Lecture 7: Dynamic Programming II (Adv)

THE UNIVERSITY OF SYDNEY

Dynamic Programming Summary

- **1D dynamic programming**
  - Weighted interval scheduling
  - Segmented Least Squares
  - Maximum-sum contiguous subarray
  - Longest increasing subsequence

- **2D dynamic programming**
  - Knapsack
  - Sequence alignment

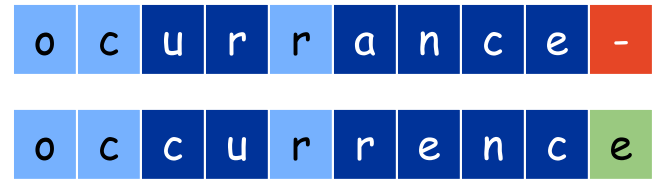
- **Dynamic programming over intervals**
  - RNA Secondary Structure

- **Dynamic programming over subsets**
  - TSP
  - k-path
  - Playlist

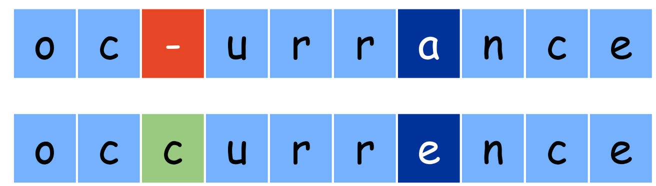
6.6 Sequence Alignment

String Similarity

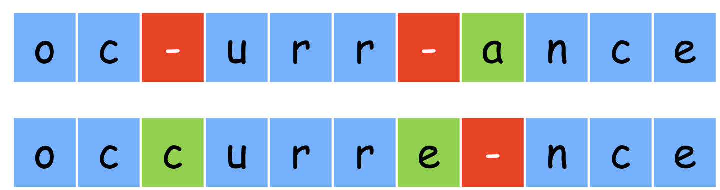
– How similar are two strings?
  - **ocurrance**
  - **occurrence**



5 mismatches, 1 gap



1 mismatch, 1 gap



0 mismatches, 3 gaps

Edit Distance

- **Applications.**
  - Basis for Unix diff.
  - Speech recognition.
  - Computational biology.

- **Edit distance.** [Levenshtein 1966, Needleman-Wunsch 1970]
  - Gap penalty δ and mismatch penalty α_{pq}.
  - Cost = sum of gap and mismatch penalties.

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

$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$

-	C	T	G	A	C	C	T	A	C	C	T
C	C	T	G	A	C	-	T	A	C	A	T

$2\delta + \alpha_{CA}$

Sequence Alignment

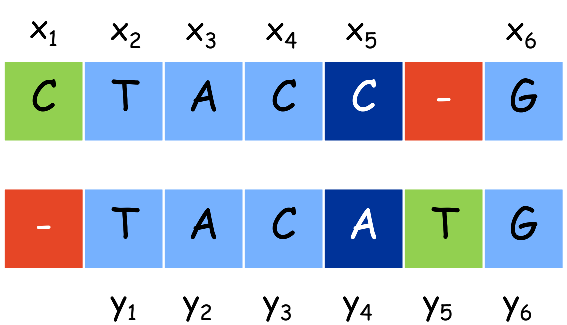
– **Goal:** Given two strings $X = x_1\ x_2 \ldots x_m$ and $Y = y_1\ y_2 \ldots y_n$ find alignment of minimum cost.

– **Definition:** 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.

– **Definition:** The pair x_i-y_j and $x_{i'}$-$y_{j'}$ cross if $i < i'$, but $j > j'$.

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

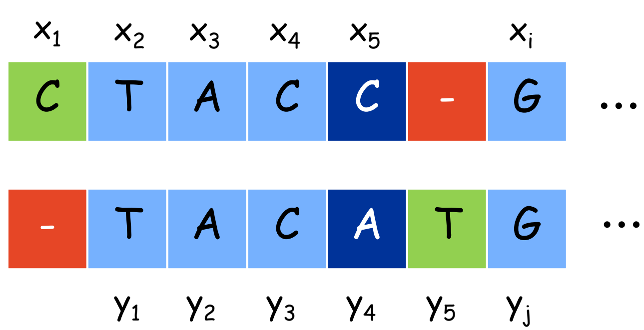
Example: CTACCG vs. TACATG.
Solution: $M = x_2$-y_1, x_3-y_2, x_4-y_3, x_5-y_4, x_6-y_6.



Sequence Alignment: Problem Structure

Step 1: **Define subproblems**

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



Sequence Alignment: Problem Structure

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

Step 2: Find recurrences

- **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 $y_1\ y_2 \ldots y_{j-1}$

Sequence Alignment: Problem Structure

- **Definition:** 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 $y_1\ y_2 \ldots y_{j-1}$

$$\boxed{\mathrm{OPI}(i, j) = \min \begin{cases} \alpha_{x_i y_j} + \mathrm{OPT}(i-1, j-1) \\ \delta + \mathrm{OPT}(i-1, j) \\ \delta + \mathrm{OPT}(i, j-1) \end{cases}}$$

Sequence Alignment: Problem Structure

Step 3: Solve the base cases

$$OPT(0,j) = j \cdot \delta \quad \text{and} \quad OPT(i,0) = i \cdot \delta$$

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

Sequence Alignment: Algorithm

```
Sequence-Alignment(m, n, x₁x₂...xₘ, y₁y₂...yₙ, δ, α) {
   for i = 0 to m
      M[0, i] = iδ
   for j = 0 to n
      M[j, 0] = jδ

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min(α[xᵢ, yⱼ] + M[i-1, j-1],
                       δ + M[i-1, j],
                       δ + M[i, j-1])
   return M[m, n]
}
```

- **Analysis.** $\Theta(mn)$ time and space.
- **English words or sentences:** m, n $\leq$ 10.
- **Computational biology:** m = n = 100,000. 10 billions ops OK, but 10GB array?

Sequence Alignment: Algorithm

```
Sequence-Alignment(m, n, x₁x₂...xₘ, y₁y₂...yₙ, δ, α) {
   for i = 0 to m
      M[0, i] = iδ
   for j = 0 to n
      M[j, 0] = jδ

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min(α[xᵢ, yⱼ] + M[i-1, j-1],
                       δ + M[i-1, j],
                       δ + M[i, j-1])
   return M[m, n]
}
```

– To get the alignment itself we can trace back through the array M.

Sequence Alignment: Linear Space

Question: Can we avoid using quadratic space?

Sequence Alignment: Linear Space

Question: Can we avoid using quadratic space?

Easy. Optimal value in $O(m + n)$ space and $O(mn)$ time.

- Compute OPT(i, •) from OPT(i-1, •).
- BUT! No longer a simple way to recover alignment itself.

Sequence Alignment: Linear Space

Question: Can we avoid using quadratic space?

Easy. Optimal value in $O(m + n)$ space and $O(mn)$ time.

