CSE 421

Dynamic Programming /

Knapsack Problem, RNA, Sequence Alignment

Yin Tat Lee

Knapsack Problem

Knapsack Problem

Given n objects and a "knapsack."

Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.

Knapsack has capacity of W kilograms.

Goal: fill knapsack so as to maximize total value.

Ex: OPT is { 3, 4 } with 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.

Stronger DP (Strengthening Hypothesis)

Let $OPT(i, w)$ = Max value of subsets of items $1, \ldots, i$ of weight $\leq w$

Case 1: $OPT(i, w)$ selects item i

- In this case, $OPT(i, w) = v_i + OPT(i - 1, w - w_i)$

Case 2: $OPT(i, w)$ does not select item i

- In this case, $OPT(i, w) = OPT(i - 1, w)$.

Take best of the two

Therefore,

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

DP for Knapsack

```
Compute-OPT(i,w)
   if M[i,w] == empty
     if (i==0)
       M[i,w]=0
     else if (w_i > w)
       M[i,w]=Comp-OPT(i-1,w)
     else
       M[i,w]= max {Comp-OPT(i-1,w), v_i + Comp-OPT(i-1,w-w_i)}
   return M[i, w]
```

recursive

```
for w = 0 to W
   M[0, w] = 0
for i = 1 to n
   for w = 1 to W
      if (w_i > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}

return M[n, W]
```

Non-recursive

DP for Knapsack

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, 2 }	0											
{ 1, 2, 3 }	0											
{ 1, 2, 3, 4 }	0											
{ 1, 2, 3, 4, 5 }	0											

W = 11

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

```
if (w_i > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}
```

DP for Knapsack

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, 2, 3 }	0											
{ 1, 2, 3, 4 }	0											
{ 1, 2, 3, 4, 5 }	0											

W = 11

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

```
if (w_i > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}
```

DP for Knapsack

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								
{ 1, 2, 3 }	0	1										
{ 1, 2, 3, 4 }	0	1										
{ 1, 2, 3, 4, 5 }	0	1										

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

```
if (w_i > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}
```

DP for Knapsack

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					
{ 1, 2, 3, 4 }	0	1										
{ 1, 2, 3, 4, 5 }	0	1										

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

```
if (w_i > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}
```

DP for Knapsack

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		
{ 1, 2, 3, 4, 5 }	0	1										

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

```
if (w_i > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}
```

DP for Knapsack

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

```
if (w_i > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}
```

Knapsack Problem: Running Time

Running time: $\Theta(n \cdot W)$

- Not polynomial in input size!
- "Pseudo-polynomial.“
- Decision version of Knapsack is NP-complete.

Knapsack approximation algorithm:

There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum in time Poly(n,$\log W$).

UW Expert in similar problems

DP Ideas so far

- You may have to define an ordering to decrease #subproblems
- You may have to strengthen DP, equivalently the induction, i.e., you have may have to carry more information to find the Optimum.
- This means that sometimes we may have to use two dimensional or three dimensional induction

