COMP 382: Reasoning about algorithms
Fall 2014

Swarat Chaudhuri & John Greiner

Unit 7: Greedy algorithms
Interval scheduling problem:
- Job $j$ starts at $s_j$, finishes at $f_j$ (no weights)
- Two jobs compatible if they don’t overlap.
- **Goal**: find largest subset of mutually compatible jobs.
Natural possibilities:

1. **[Earliest start time]** Consider jobs in ascending order of start time $s_j$

2. **[Earliest finish time]** Consider jobs in ascending order of finish time $f_j$

3. **[Shortest interval]** Consider jobs in ascending order of interval length $f_j - s_j$

4. **[Fewest conflicts]** For each job, count the number of conflicting jobs $c_j$. Schedule in ascending order of conflicts $c_j$

Only one of these works!
Greedy strategies

- breaks earliest start time
- breaks shortest interval
- breaks fewest conflicts
Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.

1. Sort jobs by finish times so that $f_1 \leq f_2 \leq \cdots \leq f_n$.
2. $A = \emptyset$
3. for $j = 1$ to $n$ {
   4.     if (job $j$ compatible with $A$)
   5.     A = A ∪ {j}
   6. }
7. return $A$

**Complexity**: $O(n \log n)$
**Theorem:** Greedy algorithm is optimal.

**Proof:** By contradiction.

- Assume greedy is not optimal, and let’s see what happens.
- Let $i_1, i_2, \ldots, i_k$ denote set of jobs selected by greedy.
- Any optimal solution must eventually disagree with greedy; consider the one that disagrees at the latest possible point.
- Let $j_1, j_2, \ldots, j_m$ denote set of jobs in this optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible $r$. 

At step $i_{r+1}$, why not replace job $j_{r+1}$ with job $i_{r+1}$? Job $i_{r+1}$ finishes before $j_{r+1}$.
Theorem: The greedy algorithm is optimal.

Proof: By contradiction.

1. Assume greedy is not optimal, and let’s see what happens.
2. Let $i_1, i_2, \ldots, i_k$ denote set of jobs selected by greedy.
3. 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 $r$. 

Greedy: $i_1$, $i_2$, $\ldots$, $i_k$ OPT: $j_1$, $j_2$, $\ldots$, $j_m$

job $i_{r+1}$ finishes before $j_{r+1}$

Solution still feasible and optimal, but contradicts maximality of $r$. 

Exercise: Selecting breakpoints

- Road trip from Houston 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.

**Truck-driver’s algorithm:** Go as far as you can before refueling.

**Question:** Is this optimal?
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 4 classrooms to schedule 10 lectures.
Algorithm: 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.
- **Example:** This schedule uses 3 classrooms

![Diagram showing interval partitioning example]
**Definition:** The depth of a set of open intervals is the maximum number of intervals that contain any given time.

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

**Example:** Depth of intervals below = 3. This means schedule below is optimal.
Consider intervals in increasing order of start time
Assign interval to any compatible classroom.

Sort intervals by starting time so that $s_1 \leq s_2 \leq \cdots \leq s_n$

```
d = 0
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 = d + 1
    }
}
```

Complexity $O(n \log n)$
Greedy algorithm

- Consider intervals in increasing order of start time
- Assign interval to any compatible classroom.

```
1 Sort intervals by starting time so that \( s_1 \leq s_2 \leq \cdots \leq s_n \)
2 \( d = 0 \)
3 for \( j = 1 \) to \( n \) {
4   if (lecture \( j \) is compatible with some classroom \( k \))
5     schedule lecture \( j \) in classroom \( k \)
6   else {
7     allocate a new classroom \( d + 1 \)
8     schedule lecture \( j \) in classroom \( d + 1 \)
9     \( d = d + 1 \)
10   }
11 }
```

Complexity

\( O(n \log n) \)
**Observation:** Greedy algorithm never schedules two incompatible lectures in the same classroom.