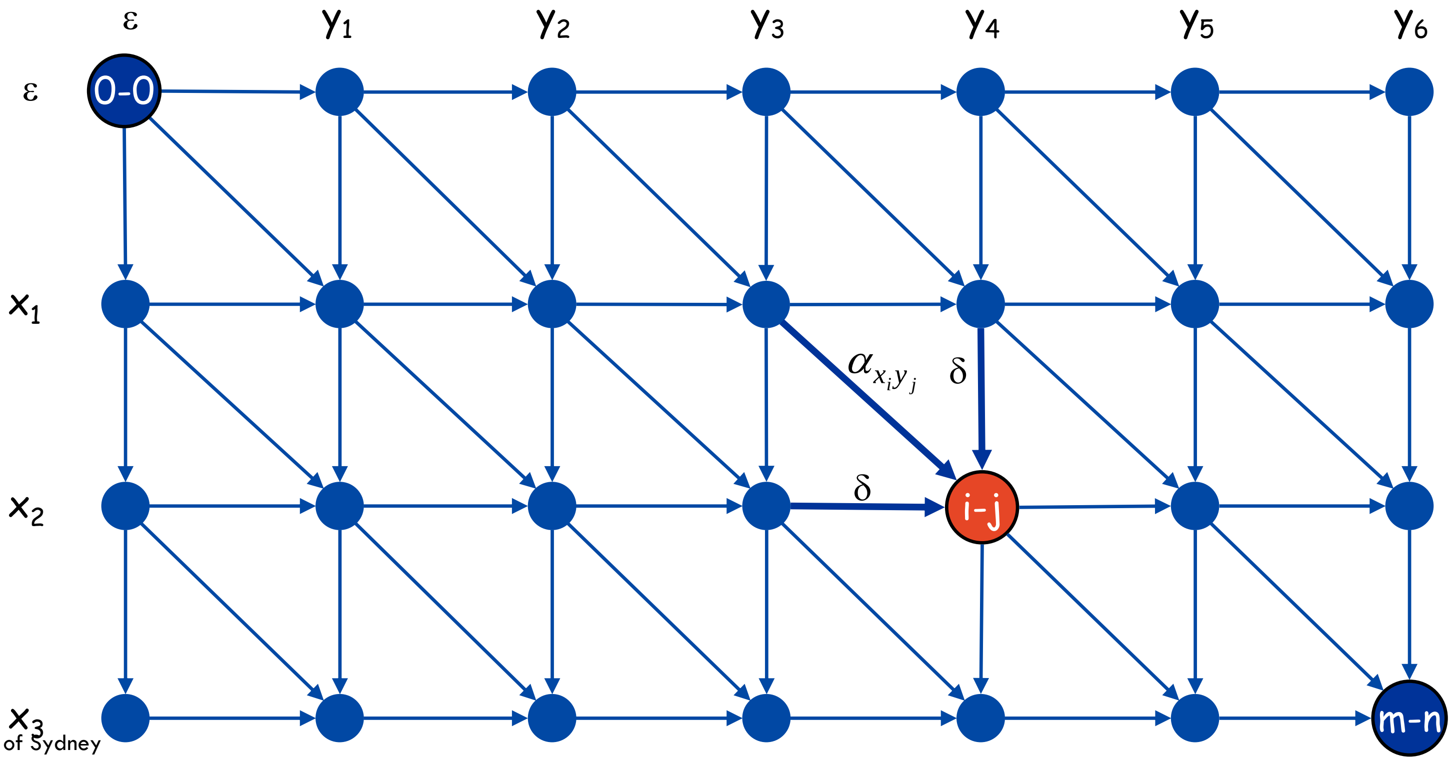
- Compute OPT(i, •) from OPT(i-1, •).
- BUT! No longer a simple way to recover alignment itself.

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.

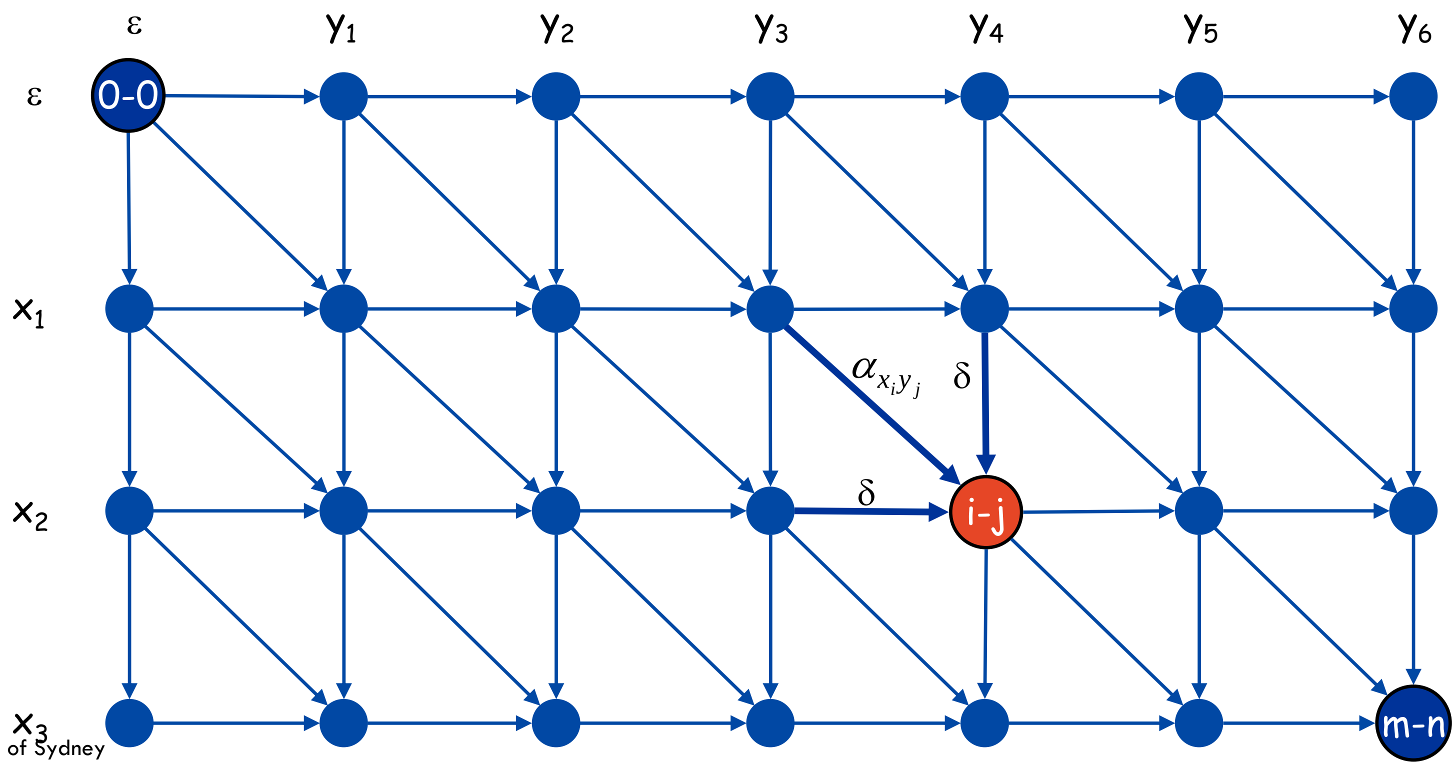
Sequence Alignment: Linear Space

- Edit distance graph.
  - $m \times n$ grid graph G_{XY} (as shown in the figure)
  - Horizontal/vertical edges have cost δ
  - Diagonal edges from (i-1, j-1) to (i,j) have cost $\alpha_{x_i y_j}$.



