3.4 Testing Bipartiteness

Recap: Graphs
- Definition of a Graph
- Representations
  - Adjacency matrix
  - Adjacency list
- Connectivity, Cycles
- Trees (Connected + No Cycles)
  - Rooted Trees
  - Binary Trees
  - Balanced Trees
- Breadth First Search
  - BFS Tree
    - O(n+m) algorithm
- Finding Connected Components

Bipartite Graphs
- Definition: An undirected graph \( G = (V, E) \) is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.
- Applications:
  - Stable marriage: men = red, women = blue.
  - Scheduling: machines = red, jobs = blue.

An Obstruction to Bipartiteness
- Lemma: If a graph \( G \) is bipartite, it cannot contain an odd length cycle.
- If: Not possible to 2-color the odd cycle, let alone \( G \).
Bipartite Graphs

Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

Case (i)

Case (ii)

Pf. (i)
- Suppose no edge joins two nodes in adjacent layers.
- By previous lemma, this implies all edges join nodes on same level.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.

Pf. (ii)
- Suppose $(x, y)$ is an edge with $x, y$ in same level $L_j$.
- Let $z = lca(x, y)$ = lowest common ancestor.
- Let $L_i$ be level containing $z$.
- Consider cycle that takes edge from $x$ to $y$, then path from $y$ to $z$, then path from $z$ to $x$.
- Its length is $1 + (j-i) + (j-i)$, which is odd.

Obstruction to Bipartiteness

Corollary. A graph $G$ is bipartite iff it contain no odd length cycle.

Directed Graphs

Directed graph. $G = (V, E)$
- Edge $(u, v)$ goes from node $u$ to node $v$.

Ex. Web graph - hyperlink points from one web page to another.
- Directedness of graph is crucial.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.
Graph Search

Directed reachability. Given a node s, find all nodes reachable from s.

Directed s-t shortest path problem. Given two nodes s and t, what is the length of the shortest path between s and t?

Graph search. BFS extends naturally to directed graphs.

Web crawler. Start from web page s. Find all web pages linked from s, either directly or indirectly.

Strong Connectivity

Def. A graph is strongly connected if every pair of nodes is mutually reachable.

Lemma. Let s be any node. G is strongly connected if and only if every node is reachable from s, and s is reachable from every node.

Pf. \( \Rightarrow \) Follows from definition.

Pf. \( \Leftarrow \) Let s be any node. s is reachable from every node. Run BFS from s in \( G^{rev} \) to find path from even in \( G^{rev} \) to every node. Return true if and only if every node is reachable from s.

Pf. \( \Rightarrow \) Let s be any node. G is strongly connected. Run BFS from s in G to find path from s to every node. Return true if and only if every node is reachable from s.

Directed Acyclic Graphs

Def. A DAG is a directed graph that contains no directed cycles.

Ex. Precedence constraints: edge \((v_i, v_j)\) means \(v_i\) must precede \(v_j\).

Def. A topological order of a directed graph \(G = (V, E)\) is an ordering of its nodes as \(v_1, v_2, \ldots, v_n\) so that for every edge \((v_i, v_j)\) we have \(i < j\).

3.6 DAGs and Topological Ordering

Precedence Constraints

Precedence constraints. Edge \((v_i, v_j)\) means \(v_i\) must occur before \(v_j\).

Applications:
1. Course prerequisite graph: course \(v_i\) must be taken before \(v_j\).
2. Compilation: module \(v_i\) must be compiled before \(v_j\). Pipeline of computing jobs: output of job \(v_i\) needed to determine input of job \(v_j\).
Directed Acyclic Graphs

**Lemma.** If $G$ has a topological order, then $G$ is a DAG.

**Proof.** (by contradiction)

- Suppose that $G$ has a topological order $v_1, \ldots, v_n$ and that $G$ also has a directed cycle $C$. Let’s see what happens.
- Let $v_i$ be the lowest-indexed node in $C$, and let $v_j$ be the node just before $v_i$, thus $(v_j, v_i)$ is an edge.
- By our choice of $i$, we have $i < j$.
- On the other hand, since $(v_j, v_i)$ is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction. □

