

Pattern Search in a Single Genome

Nuffield Science Bursary Project by
Michelle Parker

Supervisor: Professor Jotun Hein
Oxford Centre for Gene Function
Oxford University Statistics

Abstract

Searching for patterns in a string is an important challenge for biologists. Identifying and finding patterns helps to improve understanding of DNA and as a consequence enables advances in research of medical conditions. There are many different methods for searching for a single pattern in a string of DNA but the problem is finding a technique which is both efficient and realistic. Different string searching techniques of both exact string searching and approximate string searching were implemented and the efficiency and number of matches of these methods were compared.

August 2009

Table of Contents

Introduction	3
Regulatory Sequences	3
Project Brief	3
Exact String Matching	4
Naïve Algorithm	4
Rabin-Karp Algorithm	5
Knuth-Morris-Pratt Algorithm	7
Approximate String Matching ¹	9
Substitution	9
Insertions and deletions	10
Position Weight Matrices	13
Longest pattern which appears k times	14
Suffix Trees	14
Results	15
Naïve String Algorithm matching techniques	15
Rabin-Karp Algorithm	16
Naive, Rabin-Karp and Knuth-Morris-Pratt Comparison	17
Substitution	18
Insertions and deletions	19
Position Weight Matrices	19
Suffix Trees	20
Conclusion	21
Evaluation	22
Future Work	23
Appendix	24
DNA Substitution matrix	24
Results table to show the length of the longest string which is repeated k times	24
Programs	26
References	34
Acknowledgements	34

Introduction

Regulatory Sequences

In the human genome, just over 3 billion DNA base pairs occupy 23 chromosome pairs. DNA has a variety of known roles including protein-coding genes, non-coding RNA genes, regulatory sequences, repeat elements, and extensive regions of DNA which remain unknown. When genes are expressed, information from a gene is used to synthesise the gene product, which could be a protein or functional RNA. Gene expression is a complicated process involving several steps. Transcription is carried out by the enzyme RNA polymerase which creates a complementary string of RNA to a specific strand of DNA which makes up a particular gene. After being modified, RNA moves out of the nucleus and binds to a ribosome. In the process of translation, the ribosome synthesises the protein by binding together the correct amino acids which are coded for by a triplet of nucleotides.

This complex process requires regulation at every stage. Transcription factors are proteins which help with this process by activating or repressing the function of RNA polymerase. They are regulated by sequences of DNA situated a short distance away from the gene. Transcription factors bind to these regulatory sequences which can have a positive or negative affect on gene expression. Regulatory sequences are very important in controlling when and where the gene is expressed.

Searching for and identifying these regulatory sequences is an important biological challenge. These DNA motifs are usually 5 to 20 base pairs long, and can be exact or variable. The variabilities of the motifs makes identifying them very challenging. Patterns are not always exact and a search would have to take into account mutations; substitutions and insertion-deletions of single letters in the motif. Often not all of the motif is known or none of the motif is known, just the region in which there should be one. This means a computer program could be designed which finds new sequences which could be a new motif as well as finding the position of known motifs. Position weight matrices (PWMs) are also often used. These are calculated by comparing the frequency of each nucleotide in a certain position of the pattern to the background frequency of each nucleotide. PWMs give each letter a weighted score in each position and can be used to search and identify motifs.

The most recently established searching tool is the Basic Local Alignment Searching Tool (BLAST).

The Basic Local Alignment Search Tool (BLAST) finds regions of local similarity between sequences. The program compares nucleotide or protein sequences to sequence databases and calculates the statistical significance of matches. BLAST can be used to infer functional and evolutionary relationships between sequences as well as help identify members of gene families.

<http://blast.ncbi.nlm.nih.gov/Blast.cgi>

Project Brief

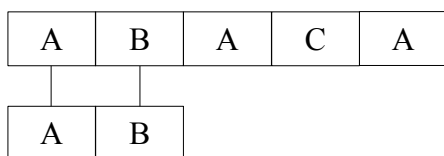
The aim of the project is to implement different ways to search for patterns within a string using Python. This would involve implementing variations of the naive string search would check for occurrences of the pattern in each position of the string, and implementing faster methods for pattern searching for example the Rabin-Karp algorithm and the Knuth-Morris-Pratt algorithm. Other methods can be used to search for non-exact signals, for example the naive string search can be modified to allow substitution, and local alignment can be used to allow a matches which contain substitutions and insertion-deletions of single letters.

These techniques can be applied to signal searching in DNA, for example searching for the common promoter signal TATAAA in chromosome 21 and using position weight matrices (PWMs) to find and rank the most probable strings.

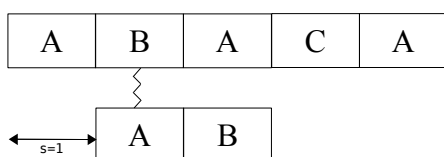
Exact String Matching

Naive Algorithm

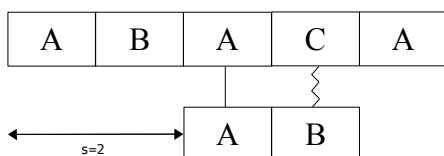
The naive algorithm is probably the most basic exact string search. Each character of the pattern is compared to a substring of the text which is the length of the pattern, until there is a mismatch or a match. If there is a mismatch, the pattern is shifted one position to the right and comparison is repeated for each character. If there are no mismatches before the program gets to the end of the pattern, then there is a match.



The pattern is AB and the string is ABACA. When the shift is 0, there is a match in both the first position and second position of both the pattern and the substring therefore there is an overall match.



When shift is 1, there is a mismatch in the first position of the pattern and the substring, so pattern shifts one position to the right without checking the second position.



When shift is 2, there is a match at the first position, and a mismatch in the second position, so pattern shifts one position to the right.

Figure 1: Naive algorithm

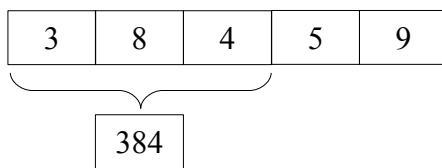
Three different matching techniques were created to be used within the code. The first matching technique uses a for loop which checks each position in the pattern in the substring from 0 to the length of the pattern. The second matching technique uses slices which checks if the pattern equals the substring in the text which is the length of the pattern. The third matching technique uses a while loop where the position in the pattern begins at 0. The position is checked to see if it is less than the length of the pattern then each position in the pattern and substring is matched. If this is carried out the position increases by one. All three methods were tested by searching for the Forkhead-box which is a pattern of length 3, TATA-box which is a pattern of length 6, the MADS-box which is a pattern length 10, and a nuclear receptor which is a pattern of length 12 in the human chromosome 21 to test which method had the fastest running time and to see how the length of the pattern affects the speed of each method.

The naive string algorithm is inefficient because information gained about a character in the text at a particular shift is not used at a different shift. This means that a single character in the text will be compared to the pattern more than once and every possible shift will be tested. The naive algorithm should be slowest when there are many mismatches which have similar prefixes to the pattern and fastest when an overall mismatch has a different prefix. This gives rise to an expected running time of $\Theta(nm)$, where n is the length of the text and m is the length of the pattern. When dealing with long patterns and long text the naive algorithm can be impractical so other methods of string matching with faster running times would be needed.

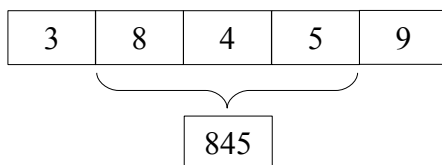
Rabin-Karp Algorithm

The Rabin-Karp algorithm is a faster technique than the naive algorithm because it does not compare each character of the string and pattern more than once. The algorithm uses hashing to convert the pattern and each substring into a value. If the value of the pattern and the substring does not equal, the two strings definitely do not match. If the value of the pattern and the substring equal, the two strings might match. The two strings are then checked to see if they match.

