Announcement: Homework 3 due February 15th at 11:59PM
Midterm Exam: Wed, Feb 20 (8PM-10PM) @ EE 170

Recap: Dynamic Programming

Key Idea: Express optimal solution in terms of solutions to smaller sub problems

Example 1: Weighted Interval Scheduling
- Goal: Maximize weight of schedule with no overlapping jobs
- \( \text{OPT}(j) = \text{weight of optimal solution that only uses jobs 1...j} \)
- \( \text{OPT}(j) = \max\{ w_j + \text{OPT}(p(j)), \text{OPT}(j-1) \} \)
  - Case 1: Optimal schedule includes job \( j \) with value \( w_j \)
    - Add job \( j \) (reward \( w_j \)) and eliminate incompatible jobs \( p(j)+1,...,j \)
  - Case 2: Optimal solution does not include item \( j \)

Example 2: Segmented Least Squares (fit points to sequence of lines)
- Goal: \( \min\{ E + cL \} \) (E - squared error, L = # lines)
- \( \text{OPT}(j) = \text{best solution only considering first j points} \)
- \( \text{OPT}(j) = \min\{c + e_{ij} + \text{OPT}(i-1) \} \)
  - Case 1: Last line fits points \( p_i,...,p_j \)
    - Cost for last line: squared error \( e_{ij} \) + adds one line \( c \)
    - Still need to fit points \( p_1,...,p_{i-1} \)
  - Case 2: Last line does not fit points \( p_i,...,p_j \)

6.4 Knapsack Problem

Knapsack problem.
- Given \( n \) objects and a "knapsack."
- Item \( i \) weighs \( w_i > 0 \) kilograms and has value \( v_i > 0 \).
- Knapsack has capacity of \( W \) kilograms.
- Goal: Fill knapsack so as to maximize total value.

Ex: \( \{ 3, 4 \} \) has value 40.

Greedy: repeatedly add item with maximum ratio \( v_i / w_i \).
Ex: \( \{ 5, 2, 1 \} \) achieves only value = 35 \( \Rightarrow \) greedy not optimal.

Knapsack Problem (Greedy)

Knapsack problem (Greedy).
- Given \( n \) objects and a "knapsack."
- Item \( i \) weighs \( w_i > 0 \) kilograms and has value \( v_i > 0 \).
- Knapsack has capacity of \( W \) kilograms.
- Goal: Fill knapsack so as to maximize total value.

Ex: \( \{ 3, 4 \} \) has value 40.

Greedy: repeatedly add item with maximum ratio \( v_i / w_i \).
Ex: \( \{ 5, 2, 1 \} \) achieves only value = 35 \( \Rightarrow \) greedy not optimal.
Knapsack Problem (Greedy)

Knapsack problem.
- Given $n$ objects and a "knapsack."
- Item $i$ weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of $W$ kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: $(3, 4)$ has value 40.

Greedy: repeatedly add item with maximum ratio $v_i / w_i$.
Ex: $(5, 2, 1)$ achieves only value $= 35 \Rightarrow$ greedy not optimal.

Dynamic Programming: False Start

Def. $OPT(i) =$ max profit subset of items $1, \ldots, i$.
- Case 1: $OPT$ does not select item $i$.
  - OPT selects best of $\{1, 2, \ldots, i-1\}$
- Case 2: $OPT$ selects item $i$.
  - accepting item $i$ does not immediately imply that we will have to reject other items
  - without knowing what other items were selected before $i$, we don’t even know if we have enough room for $i$

Conclusion. Need more sub-problems!

Dynamic Programming: Adding a New Variable

Def. $OPT(i, w) =$ max profit subset of items $1, \ldots, i$ with weight limit $w$.
- Case 1: $OPT$ does not select item $i$.
  - OPT selects best of $\{1, 2, \ldots, i-1\}$ using weight limit $w$
- Case 2: $OPT$ selects item $i$.
  - new weight limit $= w - w_i$
  - OPT selects best of $\{1, 2, \ldots, i-1\}$ using this new weight limit

Knapsack Problem: Bottom-Up

Knapsack. Fill up an $n$-by-$W$ array.

\[
\begin{align*}
\text{Input: } & n, W, w_1, \ldots, w_n, v_1, \ldots, v_n \\
\text{for } & w = 0 \text{ to } W \\
M[0, w] & = 0 \\
\text{for } & i = 1 \text{ to } n \\
\text{for } & w = 1 \text{ to } W \\
\text{if } & (w_i > w) \\
M[i, w] & = M[i-1, w] \\
\text{else} & M[i, w] = \max(M[i-1, w], v_i + M[i-1, w-w_i]) \\
\text{return } & M[n, W]
\end{align*}
\]
### Knapsack Algorithm

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
<th>W + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
<td>31</td>
</tr>
</tbody>
</table>

