COMP 382: Reasoning about algorithms
Fall 2015

Swarat Chaudhuri & John Greiner

Unit 6: Dynamic Programming
Dynamic programming

Divide-and-conquer:
- Break up a problem into two sub-problems.
- Solve each sub-problem independently.
- Combine solution to sub-problems to form solution to original problem.

Dynamic programming:
- Break up a problem into a series of *overlapping* sub-problems.
- Use caching of results from solutions
- Combine results on subproblems
Weighted interval scheduling problem:

- Job $j$ starts at $s_j$, finishes at $f_j$, and has weight or value $v_j$.
- Two jobs compatible if they don’t overlap.
- **Goal**: find maximum weight subset of mutually compatible jobs.
Consider jobs in ascending order of finish time.

Add job to subset if it is compatible with previously chosen jobs.
1. Consider jobs in ascending order of finish time.
2. Add job to subset if it is compatible with previously chosen jobs.

... It turns out this actually works in the unweighted case.

... But fails in the weighted case:
- Interval \([0, 3]\] with weight 1; \([1, 100]\] with weight 1000.
**Notation:** Label jobs by finishing time: \( f_1 \leq f_2 \leq \cdots \leq f_n \)

**Definition:** \( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \):

\( p(8) = 5, \ p(7) = 3, \ p(2) = 0. \)
Dynamic Programming: Binary Choice

**Notation:** $OPT(j) =$ value of optimal solution to the problem consisting of job requests $1, 2, \ldots, j$

**Case 1:** $OPT$ selects job $j$.
- Cannot use incompatible jobs $\{p(j) + 1, p(j) + 2, \ldots, j - 1\}$
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, p(j)$

**Case 2:** $OPT$ does not select job $j$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, j - 1$
Dynamic Programming: Binary Choice

**Notation:** $OPT(j) = $ value of optimal solution to the problem consisting of job requests $1, 2, ..., j$

**Case 1:** $OPT$ selects job $j$.
- Cannot use incompatible jobs $\{p(j) + 1, p(j) + 2, \ldots, j - 1\}$
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, p(j)$

**Case 2:** $OPT$ does not select job $j$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, j - 1$

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j - 1)\} & \text{otherwise} \end{cases}$$
Weighted Interval Scheduling: Brute Force

1. **Inputs** $n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n$

2. Sort jobs by finish times so that $f_1 \leq f_2 \leq \cdots \leq f_n$

3. **Compute** $p(1), p(2), \ldots, p(n)$

4. **Scheduling:**

   ```
   Compute-Opt(j) {
   if (j = 0)
       return 0
   else
       return max(v_j + Compute-Opt(p(j)), Compute-Opt(j - 1))
   }
   ```

Problem: Exponential number of overlapping subproblems

Swarat Chaudhuri & John Greiner

COMP 382: Reasoning about algorithms
Weighted Interval Scheduling: Brute Force

1. Inputs \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)
2. Sort jobs by finish times so that \( f_1 \leq f_2 \leq \cdots \leq f_n \)
3. Compute \( p(1), p(2), \ldots, p(n) \)
4. Scheduling:

   ```
   Compute-Opt(j) {
   if (j = 0)
       return 0
   else
       return max(v_j + Compute-Opt(p(j)), Compute-Opt(j - 1))
   }
   ```

**Problem:** Exponential number of overlapping subproblems
Weighted interval scheduling: memoization

**Memoization:** Store results of each sub-problem in a cache; lookup as needed.

1. **Inputs** $n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n$
2. Sort jobs by finish times so that $f_1 \leq f_2 \leq \cdots \leq f_n$
3. Compute $p(1), p(2), \ldots, p(n)$
4. **Scheduling:**

```plaintext
for j = 1 to n
    M[j] = empty
M[0] = 0

M-Compute-Opt(j) {
    if (M[j] is empty)
        M[j] = max(v_j + M-Compute-Opt(p(j)),
                   M-Compute-Opt(j-1))
    return M[j]
}
```
Claim: Memoized version of algorithm takes $O(n \log n)$ time.