The reason this method is theoretically faster than the naive algorithm is because substrings which are next to each other are related; there is simply one character taken off the beginning and one character added to the end when the pattern is moved along one shift to the right. This can be utilized to create a formula which calculates the next hash value by using the previous one. This means one character is not used more than once to create several hash values. An example can be taken of text which is a list of digits.



If the pattern was length 3, 3 hash values would be calculated of the 3 possible matching substrings in this text. The first substring would be a list of integers 3,8,4. The hash value would then be converted into the number 384.



The hash value of the second substring could be calculated by a simple formula.

$$845 = 10(384 - 3 \cdot 100) + 5$$

Figure 2: Rabin-Karp hashing technique

Each subsequent hash value can be calculated from the previous hash value once the first has been calculated. This can be done using the general formula

$$V_{s+1} = 10(V_s - 10^{m-1} T[s+1]) + T[s+m+1],$$

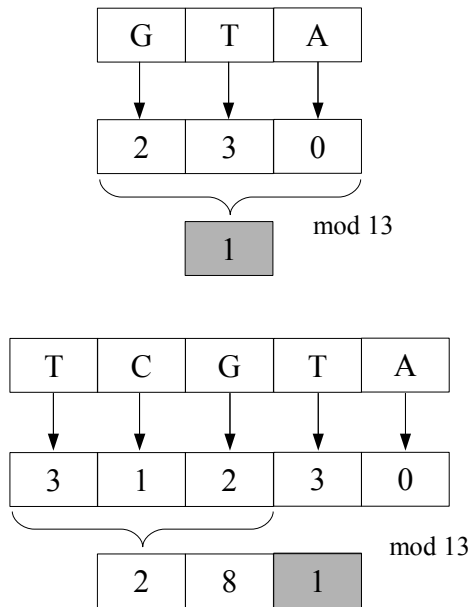
where V is the hash value, s is the shift, m is the length of the pattern, and T is used to represent a slice of the text. In the Figure 2, the third hash value can be calculated from the second hash value using the general formula. $V_s = 845$, $s = 1$, $m = 3$.

$$\begin{aligned} V_{s+1} &= 10(845 - 100 \cdot 8) + 9 \\ &= 459 \end{aligned}$$

To calculate each consecutive hash value, only a constant number of operations is needed which produces an expected running time of $\mathcal{O}(n+m)$. However, long patterns can cause a problem because the resulting hash value could become too large to be computable. This problem can be solved by using a modulus to group large values together and map them on to smaller values. This can increase the number of spurious hits but using large prime numbers as the modulus can result in a more even spread of values.

This technique can also be used in strings. Each character can be converted into a corresponding number and instead of multiplying by 10, the numbers can be multiplied by the radix, which would be the size of the alphabet used. In DNA there are only four characters; A, C, G and T so the radix would be 4. This gives rise to an adjusted general formula where d is the radix and q is the modulus.

$$V_{s+1} = (d(V_s - d^{m-1} T[s+1]) + T[s+m+1]) \bmod q.$$



Pattern of length 3 is converted into numbers using the dictionary {A:0, C:1, G:2, T:3}. The list of numbers is converted into the number 14 using the radix 4 and a modulus of 13 is taken.

Each hash value is calculated and then compared to the pattern. If the number is the same this means the pattern is checked against the substring to ensure they are a match. In this example the number 1 is the hash value of the pattern and the third substring in the text.

Figure 3: Rabin-Karp algorithm

Spurious matches are likely to occur if the modulus used is small and the pattern length is very large. These matches occur when the hash value is the same as that of the pattern but the string is not identical. The final check makes sure that these spurious matches are filtered out. Checking spurious matches take time relative to the length of the pattern so the less spurious matches there are the faster the running time.

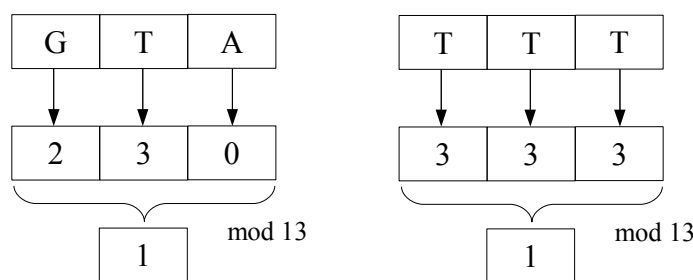


Figure 4: Spurious match

Two different methods of converting the string into numbers has been tested and compared to see which method is faster. The first method uses the ordinal number of the character and the second method creates a dictionary of all the characters contained in the string and pattern. In the first method the radix is calculated by subtracting the minimum ordinal number from the maximum ordinal number. In the second method the radix is the size of the dictionary. A slightly more efficient implementation of the Rabin-Karp algorithm using the dictionary method has also been tested against the originals to see if the slight modifications have affected the speed in any way. Different values of the modulus have also been tested.

Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt algorithm is also a faster technique than the naive algorithm because it does not re-examine characters which have been previously matched. The algorithm uses a prefix function for the pattern which computes how the pattern matches against shifts of itself. This enables the algorithm to miss out useless shifts of the pattern and speeds up the matching process. This prefix function iteratively calculates the length of the longest prefix which is also a suffix at each point along the pattern. This information is precomputed and represented in the array π .

i	0	1	2	3	4	5
$P[i]$	A	B	A	B	C	A
$\pi[i]$	0	0	1	2	0	1

Figure 5: Knuth-Morris-Pratt Prefix Function

In Figure 5, the pattern is *ababca*. The first character is *a*, which is of length one and therefore there is no suffix or prefix so the length of the longest prefix which is also a suffix is 0. The second string considered is *ab*. The prefix is *a* and the suffix is *b*. They are not the same so the π value is also 0. The third string considered is *aba*. Here, there is a prefix is *a* and the suffix is *a*. The length of the longest prefix which is also a suffix is therefore one, because the character *a* is of length one. The fourth string considered is *abab*. The longest prefix which is also a suffix is now *ab*, so the π value is 2. This precomputation has a worst running time of $\Theta(m)$ where m is the length of the pattern.

The Knuth-Morris-Pratt matcher is the second part of the Knuth-Morris-Pratt algorithm. This part refers to the array created by the prefix function while matching the pattern to the text. The prefix function and the matcher are very similar. The prefix function compares the pattern against itself and the matcher compares the text against the pattern.

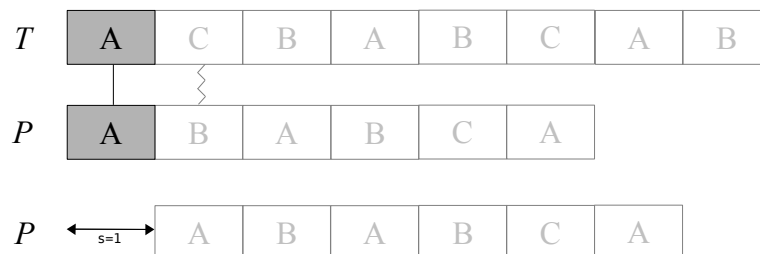


Figure 6: Knuth-Morris-Pratt Example 1

In Figure 6, the first character is a match and the second character is a mismatch. The π value of the the first character is 0 so the character of the pattern of index 0 is compared to the character which was mismatched. In this example it is a mismatch.

