Metrics and visualisation for crime analysis and genomics

Laros, J.F.J.

Chapter 8

Selection of DNA Markers

Given a genome, i.e., a long string over a fixed finite alphabet, the problem is to find short (dis)similar substrings. This computationally intensive task has many biological applications. We first describe an algorithm to detect substrings that have edit distance to a fixed substring at most equal to a given e. We then propose an algorithm that finds the set of all substrings that have edit distance larger than e to all others. Several applications are given, where attention is paid to practical biological issues such as hairpins and GC percentage. An experiment shows the potential of the methods.

8.1 Introduction

The genomes of several species have been available on the internet for a couple of years now. Because of the availability in an electronic format, computers can be used to do experiments that would normally take months or even years. Data mining research incorporates the exploration of raw data, identifying genes, designing primers, searching for differences between individuals or differences between species. Common in many of these techniques is the use of unique strings. Finding these strings is a challenging task and the main topic of this chapter.

There are a number of techniques that benefit from the existence of unique strings, segments of DNA that occur only once in the genome under consideration. Finding primers, i.e., markers for specific positions of the genome, is one of them. Primers can be used to identify genes that are associated with diseases, but also the identification or classification of blood samples can be done with primers. Furthermore unique strings can be used in phylogenetic research to make an ancestral tree of species or populations. Microarrays are also populated with a large number of unique or nearly unique strings. All these problems benefit from the possibility of generating large amounts of *dissimilar* substrings of the genome.

Some work on marker selection has been done in the past, e.g., a pragmatic

approach for a relative small number of markers can be found in [70]. In [20] we see an approach that uses an alternative of the distance-based measure: the similarity-measure in neural networks for the analysis of DNA and amino-acids sequences; [56] is an overview of the application of Evolutionary Algorithms in bioinformatics tasks like gene sequence analysis, gene mapping, DNA fragment assembly and so on; [46] presents new techniques for recognizing promoters (E. Coli) given an unlabelled DNA sequence based on feature extraction and a neural network for classification; [58] is a practical application that computes primer pairs for genome scale amplification, and also does extraction of coding sequences and primer pair optimization; [27] is a primer design program that uses fuzzy logic to calculate primer qualities, and also uses suffix trees to enhance the spacial complexity; [37] looks at the multi criteria decision process of primer design and reports that trade-off between deviations from ideal values of all of the criteria can result in a considerable speedup. A practical application for retrieving and assembling gene sequences can be found in [71], where afterwards primer pairs are designed for amplification. For more information about the related subject of multiple alignment see [72], where dot-matrices are viewed as projections of unknown n-dimensional points and alignment for n sequences is done by image reconstruction in an n-dimensional space with noise.

However, our major concern is not the design of individual primers, but rather the creation of large sets of primers with special properties. In particular, these primers should be as dissimilar as possible. For example, in PCR experiments, a lot of primers can be put together to do several experiments at once. Because the amount of primers is very large in comparison with the target DNA, chances are that the primers react with each other instead of with the DNA. To prevent this, it is necessary to generate a set of primers that will not react with each other. This is done by selecting only those primers that are highly dissimilar from all other ones. This is a computationally challenging task, because in principle all strings must be compared to each other.

In Section 8.2 we address the combinatorial background. The major algorithms are described in Section 8.3. The *Proximity Search* algorithm finds all occurrences of a string s in a given set of strings, allowing at most e errors. The *Distance Selection* algorithm produces the strings that are at distance larger than e to all other strings in the set. For both algorithms a *trie* is used for efficiently storing the given set. As distance the so-called *edit distance* is employed, with some possible variations. In Section 8.4 we mention practical applications and their implications, e.g., the hairpin problem that arises from the occurrence of self-similar strings. An extensive experiment is reported in Section 8.5. To make our task more biologically relevant we have put some additional constraints on the markers. Apart from being unique, the markers have to adhere to minimum and maximum thresholds of `GC` content and bonding energy. Another constraint is on the combinatorial nature of the string itself: it should not contain a repeating sequence. Our basic approach does not provide us with the tools to deal with all these additional restrictions. In this section the method is augmented to handle these biological requirements. We conclude in Section 8.6.

