Greedy algorithms

4.1 Interval Scheduling

Lecture 9

Course Instructor:
Sikder Huq
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**
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.
**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.

```
Sort jobs by finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).

jobs selected
A \leftarrow \emptyset
for j = 1 to n {
    if (job j compatible with A)
        A \leftarrow A \cup \{j\}
}
return A
```

**Implementation.** \( O(n \log n) \).
- Remember job \( j^* \) that was added last to \( A \).
- Job \( j \) is compatible with \( A \) if \( s_j \geq f_{j^*} \).
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:    OPT:
          j_1    j_2    \ldots    j_r    j_{r+1}    \ldots
```

Why not replace job $j_{r+1}$ with job $i_{r+1}$?
**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 \( r \).

![Diagram showing a comparison between Greedy and OPT schedules.](image)

- Job \( i_{r+1} \) finishes before \( j_{r+1} \), solution still feasible and optimal, but contradicts maximality of \( r \).
Q1: Selecting Breakpoints

Selecting breakpoints.
- Road trip from Houston 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. Question: Is this optimal?

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

\[
\begin{align*}
S &\leftarrow \{0\} \quad \text{breakpoints selected} \\
x &\leftarrow 0 \quad \text{current location} \\
\textbf{while} \ (x \neq b_n) \\
& \quad \text{let } p \text{ be largest integer such that } b_p \leq x + C \\
& \quad \textbf{if} \ (b_p = x) \\
& \quad \quad \text{return } "\text{no solution}"
\end{align*}
\]

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

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.

![Diagram showing greedy and optimal breakpoints.](image-url)
**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.
4.1 Interval Partitioning
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

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 only 3.
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 $\Rightarrow$ 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.

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

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

\text{for } j = 1 \text{ to } n \{ 
  \text{if } (\text{lecture } j \text{ is compatible with some classroom } k) 
    \text{schedule lecture } j \text{ in classroom } k 
  \text{else} 
    \text{allocate a new classroom } d + 1 
    \text{schedule lecture } j \text{ 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.
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 \( s_j + \varepsilon \).
- Key observation \( \Rightarrow \) all schedules use \( \geq d \) classrooms. \( \bullet \)
Q1: A variation on Interval Scheduling

You have a processor that can operate 24-7. People submit requests to run daily jobs on the processor. Each such job comes with a start time and an end time; if the job is accepted it must run continuously for the period between the start and end times, EVERY DAY. (Note that some jobs can start before midnight and end after midnight.)

Given a list of n such jobs, your goal is to accept as many jobs as possible (regardless of length), subject to the constraint that the processor can run at most one job at any given point of time. Give an algorithm to do this.

For example, here you have four jobs (6pm, 6am), (9pm, 4am), (3am, 2pm), (1pm, 7pm).

The optimal solution is to pick the second and fourth jobs.
Solution

Let $I_1, \ldots, I_n$ be the $n$ intervals. We call an $I_j$-restricted solution one that contains the interval $I_j$.

Here’s an algorithm, for fixed $j$, to compute an $I_j$-restricted solution of maximum size. Let $x$ be a point in $I_j$. First delete $I_j$ and all intervals that overlap it. The remaining intervals do not contain the point $x$, so we can cut the timeline at $x$ and produce an instance of the Interval Scheduling Problem from class. This takes $O(n)$ time assuming intervals are sorted by ending time.

Now, the algorithm for the full problem is to compute an $I_j$-restricted solution of maximum size for each $j = 1, \ldots, n$. This takes a total of $O(n^2)$ time. We now pick the largest of these solutions and claim that it is the optimal.

Why? Consider the optimal solution to the full problem. Suppose this produces a set of intervals $S$. There must be SOME $I_j$ in $S$, so the solution is an optimal $I_j$-restricted solution. But then our algorithm would find it.
Lecture 11
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.
Q2: 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.
Q2: 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.

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

coins selected
S \( \leftarrow \) \( \phi \)
while \( (x \neq 0) \) {
    let \( k \) be largest integer such that \( c_k \leq x \)
    if \( (k = 0) \)
        return "no solution found"
    x \( \leftarrow \) \( x - c_k \)
    S \( \leftarrow \) S \( \cup \) \{k\}
}
return S
```