Directed Acyclic Graphs

**Lemma.** If $G$ is a DAG, then $G$ has a node with no incoming edges.

**Proof.** (by contradiction)

- Suppose that $G$ is a DAG and every node has at least one incoming edge. Let’s see what happens.
- Pick any node $v$, and begin following edges backward from $v$. Since $v$ has at least one incoming edge $(u, v)$, we can walk backward to $u$.
- Then, since $u$ has at least one incoming edge $(x, u)$, we can walk backward to $x$.
- Repeat until we visit a node, say $w$, twice.
- Let $C$ denote the sequence of nodes encountered between successive visits to $w$. $C$ is a cycle. □

Directed Acyclic Graphs

**Lemma.** If $G$ is a DAG, then $G$ has a topological ordering.

**Proof.** (by induction on $n$)

- **Base case:** true if $n = 1$.
- **Given DAG on $n > 1$ nodes,** find a node $v$ with no incoming edges.
- $G - \{v\}$ is a DAG, since deleting $v$ cannot create cycles.
- By inductive hypothesis, $G - \{v\}$ has a topological ordering.
- Place $v$ first in topological ordering, and append nodes of $G - \{v\}$ in topological order. This is valid since $v$ has no incoming edges. □

**Theorem.** Algorithm finds a topological order in $O(m + n)$ time.

**Proof.**

- Maintain the following information:
  - $\text{count}(w)$ = remaining number of incoming edges
  - $S$ = set of remaining nodes with no incoming edges
- **Initialization:** $O(m + n)$ via single scan through graph.
- **Update:** to delete $v$
  - remove $v$ from $S$
  - decrement $\text{count}(w)$ for all edges from $v$ to $w$, and add $w$ to $S$ if $\text{count}(w)$ hits 0
  - this is $O(1)$ per edge □
4.1 Interval Scheduling

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 $c_j$.

Greedy template. Consider jobs in some natural order. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.

Greedy template. Consider jobs in some natural order. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 $c_j$.

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

Interval Scheduling: Greedy Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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 Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [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: 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 \(i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r\), for the largest possible value of \(r\).

Greedy:

\[ j_1 \rightarrow j_2 \rightarrow \ldots \rightarrow j_r \]

\[ \text{OPT:} \quad j_1 \rightarrow j_2 \rightarrow \ldots \rightarrow j_r \]

Greedy: solution is feasible and optimal, but contradicts maximality of \(r\).

\[ i_{r+1} \] finishes before \( j_{r+1} \)

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.

Ex: This schedule uses only 3.

Interval Partitioning: Lower Bound on Optimal Solution

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

**Key observation.** Number of classrooms needed is \( \geq \) depth.

Ex: Depth of schedule below = 3 \( \Rightarrow \) schedule below is optimal.

Ex: Depth of schedule below = 3 \( \Rightarrow \) schedule below is optimal.

Q. 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.

**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.

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

For \( j = 1 \) to \( n \): 
- If (lecture \( j \) is compatible with some classroom \( k \)) schedule lecture \( j \) in classroom \( k \) 
  - allocate a new classroom \( d + 1 \) 
  - schedule lecture \( j \) in classroom \( d + 1 \) 

End for
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.
- These $d$ jobs each end after $s_j$.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than $s_j$.
- Key observation $\Rightarrow$ all schedules use $\geq d$ classrooms.

4.4 Shortest Paths in a Graph

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
  \[ d(v) = \min_{u \in S} d(u) + \delta(u, v) \]
  add $v$ to $S$, and set $d(v) = \pi(v)$.

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 $|S| < k$.

- Let $v$ be next node added to $S$, and let $u$ be the chosen edge.
- The shortest $s$-$u$ path plus $(u, v)$ is an $s$-$v$ path of length $\pi(v)$.
- Consider any $s$-$u$ path $P$. We’ll see that if it’s no shorter than $\pi(v)$.