8.2 Combinatorial background

In this section we sketch the combinatorial background to illustrate the memory requirements. We have about $3 \cdot 10^9$ nucleotides in the human genome, so we also have about $3 \cdot 10^9$ factors of a given length ℓ (for small ℓ). These huge numbers pose a severe restriction on the necessary data structures.

We determine unique substrings by counting the occurrence of all strings of a given length ℓ. The reason we chose for counting is because using a trie would result in enormous memory usage. If for example we make a trie to count the occurrences of length ℓ, we need 4^ℓ nodes in the lowest level. The number of internal nodes will be: $\sum_{i=0}^{\ell-1} 4^i = \frac{1}{3}(4^\ell - 1)$, which makes the total amount of nodes $\frac{1}{3}(4^{\ell+1} - 1)$. The number of branches will be four times as large, i.e., $\frac{4}{3}(4^{\ell+1} - 1)$. In the case of $\ell = 18$, we need a huge amount of memory only to hold the trie. Assume that we represent each node by means of four pointers. In this case each node consists of four 32-bits (= 4 bytes) pointers, which makes the total amount of used memory $4 \cdot \frac{1}{3}(4^{18+1} - 1) \approx 3 \cdot 10^{11}$ bytes (341 Gigabytes). However, since we can only address 4 Gigabytes of memory with a 32-bits integer, we have to use 64-bits pointers, which makes the amount of memory 683 Gigabytes. Furthermore, we need some sort of counter at the leaves of the trie (the nodes at the lowest level). This will increase the amount of memory by at least another 4^{18} bytes (another 64 Gigabytes).

Let us see how much memory it would take if we store all substrings of length 18 in the complete human genome in a trie. The worst case is when all strings are unique (which is possible, since the number of combinations (4^{18}) exceeds the number of substrings ($3 \cdot 10^9$). Let us assume that up to depth 15 the trie is complete and below this it has $3 \cdot 10^9$ nodes per level. Because the number of nodes in the last couple of levels exceed 2^{32} we have to use 64-bits pointers. Up to level 15 there are $\sum_{i=0}^{\ell+1} 4^i = \frac{1}{3}(4^{l+1} - 1) \approx 3 \cdot 10^9$ nodes, all having four 64-bits pointers. This results in a memory usage of approximately $106 \cdot 10^9$ bytes. The levels 16 and 17 both have $3 \cdot 10^9$ nodes with one pointer, resulting in $48 \cdot 10^9$ bytes. The leaves only need to have a counter. This sums up to a total memory usage of approximately $157 \cdot 10^9$ bytes, which is about 146 Gigabytes. This is why we chose to use direct indexing into an array (which in this case only uses 64 Gigabytes of memory). Details are given in Section 8.5.1. In the next section, however, we can still successfully apply tries, because in that case we already have selected a relatively small subset of the set of all substrings.

8.3 Proximity search and distance selection

In this section we describe two algorithms that find (dis)similar substrings of a given string. We consider the genome, for example that of a human, as a string over a finite alphabet Σ. Usually $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$, but the alphabet may also include other symbols when parts of the genome are unknown or marked. For this genome we want to extract a set of substrings of length ℓ that are highly dissimilar to each other. To make this precise we require that the selected strings

have edit distance (or Levenshtein distance [45]) larger than e to all others. The values of ℓ and e ($1 \leq e \leq \ell$) are specified beforehand.

The *edit distance* between two strings is defined as the minimum number of simple operations that is needed to transform one string into the other; a simple operation is either a deletion, an insertion or a substitution of a single character. If only substitutions are allowed we obtain the Hamming distance. Determining the edit distance between two strings is an intensive operation (with quadratic complexity), and if we want to calculate the edit distance between all strings, we need quite a lot of calculation, of order $O(\ell^2 m^2)$, where m is the amount of strings.

By inserting the candidate strings in a trie T, we can calculate the edit distance to all the other strings in a more efficient way. So we now assume that the trie T contains a set of substrings of a given string; this set should not be too large. Our basic algorithm, referred to as *Proximity Search*, tries to find a string s in a trie with root T, allowing at most e errors, and proceeds as follows:

```
Proximity (T, e, s) ::
    if MarkedAsEndpoint (T) and s = λ then
        Visit (T);
    else
        for σ in Σ do
            if Exists (T.σ) then
                if Head (s) = σ then
                    Proximity (T.σ, e, Tail (s));
                else if e > 0 then
                    Proximity (T.σ, e − 1, Tail (s));
                if e > 0 then
                    Proximity (T.σ, e − 1, s);   % insertion
        Proximity (T, e − 1, Tail (s));          % deletion
```

Proximity Search.