1. Sort by finish time: $O(n \log n)$
2. Computing $p(j)$: $O(n)$ after sorting by start time.
3. $M$-Compute-Opt$(j)$: each invocation takes $O(1)$ time and either
   1. returns an existing value $M[j]$
   2. fills in one new entry $M[j]$ and makes two recursive calls
4. Ranking function $\Phi = \text{number of empty entries of } M$
   - initially $\Phi = n$, throughout $\Phi \leq n$
   - If new work needs to be done, $\Phi$ decreases by 1. This means at most $2n$ recursive calls.
5. Overall running time of $M$-Compute-Opt$(n)$ is $O(n)$. 
Q: Dynamic programming algorithms computes optimal value. What if we want the solution itself?

A: Do some post-processing.

```plaintext
1 Run M-Compute-Opt(n)
2 Run Find-Solution(n)

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (v_j + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```
Build up solutions bottom-up by unwinding recursion.

Iterative-Compute-Opt {
    M[0] = 0
    for j = 1 to n
        M[j] = max(v_j + M[p(j)], M[j-1])
}
Exercise: Billboard placement

You are trying to decide where to place billboards on a highway that goes East-West for $M$ miles. The possible sites for billboards are given by numbers $x_1, \ldots, x_n$, each in the interval $[0, M]$. If you place a billboard at location $x_i$, you get a revenue $r_i$.

You have to follow a regulation: no two of the billboards can be within less than or equal to 5 miles of each other.

You want to place billboards at a subset of the sites so that you maximize your revenue modulo this restriction.

How?
Let us show how to compute the optimal subset when we are restricted to the sites $x_1$ through $x_j$.

For site $x_j$, let $e(j)$ denote the easternmost site $x_i$ (for $i < j$) that is more than 5 miles from $x_j$.

$$OPT(j) = \max(r_j + OPT(e(j)), OPT(j - 1))$$
Exercise: Longest common subsequence

You are given two strings $X = x_1x_2\ldots x_n$ and $Y = y_1y_2\ldots y_n$. Find the longest common subsequence of these two strings.

Note: Subsequence and substring are not the same. The symbols in a subsequence need not be contiguous symbols in the original string; however, they have to appear in the same order.
Recursive solution

\[
OPT(i, j) = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
OPT(i - 1, j - 1) + 1 & \text{if } x_i = y_j \\
\max(OPT(i, j - 1), OPT(j, i - 1)) & \text{if } x_i \neq y_j
\end{cases}
\]
Least squares:

- Foundational problem in statistics and numerical analysis.
- Given $n$ points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$
- ... find a line $y = ax + b$ that minimizes the sum of the squared error:

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$
Least squares:

- Foundational problem in statistics and numerical analysis.
- Given $n$ points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$
- ... find a line $y = ax + b$ that minimizes the sum of the squared error:

$$\text{SSE} = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

Solution is achieved when:

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}$$

$$b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$
Points lie roughly on a sequence of several line segments.

Given \( n \) points in the plane \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\) with \( x_1 < x_2 < \cdots < x_n \), find a sequence of lines that minimizes \( \phi \).

*Tradeoff in choosing \( \phi \): goodness of fit vs. parsimony.*
Points lie roughly on a sequence of several line segments.

Given $n$ points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \cdots < x_n$, find a sequence of lines that minimizes:

- the sum $E$ of the sum-of-squared errors in each segment
- the number of lines $L$

**Tradeoff function:** $E + cL$, for some constant $c > 0$
Defining optimal answers:

- \( OPT(j) = \) minimum cost for points \( p_1, p_2, \ldots, p_j \)
- \( e(i,j) = \) minimum sum of squares for points \( p_i, p_{i+1}, \ldots, p_j \)

To compute \( OPT(j) \):

- Last segment uses points \( p_i, p_{i+1}, \ldots, p_j \) for some \( i \)
- Cost = \( e(i,j) + c + OPT(i-1) \)
Dynamic Programming

Defining optimal answers:
- $OPT(j) =$ minimum cost for points $p_1, p_2, \ldots, p_j$
- $e(i, j) =$ minimum sum of squares for points $p_i, p_i+1, \ldots, p_j$

To compute $OPT(j)$:
- Last segment uses points $p_i, p_i+1, \ldots, p_j$ for some $i$
- Cost $= e(i, j) + c + OPT(i - 1)$

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \leq i \leq j} \{ e(i, j) + c + OPT(i - 1) \} & \text{otherwise} \end{cases}$$
Defining optimal answers:

- $OPT(j) = \text{minimum cost for points } p_1, p_2, \ldots, p_j$
- $e(i, j) = \text{minimum sum of squares for points } p_i, p_{i+1}, \ldots, p_j$

To compute $OPT(j)$:

- Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some $i$
- Cost $= e(i, j) + c + OPT(i - 1)$

$$OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\min_{1 \leq i \leq j} \{e(i, j) + c + OPT(i - 1)\} & \text{otherwise}
\end{cases}$$

Complexity

$O(n^3)$: must compute $e(i, j)$ for $O(n^2)$ pairs, $O(n)$ time per pair
Bottom-up solution

Segmented-Least-Squares(n,p_1,...,p_N,c) {
    M[0] = 0
    for j = 1 to n
        for i = 1 to j
            compute the least square error e_{ij} of segment p_i,...,p_j
        for j = 1 to n
            M[j] = min_{1 \leq i \leq j} (e_{ij} + c + M[i-1])
    return M[n]
}
Exercise: Longest palindromic subsequence

Give an algorithm to find the longest subsequence of a given string A that is a palindrome.

amantwocamelsacrazyplanacanalpanama
Algorithm: RNA secondary structure

**RNA:** String $B = b_1 b_2 \ldots b_n$ over alphabet $\{A, C, G, U\}$.

**Secondary structure:** RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of the molecule.

Ex: GUCGAUUGAGCGAUGUAACACGUGGCUACGCGAGA

complementary base pairs: A-U, C-G
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.

- Free energy approximated by number of base pairs
**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.

- Free energy approximated by number of base pairs

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

**Case 1:** $i \geq j - 4$
- $OPT(i, j) = 0$ 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:

$$OPT(i, j) = 1 + \max_t \{ OPT(i, t - 1) + OPT(t + 1, j - 1) \}$$

where $t$ satisfies $i \leq t < j - 4$, and $b_t$ and $b_j$ are Watson-Crick complements.
Detour: formally proving a dynamic programming algorithm

**Induction:**
- Show that the solution is optimal in the base case
- Show optimality assuming that recursive calls return optimal solutions
Knapsack problem:

- Given $n$ objects and a “knapsack”
- Item $i$ weighs $w_i \geq 0$ and has value $v_i$
- Knapsack can carry a maximum weight of $W$.
- **Goal**: Fill knapsack so as to maximize total value.
Knapsack

\[ KS(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
KS(i - 1, w) & \text{if } w_i > w \\
\max\{KS(i - 1, w), v_i + KS(i - 1, w - w_i)\} & \text{otherwise}
\end{cases} \]
Proving our algorithm for Knapsack

- **Definition:** $OPT(i, w) = \text{set of max. profit subsets of items } 1, \ldots, i \text{ with weight limit } w$
- **Definition:** $KS(i, w) = \text{set returned by dynamic programming algorithm}$
- **Proof goal:** $KS(i, w) \in OPT(i, w)$
Proving our algorithm for Knapsack

**Definition:** $OPT(i, w) = \text{set of max. profit subsets of items } 1, \ldots, i \text{ with weight limit } w$

**Definition:** $KS(i, w) = \text{set returned by dynamic programming algorithm}$

**Proof goal:** $KS(i, w) \in OPT(i, w)$

**Base case:**
- No object to select; therefore, $OPT(i, w) = \{\emptyset\}$

**Inductive case:**

- **[Case $w_i > w:$]** $w_i$ cannot be part of any feasible solution
  - $OPT(i, w) = OPT(i - 1, w)$
  - By inductive hypothesis, $KS(i - 1, w) \in OPT(i, w)$

- **[Case $w_i \leq w:$]**
  - $OPT(i, w) = \max\{OPT(i - 1, w), v_i + OPT(i - 1, w - w_i)\}$
  - Claim follows from inductive hypothesis
For each $i, w$, a call $KS(i, w)$ is invoked only a constant number of times

... therefore, complexity bounded by $\Theta(nW)$

Not polynomial-time if $W$ is coded in binary!
The problem:

- Given a directed graph $G = (V, E)$, with edge weights $c_{ij}$ (positive or negative), find the shortest path from node $s$ to node $t$. 
Shortest paths: Negative cost cycles

- 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 (i.e., cycle-free).
Shortest paths by dynamic programming

\( OPT(i, v) = \) length of shortest \( v\text{-}t \) 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 \( w\text{-}t \) path using at most \((i - 1)\) edges.

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

Assuming there's no negative cycle, \( OPT(n - 1, v) = \) length of the shortest \( v\text{-}t \) path.
Shortest paths by dynamic programming

\( OPT(i, v) = \text{length of shortest } v-t \text{ path } P \text{ using at most } i \text{ edges.} \)

- **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.

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

Assuming there’s no negative cycle, \( OPT(n - 1, v) = \text{length of the shortest } v-t \text{ path.} \)
Shortest paths

