Sequence Similarity

Topic: how to search for strings that are “similar” to a pattern
definitions of similarity
algorithms
similarity in bioinformatics applications

Road Map

- The last few weeks we have been looking at methods for searching large amounts of text
- Exact match algorithms
  - naive search
  - Boyer-Moore
- Indexing
  - hash tables
  - used in databases, Google, other applications
- Regular expressions
  - roots in automata theory
  - input is a description of string(s) to find, not the exact text itself

Today’s Topic

- Today we will look at another variation of string search
- In an inexact match search we want to find strings “similar” to -- but not exactly the same as -- the input pattern
- also known as approximate match
- Example:
  - input is a protein sequence, e.g. human hemoglobin β
  - string from alphabet of 20 amino acid letters (A, C, D, ...)
  - database at NCBI is a collection of all known proteins
  - A search for similar sequences should return
  - other globin proteins in the human genome
  - sequences for hemoglobin proteins in other vertebrates

Regular Expressions and Proteins

- Regular expressions are often used to search protein databases
- Proteins have recurring patterns known as motifs
  - proteins with similar functions have one or more motifs in common
- Example: proteins that initiate transcription
  - a “zinc finger” has four amino acids (3 C’s and an H) that attach to a zinc ion
  - contact with zinc changes the protein’s shape, allows it to interact with DNA
- The common pattern in proteins that have this motif:
  \[ /C.*\{(2)\}H.*\{(15,17)\}C.*\{(2)\}C/ \]
- A search for this pattern in a database should turn up several “transcription factor” proteins
  - plus many others that happen to have this pattern...
Regular Expressions in MySQL

Here is what I learned when I searched for this pattern in a database of zebrafish genes:

```plaintext
mysql> select count(xref) from genes
    where aaseq regexp 'C.{2}H.{15,17}C.{2}C';
+-------------+
|         227 |
+-------------+
mysql> select xref from genes
    where aaseq regexp 'C.{2}H.{15,17}C.{2}C' limit 10;
+---------------------------------------------------+
<table>
<thead>
<tr>
<th>xref</th>
</tr>
</thead>
<tbody>
<tr>
<td>GI:68354836 GeneID:566624</td>
</tr>
<tr>
<td>GI:68354838 GeneID:567196</td>
</tr>
<tr>
<td>GI:68354052 GeneID:558046</td>
</tr>
</tbody>
</table>
```

Regular Expressions vs Inexact Match

- A regular expression is a pattern that matches many different strings in the database
  - if the expression has .., \w, etc the matches may all be different
- Why bother with inexact match?
  - Answer:
    - we can’t do a regular expression search unless we can describe the pattern
    - e.g. we can’t search for zinc fingers unless we know zinc fingers all have 3 C’s and an H at the designated locations
- With inexact match we don’t need to know how to describe the pattern
  - any string can be used as a pattern
  - the search will find other strings that are “similar” (which we still haven’t defined...)

$k$-mismatch Problem

- The simplest form of inexact string search is known as $k$-mismatch
- here $k$ means “small constant number”
- Example:
  - search for “ruby” in “rub-a-dub-dub three men in a tub” with $k = 2$
  - returns matches at
    - 0: “rub-”
    - 6: “dub-”
    - 10: “dub ”
    - 20: “tub” (?)
- It is fairly easy to modify the simple-search algorithm to allow mismatches
  - at each window position set a counter to 0
  - move the window when $P_j \neq T_{j+k}$ and counter = $k$
  - can also be done with Boyer-Moore and other fancy search algorithms
Distance

- The k-mismatch algorithm gives us a simple way to define similarity
  - recall the goal is to search a database for strings that are “similar” to a pattern P
- The similarity of strings S and T can be defined as the number of mismatches between the strings
  - the number of mismatches is often called the distance between the strings
- Examples:
  - pilgrim d = 3 biology d = 4

Edit Distance

- The k-mismatch problem is fairly simple because the strings we are comparing have the same length
- A more complicated version of inexact match allows “similar” strings to have different lengths
- In this new version of the problem differences occur because
  - a letter in S corresponds to a different letter in T
  - a new letter has been inserted into S
  - a letter has been deleted from S
- When can still use “distance” to measure similarity:
  - the edit distance between strings S and T is the number of editing operations (change/insert/delete) needed to transform S into T

