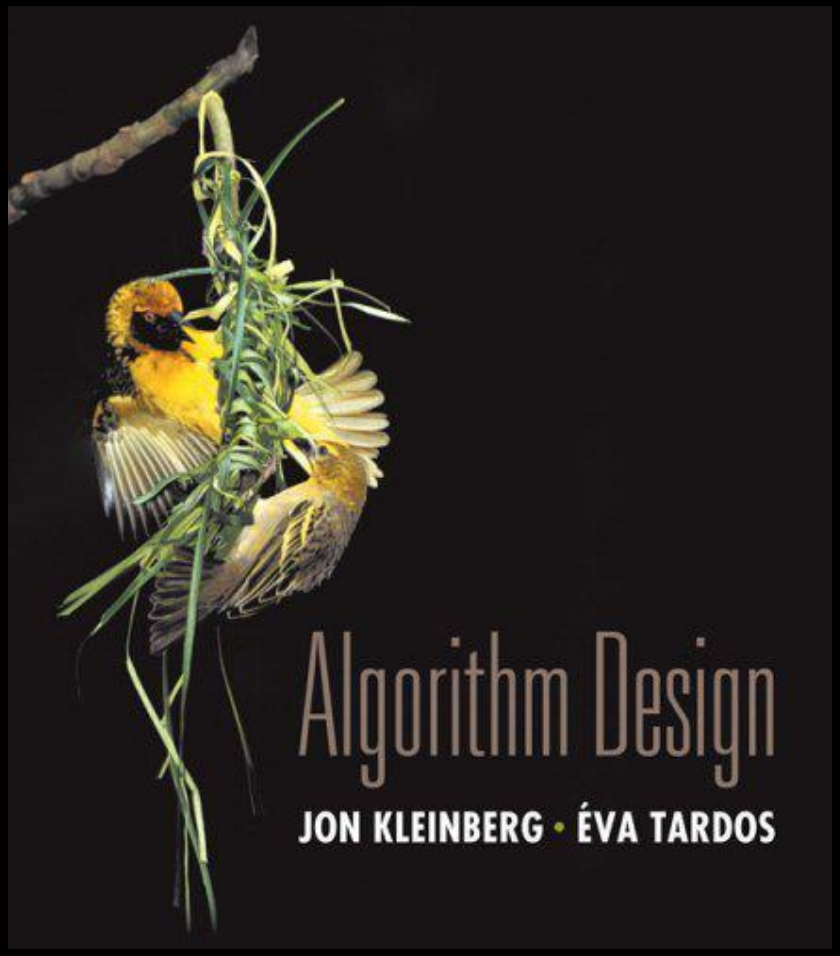


Chapter 6

Dynamic Programming

Algorithmic Paradigms

Greed. 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.

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

Dynamic Programming Applications

Areas.

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

Some famous dynamic programming algorithms.

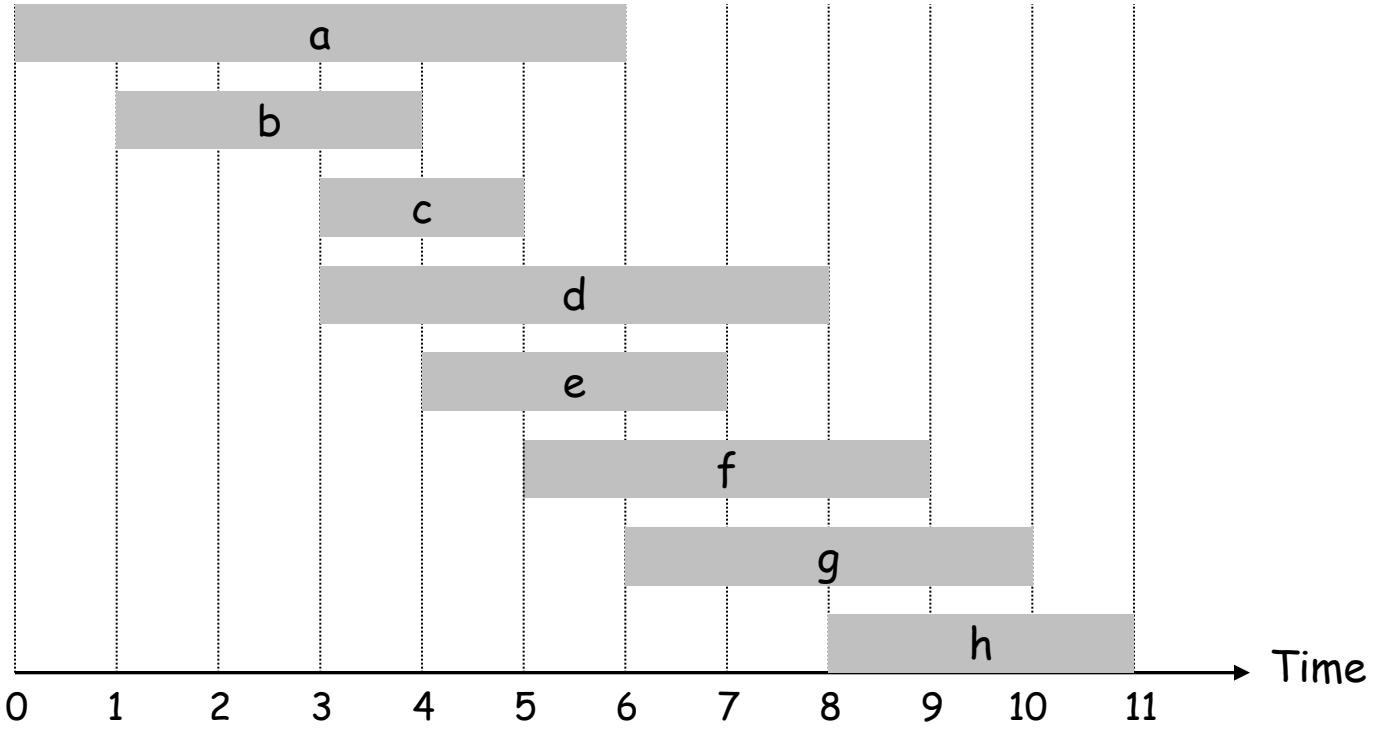
- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- Smith-Waterman for sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