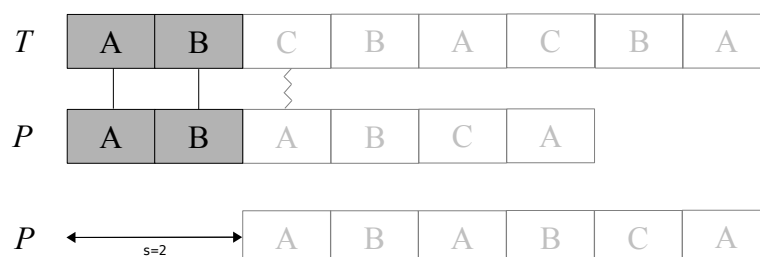


Figure 7: Knuth-Morris-Pratt Example 2

In Figure 7, the first two characters match and the third character is a mismatch. The π value of the last matched character, which is the second character, is 0 so the character of the pattern which has the index 0 is compared to the character which was mismatched. The algorithm misses out the invalid shift.

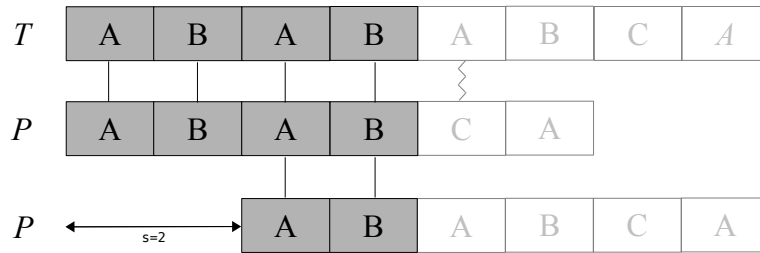


Figure 8: Knuth-Morris-Pratt Example 3

In Figure 8, the first four characters match and the fifth character is a mismatch. The π value of the last matched character, which is the fourth character is 2, so the character of the pattern which has the index 2 is compared to the character which was mismatched. In this example the overlap is 2 and the invalid shift has been missed out. The pattern is shifted to a potentially valid position. The first two characters of the shifted pattern are not checked again because they were a match in the previous shift. The information about the match is retained.

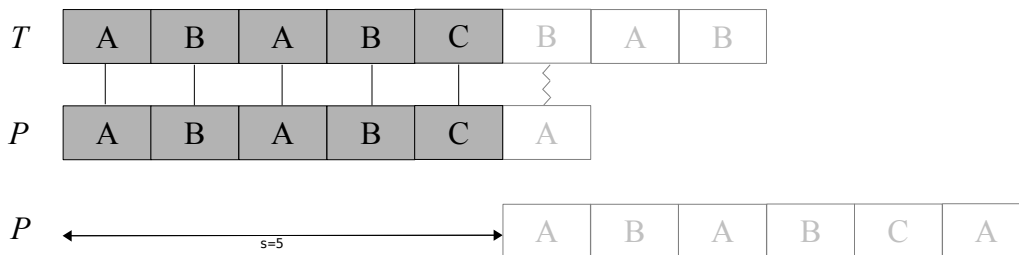


Figure 9: Knuth-Morris-Pratt Example 4

In Figure 9, the first five characters match and the sixth character is a mismatch. The π value of the last matched character, which is the sixth character is 0, so the character of the pattern which has the index 0 is compared to the character which was mismatched. There is no overlap because the character c does not appear in the rest of the pattern. This means that four invalid shifts are missed out which greatly speeds up the process of finding matches. Best running times would occur when every character in the pattern is different. Slowest running times would occur when there is a repeating motif in the pattern.

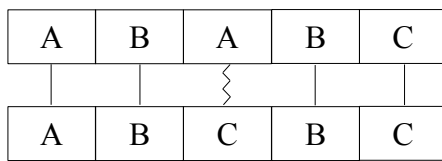
The worst case running time of the Knuth-Morris-Pratt algorithm is $\Theta(m+n)$. The prefix function has a worst case running time of $\Theta(m)$ and the matcher has a worst case running time of $\Theta(n)$. This is because the two parts of the algorithm are very similar. The prefix function matches the pattern against itself and the matcher matches the text against the pattern. Information about previous matches is retained in the algorithm, so shifts are missed out and not every character of the pattern has to be checked against the text in each new shift.

The naive, Rabin-Karp and the Knuth-Morris-Pratt algorithms were tested in order to compare which method had the fastest running time and to see how the length of the pattern affects the speed of each method. Human chromosome 21 was used as the text. The number of matches was also compared to a random data set to see if any results were particularly significant.

Approximate String Matching

Substitution

Substitution is where single letters have been replaced in a match. The matched substring is very similar to the pattern but not exactly the same.



The pattern does not match the substring in an exact search. In an approximate search where at least one mismatch is allowed, it is a match.

Figure 10: Substitution

Mutations can occur in DNA sequences, so allowing for substitutions of bases may be a more realistic approach to searching for motifs. The naive algorithm can easily be modified to include mismatches, and a limit can be set to the number of mismatches the user intends to allow. The fastest matching technique can not be used because that involves slices which are exact, but the second fastest technique is used which uses a while loop.

Even though the running time is the same as that of the naive algorithm; $O(nm)$, where n is the length of the text and m is the length of the pattern, approximate searching is slower than exact searching because it does not use slices and also because the substring only moves on to the next position when the number of mismatches exceeds the limit set by the user. Even with the limit set to one, the number of characters that are matched increases. In the naive algorithm, as soon as there is a mismatch the substring moves onto the next position. When taking into account a certain number of substitutions, the substring only moves on to the next position when there are too many mismatches. This takes longer.

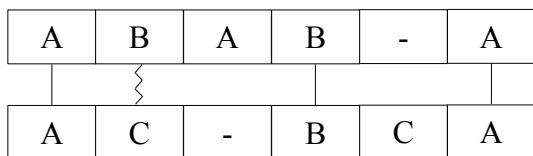
Three different implementations have been created. The first implementation simply lists all the matching positions. It gives no information about the number of mismatches or the substring which was matched. The second implementation lists all the matching positions in lists of the number of mismatches the position has, as well as the slice the position refers to. The final implementation lists the pattern which was found and how many times it was found in a sorted list. The third implementation is the most useful in investigating if a sequence is likely to have any biological significance.

The final implementation was tested on chromosome 21 and the number of matches was compared to the number of matches found in a random string which was the same length as chromosome 21. The implementation was tested by searching for the Forkhead-box which is a pattern of length 3, TATA-box which is a pattern of length 6, the MADS-box which is a pattern length 10, and a nuclear receptor which is a pattern of length 12. The running time was also compared.

Although this technique is useful, it does not allow weights to be put on each position as to how likely they are to be substituted. For example it does not take into account if one base is more likely mutate into another.

Insertions and deletions

Insertions and deletions are where characters have been added or removed. In the context of pattern searching this means gaps are left either in the pattern or in the text.



A substring of the text is 'ABABA' and the pattern is 'ACBCA'. Here there is a match which takes into account substitutions (B and C are not matched) and insertions and deletions (where there are gaps).

Figure 11: Insertions and deletions

Mutations not only involve substitutions of bases but also adding and removing bases. Considering insertions and deletions in addition to substitutions is an even more realistic approach to searching for motifs. However, the naive algorithm can not simply be modified to allow gaps; considering insertions and deletions in matching patterns to a string requires a pairwise alignment algorithm. Alignment is where strings are arranged so that the letters line up to maximise the similarity between the two strings. Pairwise alignment is the aligning of two strings. The relevant type of alignment to pattern searching is local alignment where the pattern is aligned to part of the text. However, the global alignment method was first implemented as an introduction to the method in general.

Global alignment is where the whole of one string is aligned with the whole of another string. In order to find the optimum alignment, a scoring system is used. A score is set for a gap, a match and a substitution. For example, the gap score could be -3 the match score could be 5 and the substitution score could be -1. These scores could be changed to represent a more realistic representation of the likeliness of certain mutations. Independence of the different bases is assumed and the alignment score is the sum off these scores.

