CSE182 lecture 4 notes &questions

Vineet Bafna

October 5, 2006

1 Notes

Recall that we are interested in computing local alignments of a query string of length n against a subsequence from database. Certainly, we can apply the smith waterman (local alignment) algorithm treating the entire database as a single string of length m, and computing the optimum local alignment. See Problem **??**. The number of steps, from earlier arguments is $O(nm)$. As a rough calculation, suppose,we were querying the entire human genome, against the entire mouse genome implying that $n \simeq m \simeq 3 \times 10^9$. An full-blown local alignment would require $\sim 10^{19}$ steps. Even with a fast computation of 10^{10} steps per sec., we would need $10^9 s$ (31 CPU-years) to do the computation. It is worth considering if we can do better.

A general approach to this problem is through *database filtering*. Think of a database filter as a program that rapidly eliminates a large portion of the database without losing any of the similar strings. For example, suppose we had a filter that runs in time $O(m)$ (independent of the query size), and rejects all but a fraction $f << 1$ of the database. Then, by aligning the query only to the *filtered sequence*, the total running time is reduced to $O(m + fmn)$. Suppose, we had a filter with $f = 10^{-8}$. then, the total running time for the previous query would have $\sim 10^9 + 10^{-8}10^{19} \simeq 10^{11}$ steps. At 10^{10} steps per second, we could do the query in 10 secs. This is the idea that is pursued in Blast.

2 Basics

Let us start with the assumption that the database is a random string over the characters $\{A, C, G, T\}$, each occurring independently with probability 0.25. Next, assume that the query is a string of k ones, given by

$$q = \underbrace{111\ldots111}_{k}$$

We are interested in computing

$$Pr(q \text{ is contained in a database substring})$$

As it turns out, this is somewhat difficult to compute because of the dependencies between occurrence at different positions. However, given a fixed position i in the database,

$$Pr(q \text{ occurs at position } i) = \left(\frac{1}{4}\right)^k$$

Therefore, the *expected number* of occurrences of q = $n\left(\frac{1}{4}\right)^k$. Why?

2.1 Basic probability

To see this, define an indicator variable X_i for all positions $1 \leq i \leq n$.

$$X_i = \begin{cases} 1 & \text{if } q \text{ occurs at position } i \\ 0 & \text{otherwise} \end{cases}$$

Then, number of occurrences of q is given by

$$\sum_{i=1}^{n} X_i$$

The expected number of occurrences in a random database is simply

$$E(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} E(X_i) = \sum_{i=1}^{n} 1 \cdot \frac{1}{4^m} + 0 \cdot \left(1 - \frac{1}{4^k}\right) = \frac{n}{4^k}$$

For modest values of m, this number can be quite small. For $n = 10^7, k = 12$, $\frac{n}{4^k} < 1$. So what does this all mean? Roughly, it means that the given a *small* fixed string, the probability that we will find it just by chance is quite small.

3 The Pigeonhole principle

The Pigeonhole principle simply says that
"If there are n pigeons, and $n + 1$ pigeonholes to put them in, some hole has no pigeons in it."

While seemingly obvious, the argument can be used to prove many non-obvious statements. Here is how we will use it. Suppose, we had a query of length $m = 100$, and were looking for a match to the database, with better than 90% identity (Assume it is a gapless match for now). There are at most 9 mismatches. Partition the query into 10 parts, each of size 10.

When we match q to a database string with 9 or fewer mismatches, each mis-match (pigeon) can fall into one of the partitions (pigeonholes), and at least one of the partitioned sub-strings must match exactly. This argument is conservative, and can be strengthened.

Assume that the mismatches occur at random, and we are interested in a database sequence with fewer than cm mismatches ($c < 1$). Then, the probability that a specific position has a mismatch is $\leq c$. For all i

$$Pr[\text{ The } k\text{-mer starting at position } i \text{ in the query matches exactly}] \geq (1-c)^k$$

There are $m - k \simeq m$ such k-mers, and we only need 1 to match

$$Pr[\text{ Some } k\text{-mer in the query matches exactly}] \geq 1 - \left(1-(1-c)^k\right)^m$$

Using the approximation $e^{-x} \sim 1 - x$, we have

$$Pr[\text{ Some } k\text{-mer in the query matches exactly}] \geq 1 - e^{-e^{-ck}m}$$

To see what this looks like, choose $m = 100, k = 10$, and $c = 0.2$ (implying an 80% identity), for these values, the probability of getting a matching k-mer is very close to 1. Thus, we have shown the following:

- The probability that some k-mer from the query matches some position in the database is quite small. The expected number of such matches are $\sim mn\frac{1}{4^k}$.
- if a sequence in the database matches the query with fewer than cm errors, the probability that some k-mer in the query matches exactly is very high, $\sim 1 - e^{-e^{-ck}m}$.

