Pattern Matching

Overview of pattern matching algorithms
exact matching
regular expressions
alignment

Great book:

Algorithmic Complexity

- Some notation from CS:
  - A function $f$ is $O(g(n))$ if there is value $n_0$ where $f(x) < c \cdot g(x)$ for all $x > n_0$
  - A function $f$ is $\Omega(g(n))$ if there is value $n_0$ where $c_1 \cdot g(x) < f(x) < c_2 \cdot g(x)$ for all $x > n_0$

Exact Match

- The first set of methods to consider are exact match (substring) searches
- We have a string $T$ (text) of length $m$ and a string $P$ (pattern) of length $n$
- Find all occurrences of $P$ in $T$
  - $P$: ATG
  - $T$: ATATGATACGAATGAC

- Notation used in these slides:
  - $S_i$: the $i$th symbol (character) in string $S$
  - $S_{i,j}$: the substring from $S_i$ to $S_j$

Naïve Algorithm

- The "obvious" way to search for $P$ in $T$:
  - align left ends of $P$, $T$
  - compare all letters in $P$ with corresponding letters in $T$
  - on mismatch move $P$ right one symbol

  $T$: ATATGATACGAATGAC
  $ATG$: (3 comparisons before shift)
  $ATG$: (1 comparison)
  $ATG$: (3 comparisons -- match)
  $ATG$: (1 comparison)

Better Exact Match

- Improvement: slide $P$ to the right by an amount that depends on the symbols in $T$

  $T$: ATATGATACGAATGAC
  $ATG$: (mismatch at 3rd position)
  $ATG$: (slide 2 positions)

  Note $P$ starts with "A", so move $P$ down to the next occurrence of "A" in $T$...

Preprocessing

- The improved algorithm "discovers" information about $T$ as it runs
- It is possible to gather this information before doing any comparisons
- Break the matching algorithm into two phases:
  - preprocessing
  - searching
- Preprocessing stores info about $P$ or $T$
- Searching uses the info to direct the comparisons
Preprocessing (cont’d)

- Algorithms that preprocess P:
  - linear time exact match (example next slide)
- Algorithms that preprocess T:
  - web crawlers – T is the internet, P is user’s search string
  - suffix tree algorithms (following topic)
  - BLAST and other homology search programs – T is the sequence database

Linear Time Exact Match

- The naïve algorithm requires $O(n \times m)$ comparisons
- It is possible to achieve
  - $O(n+m)$ expected behavior
  - Sublinear ($< O(n)$) average behavior
    (when P is long and P, T have few symbols in common)
- The linear algorithms preprocess the pattern P:
  - $O(n)$ steps for preprocessing
  - $O(m)$ (or better) comparisons in searching

Fundamental Preprocessing

- Called “Algorithm Z” in Gusfield
- Common aspects of several advanced methods
  - Knuth-Morris-Pratt (SIAM J Comput 1977)
  - Boyer-Moore (CACM 1977)
- For a string S define $Z_i$ ($i > 1$) to be the length of the longest substring of $S_{1..i}$ that matches the prefix $S_{1..j}$

  $S = AATACGAATG…$
  $Z_7 = 3$
  $B_7 = S_{7..9}$
  $L_7 = 7$
  $R_7 = 9$

Fundamental Preprocessing (cont’d)

- Use “Z-box” method to implement a linear time exact match
- To find a substring of T that matches P:
  - Let $S = P +$ (where $+$ is a symbol not in P or T)
  - Compute $Z_i$ for $i = 2..(n+m+1)$
  - Note $Z_i$ ($i$) for all positions
  - P occurs in T at locations i where $Z_{i+p-1} = n$
- Calculation of all $Z_i$ can be done in linear time; see Gusfield for method

Fundamental Preprocessing (cont’d)

- The ‘Z-box’ $B_i$ is the substring $S_{L_i..Z_i}$ where $Z_i > 0$
- $R_i$ is the rightmost endpoint of all Z-boxes $B_1..B_i$
  (max of (j+$Z_j$-1) over all j $| j \leq i \ AND \ Z_j > 0$)
- $L_i$ is the left end of the Z-box that ends at $R_i$