String 1	String 2	Score
A	A	5
-	C	-3
B	B	5
A	C	-1
B	-	-3
A	A	5
Total Score		8

Figure 12: Alignment scoring

A simple method would be to score every possible alignment and the highest scoring would be the optimum score. However, allowing for gaps means that the number of possible alignments is exponential to the lengths of the strings so this method is impractical. A more efficient method would be to use dynamic programming which would solve smaller subproblems which could be combined and used to solve the whole problem.

Dynamic programming is where the optimal alignment is built up from the optimal alignment of the the prefixes of the two sequences. This means that only two characters are being compared at one time. For example, there are only 3 possible ways the optimum alignment can begin. The two characters can be aligned with each other which would either be a match or a mismatch, there could be a gap in one of the strings and the first character of the other string, or vice versa. The optimal alignment would be the highest scoring from these three cases. This recursive property can be used for all characters in the two sequences so the optimum solution can be found by looking at the three possible ways the optimum solution can come from. The scores of each subsolution are stored in a dynamic programming matrix.

		A	B	A	B	A	
		0	-3	-6	-9	-12	-15
A		-3					
C		-6					
B		-9					
C		-12					
A		-15					

Figure 13: Dynamic programming matrix

The first column and first row of scores can be calculated first because they are simply the addition of the gap scores. The gap score in this example is -1 so an addition of a gap in the alignment results in the gap score being added to the preceding alignment score. The score is 0 when none of the two strings are being compared. The next step is to calculate the scores for the rest of the matrix where the optimum alignment can come from three ways.

		A
		0
A		-3
	A	-3

Figure 14: Score calculation

In Figure 14 the first character of each string is being compared. The first route is coming from above, where there is a gap in the first string but a character in the second string. The gap score is -3 so this is added to the preceding score of -3 and results in -6. The second route is the diagonal route where the two characters are aligned with each other. In this case, the two characters match so the match score which is 5 is added to the preceding score of 0 and results in 5. The third route is from the left where there is a character in the first string but a gap in the second string. The gap score is again added to the preceding score and the resulting score is -6. These three scores are compared and the highest is taken as the optimum score. This score is added to the matrix and the direction it comes from is also stored. The rest of the matrix is filled out in this way.

		A	B	A	B	A
	0	-3	-6	-9	-12	-15
A	-3	5	2	-1	-4	-7
C	-6	2	4	1	-2	-5
B	-9	-1	7	4	6	3
C	-12	-4	4	6	3	5
A	-15	-7	1	9	6	8

Figure 15: Completed matrix

Once the matrix is completed, the alignment can then be created by tracing back from the bottom right-hand corner. This position represents the optimum global alignment for the two sequences. This method of dynamic programming is much more efficient than scoring all possible alignments because the running time is $\Theta(mn)$. The amount of memory stored is also $\Theta(mn)$ which is fairly efficient for global alignment when both the strings are fairly short. However, this can be a problem when this technique is used for local alignment where the text can be very long.

Local alignment is where only part of one or more of the two sequences is aligned with the other. In the context of searching for patterns in a text, the whole pattern would be aligned with part of the text. Local alignment differs from global alignment because the two strings are not roughly the same length, and the alignment of the pattern should be able to start and end anywhere. Multiple matches must also be obtained instead of calculating the best alignment.

Recursion was first used to implement global alignment. However, this is not an efficient method for implementing local alignment so the algorithm was modified so that the dynamic programming matrix could be calculated iteratively. Storing the whole matrix was inefficient because when chromosome 21 was used as the text, it very long and the computer would run out of internal memory. However, calculating the matrix iteratively meant that not all of the matrix needed to be stored. In fact, only two columns at one time needed to be stored which would speed up the running time. After each column was calculated, the alignment was added to a list if the score was above a certain threshold. This allowed multiple matches to be found instead of just one. The matrix was also slightly modified so that the first row consisted of a list of 0s and the alignment could end anywhere on the bottom row. This was the equivalent of being able to start and end an alignment at any point in the text.

An additional improvement was to store the starting and ending point of the match instead of storing the whole alignment. These two methods were tested to see if the adjustment improved the speed. Instead of a constant match and substitution score, a more accurate method was also used which was to use a substitution matrix where the score of substitution depends on two bases. This is more accurate because some bases are more likely to mutate into others due to their chemical structure. See Appendix for an example substitution matrix.

Position Weight Matrices

A position weight matrix is a representation of a biological motif which shows the frequency of bases at each position of the pattern. This can be used to find the probability of a particular base appearing at each position and is a more accurate method of finding matches for a particular motif.

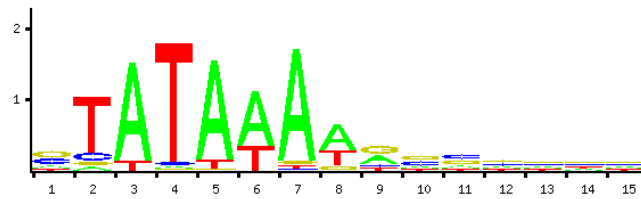


Figure 16: TATA-Box

Figure 16 shows the position weight matrix of the TATA-Box. It shows that the bases which have the highest probability of being present at the binding site are 'TATAAA', which is the sequence which was previously used to search for. However, the actual motif is much longer and there are different probabilities that different bases are present. For example in position 3 there is a higher probability that the base will be an 'A' but sometimes there may be a 'T'.

A	[61	16	352	3	354	268	360	222	155	56	83	82	82	68	77]
C	[145	46	0	10	0	0	3	2	44	135	147	127	118	107	101]
G	[152	18	2	2	5	0	10	44	157	150	128	128	128	139	140]
T	[31	309	35	374	30	121	6	121	33	48	31	52	61	75	71]

Figure 17: Frequency matrix TATA-Box

The frequency matrix is the raw figure of how many binding sites which were tested had the particular base. Each position can be converted into a probability by dividing the value by the sum of all the values for a particular position. For example in Fig. 17, the probability of the character being and 'A' in the first position would be $61/(61 + 145 + 152 + 31) = 0.16$. Once each position is converted, the probability of a match could be calculated by the following formula.

$$\prod_{i=1}^n P_i = \exp\left(\sum_{i=1}^n \log_e P_i\right)$$

The probability of a match is the product of each corresponding probability for the base present at each position. Taking a sum is a simpler and more efficient method so the natural logarithm is used instead so that the logarithm of the probabilities can be added. The formula does not have to be exponentiated either because the function is monotonic and the logarithm probabilities are easier numbers to deal with. To avoid taking the logarithm of 0, a small number is added to all the probabilities. This is negligible and does not affect the results.

Position weight matrices of different motifs are not observed on the same size data set. To be able to compare the match probabilities of one motif to another, the match probabilities are also normalised so that it is between 0 and 1 using the following formula.

$$0 \leq \frac{S - L}{H - L} \leq 1$$

S is the sum of the logarithmic probabilities, L is the lowest possible probability and H is the highest possible probability. Normalising the sum also allows a limit to be set on probabilities of the matches which are printed.

Another technique which was applied to position weight matrices was to search for the inversion of the pattern. The chromosome string is just one side of the DNA so motifs may be able to be found on the complimentary string. In order to calculate the match probability of the inverted string, the probability of the complimentary bases was used. The position weight matrix algorithm was tested and speed and number of matches compared.

