Announcement: Homework 1 due soon!
Due: January 25th at midnight (Blackboard)

Recap: Graphs
- Bipartite Graphs
  - Definition (2-Colorable)
  - Bipartite $\Rightarrow$ No Odd Length Cycles
  - Using BFS to check if graph is bipartite
- Directed Graphs
- Directed Acyclic Graphs
  - Topological Ordering
  - Algorithm to Compute Topological Order

4.1 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 natural order. Take each job provided it’s compatible with the ones already taken.
- [Earliest start time] Consider jobs in ascending order of $s_j$
- [Earliest finish time] Consider jobs in ascending order of $f_j$
- [Shortest interval] Consider jobs in ascending order of $f_j - s_j$
- [Fewest conflicts] For each job $j$, count the number of conflicting jobs $c_j$. Schedule in ascending order of $c_j$.
Interval Scheduling: Greedy Algorithms

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

Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal.

Proof. (by contradiction)
- Assume greedy is not optimal, and let’s see what happens.
- Let \( i_1, i_2, \ldots \), denote set of jobs selected by greedy.
- Let \( j_1, j_2, \ldots \) denote set of jobs in the optimal solution with
  \( i_1 = j_1, i_2 = j_2, \ldots \), for the largest possible value of \( r \).

Greedy:

\[ \begin{align*}
  j_1 & \quad j_2 \\
  j_3 & \quad j_4 \\
  \vdots & \quad \vdots
\end{align*} \]

CPT:

\[ \begin{align*}
  j_1 & \quad j_2 \\
  j_3 & \quad j_4 \\
  \vdots & \quad \vdots
\end{align*} \]

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.
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.

Example: This schedule uses only 3 classrooms.

<table>
<thead>
<tr>
<th>Time</th>
<th>9</th>
<th>9:30</th>
<th>10</th>
<th>10:30</th>
<th>11</th>
<th>11:30</th>
<th>12</th>
<th>12:30</th>
<th>1</th>
<th>1:30</th>
<th>2</th>
<th>2:30</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c</td>
<td>d</td>
<td>f</td>
<td>j</td>
<td>a</td>
<td>e</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Interval Partitioning: Lower Bound on Optimal Solution

Definition: 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.

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

Question: Does there always exist a schedule equal to depth of intervals?

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 $s_1 \leq s_2 \leq \ldots \leq s_n$.

$d \leftarrow 0$ (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.

4.4 Shortest Paths in a Graph

Shortest Path Problem

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 $s$-2-3-5-$t$ = $9 + 23 + 2 + 16 = 50$. 

Shortest path network:
- Directed graph $G = (V, E)$.
- Source $s$, destination $t$.
- Length $l_e$ = length of edge $e$.

Cost of path $s$-2-3-5-$t$ = $9 + 23 + 2 + 16 = 50$.
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 $d(v)$, add $v$ to $S$, and set $d(v) = \pi(v)$.

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

where $\ell(e)$ is the length of the edge from $u$ to $v$.

Dijkstra’s Algorithm: Proof of Correctness

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

Proof: (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 $S$.
  $$d(x) + \ell(x,y) \geq d(x) + \pi(y) \geq \pi(y) \geq \pi(v)$$

Dijkstra’s Algorithm: Implementation

For each unexplored node, explicitly maintain $\pi(v)$. Next node to explore = node with minimum $\pi(v)$.

When exploring $v$, for each incident edge $e=(v, w)$, update $\pi(w)$.

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

Optimal priority queues.

- Binary heap
  - Total time: $m \log n$
  - Individual ops: $O(\log n)$
- Fibonacci heap
  - Total time: $m + n \log n$
  - Individual ops: $O(1)$
- $d$-way heap
  - Total time: $d \log d n$
  - Individual ops: $O(\log d n)$

4.2 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: $L = \max_j (f_j - d_j)$.

Goal: schedule all jobs to minimize maximum lateness $L = \max_j L(j)$.

Ex:

<table>
<thead>
<tr>
<th>Job</th>
<th>$t_j$</th>
<th>$d_j$</th>
<th>$s_j$</th>
<th>$f_j$</th>
<th>Lateness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>7</td>
<td>3</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>10</td>
<td>4</td>
</tr>
</tbody>
</table>

Scheduling to Minimizing Lateness

- Heuristic: schedule jobs in order of $s_j$.
- Lateness: $L = \max_j (f_j - d_j)$.

Ex:...

