Dynamic Programming String Alignment

Class 28

Dynamic Programming General Strategy

1. characterize the structure of an optimal solution
2. give a recursive definition of the value of an optimal solution
3. implement the definition of step 2 with memoization added and compute the optimal value
4. construct the structure of the optimal value from the results of the computation in step 3

- sometimes step 4 is optional, depending on the original question

String Similarity

- a spell checker encounters a word not in its dictionary
- we expect it to suggest alternatives that are "close to" the typed word
- what does "close to" mean?
- how does it do that?

SUNNY SNOWY

are these two words close?

- we need a metric to measure the closeness of two strings

Metric

- a simple metric is position-by-position
- do the letters match or not?

```
S U N N Y
S N O W Y
```

$$D = \frac{\text{match positions}}{\text{total positions}} = \frac{2}{5} = 0.4$$

- if both sequences were always exactly the same length we could use this D metric

Gaps

- the thing that makes alignment so hard is the concept of gaps
- with sequences of unequal length
  (and even with the same length)
- we need to allow gaps to get the best score

```
S U N - N Y
S - N O W Y
```

with gaps the match score is now $\frac{3}{6} = 0.5$

Too Many Possible Alignments

- the need to possibly introduce a gap at any point makes optimal alignment appear to be an impossible problem
- allowing only single, interior gaps

Seq Length	**# Alignments**
5	19
6	51
10	3,139

$$\#\text{ alignments} = \binom{n-1}{0} + \binom{n-1}{1}\binom{n-2}{1} + \binom{n-2}{2}\binom{n-3}{2} + \cdots$$

continue until a "denominator" exceeds its "numerator"

Too Many Possible Alignments

- for each of those 3,139 alignments, there is a score (explained shortly)
- optimal alignment is found by calculating the score for each, then picking the best one
- this is computationally infeasible

- the key to managing the combinatorial explosion is to recognize that there is vast redundancy calculating all 3,139 alignment scores from scratch
- there are sub-alignments that are re-calculated many, many times
- we need a technique to avoid re-calculating sub-alignments we have already calculated
- this is exactly what dynamic programming does

Sub-Alignments

- for example, consider the following two alignments:

```
A - B C D - E F     A - B C D - E F
Z Y - X W V - U     Z Y - X W - V U
      (a)                 (b)
```

- the two alignments differ in the red region, but are the same in the blue region
- to calculate the overall score of both alignments, the blue region needs to be calculated only once
- and then re-used whenever it occurs again
- that's what dynamic programming does!

Alignment

- we will use the alignment score as our metric
- below are two possible alignments
- a dash represents a gap
- any number of gaps may be placed in either string
- but a gap is never aligned with a gap

S	–	N	O	W	Y		–	S	N	O	W	–	Y
S	U	N	N	–	Y		S	U	N	–	–	N	Y

Alignment Score

- there are four cases for each position
  1. the letters in the two strings match
  2. the letters in the two strings do not match (they mismatch)
  3. the letter in the first string is aligned with a gap
  4. the letter in the second string is aligned with a gap
- an alignment has a score
- we reward positions that have matching letters and penalize positions that have mismatches or gaps
- the maximum possible alignment score corresponds to the optimal alignment

S	–	N	O	W	Y		–	S	N	O	W	–	Y
S	U	N	N	–	Y		S	U	N	–	–	N	Y

Step 1

- let s be the first string of length n and t the second string of length m
- for any alignment with position i in s is aligned with position j of t there are four possibilities
  1. $s[i] = t[j]$: a match reward
  2. $s[i] \neq t[j]$: a mismatch penalty
  3. $s[i]$ is aligned with a gap in t: a gap penalty
  4. $t[j]$ is aligned with a gap in s: a gap penalty
- we wish to align s and t, possibly introducing gaps in each, to maximize the score

Step 2

- consider the ith position of s aligned with the jth position of t
  they can each be a letter of the string and 1 they are the same letter, or 2 they are not the same letter)
  3 s can have a gap
  4 t can have a gap
- let $\text{opt}(i, j)$ be the optimum score of $s[0..i]$ and $t[0..j]$
- mr, mp, and gp are the reward and penalties
- what is an expression for opt?
- at this point on Wednesday, I gave an expression for opt for coins

$$\text{opt}(i, a) = \min \begin{cases} \text{opt}(i, a - \text{denom.at(i)}) + 1 & \text{if we use this coin} \\ \text{opt}(i - 1, a) & \text{if we do not use this coin} \end{cases}$$

- can you think of a similar expression for this problem?
- this is what you have to do to create a dynamic programming solution