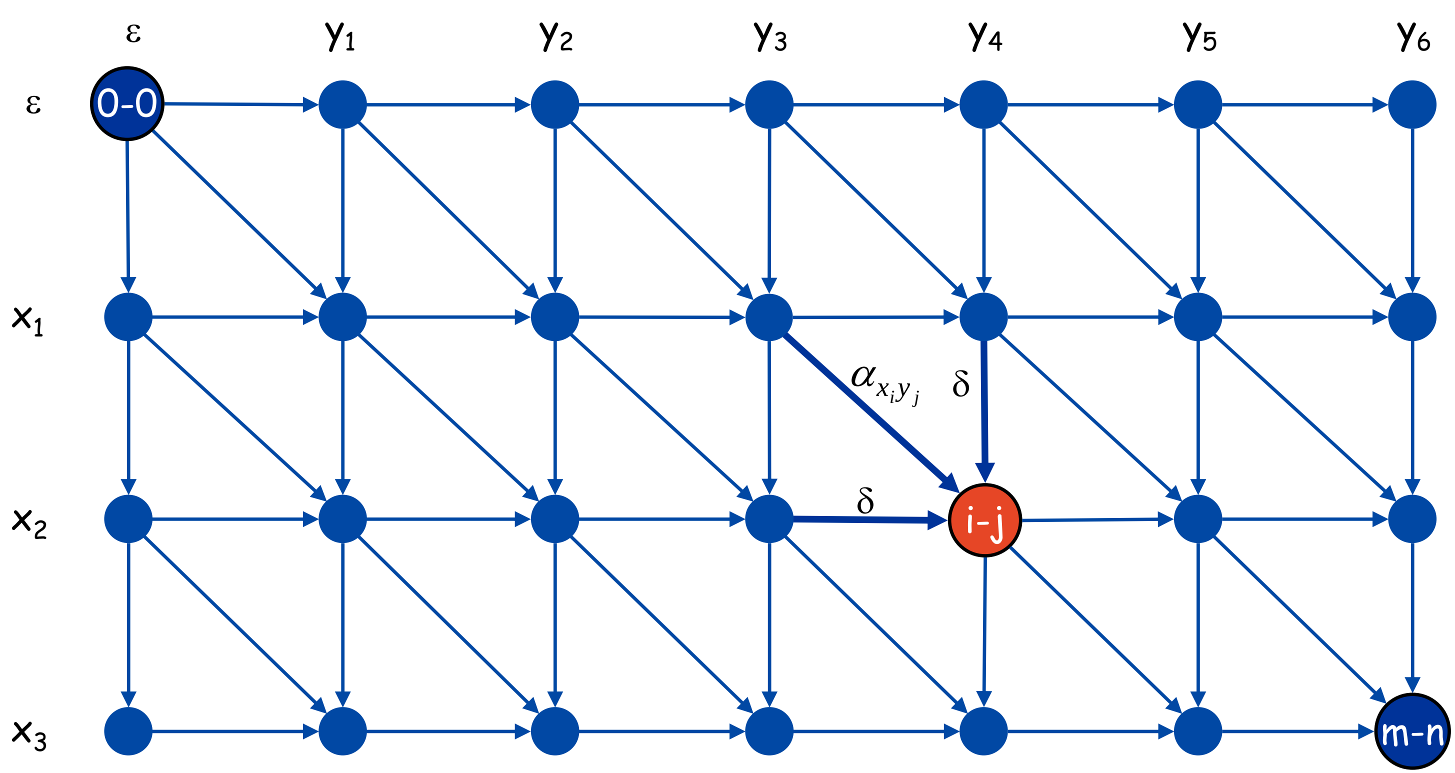
Sequence Alignment: Linear Space

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



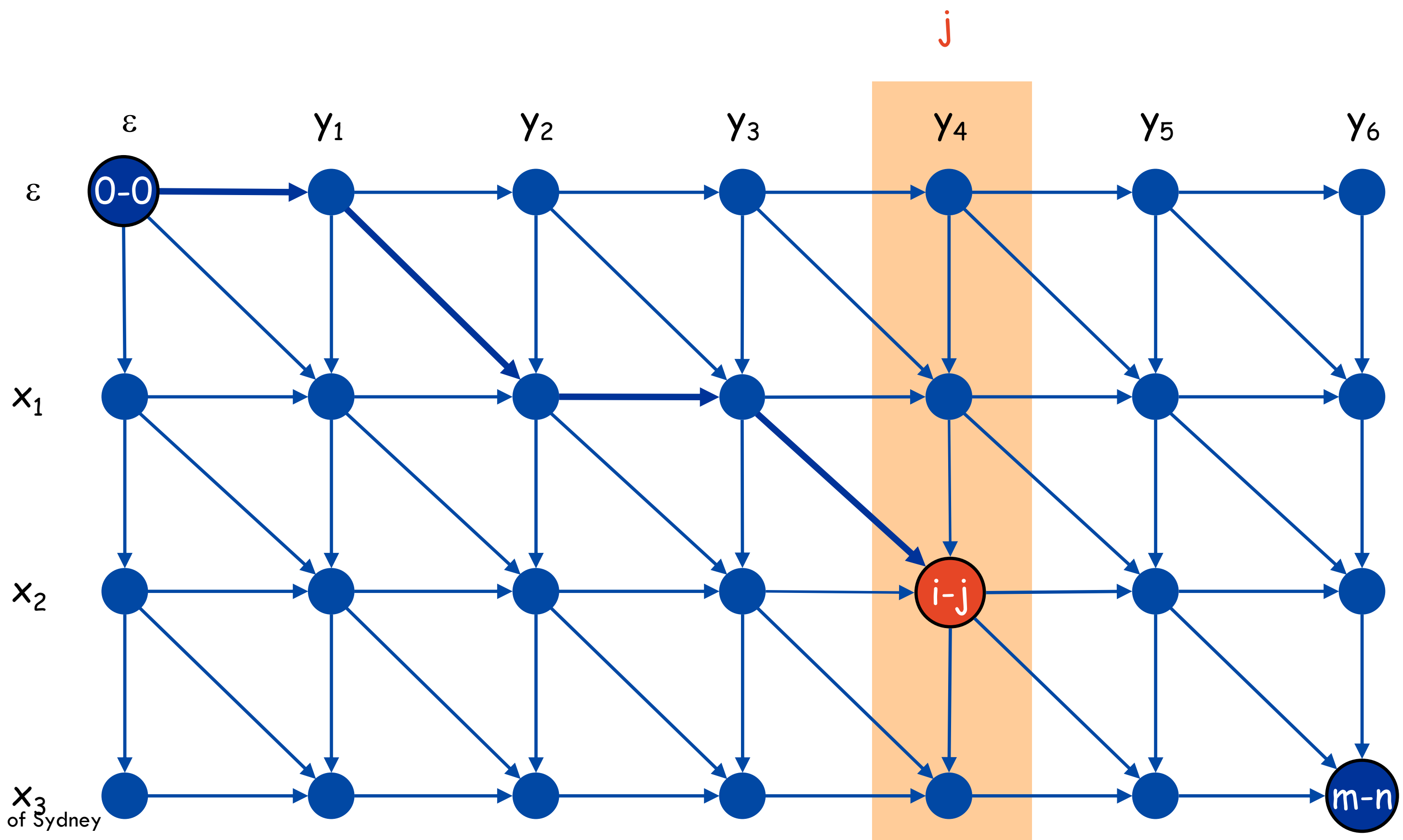
Sequence Alignment: Linear Space

$$\min\{\alpha_{x_i y_j} + OPT(i\text{-}1,\ j\text{-}1), \delta + OPT(i\text{-}1,\ j), \delta + OPT(i,\ j\text{-}1)\}$$



