Chapter 6

Dynamic Programming

Algorithm Design

JON KLEINBERG · ÉVA TARDOS

Algorithmic Paradigms

Greedy. Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer. Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Dynamic Programming History

Bellman. Pioneered the systematic study of dynamic programming in the 1950s.

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.
  - "it's impossible to use dynamic in a pejorative sense"
  - "something not even a Congressman could object to"

Reference: Bellman, R. E. Eye of the Hurricane, An Autobiography.

Dynamic Programming Applications

Areas.

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, systems, ….

Some famous dynamic programming algorithms.

- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- Smith-Waterman for sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

6.1 Weighted Interval Scheduling

Weighted Interval Scheduling

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.
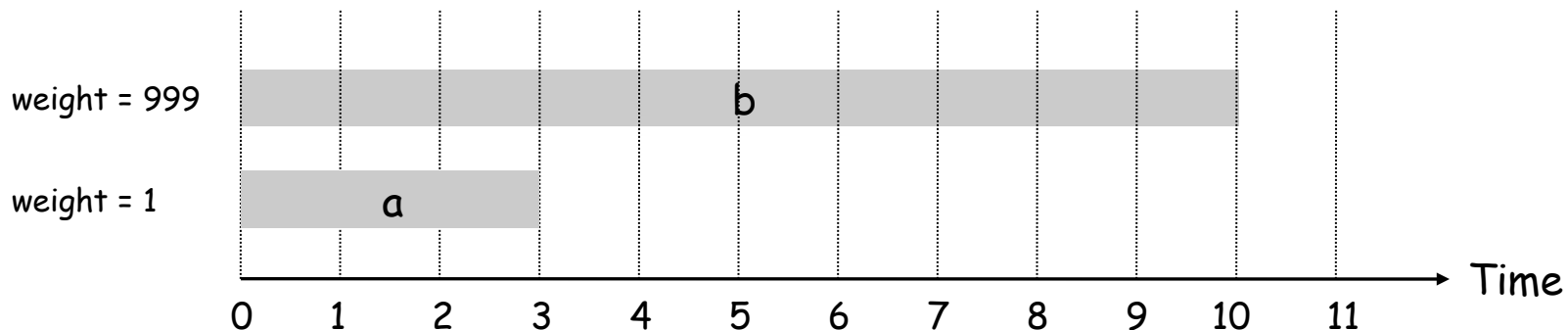- Goal: find maximum weight subset of mutually compatible jobs.
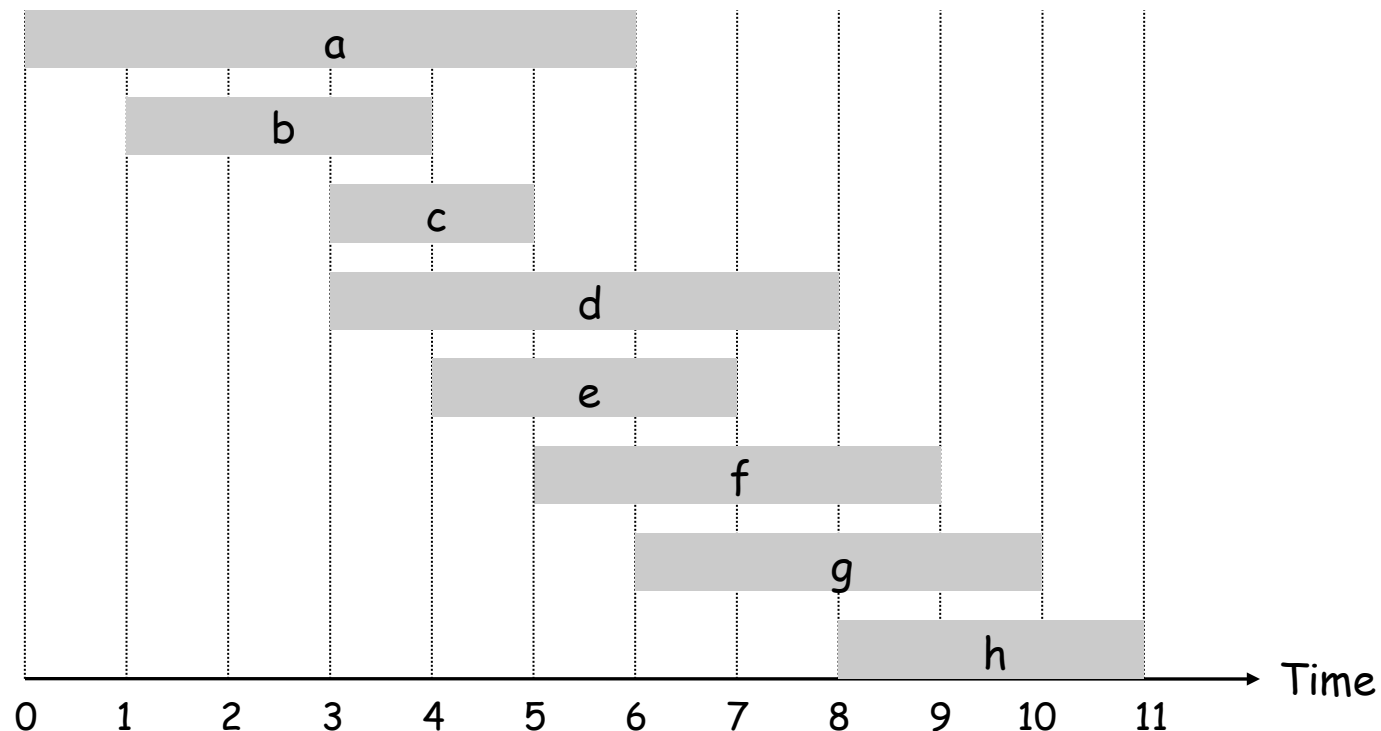
Unweighted Interval Scheduling Review

Recall. Greedy algorithm works if all weights are 1.
- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.
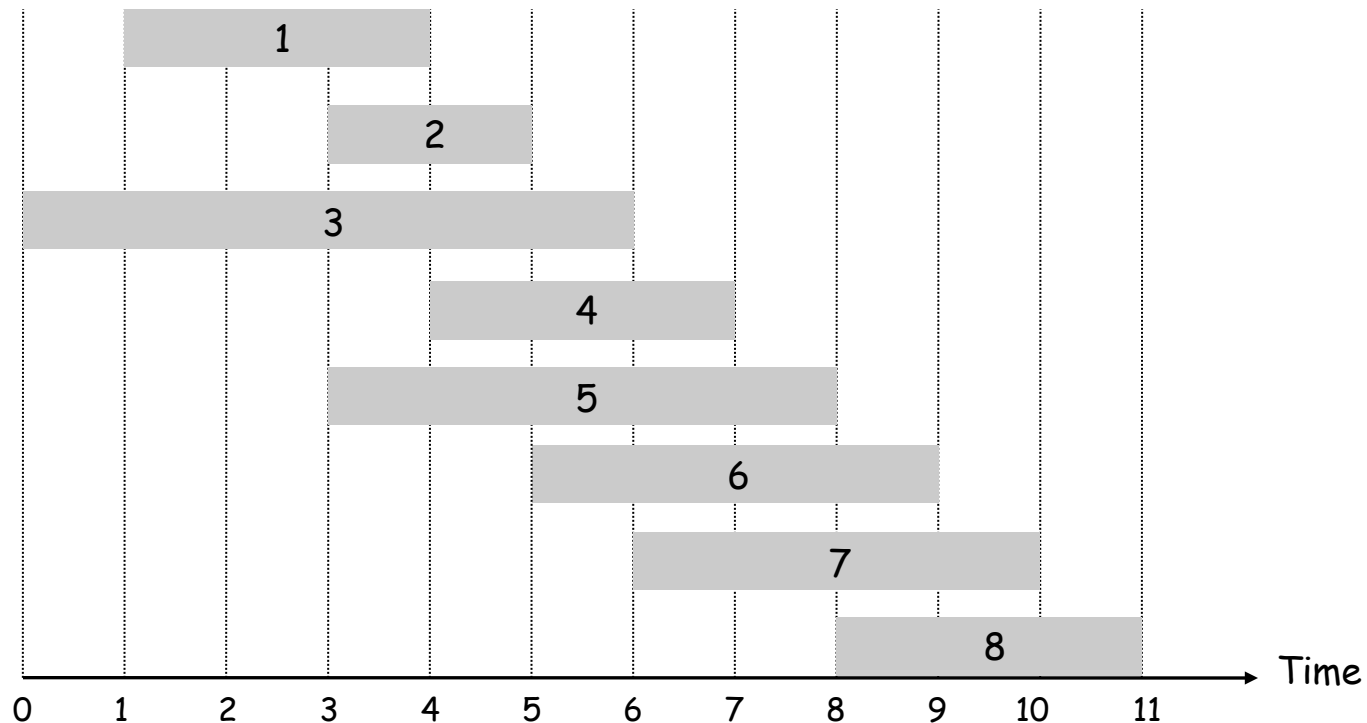
Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.
Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j.

Ex: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.