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.

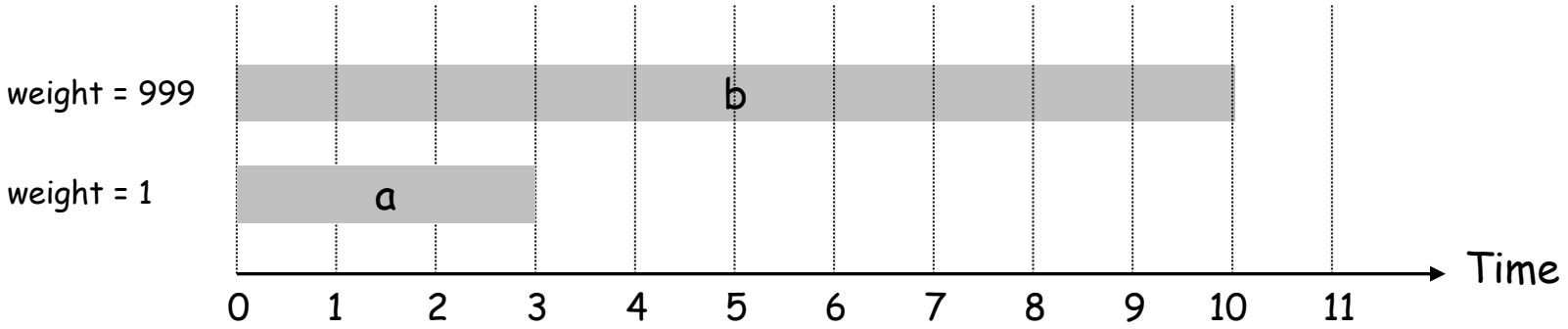


Unweighted Interval Scheduling Review

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.