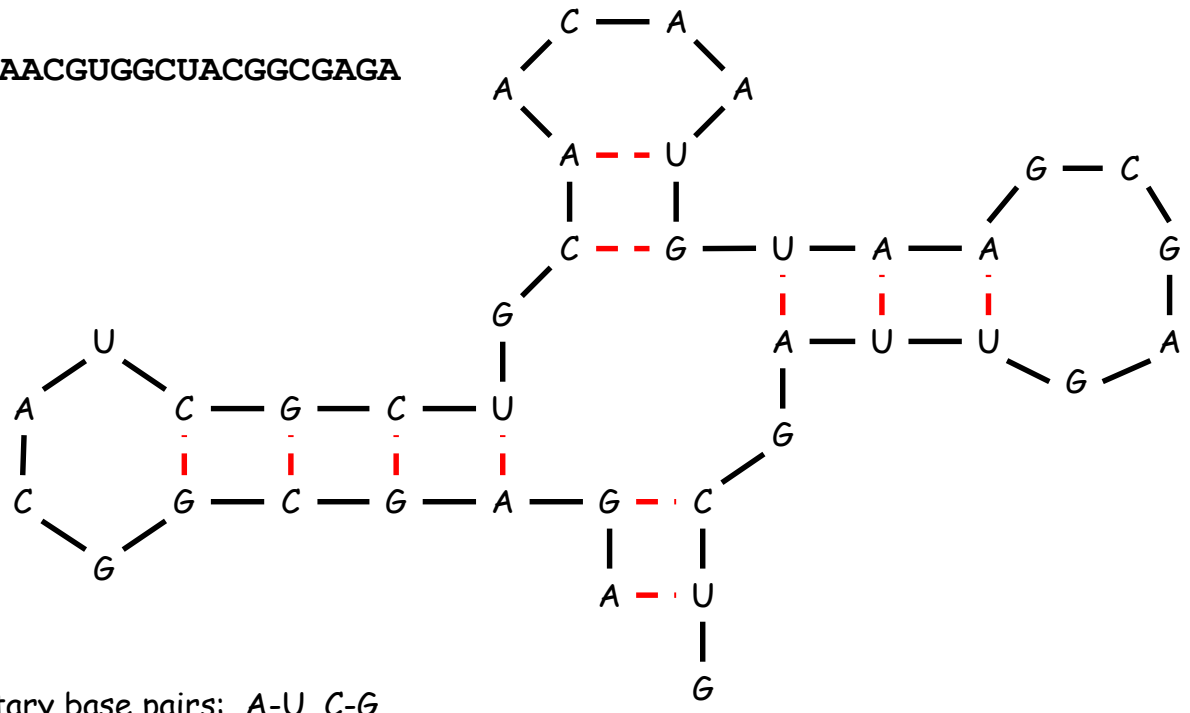
RNA Secondary Structure

RNA Secondary Structure

RNA: A 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 (Formal)

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 pair: $A - U$, $U - A$, $C - G$, 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 $i < j - 4$.

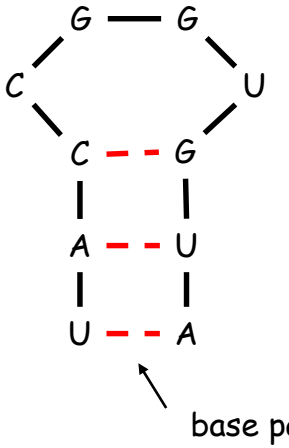
[Non-crossing.] If (b_i, b_j) and (b_k, b_l) are two pairs in S, then we cannot have $i < k < j < l$.

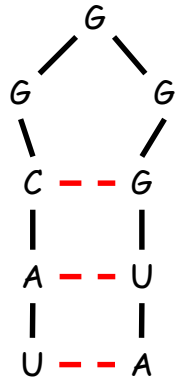
Free energy: Usual hypothesis is that an RNA molecule will maximize total free energy.

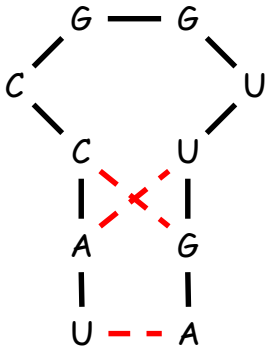
approximate by number of base pairs

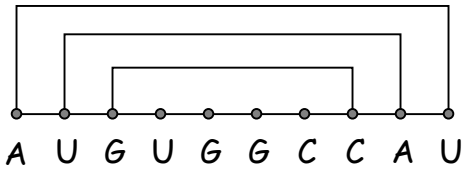
Goal: Given an RNA molecule B = $b_1 b_2 \ldots b_n$, find a secondary structure S that maximizes the number of base pairs.

Secondary Structure (Examples)

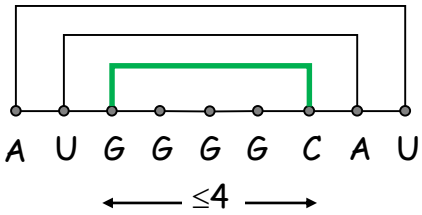




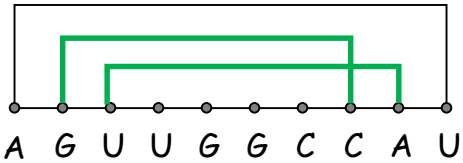




ok



sharp turn



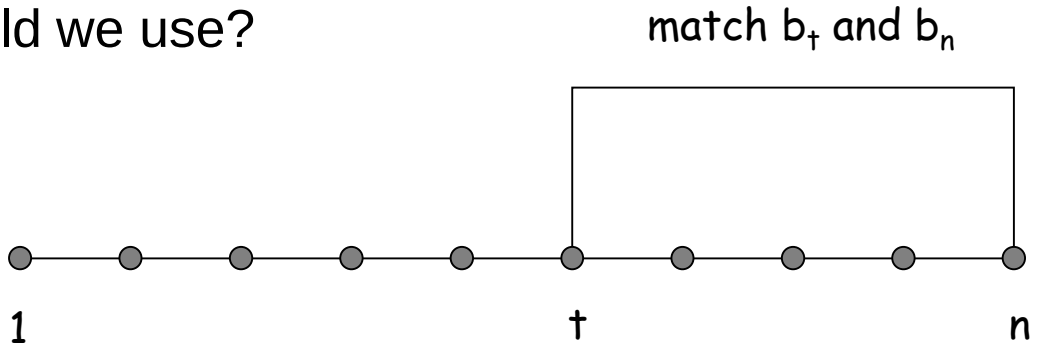
crossing

DP: First Attempt

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

Suppose b_n is matched with b_t in $OPT(n)$.

