Unique factors in the human genome

Jeroen F. J. Laros

May 24, 2005

Contents

Chapter 1

Introduction

The human genome has been charted for some time now and we have learned much from it since. If you compare the genome to a book and genes to words in it, then we are at present not able to read this book, sometimes we can read a word here and there, but most of the time we can not even make the distinction between words. Therefore it is of great interest that we try to find the genes that make up the genome. This is done by looking at differences between individuals. If for example we have two identical individuals except for the colour of their eyes, then we can perhaps identify the gene or genes that are responsible for the eye-colour by isolating the difference in the genome between the individuals in theory.

Of course it is not that simple in practice. We can not simply read a person's DNA and compare it to that of another person. The main reason is that the human genome is far too large to make this practical. Extracting someone's DNA is a very tricky job. During the process, the DNA is cut into a large number of pieces which have to be puzzled together again. For this reason it is not practical to compare two large strings of DNA.

At present there are a couple of techniques that try to find genes; all of which make use of markers in the genome. A marker is a piece of DNA that is distinguishable from all other pieces. By using markers we do not have to puzzle the DNA together anymore. We simply look whether the marker is present or not. If for example a combination of markers is always present with individuals that have blue eyes and always absent with people with brown eyes, then it is probable that the DNA surrounding these markers forms a gene that is at least partly responsible for blue eyes.

Some of these techniques make use of the unicity of these markers to duplicate certain areas of DNA. This is done by making a pair of markers which we call primers and by putting specially designed tails on these primers. Then these tails can be used to duplicate the DNA between these primers. This technique (referred to as *Multiplex Ligation-dependent Probe Amplification*, or in short MPLA [1]) works as follows. We take a strand of DNA which matches one of the primers, and we use the tail attached to the primer as a starting point for

copying. Now we have a strand of DNA that is bounded by our primer at one end. Now we can duplicate this piece of DNA and do the same to the reverse-complemented strand. By doing this repeatedly, we end up with a "soup" that contains mostly the DNA that is bounded by the primers. Because this process can be automated, the "soup" gets thick enough to work with.
Another process similar to MPLA is a technique referred to as *Polymerase Chain Reaction*, or PCR [2]. This technique also requires two primers, but this time one on each strand. The DNA between the primers is multiplied.
Another thing to do with primers can be the following. Let us assume we already know the location of a particular gene. Then we need to find a marker for this gene and we can put this marker on a so-called "bio-chip" or "micro-array". Whenever we use this chip on the DNA of an individual, it shall give a positive reaction if the gene is present and a negative one if it is not.
As can be seen, primers are of major interest for genetic engineers, and therefore primer design is too. As mentioned before, the human genome has been mapped for a large part; this has been done to look for similarities between a number of healthy people. By doing so much of the natural "noise" can be filtered out. With primer design we mean in the first place the determination of unique factors in the genome. This is the biggest part of the problem. We are not the only ones [3] interested in this. Once we have the unique factors, we can filter out simple repeats and primers that do not have the correct melting temperature or GC-count.
This project is at the request of Peter H. K. G. Taschner of the Bioinformatics Support Group of the Leids Universitair Medisch Centrum (LUMC) and in cooperation with Hendrik Jan Hoogeboom and Walter A. Kosters, both of the Leiden Institute of Advanced Computer Science (LIACS).

Chapter 2

The assignment

As mentioned before, the design problem is split into multiple parts:

1. Determining unique factors of a certain length.
2. Filtering for simple repeats.
3. Filtering for GC-count and melting temperature.

The reason for this modular approach is because the first step is rather general and is the largest part of the analysis, in other words, it requires the most work and it should only be done once. It will result in a very large file which is used as a "mother file" for the rest of the process, so we can simply change the repeats filter or the final filter without having to re-calculate the unique factors. Only one thing is certain at this point: the definition of uniqueness shall not change, the other constraints can vary (by design).
Only when a new version of the human genome is published [4], we need to re-calculate the unique factors. Of course there is a natural variation in the human genome, but the amount of DNA that differs between people is relatively small. When, for example we compare the DNA of a human and a chimpanzee, we find that 99.7% of the DNA is equal. Differences between people is even less than this 0.3%. The genome data we use is the "standard" genome: the average genome of a number of people.

2.1 Tries

In order to count all occurrences of all combinations of a certain length, we use a trie. A trie is a pattern-matching datastructure that uses failure links to keep parsing when a pattern does not match, and this way we only need to traverse the input once. The failure links resemble the ones in the KMP [5] algorithm. When failure links are applied to search trees, we call it the Aho-Corasick [6] method.

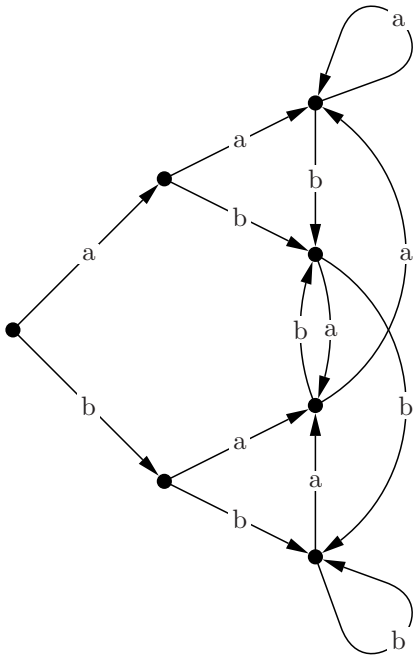

Figure 2.1: Example trie

The Aho-Corasick method is a pattern matching algorithm. It constructs a *finite automaton* or *trie* from a dictionary. This trie is then used to locate strings from this dictionary in the input text.
The tries we will be using look like the one in Figure 2.1. This trie in particular recognizes all strings of length 2, which are *aa*, *ab*, *ba* and *bb*. Tries like this have the property that if the trie is of depth N, it can recognize all strings of length N (because it is a total tree for length N). Because of this, we do not have to traverse the trie all the way when we encounter a new string. Each new string shall result in a recognized state.

Example 2.1.1. Let us say that we want to count all occurrences of strings of length 2 in the binary string `abaababaabaababaababaa`. First we construct the trie of Figure 2.1 using the Aho-Corasick method and then we walk down (once) while we follow the labels `a` and `b` respectively. From now on we stay at the lowest level of the trie.
When we now read the next letter in our string, e.g., an `a`, we only have to follow the branch labeled `a` of the state we are in, which is "accidentally" the same state we would reach when we would have started all over again.
Each time a node on the lowest level is reached, we have encountered a string

which is encoded on the branches leading from the root to the node in question.

The only difference in the DNA case is that the trie will be much larger than in the example and that it has four outgoing branches per node (instead of two) labeled `a`, `c`, `g` and `t`. It is important to note that because the trie is total and all strings have the same length, the failure links only reside on the lowest level of the trie. This is important because this property makes it possible to use an implicit datastructure mimicking the behaviour of the trie.

2.2 Determining unique factors

Figure 2.2: Double stranded DNA

Since DNA is a double stranded structure, see Figure 2.2, biologists consider a piece of DNA equal to its reverse complement. Each nucleotide in a single stranded piece of DNA is accompanied by its complement on the other strand. These pairs occur in the following combinations `a`-`t` and `c`-`g`. For example; the string `atggacaaac` is considered the same as `gtttgtccat`. This is because of two reasons.

1. Single stranded DNA can only be read in one direction.
2. In the double stranded structure each `a` on one strand has a `t` on the other strand and the other way around. The same applies to the `g` and `c` nucleotides.

So, whenever we encounter a certain string, we also encounter its reverse complement, and they should be identified. We have to take this into account when we count unique factors.

2.3 The length of the unique factors

We chose for the number 18 as the maximum length of unique factors. In the first place because it is what was requested, but there is also a statistical argument for it. We have about $3 \cdot 10^9$ nucleotides in the human genome, so we also have about $3 \cdot 10^9$ factors of length N. If we choose N to be 17 and if we assume that the genome is approximately random data, there is about $\frac{3 \cdot 10^9}{4^{17}} \approx 17.5\%$ chance that a piece of length 17 is not unique. For length 18 this chance is only 4.4%. Because we are interested in unique factors, length 18 is a good guideline.

The estimate above is only a rough one, if we want to calculate the exact percentages, we use the following argument; the chance to encounter a string of length N at a certain position is $p = \frac{1}{4^N}$, so the chance not to encounter it is $(1-p)$ and the chance that we do not encounter it L times is $(1-p)^L$. This makes the chance to encounter it at least once $1-(1-p)^L$. The estimation assumes that $(1-p)^L = 1 - Lp + \text{error}$, with $|\text{error}| \leq \frac{1}{2}L(L-1)p^2$. For our example, the error is less than $\frac{1}{50}$.
Furthermore, we can extend a factor of a certain length to a larger one, because if a factor contains a unique substring, it shall be unique itself. Note that we do not have all unique factors of a certain length when we use such extension techniques.
Another reason is that the runtime of our core algorithm increases exponentially with the length we are looking for. This compels us to keep the length as small as possible.

