Lecture 2

Sequence Alignment and Dynamic Programming

Module 1: Aligning and Modeling Genomes

- **Module 1: Computational foundations**
  - Dynamic programming: exploring exponential spaces in poly-time
  - Introduce Hidden Markov Models (HMMs): Central tool in CS
  - HMM algorithms: Decoding, evaluation, parsing, likelihood, scoring
- **This week: Sequence alignment / comparative genomics**
  - Local/global alignment: infer nucleotide-level evolutionary events
  - Database search: scan for regions that may have common ancestry
- **Next week: Modeling genomes / exon / CpG island finding**
  - Modeling class of elements, recognizing members of a class
  - Application to gene finding, conservation islands, CpG islands

Alignment: Evolution preserves functional elements!

We can ‘read’ evolution to reveal functional elements

Yeast (Kellis et al, Nature 2003), Mammals (Xie, Nature 2005), Fly (Stark et al, Nature 07)
Today’s goal:
How do we actually align two genes?

Goal: Sequence Alignment / Dynamic Programming

1. Introduction to sequence alignment
   - Comparative genomics and molecular evolution
   - From Bio to CS: Problem formulation
   - Why it’s hard: Exponential number of alignments

2. Introduction to principles of dynamic programming
   - Computing Fibonacci numbers: Top-down vs. bottom-up
   - Repeated sub-problems, ordering compute, table lookup
   - DP recipe: (1) Parameterization, (2) sub-problem space,
     (3) traversal order, (4) recursion formula, (5) trace-back

3. DP for sequence alignment
   - Additive score, building up a solution from smaller parts
   - Prefix matrix: finite subproblems, exponential paths
   - Duality: each entry ⇔ prefix alignment score; path ⇔ alignment

4. Advanced topics: three Dynamic Programming variants
   - Linear-time bounded DP (heuristic). Linear-space DP. Gaps
   - Importance of parameterization: 2-D vs. 4-D decomposition

From Bio to CS: Formalizing the problem

- Define set of evolutionary operations (insertion, deletion, mutation)
  - Symmetric operations allow time reversibility (part of design choice)

- Define optimality criterion (min number, min cost)
  - Impossible to infer exact series of operations (Occam’s razor: find min)

- Design algorithm that achieves that optimality (or approximates it)
  - Tractability of solution depends on assumptions in the formulation

Goal of alignment: Infer edit operations

Formulation 1: Longest common substring

- 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

   offset: +1

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

   offset: -2
**Formulation 2: Longest common subsequence**

- Given two possibly related strings S1 and S2
  - What is the longest common subsequence? (gaps allowed)
  - Related to: Edit distance:
    - Number of changes needed for S1 → S2
    - Uniform scoring function

**Formulation 3: Sequence alignment**

- Allow gaps (fixed penalty)
  - Insertion & deletion operations
  - Unit cost for each character inserted or deleted
- Varying penalties for edit operations
  - Transitions (Pyrimidine ⇔ Pyrimidine, Purine ⇔ Purine)
  - Transversions (Purine ⇔ Pyrimidine changes)
  - Polymerase confuses Aw/G and Cw/T more often

**How many alignments are there?**

- Longest ‘non-boring’ alignment: n+m entries
  - Otherwise a gap will be aligned to a gap → condense
- Alignment is equivalent to gap placement
  - (n+m choose n) ways to choose S1 placement
  - At each position yes/no answer of placing character
  - Exponential number of possible placements
- Exponential number of sequence alignment
  - Enumerating and scoring each of them not an option
  - Need faster solution for finding best alignment

