a simple rule to select a first request $i_1$
2. Reject all requests that are not compatible with $i_1$
3. Select the next request $i_2$
4. Reject all requests that are not compatible with $i_2$
5. ... and so on till we run out of requests
Most obvious rule:
Interval scheduling: attempt 1

- Most obvious rule:
  - Select the available request that starts earliest

Motivation: Resource starts being used as quickly as possible

Does not yield an optimal solution
Most obvious rule:

- Select the available request that starts earliest
- I.e., the one with the minimal start time $s(i)$
Interval scheduling: attempt 1

- Most obvious rule:
  - Select the available request that starts earliest
  - I.e., the one with the minimal start time $s(i)$
- Motivation: Resource starts being used as quickly as possible
Interval scheduling: attempt 1

- Most obvious rule:
  - Select the available request that starts earliest
  - I.e., the one with the minimal start time $s(i)$
- Motivation: Resource starts being used as quickly as possible
- Does not yield an optimal solution
Interval scheduling: attempt 1

- Most obvious rule:
  - Select the available request that starts earliest
  - I.e., the one with the minimal start time $s(i)$
- Motivation: Resource starts being used as quickly as possible
- Does not yield an optimal solution
Another rule:

- Select the request that requires the smallest interval of time
- I.e., the one with the minimal $f(i) - s(i)$
- Somewhat a better rule, but
- Does not yield an optimal solution
Another rule:

- Select the request that requires the smallest interval of time
Another rule:
- Select the request that requires the smallest interval of time
- I.e., the one with the minimal $f(i) - s(i)$
Another rule:
- Select the request that requires the smallest interval of time
- I.e., the one with the minimal $f(i) - s(i)$

Somewhat a better rule, but
Another rule:

- Select the request that requires the smallest interval of time
- i.e., the one with the minimal $f(i) - s(i)$

Somewhat a better rule, but

Does not yield an optimal solution
Another rule:
- Select the request that requires the smallest interval of time
- I.e., the one with the minimal $f(i) - s(i)$

Somewhat a better rule, but
- Does not yield an optimal solution
A more elaborate rule:
A more elaborate rule:
- Select the request that has the smallest number of noncompatible requests
Interval scheduling: attempt 3

- A more elaborate rule:
  - Select the request that has the smallest number of noncompatible requests
  - I.e., the interval with the fewest conflicts
A more elaborate rule:
   - Select the request that has the smallest number of noncompatible requests
   - I.e., the interval with the fewest conflicts

Somewhat a better rule, but
Interval scheduling: attempt 3

- A more elaborate rule:
  - Select the request that has the smallest number of noncompatible requests
  - I.e., the interval with the fewest conflicts

- Somewhat a better rule, but

- Does not yield an optimal solution
A more elaborate rule:

- Select the request that has the smallest number of noncompatible requests
- I.e., the interval with the fewest conflicts

Somewhat a better rule, but

Does not yield an optimal solution

![Interval scheduling diagram]
Figure 4.1 Some instances of the Interval Scheduling Problem on which natural greedy algorithms fail to find the optimal solution. In (a), it does not work to select the interval that starts earliest; in (b), it does not work to select the shortest interval; and in (c), it does not work to select the interval with the fewest conflicts.
Interval scheduling: optimal rule

Optimal rule:

Select the request that finishes first

I.e., request \( i \) for which \( f(i) \) is as small as possible

Motivation: Ensure that our resource becomes free as soon as possible while still satisfying one request

Does yield an optimal solution!
Interval scheduling: optimal rule

- Optimal rule:
  - Select the request that finishes first

Motivation: Ensure that our resource becomes free as soon as possible while still satisfying one request.

Does yield an optimal solution!
Interval scheduling: optimal rule

- Optimal rule:
  - Select the request that finishes first
  - I.e., request $i$ for which $f(i)$ is as small as possible

Motivation: Ensure that our resource becomes free as soon as possible while still satisfying one request.