Chapter 3

First attempts

3.1 The original approach

If we use the trie to count the occurrences of length N, we need 4^N nodes in the lowest level. The number of internal nodes will be: $\sum_{i=1}^{N} 4^{N-i} = \sum_{i=0}^{N-1} 4^i = \frac{1}{3}(4^N - 1)$, which makes the total amount of nodes: $\frac{1}{3}(4^{N+1} - 1)$. The number of branches will be four times the number of nodes: $\frac{4}{3}(4^{N+1} - 1)$.
In the case of $N = 18$, we need a huge amount of memory only to hold the trie. Let us assume that we use the following minimal datastructure as a node

```
struct node {
  node *child[4];
};//node
```

In this case each node consists of four 32-bits (= 4 bytes) pointers, which makes the total amount of used memory $4 \cdot (\frac{1}{3}(4^{18+1} - 1) = 366503875924$ 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). Because of this calculation (and the fact that we did not have an 64-bit operating system at hand) we did not even make an attempt to realize it.

3.2 An alternative approach

As mentioned before, we can mimic the behaviour of the trie without actually building it. We need at least a table with 4^N entries in order to count the occurrences of factors of length N; these entries correspond with the leaves of the trie. If we look at a piece of DNA as an integer written in base-4, then we can easily calculate which node of the trie will be reached.

Let $a_{N-1} \dots a_0$ be a piece of DNA in base-4, then the index in the table is:

$$\sum_{i=0}^{N-1} a_i 4^i$$

Note that we can do this only because all internal nodes are used only once and because the trie is total.
We can now make a table with $4^N = 2^{2N}$ entries, and we only have to increment the values at the indexes we encounter as integers in base-4. The next index can be calculated by removing the most significant element and concatenating the read value as the least significant element.
This approach is in principle the best we can do. It hardly needs memory at all, but the table (on disk) will always be 64 Gigabytes ($4^{18} = 64$Gb).
Another advantage in using numbers in base-4 is that it is trivial to implement them in the binary number system, making the operations very fast in comparison to string operations.

3.3 Compressing the table

However, since the human genome "only" has about $3 \cdot 10^9$ nucleotides, which is much smaller than the number of 4 letter strings of length 18, our table will mostly consist of zeros. Even when we store both the index (as explained in Section 3.2) and its value (that is 40 bits, 36 for the index and 4 for the value), we only need $40 \cdot 3 \cdot 10^9$ bits for the table (about 14 Gigabytes). This is the worst case scenario, it would mean that each factor of length 18 is unique in the human genome.
This approach however, is much more complicated than the previous one and the result is not that great either. A problem with this approach is searching for the factors afterward, because the position in the uncompressed table is the factor (coded in base-4). This position is lost when we compress the table. We could make an index for the compressed table, so we do not have to decompress the entire table when we do a lookup in the table. However, this is not very practical when the analysis is still in progress.
Because the previous approach is probably the fastest, we will use that one. And if disk space is an issue, we can always compress the result after-wards.

3.4 First results

The first results were not very encouraging. It took more than four hours on a PII at 450MHz to analyze 2 Megabytes of random data counting unique strings of length 13. These results are more than disappointing: not only are we faced with the linear increase of complexity (from 2 MB to about 700 MB for the human genome) but there is also the exponential increase from 4^{13} to 4^{18} when we count strings of length 18. Rough calculation brings us to a runtime of about $\frac{700}{2} 4^{18-13} = 1433600$ hours (163 years).

3.5 Bottlenecks

The main reason for the bad performance is probably the disk access for the output file. This is because it occurs at random positions and since disks are devices with heads and spinning magnetic disks, it is not hard to see that linear access is much faster than random disk access; the heads don't have to be repositioned when the disk is writing linearly. We can see this clearly in the case $N = 18$, because we only need to manipulate 0.7G of the 64G, and the initialization of 64G (on a modern machine) takes about an hour, while the analysis of the data (based on the calculation above) takes almost an eternity.
If we can somehow linearize the disk access for the output file, analysis should take less time than the initialization of the output file. This is because we do not need to write nearly as much data to disk during the analysis phase compared to the initialization phase.
One way of approximating that behaviour is by caching the results, and by sorting the cache before it is written to disk.
If we for example take a cache of 100M and use a naive approach to make the cache, that is: store both the index as well as the value, then we would flush the cache approximately 140 times. We can also only store a tag and calculate the disk address from the location in the cache.

Example 3.5.1. Let us say that we want to store the value x at position 1234 in a cache that has 1000 positions available, then we store the value x and the tag with a value of 4 at position 123. If we now want to increase the value of 1234, we look in the cache at position 123 and check if the tag equals 4. If it does we can increment x. Otherwise we need to write the value of position 123 to disk (at position 123 concatenated with the value of the tag) and load the correct value from disk (position 1234) in the cache and set the tag value to 4. Then we can increment x.

By doing this, we can presumably decrease the cache flushes to about 14 or 15, which results in a runtime of about 14 or 15 hours.
Note that the predictions above are only rough estimates, especially the one about the runtime. It assumes that writing 50M spread over 64G linearly to disk takes just as long as writing 64G. More assumptions like this are used.

3.6 A possible solution

Probably, using a cache is not as useful as we thought. This is because the data has a high degree of randomness. This will result in the same problems we have experienced before. However, the cache would still have a few advantages, the cached results can be incremented without having to read from disk and by flushing the cache we approximate linear writing to disk, which is faster than random writes.
These advantages are also present when we buffer the results in memory another way, especially when we use a self-sorting buffer. When the buffer is full, we can

flush part of it to disk.
A possibility is to use a tree as a buffer; the path to a leaf is the segment. If the tree gets too large, we can take a sub-tree, flush it to disk and remove it from the tree. The fuller the sub-tree is, the more performance gain we shall have. We could also keep track of the most used path in the tree and leave this part of the tree in memory for as long as possible. There are two main problems with these solutions, one is that they need huge amounts of memory, essentially the same amount as calculated in Section 3.1. The other is that the DNA is approximately random, so there is probably no such thing as a subtree that is far more larger than the others, or a path that is taken far more times than the others.

Chapter 4

Some serious attempts

4.1 Caching

We implemented a cache with 16 bit cachelines as shown in Figure 4.1.

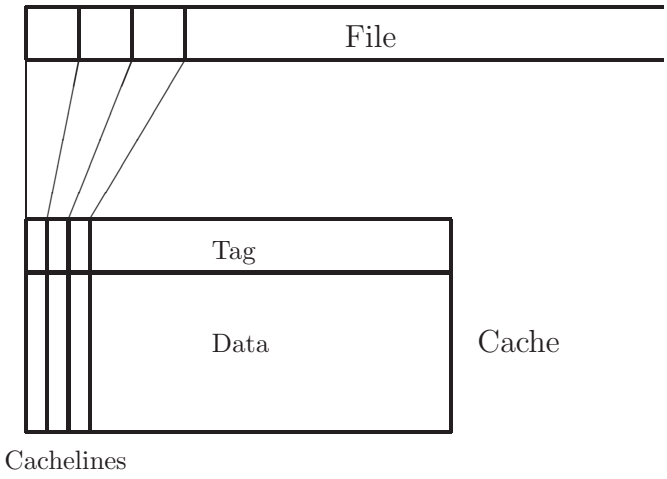

Figure 4.1: Cache

A cacheline is a datastructure consisting of two fields, a *data field* and a *tag field*. The number of *cachelines* c is less than the *filesize* f (in bytes), otherwise the file would fit in memory. Therefore the tag field is present, to hold an offset. Each cacheline addresses a number of values and which of those values is in the cacheline is stored in the tag field. We shall now go into more detail than we did in Example 3.5.1.

The size of the tag field t is given by

$$t = {}^2\log\frac{f}{c},$$

where $f = 4^N$. Note that f must be a multiple of c, otherwise we waste bits in the tag field. For this reason we always choose c to be a power of 2. f Is divisible by c because f is a power of 4 and assumed to be larger or equal to c.
Let m be a *bitmask* such that

$$m = \texttt{~}(\text{0xFFFF} \texttt{ << } t).$$

in which the `~` operator denotes the binary NOT and the `<<` operator denotes the logical SHIFT LEFT. The hexadecimal value 0xFFFF is the maximum value of an 16-bits integer. When viewed in binary, it consists only of 1's. By shifting this value by a certain value, we can use it as a bitmask to select part (in this case the lower) of an integer.
To store a value v in the cache at position i, we first need to calculate in which cacheline c_1 the data will be stored. This is calculated easily by:

$$c_1 = i \texttt{ >> } t,$$

where `>>` denotes the logical SHIFT RIGHT.
We use the mask to calculate the tag:

$$t = i \texttt{ \& } m,$$

where the `&` operator denotes the binary AND. The value is stored in the data field as follows:

$$v = i \texttt{ << } t.$$

Example 4.1.1. Let $N = 18$, then the filesize $f = 4^{18}$. Let the amount of available random access memory be 512M (536870912 bytes) and let each cacheline be 16 bits (2 bytes). This makes the number of cachelines c 268435456. The tagsize t is

$${}^2\log\frac{4^{18}}{536870912/2} = {}^2\log 256 = 8 \text{ bits},$$

this makes the tagmask

$$\texttt{~}(\text{0xFFFF} \texttt{ << } 8) = \text{0x00FF}$$

which leaves 8 bits for the datafield (making the maximum value of the datafield 255). Now let us assume we want to store something at position 0x123456789. Then

$$c_1 = \text{0x123456789} \texttt{ >> } 8 = \text{0x1234789},$$

and the tag is

$$t = \text{0x123456789} \texttt{ \& } \text{0x00FF} = \text{0x000000089},$$

So at cacheline 0x1234567 the tagfield will be set to 0x89. If the tag already has this value, we can increment the data field. If the tag has another value, then we must first add the datafield to the value on disk at position

$$(0x1234567 << 8) \mid \text{oldtag}.$$

If the datafield has value 0, then this cacheline has not been used, so we do not need to write to disk.