**Goal: Sequence Alignment / Dynamic Programming**

1. Introduction to sequence alignment
   - Comparative genomics and molecular evolution
   - From Bio to CS: Problem formulation
   - Why it’s hard: Exponential number of alignments
2. Introduction to principles of dynamic programming
   - Computing Fibonacci numbers: Top-down vs. bottom-up
   - Repeated sub-problems, ordering compute, table lookup
   - DP recipe: (1) Parameterization, (2) sub-problem space, (3) traversal order, (4) recursion formula, (5) trace-back
3. DP for sequence alignment
   - Additive score, building up a solution from smaller parts
   - Prefix matrix: finite subproblems, exponential paths
   - Duality: each entry $\equiv$ prefix alignment score; path $\equiv$ alignment
4. Advanced topics: three Dynamic Programming variants
   - Linear-time bounded DP (heuristic), Linear-space DP. Gaps
   - Importance of parameterization: 2-D vs. 4-D decomposition

**Formulation 4: Varying gap cost models**

1. Linear gap penalty
   - Same as before
2. Affine gap penalty
   - Big initial cost for starting or ending a gap
   - Small incremental cost for each additional character
3. General gap penalty
   - Any cost function
   - No longer computable using the same model
4. Frame-aware gap penalty
   - Multiples of 3 disrupt coding regions
5. Seek duplicated regions, rearrangements, …
   - Etc

**Scoring function:**

- Match(x,x) = +1
-Mismatch(x,y) = -1

**Transitions:**

- A $\rightarrow$ G, C $\rightarrow$ T common
- Transitions (Pyrimidine $\leftrightarrow$ Pyrimidine, Purine $\leftrightarrow$ Purine)
- Transversions (Purine $\leftrightarrow$ Pyrimidine changes)
- Polymerase confuses Aw/G and Cw/T more often

**How many alignments are there?**

- Longest ‘non-boring’ alignment: n+m entries
  - Otherwise a gap will be aligned to a gap → condense
- Alignment is equivalent to gap placement
  - (n+m choose n) ways to choose S1 placement
  - At each position yes/no answer of placing character
  - Exponential number of possible placements
- Exponential number of sequence alignment
  - Enumerating and scoring each of them not an option
  - Need faster solution for finding best alignment

**Need polynomial algorithm to find best alignment amongst an exponential number of possible alignments!**

**A simple introduction to the principles of Dynamic Programming**

Turning exponentials into polynomials
Computing Fibonacci Numbers

- Fibonacci numbers

\[
F_6 = F_5 + F_4 = (F_4 + F_3) + (F_3 + F_2) = \ldots
\]

Computing Fibonacci numbers: Top down

- Fibonacci numbers are defined recursively:
  - Python code:
    ```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:
    - \( T(n) = T(n-1) + T(n-2) = (\ldots) = O(2^n) \)

Computing Fibonacci numbers: Bottom up

- Bottom up 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: \( T(n) = O(n) \)

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: \( \max( \text{subsolution score} ) \)
  - 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
Hallmarks of optimization problems

1. Optimal substructure
   An optimal solution to a problem (instance) contains optimal solutions to subproblems.

2. Overlapping subproblems
   A recursive solution contains a “small” number of distinct subproblems repeated many times.

3. Greedy choice property
   Locally optimal choices lead to globally optimal solution

Dynamic Programming in Practice

- Setting up dynamic programming
  1. Find ‘matrix’ parameterization (# dimensions, variables)
  2. 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!
  3. Traversal order: sub-results ready when you need them
     • Computation order matters! (bottom-up, but not always obvious)

4. Recursion formula: larger problems = F(subparts)
  5. Remember choices: typically F() includes min() or max()
     • Need representation for storing pointers, is this polynomial!

Then start computing
  1. Systematically fill in table of results, find optimal score
  2. Trace-back from optimal score, find optimal solution

Goal: Sequence Alignment / Dynamic Programming

1. Introduction to sequence alignment
   – Comparative genomics and molecular evolution
   – From Bio to CS: Problem formulation
   – Why it’s hard: Exponential number of alignments

2. Introduction to principles of dynamic programming
   – Computing Fibonacci numbers: Top-down vs. bottom-up
   – Repeated sub-problems, ordering compute, table lookup
   – DP recipe: (1) Parameterization, (2) sub-problem space,
     (3) traversal order, (4) recursion formula, (5) trace-back