**Theorem:** Greedy algorithm is optimal.

**Proof:**

- 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 $s_j + \delta$.
- Key observation implies that all schedules use $\geq d$ classrooms.
Minimizing lateness:

- 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 $L = \max l_j$.

<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>
Greedy template: Consider jobs in some order.

- **[Shortest processing time first]** Consider jobs in ascending order of processing time $t_j$
- **[Smallest slack]** Consider jobs in ascending order of slack $d_j - t_j$.
- **[Earliest deadline first]** Consider jobs in ascending order of deadline $d_j$
Counterexamples

- **[Shortest processing time first]**
  \[(t_1, d_1) = (1, 100); (t_2, d_2) = (10, 10)\]

- **[Smallest slack]**
  \[(t_1, d_1) = (1, 2); (t_2, d_2) = (10, 10)\]
Earliest deadline first

1. Sort $n$ jobs by deadline so that $d_1 \leq d_2 \leq \cdots \leq d_n$
2. $t = 0$
3. for $j = 1$ to $n$
   4. Assign job $j$ to interval $[t, t + t_j]$
   5. $s_j = t$, $f_j = t + t_j$
   6. $t = t + t_j$
7. output intervals $[s_j, f_j]$

max lateness = 1

<table>
<thead>
<tr>
<th>$d_1 = 6$</th>
<th>$d_2 = 8$</th>
<th>$d_3 = 9$</th>
<th>$d_4 = 9$</th>
<th>$d_5 = 14$</th>
<th>$d_6 = 15$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Observation: There exists an optimal schedule with no idle time (i.e., time when the processor stays unused).

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 $j$ scheduled before $i$

- Greedy schedule has no inversions.
- All schedules with no idle time and no inversions have same maximum lateness.
  - Can only differ in the way jobs with identical deadlines are ordered
- If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.
**Claim:** Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

**Proof:** Let $l$ be the lateness before the swap, and let $l'$ be the lateness afterwards.

- $l_k' = l_k$ for all $k \neq i, j$
- $l_i' \leq l_i$
- If job $j$ is late:

  $$l'_j = f'_j - d_j \quad \text{(definition of lateness)}$$
  $$= f_i - d_j \quad \text{($j$ finishes at time $f_i$)}$$
  $$\leq f_i - d_i \quad \text{(definition of inversion)}$$
  $$\leq l_i.$$
Analysis of Greedy Algorithm

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

**Proof:**
- Let $S^*$ 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

1. **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.

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

3. **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.
1. Consider the greedy solution $A$ and an optimal solution $OPT$.

2. Find a *measure* by which greedy stays ahead of $OPT$. Let $a_1, \ldots, a_k$ be the measures of the choices of the greedy algorithm, and let $o_1, \ldots, o_m$ be the measures of the choices of the optimal algorithm.

3. Show inductively that the partial solutions constructed by greedy are always as good as partial solutions constructed by optimal.
   - Consider the point where greedy starts disagreeing with all optimal solutions.
   - Show that if optimal agreed with greedy for one more step, the optimal solution wouldn’t lose quality.
Recap: Exchange argument proofs

1. Consider the greedy solution $A$ and an optimal solution $OPT$.
2. Compare $A$ and $OPT$. Assume that $A$ costs more than $OPT$ (otherwise you are done). Now identify differences between $A$ and $OPT$. For example:
   - There's an element of $OPT$ that's not in $A$ and an element of $A$ that's not in $OPT$
   - There are two consecutive elements in $OPT$ in a different order than they are in $A$ (i.e., there is an inversion).
3. Swap the elements in question in $OPT$ (replace one element by another in the first case above; swap the elements’ order in the second case). Argue that you have a solution that’s no worse than before.
4. Argue that if you continue swapping, you can eliminate all differences between $OPT$ and $A$. 
Problem: Given currency denominations 1 (P), 5 (N), 10 (D), 25 (Q), and 100, devise a method to pay amount to customer using fewest number of coins.

