Concise Genome Indexing by Decimated Hashing with Random Collision Resolution

William H. Press

September 4, 2011

The human genome is about $N_{hg} = 3.08 \times 10^9$ bases long. The task at hand is perhaps the most basic one in genomics: Given a short "test" sequence of, say, 50 to 100 bases from somewhere in the genome, find its location.

In practice, there are other considerations as well. Is the test sequence free of errors, or do we need an algorithm that is robust against errors? Are we doing only $\lesssim 10^6$ lookups, are do we need the speed to do $\gtrsim 10^9$? Are we limited to the memory footprint of 32-bit machines (that is, $\lesssim 3$ GB), or can we afford 16 or 64 GB of memory?

Here, we first consider the case where high speed is needed, memory footprint must be small, and the test sequence is (relatively) error-free.

Naive Multi-Hashing

For sheer lookup speed, it is hard to do better than to hash the whole genome into a hash multimap structure. Specifically, starting at each base along the genome, we take n_h sequential bases as an index key and store the starting base address (that is, offset from the beginning of the genome) in a hash multimap memory, which allows the recovery of an arbitrary number of stored elements from a single key.[1] If we take $n_h \sim 25$, then most keys, but not quite all, are unique in the human genome.

To do a lookup on a test sequence of length $M \sim 100$ (say), find all stored addresses of $n_k = \lceil M/n_h \rceil \sim 4$ keys that span the test sequence, and subtract from each returned address the offset of its key in the test sequence. The desired address or addresses are the ones that occur in all n_k keys. In the absence of sequence errors, this method returns all exact matches and no others. It is very fast, < 100 operations per lookup, including the hash function.

The problem is memory footprint. A fairly tight multihash implementation[1] needs ≈ 30 bytes per stored value, amounting to 100 GB for the human genome. We want to achieve roughly the same functionality in about 100 times less memory footprint.

Other Methods

[Mention Bowtie, etc., here.]

Decimation

A trivial observation is that we don't need to store an address for every base in the genome. Instead of constructing an index key at each base, we can contemplate decimating by d, that is, storing only one in every d addresses.

Of course, at lookup time, we must now probe the hash memory at d consecutive offsets to access a single stored key. How do we know which of these d has the right phase? Easy: Only the correct phase will produce addresses in common among the n_k probes, and it is computationally easy to find these common addresses.

So decimation by d is a good thing, but not, by itself, good enough to get a factor 100. Decimating by that much would leave many sequences not indexxed by even a single key. And we have seen that we need at least two keys to mutually validate each other.

Random Collision Resolution

Let's come at this a little bit differently. Suppose we can afford the memory for a single array of N addresses into the genome, therefore amounting to $4N$ bytes. Call this the "N-array". For a 1 GB footprint, we have $N \approx 250 \times 10^6$.

We now hash N_{hg}/d index keys into the N-array. The mean coverage (mean number of collisions per array location) is

$$\lambda = \frac{N_{hg}}{Nd} \tag{1}$$

Suppose that whenever m addresses would collide into a single element in the N-array, we store exactly one of the m with uniform probability. (We explain below how to do this.)

To see what is going on, let us make the radical assumption that the number of collisions in each cell is Poisson distributed with mean λ. Consider now the lookup of a particular index key from a particular test string. This key hashes into some particular element of the N-array. We call that element "useful" if it contains (up to the known small offset) the address of our test string. The probability of this is

$$P(\lambda) = \left(\sum_{m=1}^{\infty} \frac{1}{m} \frac{1}{m!} \lambda^m e^{-\lambda} \right) \Big/ \left(1 - e^{-\lambda}\right) \tag{2}$$

Here, the numerator can be understood as the expectation over the Poisson probability distribution of $1/m$. That is, if an element is hashed from m keys, then there is only $1/m$ chance that it contains the address that we need. The denominator renormalizes the Poisson distribution to account for the fact that elements pointed to by no keys at all are not relevant to the expectation.

Although larger $P(\lambda)$ is generally good, the actual figure of merit is the density of useful starting positions in the test string – that is, the density of

starting positions that hash to useful N-array elements. This has an additional dependence on λ through d (equation (1)),

$$\frac{P(\lambda)}{d} = \left[\frac{\lambda N}{N_{hg}}\right] P(\lambda) = \frac{N}{N_{hg}}\lambda P(\lambda) \equiv \frac{N}{N_{hg}}\rho(\lambda) \tag{3}$$

with the proviso that the term in square brackets must be less than 1, since $d < 1$ makes no sense. In other words, we may choose λ no larger than N_{hg}/N.