```plaintext
Shortest-Path(G, t) {
    foreach node v ∈ V
        M[0, v] = ∞
    M[0, t] = 0

    for i = 1 to n - 1
        foreach node v ∈ V
            M[i, v] = M[i - 1, v]
        foreach edge (v, w) ∈ E
            M[i, v] = min { M[i, v], M[i-1, w] + c(v,w) }
}

O(mn) time, O(n²) space
```
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.

**Correctness**
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 $\leq i$ edges.

**Complexity**
- **Memory:** $O(m + n)$
- **Time:** $O(mn)$ worst case, but substantially faster in practice
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] + c_{vw}) {
                        M[v] = M[w] + c_{vw}
                        successor[v] = w
                    }
                }
            }
        }
        If no M[w] value changed in iteration i, stop.
    }
}
Theorem: If \( OPT(n, v) = OPT(n - 1, v) \) for all \( v \), then no negative cycles.

Theorem: If \( OPT(n, v) < OPT(n - 1, v) \) for some node \( v \), then (any) shortest path from \( v \) to \( t \) contains a cycle \( W \) of negative cost.

 Leads to an algorithm for detecting negative cycles
Exercise: Arbitrage

**Arbitrage:** Using discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. Example:

- $1 buys £0.7, £1 buys 9.5 Argentine Pesos, and 1 Argentine Peso buys $0.16
- Then a trader can start with $1 and buy $1.064

Suppose that we are given $n$ currencies $c_1, \ldots, c_n$ and an $n \times n$ table $R$ of exchange rates, such that one unit of currency $c_i$ buys $R[i,j]$ units of currency $c_j$. Give an efficient algorithm to determine whether or not there exists a sequence of currencies $(c_{i_1}, \ldots, c_{i_k})$ such that

$$R[i_1, i_2] \times R[i_2, i_3] \times \cdots \times R[i_{k-1}, i_k] \times R[i_k, i_1] > 1.$$
Exercise: Number of shortest paths

Suppose we have a directed graph with costs on the edges. The costs may be positive or negative, but every cycle in the graph has a strictly positive cost. We are also given two nodes $s, w$.

Give an efficient algorithm that computes the number of shortest $s-w$ paths in $G$. 