

Lab 2

February 19, 2008

Due Date: February 26 (One Week)

The goals of this lab are:

- Provide an exciting application area and actual data to see how data structures can really make a difference in our ability to solve important real-life problems. In particular, we study the application of comparing genomic DNA sequences to find important regulatory sites.
- Show an application where hash tables are used to create a more efficient algorithm.
- Introduce Bloom filters as way to improve the efficiency of a standard Set, Mapping, or Bucket Mapping implementation.
- Highlight the importance of being prepared to learn about a new data structure needed for an application.

1 Genomics Background

A DNA sequence is a string of characters, called *bases*, from the alphabet $\{a, c, g, t\}$. Genomic DNA encodes a large collection of *features*, including *genes* and *regulatory sites*. All DNA is subject to *mutations* that alter its sequence over time. However, functional sequence features like genes and regulatory sites are more resistant to mutation than DNA that doesn't code for anything (because natural selection usually kills off organisms with too many mutations in these features). We can therefore find these features by comparing DNA sequences from two different organisms and see which parts of the sequences have remained similar to each other.

Abstractly, we are given two long strings s_1 and s_2 of characters, and we want to find short substrings (parts of the features) common to s_1 and s_2 . In general, the common substrings might not be exactly the same because even functional sequences mutate over time; however, we often find that s_1 and s_2 still exhibit exactly matching substrings of at least 10 to 15 characters in the neighborhood of their shared features. By detecting these exact substring matches, we can find the most likely locations of the shared features in s_1 and s_2 and can then apply more sensitive but more expensive similarity search algorithms only at those locations.

Naively, we could find k -mers (substrings of length k) common to s_1 and s_2 by comparing every k -mer from one sequence to every k -mer from the other. Such an approach would take worst-case time $\Theta(|s_1| \cdot |s_2|)$, which is unacceptable because interesting DNA sequences range from thousands to billions of characters in length. (Your own genome is about three billion characters long). Fortunately, there is a much better approach.

Call s_1 the **corpus string** and s_2 the **pattern string**, and assume we are searching for common substrings of length k . We first construct a set S of every k -mer in the pattern string, remembering all offsets in the pattern string where it occurs. Then, for each k -mer in the corpus string, we check whether it occurs in the set S ; if so, we have found a match between pattern and corpus. If S supports constant-time insertions and lookups (e.g., as in open addressing), we can process the entire pattern and corpus in time $\Theta(|s_1| + |s_2| + x)$ where x is the number of common substrings actually found. In general, this time cost is much lower than the cost of the brute force algorithm. Fast substring matching based on hashing therefore forms the core of many of today's high-speed biosequence matching algorithms.

2 Bloom Filters

A Bloom filter, introduced by Burton Bloom in 1970, is a space-efficient probabilistic data structure for a limited form of the Set ADT that only supports adding an element into the set, and testing whether an element is a member of the set. False positives (saying an element is in the set when it is not) are possible, but false negatives (saying an element is not in the set when it is) are not. The more elements that are added to the set, the larger the probability of false positives.

The primary data structure within a Bloom filter is an array of m bits (Booleans). There must also be r different (independent) hash functions defined, each of which converts the hashcode of an element to an integer in $\{0, 1, \dots, m-1\}$. For the purposes of this lab, you will use the following simple (but not ideal) hash functions. The i^{th} hash function ($i = 0, \dots, r-1$) is:

$$\text{slot}_i(x) = (\text{hash}(x) + i \cdot \text{stepHash}(x)) \% m$$

where x is the hashcode of the element, and `hash` and `stepHash` are the hash functions in the provided open addressing bucket mapping implementation.

To test if an element is in the set, the slot computed by each of the r hash functions is checked, and `true` is returned exactly when all r of these slots are `true`. So if any of these r slots are `false`, then `false` is returned. Observe, that if an element is in the set then the r slots (defined by the r hash functions) will be `true`, so the check for membership is guaranteed to return `true`. Now consider when membership is tested for an element not in the set. If m and r are large enough then intuitively it is unlikely that the r slots defined by the element would just happen to all be set to `true` by other elements that were inserted. However, it can happen. There will be some false positive errors when an element is incorrectly reported by the Bloom filter to be in the set.

The *false positive error rate* is the fraction of elements that are not in the set, but reported by the Bloom filter as being in the set. More specifically, let x be the number of substrings that are correctly classified as negative by the Bloom filter, and let y be the number of substrings that are incorrectly classified as positive by the Bloom filter. The false positive error rate is $y/(x + y)$. It can be proven that the probability of a false positive error is $(1 - (1 - \frac{1}{m})^{rn})^r$. This probability of error is minimized when $r \approx 0.7m/n$. For this choice of r the probability of a false positive error is approximately $(0.6185)^{m/n} = (0.6185)^{1/\alpha}$.