Results

Naive String Algorithm matching techniques

Match 1 = for loop

Match 2 = slices

Match 3 = while loop

Pattern		Forkhead-Box	TATA-Box	MADS-Box	Nuclear Receptor	
String		GTA	TATAAA	CTATTTATAG	GGTCAAAGGTCA	
Time (s)	Match 1	1	98	111	120	118
		2	103	110	114	113
		3	99	105	127	112
		Mean	100	109	120	114
	Match 2	1	48	48	54	54
		2	49	60	54	56
		3	49	55	55	54
		Mean	49	54	54	55
	Match 3	1	65	79	73	72
		2	66	80	70	69
		3	66	79	70	82
		Mean	66	79	71	74
Number of matches		380322	26122	64	2	

The fastest matching technique in all lengths of pattern was Match 2 which used slices. Even though the while loop and for loop are doing the exact same thing, taking slices is a more efficient technique when using python because the process of either a for loop or a while loop happens automatically within the python programme which is faster than inputting every step.

Across all matching techniques, a general trend can be found that the longer the length of the pattern, the longer time it takes to find all the matches. This is as predicted because the running time is $\mathcal{O}(nm)$ so a longer pattern would increase the running time. There is an anomaly to this pattern in Match 1 where the longest sequence has a faster time than the second longest sequence. This is probably because there are not many similar substrings in the chromosome to the sequence of the nuclear receptor. There is also an anomaly in Match 3 for the TATA-box sequence. This is probably because there are many substrings in the chromosome which are an overall mismatch but have a similar prefix to the pattern. This would make the running time slower because more of the pattern would have to be checked against the substring in order to get a mismatch. The different anomalies in Match 1 and Match 3 would be due to the slightly different methods of the for loop and the while loop.

The range of the data can give a rough representation of the rate at which the time increases as the length of the pattern increases. Match 2 not only has the fastest time out of all three matching techniques, but also has the smallest range of 6 indicating that the rate that the time increases in relation to the length of the pattern is smaller than the other two matching techniques. Match 2 is clearly the best technique to use.

Finally there is a very clear pattern that the shorter the pattern, the more matches which can be found.

Rabin-Karp Algorithm

Method 1 - Rabin-Karp Algorithm using an ordinal number method to convert characters into numbers. Radix is calculated by subtracting the minimum ordinal number from the maximum ordinal number.

Method 2 – Rabin-Karp Algorithm using a dictionary method to convert characters into numbers. Radix is calculated by the length of the dictionary.

Method 3 – Slightly more efficient modified method using the ordinal number of a character to convert characters into numbers.

Pattern		Forkhead-Box	TATA-Box	MADS-Box	Nuclear Receptor	
String		GTA	TATAAA	CTATTTATAG	GGTCAAAGGTCA	
Time (s)	Method 1	1	38	36	37	36
		2	37	36	37	37
		3	37	37	37	36
		Mean	37	36	37	36
	Method 2	1	33	32	32	32
		2	32	31	32	32
		3	32	31	33	31
		Mean	32	31	32	32
	Method 3	1	30	30	31	30
		2	31	30	31	30
		3	30	30	30	30
		Mean	30	30	31	30
Number of matches		380322	26122	64	2	

Testing the algorithm revealed problems. Using the power function meant that the integers were converted into a floating point. This meant that exact matches would not necessarily be found for every pattern. For example in the original algorithm using the ordinal numbers, no matches were found for the nuclear receptor when there are two. This led to the two original methods to be modified so that the power function was not included, and a the third modified algorithm was created in an attempt to speed up the algorithm further. This third method uses the ordinal number of the character to convert the characters into numbers. This was faster than using the dictionary because if a dictionary is used, the dictionary has to be made and characters have to be looked up. The number of steps taken is greater than just converting a character into an ordinal number.

In comparison to the naive algorithm, the speed stays relatively the same and does not increase as the length of the pattern increases. This is probably because the running time of the Rabin-Karp algorithm is linear whereas the running time of the naive algorithm is quadratic, so running time will increase much more in the naive algorithm.

In the results Method 3 has a very similar running time to Method 2. This is probably because the algorithm was run on a server which has eight 3.00GHz Intel Xeon cores and 31GiB of memory, and method 3 is only faster if the computer runs out of memory. In this method, the characters are converted into numbers as they are used instead of converting the whole string into numbers at once.

Naive, Rabin-Karp and Knuth-Morris-Pratt Comparison

The fastest matching technique data was used as a comparison

Pattern		Forkhead-Box	TATA-Box	MADS-Box	Nuclear Receptor	
String		GTA	TATAAA	CTATTTATAG	GGTCAAAGGTCA	
Time (s)	Naive	1	21	21	21	21
		2	21	21	21	21
		3	21	21	21	21
		Mean	21	21	21	21
	Rabin-Karp	1	30	30	31	30
		2	31	30	31	30
		3	30	30	30	31
		Mean	30	30	31	30
	KMP	1	20	21	20	20
		2	20	21	20	19
		3	20	21	20	20
		Mean	20	21	20	20
Number of matches		380322	26122	64	2	

The naive algorithm is theoretically meant to be the slowest algorithm but for the patterns we were using to search in chromosome 21, it is faster than the Rabin-Karp algorithm and not much slower than the Knuth-Morris-Pratt algorithm. Further testing of longer patterns which were a longer string revealed that the naive algorithm running time increased as the length of the pattern increased but the Rabin-Karp algorithm and the Knuth-Morris-Pratt algorithm stayed at a similar time. The naive algorithm became slower than the Rabin-Karp when the pattern length exceeded 1500 characters in length (for this implementation). This shows that the Knuth-Morris-Pratt algorithm is the fastest for the type of patterns we are searching for and that the Rabin-Karp algorithm would be more useful if the patterns we were searching for were larger. The naive algorithm also slows down more when there are many similar strings to the pattern in the text.

Pattern	GTA	TATAAA	CTATTTATAG	GGTCAAAGGTCA
Number of matches in Chromosome 21	380322	26122	64	2
Expected number of matches	733505	11461	45	3
Number of matches in random string of DNA	732789	11459	40	3

The TATA-Box and the MADS-Box seem to have a slightly higher number of matches in comparison to the expected number and the number found in a random string. This means these two sequences could have a biological significance.

Substitution

Naive algorithm which took into account substitution was tested on all of the sequences with a maximum of 2 mismatches.

Pattern		Forkhead-Box	TATA-Box	MADS-Box	Nuclear Receptor
String		GTA	TATAAA	CTATTTATAG	GGTCAAAGGTCA
Number of matches in chromosome 21	Exact	380322	26122	64	2
	1 mismatch	4913681	376068	1842	82
	2 mismatches	16224140	2040211	26885	1284
Number of matches in random string	Exact	732789	11459	40	3
	1 mismatch	19810428	206495	1379	86
	2 mismatches	27142161	1546169	18153	1628
Naive time (s)		21	21	21	21
Substitution time (s)		96	94	93	89

Comparing the results from Chromosome 21 with the results from a random string of the same length can reveal whether the motifs are more likely to appear in the chromosome than in a random string. The results show that the TATA-Box and the MADS-Box are found many more times in the chromosome than in the random string with up to two mismatches. This indicates that the two sequences are probably biologically significant. The TATA-Box is a core binding site in most genes so a larger presence in the chromosome is expected in comparison to the random string. The MADS-Box is also a known motif so a slightly higher number of matches found in the chromosome is also expected.

However, the nuclear receptor binding site seems to have a similar number of matches to the random string which suggests the sequence is not more likely to be present on a chromosome than on another random string. This suggests the binding site may not be very common or the sequence is not present in this chromosome as a binding site.

The Forkhead-Box appears in the chromosome almost half the number of times that it appears in the random string. This Forkhead-Box is actually a binding site of the FOXC1 transcription factor which is encoded by the FOXC1 gene. This gene has an unknown specific function and is situated on chromosome 6. The frequency of this binding site is unknown and there may be none situated on chromosome 21. Another explanation could be that the triplet GTA has significantly less matches in the chromosome than in the random string because it is a preferred target for hypermutation. This means that this triplet mutates faster than average. This would explain why the triplet is present almost half the amount that it is present in a random string.