Step 2

- for any alignment with position i in s is aligned with position j of t there are four possibilities
  1. $s[i] = t[j]$: a match reward
  2. $s[i] \neq t[j]$: a mismatch penalty
  3. $s[i]$ is aligned with a gap in t: a gap penalty
  4. $t[j]$ is aligned with a gap in s: a gap penalty
- we wish to align s and t, possibly introducing gaps in each, to maximize the score

$$\mathrm{opt}(i,j) = \max \begin{cases} \mathrm{opt}(i-1,j-1) + \mathrm{mr} & \text{if } s[i] = t[j] \\ \mathrm{opt}(i-1,j-1) + \mathrm{mp} & \text{if } s[i] \neq t[j] \\ \mathrm{opt}(i-1,j) + \mathrm{gp} & \text{for a gap in } t \\ \mathrm{opt}(i,j-1) + \mathrm{gp} & \text{for a gap in } s \end{cases}$$

Memo Table

- set up the memo table
- draw an $n+1 \times m+1$ matrix
- add a dummy gap "–" to the start of each sequence
- for ease, label the matrix rows and columns with the characters of s and t

	–	S	U	N
–				
S				
N				
O				
W				

Memo Table

- cell (0,0) = 0
- this represents two empty strings aligned
- a gap represents no letter

	–	S	U	N
–	0			
S				
N				
O				
W				

Memo Table

Scoring Parameters

Match reward: 1

Mismatch penalty: -1

Gap penalty: -2

- fill in values in 1st row
- each cell adds another gap penalty

	–	S	U	N
–	0	-2	-4	-6
S				
N				
O				
W				

Memo Table

- fill in remaining cells
- for each cell (i, j) the value is the largest of

1. $(i-1, j) + \text{gp}$
2. $(i, j-1) + \text{gp}$

3a. $(i-1, j-1) + \text{mr}$
if $s[i] = t[j]$

3b. $(i-1, j-1) + \text{mp}$
if $s[i] \neq t[j]$

Scoring Parameters

Match reward (mr): 1
Mismatch penalty (mp): -1
Gap penalty (gp): -2

	–	S	U	N
–	0	-2	-4	-6
S				
N				
O				
W				

Memo Table

- fill in remaining cells
- for each cell (i, j) the value is the largest of

1. $(i-1, j) + \text{gp}$
2. $(i, j-1) + \text{gp}$

3a. $(i-1, j-1) + \text{mr}$
if $s[i] = t[j]$

3b. $(i-1, j-1) + \text{mp}$
if $s[i] \neq t[j]$

	–	S	U	N
–	0	−2	−4	−6
S	−2	1	−1	−3
N				
O				
W				

Scoring Parameters

Match reward (mr): 1
Mismatch penalty (mp): −1
Gap penalty (gp): −2

Memo Table

- fill in remaining cells
- for each cell (i, j) the value is the largest of

1. $(i-1, j) + \text{gp}$
2. $(i, j-1) + \text{gp}$

3a. $(i-1, j-1) + \text{mr}$
if $s[i] = t[j]$

3b. $(i-1, j-1) + \text{mp}$
if $s[i] \neq t[j]$

	–	S	U	N
–	0	−2	−4	−6
S	−2	1	−1	−3
N	−4	−1	0	0
O	−6	−3	−2	−1
W	−8	−5	−4	−3

Scoring Parameters

Match reward (mr): 1
Mismatch penalty (mp): −1
Gap penalty (gp): −2

Step 3

```
int opt(s, t, i, j, memo)
{
  // check and handle various base cases ...
  if (memo.at(i, j) == INF)
  {
    if (s.at(i) == t.at(j))
    {
      value1 = opt(s, t, i - 1, j - 1) + MR;
    }
    else
    {
      value1 = opt(s, t, i - 1, j - 1) + MP;
    }
    memo.at(i, j) = max(value1,
                        opt(s, t, i, j - 1) + GP,
                        opt(s, t, i - 1, j) + GP);
  }
  return memo.at(i, j);
}
```

Step 4

- we have a filled-in memo table (what does each entry represent?)
- we know the optimal alignment score (what is it?)
- now need to backtrace to see what the optimal alignment is

	–	S	U	N
–	0	−2	−4	−6
S	−2	1	−1	−3
N	−4	−1	0	0
O	−6	−3	−2	−1
W	−8	−5	−4	−3

Step 4

- the "final answer" is the bottom-right entry
- where did that entry come from?

	–	S	U	N
–	0	−2	−4	−6
S	−2	1	−1	−3
N	−4	−1	0	0
O	−6	−3	−2	−1
W	−8	−5	−4	−3

Step 4

- the "final answer" is the bottom-right entry
- where did that entry come from?

	–	S	U	N
–	0	−2	−4	−6
S	−2	1	−1	−3
N	−4	−1	0	0
O	−6	−3	−2	−1
W	−8	−5	−4	−3

Step 4

- the "final answer" is the bottom-right entry
- where did that entry come from?
- where did that -1 entry come from?

	–	S	U	N
–	0	-2	-4	-6
S	-2	1	-1	-3
N	-4	-1	0	0
O	-6	-3	-2	-1
W	-8	-5	-4	-3

Step 4

- the "final answer" is the bottom-right entry
- where did that entry come from?
- where did that -1 entry come from?