Dynamic Programming: Binary Choice

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

- Case 1: OPT selects job j.
    - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
    - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

  optimal substructure

- Case 2: OPT does not select job j.
    - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., 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

Brute force algorithm.

```
Input: n, s₁,…,sₙ , f₁,…,fₙ , v₁,…,vₙ


Sort jobs by finish times so that f₁ ≤ f₂ ≤ ... ≤ fₙ.


Compute p(1), p(2), …, p(n)


Compute-Opt(j) {
    if (j = 0)
        return 0
    else
        return max(vⱼ + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```
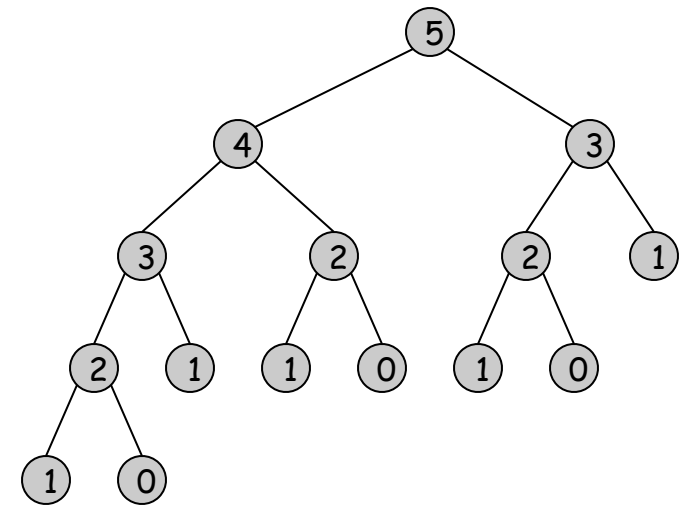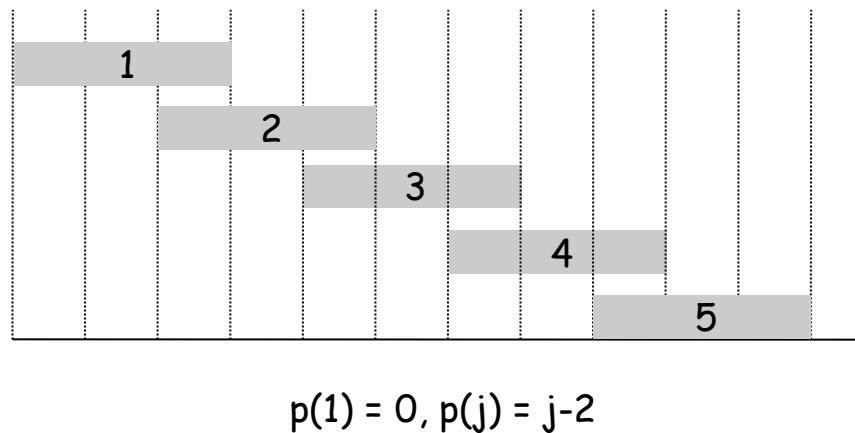
Weighted Interval Scheduling: Brute Force

Observation. Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.
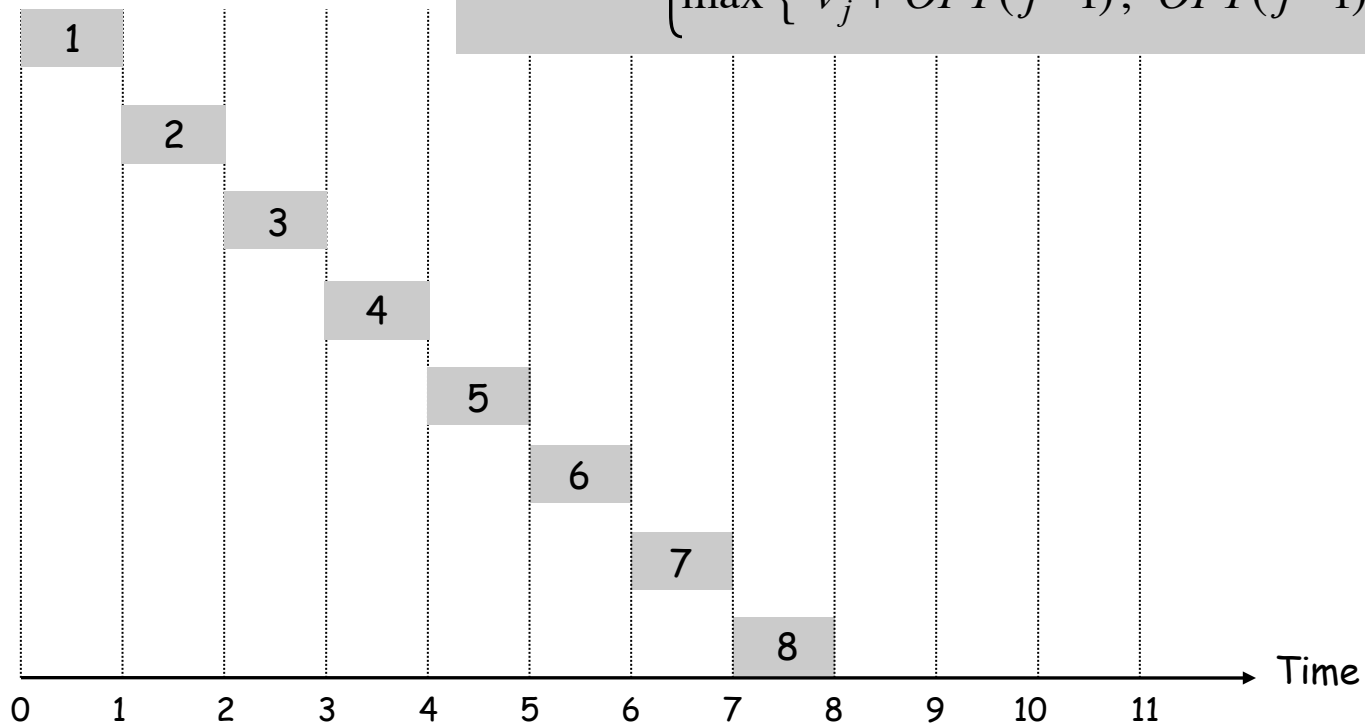

p(1) = 0, p(j) = j-2

Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. p(j) = largest index i < j such that job i is compatible with j.

Ex: p(8) = 7, p(7) = 6, p(j) = j–1. $OPT(0)$ is computed 2^n times.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left\{ v_j + OPT(j-1),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Time

0   1   2   3   4   5   6   7   8   9   10   11

Weighted Interval Scheduling: Memoization

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

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n


Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty      ← global array
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]
}
```

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes $O(n \log n)$ time.
- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n \log n)$, $O(n)$ after sorting by start time :

scan both $(s_1, s_2,..., s_n)$ & $(f_1, f_2,..., f_n)$.

- `M-Compute-Opt(j)`: each invocation takes $O(1)$ time and either
  - (i) returns an existing value `M[j]`
  - (ii) fills in one new entry `M[j]` and makes two recursive calls

- Progress measure Φ = # nonempty entries of `M[]`.
  - initially Φ = 0, throughout $\Phi \leq n$.
  - (ii) increases Φ by 1 $\Rightarrow$ at most $2n$ recursive calls.

- Overall running time of `M-Compute-Opt(n)` is $O(n)$. ▪

Remark. $O(n)$ if jobs are pre-sorted by start and finish times.
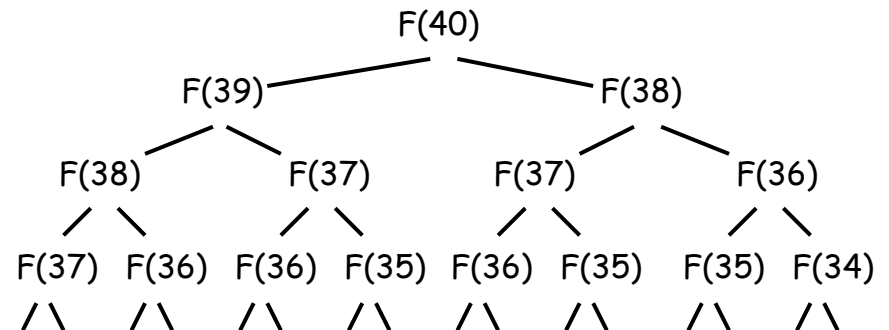
Automated Memoization

Automated memoization. Many functional programming languages (e.g., Lisp) have built-in support for memoization.

```lisp
(defun F (n)
  (if
    (<= n 1)
    n
    (+ (F (- n 1)) (F (- n 2)))))
```

Lisp (efficient)

```java
static int F(int n) {
    if (n <= 1) return n;
    else return F(n-1) + F(n-2);
}
```

Java (exponential)

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?