The speed of the naive algorithm with substitution is slower than the naive algorithm which is as expected because a sequence only becomes a mismatch when there are more than 2 mismatches instead of when there is only 1 mismatch. This means that there are many more potential matches for the naive algorithm with substitution than the naive algorithm without substitution.

Insertions and Deletions

Testing the implementation posed problems because it used a dictionary to convert the base letters into index numbers so that the substitution score could be read off the matrix. This was a problem because the dictionary did not contain the character 'N' which was used in the DNA sequence. To get around this problem all the 'N' characters were removed.

Method 1 – Local alignment algorithm which stores the alignment as you go along and returns a list of matches which have a score above a certain threshold

Method 2 – Local alignment algorithm which stores the starting and ending point of the alignment and returns a slice of the text.

String		GTA	TATAAA	CTATTTATAG	GGTCAAAGGTCA
Time (s)	Method 1	614	456	700	838
	Method 2	518	386	583	704
Number of matches in chromosome 21		12,069,889	1,094,015	8901	659
Number of matches in random string		17,919,899	721,724	7190	595

These results show that the second method was quicker as predicted. The running time increases from the sequence of length 6 and upwards. The sequence of length 3 is probably slower than the sequence of length 6 because of the huge number of matches the program found. There were more matches found in the chromosome than in the random string with all patterns except for the sequence of length 3. It appears that only the 'TATAAA' sequence could have a significantly larger number of matches in the chromosome than in the random string to suggest biological significance. The other sequences seem to have similar number of matches to the random string.

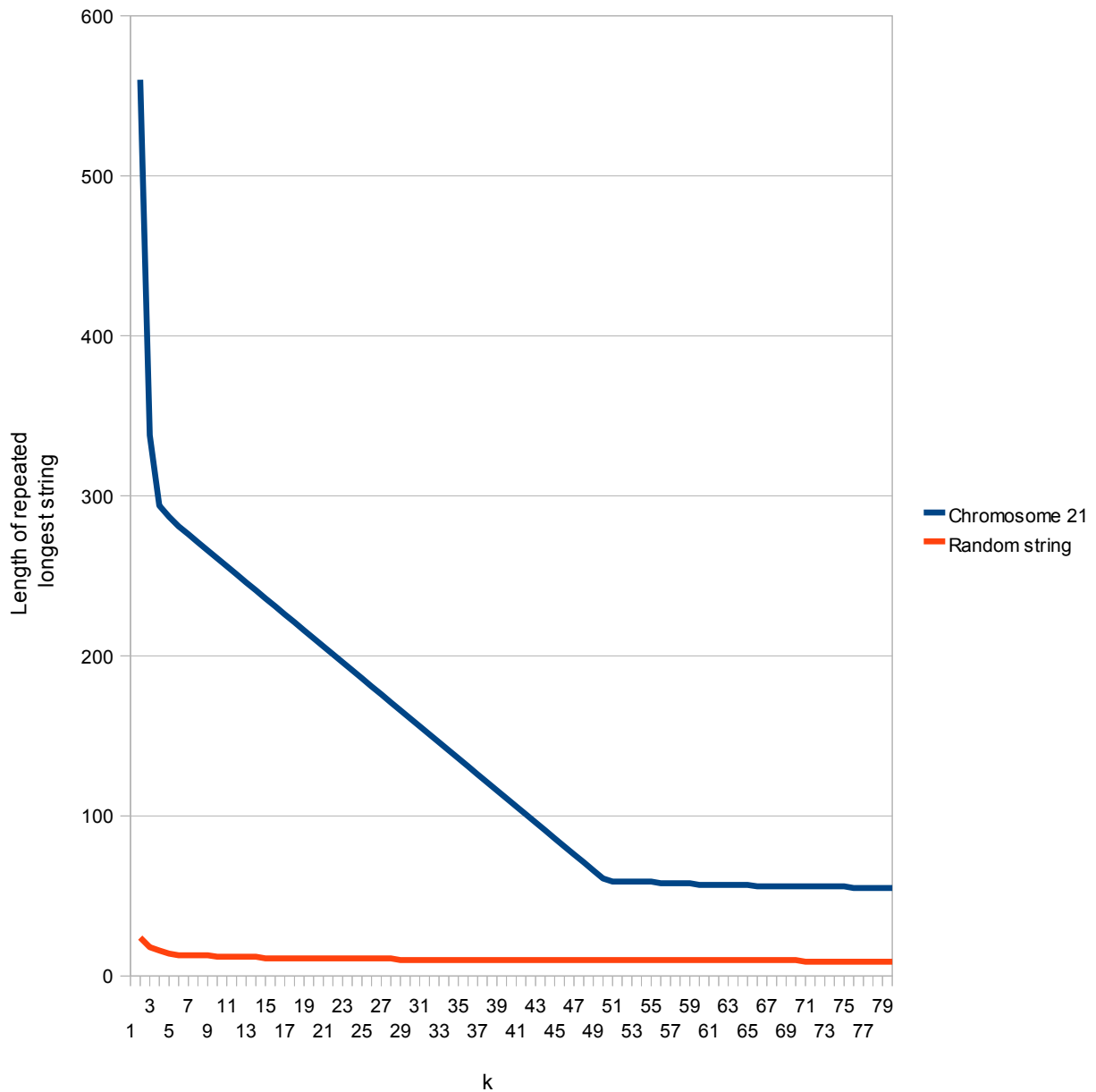
Position Weight Matrices

This was not tested on the server, but a clear difference can be seen between both versions, with and without taking into account inversions. The patterns were counted if they had a score of over 0.98. The Nuclear Receptor was not used to test this algorithm because it did not have a position weight matrix.

String		GTA	TATAAA	CTATTTATAG
Time (s)	Without inversions	360	559	419
	With inversions	720	1162	861
Number of matches	Without inversions	37708	58	1352
	With inversions	75331	108	2705

From these results you can see that the algorithm without the inversions is about half the speed of the algorithm with inversions and finds about half the number of matches. The time it takes to complete is twice as long for the algorithm with inversions because the program checks the whole of the complimentary string so effectively the length of the string is twice as long. Results have not been compared to a random string of the same length so it is difficult to tell if this is the result which would have occurred with a random string or whether the number of matches found is above average.

Graph to show the length of the longest string which appears k times



The graph has a general trend for both the chromosome and the random string that as k increases, the length of the repeated longest string decreases, which is as expected. However, the graph shows a stark contrast between the length of the repeated longest string in the chromosome and in the random string. The length of the longest string which occurs twice in the chromosome is 560 and the length in the random string is 24. As k increases in chromosome 21, the length of the repeated pattern decreases dramatically until it smooths into a straight line where the longest length decreases by 5 nucleotides each time. There is a significant contrast between the results from the chromosome and the results from the random string. This suggests that the sequence in chromosome 21 is far from random and has a very long repeating sequence made up of 5 nucleotide bases. The constant drop of 5 stops when k is 51 and the graph appears similar to the random string but at a higher number and decreasing slightly more rapidly. See Appendix for the full results table.