Suffix Trees

- A data structure known as a suffix tree can implement very efficient string searches
- Build a tree for text T in $O(m)$ steps
- Search the tree for a pattern P in $O(n)$ comparisons
  - The tree can be built without knowledge of P
  - The search for P is independent of the size of T
  - One search returns all occurrences of P
In a suffix tree for a string $S$ of length $m$:
- the interior edges are labeled with substrings of $S$
- each edge leaving a node has a different first character
- there are $m$ leaves, numbered $1..m$
- the concatenation of labels on the edges in the path from the root to node $i$ is $S_{i..m}$
  (i.e. the suffix of $S$ that starts at position $i$)

- Naïve algorithm for building tree: $O(m^2)$
- See Gusfield for $O(n)$ methods

**Exact Match with Suffix Tree**

To find a pattern $P$ in a string $T$:
- Start at root
- while $i < n$
  - Traverse edge labeled $S$ where $S_{1..k} = P_{i..k}$
  - $i += k$

Example: find “ATG”

**Exact Match in Bioinformatics**

- Situation where efficient exact string match is important: Sequence tagged site (STS)
  - DNA sequence of 200-300 bases
  - ends (20-30 bases) are unique
  - EST (expressed sequence tag) is an STS from a coding region

- Goal: given newly sequenced DNA, can we identify which part of the (human) genome it’s from?

**Exact Match in Bioinformatics (cont’d)**

- “Shotgun sequencing”
  - Clone large segment of DNA
  - Break clones at random places
  - Sequence each piece
  - Reassemble
- Use exact match to find pairs of fragments with overlapping ends

**Regular Expressions**

- Recall from last lecture: a regular expression is a pattern that may include metasymbols allowing the pattern to match more than one string
- Example: $[ATCG]$ matches TCA, TCT, TCC, TCG
- The metasymbols [ ] mean “match any one character between the brackets”
Some other metasymbols:

- . match any single character
- X+ match one or more Xs
- X* match zero or more Xs
- X(n,m) match between n and m Xs

In Perl:

- \s match any whitespace character
- \w match any word character (letter or digit)

A common technique for implementing RE pattern matching is to "compile" the RE.

Any RE can be transformed into a graph:

\[ \text{GT} \rightarrow \text{GCT} \rightarrow \text{C} \rightarrow \text{A} \rightarrow \text{T} \rightarrow \text{GAT} \]

Algorithms for REs (cont’d)

Each class of grammar in the Chomsky hierarchy corresponds to a class of automata that recognize strings defined by a grammar:

- Regular expressions (finite state machines)

\[ S \rightarrow L \rightarrow \text{GT} \rightarrow \text{GAT} \]

Regular expression libraries:

- We’ve already discussed Perl’s support for REs:
  
  \[ \text{} = /GT.*AC/; \]

  There are libraries that implement regular expression compilers in C++, Java, and other languages:

  Example: regex (www.boost.org)

  #include <boost/regex.hpp>
  regex intron("GT.*AC");
  if (regex_match(dna, what, intron))…

Aside: Perl REs Can Be Variables

In Perl the regular expression can be a variable:

\$intron = "GC.*AT";
\$dna = "TTGCTTATCC";
print "intron\n" if ($dna =~ $intron);

The Perl interpreter "compiles" the RE string on the fly:

- e.g. the RE might change from one loop iteration to the next

Regular Expressions in Bioinformatics

- Regular expressions can be used to describe patterns in short amino acid strings
  - Domain: building block of protein structure (sheets, helices)
  - Motif: short amino acid structure that defines protein features
- PROSITE database [1992]: collection of protein motifs described by patterns
Motif Example: Zinc Finger

- Transcription factors are proteins used in DNA transcription
- A common motif is the C2H2 “zinc finger”
- Two Cs and two Hs bind to a Zinc atom
- Intervening amino acids form a projection

Zinc Finger (cont’d)