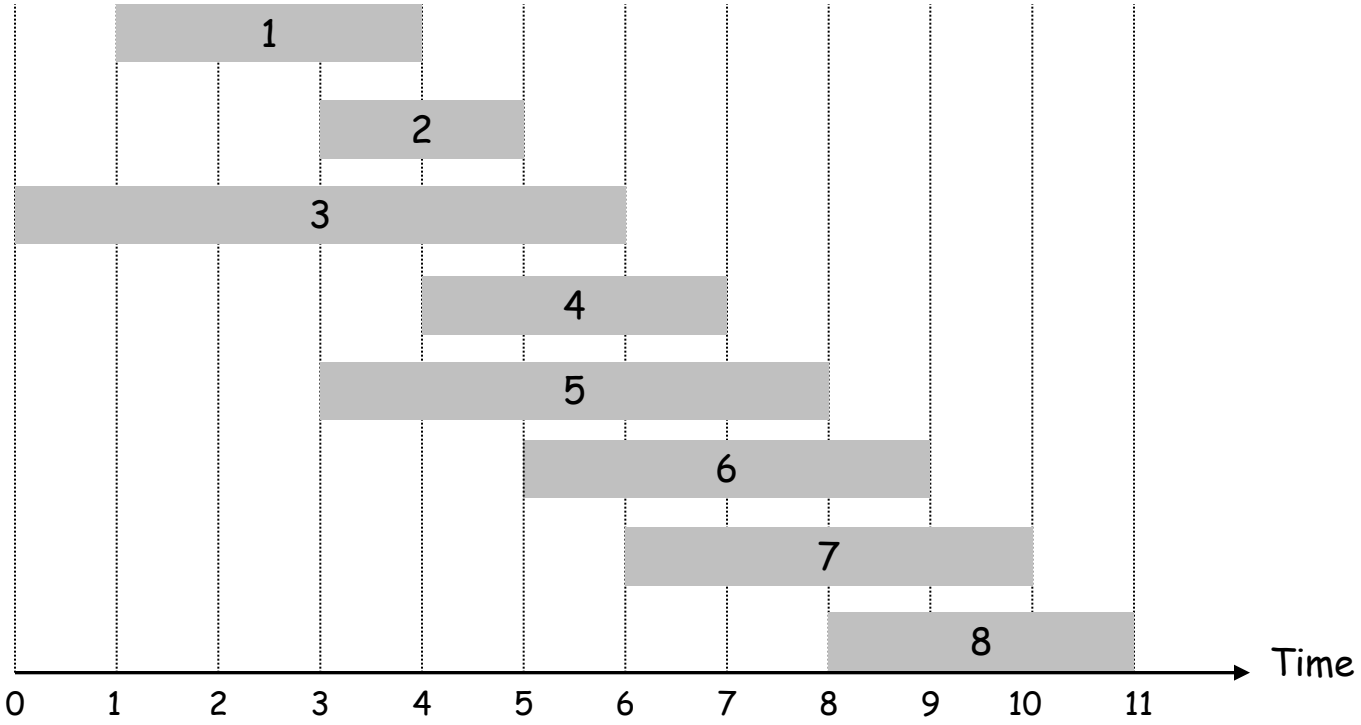


Weighted Interval Scheduling

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

INPUT: n, $s[1..n]$, $f[1..n]$, $v[1..n]$.

Sort jobs by finish time so that $f[1] \leq f[2] \leq \cdots \leq f[n]$.

Compute $p[1], p[2], \cdots, p[n]$.

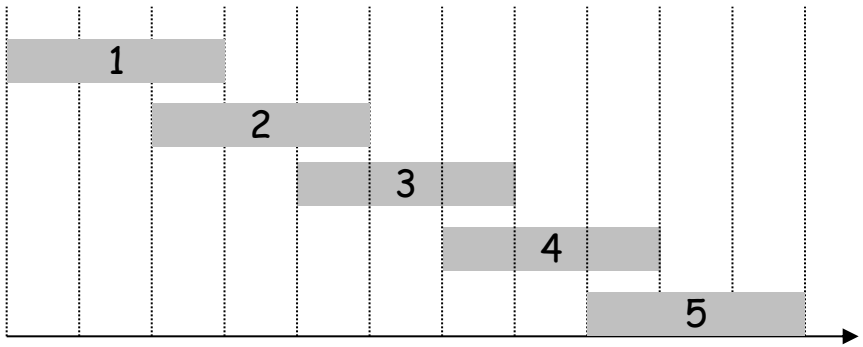
Compute – Opt(j)

```
1: if j = 0 then
2:     return 0.
3: else
4:     return max{v[j]+Compute−Opt(p[j]), Compute−Opt(j−
       1)}.
5: end if
```

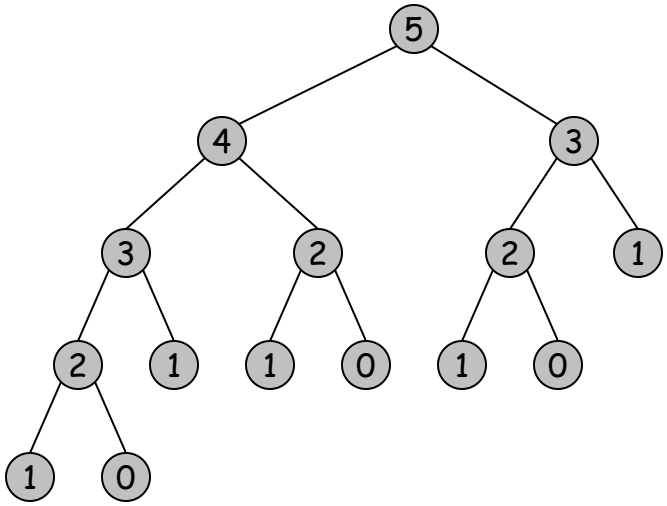
Weighted Interval Scheduling: Brute Force

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: Memoization

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

INPUT: n, $s[1..n]$, $f[1..n]$, $v[1..n]$.

Sort jobs by finish time so that $f[1] \leq f[2] \leq \cdots \leq f[n]$.