This yields an optimal solution!
Interval scheduling: optimal rule

- **Optimal rule:**
  - Select the request that finishes first
  - I.e., request $i$ for which $f(i)$ is as small as possible

- **Motivation:** Ensure that our resource becomes free as soon as possible while still satisfying one request
Optimal rule:

- Select the request that finishes first
- I.e., request $i$ for which $f(i)$ is as small as possible

Motivation: Ensure that our resource becomes free as soon as possible while still satisfying one request

Does yield an optimal solution!
Initially let $R$ be the set of all requests, and let $A$ be empty

While $R$ is not yet empty

  Choose a request $i \in R$ that has the smallest finishing time
  Add request $i$ to $A$
  Delete all requests from $R$ that are not compatible with request $i$

EndWhile

Return the set $A$ as the set of accepted requests
Let $i_1, \ldots, i_k$ be the set of requests in $A$ in the order they were added to $A$. 
Let $i_1, \ldots, i_k$ be the set of requests in $A$ in the order they were added to $A$

Let $\mathcal{O} = \{j_1, \ldots, j_m\}$ be an optimal set of requests
Let $i_1, \ldots, i_k$ be the set of requests in $A$ in the order they were added to $A$

Let $O = \{j_1, \ldots, j_m\}$ be an optimal set of requests

Goal: prove that $k = m$
Theorem

For all indices $r \leq k$ we have $f(i_r) \leq f(j_r)$

Proof.

(By induction.) For $r = 1$, the statement is obviously true. Let $r > 1$. We assume as our inductive hypothesis that the statement is true for $r - 1$. We will show it is true for $r$. By inductive hypothesis, we have $f(i_{r-1}) \leq f(j_{r-1})$. For the algorithm’s $r^{th}$ interval not to finish earlier as well, it would need to "fall behind" as shown in Figure. But then algorithm would at worst choose $j_r$ instead!

![Diagram](image)

**Figure 4.3** The inductive step in the proof that the greedy algorithm stays ahead.
Theorem
The greedy algorithm returns an optimal set $A$

Proof.
(By contradiction.) If $A$ is not optimal, then an optimal set $\mathcal{O}$ must have more requests. Using previous result with $r = k$, we have $f(i_k) \leq f(j_k)$. Since $m > k$, there is request $j_{k+1}$ in $\mathcal{O}$ that starts after $j_k$ and hence after $i_k$ ends. So after deleting all requests not compatible with $i_1, \ldots, i_k$, we still have $j_{k+1}$. But the greedy algorithms stops with request $i_k$, and its only possible when $R$ is empty.

\hfill $\square$
Sort in $O(n \log n)$ the set of requests in order of finishing time.
Interval scheduling: Implementation

- Sort in $O(n \log n)$ the set of requests in order of finishing time
- In $O(n)$ time, we construct an array $S[1 \ldots n]$ where $S[i]$ is $s(i)$
Interval scheduling: Implementation

- Sort in $O(n \log n)$ the set of requests in order of finishing time.
- In $O(n)$ time, we construct an array $S[1 \ldots n]$ where $S[i]$ is $s(i)$.
- In $O(n)$ we construct $R$ as follows:
Interval scheduling: Implementation

- Sort in $O(n \log n)$ the set of requests in order of finishing time
- In $O(n)$ time, we construct an array $S[1 \ldots n]$ where $S[i]$ is $s(i)$
- In $O(n)$ we construct $R$ as follows:
  - Select the first interval
Interval scheduling: Implementation

- Sort in $O(n \log n)$ the set of requests in order of finishing time
- In $O(n)$ time, we construct an array $S[1 \ldots n]$ where $S[i]$ is $s(i)$
- In $O(n)$ we construct $R$ as follows:
  - Select the first interval
  - Iterate through until reaching an interval for which $s(j) \geq f(1)$
Interval scheduling: Implementation

- Sort in $O(n \log n)$ the set of requests in order of finishing time
- In $O(n)$ time, we construct an array $S[1 \ldots n]$ where $S[i]$ is $s(i)$
- In $O(n)$ we construct $R$ as follows:
  - Select the first interval
  - Iterate through until reaching an interval for which $s(j) \geq f(1)$
  - Select interval $j$
Interval scheduling: Implementation