The function *Exists* (T) returns true if and only if the node (pointer) T exists. The function *MarkedAsEndpoint* (T) returns true if and only if the root of T is special, i.e., marked as the end of a word. The function *Visit* (T) visits the root node, and does the necessary bookkeeping, e.g., outputs the path and/or marks the last node. The functions *Head* (s) and *Tail* (s) return the head and the tail of the string s, respectively. Finally, $T.\sigma$ is the pointer that leads to the child that corresponds with σ; and λ denotes the empty string.

Essentially we traverse the trie in a normal depth first way, except for the fact that we allow a number of errors. If we have read a character from the trie, we compare it with the first character of the string to be matched, if it matches, we do nothing, if it doesn't match, we decrease the amount of allowed errors. If the amount of allowed errors is less than zero, we exit the function. If we encounter the end of a word in the trie (referred to as an "endpoint"), and there

are no characters left in the string, we have found a match. The end of a word is marked with a flag in the node. This node does not need to be a leaf, so we are able to compare words of different length. A similar approach can be found in [2].

The kind of trie is of no importance; we can either use a binary trie or a trie that has a maximum branching factor of that of the size of the alphabet (although it is of course advisable to use a branching factor that is a divisor of the size of the alphabet in the case that the branching factor is smaller). In our illustration, we use the latter.

We describe the algorithm through two examples, where the second one also allows for insertions and deletions. In both examples we assume that all nodes are marked as endpoints. In the first example we do not use the last three lines of our algorithm (responsible for insertions and deletions); effectively, we "visit" all strings at at most a given Hamming distance from the original one. In the examples we take $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and we always fill the trie with *all* combinations in Σ^* of the appropriate length.

Example 1 (Proximity Search, parallel alignment): In Fig. 8.1 we see a trie with all combinations of 3 letters over the alphabet $\{\mathtt{a}, \mathtt{b}\}$; all nodes in the trie are marked as endpoints. The word `abb` is highlighted as a path in the trie. If we want to find all strings of length 3 at Hamming distance 1 from the word `abb` we use our recursive algorithm. The black dots denote the strings that are found, that is, all strings with at most one mismatch. Notice that in this example no insertions or deletions are allowed.

So the outcome of our algorithm consists of `aab`, `aba`, `abb` and `bbb`. Because of the nature of our algorithm a lot of branches are not traversed, for example the subtrie under the path `ba` is skipped.

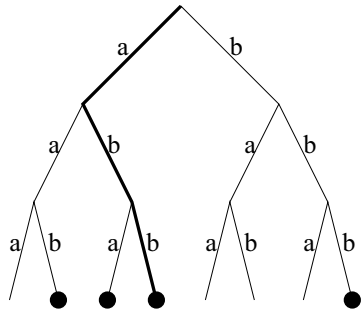

Figure 8.1: All words in $\{\mathtt{a}, \mathtt{b}\}^*$ at at most Hamming distance 1 from `abb`

The complexity of this algorithm, where we do not allow for insertions and deletions, is $(\ell k)^{e+1}$ where ℓ is the length of the string we are searching for, k is the size of the alphabet and e is the number of errors allowed. We assume that the trie is full. If this is not the case, the complexity is lower than this, and k must be replaced by some appropriate average.

Example 2 (Proximity Search, parallel alignment with insertions and dele-

tions): In Fig. 8.2 we see all strings that are at most at edit distance 1 from the word `abb`, where in this case we do allow insertions or deletions. Again, all nodes in the trie are marked as endpoints.