3. DP for sequence alignment
   – Additive score, building up a solution from smaller parts
   – Prefix matrix: finite subproblems, exponential paths
   – Duality: each entry \( \circ \) prefix alignment score; path \( \circ \) alignment

4. Advanced topics: three Dynamic Programming variants
   – Linear-time bounded DP (heuristic), Linear-space DP. Gaps
   – Importance of parameterization: 2-D vs. 4-D decomposition

Key insight: iteratively grow best alignment solution

<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>
<th>A</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>
<td></td>
</tr>
</tbody>
</table>

- Compute all alignment scores from the bottom up
  – Define \( M[i,j] \) prefix alignment score of \( S_1[1..i] \) and \( S_2[1..j] \)
  – Fill up table recursively from smaller to bigger alignments

- Express alignment of \( S_1[1..i+1] \) and \( S_2[1..j+1] \) \( \Rightarrow M[i+1,j+1] \)
  – One of three possibilities: (1) extend alignment from \( M[i,j] \)
    (2) extend from \( M[i-1,j] \), or (3) extend from \( M[i,j-1] \)
  – Only a local computation, takes \( O(1) \) time!

- Proof of correctness (cut-and-paste argument from 6.006)
  – Best alignment of \( S_1[1..i+1] \) and \( S_2[1..j+1] \) must be composed of best alignments of smaller prefix
  – Proof: otherwise could replace sub and get better overall

Computing alignments recursively: \( M[i,j]=F(smaller) \)

- Local update rules, only look at neighboring cells:
  – Compute next alignment based on previous alignment
  – Just like Fibonacci numbers: \( F[i] = F[i-1] + F[i-2] \)
  – Table lookup avoids repeated computation

- Computing the score of a cell from smaller neighbors
  \[
  M(i,j) = \max\{ M(i-1, j-1) + \text{score} \} \\
  M(i, j-1) - \text{gap} \]
  – Only three possibilities for extending by one nucleotide:
    a gap in one species, a gap in the other, a (mis)match

- Compute scores for prefixes of increasing length
  – Start with prefixes of length 1, extend by one each time, until all prefixes have been computed
  – When you reach bottom right, alignment score of \( S_1[1..m] \) and \( S_2[1..n] \) is alignment of full \( S_1 \) and full \( S_2 \)
  – (Can then trace back to construct optimal path to it)
Dynamic Programming for sequence alignment

1. Find 'matrix' parameterization
   - Prefix parameterization. Score(S1[1..i], S2[1..j]) \rightarrow M(i,j)
   - (i,j) only prefixes vs. (i,j,k,l) all substrings \rightarrow simpler 2-d matrix
2. Make sure sub-problem space is finite! (not exponential)
   - It's just n^2, quadratic (which is polynomial, not exponential)
3. Traversal order: sub-results ready when you need them
4. Recursion formula: larger problems = Func(subparts)
   - Need formula for computing M[i,j] as function of previous results
   - Single increment at a time, only look at M[i-1,j], M[i,j-1], M[i-1,j-1]
5. Remember choice: F() typically includes min() or max()
   - Remember which of three cells (top,left,diag) led to maximum

Step 1: Setting up the scoring matrix M[i,j]

Initialization:
- Top left: 0
Update Rule:
M(i,j) = \max\{ M(i-1, j) - 2 \text{ gap} \\
M(i, j-1) - 2 \text{ gap} \\
M(i-1, j-1) + 1 \text{ match} \\
M(i-1, j-1) + 1 \text{ mismatch} \}
Termination:
- Bottom right

Goal:
Find best path through the matrix
Goal: Sequence Alignment / Dynamic Programming

1. Introduction to sequence alignment
   - Comparative genomics and molecular evolution
   - From Bio to CS: Problem formulation
   - Why it’s hard: Exponential number of alignments

2. Introduction to principles of dynamic programming
   - Computing Fibonacci numbers: Top-down vs. bottom-up
   - Repeated sub-problems, ordering compute, table lookup
   - DP recipe: (1) Parameterization, (2) sub-problem space, (3) traversal order, (4) recursion formula, (5) trace-back