OPT: {4, 3} value = 22 + 18 = 40

### Knapsack Problem: Running Time

Running time: \( O(n W) \)
- Not polynomial in input size
- Only need \( \log W \) bits to encode each weight
- Problem can be encoded with \( O(n \log W) \) bits

"Pseudo-polynomial"
- Decision version of Knapsack is NP-complete. [Chapter 8]

Knapsack approximation algorithm. There exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]

### 6.5 RNA Secondary Structure

#### RNA Secondary Structure

**Secondary structure.** A set of pairs \( S = (b_i, b_j) \) that satisfy:
- [Watson-Crick.] \( S \) is a matching and each pair in \( S \) is a Watson-Crick complement: A-U, U-A, C-G, or G-C.
- [No sharp turns.] The ends of each pair are separated by at least 4 intervening bases. If \( (b_i, b_j) \in S \), then \( i < j - 4 \).
- [Non-crossing.] If \( (b_i, b_j) \) and \( (b_k, b_l) \) are two pairs in \( S \), then we cannot have \( i < k < j < l \).

**Free energy.** Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.

**Goal.** Given an RNA molecule \( B = b_1 b_2 \ldots b_n \), find a secondary structure \( S \) that maximizes the number of base pairs.
RNA Secondary Structure: Subproblems

First attempt. \( OPT(j) = \) maximum number of base pairs in a secondary structure of the substring \( b_1 b_2 \ldots b_j \).

Difficulty. Results in two sub-problems.
- Finding secondary structure in: \( b_1 b_2 \ldots b_{t-1} \).
- Finding secondary structure in: \( b_{t+1} b_{t+2} \ldots b_{n-1} \).

Dynamic Programming Over Intervals

Notation. \( OPT(i, j) = \) maximum number of base pairs in a secondary structure of the substring \( b_i b_{i+1} \ldots b_j \).

- Case 1. If \( i \geq j - 4 \).
  - \( OPT(i, j) = 0 \) by no-sharp turns condition.

- Case 2. Base \( b_j \) is not involved in a pair.
  - \( OPT(i, j) = OPT(i, j-1) \).

- Case 3. Base \( b_j \) pairs with \( b_t \) for some \( i \leq t < j - 4 \).
  - non-crossing constraint decouples resulting sub-problems
  - \( OPT(i, j) = 1 + \max_{t} \{ OPT(i, t-1) \times OPT(t+1, j-1) \} \).

Remark. Same core idea in CKY algorithm to parse context-free grammars.

Bottom Up Dynamic Programming Over Intervals

Q. What order to solve the sub-problems?
A. Do shortest intervals first.

Dynamic Programming Summary

Recipe.
- Characterize structure of problem.
- Recursively define value of optimal solution.
- Compute value of optimal solution.
- Construct optimal solution from computed information.

Dynamic programming techniques.
- Binary choice: weighted interval scheduling.
- Multi-way choice: segmented least squares.
- Adding a new variable: knapsack.
- Dynamic programming over intervals: RNA secondary structure.

6.6 Sequence Alignment

String Similarity

How similar are two strings?

- **occurrence**
- **match**

- **occurrence**
- **match**

- **occurrence**
- **match**

- **occurrence**
- **match**

- **occurrence**
- **match**

6 mismatches, 1 gap

1 mismatch, 1 gap

0 mismatches, 3 gaps
Edit Distance

Cost: \( c_{(i,j)} = \delta(i,j) + \alpha_{\text{mismatch}} \)

\[ \text{Applications:} \]
- Basis for Unix diff.
- Speech recognition.
- Computational biology.

Sequence Alignment

Goal: Given two strings \( X = x_1 x_2 \ldots x_m \) and \( Y = y_1 y_2 \ldots y_n \) find alignment of minimum cost.

Def. An alignment \( M \) is a set of ordered pairs \( x_i-y_j \) such that each item occurs in at most one pair and no crossings.

Def. The pair \( x_i-y_j \) and \( x_i'-y_j' \) cross if \( i < i' \), but \( j > j' \).

\[ \text{Def.} \quad \text{OPT}(i, j) = \text{min cost of aligning strings } x_1 x_2 \ldots x_i \text{ and } y_1 y_2 \ldots y_j. \]

Case 1: \( \text{OPT} \) matches \( x_i-y_j \).
- Pay mismatch for \( x_i-y_j \) + min cost of aligning two strings \( x_1 x_2 \ldots x_{i-1} \) and \( y_1 y_2 \ldots y_{j-1} \).

Case 2a: \( \text{OPT} \) leaves \( x_i \) unmatched.
- Pay gap for \( x_i \) and min cost of aligning \( x_1 x_2 \ldots x_{i-1} \) and \( y_1 y_2 \ldots y_j \).