4.2 Blocking

Since random disk access is slower than linear access, it is beneficial to write blocks of data instead of single bytes. Flushing cachelines is random because our input data is approximately random.
Operating systems (and even disks themselves) usually read and write to disks in a so-called blocking mode. This means that not one, but several hundreds of bytes, which we shall call a block, are read or written at once. We shall refer to the size of this block as blocksize. The blocksize is usually 512 bytes, but multiples of this value can also be used.
We can use this fact while flushing cachelines; first we determine the *blocksize* b, then we see which cacheline must be flushed and then we look if there are neighbouring cachelines that also fall into this block. Theoretically, we gain a speedup as long as each cacheline addresses less memory than the blocksize, otherwise this technique is useless. The expected speedup is proportional to:

$$\frac{c \cdot b}{f}$$

By doing this we can write n cachelines to disk in less time than it would take to make n random disk accesses.
The speedup follows indirectly from the fact that we can rely on the fact that the cachelines which have been flushed to disk are empty (and therefore ready for use without flushing) the next time we write to the cache.

4.3 Multiple passes

Even with caching and blocking, the amount of memory is still the problem. We simply do not have enough memory to make this technique work.
For example, assume that we have 1 Gigabyte of memory available; 30 bytes are needed to address this amount. For the 64 Gigabytes of output data, we need 36 bits. Because we can use the positions in memory as part of the factor (the piece of DNA translated to an integer), we need 6 bits of tag data. This only leaves us with two bits for the data field, which means we can only count to 3. Because we have $\frac{1}{64}$ of the memory we need, the cache will be full after about $\frac{1}{64}$ of the input has been read. Furthermore, we probably want to use more than

two bits to count the factors, which makes the amount of available memory even less adequate.
So we turned the problem around. We first look how much memory we have, call this amount a, for practical purposes a needs to be a power of 2. Then calculate the size of the output file (4^N, with N the length of the factors we are looking for), call this b. The number of passes can be calculated by $c = \frac{b}{a}$.
Now the algorithm proceeds

1. Calculate the amount of bits needed to address the memory by $m = {}^2\log a$, let $i = 0$ be the current pass.
2. Go in search for all numbers of length N starting with i. So in iteration i we search for i concatenated with all numbers of length m in parallel. Increase i by one.
3. Repeat step 2 until $i = c$.

We call this method the *multiple passes* method.
Note that this technique heavily relies on the fact that we have converted the DNA strings to numbers, and therefore we refer to the strings to be found as numbers in step 2.
This approach has a couple of advantages, the most important of them being:

- There is no random disk access anymore.
- The algorithm can be distributed to a maximum of c computers.
- Since the memory has an implicit data structure, we have no need for tags anymore, so effectively we have twice as much memory which we can use.

As far as we can tell, there is only one drawback, and that is that we need to parse the input c times.
However, the linear disk access seems to make up for the fact that we need to parse the input multiple times.

4.3.1 Distribution

By altering our algorithm in the following way we can distribute the calculation:

1. Choose two values i and j with $0 \leq i \leq j \leq c$, i being the pass to start with and j the pass to end with.
2. Calculate the amount of bits needed to address the memory by $m = {}^2\log a$.
3. Go in search for all numbers of length N starting with i. So in each iteration we search for i concatenated with all numbers of length m. Increase i by one.
4. Repeat step 3 until $i = j$.

We could perhaps use this technique to calculate unique factors of length 19 and larger. Of course we need 4× the amount of computers to have a speedup of 4, but with the modern superclusters at our disposal consisting of about 200 or more nodes, we can have a rather nice speedup.

4.4 Preliminary results

We took 2 Megabytes of random data as DNA input and counted all factors (for different lengths) in this input (this is about 3% of the length of the genome). The tests to get these results were done on an Intel Pentium II at 450 MHz, with 256M RAM. Since the input data is random, it means that the figures in Table 4.1 are probably "worst case" because counting random factors will result in random disk access.

length	output	original	cached	cached and blocked	mpass
8	64K	5m 19s	4.4s		2.4s
9	256K	5m 28s	5.2s		2.6s
10	1M	5m 42s	6.5s		3.6s
11	4M	8m 25s	8.9s		4.3s
12	16M	30m	17.6s		4.4s
13	64M	4h 10m	55.2s		8.2s
14	256M	∞	58m	18m	26.8s
15	1G	∞	1h 44m	46m	1m 40s
16	4G	∞	1h 19m		7m 25s
17	16G	∞	1h 35m		29m 30s
18	64G	∞	2h 24m		

Table 4.1: Different strategies

We see the length of the strings we are looking for in the first column and the output size of the table in the second column. In the next four columns we compare the different strategies. The first one is the original approach, that is, we put the table on disk and write directly to this table. This method is discussed in Section 3.2. The second one is like the first one, except we use a cache. This cache flushes a cacheline to disk if the cacheline in question is already occupied, as discussed in Section 4.1. The third approach is like the second one, except upon a cache miss we look which cachelines fall into a disk block and flush all of these cachelines to disk, leaving them empty as discussed in Section 4.2. The final approach is the multiple passes method as discussed in Section 4.3.
This differs from the other approaches because it makes maximal use of available memory and only does disk access when it is absolutely necessary. We say maximal use of memory, because we only use memory for data, not for tags. We only use the memory offset for the representation of the factors. Furthermore, it is not difficult to use only 4 bits instead of 8 to count the factors, effectively doubling the memory capacity. Note that in this case we can only count to 15,

but this is more than enough to find unique factors, not to count all factors though.

Chapter 5

The final approach

5.1 Preprocessing the data

As mentioned before, we convert the DNA data from ASCII to binary. This has three reasons:

1. The data is compressed by a factor of 4 (by using only 2 bits per nucleotide), so we need to use only $\frac{1}{4}$ as much memory.
2. Because of reason 1 we can read data at 4 times the speed.
3. Calculation with base-4 numbers is trivial compared to string operations with respect to the amount of calculation.

If we assign the right value to each letter, we can even take advantage of binary operators like the NOT operator. We have to make sure that `a` and `t` are each others complement in the encoding (as they are in nature), as well as `c` and `g` as can be seen in Table 5.1. Hence the order `a`, `c`, `g`, `t` (which happens to be the alphabetical one).

Nucleotide	encoding
`a`	00
`c`	01
`g`	10
`t`	11

Table 5.1: Binary encoding of the nucleotides

This is how we read the binary data. Since we are interested in strings of length ℓ we keep track of the last ℓ 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 problems. The first one is that the ASCII data (as available via [4]) 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 so they are for the moment of not much interest to us. The letters 'n' and 'N' denote the absence of data, so they are not of much interest either.
Repeating sequences are pieces of DNA that have a repetition with a period of 12 or less.
Large repeating sequences are the result of evolution, and these are not masked in the ASCII data. There are several processes that result in these repeats, one of them being the existence of more copies of a certain piece of DNA in the genome. Apparently, in the course of evolution, particular pieces of DNA got multiplied and copied to a different part of the genome. So we sometimes find a large part of the genome in more than one places.
Then there is a phenomenon named "molecular parasites", they are pieces of DNA that do not code for proteins, and they are "mobile". These pieces get copied and transfer themselves to other parts of the genome, so they can get copied again.
However, we do need to take these letters into account. For example, suppose we are looking for all strings of length 4. We can not convert a string like this: ATATATtntttaatNNN to ttttaat, we have to convert it to tt and ttaat, because otherwise we would also count a substring like tttt, which is not part of the original string.
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, as explained in Section 3.2. We put these offsets in a separate table because we would otherwise have to use escape sequences to differentiate between the DNA data and the offset data. Furthermore, using a separate table is faster for the same reason (we don't have to scan for escape sequences).
In our example we get the two files, one containing the binary equivalent of tttta and one containing the offsets 6, 2, 1, 5 and 3. To decode these files, 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.
An other, more trivial, way is to split the data into small files. The drawback is that for chromosome 1 alone we get 306285 files. This is not very practical.
Fortunately, only 43% of chromosome 1 is useful data, and this simplifies our problem even more (we only need to analyze about 300 Megabytes of data instead of 700, these numbers are extrapolated from Table 4.1). This would bring the total calculation time back to about 300 hours for the multiple passes method. Distribution to about 20 machines will make the calculation finish in 15 hours, about one night ...
Note that we still assume that the computers are Pentium II's at 450 MHz, so we believe we have found an adequate solution. Although the processor speed

is not the issue, we have measured speedups of 4.3 on a Pentium IV at 2.8 GHz with 512M of memory (we only made use of the processor and did not allocate more memory than in our previous tests).