3. DP for sequence alignment
   - Additive score, building up a solution from smaller parts
   - Prefix matrix: finite subproblems, exponential paths
   - Duality: each entry ↔ prefix alignment score; path ↔ alignment

4. Advanced topics: three Dynamic Programming variants
   - Linear-time bounded DP (heuristic), Linear-space DP, Gaps
   - Importance of parameterization: 2-D vs. 4-D decomposition

If time permits…

(4) Extensions to basic DP solution

- Initialization: $F(i,0), F(0,j)$ undefined for $i, j > k$
- Iteration:
  
  For $i = 1...M$
  
  For $j = \max(1, i – k) \ldots \min(N, i+k)$
  
  $F(i, j) = \max$
  
  $\begin{cases} 
  F(i-1, j-1) + s(x_i, y_j), & \text{if } j > i – k(N) \\
  F(i-1, j) – d, & \text{if } j < i + k(N) 
  \end{cases}$

- Termination: same

Slides credit: Serafim Batzoglou

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 $j = 1 \ldots N$
  
  $F(i, j) = …$

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

Answer: What is missing? (5) Returning the actual path!

- 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: $O(m\times n)$
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

\[
\text{Total Time: } cMN + cMN/2 + cMN/4 + \ldots = 2cMN = O(MN)
\]

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

Goal: Sequence Alignment / Dynamic Programming

1. Introduction to sequence alignment
   - Comparative genomics and molecular evolution
   - From Bio to CS: Problem formulation
   - Why it's hard: Exponential number of alignments
2. Introduction to principles of dynamic programming
   - Computing Fibonacci numbers: Top-down vs. bottom-up
   - Repeated sub-problems, ordering compute, table lookup
   - DP recipe: (1) Parameterization, (2) sub-problem space, (3) traversal order, (4) recursion formula, (5) trace-back
3. DP for sequence alignment
   - Additive score, building up a solution from smaller parts
   - Prefix matrix: finite subproblems, exponential paths
   - Duality: each entry \( \supset \) prefix alignment score; path \( \supset \) alignment
4. Advanced topics: three Dynamic Programming variants
   - Linear-time bounded DP (heuristic). Linear-space DP. Gaps
   - Importance of parameterization: 2-D vs. 4-D decomposition

Key insight: score is additive!

- For a given aligned pair \((i, j)\), the best alignment is:
  - Best alignment of \( S1[1..i] \) and \( S2[1..j] \)
  - + Best alignment of \( S1[1,..n] \) and \( S2[1,..m] \)
- Proof: cut-and-paste argument (see 6.046)

Additional insights

Why the 2-dimentional parameterization worked

This allows a single recursion (top-left to bottom-right) instead of two recursions (middle-to-outside to top-down)
**Goal: Sequence Alignment / Dynamic Programming**

1. Introduction to sequence alignment
   - Comparative genomics and molecular evolution
   - From Bio to CS: Problem formulation
   - Why it’s hard: Exponential number of alignments
2. Introduction to principles of dynamic programming
   - Computing Fibonacci numbers: Top-down vs. bottom-up
   - Repeated sub-problems, ordering compute, table lookup
   - DP recipe: (1) Parameterization, (2) sub-problem space, (3) traversal order, (4) recursion formula, (5) trace-back
3. DP for sequence alignment
   - Additive score, building up a solution from smaller parts
   - Prefix matrix: finite subproblems, exponential paths
   - Duality: each entry \( \leftrightarrow \) prefix alignment score; path \( \leftrightarrow \) alignment
4. Advanced topics: three Dynamic Programming variants
   - Linear-time bounded DP (heuristic). Linear-space DP. Gaps
   - Importance of parameterization: 2-D vs. 4-D decomposition

**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!

**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 \( \leftrightarrow \) Optimal alignment
- Thursday: Variations on sequence alignment
  - Local/global alignment, affine gaps, algo speed-ups
  - Semi-numerical alignment, hashing, database lookup
- Recitation:
  - Dynamic programming applications
  - Probabilistic derivations of alignment scores