Case 2b: \( \text{OPT} \) leaves \( y_j \) unmatched.
- Pay gap for \( y_j \) and min cost of aligning \( x_1 x_2 \ldots x_i \) and \( y_1 y_2 \ldots y_{j-1} \).

\[ \text{OPT}(i, j) = \begin{cases} \delta(i,j) & \text{if } i = 0 \\ \delta(i,j) + \text{OPT}(i-1, j-1) & \text{if } j = 0 \\ \min(\delta(i,j-1), \delta(i-1,j), \delta(i-1,j-1)) & \text{otherwise} \end{cases} \]

Analysis: \( O(mn) \) time and space.

English words or sentences: \( m, n \leq 10 \).

Computational biology: \( m = n = 100,000 \).

10 billions ops OK, but 10GB array?

6.7 Sequence Alignment in Linear Space

Q. Can we avoid using quadratic space?

Easy. Optimal value in \( O(m + n) \) space and \( O(mn) \) time.
- Compute \( \text{OPT}(i,-) \) from \( \text{OPT}(i-1,-) \).
- No longer a simple way to recover alignment itself.

Theorem. [Hirschberg 1975] Optimal alignment in \( O(m + n) \) space and \( O(mn) \) time.
- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.
Edit distance graph.

- Let \( f(i, j) \) be shortest path from (0,0) to (i, j).
- Observation: \( f(i, j) = \text{OPT}(i, j) \).

Sequence Alignment: Linear Space

Can compute \( f(\cdot, j) \) for any \( j \) in \( O(mn) \) time and \( O(m + n) \) space.

Edit distance graph.

- Let \( g(i, j) \) be shortest path from (0,0) to (i, j).
- Observation: \( g(i, j) = \text{OPT}(i, j) \).

Sequence Alignment: Linear Space

Can compute by reversing the edge orientations and inverting the roles of (0,0) and (m,n).
Edit distance graph.
- Let $g(i, j)$ be shortest path from $(i, j)$ to $(m, n)$.
- Can compute $g(i, j)$ for any $j$ in $O(mn)$ time and $O(m + n)$ space.

**Observation 1.** The cost of the shortest path that uses $(i, j)$ is
$f(i, j) + g(i, j)$.

**Observation 2.** Let $q$ be an index that minimizes
$f(q, n/2) + g(q, n/2)$. Then, the shortest path from $(0, 0)$ to $(m, n)$ uses $(q, n/2)$.

**Theorem.** Let $T(m, n) = \max$ running time of algorithm on strings of length at most $m$ and $n$. $T(m, n) = O(mn \log n)$.

**Remark.** Analysis is not tight because two sub-problems are of size $(q, n/2)$ and $(m - q, n/2)$. In next slide, we save log $n$ factor.

**Theorem.** Let $T(m, n) = \max$ running time of algorithm on strings of length $m$ and $n$. $T(m, n) = O(mn)$.

**Pf.** (by induction on $n$)
- $O(mn)$ time to compute $f(\cdot, n/2)$ and $g(\cdot, n/2)$ and find index $q$.
- $T(q, n/2) + T(m - q, n/2)$ time for two recursive calls.
- Choose constant $c$ so that:
  - Base cases: $m \leq 2$ or $n \leq 2$.
  - Inductive hypothesis: $T(m, n) \leq Zmn$. 

$T(m, n) \leq T(q, n/2) + T(m - q, n/2) + c\text{max}$
$\leq 2c(m/2 + (m - q)/2) + \text{max}$
$= O(m + n) + \text{max}$
$= Zmn$
6.8 Shortest Paths

Shortest Paths

Shortest path problem. Given a directed graph $G = (V, E)$, with edge weights $c_{vw}$, find shortest path from node $s$ to node $t$.

**Ex.** Nodes represent agents in a financial setting and $c_{vw}$ is cost of transaction in which we buy from agent $v$ and sell immediately to $w$.

![Graph](image)

Shortest Paths: Failed Attempts

- **Dijkstra.** Can fail if negative edge costs.

![Graph](image)

- **Re-weighting.** Adding a constant to every edge weight can fail.

![Graph](image)

Shortest Paths: Dynamic Programming

- **Def.** $OPT(i, v)$ = length of shortest $v$-$t$ path $P$ using at most $i$ edges.

  - Case 1: $P$ uses at most $i-1$ edges.
    - $OPT(i, v) = OPT(i-1, v)$

  - Case 2: $P$ uses exactly $i$ edges.
    - if $(v, w)$ is first edge, then $OPT$ uses $(v, w)$, and then selects best $w$-$t$ path using at most $i-1$ edges

$$OPT(i, v) = \begin{cases} 
0 & \text{if } i = 0 \\
\min \left\{ OPT(i-1, v), \min_{(w, v) \in E} \{ OPT(i-1, w) + c_{vw} \} \right\} & \text{otherwise}
\end{cases}$$

**Remark.** By previous observation, if no negative cycles, then $OPT(n-1, v) =$ length of shortest $v$-$t$ path.

![Graph](image)

Shortest Paths: Negative Cost Cycles

- **Observation.** If some path from $s$ to $t$ contains a negative cost cycle, there does not exist a shortest $s$-$t$ path; otherwise, there exists one that is simple.