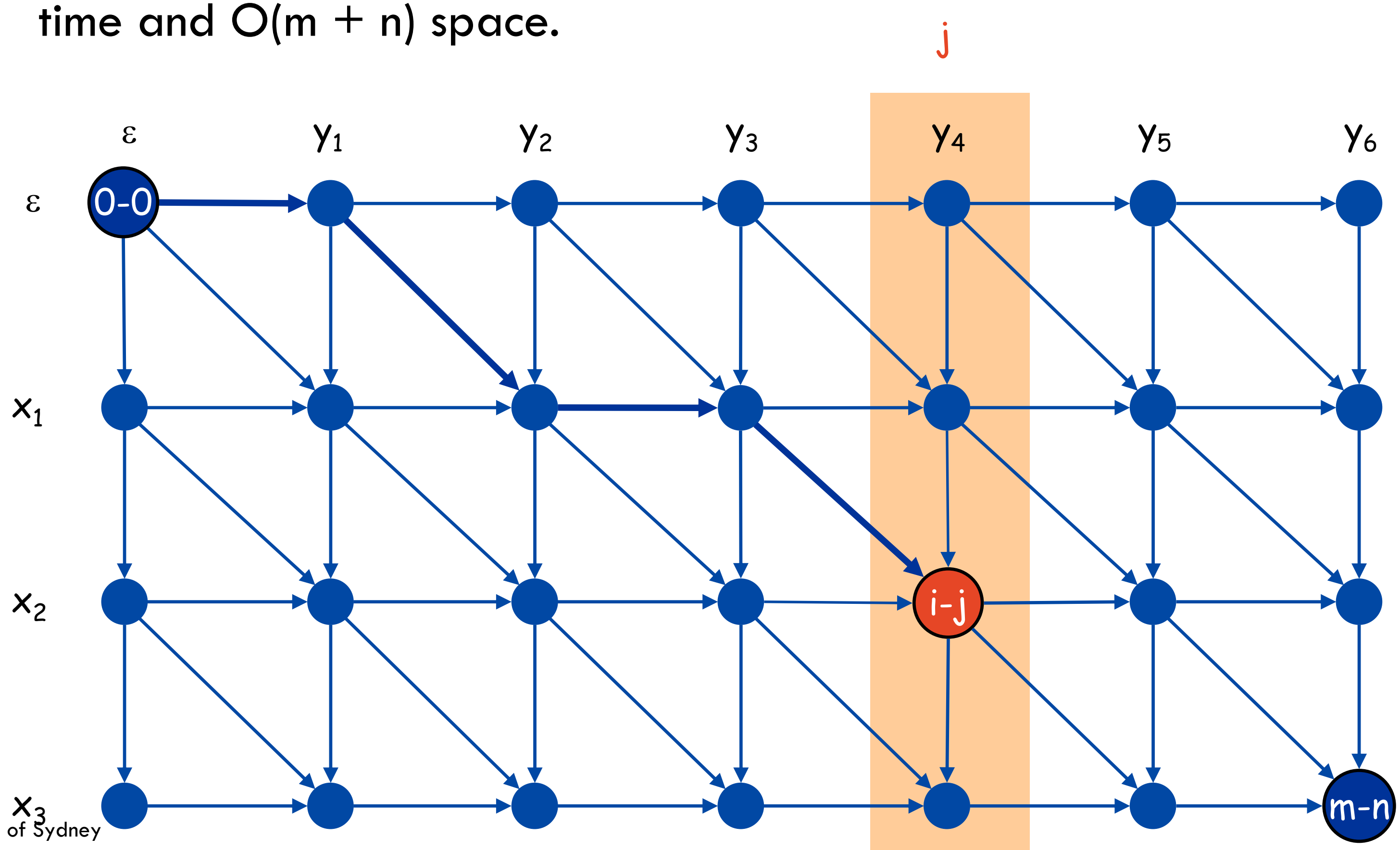
Sequence Alignment: Linear Space

- Edit distance graph.
  - Let f(i, j) be shortest path from (0,0) to (i, j).
  - Can compute f(m,n) in O(mn) time and O(mn) space.



Sequence Alignment: Linear Space

- Edit distance graph.
  - Let f(i, j) be shortest path from (0,0) to (i, j).
  - If only interested in the value of the optimal alignment we do it in O(mn) time and O(m + n) space.