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 $d(v) = \min_{u \in S} d(u) + \delta(u, v)$.
  add $v$ to $S$, and set $d(v) = \pi(v)$.

Cost of path $2-3-5-1 = 23 + 2 + 16 = 50$. 

Cost of path $= \text{sum of edge costs in path}$. 

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 $d(v) = \min_{u \in S} d(u) + \delta(u, v)$.
  add $v$ to $S$, and set $d(v) = \pi(v)$.

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 $|S| < k$.

- Let $v$ be next node added to $S$, and let $u$ be the chosen edge.
- The shortest $s$-$u$ path plus $(u, v)$ is an $s$-$v$ path of length $\pi(v)$.
- Consider any $s$-$u$ path $P$. We’ll see that if it’s no shorter than $\pi(v)$.

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 $d(v) = \min_{u \in S} d(u) + \delta(u, v)$.
  add $v$ to $S$, and set $d(v) = \pi(v)$.

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 $|S| < k$.

- Let $v$ be next node added to $S$, and let $u$ be the chosen edge.
- The shortest $s$-$u$ path plus $(u, v)$ is an $s$-$v$ path of length $\pi(v)$.
- Consider any $s$-$u$ path $P$. We’ll see that if it’s no shorter than $\pi(v)$.

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 $d(v) = \min_{u \in S} d(u) + \delta(u, v)$.
  add $v$ to $S$, and set $d(v) = \pi(v)$.
Dijkstra’s Algorithm: Implementation

For each unexplored node, explicitly maintain $\pi(v) = \min_{e=(u,v)} \{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>Binary Heap</th>
<th>Fibonacci Heap</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>PQ Operation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert</td>
<td>$\log n$</td>
<td>$1$</td>
<td>$n$</td>
</tr>
<tr>
<td>ExtractMin</td>
<td>$\log n$</td>
<td>$1$</td>
<td>$n$</td>
</tr>
<tr>
<td>ChangeKey</td>
<td>$\log n$</td>
<td>$1$</td>
<td>$n$</td>
</tr>
<tr>
<td>Total</td>
<td>$m \log n$</td>
<td>$m + n \log n$</td>
<td>$n^2$</td>
</tr>
</tbody>
</table>

Minimizing Lateness

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 $d_j - t_j$

Extra Slides

Minimizing Lateness: Greedy Algorithms

4.2 Scheduling to Minimize Lateness

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: $l_j = \max(0, f_j - d_j)$
- Goal: schedule all jobs to minimize maximum lateness $L = \max l_j$

Ex:

<table>
<thead>
<tr>
<th>$j$</th>
<th>$t_j$</th>
<th>$d_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

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 $d_j - t_j$

Counterexample
Minimizing Lateness: Greedy Algorithm

**Greedy algorithm.** Earliest deadline first.

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

\[ \text{Sort } n \text{ jobs by deadline so that } d_1 \leq d_2 \leq \ldots \leq d_n \]

Ex: \( j \) = 1 to \( n \)

Assign job \( j \) to interval \([t, t + x_j]\)

\[ t \leq t \leq t + x_j \]

Output intervals \([a_j, f_j]\)

Minimizing Lateness: No Idle Time

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

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 \( i \).

Output intervals \([a_j, f_j]\)

\[ \text{max lateness} = 1 \]

\[ \text{Sort } n \text{ jobs by deadline so that } d_1 \leq d_2 \leq \ldots \leq d_n \]

\[ \text{Assign job } j \text{ to interval } [t, t + x_j] \]

\[ t \leq t \leq t + x_j \]

\[ t, f_j \leftarrow t + x_j \]

Output intervals \([a_j, f_j]\)

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

**Cache**
- 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: $a, b, c, a, a, b$.
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: $g, a, b, c, d, e, f, a, b, c, d, e, f, g, \ldots$
Hit: $a, b, c, d, e, f$
Miss: $g$.