A. Do some post-processing.

```
Run M-Compute-Opt(n)
Run Find-Solution(n)


Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (vⱼ + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

- \# of recursive calls ≤ n ⇒ O(n).

Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(v_j + M[p(j)], M[j-1])
}
```
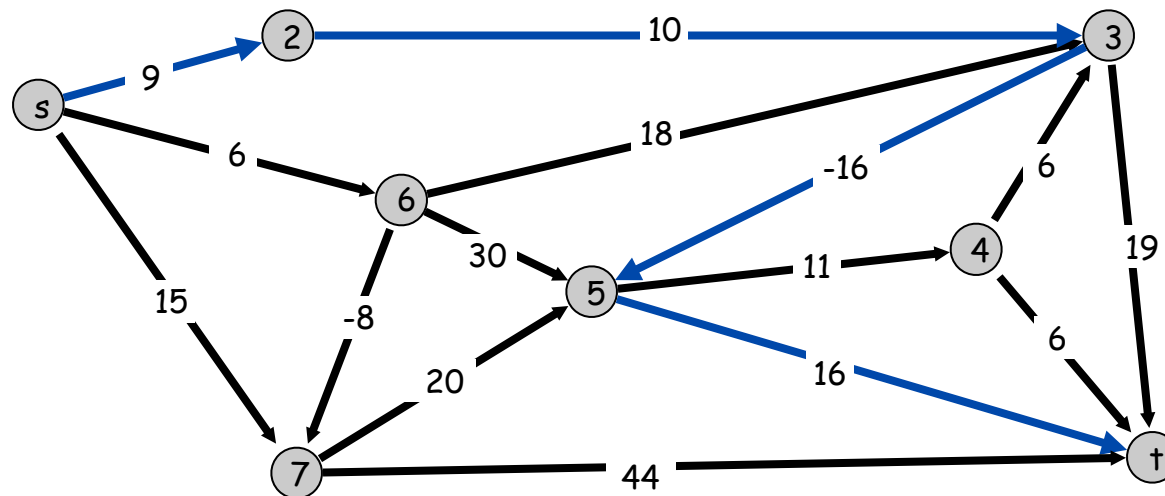
6.8 Shortest Paths

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.
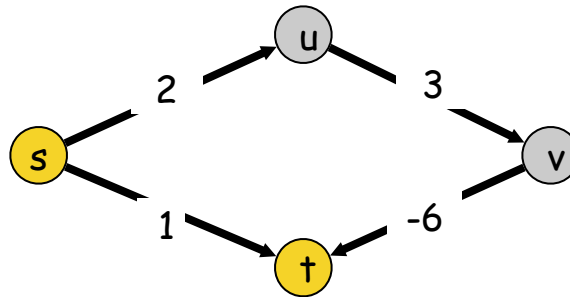
allow negative weights

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.
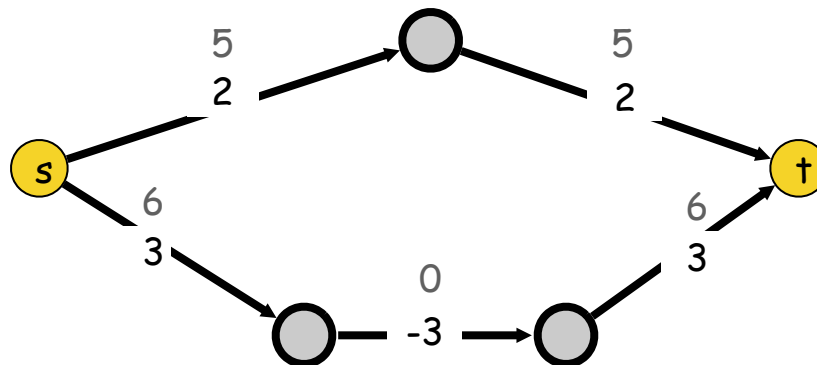
Shortest Paths: Failed Attempts
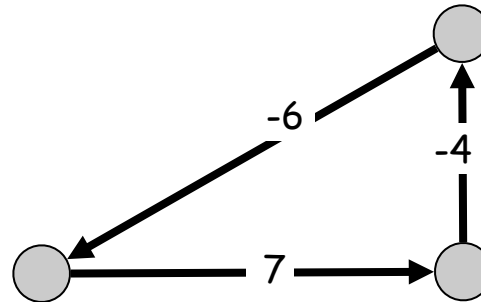
Dijkstra. Can fail if negative edge costs.

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

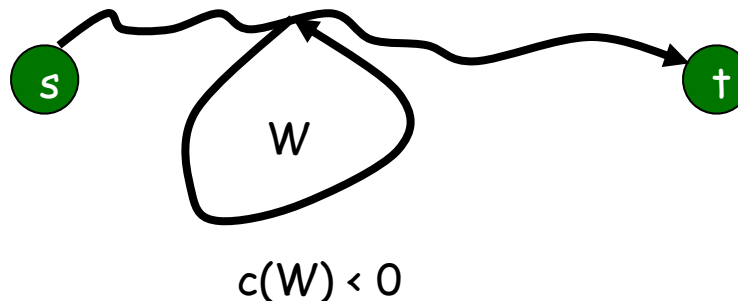# Shortest Paths: Negative Cost Cycles

Negative cost cycle.

-6

-4

7

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.

s

t

W

c(W) < 0

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

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

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

Shortest Paths: Implementation

```
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 (u, w) ∈ E
            M[i, u] ← min { M[i,u], M[i-1,w] + c_uw }
}
```

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

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

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

Bellman-Ford: Efficient Implementation

```
Push-Based-Shortest-Path(G, 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.
    }
}
```

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.
Pf. 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.

Pf. (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 ⇒ W has negative cost.

c(W) < 0
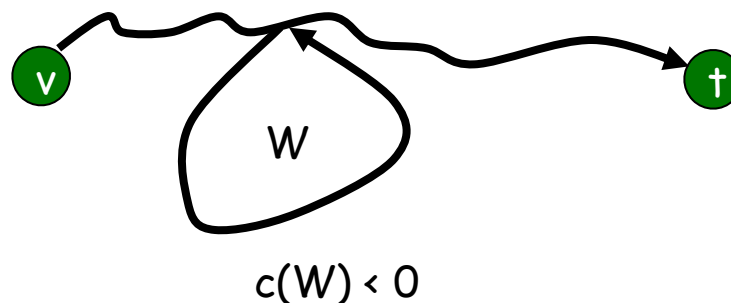
Detecting Negative Cycles

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
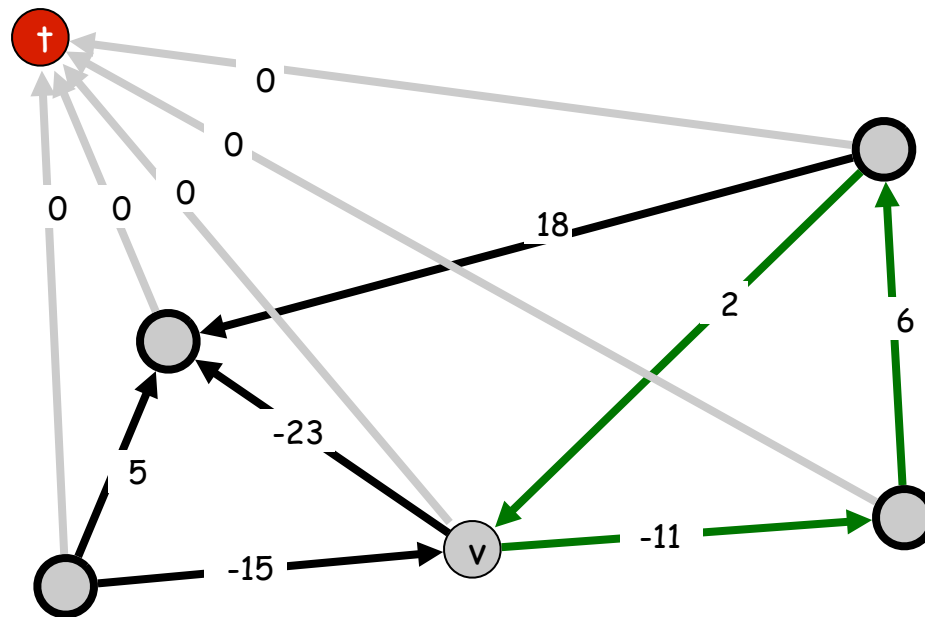
Detecting Negative Cycles: Application

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

Remark. Fastest algorithm very valuable!
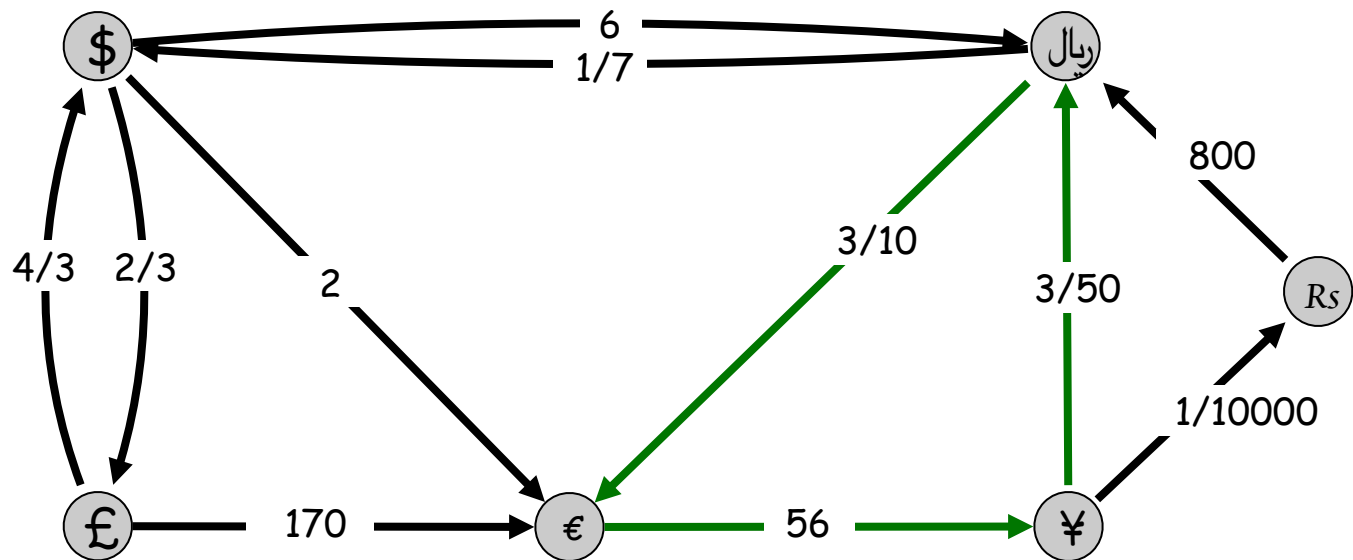
Detecting Negative Cycles: Summary

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.
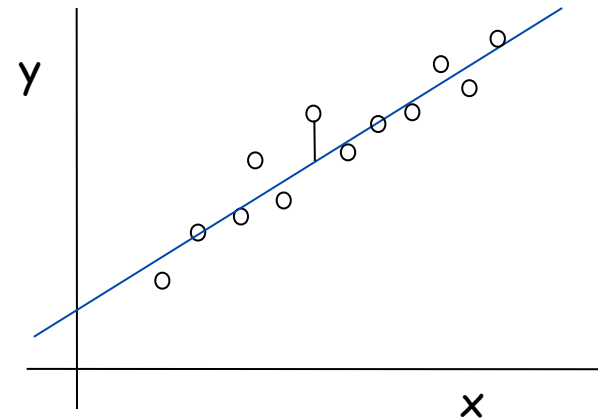- See p. 288 for improved version and early termination rule.