Copyright 2000, Kevin Wayne
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 $d_1 \leq d_2 \leq \ldots \leq d_n$

Ex: $j = 1$ to $n$
- Assign job $j$ to interval $[t_j, t_j + b_j]$
- $t \leftarrow t + b_j$
- $b_j \leftarrow 0$

Output intervals $[s_j, f_j]$

Minimizing Lateness: No Idle Time

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

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

Pf. Let $i < j$ be the lateness before the swap, and let $i'$ be it afterwards.
- $i' = i$ for all $k \neq i, j$
- $i'_j \leq i_j$
- If job $j$ is late:

Minimizing Lateness: Inversions

Def. Given a schedule $S$, an inversion 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: Analysis of Greedy Algorithm

**Theorem.** Greedy schedule $S$ is optimal.

**Proof (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.
- **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.
- **Other greedy algorithms.** Kruskal, Prim, Dijkstra, Huffman, ...

4.3 Optimal Caching

**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.
  - **Proof.** Algorithm and theorem are intuitive; proof is subtle.

Optimal 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.

**Example (Ex):** $k = 2$, initial cache = $ab$.
- **Requests:** $a, b, c, b, c, a, a, b$.
- **Optimal eviction schedule:** 2 cache misses.

Optimal Offline Caching

- **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.

Optimal Offline Caching: Farthest-In-Future

- **Theorem.** (Bellady, 1960s) FF is optimal eviction schedule.
  - **Proof.** Algorithm and theorem are intuitive; proof is subtle.
Reduced Eviction Schedules

**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 and Case 2 diagrams]

Farthest-In-Future: Analysis

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

**Pf.** (continued)
- **Case 3:** (no item in the cache). $S_U$ evicts $e$, $S$ evicts $f = e$.

$S$'s construction of $S'$ from $S$ by evicting $e$ instead of $f$:

- New $S'$ agrees with $S_U$ 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 $j'$.

![Diagram showing the construction of $S'$]

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.

**LIFO.** Evict page brought in most recently.
- **LRU.** Evict page whose most recent access was earliest.
- LIFO is $k$-competitive. ([Section 13.8])
- LRU is arbitrary bad.

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

- Provides basis for understanding and analyzing online algorithms.
- FF is optimal. (Section 13.8)
- LIFO is arbitrary bad.
### Extra Slides

#### 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.

**Q.** Is cashier's algorithm optimal?

- Sort coins denominations by value: \(c_1 < c_2 < \ldots < c_n\).
- \(k\) is the largest integer such that \(c_k \leq x\).
- if \(k = 0\) return "no solution found".
- \(s\) is the set of coins selected.

- while \((k \neq 0)\) do
  - let \(k\) be largest integer such that \(c_k \leq x\).
  - if \((k = 0)\) return "no solution found".
  - \(s \leftarrow s \cup \{k\}\).

- return \(s\).

**Q.** Is cashier's algorithm optimal?

**Coin-Changing: Analysis of Greedy Algorithm**

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

**Pf.** (by induction on \(x\))

1. Consider optimal way to change \(c_k \leq x < c_{k+1}\) : greedy takes coin \(k\).
2. We claim that any optimal solution must also take coin \(k\).
3. - if not, it needs enough coins of type \(c_1, \ldots, c_{k-1}\) to add up to \(x\).
- Thus, 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.

<table>
<thead>
<tr>
<th>(k)</th>
<th>(c_k)</th>
<th>All optimal solutions must satisfy</th>
<th>Max value of coins (1, 2, \ldots, 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+5 \leq 9)</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>(Q \leq 3)</td>
<td>(20+4 \leq 24)</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>equivalent</td>
<td>(75+24 \leq 99)</td>
</tr>
</tbody>
</table>

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

**Counterexample.** 140¢.

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

Selecting Breakpoints: Greedy Algorithm

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

Greedy:

Princeton: C C C C C C C C C

Palo Alto:

1 2 3 4 5 6 7

Implementation. O(\log n).

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

Selecting Breakpoints: Correctness

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

1. Assume greedy is not optimal; let’s see what happens.
2. Let \( 0 = g_0 < g_1 < \ldots < g_p = L \) denote set of breakpoints chosen by greedy.
3. Let \( 0 = f_0 < f_1 < \ldots < f_q = L \) denote set of breakpoints in an optimal solution with \( f_1 \geq g_1, f_2 \geq g_2, \ldots, f_q \geq g_q \) for largest possible value of \( q \).
4. Note: \( g_{q+1} \) is by greedy choice of algorithm.

Another optimal solution has one more breakpoint in common \( \implies \) contradiction.

Selecting Breakpoints: Correctness

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

1. Assume greedy is not optimal; let’s see what happens.
2. Let \( 0 = g_0 < g_1 < \ldots < g_p = L \) denote set of breakpoints chosen by greedy.
3. Let \( 0 = f_0 < f_1 < \ldots < f_q = L \) denote set of breakpoints in an optimal solution with \( f_1 \geq g_1, f_2 \geq g_2, \ldots, f_q \geq g_q \) for largest possible value of \( q \).
4. Note: \( g_{q+1} \) by greedy choice of algorithm.

Another optimal solution has one more breakpoint in common \( \implies \) contradiction.

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.

Copyright 2000, Kevin Wayne