Example: $34 = 25 + 5 + 1 + 1 + 1 + 1$

Cashier’s algorithm: At each iteration, add coin of the largest value that doesn’t take you past the amount to be paid.
Exercise: Coin changing

**Problem:** Given currency denominations 1 (P), 5 (N), 10 (D), 25 (Q), and 100, devise a method to pay amount to customer using fewest number of coins.

- **Example:** \(34 = 25 + 5 + 1 + 1 + 1 + 1\)

**Cashier’s algorithm:** At each iteration, add coin of the largest value that doesn’t take you past the amount to be paid.

**Goal**

Prove that the algorithm is optimal
Suppose that at the point where greedy diverges from all \( OPT \), we need to make change for amount \( x \).

Let \( c_k \leq x < c_{k+1} \): greedy takes coin \( k \).

Claim: 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.

<table>
<thead>
<tr>
<th>( c_k )</th>
<th>Constraints on ( OPT )</th>
<th>Max. value of coins 1, 2, \ldots, ( k - 1 ) in ( OPT )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( P \leq 4 )</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>( N \leq 1 )</td>
<td>( 4 + 5 = 9 )</td>
</tr>
<tr>
<td>10</td>
<td>( N + D \leq 2 )</td>
<td>( 20 + 4 = 24 )</td>
</tr>
<tr>
<td>25</td>
<td>( Q \leq 3 )</td>
<td>( 75 + 24 = 99 )</td>
</tr>
<tr>
<td>100</td>
<td>no limit</td>
<td></td>
</tr>
</tbody>
</table>
Case when greedy is suboptimal

Greedy algorithm is suboptimal for US postal denominations:

- 1, 10, 21, 34, 70, 100, 350, 1225, 1500

Counterexample:

140 cents

Greedy: 100, 34, 1, 1, 1, 1, 1, 1

Optimal: 70, 70
Case when greedy is suboptimal

Greedy algorithm is suboptimal for US postal denominations:
- 1, 10, 21, 34, 70, 100, 350, 1225, 1500

**Counterexample:** 140 cents
- Greedy: 100, 34, 1, 1, 1, 1, 1, 1
- Optimal: 70, 70
Dijkstra’s shortest-path algorithm

Shortest path network

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

Goal: Find shortest directed path from $s$ to $t$
Dijkstra’s algorithm

1. Maintain a set of explored nodes $S$ for which we have determined the shortest path distance $d(u)$ from $s$ to $u$
2. Initialize $S = \{s\}$, $d(s) = 0$
3. Repeatedly choose unexplored node $v$ which minimizes

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

Add $v$ to $S$, and set $d(v) = \pi(v)$
**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 \( S \); \( u-v \) 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 \( P \).
  - Let \( x-y \) be the first edge in \( P \) that leaves \( S \), and let \( P' \) be the subpath to \( x \). \( P \) is already longer than \( \pi(v) \) as soon as it leaves \( S \).
Implementing Dijkstra’s algorithm

For each unexplored node, explicitly maintain

\[ \pi(v) = \min_{e=(u,v): u \in S} d(u) + l_e. \]

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

\[ \pi(w) = \min\{\pi(w), \pi(v) + l_e\}. \]

- Maintain a priority queue of unexplored nodes, prioritized by \( \pi(v) \).
Implementing Dijkstra’s algorithm

For each unexplored node, explicitly maintain

\[ \pi(v) = \min_{e=(u,v): u \in S} d(u) + l_e. \]

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

\[ \pi(w) = \min\{\pi(w), \pi(v) + l_e\}. \]

- Maintain a priority queue of unexplored nodes, prioritized by \( \pi(v) \).
  - \( O(mn) \) time if you’re using an array for a priority queue
    - Insertion and finding the minimum takes \( O(n) \) time
  - \( O(m \log n) \) time if you’re using a binary heap
    - \( O(\log n) \) time to reorganize heap after each update to \( \pi(w) \)
Minimum spanning tree: Given a connected graph $G = (V, E)$ with real-valued edge weights $c_e$, an MST is a subset of the edges $T \subseteq E$ such that $T$ is a spanning tree whose sum of edge weights is minimized.

