4.1 Interval Scheduling

Interval Scheduling

Interval scheduling.
- Job $j$ starts at $s_j$ and finishes at $f_j$.
- Two jobs compatible if they don’t overlap.
- Goal: find maximum subset of mutually compatible jobs.

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 $s_j$.
- [Earliest finish time] Consider jobs in ascending order of finish time $f_j$.
- [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 $c_j$.
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

**Greedy algorithm.** Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

\[
\text{Sort jobs by finish times so that } f_1 \leq f_2 \leq \ldots \leq f_n.
\]

\[
\begin{align*}
A & \leftarrow \emptyset \\
\text{for } j = 1 \text{ to } n \{ \\
\quad & \text{if (job } j \text{ compatible with } A) \\
\quad & \qquad A \leftarrow A \cup \{j\} \\
\} \\
\text{return } A
\end{align*}
\]

**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, and let's see what happens.
- Let \(i_1, i_2, \ldots, i_k\) denote set of jobs selected by greedy.
- Let \(j_1, j_2, \ldots, j_m\) denote set of jobs in the optimal solution with \(i_1 = j_1, i_2 = j_2, \ldots, i_k = j_k\) for the largest possible value of \(r\).

**Greedy:**

```
  i_1   i_2   i_3   i_4
  \downarrow
```

**OPT:**

```
  j_1   j_2   j_3   j_4
  \uparrow
```

why not replace job \(j_{r+1}\) with job \(i_{r+1}\)?
4.1 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.

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

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

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

Q: Does there always exist a schedule equal to depth of intervals?
4.2 Scheduling to Minimize Lateness

Interval Partitioning: Greedy Algorithm

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

Sort intervals by starting time so that \( s_1 \leq s_2 \leq \ldots \leq s_n \).

\[
d \leftarrow 0 \quad \text{number of allocated classrooms}
\]

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.

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 \( L = \max \ell_j \).

<table>
<thead>
<tr>
<th>( j )</th>
<th>( s_j )</th>
<th>( t_j )</th>
<th>( d_j )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>14</td>
<td>19</td>
</tr>
</tbody>
</table>

Ex: lateness = 0 \( \downarrow \) lateness = 2 \( \downarrow \) max lateness = 6

<table>
<thead>
<tr>
<th>( d_j )</th>
<th>9</th>
<th>8</th>
<th>15</th>
<th>6</th>
<th>14</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

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 \( s_j + t_j \).
- Key observation \( \Rightarrow \) all schedules use \( \geq d \) classrooms. *
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 Algorithm

Greedy algorithm. Earliest deadline first.

Sort $n$ jobs by deadline so that $d_1 \leq d_2 \leq \ldots \leq d_n$.

$t \leftarrow 0$

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 $[s_j, f_j]$

max lateness = 1

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: No Idle Time

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

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

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 $i$.  

<table>
<thead>
<tr>
<th>before swap</th>
<th>j</th>
<th>i</th>
</tr>
</thead>
</table>

**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: 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.

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 $i$.  

<table>
<thead>
<tr>
<th>before swap</th>
<th>j</th>
<th>i</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>after swap</th>
</tr>
</thead>
</table>

- $f'_{j} = f_{j} - d_j$ (definition)
- $f'_{j} = f_{j} - d_j$ (j finishes at time $f$)
- $f'_{j} = f_{j} - d_j$ (i finishes at time $f$)
4.3 Optimal Caching

Optimal Offline Caching

**Caching.**
- Cache with capacity to store $k$ items.
- Sequence of $m$ item requests $d_1, d_2, ..., d_m$.
- 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.