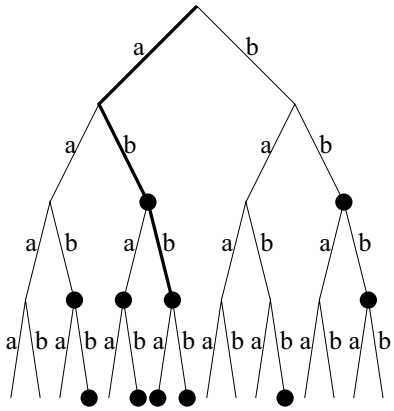

Figure 8.2: All words in $\{\mathtt{a}, \mathtt{b}\}^*$ at at most edit distance 1 from `abb`

In this case the complexity of the algorithm is $(3\ell k)^{e+1}$ The extra 3 is the result of the three times we enter the recursion per node. Henceforth, we shall refer to this algorithm as *Proximity Search*.

However, we want to know which strings are highly dissimilar. To do that we extend our algorithm in the following way. Again, let T be a trie containing a set of substrings of a given string, where the endpoints are marked as such. Initially, all these nodes are marked as "good". Let $L = L(T)$ be the set of all endpoints in the trie. First we do a Proximity Search for each string $s \in L$. If it finds more than one string, mark all these strings (including s itself) as "bad". If we are about to start a search for a string that is marked as "bad", skip it. The reason we can skip this search is because the distance between two strings is symmetric, and therefore we shall encounter all strings that would be marked as "bad" at a later stage in this part of the algorithm. This is an optimization which makes the algorithm run faster as it progresses. When this part of the algorithm is finished, we traverse the trie one more time and extract all strings that are not marked as "bad". The result is a set of strings that have a distance to each other string that exceeds a preset distance e. We shall refer to this algorithm as *Distance Selection*: it marks all strings in T as "bad" that have distance $\leq e$ to some other string in T (except for itself):

```
DistanceSelection (T, L, e) ::
  for s in L do
    if not IsBad (s, T) then
      R := empty trie;
      Proximity (T, e, s);
      if ManyEndPoints (R) then
        MarkAsBad (T, R);
```

Proximity Search.

The function *IsBad* (s, T) returns true if and only if the node at the end of the path s is marked as "bad". The function *ManyEndPoints* (T) returns true if and only if the trie T contains more than one node that is marked as an endpoint. The function *MarkAsBad* (T, R) marks all strings that occur in R as well as in T as "bad" in trie T. In this case, we let *Visit* (T) (from the function *Proximity*) add a string (with last node marked as endpoint) to an initially empty trie R of solutions it has encountered.

Example 3 (Distance Selection): Suppose the genome of a certain species is built up out of the letters from the alphabet $\{\texttt{a}, \texttt{b}\}$, for example `abaababaabaaba`. We consider all substrings of length $\ell = 5$. There are only four unique substrings in this genome, the other two (`abaab` and `baaba`) both occur three times. The four unique strings are put into a trie as shown in Fig. 8.3, where the leaves are the only endpoints. The first child is always an `a` and the second one is always a `b`. After the application of the sequential Distance Selection algorithm (with

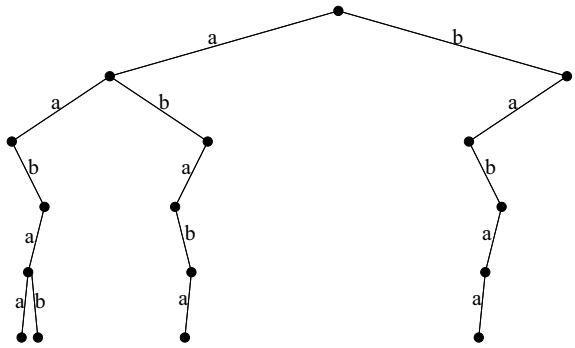

Figure 8.3: Trie of unique strings

distance $e = 1$) only one string is left: the string `ababa`.

The Distance Selection algorithm can be improved by propagating the "bad" label to the parent if all children are marked as "bad". If we find a string in a subtrie that is marked as "bad", we do not have to search the rest of the subtrie because all strings are already marked. This optimization can improve the speed

of the Distance Selection algorithm a lot, especially if e is large.

To make this algorithm practical for DNA marker selection, we have to perform some adjustments. Most of them are necessary to make the initial trie as small as possible. If we don't do that, the trie will most likely become too large to handle, cf. Section 8.2. This has also been done in Example 3. However, this means that the selected strings are far away from each other, but not necessarily far away from all other substrings of the original genome. We shall give the different steps in the next section.