Compute $p[1], p[2], \cdots, p[n]$.

```
1: for j = 1 to n do
2:     M[j] ← empty.
3: end for
4: return M[0] ← 0.
```

M – Compute – Opt(j)

```
1: if M[j] is empty then
2:     M[j] ← max{v[j]+M–Compute–Opt(p[j]), M–Compute–
       Opt(j – 1)}.
3: end if
4: 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 $p(\cdot)$: $O(n \log n)$ via binary search.

- `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 $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases Φ by 1 $\Rightarrow$ at most $2n$ recursive calls.

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

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.

Find – Solution(j)

```
1: if j = 0 then
2:     return ∅.
3: else if v[j] + M[p[j]] > M[j − 1] then
4:     return {j} ∪ Find – Solution(p[j]).
5: else
6:     return Find – Solution(j − 1).
7: end if
```

- # of recursive calls ≤ n ⇒ O(n).

Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

$BOTTOM - UP(n, s_1, \cdots, s_n, f_1, \cdots, f_n, v_1, \cdots, v_n)$

1: Sort jobs by finish time so that $f_1 \leq f_2 \leq \cdots \leq f_n$.
2: Compute $p[1], p[2], \cdots, p[n]$.
3: $M[0] \leftarrow 0$.
4: **for** $j = 1$ to n **do**
5: $\quad M[j] \leftarrow \max\{v_j + M[p(j)], M[j-1]\}$.
6: **end for**

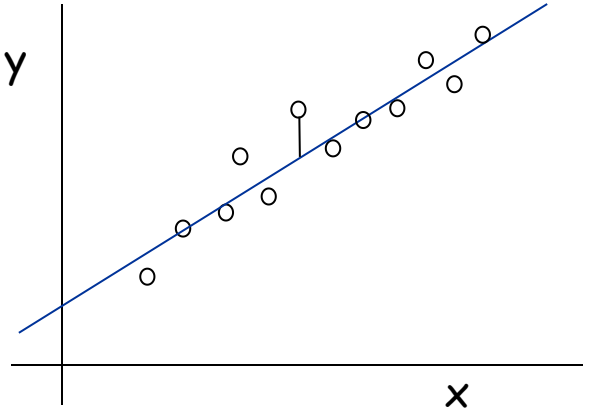
6.3 Segmented Least Squares

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

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

Segmented Least Squares

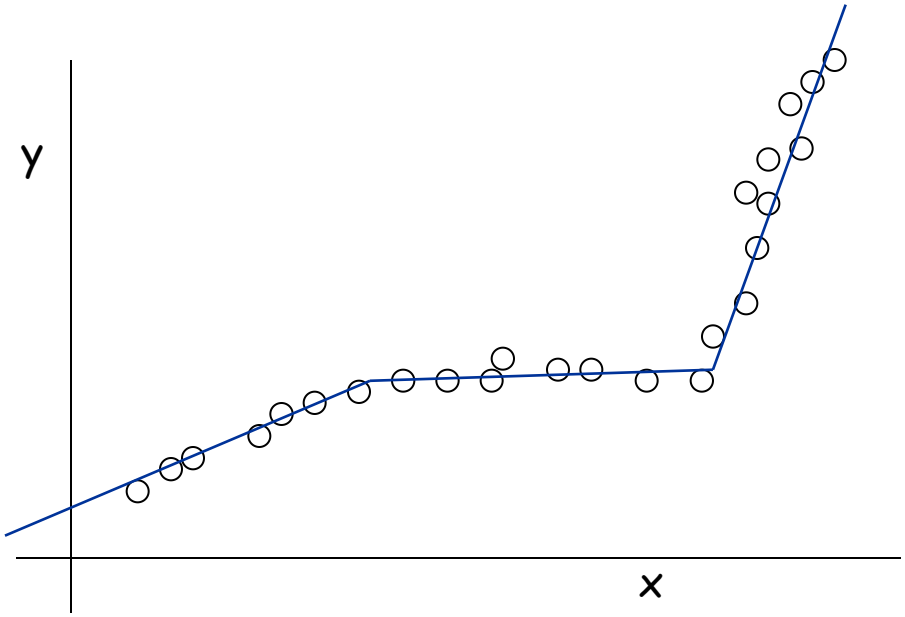
Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with
- $x_1 < x_2 < \ldots < x_n$, 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

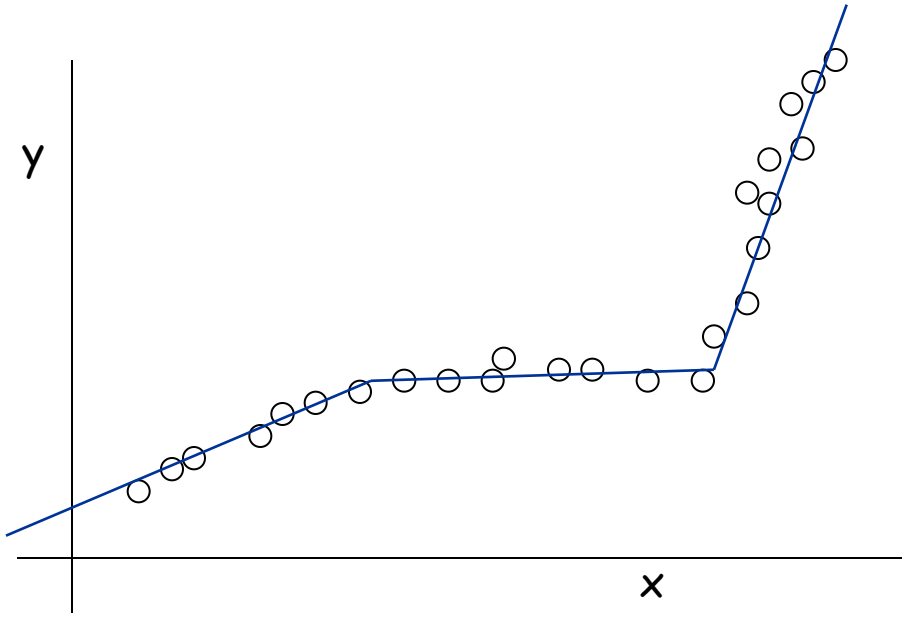
number of lines



Segmented Least Squares

Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with
- $x_1 < x_2 < \ldots < x_n$, 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).

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

Segmented Least Squares: Algorithm

$SEGMENTED - LEAST - SQUARES(n, p_1, \cdots, p_n, c)$

