Sequence Alignment and Dynamic Programming

Reminder: Last lecture / recitation

- Schedule for the term
  - 'Foundations' till midterm
  - 'Frontiers' lead to final project
  - Duality: basic problems / fundamental techniques
- Biology introduction
  - DNA, RNA, protein, transcription, translation
  - Why computational biology
- Comparative genomics for motif discovery
  - First recitation: Counting conserved motif instances
  - Evaluating significance of evolutionary conservation
- Comparative genomics is everywhere!
  - Problem set 1: dating vertebrate whole-genome duplication
  - Problem set 2: discover genes using their conservation properties
  - Problem set 3: discover all motifs across entire yeast genome
  - Problem set 4: reversing human/mouse genome rearrangements

Today’s goal:
How do we actually align two genes?

Counting ‘conserved’ motif instances

We can ‘read’ evolution to reveal functional elements

Challenges in Computational Biology

Comparing Genomes

DNA, RNA, protein, transcription, translation

Duality: basic problems / fundamental techniques
Genomes change over time

A C G T C A T C A
mutation
A C G T G A T C A
A C T G T C A
deletion
A C G T C A
A C T G T C A
end

Goal of alignment: Infer edit operations
begin
A C G T C A T C A
end

Question 1: Aligning two (ungapped) strings
- Given two possibly related strings S1 and S2
  - What is the longest common substring? (no gaps)

S1 A C G T C A T C A
S2 T A G T G T C A

Scoring function:
Match(x,x) = +1
Mismatch(A,G)= -½
Mismatch(C,T)= -½
Mismatch(x,y) = -1

S1 A C G T C A T C A
S2 T A G T G T C A
offset: +1

S1 A C G T C A T C A
S2 T A G T G T C A
offset: -2

S1 A C G T C A T C A
S2 T A G T G T C A

Q2: Aligning two (possibly gapped) sequences
- Given two possibly related strings S1 and S2
  - What is the longest common subsequence? (gaps allowed)

S1 A C G T C A T C A
S2 T A G T G T C A

Edit distance:
- Number of changes needed for S1ÆS2
- Uniform scoring function

How can we compute best alignment
- Given additive scoring function:
  - Cost of mutation (AG, CT, other)
  - Cost of insertion / deletion
  - Reward of match
- Need algorithm for inferring best alignment
  - Enumeration?
  - How would you do it?
  - How many alignments are there?

Can we simply enumerate all possible alignments?
- Ways to align two sequences of length m, n
  \[
  \binom{n + m}{m} = \frac{(m + n)!}{(m)! \cdot 2^{m+n}} \approx \frac{2^{m+n}}{\sqrt{\pi \cdot m}}
  \]
- For two sequences of length n

<table>
<thead>
<tr>
<th>n</th>
<th>Enumeration</th>
<th>Today’s lecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>184,756</td>
<td>100</td>
</tr>
<tr>
<td>20</td>
<td>1.40E+11</td>
<td>400</td>
</tr>
<tr>
<td>100</td>
<td>9.00E+58</td>
<td>10,000</td>
</tr>
</tbody>
</table>
Key insight: score is additive!

- Compute best alignment recursively
  - For a given aligned pair \((i, j)\), the best alignment is:
    - Best alignment of \(S[1..i]\) and \(S[1..j]\)
    - Best alignment of \(S[i..n]\) and \(S[j..m]\)

Key insight: re-use computation

Identical sub-problems! We can reuse our work!

Solution #1 – Memoization

- Create a big dictionary, indexed by aligned seqs
  - When you encounter a new pair of sequences
  - If it is in the dictionary:
    - Look up the solution
  - If it is not in the dictionary
    - Compute the solution
    - Insert the solution in the dictionary
- Ensures that there is no duplicated work
  - Only need to compute each sub-alignment once!

Solution #2 – Dynamic programming

- Create a big table, indexed by \((i, j)\)
  - Fill it in from the beginning all the way till the end
  - You know that you’ll need every subpart
  - Guaranteed to explore entire search space
- Ensures that there is no duplicated work
  - Only need to compute each sub-alignment once!
- Very simple computationally!

Top down approach

Bottom up approach

A simple introduction to Dynamic Programming

- Fibonacci numbers

Fibonacci numbers are ubiquitous in nature

- Romanesque spirals
- Nautilus size
- Coneflower spirals
- Leaf ordering
Computing Fibonacci numbers: Top down

- Goal: Compute n-th Fibonacci number.
  - $F(0)=1$, $F(1)=1$, $F(n)=F(n-1)+F(n-2)$
  - $1,1,2,3,5,8,13,21,34,55,89,144,233,377,\ldots$
- Top-down approach:
  - Python code
    ```python
def fibonacci(n):
    if n==1 or n==2: return 1
    return fibonacci(n-1) + fibonacci(n-2)
    ```
  - Analysis: $T(n) = T(n-1) + T(n-2) = (\cdots) = O(2^n)$

Computing Fibonacci numbers: Bottom up

- Top-down approach
  - Python code
  ```python
  def fibonacci(n):
      fib_table[1] = 1
      fib_table[2] = 1
      for i in range(3,n+1):
          fib_table[i] = fib_table[i-1]+fib_table[i-2]
      return fib_table[n]
  ```
  - Analysis: $T(n) = O(n)$

What have we learned?