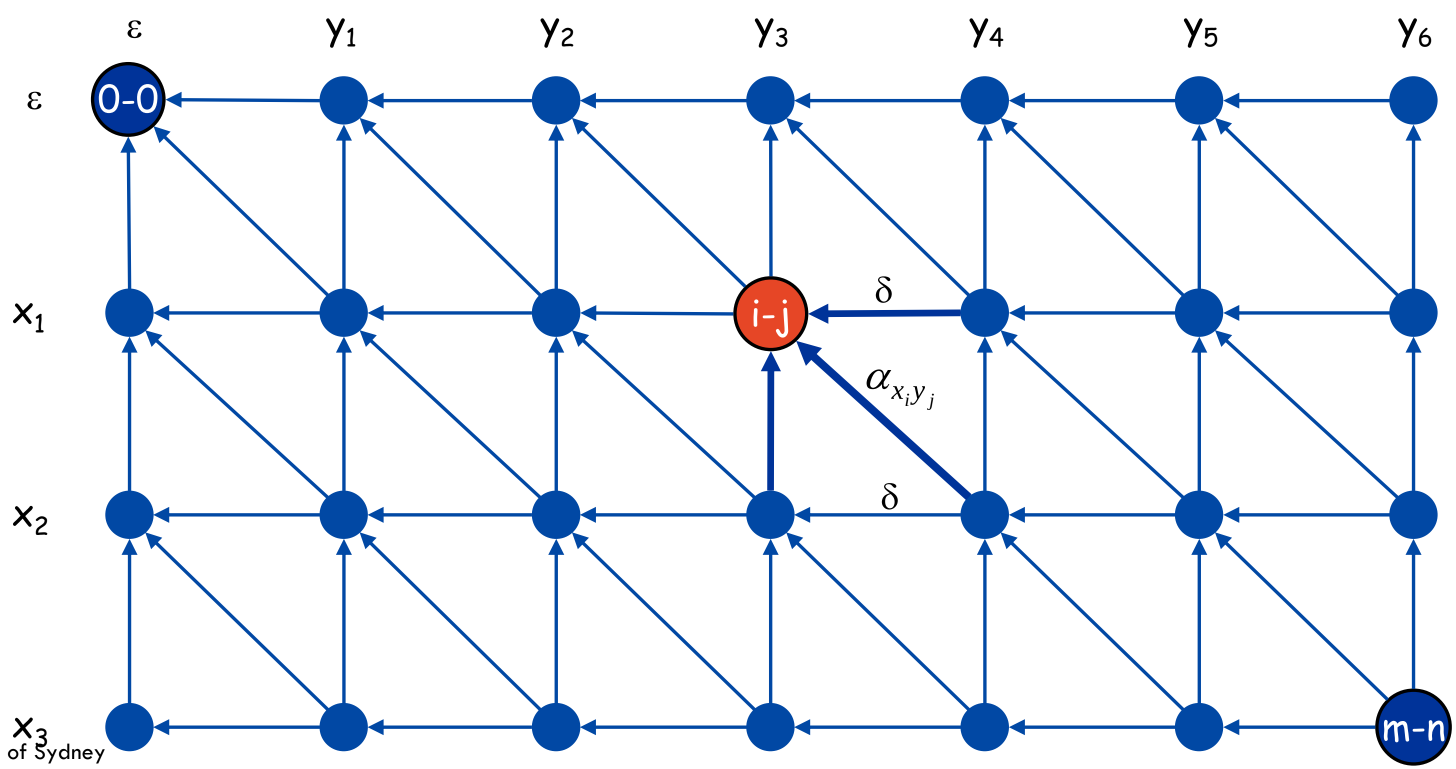
Sequence Alignment: Linear Space

– Edit distance graph.
  – Let f(i, j) be shortest path from (0,0) to (i, j).
  – If only interested in the value of the optimal alignment we do it in O(mn) time and O(m + n) space.

```
Space-Efficient-Alignment(X,Y) {
   array B[0..m,0..1]
   for i = 0 to m
      B[0, i] = iδ
   for j = 1 to n
      B[0,1] = jδ       #corresponds to A[0,j]
      for i = 1 to m
         B[i,1] = min(α[xi, yj] + B[i-1,0],
                          δ + B[i-1,1],δ + B[i,0])
      endFor
      Move column i of B to column 0 #(B[i,0]=B[i,1])
   endFor
}
```

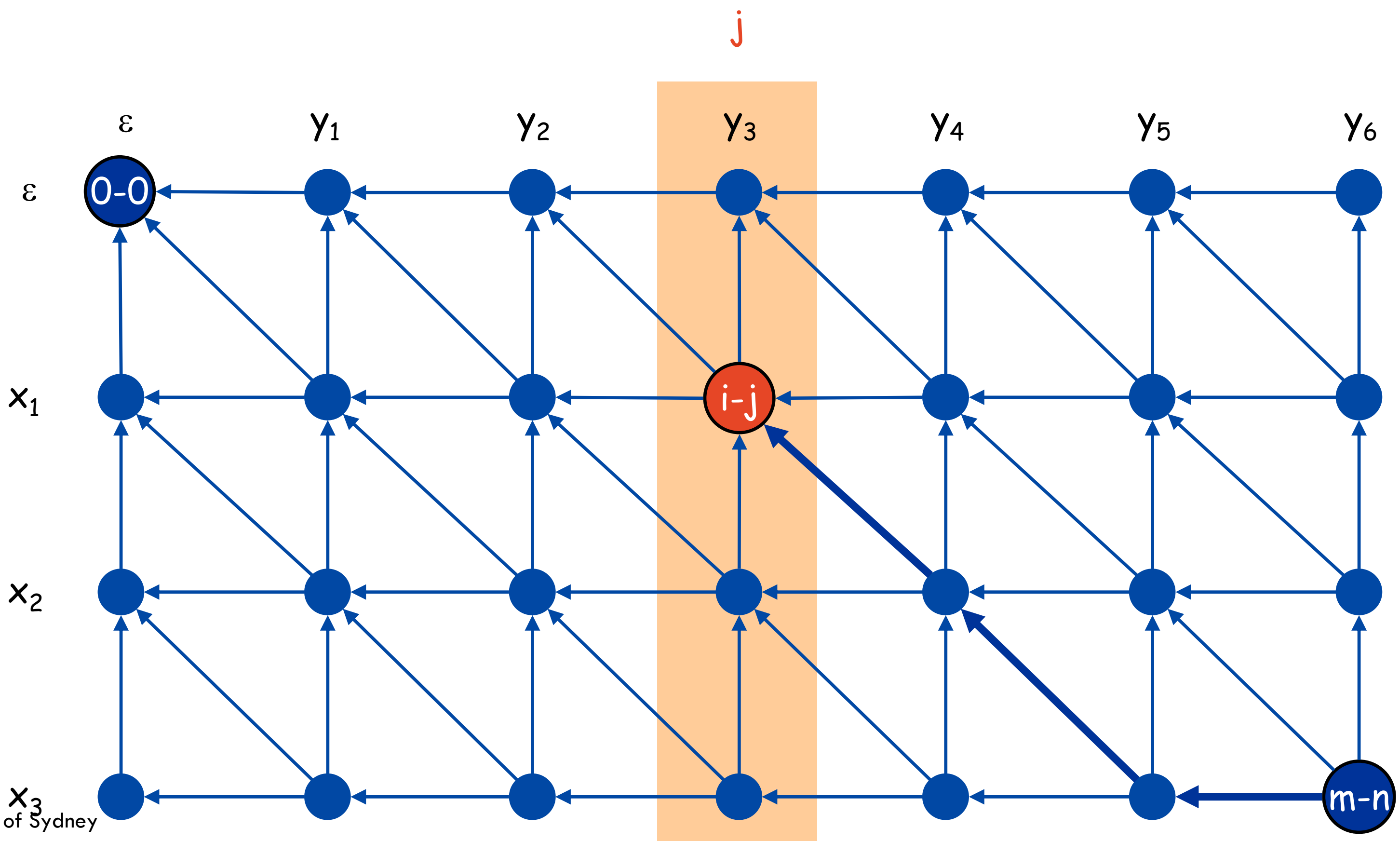
Sequence Alignment: Linear Space

- Edit distance graph.
  - Let g(i, j) be shortest path from (i, j) to (m, n).
  - Can compute by reversing the edge orientations and inverting the roles of (0, 0) and (m, n)