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:  a b c d e f

future queries:  g a b c e d a b b a c d e a f a d e f g h ...
```

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.
Farthest-In-Future: Analysis

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

Pf. (by induction on number of unreduced items)
- 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

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

<table>
<thead>
<tr>
<th>$j$</th>
<th>$j+1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$S'$</td>
</tr>
<tr>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>$e$</td>
<td>$f$</td>
</tr>
<tr>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>$d$</td>
<td>$f$</td>
</tr>
</tbody>
</table>

- 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 a different action, and let $g$ be item requested at time $j'$.

<table>
<thead>
<tr>
<th>$j'$</th>
<th>$S$</th>
<th>$S'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>same</td>
<td>$e$</td>
<td>same</td>
</tr>
</tbody>
</table>

- 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 $e' = e$, $S'$ evicts $e'$ and brings $e$ into the cache; now $S$ and $S'$ have the same cache

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

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

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

Shortest Path Problem

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

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.

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.

Caching Perspective

Online vs. offline algorithms.

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'.

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

otherwise S' would take the same action

• Case 3c: g = e, f. S must evict e.
  Make S' evict f; now S and S' have the same cache.

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

4.4 Shortest Paths in a Graph

Shortest path network.

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'.

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

must involve e or f (or both)

otherwise S' would take the same action

• Case 3c: g = e, f. S must evict e.
  Make S' evict f; now S and S' have the same cache.

\[ j' \begin{array}{c|c|c} \text{some} & g & \text{same} & g \\ \hline S & S' \end{array} \]
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

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

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

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

\[ S \]

\[ d(u) \]

\[ \ell_{e} \]

\[ S \]

\[ d(v) \]

\[ \pi(v) \]

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 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 $P$. We’ll see that it’s 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 $S$.

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

- nonnegative weights
- inductive hypothesis
- defn of $\pi(y)$
- Dijkstra chose $v$ instead of $y$

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

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

\[ S \]

\[ d(u) \]

\[ \ell_{e} \]

\[ S \]

\[ d(v) \]

\[ \pi(v) \]

Dijkstra’s Algorithm: Implementation

For each unexplored node, explicitly maintain

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

- Next node to explore = node with minimum $\pi(v)$.
- When exploring $v$, for each incident edge $e = (v, w)$, update

$$\pi(w) = \min\{ \pi(w), \pi(v) + \ell_e \}.$$

Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$.

<table>
<thead>
<tr>
<th>PQ Operation</th>
<th>Dijkstra</th>
<th>Array</th>
<th>Binary heap</th>
<th>d-way Heap</th>
<th>Fib heap $†$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$d \log n$</td>
<td>1</td>
</tr>
<tr>
<td>ExtractMin</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$d \log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>ChangeKey</td>
<td>$m$</td>
<td>1</td>
<td>$\log n$</td>
<td>$d \log n$</td>
<td>1</td>
</tr>
<tr>
<td>IsEmpty</td>
<td>$n$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>$n^2$</td>
<td>$m \log n$</td>
<td>$m \log \log n$</td>
<td>$m + n \log n$</td>
<td></td>
</tr>
</tbody>
</table>

† Individual ops are amortized bounds
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.

Q. Is cashier’s algorithm optimal?

Greedy 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: Analysis of Greedy Algorithm

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

\[\text{Pf. (by induction on } x\text{)}\]

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

<table>
<thead>
<tr>
<th>( k )</th>
<th>( c_k )</th>
<th>All optimal solutions must satisfy</th>
<th>Max value of coins 1, 2, ..., ( k-1 ) in any OPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>( P \leq 4 )</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>( N \leq 1 )</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>( N + D \leq 2 )</td>
<td>( 4 \times 5 = 9 )</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>( Q \leq 3 )</td>
<td>( 20 \times 4 = 24 )</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>no limit</td>
<td>( 75 \times 24 = 99 )</td>
</tr>
</tbody>
</table>

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: \(0 = b_0 < b_1 < b_2 < \ldots < b_n = L\)

\[
\begin{align*}
S & \leftarrow \{0\} \quad \text{breakpoints selected} \\
x & \leftarrow 0 \quad \text{current location}
\end{align*}
\]

while \((x + b_0)\)

let \(p\) be largest integer such that \(b_p \leq x + C\)

if \((b_p = x)\)

return "no solution"

\[
\begin{align*}
x & \leftarrow b_p \\
S & \leftarrow S \cup \{p\}
\end{align*}
\]

return \(S\)

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

- Use binary search to select each breakpoint \(p\).

Selecting Breakpoints: Correctness

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let’s see what happens.
- Let \(0 = g_0 < g_1 < \ldots < g_p = L\) denote set of breakpoints chosen by greedy.
- Let \(0 = f_0 < f_1 < \ldots < f_q = L\) denote set of breakpoints in an optimal solution with \(f_0 = g_0, f_1 = g_1, \ldots, f_r = g_r\) for largest possible value of \(r\).
- Note: \(g_{r+1}, f_{r+1}\) by greedy choice of algorithm.

Greedy: \(g_0, g_1, g_2, g_3, \ldots, g_{r+1}\)

OPT: \(f_0, f_1, f_2, f_3, f_4, \ldots, f_r\)

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.