```
1: for j = 1 to n do
2:     for i = 1 to n do
3:         Compute the least squares e(i, j) for the segment
           p_i, p_{i+1}, ..., p_j.
4:     end for
5: end for
6: M[0] ← 0.
7: for j = 1 to n do
8:     M[j] ← min_{1≤i≤j}{e_ij + c + M[i − 1]}.
9: end for
10: return M[n].
```

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

- Bottleneck = computing e(i, j) for $O(n^2)$ pairs, $O(n)$ per pair using previous formula.

6.4 Knapsack Problem

Knapsack Problem

Knapsack problem.

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ and has value $v_i > 0$.
- Knapsack has capacity of W.
- Goal: fill knapsack so as to maximize total value.

Ex: { 3, 4 } has value 40.

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Greedy: repeatedly add item with maximum ratio v_i / w_i.

Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

Dynamic Programming: Adding a New Variable

Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w

- Case 2: OPT selects item i.
  - new weight limit = $w - w_i$
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1, w-w_i)\,\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

$KNAPSACK(n, W, w_1, \cdots, w_n, v_1, \cdots, v_n)$

```
1: for w = 0 to W do
2:     M[0, w] ← 0.
3: end for
4: for i = 1 to n do
5:     for w = 1 to W do
6:         if w_i > w then
7:             M[i, w] ← M[i − 1, w].
8:         else
9:             M[i, w] ← max{M[i − 1, w], v_i + M[i − 1, w − w_i]}.
10:        end if
11:    end for
12: end for
13: return M[n, W].
```

Knapsack Algorithm