Edit Distance Examples

- “mac” 㱺 “pc”
  - delete m  mac 㱺 ac  d = 2
  - change a to p  ac 㱺 pc
- “management” 㱺 “menagerie”
  - delete t  management 㱺 managemen  d = 5
  - delete n  managemen 㱺 manageme
  - change a to e  managemen 㱺 menageme
  - change m to r  menageme 㱺 menagere
  - insert i  menagere 㱺 menagerie
- “prolog” 㱺 “perl”
- “palatino” 㱺 “alternate”

An Editing Machine

- There are a lot of different sequences of operations that will transform S to T
  - the similarity should be defined by the minimum edit distance
  - how do we find the shortest sequence of operations?
- Start by defining a systematic way to approach the edit distance problem:
  - write the string S on a tape
  - feed the tape into an automaton which will write T on its output tape
- At each step the machine can
  - copy the current letter from S to T
  - read a letter from S, change it, write it to T
  - read a letter from S, write nothing to T (i.e. delete)
  - write a letter to T without moving the input tape (i.e. insert)
Editing Machine (cont’d)

- The “prolog” to “perl” transformation using this machine:
  - copy p
  - insert e
  - copy r
  - delete o
  - copy l
  - delete o
  - delete g
- Total of 7 operations

Alternative Edits

- The editing machine will read S from left to right and apply the editing operations in order
- But there are still many different ways to transform S into T
- Some examples (where S has n letters and T has m letters):
  - If $n > m$: do n “changes” to write the first n letters of T, followed by $m - n$ “deletes” to skip past the remaining letters of S
    - distance = n
  - If $n < m$: do n erases followed by m inserts
    - distance = $m + n$
  - Neither of these seem very satisfactory
    - When strings are very similar we want the machine to many copies and few changes
    - E.g. “bridge” to “badge” should have one delete, one change -- the other letters are the same -- so the distance should only be 2

Minimum Edit Distance

- What we need is a cost function that assigns costs to each operation
- The cost of a sequence of operations is the sum of costs of all operations
- Minimum edit distance -- our measure of similarity -- is defined by the sequence of edit operations that has the lowest cost
- A simple cost function might be:
  - 0 to copy a letter from input to output
  - 1 to change a letter
  - 2 to insert or delete
- This simple function “rewards” the machine for finding the letters that are the same in both strings
- It also “penalizes” the machine for making too many insertions or deletions

Minimum Edit Distance Examples

- Using the simple cost function from the previous slide:
  - For “prolog” => “perl”:
    - copy p, change r to o, change o to e, copy l, delete o, delete g
    - total cost = 6
  - For “human” => “chimpanzee”:
    - insert c, copy h, change u to i, copy m, insert p, copy a, copy n, insert z, insert e
    - insert e
    - total cost = 11
  - In both of these cases the sequence of editing operations shown has the lowest cost of all possible sequences
  - We’ll see how to prove this in a later slide...
Algorithm / Not an Algorithm

- In the lecture on string searching we saw how an automaton provides a “model” for an efficient algorithm
  - preprocess the pattern $P$, build an FSA, ...
  - reduced search time from $O(n \times m)$ to $O(n)$
- Does the editing machine also provide a similar model? i.e. does it describe an algorithm that we can implement in Ruby?
  - input: string $S$ with $n$ letters
  - output: string $T$ with $m$ letters

```plaintext
i ← 0
while $i < n$ do one of these steps:
  • append $S_i$ to $T$, $i ← i+1$
  • append $f(S_i)$ to $T$, $i ← i+1$
  • append a letter to $T$
  • $i ← i+1$
```

Algorithm / Not an Algorithm (cont’d)

- Pro:
  - there is a set of very specific operations (one of the four editing operations)
  - each one is effective (it makes progress toward the final output)
- Con:
  - we haven’t told the machine which of the four operations to do at each step
  - we haven’t specified what letters to use in changes
  - we haven’t explained what to insert

```
while $i < n$ do one of these steps:
  • append $S_i$ to $T$, $i ← i+1$
  • append $f(S_i)$ to $T$, $i ← i+1$
  • append a letter to $T$
  • $i ← i+1$
```