3.1 Why is Blast fast?

Instead of doing the local alignment computation for the entire $m \times n$ matrix, we look for exact matches to the $\sim m$ query keywords (each of size k) in the database. When an exact match is found, the region around the match is aligned. Do some calculations to see how this might improve the speed. Note that the exact match computation can be done in $O(n)$ time. We leave that to subsequent lectures.

4 P-value computation

Interestingly, the same principle of computing exact matches in a random database is also used in the P-value computation. First, what is a P-value? Suppose your search leads to a match in the database with score S. The P-value is the probability that a match of similar or better 'score' could be obtained just by chance search of a large database.

The most obvious way to compute P-value is by a direct estimate. Compute a large number of random alignments. The fraction of times you score S or better is an estimate. However, this approach is inefficient because you need a large number of trials. To get around this, we try to approximate the distribution by a known one. Then, the P-value of the distribution is obtained by computing the parameters of the distribution.

It was initially assumed that Blast scores were normally distributed. This led to the Z-score. For any match in the database, permutations of the database match are aligned to the query (to simulate a random database with the same nucleotide distribution), and the mean and std. deviation μ, σ are computed. The Z-score is $\frac{X-\mu}{\sigma}$ is the normally distributed with mean 0, and s.d. 1. The P-value can now be looked up from a table.

Unfortunately, this idea turns out to be incorrect. Karlin and Altschul show that at least for ungapped alignments, the distribution of scores follows an exponential, and not Normal distribution. To understand this, suppose we wanted a perfect match to a substring of length k of the query. If the probability of a character match is p, the probability of a match of length k is p^k. By earlier arguments, the expected number of matches (Why?) is

$$\Lambda = (n-k)(m-k)p^k = nmp^k = nme^{-k\log(\frac{1}{p})}$$

It can be shown that the *number* of matches of length k is Poisson distributed. The probability that there are u matches of length k is

$$Pr(u) = \frac{e^{-\lambda}\lambda^k}{k}$$

The P-value is now given by

$$Pr(u > 0) = 1 - Pr(0) = 1 - e^{-\lambda}$$

Thus, if our score function was 1 for match, and $-\infty$ for mismatches and indels, we could compute the P-value exactly. Karlin and Altschul show that other score functions also have the same property. Thus, the expected number of hits of score S is given by

$$E = Kmne^{-\lambda S} = mn2^{\left(-\frac{\lambda S - \ln(k)}{\ln(2)}\right)}$$

Here, the values K, λ are computationally determined to 'fit' the Poisson distribution. The term $\frac{\lambda S - \ln(k)}{\ln(2)}$ is called the *bit-score*. The P-value, or the probability that a 'random' score S exceeds x is given by

$$Pr(S \geq x) = 1 - e^{-Kmne^{-\lambda x}}$$

For small P-values, the E value, and P-values are very similar.

5 Questions

1. The recurrence for the time used by space saving approach is $T(nm) = cnm + T(\frac{nm}{2})$. What is the total time taken, and how does it compare to the time taken by a non space saving algorithm?

2. Why is the default word-size 11 for DNA, but only 3 for proteins?

3. What is the pigeonhole principle?

4. Toss a coin many times: If it is HEADS, and the previous time was HEADS too, you get a dollar. Otherwise you lose a dollar. What is the money you expect to get after 100 tosses?

5. Consider a DNA query string of length $m = 100$, and a DNA database of randomly chosen nucleotides of length $n = 10^7$. What is the expected number of exact database matches of k length substrings from the query? Program this and check computationally if your theory matches the experiment. How do the expected number of matches change with increase in k (Try $k = 3 \ldots 15$).

6. Repeat the same computational experiment as before. However, instead of looking at exact matches, give a score of 1 for a match, and -2 for a mismatch, with no indels allowed. Compute the number of 'hits' as you change the threshold score. Does the behaviour change too much from the exact match case? Note that this is the intuition that allows us to give P-values, and E-values to the score.

7. Suppose you run Blast but do not get any hit back? What are the possible reasons, and how will you change the input parameters so as to improve the likelihood of getting a hit.

8. Consider a DNA query string of length $m = 100$, and a DNA database of randomly chosen nucleotides of length $n = 10^7$. If you get a hit to your query with a *bit-score* $S = 70$, what is the E-value of your hit?

9. Suppose a query of length 100 is homologous to a database sequence with bit-score 80, and E-value of 10^-5. What is the E-value of a query of length 400, and bit-score 95?

10. Suppose you got a hit back with score $S = 89$. Repeated trials with a shuffled query string gives you the following numbers:

    $$85, 15, 90, 30, 25, 70, 65, 77, 45, 69, 61, 72$$

    estimate a p-value for your hit. Suggest ways in which you can improve your confidence into the significance of your hit.