There are two important advantages of Bloom filters. First, they are very space efficient. A Bloom filter with a 1% false positive error rate and an optimal value of r requires only about 9.6 bits per element regardless of the size of the elements. Also the time to either add elements or to check whether an element is in the set is a fixed constant, $O(r)$, completely independent of the number of items already in the set. The second advantage is that in a parallel implementation a Bloom filter really shines because the r table slots can be computed and accessed in parallel since the task to perform for them is completely independent. In addition, Bloom filters allow an explicit trade-off between space and the error rate. If deletion is required then **counting filters**, which maintain an integer versus a bit per table entry can be used.

3 Using Bloom Filters to Speed-Up Genome Similarity Searches

In this section, we describe the way in which Bloom filters in conjunction with specialized hardware can be used to speed-up the task of finding matching subsequences between two genomes. There are applications in network routing that use Bloom filters in this same manner. To dramatically improve the speed of these similarity searches, a specialized chip is used that has a very limited amount block RAM which is extremely fast to access. Off of this specialized chip (off-chip) there is a much larger quantity of S-RAM which is significantly slower to access than the block RAM. Another important feature of our genomics application is that the majority of the searches that occur are unsuccessful searches. By implementing (in hardware) a Bloom filter on the specialized chip, most of the searches can be filtered out and do not need to be performed in the off-chip hash table that is used to map each k -mer in the pattern string to the set of offsets where it occurs. Thus a bucket mapping is the appropriate data structure choice for the off-chip portion of this design.

So you will keep two completely independent data structures, the Bloom filter (that would be on the specialized hardware in the real implementation) and an open addressing bucket mapping from the k -mer to the set of offsets where that k -mer begins in the pattern string (that would be off-chip). More specifically, all k -mers in the pattern string are inserted in both the Bloom filter (as the element) and the bucket mapping (as the tag). Since the number of searches is much larger than the number of elements inserted, this cost is acceptable. During a search, any k -mer that is reported by the Bloom filter as not being in the set is known to be an unsuccessful search, so no further action is needed. Only when the Bloom filter reports that an element is in the set, is a search in the off-chip bucket mapping made. There will be some false positive errors when an element is incorrectly reported by the Bloom filter to be in the set. Such an error will cause an unsuccessful search to be performed in the off-chip bucket mapping. However, if designed properly, the Bloom filter will have very low false positive error rates which means that most of the searches performed on the off-chip bucket mapping will be successful searches that will return all locations in the pattern string that hold the desired substring.

4 Provided Data Sets

We have provided you with three data sets. The first includes some short sequences that you can use to help debug and test your implementation. For this data set use a match length of 3.

The second data set includes sequences from the beta globin locus control region. This part of the genome encodes several different forms of the hemoglobin protein used to transport oxygen in red blood cells. The locus control region is part of a mechanism that controls which of these different forms of hemoglobin is produced at

each stage of a person’s life (fetal, neonatal, and afterwards). The two sequences in this data set come from the human and mouse genomes. Note that the positions of most of the matches fall into distinct clusters, which correspond to individual “hot-spots” within the region that contain the conserved regulatory sites. For the second data set use a match length of 25.

The third data set holds sequences come from the `mnd2` locus. The `mnd2` gene encodes a protein important to correct functioning of the nerves that carry impulses from the brain to the muscles. In mice, mutations in this gene cause “motor neuron degeneration 2” syndrome, which is closely related to certain kinds of muscular dystrophy in humans. The two sequences in this data set come from the human and mouse genomes. For the third data set use a match length of 18.

5 Your Assignment

You have been provided with an implementation of a limited version of a bucket mapping implemented that uses open addressing (`LimitedOpenAddressingBucketMapping`), and an application program (`SequenceMatch`) that performs the basic genome similarity search without the use of a Bloom filter.

For your Bloom filter use 0.25 as the default value for the target load. As with open addressing, you also want to allow the application to provide a value for the target load in the constructor. In Part III you are asked to perform tests with target loads of 0.25 and 0.125.

You can select the size, m , of the Bloom filter, exactly as it is done for the provided open addressing bucket mapping. More specifically, for this genomics application, an estimated value (really an upper bound) `capacity` for the number of tags to insert is provided to the constructor, and also a desired load is provided (as an argument or using the default value). Recall that the for a bucket mapping actual load $\alpha = t/m$ (when $d = 0$) where t is the number of tags and m is the hash table size. Solving for m yields that $m = t/\alpha$. So you should allocate the size m for the underlying array for the Bloom filter to be the nearest power of 2 that is at least as big as `capacity/load` since `capacity` is the given upper bound for t and `load` is the desired value for α .

You do **not** need to include any resizing of the Bloom filter table in this lab. In fact, with just the Bloom filter itself, it would not be possible to resize the table, since the elements are not stored. The only way that you could resize the Bloom filter, would be to have access to the open addressing bucket mapping so that you could iterate through all elements and reinsert them into the Bloom filter.

