Algorithm Design and Analysis

Lecture 8

- Max. lateness cont’d
- Optimal Caching

Adam Smith
Scheduling to Minimizing 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:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>$d_j$</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

Lateness = 2
Lateness = 0
Max lateness = 6
Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

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

- [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 processing time first]** Consider jobs in ascending order of processing time \( t_j \).

<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>

  **Counterexample**

- **[Smallest slack]** Consider jobs in ascending order of slack \( d_j - t_j \).

<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**
Minimizing Lateness: Greedy Algorithm

**Greedy algorithm.** Earliest deadline first.

Sort \( n \) jobs by deadline so that \( d_1 \leq d_2 \leq \ldots \leq d_n \)

\[
\begin{align*}
    & t \leftarrow 0 \\
    & \text{for } j = 1 \text{ to } n \\
    & \quad \text{Assign job } j \text{ to interval } [t, t + t_j] \\
    & \quad s_j \leftarrow t, f_j \leftarrow t + t_j \\
    & \quad t \leftarrow t + t_j \\
    & \text{output intervals } [s_j, f_j]
\end{align*}
\]

max lateness = 1
Minimizing Lateness: No Idle Time

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

<table>
<thead>
<tr>
<th>d = 4</th>
<th>d = 6</th>
<th>d = 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2</td>
<td>3 4 5 6</td>
<td>7 8 9 10 11</td>
</tr>
</tbody>
</table>

**Observation.** The greedy schedule has no idle time.
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.
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 $\ell$ be the lateness before the swap, and let $\ell'$ be it afterwards.

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

\[
\begin{align*}
\ell'_j &= f'_j - d_j \quad \text{(definition)} \\
&= f_i - d_j \quad (j \text{ finishes at time } f_i) \\
&\leq f_i - d_i \quad (i < j) \\
&\leq \ell_i \quad \text{(definition)}
\end{align*}
\]
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, 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 $S^*$
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.

**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.
4.3 Optimal Caching
Optimal Offline Caching

Caching.
- Cache with capacity to store \( k \) items.
- Sequence of \( m \) item requests \( d_1, d_2, \ldots, d_m \).
- 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.
Suggestions for greedy approaches?
Optimal Offline Caching: Farthest-In-Future

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

current cache:  

Current cache: a b c d e f

future queries:  

Future queries: g a b c e d a b b a c d e a f a d e f g h ...

<table>
<thead>
<tr>
<th>![cache miss]</th>
<th>![eject this one]</th>
</tr>
</thead>
</table>

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

**Intuition.** Can transform an unreduced schedule into a reduced one with no more cache misses. (Idea is to defer unreduced requests. Proof will be an exercise.)

\[
\begin{array}{cccc}
\text{a} & \text{a} & \text{b} & \text{c} \\
\text{a} & \text{a} & \text{x} & \text{c} \\
\text{c} & \text{a} & \text{d} & \text{c} \\
\text{d} & \text{a} & \text{d} & \text{b} \\
\text{a} & \text{a} & \text{c} & \text{b} \\
\text{b} & \text{a} & \text{x} & \text{b} \\
\text{c} & \text{a} & \text{c} & \text{b} \\
\text{a} & \text{a} & \text{b} & \text{c} \\
\text{a} & \text{a} & \text{b} & \text{c} \\
\end{array}
\]

an unreduced schedule

\[
\begin{array}{cccc}
\text{a} & \text{a} & \text{b} & \text{c} \\
\text{a} & \text{a} & \text{b} & \text{c} \\
\text{c} & \text{a} & \text{b} & \text{c} \\
\text{d} & \text{a} & \text{d} & \text{c} \\
\text{d} & \text{a} & \text{d} & \text{c} \\
\text{a} & \text{a} & \text{d} & \text{c} \\
\text{b} & \text{a} & \text{d} & \text{b} \\
\text{c} & \text{a} & \text{c} & \text{b} \\
\text{a} & \text{a} & \text{c} & \text{b} \\
\text{a} & \text{a} & \text{c} & \text{b} \\
\end{array}
\]

a reduced schedule
**Farthest-In-Future: Analysis**

**Theorem.** FF is optimal eviction algorithm.

**Pf.** (by induction on number or requests $j$)

**Invariant:** There exists an optimal reduced schedule $S$ that makes the same eviction schedule as $S_{FF}$ through the first $j+1$ requests.

Let $S = \text{opt. reduced}$ 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

\[
\begin{array}{c|c|c|c}
\text{j} & \text{same} & \text{e} & \text{f} \\
\hline
S & \text{same} & \text{e} & \text{f} \\
\hline
\text{j+1} & \text{same} & \text{e} & \text{d} \\
\hline
S & \text{same} & \text{d} & \text{f} \\
\hline
\end{array}
\]

- 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 \textbf{first} time after \( j+1 \) that \( S \) and \( S' \) take a different action, and let \( g \) be item requested at time \( j' \).

\[
\begin{array}{c|c|c}
\text{j'} & \text{same} & e \\
\text{S} & & \\
\end{array}
\begin{array}{c|c|c}
\text{same} & f \\
\text{S'} & & \\
\end{array}
\]

\[\uparrow\]

**Case 3a:** \( g = e \). Can't happen with Farthest-In-Future since there must be a request for \( f \) before \( e \).

**Case 3b:** \( g = f \). Element \( f \) can't be in cache of \( S \), so let \( e' \) be the element that \( S \) evicts.
- if \( e' = e \), \( S' \) accesses \( f \) from cache; now \( S \) and \( S' \) have same cache
- if \( e' \neq e \), \( S' \) evicts \( e' \) and brings \( e \) into the cache; now \( S \) and \( S' \) have the same cache

\[\uparrow\]

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 \textbf{first} time after \( j+1 \) that \( S \) and \( S' \) take a different action, and let \( g \) be item requested at time \( j' \).

\[ \uparrow \]

Case 3c: \( g \neq e, f \). \( S \) must evict \( e \).

Make \( S' \) evict \( f \); now \( S \) and \( S' \) have the same cache.

\[ \downarrow \]

otherwise \( S' \) would take the same action
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 most fundamental online problems in CS.

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.