Q1. 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)

- Consider optimal way to change $c_k \leq x < c_{k+1}$: 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-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, ..., $k-1$ 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>$4 + 5 = 9$</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>$Q \leq 3$</td>
<td>$20 + 4 = 24$</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>no limit</td>
<td>$75 + 24 = 99$</td>
</tr>
</tbody>
</table>
**Coin-Changing: Analysis of Greedy Algorithm**

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

**Counterexample.** 140¢.
- **Greedy:** 100, 34, 1, 1, 1, 1, 1, 1.
- **Optimal:** 70, 70.
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.
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: $\ell_j = \max \{ 0, f_j - d_j \}$.
- Goal: schedule all jobs to minimize maximum lateness $L = \max \ell_j$.

Ex:

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

Lateness:
- $d_3 = 9$: lateness = 0
- $d_2 = 8$: lateness = 2
- $d_4 = 9$: lateness = 6
- $d_5 = 14$: max lateness = 6
- $d_6 = 15$: max lateness = 6
Minimizing Lateness: Greedy Algorithms

**Greedy template.** Consider jobs in some order.

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

- **[Earliest deadline first]** Consider jobs in ascending order of deadline $d_j$.

- **[Smallest slack]** Consider jobs in ascending order of slack $d_j - t_j$. 
Minimizing Lateness: Greedy Algorithms

**Greedy template.** Consider jobs in some order.

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

  
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>$d_j$</td>
<td>100</td>
<td>10</td>
</tr>
</tbody>
</table>

  counterexample

- **[Smallest slack]** Consider jobs in ascending order of slack $d_j - t_j$.

  
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>$d_j$</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

  counterexample
Minimizing Lateness: Greedy Algorithm

Greedy algorithm. Earliest deadline first.

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

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

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>
Minimizing Lateness: No Idle Time

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

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: $d_i < d_j$ but $j$ scheduled before $i$.

**Observation.** Greedy schedule has no inversions.

**Observation.** All schedules with no idle time and no inversions have same maximum lateness.

**Observation.** If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.
Minimizing Lateness: Inversions

**Def.** An inversion in schedule $S$ is a pair of jobs $i$ and $j$ such that: $d_i < d_j$ but $j$ scheduled before $i$.

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

![Diagram showing an example of an inversion and its effect after a swap.](image-url)
Minimizing Lateness: Inversions

**Def.** An inversion in schedule $S$ is a pair of jobs $i$ and $j$ such that: $d_i < d_j$ but $j$ scheduled before $i$.

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

**Pf.** Let $\ell$ be the lateness before the swap, and let $\ell'$ be it afterwards.

- $\ell'_k = \ell_k$ for all $k \neq i, j$
- $\ell'_i \leq \ell_i$
- If job $j$ is late:
  
  $\ell'_j = f'_j - d_j = f_i - d_j \leq f_i - d_i \leq \ell_i$ (definition)

\[ f'_i \]

\[ f'_j \]
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 $S^*$.


Q3: Subsequences

Suppose you have a collection of possible events (e.g., possible transactions) and a sequence S of n events. A given event may occur multiple times—e.g., you could have an event “buy Google stock” multiple times in a log of transactions.

A sequence S’ is a subsequence of a sequence S if there’s a way to delete certain events from S such that the remaining sequence equals S’. For example, the reason to do this could be pattern matching.

Give an algorithm that takes two sequences of events—S’ of length m and S of length n—and decides in time $O(m+n)$ whether S’ is a subsequence of S.
**Solution**

**Greedy algorithm:** Let the $i$-th event of $S$ be $S(i)$. Find the first event in $S$ that matches $S'(1)$, then the second event in $S$ that matches $S'(2)$, and so on. The running time is $O(m+n)$.

It is easy to show that if the algorithm finds a match, then $S'$ is in fact a subsequence of $S$.

**More difficult direction:** if the algorithm does not find a match, then no match exists.

The proof of this is by contradiction. Suppose $S'$ matches the subsequence $S(l_1).S(l_2)...S(l_m)$. Suppose GREEDY produces the sequence $S(k_1).S(k_2)....$ Show that greedy can produce a match all the way up to $S(k_m)$ and also $k_i \leq l_i$ for all $i$. This is done in a way similar to the proof in interval scheduling.
4.4 Shortest Paths in a Graph
Shortest Path Problem

Shortest path network.
- Directed graph $G = (V, E)$.
- Source $s$, destination $t$.
- Length $\ell_e = \text{length of edge } e$.