5.1.1 Preparation in practice

We have made a program that converts the ASCII input to binary and skips all non-{`a`, `c`, `g`, `t`} characters. We also have to make a file that has indexes to determine when a strand of DNA is bounded by invalid characters. By doing this we have compressed chromosome 1 (238 Megabytes) to 26 Megabytes of binary input and 1.2 Megabytes of indexes. That is only 11.2% of the size of the original input. For convenience we have added a comment field in our file; this enables the user to put the contents of a custom made *comment file* into the input file. It might be useful to put data like the date, the order of chromosomes and things like that in this field. The options for this preparation program are given in Table 5.2.

Name	default	comment
`-i`	NULL	each `-i` option gives an input file
`-o`	chromFa.bin	name of the output file
`-c`	NULL	name of the comment file

Table 5.2: Preparation options

This program can take more than one input file, because the genome data we use is partitioned in chromosomes. A typical use would be: `-i chr1.fa -i chr2.fa` ...`-i chrM.fa -c comment.txt`, where `chrX.fa` are the chromosomes in ASCII format, and `chrM.fa` is the mitochondrial DNA. The "`-i chr*.fa`" convention can not be used because this would result in a line like "`-i chr1.fa chr2.fa` ... the reason we did not implement this is because this wildcard expansion may give a different order than we would expect. For example; when we have the files 1, 2, 3, ..., 10, 11, 12, the expansion of `*` would result in 1, 10, 11, 12, 2, 3 .... The result of this program is a file with the format described in Table 5.3.

Name	offset	size	comment
ID	0x0	0x9	"binDNA01"
Reserved			reserved for flags and variables
Index pointer	0x100	0x4	pointer to index field
Index size	0x104	0x4	size of index field
Boundaries pointer	0x108	0x4	pointer to boundaries field
Boundaries size	0x10c	0x4	size of boundaries field
Info pointer	0x110	0x4	pointer to info field
Info size	0x114	0x4	size of the info field
Reserved			reserved for pointers
Data	0x200	variable	raw DNA data
Index	variable	variable	indexes
Boundaries	variable	variable	boundaries of chromosomes
Info	variable	variable	the info field

Table 5.3: Input file format

The ID string is for internal purposes, is is used by other utilities to make sure that the user has given the correct file for input. The index pointer points to an array of 32-bits indices and the index size is the number of elements in this array. The boundaries pointer points to a similar array and the info pointer points to the info field.
The index array is the array discussed in Section 5.1, except that the array is no longer in a different file, hence the index pointer and size.
The boundaries array contains the offsets of the different chromosomes, and the info field is discussed above.

5.1.2 Preparation in practice, the final version

Due to new developments in genetics, the input is treated differently than expected. Much of the DNA we used to call "junk DNA" might be interesting after all and genes might consist of fewer nucleotides than expected. That is why we must take the whole genome including the masked repeats, which makes our problem a bit more difficult, about a factor of ten (fortunately this is nothing compared to the complexity of the problem).
So when preparing the data, we only filter out the `n` and `N`'s, and convert other symbols into lower case.

5.2 Double stranded DNA

Our input data is single stranded DNA, but in practice we use double stranded DNA, see Figure 2.2. This introduces a problem. Each time we count the occurrence of a primer, we also have to count the reverse complement of the primer, but since the reverse complement is unique for each primer, it is not necessary to count each primer twice (once for the primer itself and once for

its reverse complement), but counting the pairs is good enough (or even better from a biological point of view).
What we mean by this is that whenever we encounter a string (and its reverse-complement), we do not increase the counter of the string *and* the reverse-complement, but we choose one of the two as a representative of the pair. We do not increase the counter by two when we encounter a certain pair, but by one.
Then there is the issue that we need to determine where the primer pair is stored. Remember that the primers also form a number which is the offset in the output file. We chose to store the information about a primer pair at the lowest offset.

5.2.1 Uniqueness

We use the term uniqueness in the following way. A factor is unique if the factor itself can be found precisely once and if the reverse complement can not be found on single stranded DNA.
An efficient way of calculating the reverse complement is the following; since we keep track of the last ℓ nucleotides, we can also keep track of the last ℓ nucleotide complements in the reverse order. All we do is shift the reverse-complement string to the left (instead of the right) and shift the complement of the read nucleotide in at the most significant side. This also is a rather inexpensive operation.
We identify a pair of complementary strings, and choose one of them as a representative. Since we know at all times the string itself and its reverse-complement, we only have to look at which is the smallest (lexicographical ordered), and this can be done with the standard `<` operator.

5.3 Output

Name	default	comment
`-i`	chromFa.bin	name of the input file
`-o`	chromFa.out	name of the output file
`-c`	NULL	name of the comment file
`-l`	18	length of the primers
`-m`	64	free memory in Megabytes
`-b`	0	begin pass
`-p`	$\frac{4^{\text{length}}}{\text{memsize}}$	number of passes
`-a`	2	compression algorithm 0=none, 1=fast, 2=strong
`-s`	NULL	if given, treat the input data as single strand

Table 5.4: Analysis options

Table 5.4 lists the options for the analysis program. The `-i` option must be followed by a filename created by the preprocessing program. The length of the primers we are looking for can be adjusted and the amount of free memory too. Because this program can be distributed to multiple computers in parallel, the possibility is given to set the upper and lower boundary for the passes, as explained in Section 4.3.1. The compression methods are discussed in Section 5.5. For testing purposes, we can still treat the data as a single strand. The `-b` option must be followed by an integer between 0 and $\frac{4^{\text{length}}}{\text{memsize}}$.

Name	offset	size	comment
ID	0x0	0x9	"binCNT01"
Compression flag	0x9	0x1	indicate compression method
Length	0xa	0x1	length of the primers
Begin	0xb	0x4	begin pass
Passes	0xf	0x4	number of passes
DNA type	0x12	0x1	Single or double DNA
Reserved			reserved for flags and variables
Info pointer	0x100	0x8	pointer to info field
Info size	0x108	0x4	size of the info field
Reserved			reserved for pointers
Data	0x200	variable	raw data
Info	variable	variable	the info field

Table 5.5: Output file format

The output file is the result of the analysis program. The format is shown in Table 5.5. Again, we use an ID string to make sure other programs can verify that their input file is of this format. The compression flag indicates the method of compression as discussed in Section 5.5, it is needed to decompress the file, otherwise the decompression program would not know which method to use. The length of the primers is also included, as well as the begin and end pass. The latter two values are of importance when the analysis has been distributed; the results must be put together again in the correct order.

5.4 Results

The first results, shown in Table 5.6, are obtained from analyzing chromosome 1. The original size of this chromosome is 238M, compressed it is 26M plus 1.2M of indexes. This accounts for about 9% of the total genome.

length	expected output	output (compressed)	passes	time
8	64K	64K	1	15.0s
9	256K	258K	1	19.1s
10	1M	1.1M	1	30.1s
11	4M	4.9M	1	39.8s
12	16M	21M	1	48.6s
13	64M	71M	1	1m 9s
14	256M	175M	2	1m 54s
15	1G	273M	8	3m 59s
16	4G	332M	32	10m 54s
17	16G	364M	128	39m

Table 5.6: Multiple passes

The analysis is done on a Pentium II at 450MHz with 256M of memory, half of which is used as buffer (the `-m` option of our program). The result is compressed before it is written to disk. Note that for small output files, compression results in a larger file because compression (Section 5.5) relies upon the fact that the table is sparse. The compression algorithm starts to do significant compression when we analyze lengths of 14 and above.

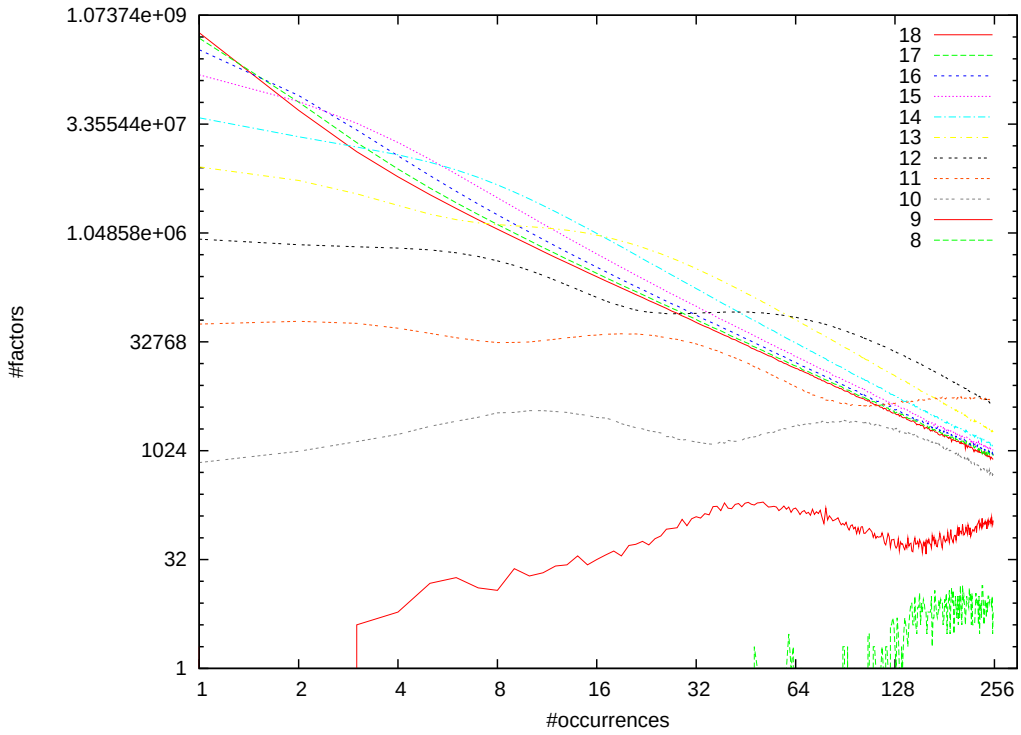

Figure 5.1: The non-repeating part of the genome

In Figure 5.1 we see the result of an analysis done on the non-repeating, single stranded part of the entire genome. In Figure 5.2 we see the result of an analysis done on the entire single stranded genome, including the repeating part. In these figures the number of occurrences are on the x-axis, so all unique primers are on position 1, the ones that occur two times are on position 2. The number of these occurrences are on the y-axis. So for example there are no unique factors of length 9 in Figure 5.1 and there are 4 factors that occur 3 times.