- Sort in $O(n \log n)$ the set of requests in order of finishing time.
- In $O(n)$ time, we construct an array $S[1 \ldots n]$ where $S[i]$ is $s(i)$.
- In $O(n)$ we construct $R$ as follows:
  - Select the first interval
  - Iterate through until reaching an interval for which $s(j) \geq f(1)$
  - Select interval $j$
  - Repeat until we have no more intervals left.
Scheduling all intervals: Interval partitioning

- **Input.** Set of jobs with start times and finish times; Set of identical resources.
- **Goal.** Schedule *all* the requests using as few resources as possible.

![Diagram](image)

**Figure 4.4** (a) An instance of the Interval Partitioning Problem with ten intervals (a through j). (b) A solution in which all intervals are scheduled using three resources: each row represents a set of intervals that can all be scheduled on a single resource.
**Definition.** The *depth* of a set of intervals is the maximum number that pass over any single point on the time-line.
**Definition.** The *depth* of a set of intervals is the maximum number that pass over any single point on the time-line.

**Observation.** Number of resources needed is at least the depth of the set of intervals.
Definition. The depth of a set of intervals is the maximum number that pass over any single point on the time-line.

Observation. Number of resources needed is at least the depth of the set of intervals.

Question. Does there always exist a schedule equal to depth of intervals?
Sort the intervals by their start times, breaking ties arbitrarily.
Let $I_1, I_2, \ldots, I_n$ denote the intervals in this order.
For $j = 1, 2, 3, \ldots, n$
    For each interval $I_i$ that precedes $I_j$ in sorted order and overlaps it
        Exclude the label of $I_i$ from consideration for $I_j$
    Endfor
    If there is any label from $\{1, 2, \ldots, d\}$ that has not been excluded then
        Assign a nonexcluded label to $I_j$
    Else
        Leave $I_j$ unlabeled
    Endif
Endfor
**Theorem**

*Every interval will be assigned a label, and no two overlapping intervals will receive the same label.*

**Proof.**

It's easy to see that every interval end up with a label. Now, consider any two intervals $I$ and $I'$ that overlap and suppose $I$ precedes $I'$ in the sorted order. Then when $I'$ is considered by the algorithm, $I$ is in the set of intervals whose labels are excluded from consideration; thus, the algorithm will not assign to $I'$ the same label as $I$. \[\square\]
Theorem

The greedy algorithm 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

Input.

- 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 job $j$ 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 $L = \max l_j$. 

Scheduling to minimize lateness

▶ **Input.**
  - 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 $L = \max l_j$. 

Prof. Dr. Brahim Hnich
Algorithm Design Greedy Algorithms
Input.

- Single resource processes one job at a time;
- Job $j$ requires $t_j$ units of processing time and is due at time $d_j$;
- Lateness: $l_j = \max\{0, f_j - d_j\}$. 

Goal. Schedule all jobs to minimize maximum lateness $L = \max l_j$. 

Prof. Dr. Brahim Hnich
Algorithm Design  Greedy Algorithms
Scheduling to minimize lateness

- **Input.**
  - 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 $L = \max l_j$. 
Input.
- 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: $l_j = \max\{0, f_j - d_j\}$. 
Scheduling to minimize lateness

Input.
- 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
\[ L = \max l_j. \]
Figure 4.5 A sample instance of scheduling to minimize lateness.
A simple rule:

Consider jobs in ascending order of processing time $t_j$.

Motivation: Get the short jobs out of the way quickly.

Too simplistic, it completely ignores the deadlines.

Does not yield an optimal solution.
Scheduling to minimize lateness: attempt 1

- A simple rule:
  - Consider jobs in ascending order of processing time $t_j$. 

Motivation: Get the short jobs out of the way quickly

Too simplistic, it completely ignores the deadlines

Does not yield an optimal solution
Scheduling to minimize lateness: attempt 1

- A simple rule:
  - Consider jobs in ascending order of processing time $t_j$.
  - Motivation: Get the short jobs out of the way quickly.
A simple rule:
  - Consider jobs in ascending order of processing time $t_j$.