6.3 Segmented Least Squares

Segmented Least Squares

Least squares.

- Foundational problem in statistic 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:

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

Solution. Calculus $\Rightarrow$ min error is achieved when

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

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

Segmented Least Squares

Segmented least squares.
- Points lie roughly on a sequence of several line segments.
- Given n points in the plane (x_1, y_1), (x_2, y_2) , . . . , (x_n, y_n) with
- $x_1 < x_2 < ... < x_n$, find a sequence of lines that minimizes f(x).

Q. What's a reasonable choice for f(x) to balance accuracy and parsimony?
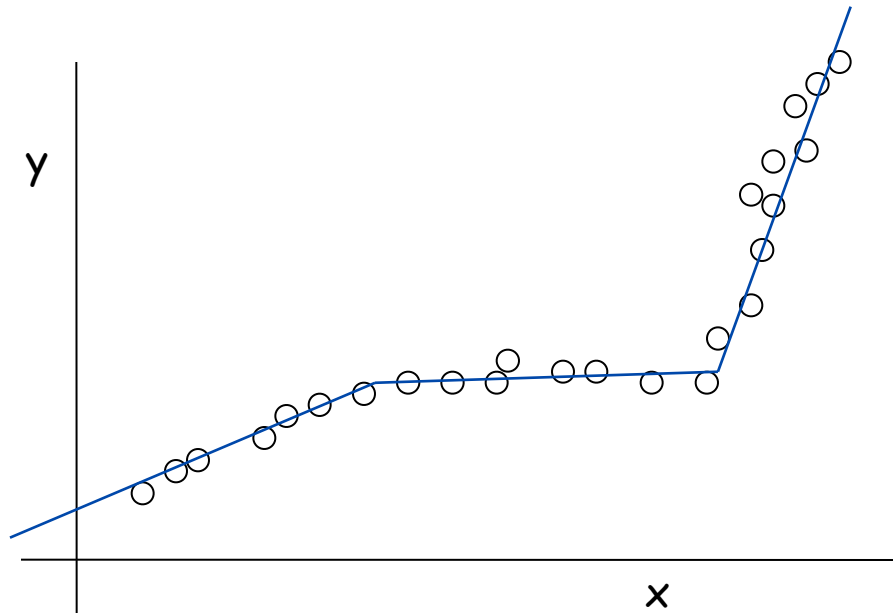
goodness of fit

number of lines

Segmented Least Squares
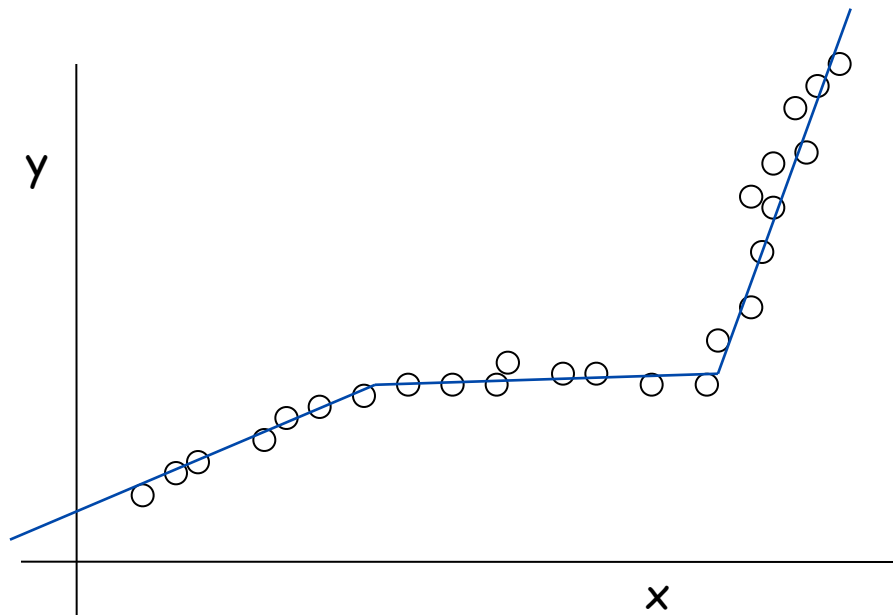
Segmented least squares.
- Points lie roughly on a sequence of several line segments.
- Given n points in the plane (x_1, y_1), (x_2, y_2) , . . . , (x_n, y_n) with
- $x_1 < x_2 < ... < x_n$, find a sequence of lines that minimizes:
  - the sum of the sums of the squared errors E in each segment
  - the number of lines L
- Tradeoff function: E + c L, for some constant c > 0.

Dynamic Programming: Multiway Choice

Notation.

- 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).

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

Segmented Least Squares: Algorithm

```
INPUT: n, p₁,…,pₙ , c

Segmented-Least-Squares() {
   M[0] = 0
   for j = 1 to n
      for i = 1 to j
         compute the least square error eᵢⱼ for
         the segment pᵢ,…, pⱼ


   for j = 1 to n
      M[j] = min ₁ ≤ ᵢ ≤ ⱼ (eᵢⱼ + c + M[i-1])


   return M[n]
}
```

can be improved to $O(n^2)$ by pre-computing various statistics

Running time. $O(n^3)$.

- Bottleneck = computing e_{ij} for $O(n^2)$ pairs, $O(n)$ per pair using previous formula.
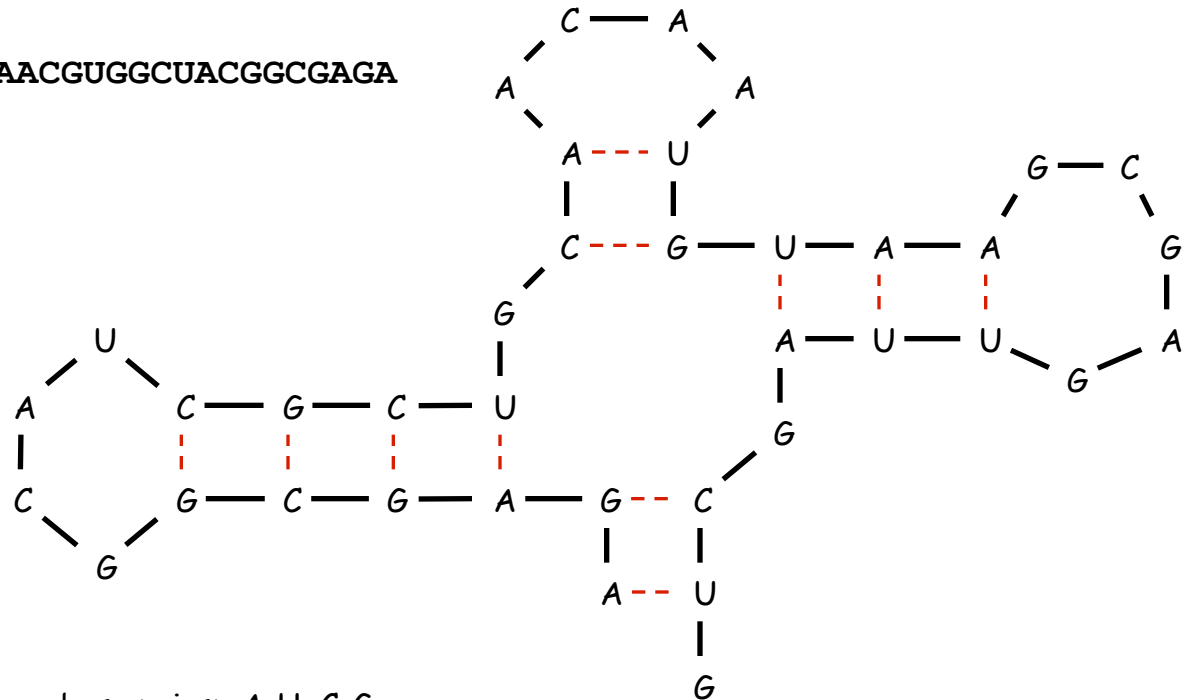
6.5 RNA Secondary Structure

RNA Secondary Structure

RNA. String $B = b_1 b_2 ... b_n$ 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 behaviour of molecule.

Ex: GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA

complementary base pairs: A-U, C-G

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.

approximate by number of base pairs

Goal. Given an RNA molecule B = $b_1 b_2 ... b_n$, find a secondary structure S that maximizes the number of base pairs.