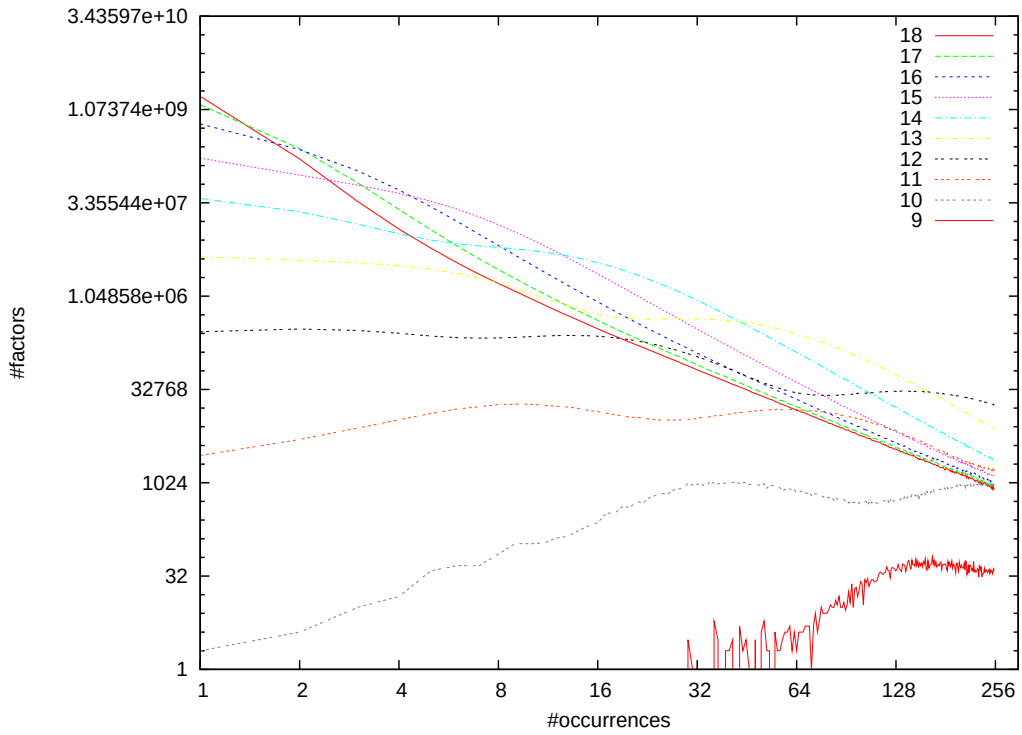

Figure 5.2: The whole genome

In Figure 5.3 we see the results of the first analysis (for length 18) done on double stranded DNA.

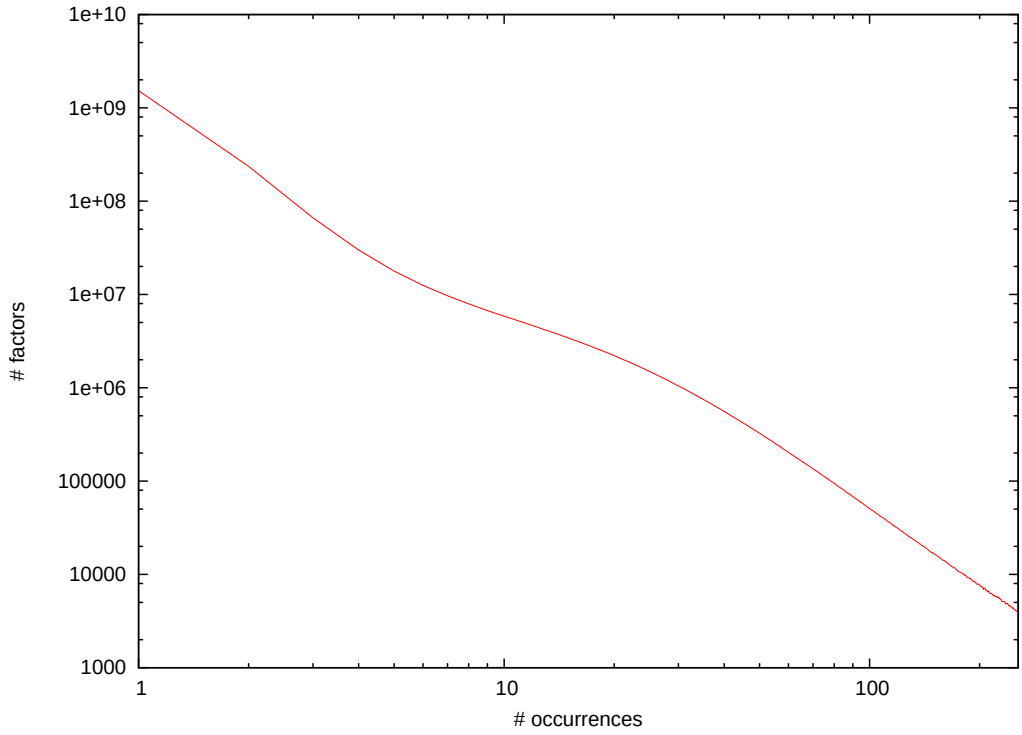

Figure 5.3: The whole double stranded genome (length 18)

5.5 Compression

We have implemented three schemes for the output file (shown in Table 5.5): no compression, fast compression and strong compression. The first one is self-explanatory, the second one goes as follows. If we read a zero, we stop writing until we read a non-zero, then we write a zero and the amount of skipped positions, this technique is sometimes referred to as "run length encoding". If for example we compress the string 022300000033202001 (for simplicity we only count up to 4), we write it as 0$\underline{1}$22306$\underline{3}$320$\underline{1}$20$\underline{2}$1. As you can see, this method assumes that there are a lot of chained zeroes in the output file. A weak point in this method is that we use an integer to count the skipped positions. This is not done in the strong compression algorithm, here we take a byte and if its value exceeds 127, then we take another byte. A string with 130 zeroes would be encoded as 0$\underline{1}$0$\underline{2}$ (128 + 2). This method is especially good when the amount of chained zeroes is not that large (smaller than 127, but close to it) on average.

5.6 Unicity

When we have a unique factor of length N, we must also have two unique factors of length $N + 1$, and these two factors are very similar for they have N nucleotides in common. This means that when we have a unique factor of a certain length, it will result in clusters of unique factors of larger lengths. We could make use of this principle to find unique factors of large length, without doing an extensive analysis. Note that by doing this we shall not find *all* unique factors of this large length in the genome, but since the amount of unique factors of length 18 is vast, we may not need to do analysis on larger lengths because we already have so many factors. Note that the larger the length, the higher the probability that a given factor is unique within the genome. When we take a piece of DNA of length 30, the chance it is unique is very large, except when it is part of a repeating part of the genome, which would also be detected when doing analysis on lower lengths. Formally, we can say that if the string x is unique then any string on the genome containing x is also unique.

5.7 Shortcomings.

There is really only one shortcoming in the solution we chose, and that is that the counting information is separated from the genome. To solve this problem we chose to reserve two bits after each nucleotide in the input. We have decided to count to 4 instead of 256, his is done because counting to 4 is more than enough to determine unicity. This output format is quite different from the one we discussed earlier.
Remember that `a`, `c`, `g` and `t` are encoded as 00, 01, 10 and 11 respectively. So instead of `atgcagat` we write `a0t0g0c0a0g0a0t0` (0 is 00 in bits), which makes the input file twice as large as previously. The idea behind this is that the information can be re-integrated with the input again. Let us say that we are looking for factors of length 3. When we see a string like `a0t0g3c2a0g1a1t0` it means that the string `atg` occurs 4 times, `tgc` occurs 3 times and so on. We shall refer to this inserted data as a *mask*. Note that we denote the number of times a factor is present in the genome by $N - 1$ if it is present N times. We do this because in this representation, we will not encounter strings 0 times and it would be a waste when we did not make use of the number 0, which is, in this case, where we use two bits for counting, $\frac{1}{4}$ of the possible permutations. This also means, however, that we *do* need to use more bits in the actual counting part of the analysis. We must initialize the memory to a default value (usually 0) and this value can no longer be used in the counting process. That is why we use 4 bits to count the data in memory. We chose this particular number because it is still rather simple to implement; when using 3 bits this would be far more difficult, and it would require more processing time, counteracting the gain we get from the increase in memory positions.
The borders of the chromosomes and the borders determined by `n`'s pose a small difficulty. When for example we count all occurrences of length 18, the first 17

positions must be skipped. We chose to keep the counting data in its initial state and to store the analysis length (in this case 18) in the header of the file. A program that reads this file must take this problem into account.
In order to produce this output, we doubled the number of passes, for each pass to analyze the input, we need an additional pass to re-integrate the results in the form of a mask.
An additional advantage of this technique is that the output file is always of the same length ($3 \times 10^9 \cdot 4$ bits).