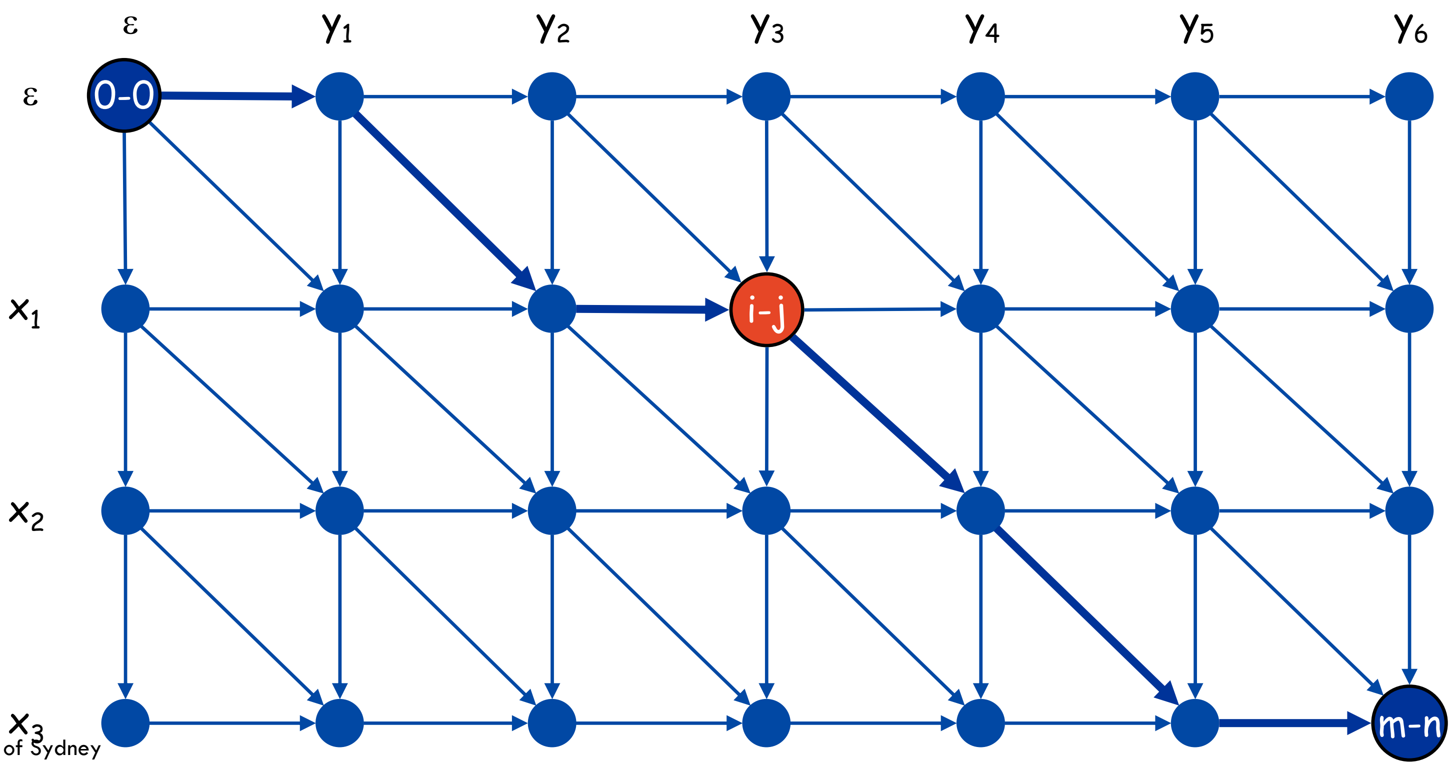
Sequence Alignment: Linear Space

- Edit distance graph.
  - Let $g(i, j)$ be shortest path from (i, j) to (m, n).
  - Can compute $g(\bullet, j)$ for any j in $O(mn)$ time and $O(m + n)$ space.



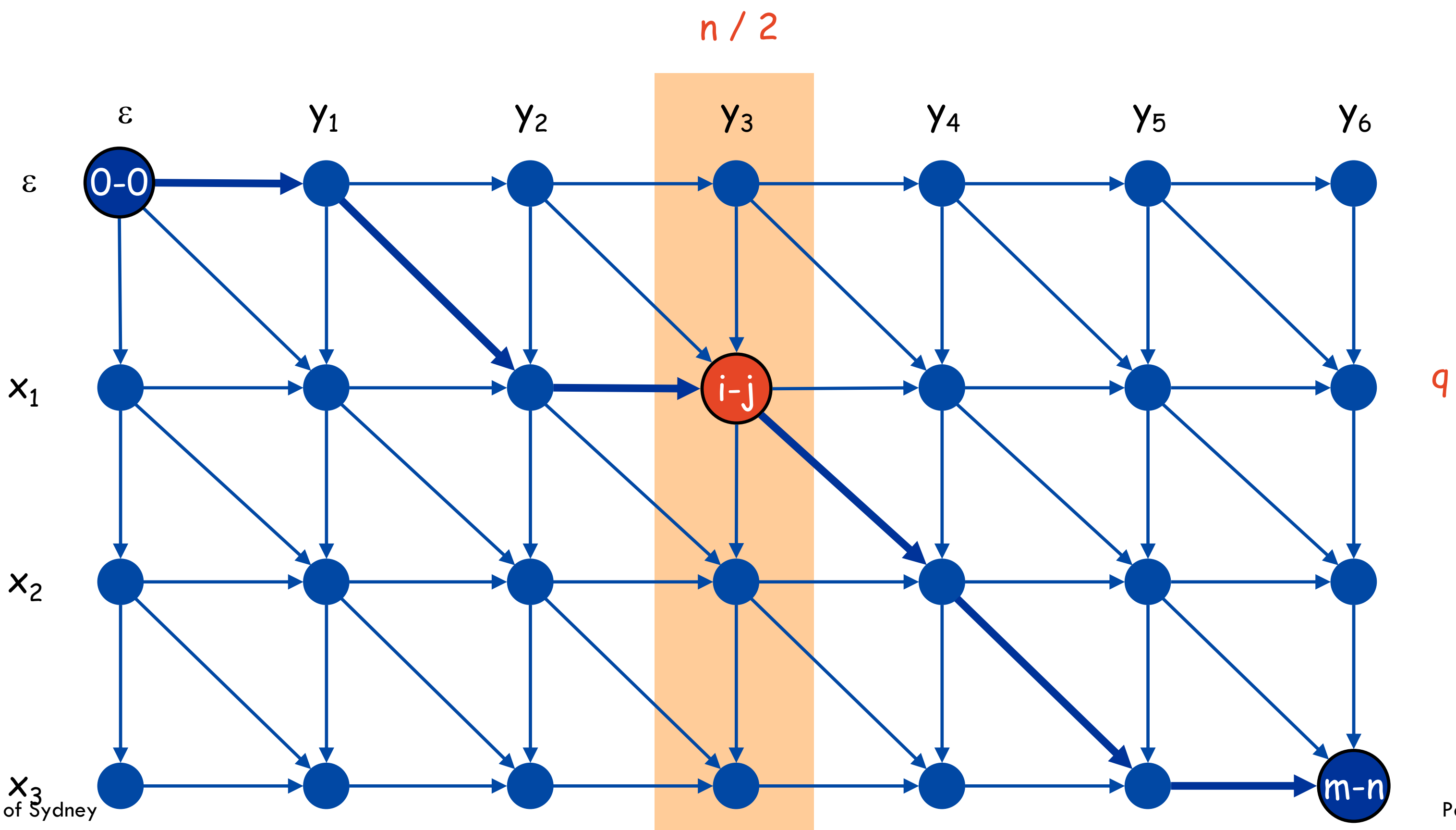
Sequence Alignment: Linear Space

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).