Cayley’s theorem: There are $n^{n-2}$ spanning trees of $K_n$. 
Kruskal’s algorithm: Start with $T = \emptyset$. Consider edges in ascending order of cost. Insert edge $e$ in $T$ unless doing so would create a cycle.

Reverse-Delete algorithm: Start with $T = E$. Consider edges in descending order of cost. Delete edge $e$ from $T$ unless doing so would disconnect $T$.

Prim’s algorithm: Start with some root node $s$ and greedily grow a tree $T$ from $s$ outward. At each step, add the cheapest edge $e$ to $T$ that has exactly one endpoint in $T$. 
**Simplifying assumption:** All edge costs $c_e$ are distinct.

**Cut property:** Let $S$ be any subset of nodes, and let $e$ be the min cost edge with exactly one endpoint in $S$. Then the MST contains $e$.

**Cycle property:** Let $C$ be any cycle, and let $f$ be the max cost edge belonging to $C$. Then the MST does not contain $f$. 

![Diagram of cut and cycle properties](image)
Cycles and cuts

**Cycle:** Set of edges of the form $a-b$, $b-c$, $c-d$, $\ldots$, $y-z$, $z-a$.

**Cutset:** A cut is a subset of nodes $S$. The corresponding cutset $D$ is the subset of edges with exactly one endpoint in $S$. 
Claim: A cycle and a cutset intersect in an even number of edges.

Proof:

Cycle $C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1$
Cutset $D = 3-4, 3-5, 5-6, 5-7, 7-8$
Intersection = 3-4, 5-6
Simplifying assumption: All edge costs $c_e$ are distinct.

Cut property: Let $S$ be any subset of nodes, and let $e$ be the min cost edge with exactly one endpoint in $S$. Then the MST $T^*$ contains $e$. 

Proof by exchange argument: Suppose $e$ does not belong to $T^*$, and let's see what happens. Adding $e$ to $T^*$ creates a cycle $C$ in $T^*$. Edge $e$ is both in the cycle $C$ and in the cutset $D$ corresponding to $S$. Therefore, there exists another edge, say $f$, that is in both $C$ and $D$. $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree. Since $c_e < c_f$, cost($T'$) < cost($T^*$). This is a contradiction.
Simplifying assumption: All edge costs $c_e$ are distinct.

Cut property: Let $S$ be any subset of nodes, and let $e$ be the min cost edge with exactly one endpoint in $S$. Then the MST $T^*$ contains $e$.

Proof by exchange argument:
- Suppose $e$ does not belong to $T^*$, and let’s see what happens.
- Adding $e$ to $T^*$ creates a cycle $C$ in $T^*$.
- Edge $e$ is both in the cycle $C$ and in the cutset $D$ corresponding to $S$. Therefore, there exists another edge, say $f$, that is in both $C$ and $D$.
- $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
- Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$.
- This is a contradiction.
Simplifying assumption: All edge costs $c_e$ are distinct.

Cycle property: Let $C$ be any cycle in $G$, and let $f$ be the max cost edge belonging to $C$. Then the MST $T^*$ does not contain $f$. 

Proof by exchange argument: Suppose $f$ belongs to $T^*$, and let's see what happens. Deleting $f$ from $T^*$ creates a cut $S$ in $T^*$. Edge $f$ is both in the cycle $C$ and in the cutset $D$ corresponding to $S$. Therefore, there exists another edge, say $e$, that is in both $C$ and $D$. $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree. Since $c_e < c_f$, cost($T'$) < cost($T^*$). This is a contradiction.
Simplifying assumption: All edge costs $c_e$ are distinct.

Cycle property: Let $C$ be any cycle in $G$, and let $f$ be the max cost edge belonging to $C$. Then the MST $T^*$ does not contain $f$.