Nondeterministic Algorithm

- If we fix the problem with choosing letters we will have an algorithm
- Change the problem specification:
  - $T$ is a second input (e.g. on a second tape)
  - output is a sequence of operations that transforms $S$ into $T$
- Now we have what is known as a nonCDeterministic algorithm
  - allow the machine to make the “right” choice at each step

```plaintext
i ← 0
j ← 0
while $i < n$ and $j < m$ do one of these steps:
  • write “copy $S_i$”, $i ← i+1$, $j ← j+1$
  • write “change $S_i$ to $T_j$”, $i ← i+1$, $j ← j+1$
  • write “insert $T_j$”, $j ← j+1$
  • write “delete $S_i$”, $i ← i+1$
```

Nondeterministic Algorithm (cont’d)

- It might not seem like the pseudocode for a nondeterministic algorithm is very useful
  - how do we translate this into a working Ruby program??
- A nondeterministic algorithm provides a “framework” for thinking about the final solution
  - now that we have a general idea of how to solve the problem we can focus our attention on the subproblem of choosing the best sequence of operations
- without this framework we had no guidelines for choosing which part of $S$ to copy or change
- One of the lectures next week will use this idea of nondeterminism to explore ideas of what is computable and what is not
- Several chapters in NTO also use the idea of nondeterminism
Similarity of Biosequences

- In the introduction we said we wanted to be able to search NCBI for similar sequences
- Edit distance can also be used to define similarity in genes and proteins
- Here are the first 20 letters in the protein sequences for hemoglobin from three mammals:
  - **human:** MALSAEDRALVRLWKKLGS
  - **mouse:** MARSQDDQNLVIALWKRMGS
  - **rat:** MALSQTDHRLVIALWKRMGS
- The stars show places where the letters are the same
- as one might expect there are 3 fewer changes between mouse/rat
- if distance is “number of identical letters”, the mouse protein is closer to rat than to human

Edit Distance for Biosequences

- For the “prolog” to “perl” example we used an intuitive cost function that rewards similarity and penalizes inserts and deletes
- For biological sequences the cost function should be based on the likelihood of mutations that lead to the differences
- Single letter changes are caused by a point mutation to underlying DNA
- Some changes are more likely than others:
  - D (aspartic acid) and E (glutamic acid) have very similar chemical properties
  - a protein might function equally well if a mutation changes D to E
  - W (tryptophan) is a larger molecule and has very different chemical properties (e.g. it is hydrophobic -- it is repelled by water)
  - a change from D to W is unlikely

Similar Pairs

- Here is the comparison of hemoglobins again, this time indicating similarities between letters
  - **human:** MALSAEDRALVRLWKKLGS
  - **mouse:** MARSQDDQNLVIALWKRMGS
  - **rat:** MALSQTDHRLVIALWKRMGS
- The cost function used for this comparison:
  - L/R -10 Q/R -2
  - Q/A -5 + Q/H 0
  - + D/E 2 W/A -16
  - D/T -6 W/R -3
  - + H/L 0

Insertions and Deletions in Biosequences

- Insertions and deletions also occur in genes and proteins
  - cross-overs or other breaks and recombinations in DNA replication can move large chunks of genes
  - These events are much less frequent than point mutations
  - Programs that compare proteins or DNA use a “gap penalty” as part of the cost function
- Total length of the portions of hemoglobin used in the example:
  - **human:** 142 aa
  - **mouse:** 145 aa
  - **rat:** 143 aa
- a comparison of the complete sequences would have to use some number of insertions and/or deletions to account for different lengths
Dot Plots

- One way to visualize the degree of similarity between two strings is to make a "dot plot."
- For two strings $S$ (length $n$) and $T$ (length $m$):
  - draw a matrix with $n$ rows and $m$ columns
  - label the rows with letters from $S$ ($i = 1..n$)
  - label the columns with letters from $T$ ($j = 1..m$)
  - put a "dot" in row $i$, column $j$ if $S_i = T_j$

How does this help?
- common substrings show up as groups of dots along a diagonal