Part 1: Study Empirical Performance of Open Addressing (10 points)

Each time you examine an entry in the underlying hash table used within the bucket mapping, it is called a *probe*. For an unsuccessful search, do not forget the probe that ends the search (i.e., the one that ends at an empty slot). In this part you will empirically measure how the number of probes required by open addressing varies with the load factor for the second and third data sets. In particular, you are to compute the average number of probes over all calls of `mapping.get` within `SequenceMatch`. To compute the number of probes, you will want to add an instance variable to the provided `SimplifiedOpenAddressingBucketMapping` to count the number of probes. You’ll also want to add a public method to get the value of this counter, and perhaps also one to reset it to 0. You will also want to make some small modifications to `SequenceMatch` to compute the average number of probes made by the calls to `mapping.get` and to compute the actual load for the hash table. Remember that for open addressing, the load is the fraction of the hash table slots that are in use. So for a bucket mapping, the load is the number of tags over the hash table size.

For the second and third data sets (with the match lengths given in Section 4) compute the average number of probes for specified loads (e.g., the parameter to the constructor) of 0.10, 0.25, 0.5, 0.75, and 0.95. You should also report the actual load, and the expected number of probes in an unsuccessful search based on the actual load (as predicted by the theory presented in class). Observe that the overwhelming majority of the probes are unsuccessful so this should be a good estimate. (If desired, you can compute the percentage of the searches that are successful and use this to adjust the estimate.)

Part 2: Implement a Bloom Filter and Apply it (30 points)

For this part you are to implement a Bloom filter (as a new class), and then modify `SequenceMatch` to use the Bloom filter as described in Section 3. While in the real application the Bloom filter is implemented on a multi-processor chip, your implementation will compute the r hash functions, one at a time (sequentially). To help confirm that your implementation is correct you are required to output how many substrings were filtered

out, and the number of false positive substrings that passed through the filter. As an example, for the second data set when 3 hash functions were used by the Bloom filter, the matches are:

```
8530 11918 tgagtcagctgaggcttagggtgt
8531 11919 gagtcagctgaggcttagggtgtg
8532 11920 agtcagctgaggcttagggtgtgt
8533 11921 gtcagctgaggcttagggtgtgtg
10754 4949 aaacaaacaaacaaacaaacaaaca
10755 4950 aaacaaacaaacaaacaaacaaaca
```

It should report that there were 18343 substrings filtered out and the number of false positives was 1437.

Part 3: Study the Behavior of the Bloom Filter (10 points)

In this section you are to explore the properties of the Bloom filter as you change both the desired load (and thus the table size m), and the number of different hash functions used.

For the third data test (with a match length of 18) test you should create two graphs (using Excel or whatever program you'd prefer), one when the specified load is 0.25 and the second when the specified load is 0.125. In these graph the x -axis should be the number of hash functions (r) used by the Bloom filter for r ranging from 1, 2, ..., 10. The y -axis should be the false positive error rate (with the range shown being roughly that which occurred for your tests).

To help us check your computation of the false positive error rate, report the false positive error rate you computed when 5 hash functions are used when the load is 0.25.

Finally, you are to look at how well does the theory that the optimal choice for the number of hash functions is roughly $.7/\alpha$ (where α is the actual load) matches what you found? For this optimal choice for the number of hash functions, how well does $(0.6185)^{1/\alpha}$ compare to what you saw empirically?

What to Submit

You should have a cover sheet (with all portions filled in including your name and signature) stapled to the front. If desired, you could submit Parts 2 and 3 without submitting Part 1.

1. For Part 1 please describe in English how you computed the actual load factor. You do not need to submit any code for this part. Please save your code so that it can be sent by email upon request if we feel a need to see it.

For the 10 experiments you were asked to run you should report the actual load, the expected value for the number of probes per search under the assumption that all searches all unsuccessful, and the actual value for the average number of probes taken over all calls to `get`.

2. For Part 2 you should submit the modified version of `SequenceMatch` with the added or changed portions clearly highlighted or marked. You should also submit your Bloom filter class. Finally, submit the output for all the three data sets when 3 hash functions are used (and using the substring length specified with the data sets). You should report how many substrings were filtered out and how many false positives occurred.

If your implementation is buggy and you were unable to fix the bugs, please tell us what you think is wrong and give us a test case that shows the error. You'll lose fewer points for a wrong answer if you indicate that you know it's wrong (and, if possible, why it's wrong).

3. For Part 3, you should submit the two graphs described. (If you modify `SequenceMatch`, or create a different driver for this purpose, to output the needed data as a comma separated file, you can then load that into Excel without having to retype anything.) Also to help us check for correctness, you are required to state the false positive error rate for the third data set when the specified load is 0.25 and 5 hash functions are used. Finally, you should include a brief discussion about how well the theory summarized in this handout about the optimal choice for the number of hash functions, and the expected false positive rate compare to what you found in practice for these experiments.