What IH should we use?



Difficulty: This naturally reduces to two subproblems

- Finding secondary structure in $b_1, \ldots, b_{t-1}$, i.e., OPT(t-1)
- Finding secondary structure in $b_{t+1}, \ldots, b_n$, ???

DP: Second Attempt

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

The most important part of a correct DP; It fixes IH

Case 1: If $j - i \leq 4$.

- $OPT(i,j) = 0$ 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
- $OPT(i,j) = \max_{i \leq t < j-4} \{ 1 + OPT(i,t-1) + OPT(t+1,j-1) \}$

Recursive Code

```
Let M[i,j]=empty for all i,j.

Compute-OPT(i,j){
   if (j-i <= 4)
     return 0;
   if (M[i,j] is empty)
      M[i,j]=Compute-OPT(i,j-1)
      for t=i to j-5 do
         if (b_t,b_j is in {A-U, U-A, C-G, G-C})
           M[i,j]=max(M[i,j], 1+Compute-OPT(i,t-1) +
                      Compute-OPT(t+1,j-1))
   return M[j]
}
```

Does this code terminate?
What are we inducting on?
Key question: is there any loop in the recursion?

Formal Induction

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

Base Case: $OPT(i,j) = 0$ for all i,j where $|j-i| \leq 4$.

IH: For some $\ell \geq 4$, Suppose we have computed $OPT(i,j)$ for all i,j where $|i-j| \leq \ell$.

IS: Goal: We find $OPT(i,j)$ for all i,j where $|i-j| = \ell+1$. Fix i,j such that $|i-j| = \ell+1$.

Case 1: Base b_j is not involved in a pair.

- $OPT(i,j) = OPT(i,j-1)$ [this we know by IH since $|i-(j-1)| = \ell$]

Case 2: Base b_j pairs with b_t for some $i \leq t < j-4$

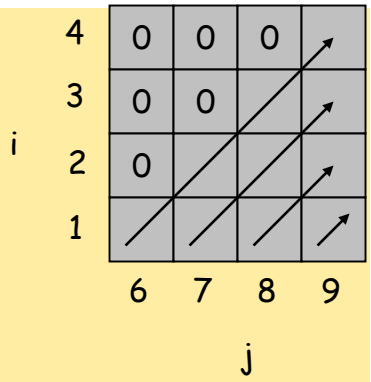
- $OPT(i,j) = \max_{i \leq t < j-4} \{ 1 + OPT(i,t-1) + OPT(t+1,j-1) \}$

We know by IH since difference $\leq \ell$

Bottom-up DP

```
for ℓ = 1, 2, …, n-1
  for i = 1, 2, …, n-1
    j = i + ℓ
    if (ℓ <= 4)
      M[i,j]=0;
      else
        M[i,j]=M[i,j-1]
        for t=i to j-5 do
          if (b_t, b_j is in {A-U, U-A, C-G, G-C})
            M[i,j]=max(M[i,j], 1+ M[i,t-1] + M[t+1,j-1])

  return M[1, n]
}
```



Running Time: $O(n^3)$

Lesson

We may not always induct on i or w to get to smaller subproblems.

We may have to induct on $|i - j|$ or $i + j$ when we are dealing with more complex problems.

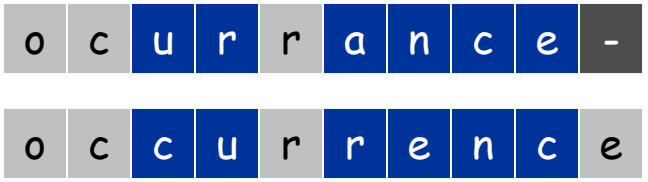
Sequence Alignment

Word Alignment

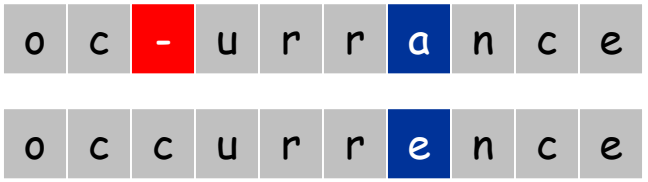
How similar are two strings?