Dot Plot Examples

- A variation is to use colors to indicate the degree of similarity
  - This plot shows the first 20 letters of human vs mouse hemoglobin

Paths in the Matrix

- The matrix can help us identify a sequence of editing operations to transform $S$ into $T$
  - start at the upper left corner
  - find a path to the lower right corner
  - each move can go right, down, or along the diagonal
- A line segment connecting adjacent cells corresponds to an editing operation:
  - right: insert (add $T_j$)
  - down: delete (skip $S_i$)
  - diagonal: copy (if $S_i = T_j$) or change (if $S_i \neq T_j$)
  - (where $i, j$ identify the cell at the head of the arrow)
Number of Paths

- Now we have an idea of how many different sequences of editing operations are possible: it’s the number of paths that lead from the upper left to the lower right.
  - for an $n \times m$ matrix: almost $3^{n+m}$
- What are the paths for the two “obvious” ways of changing $S$ to $T$?
  (a) four copies/changes (“prol” to “perl”), followed by two deletes?
  (b) six deletes, followed by four inserts?

Best Path

- The matrix is a very effective visual aid.
  - our eyes quickly pick out patterns like diagonal lines.
- But do we know a path that includes diagonals is the best path?

Adding Scores to the Matrix

- To find the best path we can put the cost of an editing operation in the matrix.
- Instead of putting a dot in a cell, we will fill the matrix with numbers.
  - the entry in row $i$, column $j$ will be the cost of the best path for transforming $S_{1..i}$ into $T_{1..j}$.
- First step:
  - add an extra row to the top
  - add an extra column to the left
  - put a 0 in the upper left cell.

Adding Scores (cont’d)

- Now let’s look at the upper left corner.
- The cost of moving to the right to cell (0,1) is 2: this move corresponds to inserting a letter from $T$, and the cost of an insert is 2.
- Similarly, the cost of moving down is also 2: it’s the cost of skipping a letter from $S$.
- Now consider cell (1,1) -- there are three ways to get here:
  - a move from the left
  - a move from above
  - a move along the diagonal.
- Since we know the costs to get to those cells, we can figure out the best cost to put in cell (1,1).
Adding Scores (cont’d)

- Since the cost in cell (0,1) is 2, and moving down from (0,1) to (1,1) would cost 2 more, the cost of this path is 4.
- The same reasoning applies to the arrow that goes from (1,0) to (1,1) -- this path would also cost 4.
- Now consider the move from (0,0) to (1,1).
  - There are 2 p’s here, so the cost of the move is the cost of a match (0) plus the cost in the cell we start from.
  - Total cost: 0.
- Therefore the best way to get to (1,1) is from (0,0).

Minimum Cost

- Now all we have to do to find the lowest cost set of editing operations is follow the path back from the cell in the lower right.
- For this example it turns out the best path is one that copies/replaces the first four letters, then deletes the last two.
  - This path has 6 steps.
- More important is the cost of this path: it is the lowest of any path.
  - In general the lowest cost may not be the shortest.

The Completed Table

- As scores are added to the table, record the path used to get to the cell.
  - For example, the 0 in cell (1,1) was the result of moving there from (0,0).
  - Store an arrow that points back to (0,0) from (1,1).

Another Minimum Cost Example

- The matrix for an earlier edit distance example: transform “human” to “champanzee”.
- What is the sequence of operations defined by this path?
Back to Biology

- In this example of evaluating the similarity of two DNA strings the path that includes the two diagonal runs may not be the lowest cost.

**Diagram:**

- Including this path requires an insert and a delete, both of which are expensive.
- The least expensive path may use this segment, even though it has fewer matches.

Efficiency

- The algorithm that builds a matrix and fills it with costs is \( O(n \times m) \)
  - if there are \( n \) characters in \( S \) and \( m \) in \( T \) the matrix will have \( n \times m \) cells
  - To put a number in a cell we just have to look at three neighboring cells
  - There are \( 3 \times n \times m \) comparisons, so the algorithm is \( O(n \times m) \)
- If the strings are roughly the same length \( (n = m) \):
  - The algorithm to find the lowest cost edit distance is \( O(n^2) \)
- But earlier we said there were \( 3^{n+m} \) paths
- Why isn’t this algorithm \( O(3^n) \)?