Proof by exchange argument:

- Suppose $f$ belongs to $T^*$, and let’s see what happens.
- Deleting $f$ from $T^*$ creates a cut $S$ in $T^*$.
- Edge $f$ is both in the cycle $C$ and in the cutset $D$ corresponding to $S$. Therefore, there exists another edge, say $e$, that is in both $C$ and $D$.
- $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
- Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$
- This is a contradiction.
Prim’s algorithm [Jarnk 1930, Dijkstra 1957, Prim 1959]:

- Initialize $S = \text{any node}$.
- Apply cut property to $S$.
- Add min cost edge in cutset corresponding to $S$ to $T$, and add one new explored node $u$ to $S$. 
Implementing Prim’s algorithm

Use a priority queue a la Dijkstra.

- Maintain set of explored nodes \( S \).
- For each unexplored node \( v \), maintain attachment cost \( a[v] \)
  - cost of cheapest edge \( v \) to a node in \( S \).
- \( O(mn) \) with an array; \( O(m \log n) \) with a binary heap.

```
Prim(G, c) {
    foreach \((v \in V)\) \( a[v] = \infty \)
    Initialize an empty priority queue \( Q \)
    foreach \((v \in V)\) insert \( v \) onto \( Q \)
    Initialize set of explored nodes \( S = \emptyset \)
    while \((Q \text{ is not empty})\) {
        \( u = \text{delete min element from } Q \)
        \( S = S \cup \{ u \} \)
        foreach \((\text{edge } e = (u, v) \text{ incident to } u)\)
            if \((v \notin S) \text{ and } (c[e] < a[v])\)
                decrease priority \( a[v] \) to \( c[e] \)
    }
}
```
Kruskal’s algorithm [Kruskal, 1956]

- Consider edges in ascending order of weight.
- Case 1: If adding $e$ to $T$ creates a cycle, discard $e$ according to cycle property.
- Case 2: Otherwise, insert $e = (u, v)$ into $T$ according to cut property where $S =$ set of nodes in $u$’s connected component.
Implementing Kruskal’s algorithm

**Implementation:** Use the union-find data structure.

- Build set $T$ of edges in the MST.
- Maintain set for each connected component.
- $O(m \log n)$ for sorting and $O(m \alpha(m, n))$ for union-find.
- $\alpha(m, n)$ is essentially a constant

```
Kruskal(G, c) {
    Sort edges weights so that $c_1 < c_2 < \cdots < c_m$
    $T = \emptyset$
    foreach $(u \in V)$ make a set containing singleton $u$
    for $i = 1$ to $m$
        $(u, v) = e_i$
        if $(u$ and $v$ are in different sets) {
            $T = T \cup \{e_i\}$
            merge the sets containing $u$ and $v$
        }
    return $T$
}
```
Suppose you are given a connected graph $G$ (edge costs are assumed to be distinct). A particular edge $e$ of $G$ is specified. Give a linear-time algorithm to decide if $e$ appears in a MST of $G$. 
Observation: Edge $e = (v, w)$ does not belong to an MST if and only if $v$ and $w$ can be joined by a path exclusively made of edges cheaper than $e$.

Algorithm: delete from $G$ edge $e$, as well as all edges that are more expensive than $e$. Now check connectivity.
Suppose you are given a connected graph $G$ with edge costs that are all distinct. Prove that $G$ has a unique minimum spanning tree.
Exercise: Near-tree

Let us say a graph $G = (V, E)$ is a near-tree if it is connected and has at most $n + 8$ edges, where $n = |V|$.

Give an algorithm with running time $O(n)$ that takes a near-tree with costs on its edges, and returns a minimum spanning tree of $G$. You may assume that all the edge costs are distinct.
Boruvka’s algorithm

1. Boruvka($G$):
   2. Initialize a forest $T$ to be a set of one-node trees
   3. while $T$ has more than one component:
      4. for each component $C$ of $T$:
         5. $S :=$ empty set of edges
         6. for each node $v$ in $C$:
            7. Find the cheapest edge from $v$ to a node outside $C$, and add it to $S$
            9. Add the cheapest edge in $S$ to $T$