For the purposes sketched in Example 1 and Example 2 we can also use a directed acyclic graph (DAG) for memory efficiency, but in our examples we use a trie. This is in the first place because the examples are more clear and secondly because for the Distance Selection algorithm a DAG can not be used. This is because in a DAG more than one path can lead to a node that is marked as the end of a word. Marking such a node as "bad" could disqualify more than one word in the trie and could lead to undesirable results.

8.4 Applications

We indicate some biological applications for the algorithms proposed in the previous section.

8.4.1 Primer pair selection

Primers are used in techniques like the *Polymerase Chain Reaction* (PCR) [17] and *Multiplex Ligation-Dependent Probe Amplification* (MLPA) [64].

For primer selection, we use the following scheme (not necessarily in this order):

1. Determine all unique strings of length ℓ.
2. Filter for simple repeats.
3. Filter on `GC`-count.
4. Filter on bonding energy.
5. Put the remaining strings in a trie and use the Distance Selection algorithm to select the best strings.

The resulting strings are guaranteed to be highly dissimilar to all other strings in the resulting set. In general it is not true that they are highly dissimilar to all strings in the original set. Because these strings are used for primers, the main concern is that the primers do not stick together in real life experiments. The fact that they may bind to other parts of the genome is of less concern. The reason for that is because primers are used in pairs, and the chance of a primer pair having the same problems, is neglectable.

8.4.2 DNA marker selection

For DNA markers in general the story is somewhat different. In contrast with primers, general DNA markers are not used in pairs. Therefore the emphasis must be more on bonding to the DNA in the genome. Thus the edit distance between the markers becomes as important as the edit distance between the markers and all other substrings in the genome. Therefore we use a slightly adapted technique to select them; essentially it is the same as the one we used to select primers, except in the last step, where we use the Distance Selection algorithm in a different way.

We again extract all strings of length ℓ from the genome and test them to the trie with the Distance Selection algorithm (instead of the strings that are in the subset (and in the trie)).

8.4.3 Other applications

With a couple of alterations, we can use the same techniques to check primers for the formation of *hairpins* or *primer-dimers*. A hairpin in this context means that a primer is its own reverse complement, or at least up to a certain degree. The reverse complement is obtained by reversing the string and replacing every nucleotide with its complement (`A`↔`T`, `C`↔`G`). If a primer's head is the reverse complement of its tail, chances are that the primer folds and binds to itself. This is what we call a hairpin. A primer-dimer is a similar structure, where two primers stick together. There are two sorts of primer-dimers: the ones that stick with the heads to each other, and the ones that stick with the tails. Both hairpins and primer-dimers are undesirable structures that should be removed from a set of primers.

Although many programs can check things like this, we use the structure of the trie again to check every combination at once. Another advantage is that we can allow for one or more errors in the formation of hairpins or primer-dimers.

For the normal formation of primer-dimers, we first make the reverse complement of the primer we are checking. The reason for this is that using the reverse complement does not have any impact on the trie. Otherwise we would have to use the normal reverse and check if `a` matches to `t` instead of itself. With this approach, we don't have to worry about hairpins any more, because the algorithm above has already filtered them out. Take for example the primer in Fig. 8.4; this is a primer that has formed a hairpin.

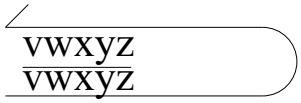

Figure 8.4: A primer in hairpin configuration

But in the algorithm, we have also tested the primer to itself in the configuration shown in Fig. 8.5, so there is no need to do it again.

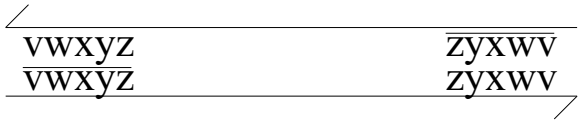

Figure 8.5: Two identical primers that could form a hairpin

8.5 Experiments

As an illustration of the methods described in this chapter, the pairwise alignment of many primers using a trie, we have done an experiment on a part of the human genome. The reason we only took a part is that in this case the numbers are somewhat easier to interpret.

In this experiment, we select pairs of primers as they are commonly used in pairs in real life experiments like PCR. Rather than just selecting a set of markers that are only tested for their uniqueness, we have chosen to implement also some restrictions that have a biochemical motivation. Of course, restricting the number of markers before computing their edit distance will reduce the computational effort for that last step.