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Problem: Running Time

Running time. $\Theta(n\ W)$.

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete. [Chapter 8]

Knapsack approximation algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]

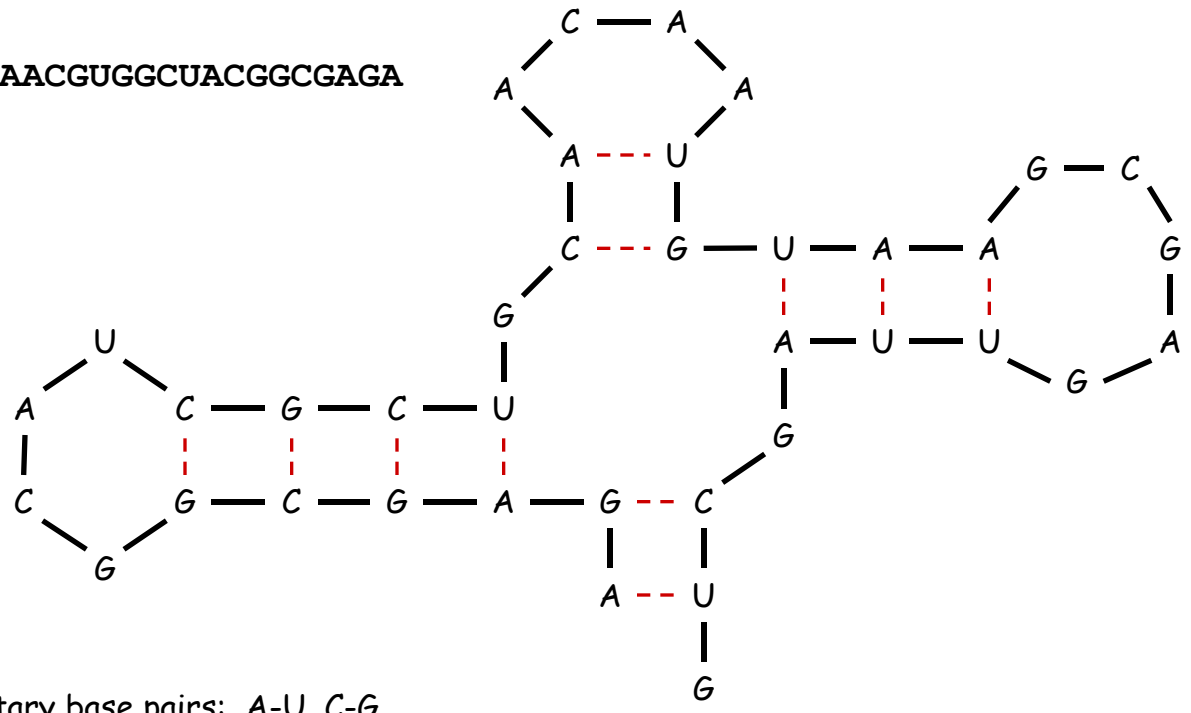
6.5 RNA Secondary Structure

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 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 $i < k < j < l$.

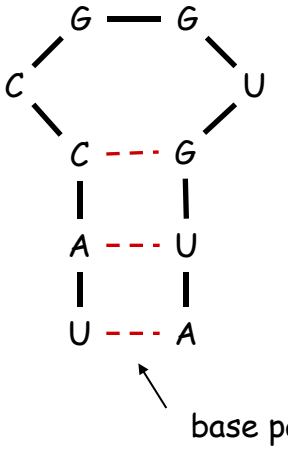
Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.

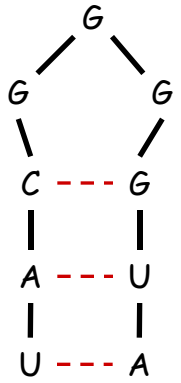
approximate by number of base pairs

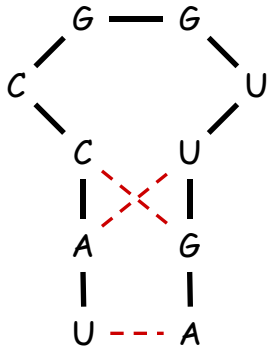
Goal. Given an RNA molecule $B = b_1 b_2 \ldots b_n$, find a secondary structure S that maximizes the number of base pairs.

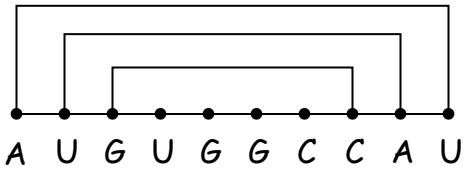
RNA Secondary Structure: Examples

Examples.

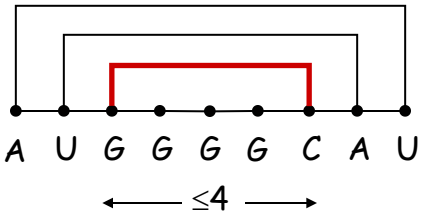




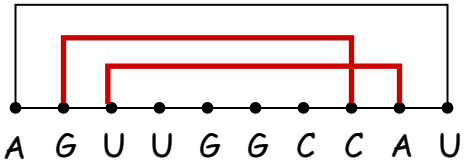




ok



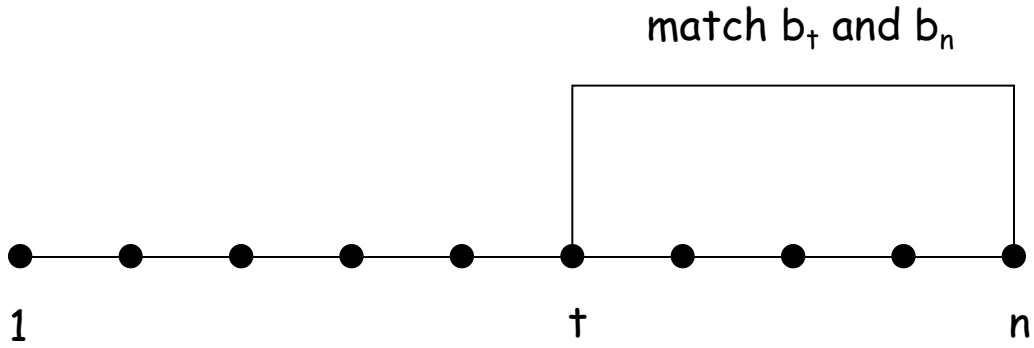
sharp turn



crossing

RNA Secondary Structure: Subproblems

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



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 $b_i b_{i+1} \ldots b_j$.

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

take max over t such that $i \leq t < j-4$ and b_t and b_j are Watson-Crick complements

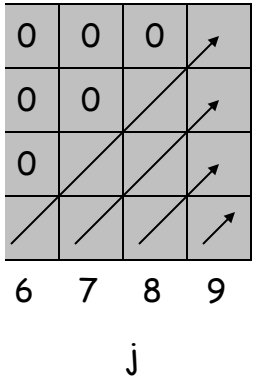
Bottom Up Dynamic Programming Over Intervals

Q. What order to solve the sub-problems?

A. Do shortest intervals first.

$RNA(n, b_1, \cdots, b_n)$