- Principles of dynamic programming
  - Reveal identical sub-problems
  - Order computation to enable result reuse
  - Systematically fill in table of results
  - Express larger problems from their subparts
- Ordering of computations matters
  - Naïve top-down approach very slow
    - results of smaller problems not available
    - repeated work
  - Systematic bottom-up approach successful
    - Systematically solve each sub-problem
    - Fill-in table of sub-problem results in order
    - Look up solutions instead of recomputing.

How do we apply dynamic programming to sequence alignment?

Key insight: score is additive!

- Compute best alignment recursively
  - For a given aligned pair $(i,j)$, the best alignment is:
    - $S[i..i]$ and $S[j..j]$ if $i=j$
    - $S[i..i]$ and $S[j..j]$ if $i=j$
  - $S[1..i]$ and $S[1..j]$ if $i\neq j$
  - $S[1..i]$ and $S[1..j]$ if $i\neq j$

Store score of aligning $(i,j)$ in matrix $M(i,j)$
Filling in the dynamic programming matrix

- **Local update rules:**
  - Compute next alignment based on previous alignment
  - Just like Fibonacci numbers: \( F[i] = F[i-1] + F[i-2] \)
  - Table lookup!

Duality: seq. alignment ⇔ path through the matrix

**Goal:** Find best path through the matrix
4. Choosing optimal paths

**Initialization:**
- Top left: 0

**Update Rule:**
\[ A(i,j) = \max \{ \]
- \[ A(i-1, j) - 2 \]
- \[ A(i, j-1) - 2 \]
- \[ A(i-1, j-1) - 1 \]
\[ \} \]

**Termination:**
- Bottom right

---

5. Rewarding matches

**Initialization:**
- Top left: 0

**Update Rule:**
\[ A(i,j) = \max \{ \]
- \[ A(i-1, j) - 2 \]
- \[ A(i, j-1) - 2 \]
- \[ A(i-1, j-1) + 1 \]
\[ \} \]

**Termination:**
- Bottom right

---

**What is missing?**
- We know how to compute the best score:
  - Simply the number at the bottom right entry
- But we need to remember where it came from:
  - Pointer to the choice we made at each step
- Retrace path through the matrix:
  - Need to remember all the pointers

Time needed: \( O(mn) \)
Space needed: \( O(mn) \)

Can we do better than that?

---

**Bounded Dynamic Programming**

**Initialization:**
\[ F(i,0), F(0,j) \text{ undefined for } i,j > k \]

**Iteration:**
For \( i = 1 \ldots M \)
For \( j = \max(1, i-k) \ldots \min(N, i+k) \)
\[ F(i, j) = \max \{ F(i, j-1) - d, \quad \text{if } j > i-k(N) \]
\[ F(i-1, j) - d, \quad \text{if } j < i+k(N) \}
\[ \} \]

**Termination:**
same

---

**Linear space alignment**

It is easy to compute \( F(M, N) \) in linear space

Allocate (column[1])
Allocate (column[2])

For \( i = 1 \ldots M \)
If \( i > 1 \), then:
Free( column[i-2] )
Allocate( column[i] )

For \( j = 1 \ldots N \)
\[ F(i,j) = \ldots \]

What about the pointers?

---

**Finding the best back-pointer for current column**

- Now, using 2 columns of space, we can compute for \( k = 1 \ldots M \), \( F(M/2, k), F(M/2, N-k) \)

PLUS the backpointers
Best forward-pointer for current column

- Now, we can find $k'$ maximizing $F(M/2, k) + F(M/2, N-k)$
- Also, we can trace the path exiting column $M/2$ from $k'$

Recursively find midpoint for left & right

- Iterate this procedure to the left and right!

Total time cost of linear-space alignment

- Dynamic programming
  - Reuse of computation
  - Order sub-problems. Fill table of sub-problem results
  - Read table instead of repeating work (ex: Fibonacci)
- Sequence alignment
  - Edit distance and scoring functions
  - Dynamic programming matrix
  - Matrix traversal path $\Rightarrow$ Optimal alignment
- Thursday: Variations on sequence alignment
  - Local and global alignment
  - Affine gap penalties
  - Algorithmic speed-ups
- Recitation:
  - Dynamic programming applications
  - Probabilistic derivations of alignment scores

Summary

Total Time: $cMN + cMN/2 + cMN/4 + \ldots \approx 2cMN = O(MN)$

Total Space: $O(N)$ for computation,
$O(N+M)$ to store the optimal alignment