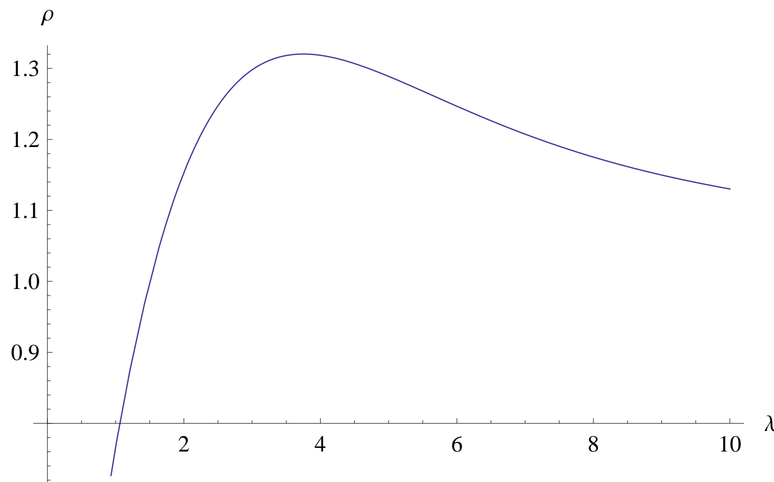

Figure 1: Dimensionless density of useful cells ρ as a function of coverage parameter λ.

Independent of the values of N and N_{hg}, we see that the optimum occurs for the value of λ that maximizes the function $\rho(\lambda)$, which is plotted in Figure 1. The function has a maximum of about 1.320 at $\lambda = 3.75$. At the easier-to-remember value $\lambda = 4.$, the function value is $\rho(4.) = 1.319$, hardly different, with $P(4) = 0.324$.

It is perhaps surprising that the optimal value of λ is so large, implying ≈ 4 collisions per N-array element. Let's see how the numbers work out. For $N_{hg} = 3 \times 10^9$, $N = 250 \times 10^6$, $\lambda = 4$, we have from equation (1),

$$d = \frac{N_{hg}}{4N} = 3 \tag{4}$$

so we decimate by 3 (compute index keys every third starting base). The mean distance between useful keys is

$$\frac{d}{P(\lambda)} = \frac{N_{hg}}{\rho_{\max} N} = 9.23 \tag{5}$$

We might ask (for example) what is the length of test sequence needed so that there is less than 1% probability of failing to get 2 or more useful keys:

$$e^{-\mu} + \mu e^{-\mu} \leq 0.01 \Rightarrow \mu \geq 6.6 \tag{6}$$

So $9.23 \times 6.6 = 60$ bases estimates the required length of test sequence.

Constructing the Index

As implied above, the index is constructed by marching through the genome with stride d, hashing genomic addresses into the N-array. The only detail requiring explanation is how, when m addresses will collide in one location, a random one is chosen. This requires an additional byte array of length N, initialized to zero, which keeps track of the number of keys that have hashed to each location in the N-array. When the mth such key is encountered, it is allowed to overwrite the N-array location with probability $1/m$. It is easy to see by induction that this results in an equiprobable distribution without needing to know in advance the final value of m. (Collision counts are arbitrary capped at $m = 255$, but by this value there is essentially no contribution to the utility.)

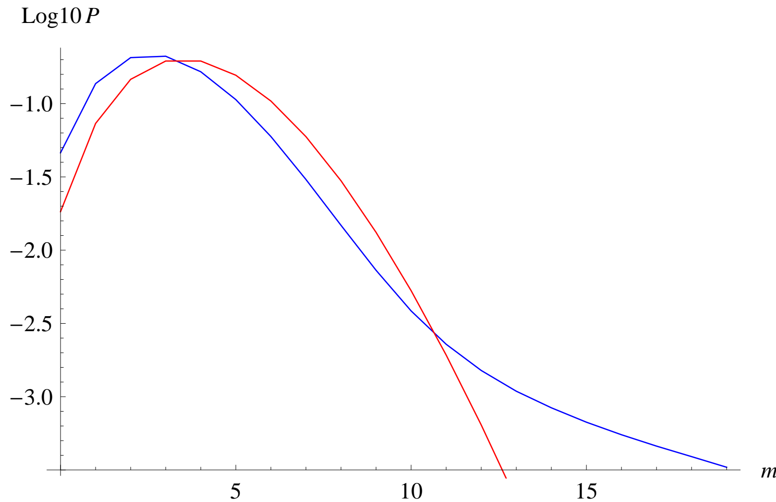

Figure 2: Empirical distribution of number of hash collisions for the human genome with $\lambda = 4$ (blue), and Poisson distribution with the same λ (red).

Beyond the Poisson Assumption

We have used the Poisson model to estimate an optimal value $\lambda \approx 4$, implying (for the other parameters above) $d = 3$. For the actual human genome, as a byproduct of constructing the index with an particular value of index key length n_h, we can evaluate the empirical analog of equation (2),

$$P_{\text{obs}} = \left(\sum_{m=1}^{\infty} \frac{1}{m} P_m \right) \Big/ (1 - P_0) \qquad (7)$$

where P_m is now the *o*bserved probability that an N-array element has m collisions. For genome assembly hg18, with $n_h = 20$, we find $P_{\text{obs}} \approx 0.45$. This is significantly better than the Poisson model's value best value 0.32. The explanation is that hg18 has more keys with high multiplicity than the Poisson model. Since there is little utility in these anyway ($1/m$ being small), overpopulating them results in lower multiplicities for the other keys, hence larger average $1/m$. This effect is illustrated in Figure 2.

Implementation Tests

[Describe tests.]

References

[1] Press WH, Teukolsky SA, Vetterling WT, and Flannery BP (2007) *Numerical Recipes: The Art of Scientific Computing*, Third Edition (Cambridge University Press) §7.6.4.