Chapter 6

The final approach with an extension

6.1 The file format for the final multiple passes solution

Now we discuss the file format for the multiple passes method, this is the solution we ultimately chose to use, because this method has the best performance.

Name	offset	size	comment
ID	0x0	0x9	"binDNA01"
Output flag	0x9	0x1	0 if it is DNA data, 1 if it is DNA data + mask
Reserved			reserved for flags
Length	0x50	0x4	length of the primers
Reserved			reserved for variables
Index pointer	0x100	0x4	pointer to the index field
Index size	0x104	0x4	size of the index field
Boundary pointer	0x108	0x4	pointer to the boundaries field
Boundary size	0x10c	0x4	size of the the boundaries field
Info pointer	0x110	0x4	pointer to info field
Info size	0x114	0x4	size of the info field
Reserved			reserved for pointers
Data	0x200	variable	raw data
Info	variable	variable	the info field

Table 6.1: Input/output file format

The output flag denotes whether or not the data has been processed yet. If this flag is 0, the value of the Length variable is meaningless. The length variable denotes the length of the unique primers we searched for. The index pointer

points to an array of indexes; these indexes denote the pieces of DNA bounded by either chromosome boundaries or unknown nucleotides (n). The size of this array is also in the header because this is rather convenient for memory allocation. The boundaries pointer is similar to the index pointer, except that the array it points to only denotes chromosome boundaries. The info pointer points to the information field.
At three places in the header space is reserved for future expansions; there is room for flags, variables and pointers. We chose not to mix these things because the file format is more structured like this.

6.1.1 Some statistics

We ran the `Mpass` program on a Pentium IV at 2.8 GHz with 256 Megabytes of available memory. Table 6.2 shows the runtimes for length 9 ... 18. Analysis has been done on the entire genome including the mitochondrial DNA and the repeating sequences.

Length	passes	runtime
9	1	7m
10	1	8m
11	1	13m
12	1	17m
13	1	20m
14	1	22m
15	2	31m
16	8	1h 20m
17	32	4h 37m
18	128	21h 6m

Table 6.2: Runtimes for the `Mpass program`

Figure 6.1 shows the same data, represented graphically. We can see that at length 15 and above, we have run out of memory and we need to do more than one pass over the input. Hence the graph gets more steep from this point on.

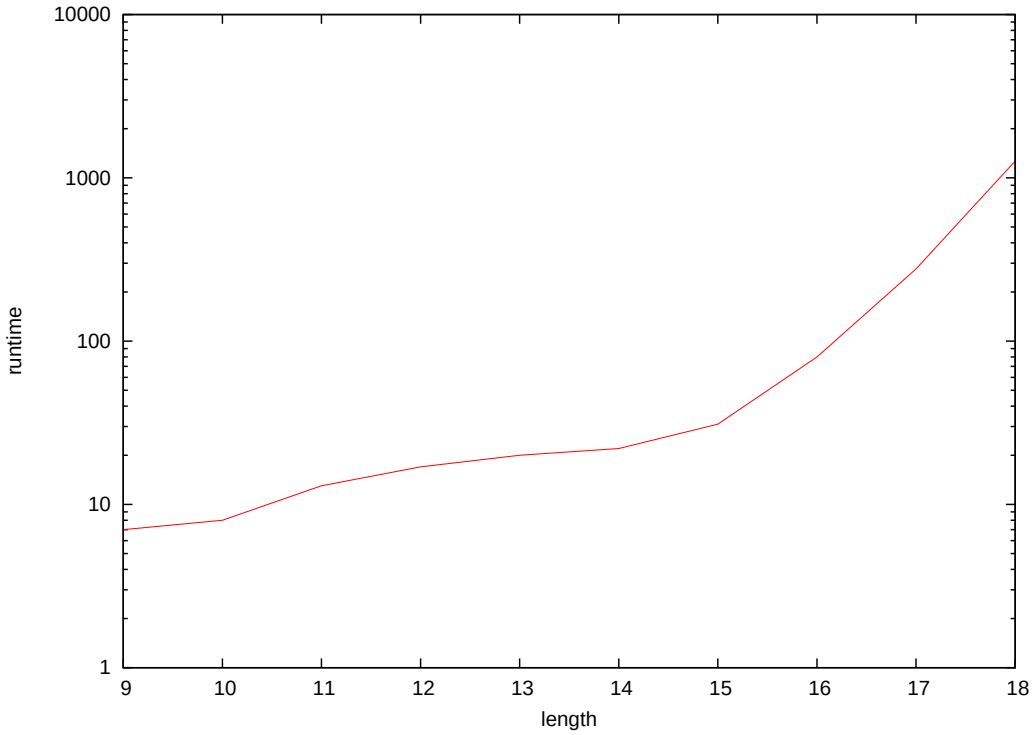

Figure 6.1: Runtimes in minutes for the `Mpass program for factors of given length`

Figure 6.2 essentially shows the same data as Figure 5.3, but this time we show the results for lengths 11 ... 18. We could not do this before, because we did not have enough disk space to store the results of the analysis. With the new (masked) approach, we have more than enough disk space to do this.

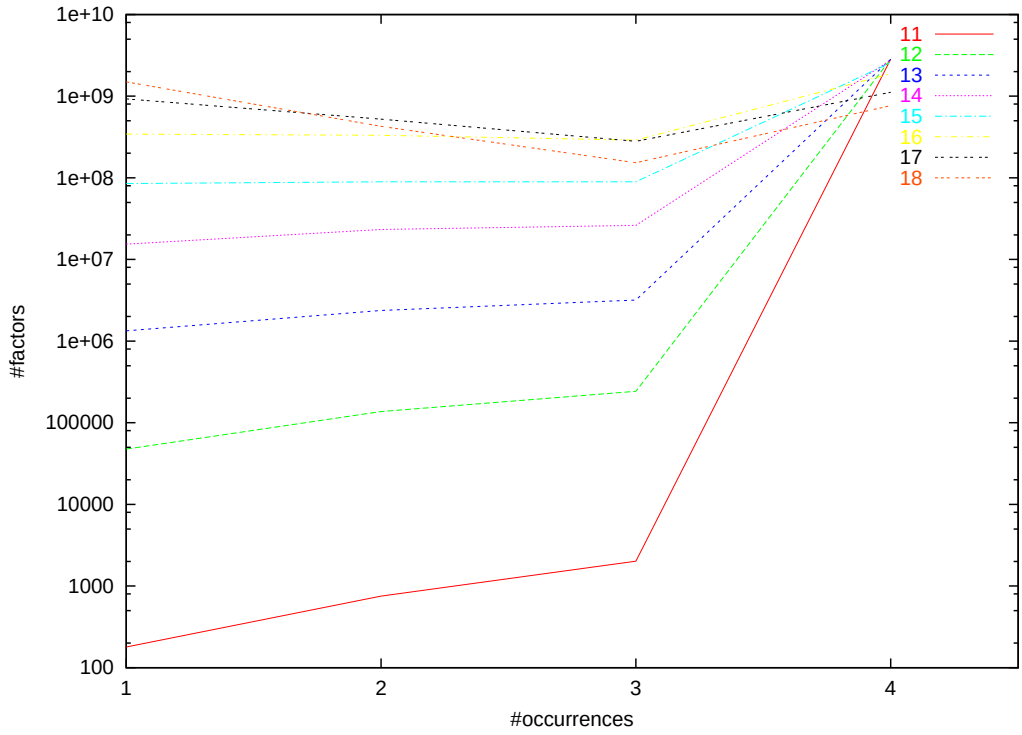

Figure 6.2: The entire genome counted

The "18"-line crosses the y-axis at point 1497154670, which means that 52% of the number of factors of length 18 in the human genome are unique.

6.2 Filtering

After having solved the counting problem, we took on the other problems, like determining the GC-percentage and the *melting temperature*, the temperature at which a string of DNA lets go of its reverse complement.
We chose to keep track of both variables on the fly while reading in each new nucleotide, instead of calculating them over and over again. This was done because most factors we are looking for are unique.

6.2.1 GC-percentage

Because we already make the reverse complement of the input string on the fly (just like we did in the first stage), keeping track of the GC-percentage is not very difficult. The only thing we need to do is observing what we shift out and what we shift in. So when we look at the least significant nucleotide of the reverse complement, and this is a `c` or a `g`, we decrease the GC-count; then we

shift in the new nucleotide, and if the least significant nucleotide on the original strand is a c or a g, we increase the GC-count. The percentage, of course, is calculated by dividing the GC-count by the length.