Conclusion

Using slices in the naïve algorithm was found to be the fastest matching technique because this is a more efficient method than using a for loop or a while loop when using python. Using ordinal numbers to convert a string into a list of numbers in the Rabin-Karp algorithm was also found to be the fastest technique. In theory, the Rabin-Karp and the Knuth-Morris-Pratt algorithm were meant to be much more efficient than the naïve algorithm. However, testing the programs on chromosome 21 revealed different results. The naïve algorithm was clearly faster than the Rabin-Karp algorithm and was not much slower than the Knuth-Morris-Pratt algorithm. This experiment showed that Rabin-Karp is not more efficient than the naïve in this circumstance. However, further testing of much longer patterns showed that the running time of the naïve algorithm increased proportionally to the length of the pattern whereas the running time did not increase much for both the Rabin-Karp and the Knuth-Morris-Pratt. This is supported by the fact that the naïve algorithm has a running time of $\Theta(nm)$ whereas the Rabin-Karp and Knuth-Morris-Pratt algorithm had a running time of $\Theta(n+m)$. This means that for patterns longer than about 1500 characters, the Rabin-Karp would become more efficient than the naïve but may not be as efficient for the type of pattern searching the experiment carried out. Exact matching is useful to biologists, so fast algorithms are vital for saving time.

The TATA-Box and the MADS-Box had a significantly higher number of matches in the chromosome than in the random string with exact matches which suggests these two sequences have a biological significance and appear often in the chromosome. Searching for patterns with one or two substitutions also resulted in a much higher number of matches in the chromosome than in the random string which supports the significance. The approximate searching with substitutions was a simple amendment to the naïve algorithm so the fact that the running time increases as the number of substitutions allowed increases is expected.

Using local alignment which included insertions and deletions of letters as well as substitutions is an attempt to take into account mutations of the DNA and therefore be a more realistic model. However, it is known that the TATA-Box and the MADS-Box appear many more times in the chromosome than in the random string but the results did not show that. The results for the MADS-Box in the chromosome was a similar number to the number in random string but this could be because the choice of the scores which were set for the substitutions and gaps.

Searching using position weight matrices is an accurate method for finding matches for a particular motif. Results were not compared with the random string so it is difficult to conclude if the technique would have increased the number of matches for a particular motif in the chromosome compared to the naïve method. Searching for inversions of the pattern which corresponds to searching for the pattern in the opposite direction in the complimentary string also adds another dimension to the number of matches found and would need to be investigated further.

Creating a suffix tree for the chromosome aimed to answer the question "What is the length of the longest string which appears k times?". The results of the chromosome are clearly very different from the results in the random string which is evidence that the sequence of bases on the chromosome is far from random. It is an unexpected find that from when $k > 6$, the length of the longest string decreases by a constant number each time producing a part of the graph to be a straight line. The length decreases by 5 nucleotide bases each time which suggests there is a very long repeating pattern in the chromosome where each repeating unit has a length of five.

Evaluation

Overall, the experiments went well and the running time was short when the server was used so that programs could be tested thoroughly in the time that was had. The naïve program worked well when only part of the chromosome was tested but when the whole chromosome was tested, there were some memory problems and the program would run very slowly. This was solved by using an external server which has eight 3.00GHz Intel Xeon cores and 31GiB of memory. The naïve program then ran very fast and even ran faster than the Rabin-Karp program. The reason for this could be looked into further, for example if the cause was the length of motif that was being searched for or if python made the naïve method very efficient by having an efficient slicing method. The Rabin-Karp algorithm posed a few problems. The first problem which was solved was how to convert the a list of characters into a list of numbers. The most efficient way to solve this was to use the ordinal numbers which is a quicker procedure than using a dictionary to assign numbers. Another problem was choosing the modulus to use. Different numbers were tested to see if they affected the running time. A large prime number was decided on so that there would be a more even spread of values. There was also problems with using the power function because it converted the numbers into floating point numbers when integer numbers were needed. This was discovered after several tests and the program was changed and a power function was defined within the program instead of using an imported math function. The Knuth-Morris-Pratt was the hardest program to fully understand but did not not pose too many problems. To extend the project, the Boyer-Moore algorithm, another exact string matching algorithm could also be implemented and compared to the others.

The substitution program did not pose many problems because it was a simple modification of the naïve program. The creation of the matrices in the programs using alignment was originally solved by using recursion. However, this was an unconventional way of implementing the technique so several iterative versions were produced. The difficulty with using alignment was that a threshold had to be set to decide if the position found was a match to the pattern or not, and a gap score and a substitution matrix had to be created. A slight change in the scoring would change the sequences in the chromosome which were counted as matches. An improvement would be to investigate the most realistic scores to use for human DNA.

Another problem encountered was that the scoring matrices took up too much memory and the running time would increase. To get around this problem the alignment program was implemented iteratively and only two columns of scores were stored at one time. This greatly speeded up the running time of the program. Overall, using alignment was a much more realistic way of searching for motifs because the substitution program required the user to set the number of substitutions which were allowed, whereas the using local alignment meant that the number of substitutions automatically changed and differed according to the scoring of the rest of the sequence. To extend the project, the gap scoring could have been developed to be affine gap scores instead of linear gap scores. This means that the gap penalty would not increase by much after one gap was already there. This is because the probability of having one gap is closer to the probability of having two gaps than it is to having no gaps. The alignment program also did not work unless the N characters were removed because they were not included in the dictionary which was used.

Implementing a search which used position weight matrices did not pose any problems. However, implementing a program which would create a compact suffix tree would have taken more time than was available so this was solved by downloading an implementation of the compact suffix tree in C. Using python for this would have been slower anyway and using this downloaded implementation saved a lot of time and results were just as good. Thanks to Dr Adam Novak for modifying the downloaded program.

Future Work

This project could have been extended in many ways. The number of tests could be increased to obtain more reliable and significant results and a search of motifs could be conducted over the whole genome. These results could also be compared to commonly used motif scanners. The methods used in the project could also be used to search for regions which contain several motifs in order to identify promoter regions. The Ukkonen compact suffix tree algorithm could also be implemented. Searching for inversions of the pattern could also be carried out using all techniques instead of just when position weight matrices are used.

Appendix

DNA Substitution matrix

	A	C	G	T
A	3	-5	-2	-5
C	-5	3	-5	-2
G	-2	-5	3	-5
T	-5	-2	-5	3

Results table to show the length of the longest string which is repeated k times

k	Chromosome 21	Random string
2	560	24
3	338	18
4	294	16
5	287	14
6	281	13
7	276	13
8	271	13
9	266	13
10	261	12
11	256	12
12	251	12
13	246	12
14	241	12
15	236	11
16	231	11
17	226	11
18	221	11
19	216	11
20	211	11
21	206	11
22	201	11
23	196	11
24	191	11
25	186	11
26	181	11
27	176	11
28	171	11
29	166	10
30	161	10
31	156	10
32	151	10
33	146	10
34	141	10

k	Chromosome 21	Random string
35	136	10
36	131	10
37	126	10
38	121	10
39	116	10
40	111	10
41	106	10
42	101	10
43	96	10
44	91	10
45	86	10
46	81	10
47	76	10
48	71	10
49	66	10
50	61	10
51	59	10
52	59	10
53	59	10
54	59	10
55	59	10
56	58	10
57	58	10
58	58	10
59	58	10
60	57	10
61	57	10
62	57	10
63	57	10
64	57	10
65	57	10
66	56	10
67	56	10
68	56	10
69	56	10
70	56	10
71	56	9
72	56	9
73	56	9
74	56	9
75	56	9
76	55	9
77	55	9
78	55	9
79	55	9
80	55	9