ocurrance

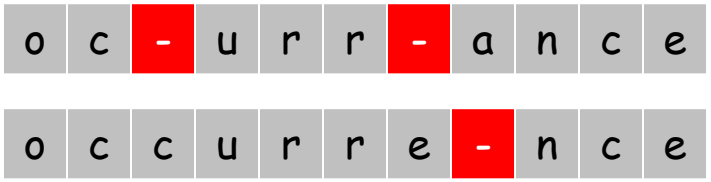
occurrence



5 mismatches, 1 gap



1 mismatch, 1 gap



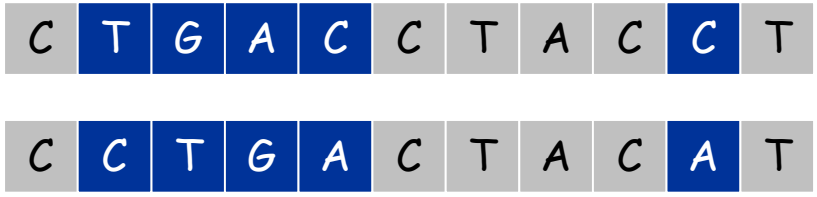
0 mismatches, 3 gaps

Edit Distance

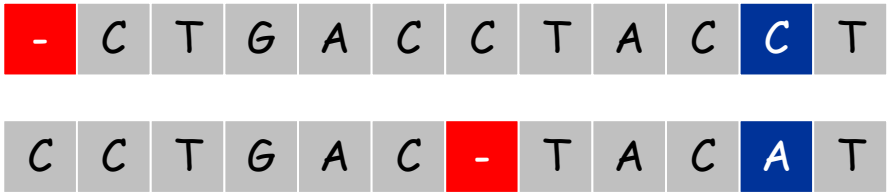
Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]
Cost = # of gaps + #mismatches.

Applications.

- Basis for Unix diff and Word correct in editors.
- Speech recognition.
- Computational biology.



Cost: 5



Cost: 3

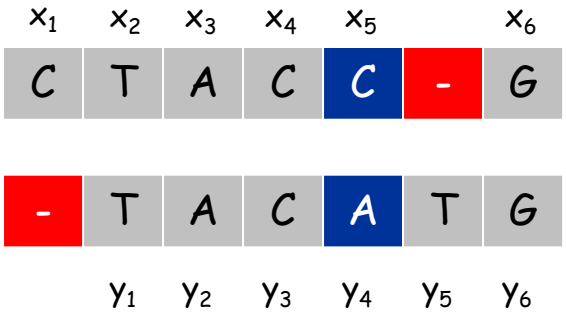
Sequence Alignment

Given two strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ find an alignment with minimum number of mismatch and gaps.

An alignment is a set of ordered pairs $(x_{i_1}, y_{j_1}), (x_{i_2}, y_{j_2}), \ldots$ such that $i_1 < i_2 < \cdots$ and $j_1 < j_2 < \cdots$

Example: CTACCG vs. TACATG.
Sol: We aligned
x_2-y_1, x_3-y_2, x_4-y_3, x_5-y_4, x_6-y_6.

So, the cost is 3.



DP for Sequence Alignment

Let $OPT(i, j)$ be min cost of aligning $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$

Case 1: OPT matches x_i, y_j

- Then, pay mis-match cost if $x_i \neq y_j$ + min cost of aligning $x_1, \ldots, x_{i-1}$ and $y_1, \ldots, y_{j-1}$ i.e., $OPT(i-1, j-1)$

Case 2: OPT leaves x_i unmatched

- Then, pay gap cost for x_i + $OPT(i-1, j)$

Case 3: OPT leaves y_j unmatched

- Then, pay gap cost for y_j + $OPT(i, j-1)$

```
M[i, j] = min( (x_i=y_j ? 0:1) + M[i-1, j-1],
               1 + M[i-1, j],
               1 + M[i, j-1])
```

Induction

What is the order of induction? (i.e. why there is no loop?)
We can do induction on $i + j$.