Example 6.2.1. We determine the GC-percentage of all strings of length 4 in the string attagcaaga.

string	change	GC-percentage
atta		$\frac{0}{4}$
$\underline{\texttt{a}}\texttt{tta}\overline{\texttt{g}}$	$+1$	$\frac{1}{4}$
$\underline{\texttt{t}}\texttt{tag}\overline{\texttt{c}}$	$+1$	$\frac{2}{4}$
$\underline{\texttt{t}}\texttt{agc}\overline{\texttt{a}}$		$\frac{2}{4}$
$\underline{\texttt{a}}\texttt{gca}\overline{\texttt{a}}$		$\frac{2}{4}$
$\underline{\texttt{g}}\texttt{caa}\overline{\texttt{g}}$	$-1+1$	$\frac{2}{4}$
$\underline{\texttt{c}}\texttt{aag}\overline{\texttt{a}}$	-1	$\frac{1}{4}$

The underlined nucleotides are shifted out, the overlined ones are shifted in.

We actually use both the string itself and its reverse complement. This is because it is faster to extract the least significant bits (this operation is only one bitwise AND, otherwise a SHIFT of variable size must be used). So instead of observing what is shifted out at the most significant bits of the original string we observe what is shifted out at the least significant bits of the reverse complement. We can use this technique because a g or a c on one end of the fragment implies a c or a g on the other end of the reverse-complement.

6.2.2 Temperature

An algorithm to calculate the melting temperature of a piece of DNA is given in [7]. We chose to re-write this algorithm to act like the one calculating the GC-percentage. By doing this we can easily and swiftly calculate the temperature on the fly for many strings of DNA in succession. Another program that calculates melting temperature and GC-percentages is Primer3 [8]. The formula for the temperature is mainly based on the number of successive pairs occurring in the fragment. We only have to look at the two last nucleotides on the reverse complement (the first ones to be shifted out), do a lookup in a table, and subtract the result from the temperature. Then we do the shift and increase the temperature with the result of a lookup in a table while we look at the two last nucleotides of the original string (the last ones that have been shifted in). Because internally we use integers to keep track of the temperature, we do not encounter rounding errors.

Interaction	ΔH	ΔS
aa/tt	9.1	24.0
at/ta	8.6	23.9
ta/at	6.0	16.9
ca/gt	5.8	12.9
gt/ca	6.5	17.3
ct/ga	7.8	20.8
ga/ct	5.6	13.5
cg/gc	11.9	27.8
gc/cg	11.1	26.7
gg/cc	11.0	26.6

Table 6.3: Nearest-neighbour thermodynamics

Example 6.2.2. We determine the melting temperature of all strings of length 6 in the string attagcaaga. We start with the initial calculation

(at) + (tt) + (ta) + (ag) + (gc)

After this we can re-use the result like we did with the GC-count.

string	lookup
attagcā	– (at) + (ca)
ttagcaā	– (tt) + (aa)
tagcaag̅	– (ta) + (ag)
agcaagā	– (ag) + (ga)

Again we use both the string itself and its reverse complement, for the same reason we did this when calculating the GC-percentage. Also analogous to determining the GC-percentage we assume that the melting temperature is invariant under reverse-complementation, which must be the case, because it is in the definition of melting temperature (and it can also be seen from the lookup tables we use, see Table 6.3).

6.3 Filtering simple repetitions

Simple repeats are not of much interest to biologists, that is why we need to filter them out.
Repeats can be denoted in the following way:

- A letter from the alphabet is denoted with n.
- A string of a certain length is denoted with $(\underbrace{n \dots n}_{m})$ with $m \in \mathbb{N}^+$.

- A repetition of a string or letter is denoted by an exponent.

So (nn) means all strings of length 2. And $(nnn)^2$ means all strings of length 6 with substrings of 3 repeated 2 times.

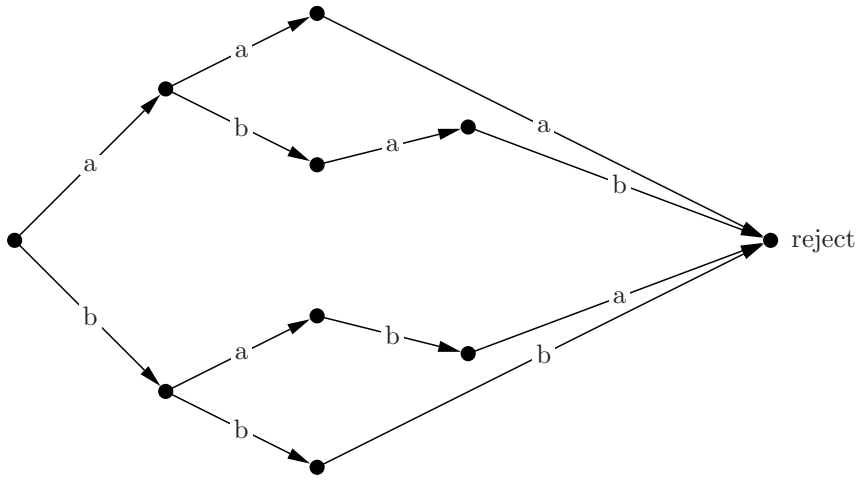

Figure 6.3: Unwired repetition trie

Figure 6.3 shows an unwired trie to filter out the repetitions $(n)^3$ and $(nn)^2$ over the two letter alphabet $\{a, b\}$, 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. Note that this trie is not yet *wired*. The wires are failure links, they point to an other possible repeat if any, and otherwise they point to the root. The trie shown in Figure 6.4 is technically speaking not wired, because the wires have already been converted into alternatives.

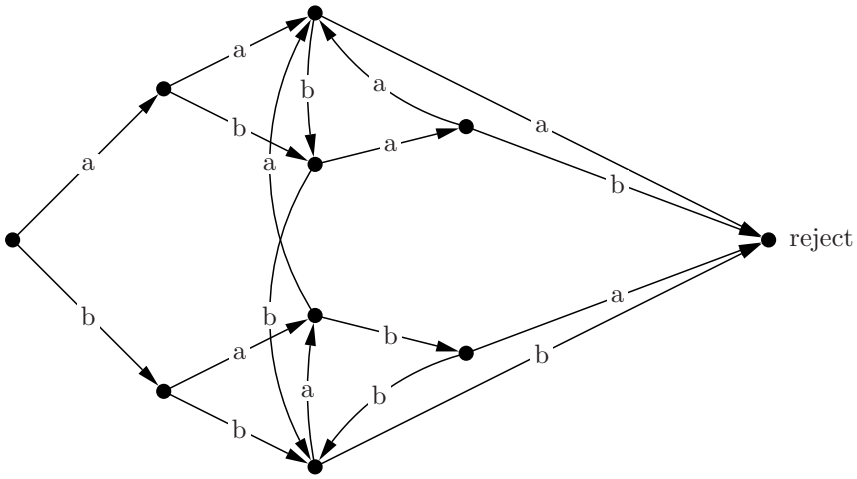

Figure 6.4: Wired repetition trie

We use a trie to filter out these repeats, but the standard Aho-Corasick method fails in this case, because we may insert a repeat which has a repeat in it as substring. The problem is that the strings we use to build the Aho-Corasick trie may not be substrings of each other. To prevent this we filter out the strings that have substrings which are repeats themselves.

Example 6.3.1. For simplicity we take a two-letter alphabet $\{a, b\}$.
If we make a trie which filters out the strings $(n)^2$ and the strings $(nnn)^2$, we get a trie which has the paths aa and $baabaa$ leading from its root to a final state. The first path aa is an instance of the rule $(n)^2$ and the second path is an instance of $(nnn)^2$.
The trie should filter out all occurrences of the string aa, but a string like $baabab$ is permitted by the trie because when matching this string, we see that the first two letters do not match the rule $(n)^2$ (it does not start with either aa or bb). It does, however match the first five letters of one of the $(nnn)^2$ instances ($baabaa$). Because the last letter does not match, the string is permitted by the trie, even though it has aa as a substring.

A way to solve this problem is to build the trie in a different manner. What we need to do is checking the repeats that we insert into the trie for repeats

themselves. We can use the trie for this, provided we insert the repeats in order of length, the smallest ones first.

Example 6.3.2. Like the previous example, we use an alphabet $\{a, b\}$ and we make a trie which filters out the repetitions $(n)^2$ and $(nnn)^2$.
First we insert all strings in $(n)^2$: aa and bb.
Now we insert the strings in $(nnn)^2$. When we encounter the repetition $baabaa$ the trie will reject this repetition, because it has the substring aa in it (twice). So this particular repetition may not be inserted.

By testing the repetitions for repetitions of lower length, we insert only those repetitions that add filtering rules. Leaving out the other repetitions does no harm either, because the strings they would filter out are filtered out already by the same rules that disqualified the repetition in the first place.

6.4 Pairing

The medical engineers wanted to have a primer set that (roughly) had the following characteristics:

- The primers must be of length 18.
- Each primer must have a GC-percentage between 20 and 80 percent.
- Each primer must have a melting temperature between 60° and 63° Celsius.
- Each primer pair must have 480–520 nucleotides between them.
- The space between the pairs must be about 10000 nucleotides.

To do this we chose to search for a primer, then skip 480 nucleotides and search for the next primer. If the next primer is found within the next 40 positions, we have successfully formed a pair. Otherwise we go back to the first primer, throw it away and start all over again.
Unlike the Primer software we do not check the primer pairs for similarity (using alignment scores). Although it can be done for each pair separately, it is unfeasible to check all primers found (for the genome) for mutual similarities.