	–	S	U	N
–	0	-2	-4	-6
S	-2	1	-1	-3
N	-4	-1	0	0
O	-6	-3	-2	-1
W	-8	-5	-4	-3

Step 4

- the "final answer" is the bottom-right entry
- where did that entry come from?
- where did that -1 entry come from?
- continue the backtrace to (0,0)

	–	S	U	N
–	0	-2	-4	-6
S	-2	1	-1	-3
N	-4	-1	0	0
O	-6	-3	-2	-1
W	-8	-5	-4	-3

Step 4

- the completed backtrace

	–	S	U	N
–	0	−2	−4	−6
S	−2	1	−1	−3
N	−4	−1	0	0
O	−6	−3	−2	−1
W	−8	−5	−4	−3

Step 4

- the completed backtrace
- there are three kinds of arrows:
  1. ↖ match or mismatch
  2. ↑ gap in top sequence
  3. ← gap in left sequence

	–	S	U	N
–	0	−2	−4	−6
S	−2	1	−1	−3
N	−4	−1	0	0
O	−6	−3	−2	−1
W	−8	−5	−4	−3

Step 4

- the completed backtrace
- there are three kinds of arrows:
  1. ↖ match or mismatch
  2. ↑ gap in top sequence
  3. ← gap in left sequence
- final alignment:

S	U	N	–
S	N	O	W

	–	S	U	N
–	0	−2	−4	−6
S	−2	1	−1	−3
N	−4	−1	0	0
O	−6	−3	−2	−1
W	−8	−5	−4	−3

Step 4

- we notice there is an alternate backtrace
- what's up with that?

	–	S	U	N
–	0	−2	−4	−6
S	−2	1	−1	−3
N	−4	−1	0	0
O	−6	−3	−2	−1
W	−8	−5	−4	−3

Step 4

- we notice there is an alternate backtrace
- what's up with that?
- alternate alignment with same optimal score:

S	U	–	N
S	N	O	W

	–	S	U	N
–	0	−2	−4	−6
S	−2	1	−1	−3
N	−4	−1	0	0
O	−6	−3	−2	−1
W	−8	−5	−4	−3

Historical Note

- string alignment is the most famous example of dynamic programming
- after mathematician Bellman invented d.p. in the 1950's, computer scientists quickly applied it to string alignment
- 1960's: the structures of nucleic acids and proteins were being resolved, biologists needed to align amino and nucleic acid sequences — long strings of letters

```
                        10        20        30        40        5
E.coli [a]        SDKIIHLTDDSFDTDVLKADG-AILVDFWAEWCGPCKMIAPILDEIADEY
yeast [b]          MVTQLKSASEYDSALASGDK-LVVVDFFATWCGPCKMIAPMIEKFAEQY
spinach f [c]  MEAIVGKVTEVNKDTFWPIVKAAGDKPVVLDMFTQWCGPCKAMAPKYEKLAEEY
human [d]         VKQIESK-TAFQEALDAAGDKLVVVDFSATWCGPCKMIKPFFHSLSEKY
structure            β         α         β         α
```

History

- in 1970 Needleman and Wunsch, two biologists, painfully and with much struggle re-invented d.p.
- if they had talked to computer scientists, they'd have had the answer in 10 minutes
- this a foundation point of bioinformatics

Realism

- when used to spell-check, this algorithm works
- when used in a biological context, it is a bit simplistic
- in nucleic acids (DNA and RNA), a number of issues arise
  - letter mismatches are far more common than gaps
  - purine-purine (A,G) and pyrimidine-pyrimidine (C,T,U) mismatches are much more common than purine-pyrimidine mismatches
  - non-alignments in position mod 3 are much more common than in positions mod 1 or 2
  - gaps in multiples of length 3 are much more common than gap lengths of any other size
- we can accommodate some of these issues by modifying the penalties
- accommodating others requires more sophisticated affine penalty algorithms

Alignment Variations

- the string alignment presented here is global alignment: align two strings in their entirety
- there are three main variations
  - semiglobal alignment: interior gaps are penalized, but terminal "gaps" are not
  - local alignment: interior gaps are penalized, terminal gaps are not, and a partial alignment is abandoned when its score becomes worse than some threshold
  - longest common subsequence: gaps are completely ignored

Semiglobal Alignment

- find the spot in a very long string s where a relatively short substring t best aligns
- semiglobal alignment is how you search a 4-billion DNA genome for the spot where your gene of interest matches
- since terminal "gaps" are not penalized, 0th column and 0th row have value 0, not multiples of gap penalties
- start traceback from best value in last row, not bottom-right corner

Local Alignment

- in the course of evolution, some regions of a gene have diverged significantly
- other regions are highly conserved
- what are the regions of conservation?
- find the sections of two sequences that are very similar, and do not stress about the sections that are not very similar
- terminal gaps are not penalized
- start traceback from best value in entire matrix, stop at threshold value, repeat