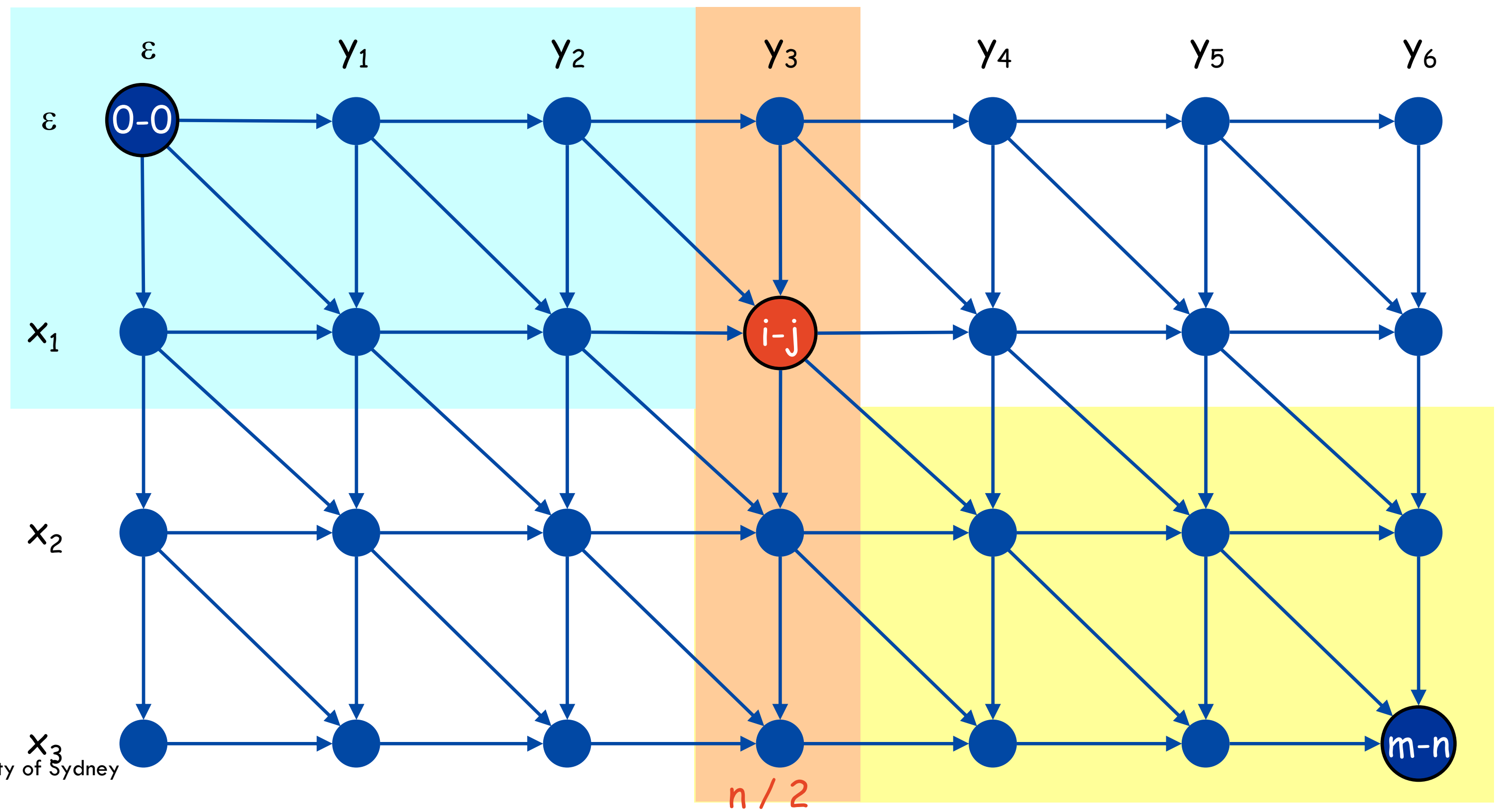


Sequence Alignment: Linear Space

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.



Pseudocode

```
Divide-and-Conquer alignment(X,Y) {
   If |X|≤2 or |Y|≤2 then
      OptimalAlignment(X,Y)    #Alg using quadratic space
   f(·,n/2)=Space-Efficient-Alignment(X,Y[1..n/2])
   g(·,n/2)=Backward-S-E-Alignment(X,Y[n/2..n])
   Let q be the index minimizing f(q,n/2)+g(q,n/2)
   Add (q,n/2) to the global matching
   Divide-and-Conquer alignment(X[1..q],Y[1..n/2])
   Divide-and-Conquer alignment(X,Y)
}
```

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(mn log n).

$$T(m,n) \;\le\; 2T(m,\; n/2) + O(mn) \;\;\Rightarrow\;\; T(m,n) \;=\; O(mn\log n)$$

Remark: Analysis is not tight because two subproblems are of size (q, n/2) and (m - q, n/2).

Sequence Alignment: Running Time Analysis

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

Proof: (by induction on n)

- $O(mn)$ time to compute $f(\bullet, n/2)$ and $g(\bullet, n/2)$ and find index q.
- $T(q, n/2) + T(m - q, n/2)$ time for two recursive calls.

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).

Proof: (by induction on n)

- O(mn) time to compute f(•, n/2) and g (•, n/2) and find index q.
- T(q, n/2) + T(m - q, n/2) time for two recursive calls.

- For some constant c we have:

$$
\begin{aligned}
T(m,\ 2) &\leq cm \\
T(2,\ n) &\leq cn \\
T(m,\ n) &\leq cmn + T(q,\ n/2) + T(m-q,\ n/2)
\end{aligned}
$$

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).

Proof: (by induction on n)

- O(mn) time to compute f(•, n/2) and g (•, n/2) and find index q.
- T(q, n/2) + T(m - q, n/2) time for two recursive calls.
- For some constant c we have:

$$
\begin{aligned}
T(m,\ 2) &\leq cm \\
T(2,\ n) &\leq cn \\
T(m,\ n) &\leq cmn + T(q,\ n/2) + T(m-q,\ n/2)
\end{aligned}
$$

- Base cases: m ≤ 2 or n ≤ 2.
- Inductive hypothesis: T(m', n') ≤ 2cm'n' for m'+n' < m+ n.

$$
\begin{aligned}
T(m,n) &\leq T(q,n/2) + T(m-q,n/2) + cmn \\
&\leq 2cqn/2 + 2c(m-q)n/2 + cmn \\
&= cqn + cmn - cqn + cmn \\
&= 2cmn
\end{aligned}
$$

Sequence Alignment: Running Time Analysis

Theorem: An optimal alignment can be computed in $O(mn)$ time using $O(m+n)$ space.