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

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

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

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

![Diagram of Dijkstra's Algorithm](image)
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

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

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

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

\[
\ell(P) \geq \ell(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq \pi(y) \geq \pi(v)
\]

\[
\text{nonnegative weights} \quad \text{inductive hypothesis} \quad \text{defn of } \pi(y) \quad \text{Dijkstra chose } v \text{ instead of } y
\]
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_d n )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>ExtractMin</td>
<td>( n )</td>
<td>( n )</td>
<td>( \log n )</td>
<td>( d \log_d n )</td>
<td>( \log n )</td>
</tr>
<tr>
<td>ChangeKey</td>
<td>( m )</td>
<td>( 1 )</td>
<td>( \log n )</td>
<td>( \log_d 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_{m/n} n )</td>
<td>( m + n \log n )</td>
<td></td>
</tr>
</tbody>
</table>

† Individual ops are amortized bounds
4.5 Minimum Spanning Tree

Lecture 13
Minimum Spanning Tree

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.

$G = (V, E)$

$T$, $\sum_{e \in T} c_e = 50$

Cayley's Theorem. There are $n^{n-2}$ spanning trees of $K_n$.

can't solve by brute force
Applications

MST is fundamental problem with diverse applications.

- Network design.
  - telephone, electrical, hydraulic, TV cable, computer, road

- Approximation algorithms for NP-hard problems.
  - traveling salesperson problem, Steiner tree

- Indirect applications.
  - max bottleneck paths
  - LDPC codes for error correction
  - image registration with Renyi entropy
  - learning salient features for real-time face verification
  - reducing data storage in sequencing amino acids in a protein
  - model locality of particle interactions in turbulent fluid flows
  - autoconfig protocol for Ethernet bridging to avoid cycles in a network

- Cluster analysis.
**Greedy Algorithms**

**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 $T$.

**Remark.** All three algorithms produce an MST.
Greedy Algorithms

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 $f$. 

\begin{itemize}
  \item $e$ is in the MST
  \item $f$ is not in the MST
\end{itemize}
Cycles and Cuts

**Cycle.** Set of edges the form a-b, b-c, c-d, ..., 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 $S$. 
**Cycle-Cut Intersection**

**Claim.** A cycle and a cutset intersect in an even number of edges.

*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 $= 3-4, 5-6$*

**Pf.** (by picture)
Greedy Algorithms

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 $e$.

Pf. (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 $S$.
  $\Rightarrow$ 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. \qed
Greedy Algorithms

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 $f$.

Pf. (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 $S$.
  $\Rightarrow$ 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. $\blacksquare$
Prim's Algorithm: Proof of Correctness

Prim's algorithm. [Jarník 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 $S$. 
Implementation. Use a priority queue ala Dijkstra.
- Maintain set of explored nodes $S$.
- For each unexplored node $v$, maintain attachment cost $a[v] = \text{cost of cheapest edge } v \text{ to a node in } S$.
- $O(n^2)$ with an array; $O(m \log n)$ with a binary heap.

```c
Prim(G, c) {
    foreach (v ∈ V) a[v] ← ∞
    Initialize an empty priority queue Q
    foreach (v ∈ V) insert v onto Q
    Initialize set of explored nodes S ← ∅

    while (Q is not empty) {
        u ← delete min element from Q
        S ← S ∪ { u }
        foreach (edge e = (u, v) incident to u)
            if ((v ∉ S) and (c_e < a[v]))
                decrease priority a[v] to c_e
    }
}
Lecture 14
Kruskal's Algorithm: Proof of Correctness

**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.

---

**Case 1**

**Case 2**
Implementation: 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.

\[
m \leq n^2 \Rightarrow \log m \text{ is } O(\log n)
\]

\[
\text{essentially a constant}
\]

Kruskal($G$, $c$) {
    Sort edges weights so that $c_1 \leq c_2 \leq \ldots \leq c_m$.
    $T \leftarrow \emptyset$

    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 connected components) {
            $T \leftarrow T \cup \{e_i\}$
            merge the sets containing $u$ and $v$
        }
    return $T$
}
Lexicographic Tiebreaking

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts to break any ties.

Impact. Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs. e.g., if all edge costs are integers, perturbing cost of edge e, by $i / n^2$
Lexicographic Tiebreaking

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts to break any ties.

Impact. Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs.

Implementation. Can handle arbitrarily small perturbations implicitly by breaking ties lexicographically, according to index.