RNA Secondary Structure: Examples

Examples.

base pair

A U G U G G C C A U

A U G G G G C A U

⟵ ≤4 ⟶
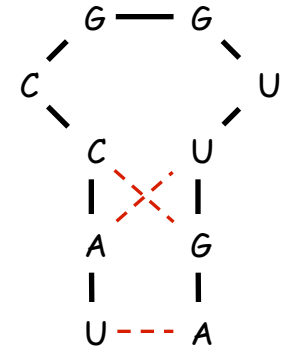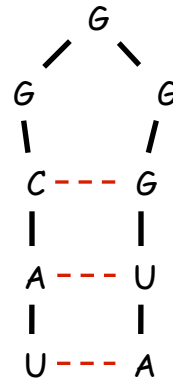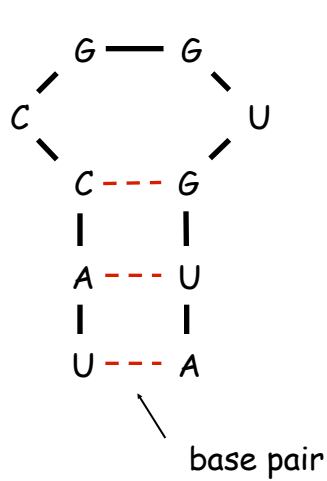
A G U U G G C C A U

ok   sharp turn   crossing

RNA Secondary Structure: Subproblems

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

match b_t and b_n

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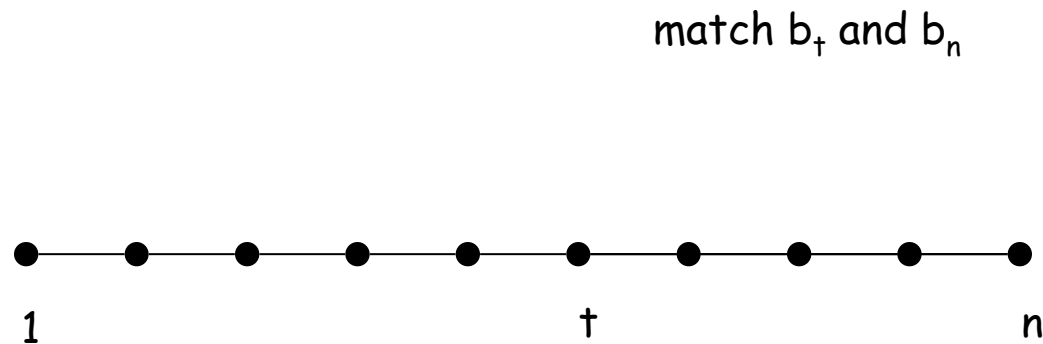
Difficulty. Results in two sub-problems.
- Finding secondary structure in: $b_1 b_2 \ldots b_{t-1}$.   ⟵ OPT(t-1)
- Finding secondary structure in: $b_{t+1} b_{t+2} \ldots b_{n-1}$.   ⟵ need more sub-problems

Dynamic Programming Over Intervals

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

- 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, $t-1$) + OPT($t+1$, $j-1$) }

$i \leq t < j-4$ and b_t and b_j
are Watson-Crick complements

i	i+1	i+2						j-5	j-4	j-3	j-2	j-1	j
A	U	G	C	U	G	U	C	A	U	G	U	C	U	A

Remark. Same core idea in Cocke–Younger–Kasami 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.

```
RNA(b₁,…,bₙ) {
    for k = 5, 6, …, n-1
        for i = 1, 2, …, n-k
            j = i + k
            Compute M[i, j]

    return M[1, n]
}
```

using recurrence

Running time. $O(n^3)$.

6.6 Sequence Alignment

String Similarity

How similar are two strings?

- **ocurrance**
- **occurrence**

o	c	u	r	r	a	n	c	e	-

o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

o	c	-	u	r	r	a	n	c	e

o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

o	c	-	u	r	r	-	a	n	c	e

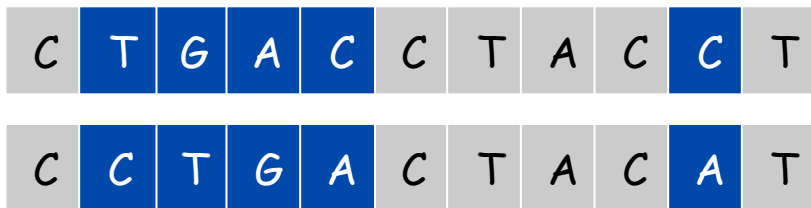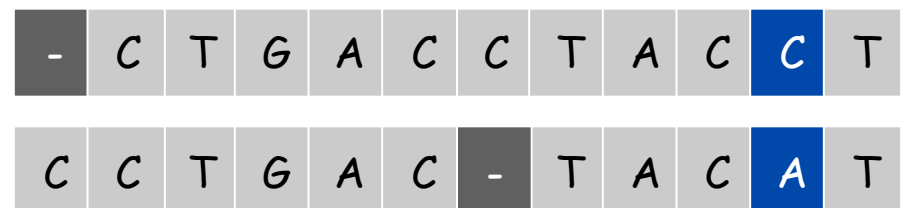| o	c	c	u	r	r	e	-	n	c	e

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; mismatch penalty α_{xy}. ($\alpha_{uu} = 0$)
- 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.

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 pairs x_i, y_j and $x_{i'}, y_{j'}$ cross if $i < i'$, but $j > j'$.

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

Ex: CTACCG **vs.** TACATG.

Sol: $M = x_2, y_1,\ x_3, y_2,\ x_4, y_3,\ x_5, y_4,\ x_6, y_6.$

x_1	x_2	x_3	x_4	x_5		x_6
C	T	A	C	C	-	G

	y_1	y_2	y_3	y_4	y_5	y_6
-	T	A	C	A	T	G

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 $\alpha_{x_i y_j}$ 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 δ 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 δ for y_j and min cost of aligning $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_{j-1}$

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

Sequence Alignment: 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: Linear Space

Q. 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, •).
- No longer a simple way to recover alignment itself.

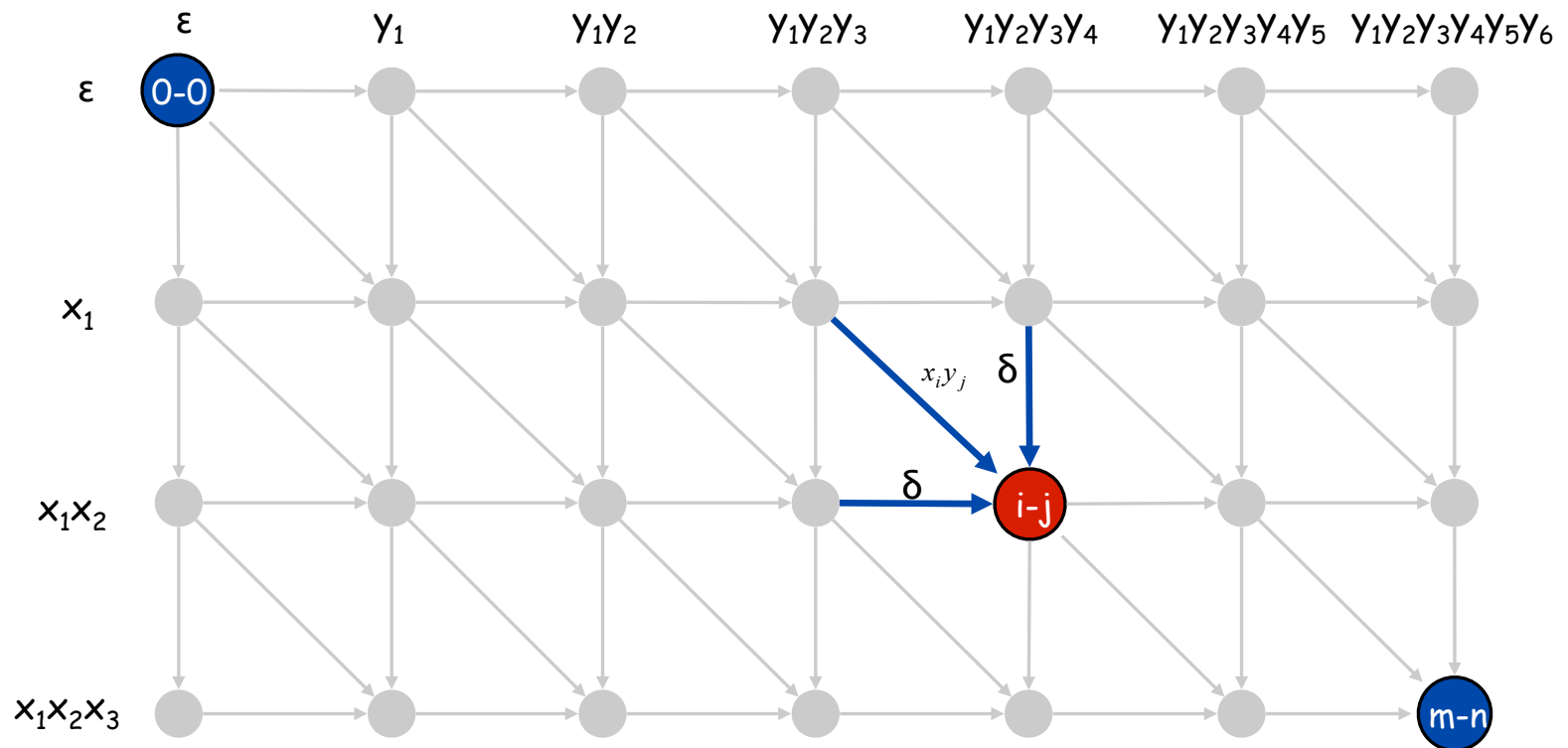Theorem. [Hirschberg 1975] Optimal alignment in O(m + n) space and O(mn) time.
- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.