![Graph](image)

Shortest Paths: Implementation

- **Analysis.** $\Theta(mn)$ time, $\Theta(n^2)$ space.

- **Finding the shortest paths.** Maintain a "successor" for each table entry.

```java
Shortest-Path(G, t) {
    foreach node $v \in V$
        $M[0, v] \leftarrow \infty$
        $M[0, t] \leftarrow 0$
    for $i = 1$ to $n-1$
        foreach node $v \in V$
            $M[i, v] \leftarrow M[i-1, v]$
        foreach edge $(v, w) \in E$
            $M[i, v] \leftarrow \min \{ M[i, v], M[i-1, w] + c_{vw} \}$
}
```
Shortest Paths: Practical Improvements

Practical improvements.
- Maintain only one array $M[v]$ = shortest $v$-$t$ path that we have found so far.
- No need to check edges of the form $(v, w)$ unless $M[w]$ changed in previous iteration.

Theorem. Throughout the algorithm, $M[v]$ is length of some $v$-$t$ path, and after $i$ rounds of updates, the value $M[v]$ is no larger than the length of shortest $v$-$t$ path using at most $i$ edges.

Overall impact.
- Memory: $O(m + n)$.
- Running time: $O(mn)$ worst case, but substantially faster in practice.

Bellman-Ford: Efficient Implementation

```
Push-Based-Shortest-Path(G, s, t) {
    foreach node v ∈ V {
        M[v] ← ∞
        successor[v] ← ∅
    }
    M[t] = 0
    for i = 1 to n-1 {
        foreach node w ∈ V {
            if (M[w] has been updated in previous iteration)
                foreach node v such that (v, w) ∈ E {
                    if (M[v] > M[w] + cvw) {
                        M[v] ← M[w] + cvw
                        successor[v] ← w
                    }
                }
            if no M[w] value changed in iteration i, stop.
        }
    }
}
```

Distance Vector Protocol

Communication network.
- Node = router.
- Edge = direct communication link.
- Cost of edge = delay on link.

Dijkstra's algorithm. Requires global information of network.
Bellman-Ford. Uses only local knowledge of neighboring nodes.
Synchronization. We don't expect routers to run in lockstep. The order in which each `foreach` loop executes in not important. Moreover, algorithm still converges even if updates are asynchronous.

Distance vector protocol.
- Each router maintains a vector of shortest path lengths to every other node (distances) and the first hop on each path (directions).
- Algorithm: each router performs $n$ separate computations, one for each potential destination node.
- "Routing by rumor."

Example: RIP, Xerox XNS RIP, Novell's IPX RIP, Cisco's IGRP, DEC's DNA Phase IV, AppleTalk's RTMP.

Caveat. Edge costs may change during algorithm (or fail completely).

Path Vector Protocols

Link state routing.
- Each router also stores the entire path, not just the distance and first hop.
- Based on Dijkstra's algorithm.
- Avoids "counting-to-infinity" problem and related difficulties.
- Requires significantly more storage.

Example: Border Gateway Protocol (BGP), Open Shortest Path First (OSPF).
6.10 Negative Cycles in a Graph

Detecting Negative Cycles

Lemma. If \( \text{OPT}(n,v) = \text{OPT}(n-1,v) \) for all \( v \), then no negative cycles.


Lemma. If \( \text{OPT}(n,v) < \text{OPT}(n-1,v) \) for some node \( v \), then (any) shortest path from \( v \) to \( t \) contains a cycle \( W \). Moreover \( W \) has negative cost.

Proof (by contradiction)

1. Since \( \text{OPT}(n,v) < \text{OPT}(n-1,v) \), we know \( P \) has exactly \( n \) edges.
2. By pigeonhole principle, \( P \) must contain a directed cycle \( W \).
3. Deleting \( W \) yields a \( v-t \) path with < \( n \) edges \( \rightarrow W \) has negative cost.

Detecting Negative Cycles: Application

Currency conversion. Given \( n \) currencies and exchange rates between pairs of currencies, is there an arbitrage opportunity?

Remark. Fastest algorithm very valuable!

Detecting Negative Cycles: Summary

Bellman-Ford: \( O(mn) \) time, \( O(n + m) \) space.
- Run Bellman-Ford for \( n \) iterations (instead of \( n-1 \)).
- Upon termination, Bellman-Ford successor variables trace a negative cycle if one exists.
- See p. 304 for improved version and early termination rule.