```java
boolean less(i, j) {
    if (cost(e_i) < cost(e_j)) return true
    else if (cost(e_i) > cost(e_j)) return false
    else if (i < j) return true
    else return false
}
```

↑ e.g., if all edge costs are integers, perturbing cost of edge e, by \( i / n^2 \)
Q4: Membership in MST

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 $G$. 
**MST properties**

**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 $f$.
Use the following observation: Edge \( e = (v,w) \) belongs to an MST if and only if \( v \) and \( w \) cannot be joined by a path exclusively made of edges cheaper than \( e \).

Proof: \( \rightarrow \) Cycle property.

\( \leftarrow \) Consider the set \( S \) of nodes that are reachable from \( v \) using edges cheaper than \( e \). By assumption, \( w \) is not in \( S \). Now note that \( e \) is the cheapest edge in the cutset of \( S \). Apply the cut property.

This gives us an algorithm: delete from \( G \) edge \( e \), as well as all edges that are more expensive than \( e \). Now check connectivity.
Q5: Near-tree

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 that runs in $O(n)$ time, has as input a weighted near-tree $G$, and returns an MST of $G$. All edge-costs can be assumed to be distinct.
Apply the cycle property nine times. Perform BFS until you find a cycle in the graph, then delete the heaviest edge on this cycle. We know that this edge is not in any MST. Repeat...
Lecture 15
Q6: Uniqueness of spanning tree

Suppose you are given a connected graph $G$ where edge costs are all distinct. Prove that $G$ has a UNIQUE minimum spanning tree.
By contradiction. Suppose you have two distinct MSTs $T$ and $T'$. Then there is an edge that is in $T'$ and not in $T$. Add this edge to $T$; this gives you a cycle. Consider the max weight edge in this cycle. This edge appears in either $T$ or $T'$, violating the cycle property.
Q7: Graphs with specified degrees

Given a list of natural numbers $d_1, \ldots, d_n$, show how to decide in polynomial time if there exists an undirected graph $G = (V, E)$ where the node degrees are precisely the numbers $d_1, \ldots, d_n$. 
If any of the degrees is 0, this should be an isolated node in the graph; so we can just delete that degree from the list.

Let us now sort the list so that $d_1 \geq \ldots \geq d_n > 0$. Let $d_n = k$.
Now consider the list $L = \{d_1 - 1, \ldots, d_k - 1, d_{k+1}, \ldots, d_{n-1}\}$.

Claim: the graph we want exists iff there is a graph whose degrees are the items of $L$.

Note that $L$ has one less element than the original list. So we can proceed recursively to check if $G$ satisfies the desired property.
Proof of Claim from previous slide

We prove the two directions of the “if and only if” separately.

($\Leftarrow$) If a graph $G$ whose degrees are the elements of $L$ exist, then we can add a node $v_n$ and connect it to the first $d_n$ nodes of $G$ in the descending order of degrees.

($\Rightarrow$) Suppose there’s no graph whose degrees are the elements of $L$, but at the same time, there’s a graph $G$ of the sort demanded by the problem. Let $v_i$ be the node with degree $d_i$ in $G$. By assumption, we have $i$ and $j$, with $j < i$, such that $G$ has an edge of the form $(v_n, v_i)$ but no edge of the form $(v_n, v_j)$. But because of the way the list is sorted, we have $d_j \geq d_i$. This means there is some $v$ such that $G$ has an edge $(v_j, v)$ but no edge $(v_i, v)$.

Now let us apply an exchange argument. Let us replace $G$ by a graph $G'$ which is exactly like $G$, but where: (1) there is no edge $(v_n, v_i)$ or $(v_j, v)$, and (2) there are edges $(v_i, v)$ and $(v_n, v_j)$. Applied repeatedly, this transformation gives us a graph whose degrees belong to $L$, giving a contradiction.
Q8: Fastest travel time

