6. Dynamic Programming II

6.1 Sequence alignment

Editing distance in cost of strings differences:

- Cost = sum of gap and mismatches penalties.

C	T	–	G	A	C	C	T	A	C	G
C	T	G	G	A	C	G	A	A	C	G

cost = $\delta + \alpha_{CG} + \alpha_{TA}$

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i : x_i \text{ unmatched}} \delta + \sum_{j : y_j \text{ unmatched}} \delta}_{\text{gap}}$$

Def. An alignment M is a set of ordered pairs xi – yj such that each item occurs in at most one pair and no crossings.

Def. OPT(i, j) = min cost of aligning

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

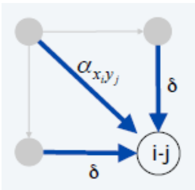
SEQUENCE-ALIGNMENT $(m, n, x_1, \ldots, x_m, y_1, \ldots, y_n, \delta, \alpha)$

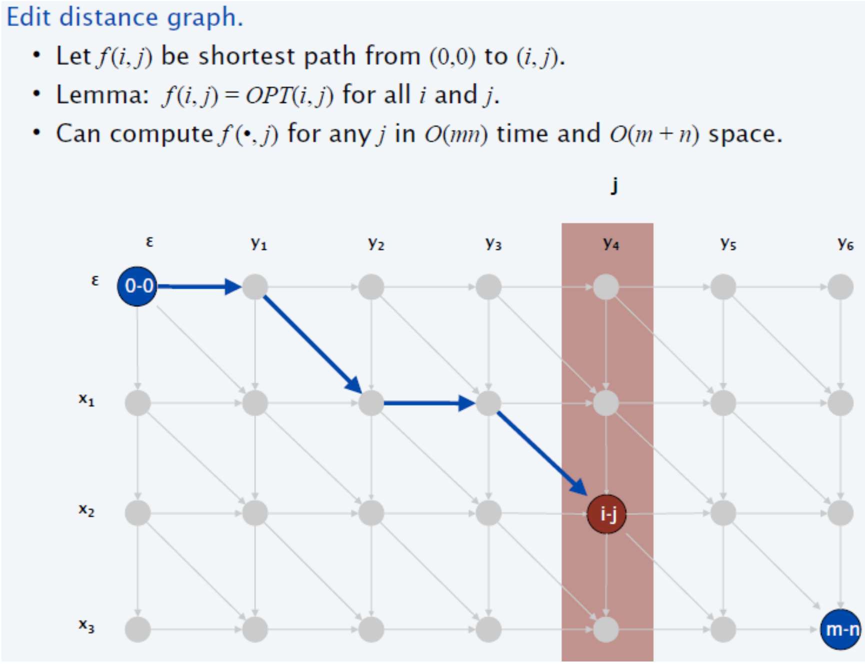
```
FOR i = 0 TO m
   M[i, 0] ← i δ.
FOR j = 0 TO n
   M[0, j] ← j δ.

FOR i = 1 TO m
   FOR j = 1 TO n
      M[i, j] ← min { α[x_i, y_j] + M[i – 1, j – 1],
                      δ + M[i – 1, j],
                      δ + M[i, j – 1]).

RETURN M[m, n].
```

Theorem. The dynamic programming algorithm computes the edit distance (and optimal alignment) of two strings of length m and n in Θ(mn) time and Θ(mn) space.

6.2 Hirschberg's algorithm

Theorem. There exist an algorithm to find an optimal alignment in O(mn) time and O(m + n) space. Combination of divide-and-conquer and dynamic programming.   f (i,j) = OPT(i,j) for all i and j.





Let g (i, j) be shortest path from (i, j) to (m, n). Can compute g(•, j) for any j in O(mn) time and O(m + n) space.

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

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

Align xq and yn / 2 → **Conquer.** At every recursion we reduce the possible space of choices.

Theorem. Running time analysis warmup → T(m, n) = O(mn log n).

Theorem. Running time analysis → T(m,n) = O(mn).

6.3 Bellman-ford

Shortest path problem.

Failed shortest path algotihms

Dijkstra: fail with negative weights.

Reweighting: adding a constant (to avoid negative) can fail.

Def. A negative cycle is a directed cycle with negative weight sum.

Lemma 1. If some path from v to t contains a negative cycle, then there does not exist a cheapest path from v to t.

Lemma 2. If G has no negative cycles, then there exists a cheapest path from v to t that is simple (and has ≤ n – 1 edges).

Dynamic programming:

Def. OPT(i, v) = cost of shortest v⤳t path that uses ≤ i edges.

- Case 1: Cheapest v⤳t path uses ≤ i – 1 edges. OPT(i, v) = OPT(i – 1, v).
- Case 2: Cheapest v⤳t path uses exactly i edges. if (v, w) is first edge, then OPT uses (v, w), and then selects best w⤳t path using ≤ i – 1 edges.

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

Observation. If no negative cycles, OPT(n – 1, v) = cost of cheapest v⤳t path.

```
SHORTEST-PATHS (V, E, c, 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_vw }.
-----------------------------------------
```

The dynamic programming algorithm computes the cost of the cheapest v⤳t path for each node v in Θ(mn) time and Θ(n^2) space.

To solve maintain successor and compute costs for edges such that: M[i,v]=M[i-1,w] + c vw

Improvements: space optimization: maintain cheapest path found so far and the successor.

Performance optimization: if d(w) wasn't updated the last time, stop.

```
BELLMAN-FORD (V, E, c, t)
─────────────────────────────
FOREACH node v ∈ V
    d(v) ← ∞.
    successor(v) ← null.
d(t) ← 0.
FOR i = 1 TO n – 1
    FOREACH node w ∈ V
        IF (d(w) was updated in previous iteration)
            FOREACH edge (v, w) ∈ E
                IF ( d(v) > d(w) + c_vw)
                    d(v) ← d(w) + c_vw.
                    successor(v) ← w.
    IF no d(w) value changed in iteration i, STOP.
─────────────────────────────
```

1 pass

Lemma 3. Throughout Bellman-Ford algorithm, d(v) is the cost of some v⤳t path; after the i^th pass, d(v) is no larger than the cost of the cheapest v⤳t path using ≤ i edges.

Theorem 2. Given a digraph with no negative cycles, Bellman-Ford computes the costs of the cheapest v⤳t paths in O(mn) time and Θ(n) extra space.

Lemma 4. If the successor graph contains a directed cycle W,
then W is a negative cycle.

Theorem 3. Given a digraph with no negative cycles, Bellman-Ford finds the cheapest s⤳t paths in O(mn) time and Θ(n) extra space.

6.4 Distance vector protocols

Dijkstra's algorithm. Requires global information of network.

Bellman-Ford. Uses only local knowledge of neighboring nodes.

DVP:

- **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.

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

6.5 Negative cycles in a diagraph

Negative cycle detection problem. Given a digraph $G = (V, E)$, with edge weights c vw, find a negative cycle (if one exists).

Lemma 5. If $OPT(n, v) = OPT(n - 1, v)$ for all v, then no negative cycle can reach t.

Lemma 6. If $OPT(n, v) < OPT(n - 1, v)$ for some node v, then (any) cheapest path from v to t contains a cycle W. Moreover W is a negative cycle.

Theorem 4. Can find a negative cycle in $\Theta(mn)$ time and $\Theta(n2)$ space.

Theorem 5. Can find a negative cycle in $O(mn)$ time and $O(n)$ extra space.

Remark. See p. 304 for improved version and early termination rule.