6.7 Sequence Alignment in Linear Space

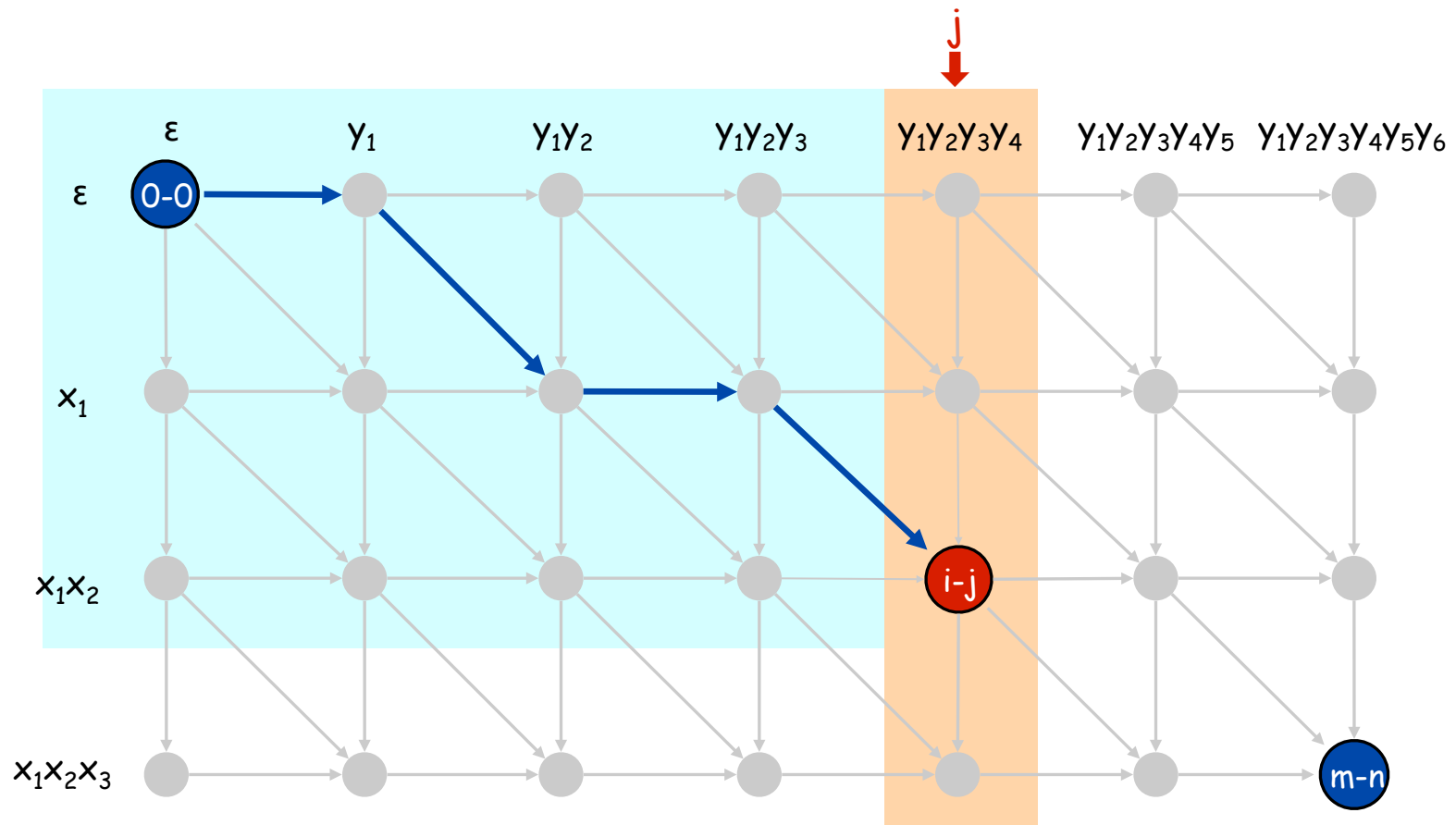# Sequence Alignment: Linear Space

Edit distance graph.
- Let f(i, j) be cost of shortest path from (0,0) to (i, j).
- Observation: f(i, j) = OPT(i, j).
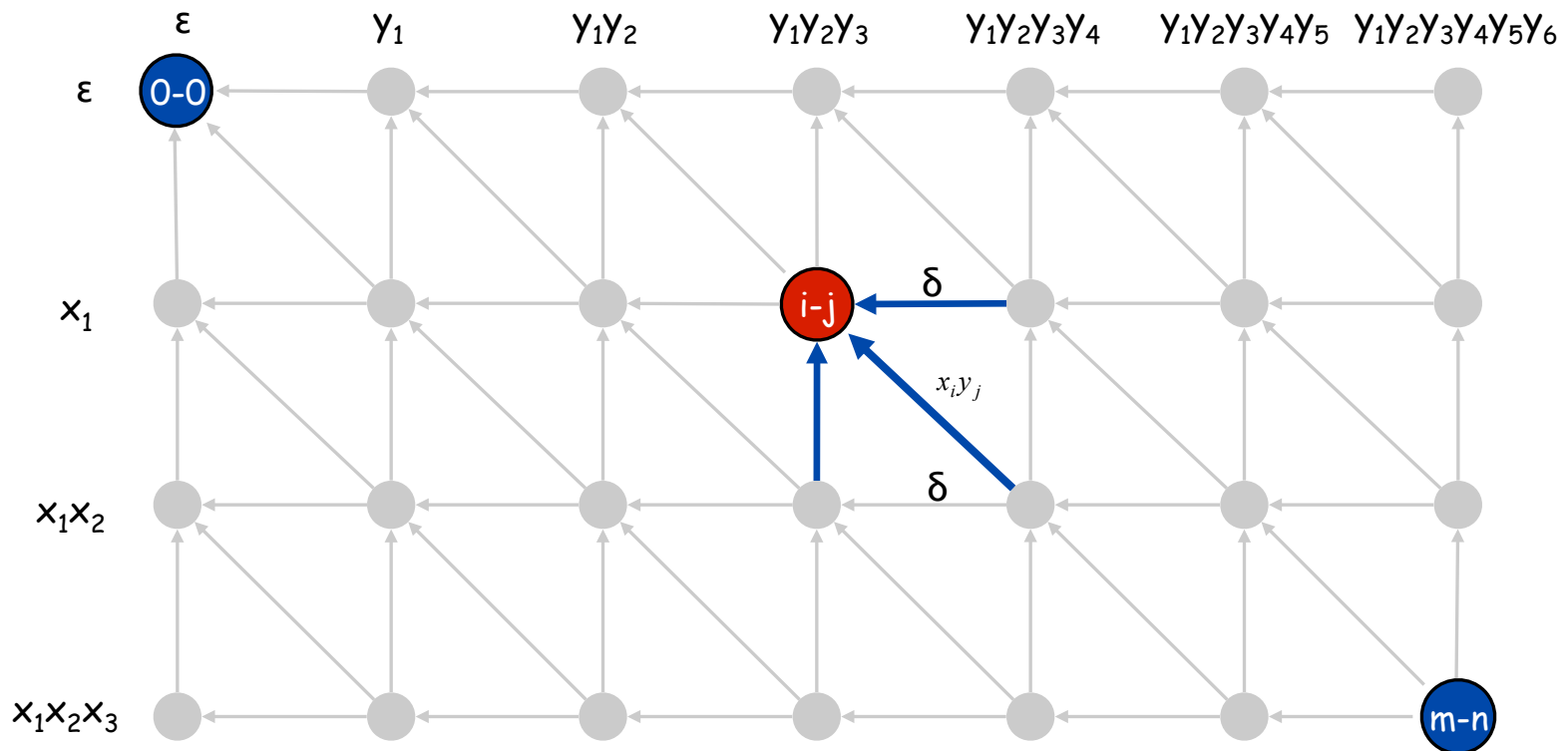
Sequence Alignment: Linear Space

Edit distance graph.

- Let f(i, j) be cost of shortest path from (0,0) to (i, j).
- Can compute f (·, j) for any j in O(mn) time and O(m + n) space.