The computer on which we ran our experiment is a Pentium IV based machine at 1,8 GHz with 256 Megabytes of internal memory. The operating system is SUSE Linux 9.2 with a 2.6.8 kernel.

We summarize the results of this experiment below. The last step of the experiment was performed using the algorithmic technique sketched before. In the remainder of the section we explain some additional details of our experiment, involving the selection of unique primers and filtering the primers based on combinatorial and biochemical properties.

The results are as follows:

Finding markers We took the human chromosome 1 from [69] (version June 26, 2006) which consists of 247,249,719 basepairs. In the first step of our analysis we extracted all unique strings of length 16. There were 123,685,514 of them (out of a grand total of 247,249,703). We need exactly 2 Gigabytes of memory to count all occurrences of length 16 and we ran our experiment on a computer with 256 Megabytes of memory. To be on the safe side, we chose the amount of memory available for our program to be 128 Megabytes, so we had to make 16 passes over our input data. The program was run as root to ensure memory locking. This process took 12 minutes and 49 seconds.

Filtering repeats Simple repeats are strings of low complexity, and in some sense not of much interest to biologists. We denote a repeat in the following way. By $(\underbrace{x \ldots x}_{s})^r$ we denote a repetition of r copies of a string of length s. So for instance, $(xxx)^2$ means a string of length 6 consisting of two equal halves of length 3, like *abbabb*; note that the character x may represent different charac-

ters. In this notation, we filtered out all of the following simple repeats: $(x)^5$, $(xx)^5$, $(xxx)^4$, $(xxxx)^3$, $(xxxxxxx)^2$, $(xxxxx)^3$, $(xxxxxxxx)^2$ and $(xxxxxx)^3$. This resulted in a set with 115,695,993 elements. This program was also run as root to ensure memory locking and took 2 minutes and 34 seconds.

GC content and temperature After that, we selected primer pairs based on a GC count between 20 % and 80 %, and a melting temperature between 60 °C and 63 °C. The internal spacing of the primers was between 480 and 520 basepairs and the spacing between the primer pairs had a minimum of 9,800 basepairs. This resulted in 41,565 primers. This process took 28 seconds.

Edit distance After the Distance Selection (with respect to the formation of primer-dimers and normal alignment) with minimum edit distance 2, there were 35,781 primers left. This process took took 8 minutes and 47 seconds. If we set the edit distance to 3 only 7,530 primers remained.

This experiment shows that Distance Selection can make an enormous difference with respect to primer selection.

8.5.1 Finding markers: Determining unique substrings

We have chosen to count occurrences of length 16 strings, in a rather direct way, making multiple passes over the input data. At each pass we count the occurrences of a "block" of possible length 16 segments and write the result to disk afterwards. This algorithm is designed to use linear disk access as much as possible. All random access is done in memory only.

First, we convert the DNA data from ASCII to binary. This has three advantages. As the data is compressed by a factor of 4 (using only 2 bits per nucleotide), we need to use only one-fourth of the original memory. As a direct consequence, we can read data at 4 times the speed, for each pass. Finally, calculation with base-4 numbers is trivial compared to string operations with respect to the amount of calculation. The number of passes is directly related to the amount of data that fits in the memory.

Assigning the right two bit code to each letter, we can even take advantage of binary operators like the NOT operator, if we choose `a` and `t` as each others complement in the encoding (as they are in nature), as well as `c` and `g`. Hence we take the order `a`, `c`, `g`, `t` (which happens to be the alphabetical one), and code these nucleotides as 00, 01, 10, and 11 in binary.

Since we are interested in strings of length 16 we keep track of the last 16 nucleotides. When we read a new nucleotide, the string is shifted to the right by two bits, and the new two bits are shifted in (at the least significant end). This is a very inexpensive method to keep track of the last read string.

Of course there are some additional complications in preprocessing the data. The first one is that the data (as available via [69]) does not consist of only four letters. It has an alphabet of 10 letters. These are the normal letters, {`a`, `c`, `g`, `t`}, the normal letters written in capital, {`A`, `C`, `G`, `T`}, and the letters {`n`, `N`}. The

capitals denote repeating segments or sequences of "low complexity" in the DNA (pieces of DNA that have a repetition with a period of 12 or less), which are biologically less interesting regions. The letters `n` and `N` denote the absence of data. We have decided to look for markers in the lower case segments of {`a`, `c`, `g`, `t`} only.

