6.085/6.895 Recitation 3 - September 22, 2006

1 Announcements

- Problem Set 1 is due Tuesday 8pm
- A submission box will be in the elevator bank of the Gates tower 5th floor
- Office hours will be Tuesday 4-5pm (next week only)
- #3 depends on gnuplot. Try not to wait until the last minute to make sure it works on your computer. If all else fails, use Athena.

2 Longest common substring using suffix trees

A naïve algorithm to determine the longest common (contiguous) substring of two strings of length $m$ and $n$ runs in $O(mn)$ time. Using suffix tree algorithms, we can solve the same problem in $O(m+n)$ time. For this problem, we use a generalized suffix tree, which stores the suffixes of several different strings in one tree. The leaves of the tree (corresponding to the end of each suffix) have a pointer to the sequence(s) and position(s) where that suffix appears. Here’s a generalized suffix tree on $S\# = \text{CAGCA}$ and $S\$ = \text{AGCAGT}$.

![Generalized Suffix Tree Diagram]

We can construct the generalized suffix tree in $O(m+n)$ time using a concatenation trick. We append unique termination characters to the strings, and then concatenate them together. We can then apply any suffix tree construction algorithm to this artificial string to obtain the generalized suffix tree. A postprocessing step is required to truncate the suffixes at the termination characters, which are guaranteed to occur on a leaf edge. This just requires a traversal of the tree, so it’s still linear time. (Of course, the suffix tree algorithms can be modified to directly construct generalized trees.)

Given the generalized suffix tree for the two strings, the longest common substring begins at the root and ends at the deepest node below which suffixes of both strings still occur. By “deepest” we mean according to the total string length along the path. To find the node representing the LCS, we need to mark up each node in the tree with its “depth” and whether suffixes of both strings are among its children, and then search the tree. All of this can be accomplished with a few depth-first-search traversals, so the overall runtime is $O(m+n)$, including building the suffix tree.
3 All about BLAST

In the BLAST (Basic Local Alignment Search Tool) algorithm, we preprocess the database by indexing all occurrences of all W-mers, and store a lookup table mapping a W-mer to all the positions in the database where it occurs. To process a query, we scan the query sequence Q. For each W-mer in Q, we use the precomputed lookup table to find the locations of all its occurrences in the database. We then look at each of those locations to determine if we can extend the W-mer match into a statistically significant “hit”.

NCBI BLAST

In the actual implementation of BLAST from NCBI, this strategy is reversed. For each query, NCBI BLAST hashes Q and scans the entire database against it. To understand why, consider the practical size of the index. To represent the 23 amino acids, we would need a minimum of 5 bits, so practically we need 1 byte per letter. However, the index needs to store about $n$ pointers to the $n$ unique locations in the database. So the size of the index must be $O(n \lg n)$ bits, and since $n >> 23$, the index would be several times the size of the database itself!

Furthermore, consider that, in 1990 when BLAST was first developed, there would be no hope of storing a large database in main memory, and the original BLAST strategy would be in trouble because random access to the hard disk (to load the region surrounding each W-mer match) is extremely slow. The BLAST strategy as originally envisioned only makes sense if you have low-latency access to the index and database.

BLAT

Today, reasonably large databases can actually be stored in RAM, and there is a tool called BLAT (BLAST-Like Alignment Tool), written by Jim Kent at UCSC, which uses the original database-hashing strategy. BLAT uses some heuristics to keep the size of the index under control, such as ignoring instances of “over-occurring” W-mers that appear too often in the database to be useful, thus reducing the number of pointers the index must store. BLAT can index the human genome in about 1GB of memory. BLAT queries are much faster than NCBI BLAST once the database has been indexed. Still, NCBI BLAST is needed for very large databases.

Other Flavors of BLAST

Many different extensions to the original BLAST algorithm have been developed. Today BLAST can produce alignments with small gaps, by using dynamic programming in the neighborhood of multiple W-mer matches. BLASTX and TBLASTX are able to translate DNA sequences into protein sequences on-the-fly. This lets you search protein sequences against DNA databases and vice versa. In DNA-DNA queries, it acts like a very special comb on protein-coding regions that increases the sensitivity of the search for a given W.