```
1: for k = 5 to n − 1 do
2:     for i = 1 to n − k do
3:         j ← i + k.
4:         Compute M[i, j] using formula.
5:     end for
6: end for
7: return M[1, n].
```



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

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.
- Adding a new variable: knapsack.
- Dynamic programming over intervals: RNA secondary structure.

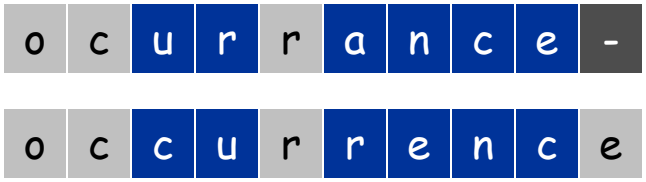
Top-down vs. bottom-up: different people have different intuitions.

6.6 Sequence Alignment

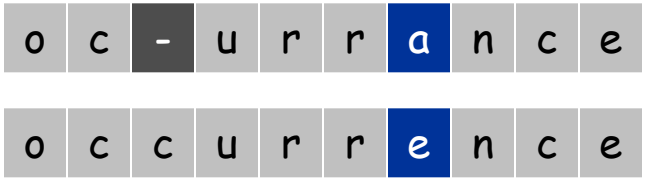
String Similarity

How similar are two strings?

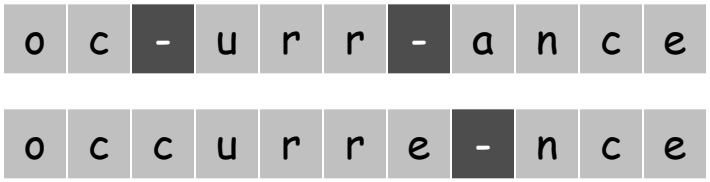
- ocurrance
- occurrence



5 mismatches, 1 gap



1 mismatch, 1 gap



0 mismatches, 3 gaps

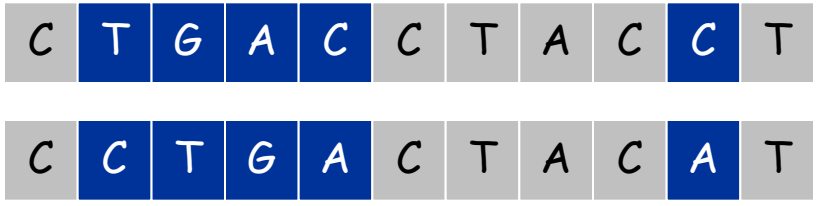
Edit Distance

Applications.

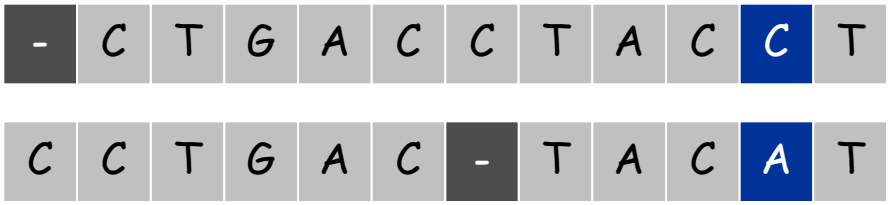
- Basis for Unix diff.
- Speech recognition.
- Computational biology.

Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty δ; mismatch penalty α_{pq}.
- Cost = sum of gap and mismatch penalties.



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



$2\delta + \alpha_{CA}$

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.

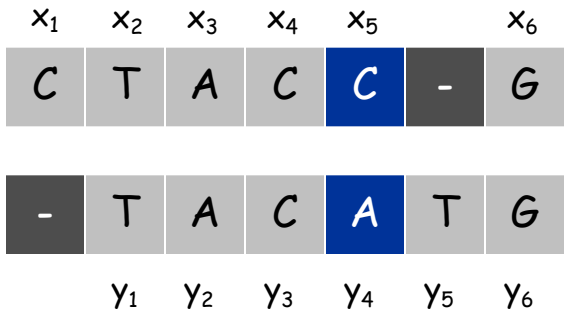
Def. An alignment M is a set of ordered pairs x_i-y_j such that each item occurs in at most one pair and no crossings.

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

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

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.



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

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

Sequence Alignment: Algorithm

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