However, we cannot simply erase the other letters. For example, suppose we convert the string `ATATATttNttaatNnn` to `ttttaat`, then we will have found a string which is not part of the original DNA. One way of circumventing this problem is to remove all capitals and the letter `n`, and by keeping track of when in the output file we need to re-calculate the current number. We put these offsets in a separate table. In this way we avoid having to use additional escape sequences in the binary code of the DNA, making the code shorter and faster to read.

In our example we get two tables, one containing the binary equivalent of `tttta` and one containing the offsets 6, 2, 1, 5 and 3. To decode these tables, we take the offsets table and first write 6 `n`'s, then we decode 2 nucleotides, then we write 1 `n` and 5 more nucleotides. Finally we write 3 more `n`'s, which results in `nnnnnnttnttaatnnn`, the same string we had before, except all the capitals have been converted to `n`'s.

Our input data stores only a single strand of the DNA, but in practice one considers both strands. Each time we find the occurrence of a string, we also have to take into account the reverse complement of the string, as this is also present in the DNA. Rather than counting both strings however, we have chosen to take one of the two as a representative of the pair, incrementing only the counter of the smallest string (lexicographically ordered). Comparing strings can be done with the standard `<` operator, another advantage of the binary recoding. Of course, we keep track of the reverse complement on the fly, shifting and adding new bits (but in the other direction). Again this is a rather inexpensive operation.

In an additional experiment we have selected unique primers not only for the first chromosome, but also for the full genome. As an illustration, in Fig. 8.6 we see the number of occurrences of unique strings of length 12 for each consecutive series of 100,000 basepairs on the full human genome. The vertical dotted lines denote the chromosome boundaries; these are ordered as follows: 1 to 22, then X, Y, and finally the mitochondrial DNA. This mitochondrial DNA is so small that it does not show up in these graphs. The histogram shows the "hotspots" where many markers are found. The white bands (at offsets 1,300 and 15,900 for example) are due to unknown or unstable DNA. It is missing in our input (a large series of `n`'s).

8.5.2 Filtering out simple repeats

Strings containing short repetitive substrings are not of much interest to biologists, and that is why we filter out such "simple repeats".

This implies that we have to perform a certain pattern matching on the markers found. Since the markers all have a fixed length we do not have to