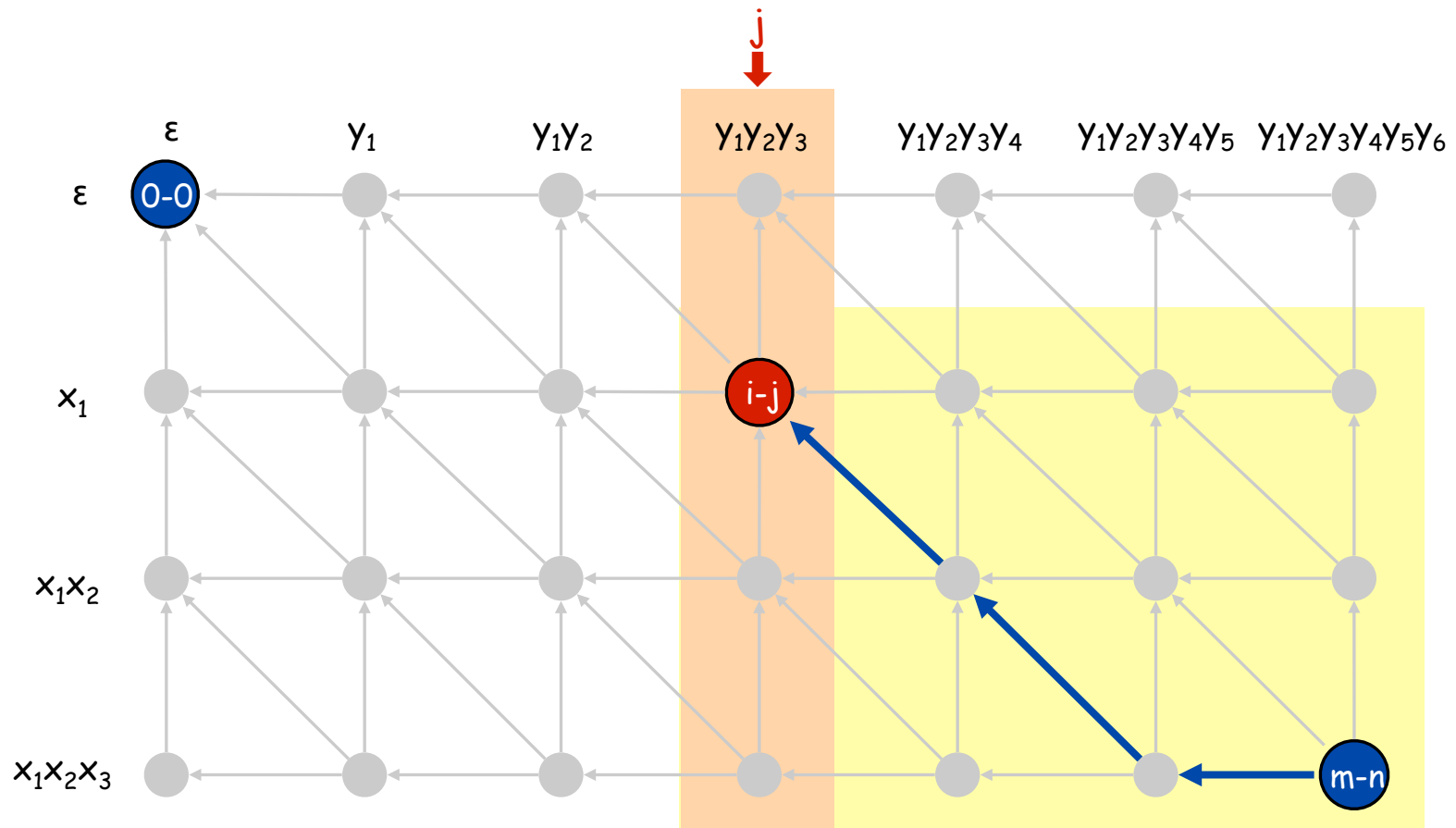# Sequence Alignment: Linear Space

Edit distance graph.

- Let $g(i, j)$ be cost of 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)

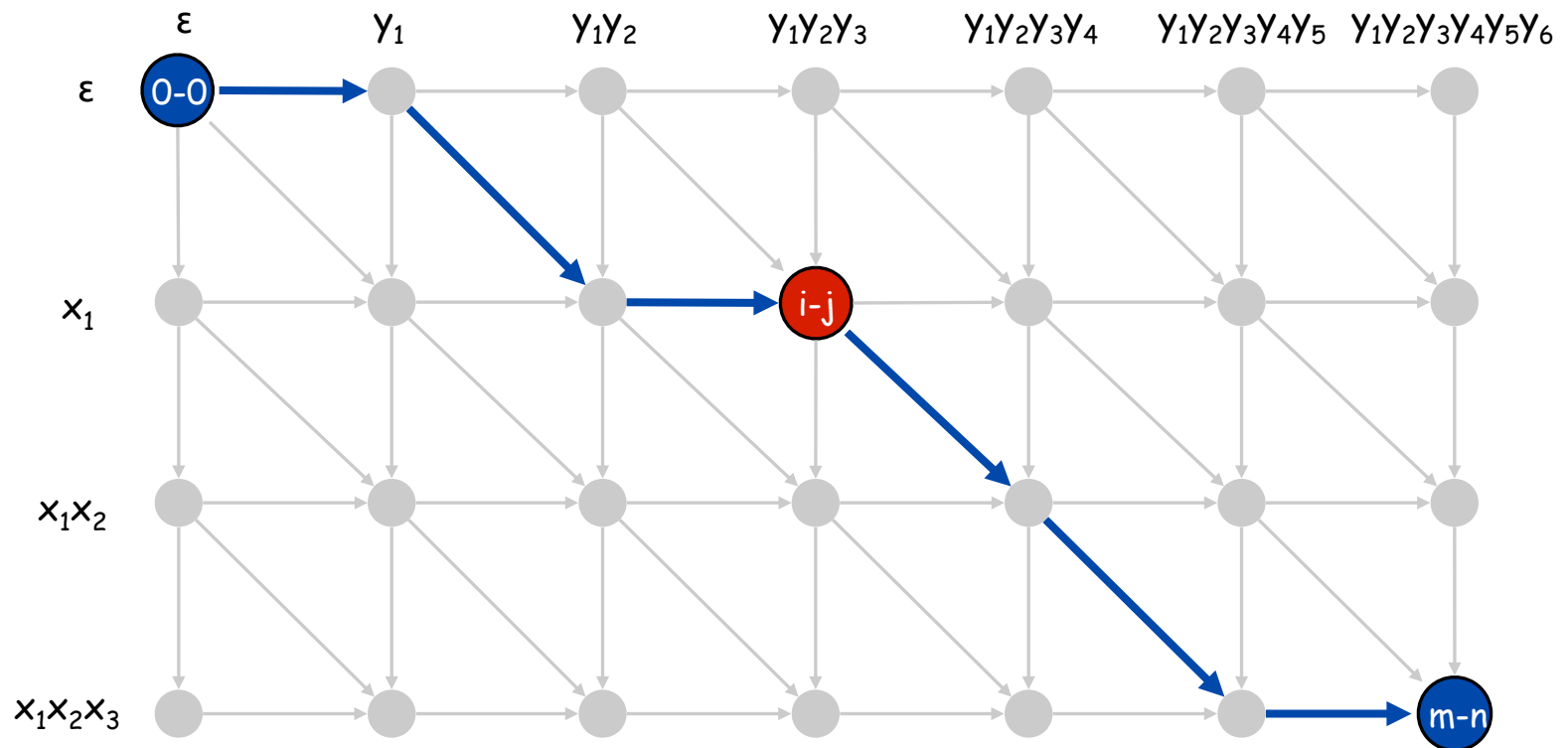# Sequence Alignment: Linear Space

Edit distance graph.

- Let $g(i, j)$ be cost of shortest path from (i, j) to (m, n).
- Can compute $g(\cdot, j)$ for any j in $O(mn)$ time and $O(m + n)$ space.
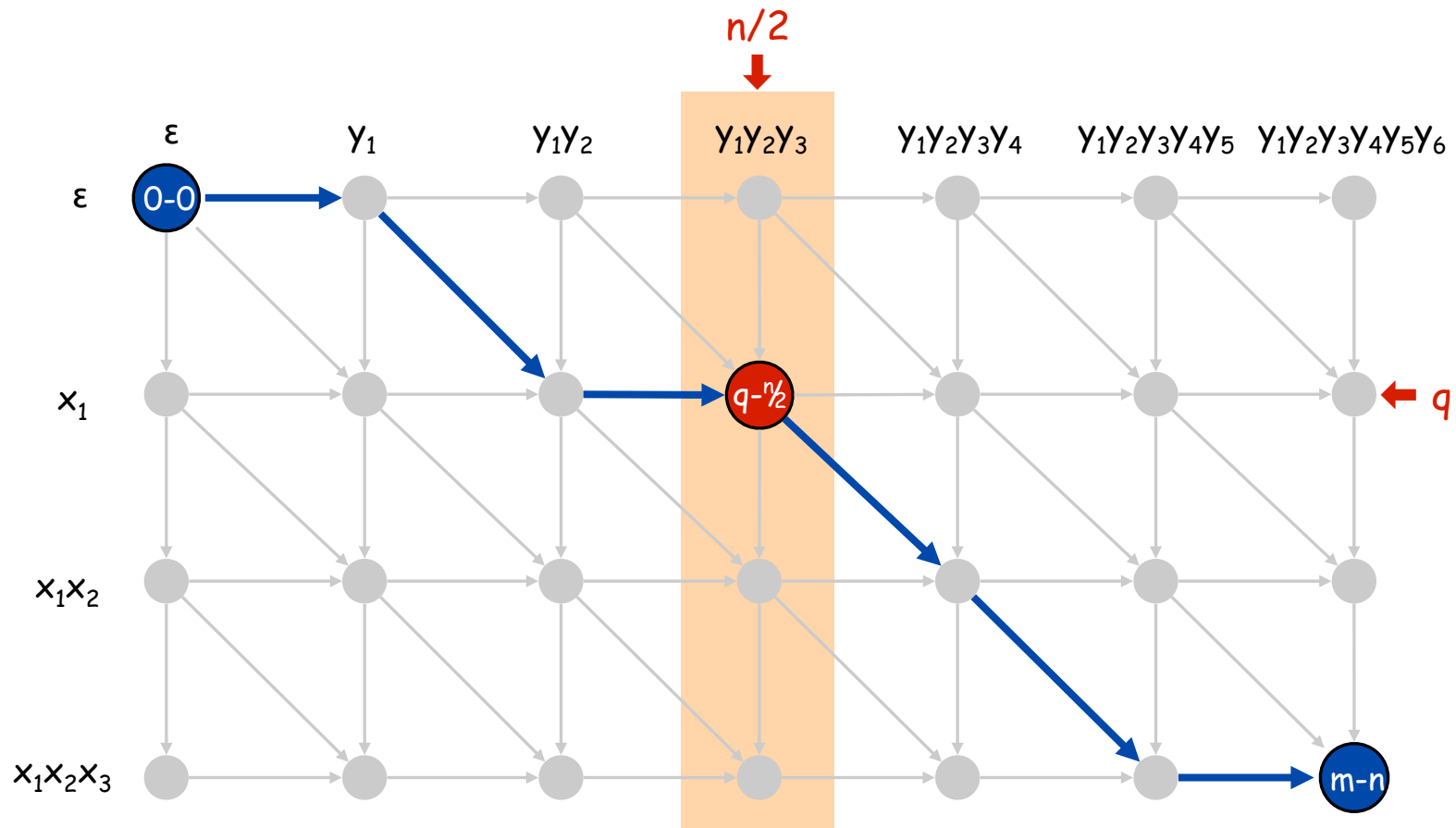
Sequence Alignment: Linear Space