Chapter 7

Utilities

Now we shall briefly discuss the programs written for our purposes. Note that most of the discarded ideas are also implemented, but they are not discussed here.

7.1 Frontend

The first programs written focus on preprocessing. These are fast programs that simplify the core analysis by converting the DNA to binary data, thereby making it possible to use fast binary operators instead of string operators.

7.1.1 `Comb`

This program converts ASCII input to binary input, and it also reserves space for the output of the `Mpass` program. By doing this conversion beforehand, we do not have to do conversions in the critical phase of the core analysis; it also reduces the size of the input data by a factor of at least 4. We say at least because all unknown nucleotides and known repeats (multiple `n`'s) are filtered out, but the number of `n`'s is kept, otherwise we would lose the offset in the genome. So if there are a relative large number of `n`'s present, we reduce the filesize even more.
This program recognizes the following options:

option	function	comment
`-i filename`	name of an input filename	multiple `-i`'s may be given
`-o`	name of the output file	
`-c`	name of the comment file	optional

The program can take more than one input file, because our input data consists of multiple files (chromosome 1, chromosome 2 and so on).

7.1.2 Dcomb

This program converts binary input data to ASCII. Thus of every four bits, the first two are decoded into a nucleotide, the other two bits are discarded. The sequences of `n`'s are displayed as the number of `n`'s followed by the letter `n`. The program is used mainly for debugging purposes, but it could also be used to view the binary file.
This program recognizes the following options:

option	function
`-i filename`	name of the input filename

The output is written to standard output. No extensive work is done on this program because it is not used often in practice.

7.2 Analysis

7.2.1 Mpass

The core program, that counts the occurrence of factors of a certain length. It is named `Mpass` because it usually makes more than one pass over its input.
This program does not create an output file (like it did in previous versions, see Table 5.5). Instead it manipulates the input file by filling up the zeroes inserted by `Comb`.
This program recognizes the following options:

option	function	default value
`-i filename`	name of the input and output filename	chromFa.bin
`-l`	length of the factors to search for	18
`-m`	amount of available memory in Megabytes	64

7.2.2 Repeats

This program filters out the following simple repeats.

- $(n)^5$
- $(nn)^5$
- $(nnn)^4$
- $(nnnn)^3$
- $(nnnnn)^3$
- $(nnnnnn)^3$
- $(nnnnnnn)^2$

- $(nnnnnnnn)^2$
- $(nnnnnnnnn)^2$

This program only takes one pass over the input data.

7.2.3 `Search`

The second stage in analysis. It filters out unique primers for which the GC-count and/or temperature is not good. This program only takes one pass over the input data.
This program recognizes the following options:

option	function	default value
`-i filename`	name of the input filename	chromFa.bin
`-g`	minimal GC-percentage	20
`-h`	maximum GC-percentage	80
`-t`	minimal melting temperature in degrees Celsius	60
`-u`	maximum melting temperature in degrees Celsius	63
`-d`	concentration of annealing primers in mol/ml	50
`-e`	salt concentration in mol/ml	50
`-n`	minimal internal size	480
`-m`	maximum internal size	520
`-x`	minimum external size	9800

The unique factors we have found must be subjected to more filters in order to be used in practice. Since these filters will reduce the amount of approved factors, it could be that the number of approved factors is reduced to such a small number that there is nothing left to work on. Therefore we have made these options available on the command line, to "play" with.
Some of these options need some more explanation.
The GC-percentages options throw out all factors that have a GC-percentage outside the given boundaries. The same applies to the melting temperatures. The concentration of annealing primers option is used in the calculation of the melting temperature, as is the salt concentration option.
This program does not give all primers that have the right GC-count and temperature. Under normal circumstances there would still be far too many primers found. For practical applications with primers, we only need some primer pairs, with a certain space between them. That is why we implemented the internal size options. When a primer is found, we skip an amount of nucleotides equal to the minimal internal size given on the command line and search for the second primer. When the maximum internal size is reached before a primer pair could be found, we discard the *first* primer, skip one nucleotide and repeat the pairing process.
The last option discussed here is the minimum external size option. This is the minimum size between the right primer of one pair and the left primer of the

next pair. This option is used to skip a number of nucleotides after a pair has been formed and before a new pair must be formed. There is no maximum to the size that can be skipped.
The output of this program is written to standard output.

7.3 Backend

7.3.1 `Dcall`

This program is very similar to the `Dcomb` program, but it also shows the data in between. It also recognizes the same options as the `Dcomb` program does. Like `Dcomb`, this program is used mainly for debugging purposes. The main advantage over `Dcomb` is that *after* analysis by `Mpass` you can see the result of the analysis with this program. The output is written to standard output.

7.3.2 `Getinfo`

This program extracts the information field from a binary data file. This information field is either set by `Comb`, or `Setinfo`. It only recognizes the `-i` option, with which the input file is passed to the program. The content of the information field is written to standard output.

7.3.3 `Setinfo`

This program can manipulate the info field of a binary data file. It is very useful if for some reason the information field has the wrong data. This can easily happen when doing an analysis which manipulates the data file. Things like this can be stored in the data field.
The program recognizes the following options:

option	function
`-i filename`	name of the input/output filename
`-c`	name of the comment file

Since the comment file is a plain text file, it can be used to store anything. We ourselves put the following data in it:

- The name of the person who has made the file.
- The date the file was created.
- The contents of the file (which chromosomes and in what order).
- Whether or not analysis has been performed already on the data.

The last item is strictly speaking not needed, because the data file has its own flag to see if analysis has been performed or not, but this is still convenient for the end-user.

7.3.4 `Statistics`

This program counts how many factors are unique, how many are present two times in the genome, how many are present three times and how many are present four times or more. This program only recognizes the `-i` option for the input filename.

Chapter 8

Conclusions

We conclude that determining all unique primers in the genome is possible up to a certain length. In principle, this problem should be solvable in constant time, regardless of the length of the primers. If we were able to store the primer and the times we encounter it in main memory, then the problem is solved. Because of memory limitations, we had to take an approach that required an exponential amount of memory, depending upon the length of the primers. This approach however, performed better at lower primer lengths. Fortunately, these lengths were sufficient.
The post-processing, consisting of determining the melting temperature, the GC-count and filtering out simple repetitions can already be done in constant time. The output is, because of the way we re-integrate it in the genome, of constant length, regardless of the lengths of the primers we are counting, but dependent on the maximum value we wish to count up to.

Chapter 9

Further research

The selected primers could be aligned to each other, alignment is a technique to check for similarities between primers. If two primers are too much alike, both should be disqualified as primers. Since we search for unique primers, we shall never find two primers that are perfectly aligned, but mathematical uniqueness is no guarantee for good results in practice. A program that aligns large quantities of primers is `Bulkal` [9], this program is part of a modified Primer3 distribution.
The program `Bulkal` could also be combined with the `Search` program. If we keep track of which primers have been selected, and let `Bulkal` test newly found primers against the list, we should end up with a good list of primers if we throw the primers that have a high alignment score off the list.
The `Mpass` program could be altered to be distributable over several processors, as suggested in Section 4.3.1, this way we could find unique factors of even greater length.
As mentioned in Chapter 8, when computers in the future have enough memory capacity, a program can be written to solve the counting problem in constant time. A hash table with 3×10^9 entries would suffice.
The selection of primers could be done on lower lengths before doing so on the target length. By doing this, we avoid selecting unique primers that are the result of uniqueness of a substring of it. This would in turn simplify a possible alignment post-processing step, because primers with a common substring will have a high alignment score.
It might be a good idea to gather some statistics on the primer pair selection process (the `Search` program). For example, how large the actual gaps are between the primer pairs (external size).

Bibliography

[1] Schouten, J.P., McElgunn, C.J., Waaijer, R., Zwijnenburg, D., Diepvens, F. and Pals, G. Relative quantification of 40 nucleic acid sequences by multiplex ligation-dependent probe amplification. Nucleic Acid Research 30, No. 12, e57:1–13, 2002.

[2] Dieffenbach, C.W. and Dveksler, G.S. PCR primer: A laboratory manual. CSHL Press, Cold Spring Harbor, USA, 1995.

[3] Yamada, T. GenomeSURF: Genome-wide Swift Unique Region Finder, http://surf.gi.k.u-tokyo.ac.jp/

[4] UCSC Genome Bioinformatics, http://genome.ucsc.edu/

[5] Knuth, D., Morris, J., and Pratt, V. Fast pattern matching in strings. SIAM J. Computing 6, 323–350, 1977.

[6] Aho, A. and Corasick, M. Efficient string matching: An aid to bibliographic search. Communications of the ACM 18, 333–340, 1975.

[7] Breslauer, K.J., Frank, R., Blöcker, H. and Marky, L.A. Predicting DNA duplex stability from the base sequence. Proc. Natl. Acad. Sci. USA 83, 3746–3750, 1986.

[8] Rozen, S., Skaletsky, H.J. Primer3, http://www-genome.wi.mit.edu/genome_software/other/primer3.html, 1996, 1997, 1998.

[9] Laros, J.F.J. Primer design for MPLA experiments. Project report, Leiden University, 2005.