```
1:  for i = 0 to m do
2:      M[i, 0] ← iδ.
3:  end for
4:  for j = 0 to m do
5:      M[0, j] ← jδ.
6:  end for
7:  for i = 1 to m do
8:      for j = 1 to n do
9:          M[i, j] ← min{α[x_i, y_j] + M[i − 1, j − 1], δ + M[i − 1, j], δ +
            M[i, j − 1]}.
10:     end for
11: end for
12: return M[m, n].
```

Analysis. $\Theta(mn)$ time and space.

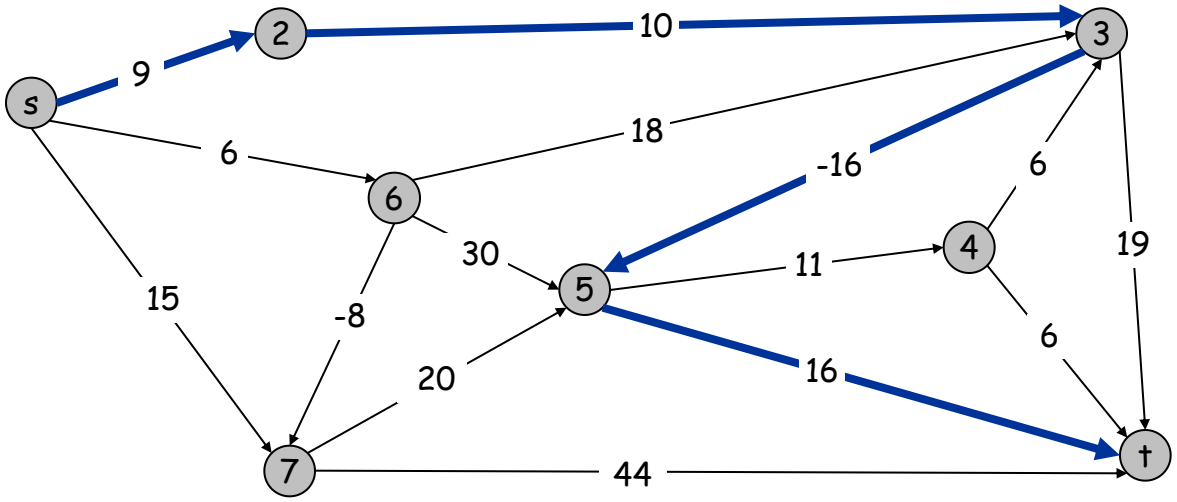
6.7 Shortest Paths

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.

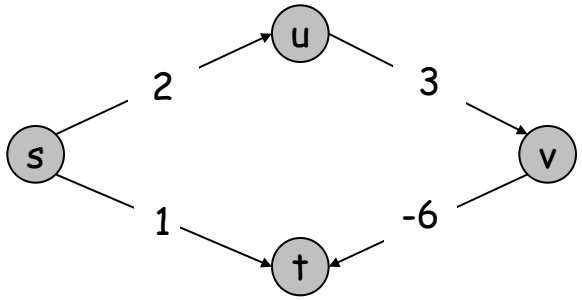
↖ allow negative weights

Ex.

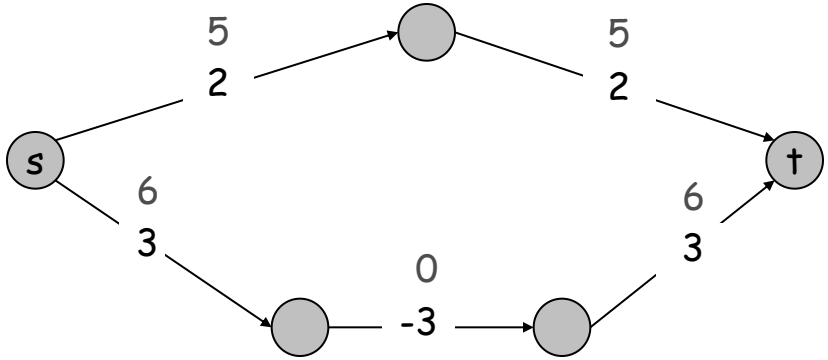


Shortest Paths: Failed Attempts

Dijkstra. Can fail if negative edge costs.

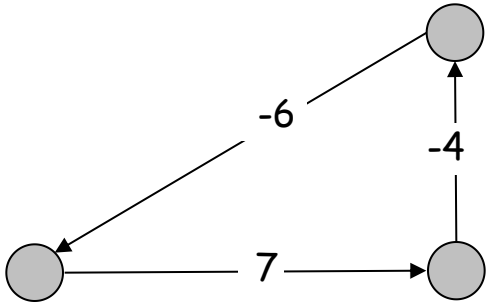


Re-weighting. Adding a constant to every edge weight can fail.

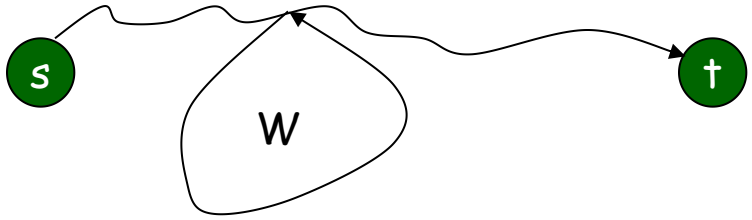


Shortest Paths: Negative Cost Cycles

Negative cost cycle.



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.



$c(W) < 0$

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

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

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

Shortest Paths: Implementation

SHORTEST – PATH(V, E, c, s, t)

```
 1: for node v ∈ V do
 2:     M[0, v] ← ∞.
 3: end for
 4: M[0, t] ← 0.
 5: for i = 1 to n − 1 do
 6:     for node v ∈ V do
 7:         M[i, v] ← M[i − 1, v].
 8:         for edge (v, w) ∈ E do
 9:             M[i, v] ← min{M[i, v], M[i − 1, w] + c_vw}.
10:         end for
11:     end for
12: end for
13: return M[n − 1, s].
```

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

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