- A regular expression that describes this motif is C\(\cdot\)C\(\cdot\)C\(\cdot\)H\(\cdot\)H\(\cdot\)C\(\cdot\)\[MLIV]\H
- The above expression is from the gref program of the SEALS system [Walker, NCBI]
  - `gref expr human.fa` will print sequences in a Fasta file that match this pattern

Alignment

- A string alignment algorithm can be used to compare two strings that are not 100% identical
- Examples:
  - similarity search: given a database of strings T find members that are “similar” to a string S
  - evolution: what is the degree of similarity between strings S1 and S2? Is the pair (S1,S2) more similar than (S1,S3)?

Alignment (cont’d)

- Rest of this lecture:
  - definitions
  - quantifying similarity
  - basic (hand) alignment method
- Next lecture:
  - dynamic programming algorithm
  - Future: applications based on alignment
    - similarity search (e.g. BLAST)
    - phylogeny inference

Definitions

- An alignment of two strings is a mapping that associates each character of one string with (a) a character in the other string, or (b) no character (a “gap”)
- Example: alignment of two 7-character strings
  - AA-TACCG
  - AAAACCG
- In general the two strings do not have to be the same length before alignment

Definitions (cont’d)

- A global alignment maps each character of each string
  - the result is two equal-length strings (but gaps may occur on either end)
- A local alignment maps substrings of each string
  - useful for comparing motifs or domains
  - best for less similar strings
- A multiple alignment is a simultaneous mapping between sets of three or more strings
  - AA-TACCG
  - AAAACCG
  - AAAACCG
Sources of Dissimilarity

- Before trying to quantify similarity, we should know where the dissimilarities come from
- Examples:
  - measurement errors (e.g. in sequencing items in a DB or in a new specimen to be compared)
  - small scale evolutionary events
    - replacement
    - insertion
    - deletion

Edit Distance

- A simple method for measuring similarity uses an edit distance
- Use four basic string operations that operate on a single string:
  - match (M)
  - replace (R)
  - insert (I)
  - delete (D)
- An edit transcript is a sequence of edit operations

Edit Distance (cont’d)

- Example (Gusfield): the following transcript, when applied to $S_1$, yields $S_2$:

  Transcript: RIMDMDMMI
  $S_1$: v-intner-
  $S_2$: wri-t-ers

Alignment as Optimization

- The edit distance between two strings is the number of edit operations (R,D,I) in a transcript that transforms one string into the other
- The concept of edit distance allows us to specify the goal of an alignment algorithm:
  - Align strings $S_1$ and $S_2$ by finding a transcript that transforms $S_1$ into $S_2$ with a minimum edit distance
- Looking ahead: dynamic programming is an optimization method well suited to this application.

Cost Functions

- Edit distances can be adapted to model more realistic situations
- Examples:
  - if insertions and deletions are less likely than mismatches (e.g. point mutations vs cross-overs) assign a different weight to I/D vs R
  - starting a gap may be more expensive than extending a gap
  - individual substitutions may have different weights (e.g. in proteins $S_i$ T more likely than $S_i$ Y)

Dot Matix Method

- To transform $S_1$ (n chars) into $S_2$ (m chars) using a simple edit distance with $I = D = R = 1$ and $M = 0$:
  - draw a n x m matrix
  - label rows with characters of $S_1$, columns with characters of $S_2$
  - enter a dot in each cell where the row and column labels are the same

Example: matrix for ATTCA vs ATGTA

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>T</th>
<th>C</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTCA</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>ATGTA</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
**Dot Matrix (cont'd)**

- Find a minimum weight path from the upper left to the lower right:
  - Diagonal: M or R
  - Vertical: D
  - Horizontal: I
- Note there may be several equal-weight paths.

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

<table>
<thead>
<tr>
<th>A</th>
<th>T</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M</th>
<th>M</th>
<th>R</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M</th>
<th>M</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
</tr>
</tbody>
</table>

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

Note: The path `AT-GT-A` and `AT-TCA` are equal in weight.