**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.

**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.

Farthest-In-Future: Analysis

**Theorem.** FF is optimal eviction algorithm.
**Pf.** (by induction on number of requests $j$)

- Let $S$ be reduced schedule that satisfies invariant through $j$ requests.
- We produce $S'$ that satisfies invariant after $j+1$ requests.

- Consider $(j+1)$st request $d = d_{j+1}$.
- Case 1: $d$ is already in the cache.
- Case 2: $d$ is not in the cache and $S$ and $S_f$ evict the same element.

Invariant: There exists an optimal reduced schedule $S'$ that makes the same eviction schedule as $S_f$ through the first $j+1$ requests.

$S'$ is a reduced schedule that satisfies invariant through $j$ requests.
We produce $S'$ that satisfies invariant after $j+1$ requests.
- Consider $(j+1)$st request $d = d_{j+1}$.
- Case 1: $d$ is already in the cache.
- Case 2: $d$ is not in the cache and $S$ and $S_f$ evict the same element.

$S'$ satisfies invariant.
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 the item requested at time $j'$.

- 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 the same cache.
  - if $e' = e$, $S'$ accesses $f$ and brings $e$ into the cache; now $S$ and $S'$ have the same cache.

- Case 3c: $g \neq e, f$. $S$ must evict $e$.
  - Make $S'$ evict $f$; now $S$ and $S'$ have the same cache.

Note: $S'$ is no longer reduced, but can be transformed to a reduced schedule that agrees with $S$ through step $j+1$.

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.

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.

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?

Cashier’s algorithm:

Sort coins denominations by value: \( c_1 < c_2 < \ldots < c_n \).

\[ \begin{align*}
\text{let } k & = \text{largest integer such that } c_k \leq x \\
\text{if } (k = 0) & \text{ return "no solution found"} \\
\text{let } x' & = x - c_k \\
S & = S \cup \{k\} \\
\text{return } S
\end{align*} \]

Coin-Changing: Analysis of Greedy Algorithm

Observation. Greedy algorithm is sub-optimal for US postal denominations: 1, 5, 10, 25, 70, 100, 150, 1225, 1500.

Counterexample. \( 140 \),

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

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: make as few refueling stops as possible.

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

Selecting Breakpoints: Greedy Algorithm

Road trip from Princeton to Palo Alto along fixed route.

Sort breakpoints so that: 0 = \( b_0 \), \( b_1 \), \( b_2 \), \ldots, \( b_n = L \).

\[ \begin{align*}
\text{let } k & = \text{largest integer such that } c_k \leq x + C \\
\text{if } (c_k = x) & \text{ return "no solution"} \\
\text{let } x' & = c_k \\
S & = S \cup \{k\} \\
\text{return } S
\end{align*} \]

Selecting Breakpoints: Truck Driver’s Algorithm

Truck driver's algorithm.

- Sort breakpoints so that: 0 = \( b_0 \), \( b_1 \), \( b_2 \), \ldots, \( b_n = L \).
- \( S \subseteq \{0\} \) breakpoints selected
- \( k = \ldots \) current breakpoint

\[ \begin{align*}
\text{let } p & = \text{largest integer such that } b_p \leq x + C \\
\text{if } (b_p < x) & \text{ return "no solution"} \\
\text{let } x' & = b_p \\
S & = S \cup \{p\} \\
\text{select } k
\end{align*} \]

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

- Use binary search to select each breakpoint \( p \).
Theorem. Greedy algorithm is optimal.

Proof (by contradiction):
1. Assume greedy is not optimal, and 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 $f_0 = g_0$, $f_1 = g_1$, $\ldots$, $f_r = g_r$ for largest possible value of $r$.
4. Note: $g_{r+1} > f_{r+1}$ by greedy choice of algorithm.

Another optimal solution has one more breakpoint in common with greedy solution, which is a 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.