Recap: Dynamic Programming

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

Example 1: Knapsack Problem
- Two Dimensional Solution \( OPT(j, w) \)
- \( OPT(j, w) = \max\{ v_j + OPT(w-w_j), OPT(j-1, w) \} \)
  - Case 1: Optimal solution includes item \( j \) with value \( v_j \)
    - Add item \( j \) and reduce remaining capacity to \( w-w_j \)
  - Case 2: Optimal solution does not include item \( j \)

Example 2: RNA Secondary Structure
- Goal: Maximize number of matched base pairs
- Constraints: No Sharp Turns, Watson-Crick Complements, No Crossing Edges
- \( OPT(i, j) = \) maximum number of base pairs in a secondary structure of the substring \( b_i b_{i+1} \ldots b_j \)
- \( OPT(i, j) = \max (OPT(i, j-1), \max \{ 1 + OPT(i, t-1) + OPT(t+1, j-1) \}) \)

6.6 Sequence Alignment

String Similarity

How similar are two strings?
- \( \text{COVARIANCE} \)
- \( \text{COCCURENCE} \)

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 \( j > j' \).

Cost: \( \sum_{i=1}^{m} c_{x_i x_i} + \sum_{j=1}^{n} c_{y_j y_j} + \sum_{i=1}^{m} \sum_{j=1}^{n} c_{x_i y_j} \) if \( i \neq i' \) and \( j \neq j' \)

Example: \( \text{STANDO VS. TADDLE} \)
Sol: \( M = x_2 y_4, x_4 y_2, x_5 y_5, x_6 y_6 \)
Sequence Alignment: Problem Structure

Def. $OPT(i, j) =$ min cost of aligning strings $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$.

- Case 1: $OPT$ 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: $OPT$ 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: $OPT$ leaves $y_j$ unmatched.
  - pay gap for $y_j$ and min cost of aligning $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_{j-1}$.

$$OPT(i, j) = \begin{cases} \delta_{i, j} & \text{if } i = 0 \\ \min \left( \alpha_{x_i, y_j} + OPT(i-1, j-1), \\ \delta + OPT(i-1, j), \\ \delta + OPT(i, j-1) \right) & \text{otherwise} \end{cases}$$

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 $OPT(i, j)$ from $OPT(i-1, j)$.
- 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.

Sequence Alignment: Linear Space

- Let $f(i, j)$ be shortest path from $(0,0)$ to $(i, j)$.
- Observation: $f(i, j) = OPT(i, j)$.

Edit distance graph.

- Let $f(i, j)$ be shortest path from $(0,0)$ to $(i, j)$.
- Observation: $f(i, j) = OPT(i, j)$.

Sequence Alignment: Linear Space

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

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

Sequence Alignment: Linear Space
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 \((q, n/2)\).

Sequence Alignment: Linear Space

Divide: find index \( q \) that minimizes \( f(q, n/2) + g(q, n/2) \) using DP.
- Align \( x_q \) and \( y_{n/2} \).
Conquer: recursively compute optimal alignment in each piece.

Sequence Alignment: Running Time Analysis Warmup

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

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.

Sequence Alignment: Running Time Analysis

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

Pf. (by induction on \( n \))
- \( O(m\log n) \) 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: \( m = 2 \) or \( n = 2 \).
  - Inductive hypothesis: \( T(m, n) \leq 2mn \).

6.8 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 \( t \).

\( \text{Ex. Nodes represent agents in a financial setting and } c_{vw} \text{ is cost of transaction in which we buy from agent } v \text{ and sell immediately to } w. \text{ } \)
Shortest Paths: Failed Attempts

Dijkstra: Can fail if negative edge costs.

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

Shortest Paths: Dynamic Programming

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

- Case 1: \( P \) uses at most \( i-1 \) edges.
  - \( \text{OPT}(i, v) = \text{OPT}(i-1, v) \)

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

Remark: By previous observation, if no negative cycles, then \( \text{OPT}(n-1, v) \) = length of shortest \( v \)-\( t \) path.

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 \( i \) edges.

Overall impact:
- Memory: \( O(n \times m) \)
- 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.
    }
}
```

Analysis: \( O(mn) \) time, \( O(n^2) \) space.

Finding the shortest paths: Maintain a "successor" for each table entry.
6.9 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 node loop executes is not important. Moreover, algorithm still converges even if updates are asynchronous.

- **Ex.** 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).


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.
  - **Pf.** Bellman-Ford algorithm.

  - **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.
  - **Pf.** (by contradiction)
    - Since \( \text{OPT}(n,v) = \text{OPT}(n-1,v) \), we know \( P \) has exactly \( n \) edges.
    - By pigeonhole principle, \( P \) must contain a directed cycle \( W \).
    - Deleting \( W \) yields a \( v \rightarrow t \) path with \( n-1 \) edges \( \Rightarrow W \) has negative cost.
Detecting Negative Cycles

Theorem. Can detect negative cost cycle in $O(mn)$ time.
- Add new node $t$ and connect all nodes to $t$ with 0-cost edge.
- Check if $OPT(n, v) = OPT(n-1, v)$ for all nodes $v$.
  - if yes, then no negative cycles
  - if no, then extract cycle from shortest path from $v$ to $t$

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, $O(m + n)$ 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.