Sequence Alignment: History

– Needleman and Wunsch 1970 $O(n^3)$

– Sankoff 1972 $O(n^2)$
[see also Vintsyuk'68 for speech processing
Wagner and Fisher'74 for string matching]

– Still an active research area (experimental research)
Chakraborty and Angana'13 (claimed 54-90% speedup)

Generalising the algorithm

Problem:

Nature often inserts or removes entire substrings of nucleotides (creating long gaps), rather than editing just one position at a time.

The penalty for a gap of length 10 should not be 10 times the penalty for a gap of length 1, but something significantly smaller. Can we modify the scoring function in which the penalty for a gap of length k is:

$$\delta_0 + \delta_1 \cdot k \quad ?$$

Dynamic Programming Summary

- **1D dynamic programming**
  - Weighted interval scheduling
  - Segmented Least Squares
  - Maximum-sum contiguous subarray
  - Longest increasing subsequence
- **2D dynamic programming**
  - Knapsack
  - Sequence alignment
- **Dynamic programming over intervals**
  - RNA Secondary Structure
- **Dynamic programming over subsets**
  - TSP
  - k-path
  - Playlist

6.10 Negative Cycles in a Graph

Detecting Negative Cycles

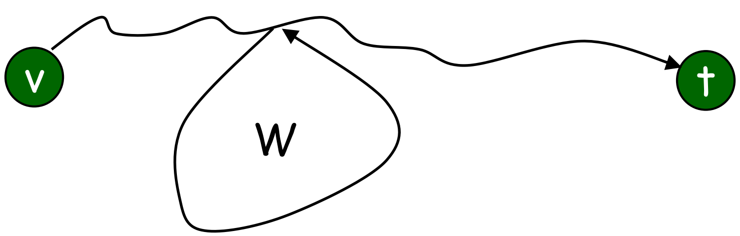
Lemma: If OPT(n,v) = OPT(n-1,v) for all v, then no negative cycles.

Proof: Bellman-Ford algorithm.

Lemma. If OPT(n,v) < 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)

- Since OPT(n,v) < 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-t path with < n edges $\Rightarrow$ W has negative cost.

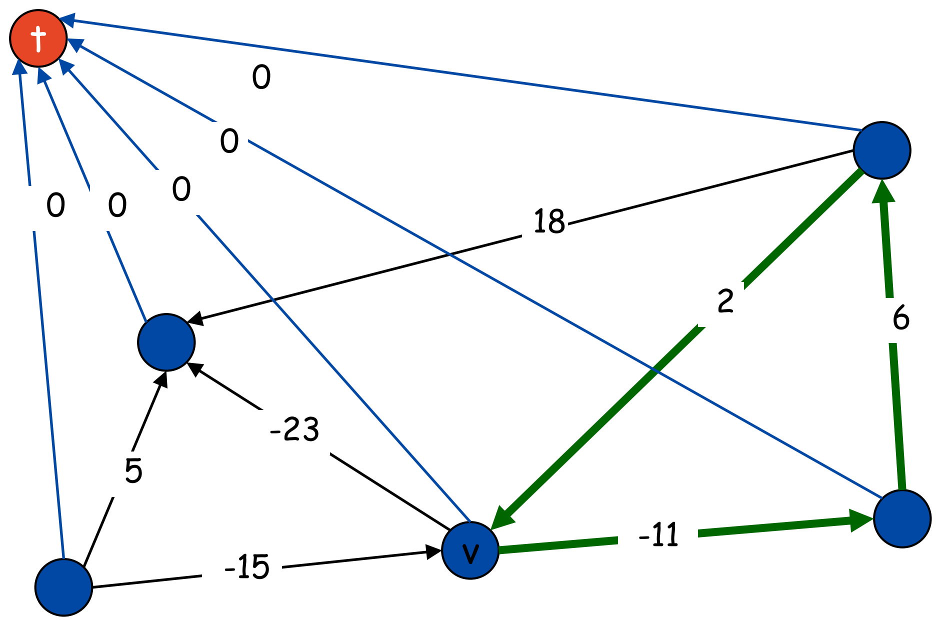


c(W) < 0

Detecting Negative Cycles

Theorem: Can detect negative cost cycle in O(mn) time.

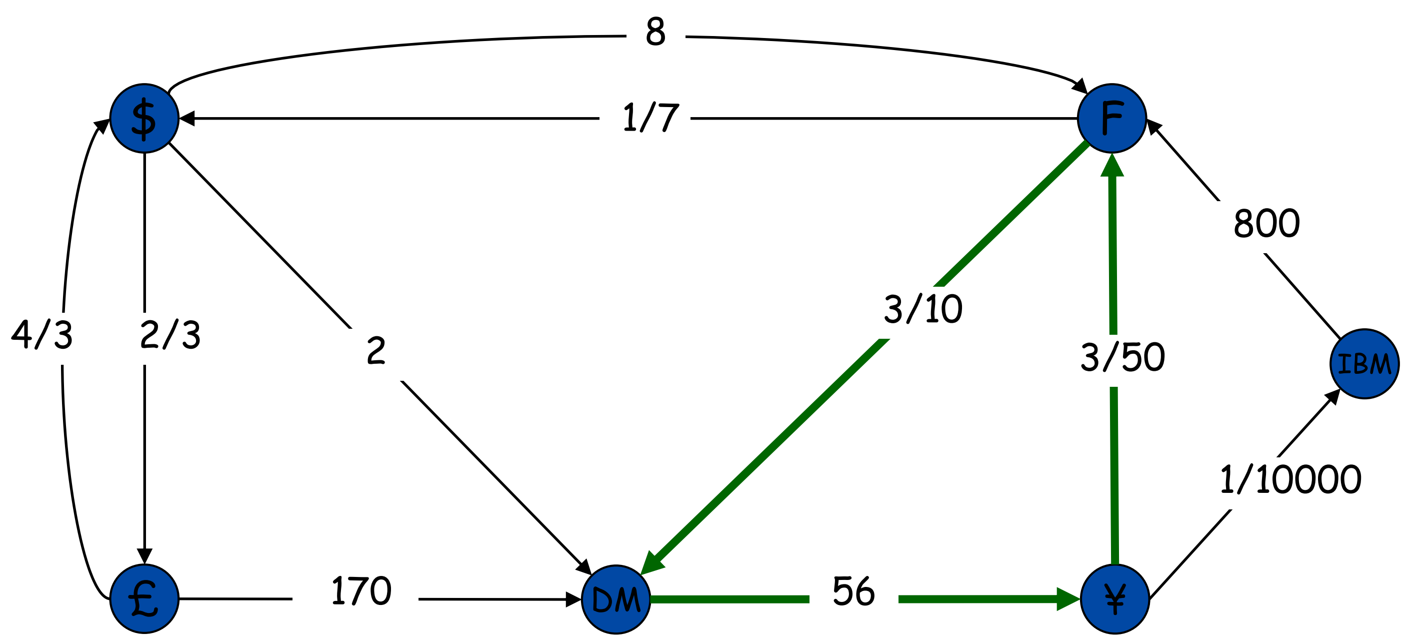
- Add new node t and connect all nodes to t with 0-cost edge.
- Check if OPT(n, v) = OPT(n-1, v) for all nodes v.
  - if yes, then no negative cycles
  - if no, then extract cycle from shortest path from v to t



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.