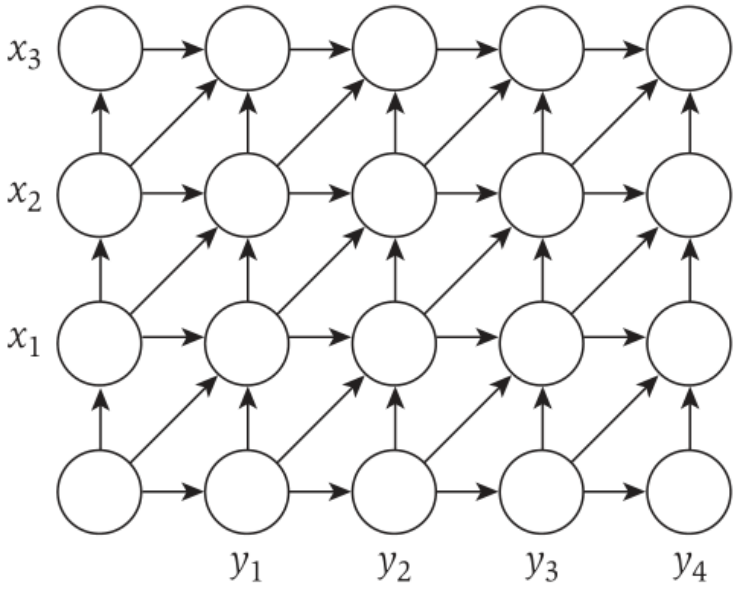


Figure 6.17 A graph-based picture of sequence alignment.

Bottom-up DP

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

  for i = 1 to m
     for j = 1 to n
        M[i, j] = min( (xi=yj ? 0:1) + M[i-1, j-1],
                       1 + M[i-1, j],
                       1 + M[i, j-1])
  return M[m, n]
}
```

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

English words or sentences: m, n $\leq$ 10,..,20.

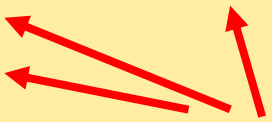
Computational biology: m = n = 100,000. 10 billions ops OK, but 10GB array?

Optimizing Memory

If we are not using strong induction in the DP, we just need to use the last (row) of computed values.

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

  for i = 1 to m
     for j = 1 to n
        M[i, j] = min( (xi=yj ? 0:1) + M[i-1, j-1],
                       1 + M[i-1, j],
                       1 + M[i, j-1])
  return M[m, n]
}
```



Just need $i-1, i$ rows to compute M[i,j]

DP with $O(m+n)$ memory

- Keep an Old array containing values of the last row
- Fill out the new values in a New array
- Copy new to old at the end of the loop

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn) {
  for i = 0 to m
     O[i] = i
  for i = 1 to m
     N[0]=i
     for j = 1 to n
        N[j] = min( (xi=yj ? 0:1) + O[j-1],
                    1 + O[j],
                    1 + N[j-1])
     for j = 1 to n
        O[j]=N[j]
  return N[n]
}
```

M[i-1, j-1]

M[i-1, j]

M[i, j-1]

```
M[i, j] = min( (x_i=y_j ? 0:1) + M[i-1, j-1],
               1 + M[i-1, j],
               1 + M[i, j-1])
```

Shortest Path

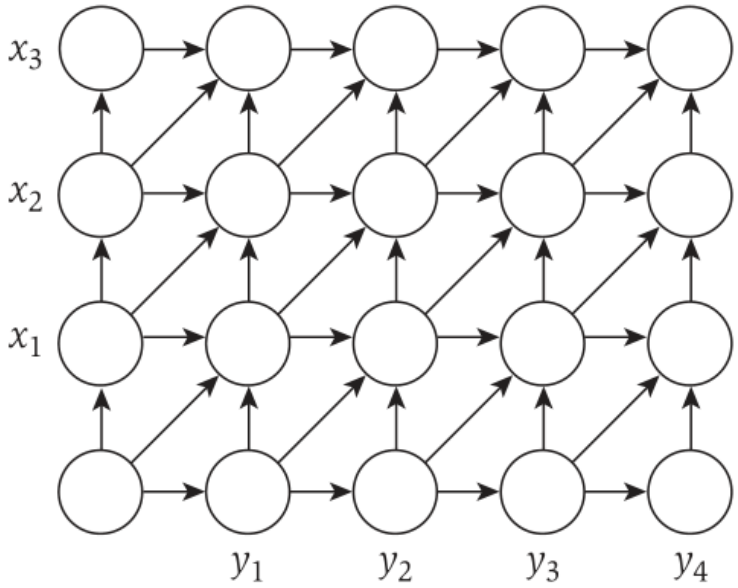


Figure 6.17 A graph-based picture of sequence alignment.

How to recover the alignment?

Hint: bidirectional search.

Lesson

Advantage of a bottom-up DP:

It is much easier to optimize the space.

By the way, edit distance

- can be computed in $O(\frac{n^2}{\log^2 n})$ (1980).
- can be approximated in $O(n^{1+\varepsilon})$ (~2010).
- cannot be solved in $O(n^{2-\delta})$ exactly (2015).