Aside: Fibonacci Numbers

- To see what’s going on here, consider a simple Ruby program that computes the Fibonacci sequence:
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
  - The first two numbers \( F_0 \) and \( F_1 \) are both 1
  - For all \( i \geq 2 \) the \( i^{th} \) number is the sum of the previous two: \( F_i = F_{i-2} + F_{i-1} \)

```
def fib(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
end
>> fib(7)
21
```

- \( O(2^n) \)
  - The Ruby method on the previous slide is simple and elegant.... and horribly slow
  - Consider how much work it has to do to compute \( \text{fib}(4) \)
  - Each circle in this diagram corresponds to a call to \( \text{fib} \):
    - Note \( \text{fib}(2) \) is called twice:
      - Once for \( \text{fib}(4) \) and once for \( \text{fib}(3) \)
That didn’t look too bad -- but look at what happens for fib(6):

```
now fib(2) is called 5 times
```

also note fib(4) is called twice...

Things get out of hand very quickly:
```
<table>
<thead>
<tr>
<th>fib(n)</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib(6)</td>
<td>26</td>
</tr>
<tr>
<td>fib(8)</td>
<td>68</td>
</tr>
<tr>
<td>fib(10)</td>
<td>178</td>
</tr>
<tr>
<td>fib(12)</td>
<td>466</td>
</tr>
<tr>
<td>fib(14)</td>
<td>1220</td>
</tr>
<tr>
<td>fib(16)</td>
<td>3194</td>
</tr>
<tr>
<td>fib(18)</td>
<td>8362</td>
</tr>
<tr>
<td>fib(20)</td>
<td>21892</td>
</tr>
<tr>
<td>fib(22)</td>
<td>57314</td>
</tr>
<tr>
<td>fib(24)</td>
<td>150050</td>
</tr>
<tr>
<td>fib(26)</td>
<td>392836</td>
</tr>
<tr>
<td>fib(28)</td>
<td>1028458</td>
</tr>
<tr>
<td>fib(30)</td>
<td>2692538</td>
</tr>
</tbody>
</table>
```

$O(2^n)$ (cont’d)

A Better Way

A major performance improvement is possible if we use an array to keep track of values we have computed so far.

Each time `fib(n)` is called check to see if it's in the array -- if not compute and it save it before returning it's value:

```python
def fib(n):
    if n == 0 or n == 1
        return 1
    else:
        if $a[n] == nil$
            $a[n] = fib(n-2) + fib(n-1)$
        end
        return $a[n]$
extend
```

$a$ means “global array $a$” (a global variable is one that is accessible to but not part of the method)

Back to Edit Distance

So that's a clever way to improve Fibonacci -- what does it have to do with sequence similarity?

Answer:
- the cells in the matrix are like the array in `fib`
- we just have to compute the cost of reaching a cell once
- when we compute the cost, store it in the matrix so it is available for future path calculations

This technique is a general method known as *dynamic programming*

CS majors learn about DP and other optimization algorithms in CIS 315
Review

- A useful extension to the string search algorithms we have been studying is to allow searches for text that is similar to the input pattern.
- Definitions of “similar”:
  - distance (number of characters that differ between the two strings)
  - edit distance (number of copy/insert/delete operations)
- A dot-plot is an informal (but useful) way to display two strings
  - similarities show up as dots arranged on a diagonal
  - use this as a guide to create a sequence of edit operations
- A dynamic programming algorithm systematically fills in a matrix with path costs
  - guaranteed to find the optimal (lowest cost) path in $O(n^2)$ steps
  - a path corresponds to a sequence of editing operations

Skills

- You should be able to
  - find the matches to a pattern found by $k$-mismatch
  - apply a sequence of editing operations to a string $S$
  - given $S$ and $T$ come up with at least one sequence (not necessarily the optimal sequence) that transforms $S$ into $T$
  - evaluate the similarity of strings $S$ and $T$ if you are given a cost function and a sequence of operations
- Extra credit type opportunity (on the problem set):
  - given $S$ and $T$ and a cost function create the matrix, fill it in with costs, show the optimal path