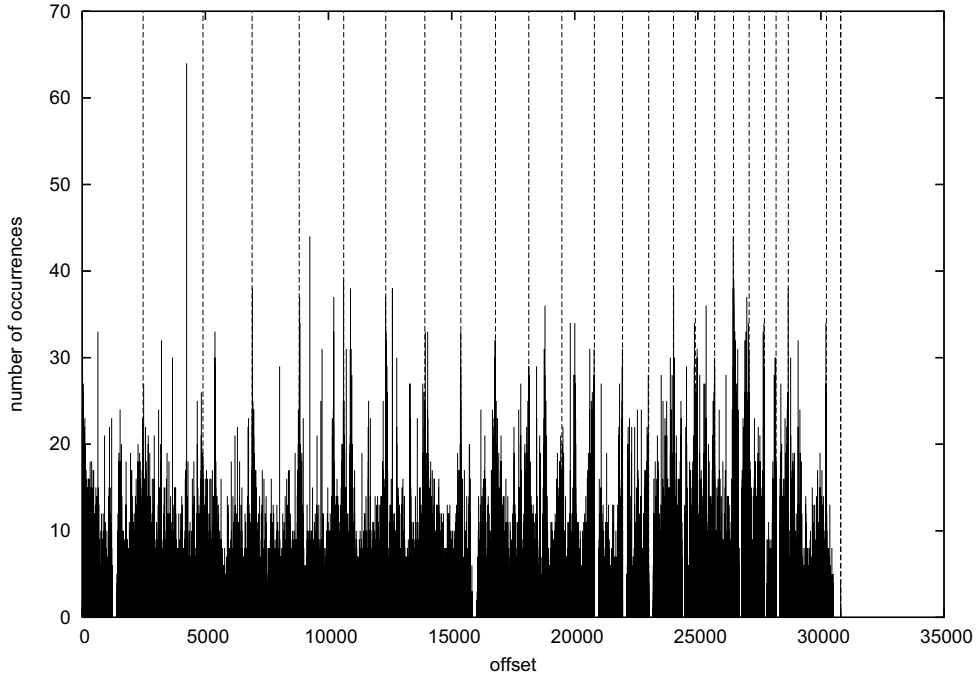

Figure 8.6: Occurrence of unique strings in the human genome

search for repetitions of unbounded length. Instead we can determine the unwanted substrings beforehand and search for them in the markers. We have stored the simple repeats in a trie, using the technique of Aho-Corasick [1], which is basically Knuth-Morris-Pratt [38] pattern matching, but applied to a set of patterns rather than a single pattern. This makes it possible to scan for all forbidden repeats in parallel.

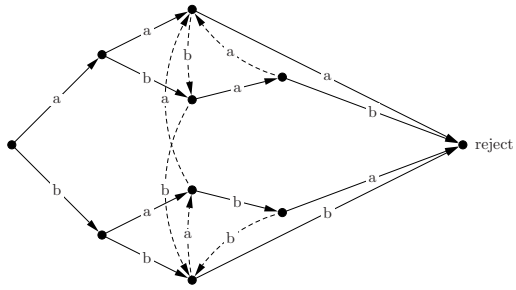

Figure 8.7: Wired repetition trie

After computing all simple repeats they are stored in a trie. As an illustration, Fig. 8.7 shows a trie to filter out the repetitions $(x)^3$ and $(xx)^2$ over the two letter alphabet $\{\mathtt{a}, \mathtt{b}\}$, i.e., `aaa`, `bbb`, `abab` and `baba`. Note that `aaaa` and `bbbb` are already excluded. Whenever we reach the node at the right, we have

found a simple repetition and we can disqualify that part of the input. This trie is "wired" by dashed edges that are added in a preprocessing phase, asserting that we always keep track of the longest prefix of one of the unwanted strings while scanning the marker. As usual with the Aho-Corasick method, we have to take special care when one of the unwanted repeats is a substring of another one, like `aa` and `baabaa` for the repeats $(x)^2$ and $(xxx)^2$.

8.5.3 GC content and temperature

The practical use of primers in experiments is governed by bounds on the GC content of the primer and its melting temperature, and tools like Primer3 [60] allow the user to specify bounds. Keeping track of the GC-percentage is an easy task if we keep track of the symbols that are shifted in and out of the current substring of length 16, while reading the data. So when the least significant nucleotide of the reverse complement is `c` or `g`, we decrease the GC-count (it is shifted out in the next step). Then we shift in the new nucleotide, and if the least significant nucleotide on the original strand is `c` or `g`, we increase the GC-count. We actually use both the string itself and its reverse complement, because it is faster to extract the least significant bits.

A formula to calculate the melting temperature of a piece of DNA is given in [7]. We chose to implement this in an algorithm acting like the one calculating the GC-percentage.

The formula for the temperature is mainly based on the successive pairs occurring in the fragment, each of the pairs contributing to the bonding energy of the primer. We keep track of the two last nucleotides on the reverse complement (the first ones to be shifted out), and the next ones shifted in, and we can update the melting temperature by a simple table lookup. Internally we use integers to represent the temperature, to avoid rounding errors.

8.6 Conclusions and further research

We conclude that we can select primers and DNA markers that have a high chance of performing well in real laboratory experiments and microarrays.

As can be seen from our experiment in Section 8.5, most primers are disqualified because of the GC content and bonding energy, so intuitively one might think that this should be the first step in our analysis. This might be a good solution if the GC content and bonding energy are known in advance, which is often not the case in practice. With the current solution we can still tinker with the options, and that is the reason that the focus is primarily on unique strings.

Instead of working with tries we would like to experiment with *Compact Directed Acyclic Word Graphs* (CDAWG) [33]; this datastructure behaves like a trie (for our purposes), but is more compact. Furthermore the online algorithm stated in [33] is very attractive for these kinds of problems.

At present, the output of the Distance Selection algorithm is not used iteratively. We could for example still use the other primer in a primer pair if one

primer is discarded. Since all these ideas are still very new, lots of experiments should be done.