Motivation: Get the short jobs out of the way quickly

Too simplistic, it completely ignores the deadlines
A simple rule:

- Consider jobs in ascending order of processing time $t_j$.

Motivation: Get the short jobs out of the way quickly

Too simplistic, it completely ignores the deadlines

Does not yield an optimal solution
A simple rule:
- Consider jobs in ascending order of processing time $t_j$.

Motivation: Get the short jobs out of the way quickly

Too simplistic, it completely ignores the deadlines

Does not yield an optimal solution

counterexample

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>$d_j$</td>
<td>100</td>
<td>10</td>
</tr>
</tbody>
</table>
Another rule: smallest slack
Another rule: smallest slack

Consider jobs in ascending order of slack $d_j - t_j$. 
Another rule: smallest slack
  - Consider jobs in ascending order of slack $d_j - t_j$.

Does not yield an optimal solution
Another rule: smallest slack

- Consider jobs in ascending order of slack $d_j - t_j$.
- Does not yield an optimal solution
Another rule: smallest slack
   - Consider jobs in ascending order of slack $d_j - t_j$.

Does not yield an optimal solution
Scheduling to minimize lateness: attempt 2

- Another rule: smallest slack
  - Consider jobs in ascending order of slack $d_j - t_j$.
- Does not yield an optimal solution

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>$d_j$</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

counterexample
Optimal rule: Earliest deadline first
Scheduling to minimize lateness: optimal rule

- Optimal rule: Earliest deadline first
  - Consider jobs in ascending order of their deadlines.
Scheduling to minimize lateness: optimal rule

- Optimal rule: Earliest deadline first
  - Consider jobs in ascending order of their deadlines.
- Motivation: Jobs with earlier deadlines get completed earlier
Order the jobs in order of their deadlines
Assume for simplicity of notation that $d_1 \leq \ldots \leq d_n$
Initially, $f = s$
Consider the jobs $i = 1, \ldots, n$ in this order
  Assign job $i$ to the time interval from $s(i) = f$ to $f(i) = f + t_i$
  Let $f = f + t_i$
End
Return the set of scheduled intervals $[s(i), f(i)]$ for $i = 1, \ldots, n$
**Observation.** There exists an optimal schedule with no idle time.
Observation. There exists an optimal schedule with no *idle* time.

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
\end{array}
\]
Observation. There exists an optimal schedule with no *idle* time.

Observation. The greedy schedule has no idle time.
Definition. An *inversion* in schedule $S$ is a pair of jobs $i$ and $j$ such that: $d_i < d_j$ but $j$ is scheduled before $i$. 
**Definition.** An *inversion* in schedule $S$ is a pair of jobs $i$ and $j$ such that: $d_i < d_j$ but $j$ is scheduled before $i$. 

![Diagram showing an inversion in a schedule with jobs i and j, and an arrow indicating the inversion.](image-url)
Definition. An *inversion* in schedule $S$ is a pair of jobs $i$ and $j$ such that: $d_i < d_j$ but $j$ is scheduled before $i$.

Observation. The greedy schedule has no inversions.
Scheduling to minimize lateness: algorithm analysis

**Definition.** An *inversion* in schedule \( S \) is a pair of jobs \( i \) and \( j \) such that: \( d_i < d_j \) but \( j \) is scheduled before \( i \).

**Observation.** The 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.
Theorem

Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

Proof.

The only thing to worry about is the lateness of job $i$. After the swap, job $i$ finishes at time $f(j)$. If job $i$ is late in new schedule, then its lateness is $f(j) - d_i$. But, $i$ cannot be more late in new schedule than job $j$ in old schedule since $f(j) - d_i < f(j) - d_j$. □
Theorem

*Greedy schedule $S$ is optimal.*

**Proof.**

Define $S^*$ to be an optimal schedule that has the fewest number of inversions, and let’s see what happens. 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 $S^*$. □
Lessons learned

**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.
Lessons learned

**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.

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

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.

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

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.