Suppose you have found a travel website that can predict how fast you’ll be able to travel on a road. More precisely, given an edge $e = (v, w)$ on a road network, and given a proposed starting time $t$ from location $v$, the site returns a value $f_e(t)$ that gives you the predicted arrival time at $w$. It’s guaranteed that:

1. $f_e(t) \geq t$ for all edges $e$;
2. $f_e(t)$ is a monotone function of $t$.

Otherwise, the functions may be arbitrary.

You want to use this website to determine the fastest way to travel from a start point to an intended destination. Give an algorithm to do this using a polynomial number of queries to the website.
Algorithm: $r(u)$ is used to recover the fastest path

$S = \{s\}; \ d(s) = 0$

While $S \neq V$

Select a node $v$ not in $S$ with at least one edge from $S$ for which $d'(v) = \min_{e=(u,v): u \in S} f_e(d(u))$ is as small as possible

Add $v$ to $S$ and define $d(v) = d'(v)$ and $r(v) = u$

Proof of correctness: Similar to Dijkstra’s algorithm.
Q9: Topological ordering

Give an extension of the topological sort algorithm that we studied in class that does the following. The input of the algorithm is an arbitrary directed graph $G$ that may or may not be a DAG. The output is one of two things: (a) a topological ordering, thus establishing that $G$ is a DAG; or (b) a cycle in the graph, establishing that $G$ is not a DAG. The runtime of the algorithm should be $O(m + n)$. 
Use the topological sort algorithm. If you find that every node has an incoming edge, report a cycle. How to find this cycle? Just follow the incoming edges backwards. At some point you will have to revisit a node, as there are only \( n \) nodes. At this point you have your cycle.
Q10: Placing cellphone towers

Consider a long country road with houses scattered sparsely along it. You want to place cell phone base stations at certain points along the road, so that every house is within 4 miles of the base station.

Give an efficient algorithm that achieves this goal.
Answer

**Greedy algorithm:**
1. Start at the western end; keep moving east until there’s a house exactly 4 miles to the west.
2. Place a base station at this point.
3. Delete all the houses covered by this station.

**Proof strategy:** Greedy stays ahead.

Consider greedy placement \(S = \{s_1, \ldots, s_k\}\). Let \(j\) be the index at which \(S\) disagrees with every optimal placement. Take an optimal schedule \(T = \{t_1, \ldots, t_k\}\) that agrees with \(S\) until the index \(j\). Now, \(s_j \geq t_j\), as greedy places \(s_j\) as far to the east as possible.

But then you can perturb \(T\) to get an optimal solution that agrees with \(S\) for one more step.
Q11: Dijkstra with negative edge costs

Give an example of a directed graph with negative edge costs where the algorithm produces an incorrect output.

Answer:

$G(V,E)$

$V = \{s, u, v, w, t\}$

$E = \{(s,u,3), (s,v,2), (u,w,4), (v,t,3), (w,t,-15)\}$
Q12: Number of weighted shortest paths

Given a weighted directed graph (positive edge weights), find the number of shortest paths from node u to node v.
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 $\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e$.

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

shortest path to some $u$ in explored part, followed by a single edge $(u, v)$
Answer: Modify Dijkstra’s algorithm

MODIFIED-DIJKSTRA \((G = (V; E); s \text{ in } V)\)
Set \(d[v] = \infty\) and \(count[v] = 0\) for every node \(v\) in \((V - \{s\})\)
Set \(d[s] = 0\) and \(count[s] = 1\)
\(S = \emptyset, Q = V\)
while \(Q \neq \emptyset\)
    Let \(u\) be a node in \(Q\) such that \(d[u] \leq d[v]\) for all nodes \(v\) in \(Q\)
    \(Q = Q - \{u\}; \text{ add } u \text{ to } S\)
    for each node \(v\) adjacent to \(u\)
        if \(d[v] > d[u] + w(u, v)\)
            \(d[v] = d[u] + w(u, v)\)
            \(count[v] = count[u]\)
        else if \(d[v] = d[u] + w(u, v)\)
            \(count[v] = count[v] + count[u]\)
    return \(count[v]\) for every node \(v\) in \(V\)