Sequence Alignment and Dynamic Programming

Tue Sept 11, 2007

Challenges in Computational Biology

| 1 | Genome Assembly |
| 2 | Gene Finding |
| 3 | Regulatory motif discovery |
| 4 | Sequence alignment |
| 5 | Comparative Genomics |
| 6 | Evolutionary Theory |
| 7 | Gene expression analysis |
| 8 | Database lookup |
| 9 | Protein network analysis |
| 10 | Cluster similarity |
| 11 | Gibbs sampling |
| 12 | Emerging network properties |

Reminder: Last lecture / recitation

- Schedule for the term
  - 'Foundations' till midterm
  - 'Frontiers' lead to final project
- 'Foundations' till midterm

Today: 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

Evolution preserved functional elements!

How do we actually align two genes?

We can 'read' evolution to reveal functional elements

Today’s goal:

Evolution preserved functional elements!
Comparing Genomes

Genomes change over time

Mutation
Deletion
Insertion

Goal of alignment: Infer edit operations

Question 1: Aligning two (ungapped) strings

Given two possibly related strings $S_1$ and $S_2$

What is the longest common substring? (no gaps)

<table>
<thead>
<tr>
<th>S1</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
<td>T</td>
<td>A</td>
<td>G</td>
<td>T</td>
<td>G</td>
<td>T</td>
<td>C</td>
<td>A</td>
</tr>
</tbody>
</table>

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

Scoring:
- A: +1 -½ -1 -1
- G: -½ +1 -1 -1
- T: -1 -1 +1 -½
- C: -1 -1 -½ +1

Q2: Aligning two (possibly gapped) sequences

Given two possibly related strings $S_1$ and $S_2$

What is the longest common subsequence? (gaps allowed)

<table>
<thead>
<tr>
<th>S1</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
<td>T</td>
<td>A</td>
<td>G</td>
<td>T</td>
<td>G</td>
<td>T</td>
<td>C</td>
<td>A</td>
</tr>
</tbody>
</table>

Edit distance:
- Number of changes needed for $S_1 \rightarrow S_2$
- 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 \( m, n \)
  \[
  \binom{n + m}{m} = \frac{(m + n)!}{(m!)^2} \approx \frac{2^{m+n}}{\sqrt{\pi \cdot m}}
  \]

- 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!

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

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!

Key insight: re-use computation

<table>
<thead>
<tr>
<th></th>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACGTCATCA</td>
<td>TAGTGCA</td>
<td></td>
</tr>
<tr>
<td>TAGTGCA</td>
<td>ACGTCATCA</td>
<td></td>
</tr>
<tr>
<td>ACGTCATCA</td>
<td>TAGTGCA</td>
<td></td>
</tr>
<tr>
<td>TAGTGCA</td>
<td>ACGTCATCA</td>
<td></td>
</tr>
<tr>
<td>ACGTCATCA</td>
<td>TAGTGCA</td>
<td></td>
</tr>
<tr>
<td>ACGTCATCA</td>
<td>TAGTGCA</td>
<td></td>
</tr>
</tbody>
</table>

Identical sub-problems! We can reuse our work!

Solution #2 – Dynamic programming

- Fibonacci numbers
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

A simple introduction to Dynamic Programming

- Fibonacci numbers
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
Fibonacci numbers are ubiquitous in nature:

- Rabbits per generation
- Leaves per height
- Romanesque spirals
- Nautilus size
- Coneflower spirals
- Leaf ordering

**Computing Fibonacci numbers: Top down**

- Fibonacci numbers are defined recursively:
  ```python
def fibonacci(n):
    if n==1 or n==2: return 1
    return fibonacci(n-1) + fibonacci(n-2)
  ```
- 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,…

**Analysis:**

- T(n) = T(n-1) + T(n-2) = (…) = \(O(2^n)\)

**Computing Fibonacci numbers: Bottom up**

- Top-down approach
  ```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) = \(O(n)\)

**Lessons from iterative Fibonacci algorithm**

- What did the iterative solution do?
  - Reveal identical sub-problems
  - Order computation to enable result reuse
  - Systematically filled-in table of results
  - Expressed 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

**Dynamic Programming in Theory**

- Hallmarks of Dynamic Programming
  - Optimal substructure: Optimal solution to problem (instance) contains optimal solutions to sub-problems
  - Overlapping subproblems: Limited number of distinct subproblems, repeated many many times

- Typically for optimization problems (unlike Fib example)
  - Optimal choice made locally: max(subsolution score)
  - Score is typically added through the search space
  - Traceback common, find optimal path from indiv. choices

- Middle of the road in range of difficulty
  - Easier: greedy choice possible at each step
  - DynProg: requires a traceback to find that optimal path
  - Harder: no opt. substr., e.g. subproblem dependencies

**Dynamic Programming in Practice**

- Setting up dynamic programming
  - Find ‘matrix’ parameterization (# dimensions, variables)
  - Make sure sub-problem space is finite! (not exponential)
    - If not all subproblems are used, better off using memoization
    - If reuse not extensive, perhaps DynProg is not right solution!
  - Traversal order: sub-results ready when you need them
    - Computation order matters! (bottom-up, but not always obvious)
  - Recursion formula: larger problems = F(subparts)
    - Typically F(i) includes min() or max(): remember choices!
  - Systematically fill in table of results, finding optimal score
  - Trace-back from optimal score \( \rightarrow \) optimal solution
How do we apply dynamic programming to sequence alignment?

Key insight: score is additive!

```
S1   ACGTCA T
S2   TAGTGC T
```

• Compute best alignment recursively
  – For a given aligned pair (i, j), the best alignment is:
    • Best alignment of S1[1..i] and S2[1..j]
    • + Best alignment of S1[i+1..n] and S2[j+1..m]

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!

0. Setting up the scoring matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Initialization:
• Top left: 0

Update Rule:
A(i,j) = max{

Termination:
• Bottom right

Duality: seq. alignment ⇔ path through the matrix

Goal:
Find best path through the matrix

Store score of aligning (i,j) in matrix M(i,j)
1. Allowing gaps in s

Initialization:
- Top left: 0

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

Termination:
- Bottom right

2. Allowing gaps in t

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

3. Allowing mismatches

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

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:
\[ A(i,j) = \max\{ \]
- \[ A(i-1,j) - 2 \]
- \[ A(i,j-1) - 2 \]
- \[ A(i-1,j-1) \pm 1 \]

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: \(O(m \times n)\)

Space needed: \(O(m \times n)\)

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 \( j = \max(1, i-k) \ldots \min(N, i+k) \)
\[
F(i, j) = \max \begin{cases} 
F(i-1, j-1) + s(x_i, y_j) \\
F(i-1, j) - d, & \text{if } j > i - k(N) \\
F(i, j-1) - d, & \text{if } j < i + k(N) 
\end{cases}
\]

**Termination:** same

---

**Summary**

- **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**
  - Hashing
  - BLAST
  - Linear-time search
- **Recitation**
  - Local and global alignment
  - Algorithmic speed-ups