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 $f(q, n/2) + g(q, n/2)$.
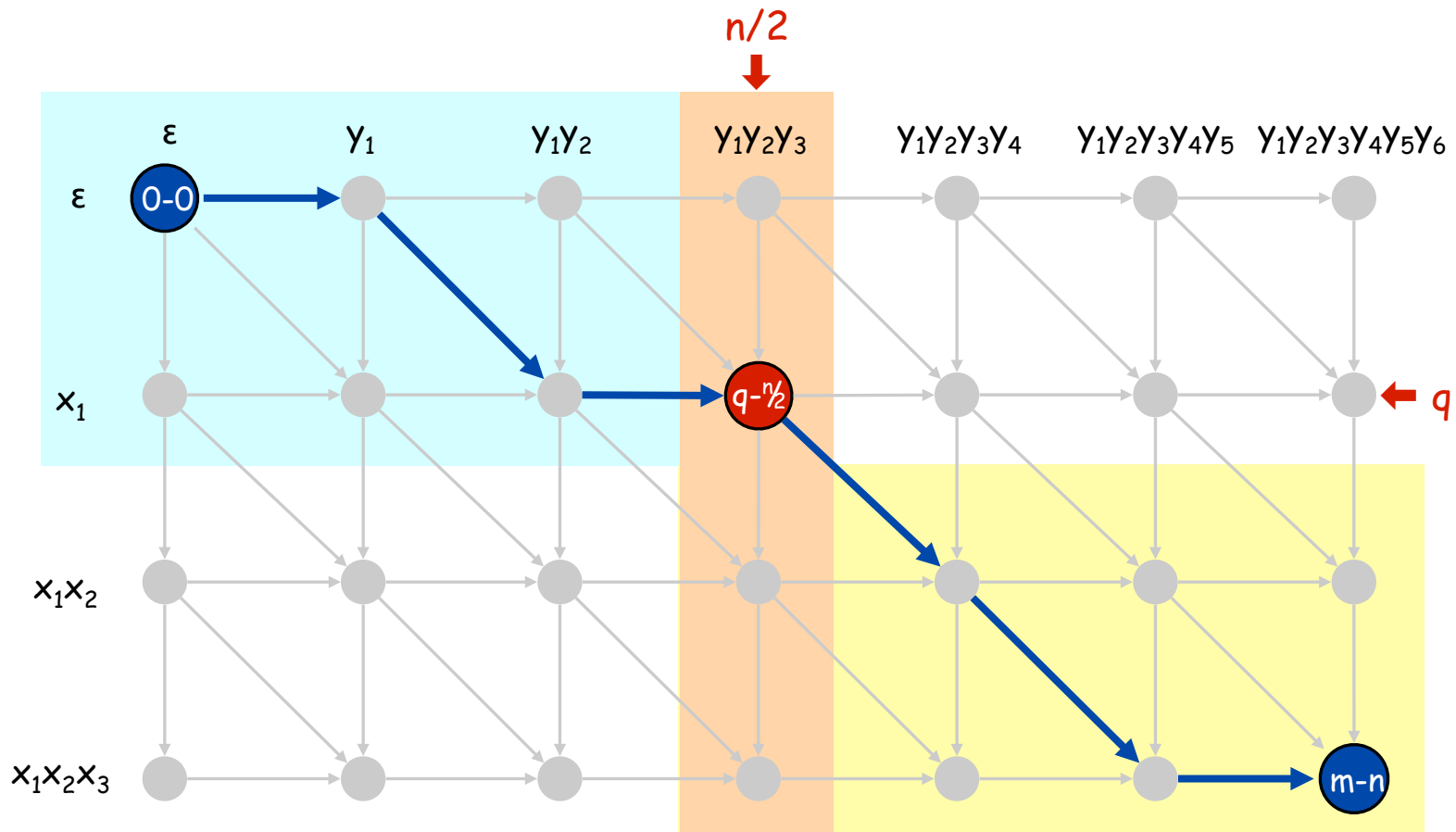Then, the shortest path from $(0, 0)$ to (m, n) uses $(q, n/2)$.

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.

Sequence Alignment: Running Time Analysis Warmup

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

$$T(m, n) \in 2T(m, n/2) + O(mn) \implies T(m, n) \in O(mn \log n)$$

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.

Sequence Alignment: Running Time Analysis

Theorem. Let T(m, n) = max running time of algorithm on strings of lengths at most m and n. T(m, n) ∈ O(mn).

Pf. (by induction on n)

- O(mn) time to compute f(•,$\frac{n}{2}$) and g (•,$\frac{n}{2}$) and find index q.
- T(q,$\frac{n}{2}$) + T(m−q,$\frac{n}{2}$) time for two recursive calls.
- Choose constant c so that:

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

- Let's prove T(m,n) ≤ 2cmn for m,n≥2.
- Base cases: m = 2 or n = 2: T(2,n) ≤ cn ≤ 2cmn, T(m,2) ≤ cm ≤ 2cmn.
- Inductive hypothesis: T(i, j) ≤ 2cij for 2≤i≤m, 2≤j≤n, i+j<m+n.

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

Dynamic Programming Summary

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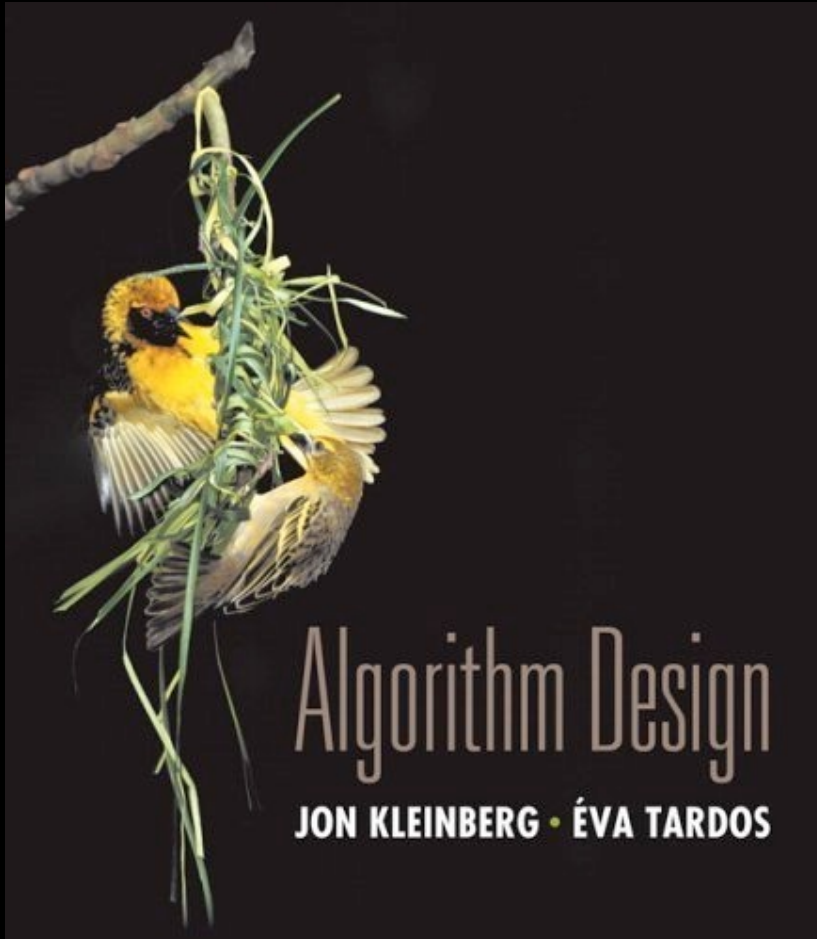- Construct optimal solution from computed information.

Dynamic programming techniques.
- Binary choice: weighted interval scheduling.
- Multi-way choice: segmented least squares. —— Viterbi algorithm for HMM also uses DP to optimize a maximum likelihood tradeoff between parsimony and accuracy
- Adding a new variable: knapsack.
- Dynamic programming over intervals: RNA secondary structure.

CKY parsing algorithm for context-free grammar has similar structure

Top-down vs. bottom-up: different people have different intuitions.

Chapter 6

Dynamic Programming