Naïve Algorithm

```
#!/bin/env python
```

```
import sys
```

```
#Every element in the pattern is matched to the length of the pattern on  
#each element of the string. If it is matched, function returns TRUE.  
#x = position in string.
```

```
def match1(string,pattern,x):  
    for y in range(len(pattern)):  
        #If pattern matches the length of string function returns TRUE,  
        #otherwise it is a mismatch and returns FALSE.  
        if pattern[y] == string[x+y]:  
            pass  
        else:  
            return False  
    return True
```

```
#Pattern is matched to a slice of the string which is the length of the pattern  
#and with shift of x. If it is matched, function returns TRUE.  
#x = position in string.
```

```
def match2(string, pattern, x):  
    return pattern == string[x:x+len(pattern)]
```

```
#Each element of the pattern is matched to the corresponding element in the  
#string with shift of x.  
#x = position in string.
```

```
def match3(string,pattern,x):  
    y = 0  
    #When the position of the element is less than the length of pattern the  
    #element is matched. If every element is matched the function returns TRUE.  
    while y < len(pattern):  
        #If element in pattern matches corresponding element in string next  
        #element is matched. If elements do not match it is a mismatch and  
        #function returns FALSE.  
        if pattern[y] == string[x+y]:  
            pass  
        else:  
            return False  
        y = y + 1  
    return True
```

```
#Naive string match. Pattern used as a "sliding template" over the string.  
#match = match function used to compare section of string and pattern.
```

```
def naive(string, pattern, match=match1):  
    n = len(string)  
    m = len(pattern)  
    count = 0  
    #Every element in the string is matched to every element in the pattern  
    #using match function.  
    for x in range(n-m+1):  
        #If match function returns TRUE, position on string will be printed.  
        if match(string,pattern,x):  
            count = count + 1  
            #print x  
    print "Count is", count
```

```
f=open(sys.argv[1],'r')  
str=f.readline()  
f.close()
```

```
naive(str,sys.argv[2])
```

Rabin-Karp Algorithm

```
#!/bin/env python
```

```
import sys
from math import pow

#Pattern is matched to a slice of the string which is the length of the pattern
#and with shift of x. If it is matched, function returns TRUE.
#x = position in string.
def match2(string, pattern, x):
    return pattern == string[x:x+len(pattern)]

#Rabin-Karp string match. Pattern is assigned a number and compared to assigned
#number of the string which is the length of the pattern. Each assigned number
#on the string is calculated using a formula so that function does not have to
#compare each element with the pattern more than once.
def Rabin_Karp(string, pattern, modulus=16647133, match=match2):
    #Radix set as the number of graphic characters
    radix = 256
    n = len(string)
    m = len(pattern)
    #Pattern is converted into the ordinal number.
    p=0
    for i in pattern:
        p = (radix * p + ord(i)) % modulus
    #First length of pattern in string is converted into the ordinal number.
    s=0
    for i in string[:m]:
        s = (radix * s + ord(i)) % modulus
    count = 0
    #Pattern and first string number compared. If it matches, 0 is printed.
    if s == p and match(string, pattern, 0):
        print 0
        count = 1
    #Instead of using a power function a loop is used. This allows the
    #modulus to be taken with each iteration.
    number = 1
    for i in range(m-1):
        number = (number * radix) % modulus
    #Each position has a number assigned to it calculated up to the length of
    #the pattern. Only next element is used by using a formula so function does
    #not run through each element of the string more than once.
    for i in range(1, n - m + 1):
        s = (radix * (s - number * ord(string[i-1])) + ord(string[i+m-1])) % modulus
        #If the number assigned to the pattern and the number assigned to the
        # string which is the length of the pattern in position i are the same,
        # the position is printed.
        if s == p and match(string, pattern, i):
            print i
            count = count + 1
    print "count is", count

f=open(sys.argv[1], 'r')
str=f.readline()
f.close()

Rabin_Karp(str, sys.argv[2])
```

Knuth-Morris-Pratt

```
#!/usr/bin/env python
```

```
import sys
```

```
def KMP(string, pattern):
```

```
    #Prefix function computes how the pattern matches against shifts of itself.
```

```
    def prefixfunction(pattern):
```

```
        m = len(pattern)
```

```
        pi = []
```

```
        #Number of positions created in the list pi is the length of the pattern.
```

```
        for i in range(m):
```

```
            pi.append("")
```

```
        #First pi value is always 0 because there is only one character so
```

```
        #the length of the prefix and suffix is 0.
```

```
        pi[0] = 0
```

```
        #k is the overlap.
```

```
        k = 0
```

```
        for q in range(1,m):
```

```
            #While variable k is larger than 0 and the character in the pattern of
```

```
            #index k does not equal the character in the pattern of index q.
```

```
            while k > 0 and pattern[k] != pattern[q]:
```

```
                k = pi[k-1]
```

```
            if pattern[k] == pattern[q]:
```

```
                k = k + 1
```

```
            pi[q] = k
```

```
        return pi
```

```
def KMPmatcher(string, pattern):
```

```
    n = len(string)
```

```
    m = len(pattern)
```

```
    pi = prefixfunction(pattern)
```

```
    q = 0
```

```
    count = 0
```

```
    for i in range(n):
```

```
        while q > 0 and pattern[q] != string[i]:
```

```
            q = pi[q-1]
```

```
        if pattern[q] == string[i]:
```

```
            q = q + 1
```

```
        if q == m:
```

```
            count = count + 1
```

```
            #print i-(m-1)
```

```
            q = pi[q-1]
```

```
    print "Count is", count
```

```
    KMPmatcher(string, pattern)
```

```
f=open(sys.argv[1], 'r')
```

```
str=f.readline()
```

```
f.close()
```

```
KMP(str, sys.argv[2])
```

Substitution

```
#!/bin/env python

#Prints a sorted list of the matches and the number of each match in lists
#Prints count of how many matches in each list
#Prints total count

import sys

#The number of lists created is the number of mismatches which has been set
#as the limit.
list = []
for i in range(int(sys.argv[3]) + 1):
    list.append({})

# Goes through each position in the pattern and substring.
# If there are no mismatches the function returns TRUE. If there are
# less mismatches than the limit set, the function returns TRUE.
def match(string,pattern,limit,x):
    y = 0
    mismatch = 0
    while y < len(pattern):
        #If pattern and substring do not equal, one is added to mismatch
        #tally. If there are more mismatches than the limit, the function
        #returns false
        if pattern[y] != string[x+y]:
            mismatch += 1
            if mismatch > limit:
                return False
        y = y + 1
    substring = string[x:x+len(pattern)]
    #If substring is in the list one is added to the tally, if it is not
    #in the list the substring list is created.
    if substring in list[mismatch]:
        list[mismatch][substring] += 1
    else:
        list[mismatch][substring] = 1
    return True

# Naive string match with substitution.
# limit = maximum number of substitutions allowed for it to be a match
def naive(string, pattern, limit):
    n = len(string)
    m = len(pattern)
    #Match function carried out for each position.
    for x in range(n-m+1):
        match(string,pattern,limit,x)
    #For each list the number of mismatches the list refers to and the
    #count of each list is printed followed by the slice which was matched
    #and the number of occurrences of that slice.
    for i in range(len(list)):
        print "Number of mismatches =", i, "Count =", sum(list[i].values())
        for j in sorted(list[i].keys(), lambda x,y: list[i][y]-list[i][x]):
            print j, list[i][j]
    #Total number of matches is printed.
    print "Total count is", sum(sum(x.values()) for x in list)

f=open(sys.argv[1],'r')
str=f.readline()
f.close()

naive(str,sys.argv[2], int(sys.argv[3]))
```

Insertions and Deletions

```
#!/bin/env python
```

```
#Only position of alignment saved. Slice taken from string which is  
#starting and ending point in the chromosome.
```

```
import sys
```

```
def alignment(text, pattern, matrix = [[3,-5,-2,-5], [-5,3,-5,-2], [-2,-5,3,-5],  
[-5,-2,-5,3]], gap = -3):
```

```
    dic = {'A':0, 'C':1, 'G':2, 'T':3}  
    matches = []
```

```
    def matrixscore(T,P):  
        return matrix[dic[P]][dic[T]]
```

```
    def score(text, pattern):  
        m = len(pattern)  
        n = len(text)  
        newall = [""]*(m+1)  
        oldall = [""]*(m+1)  
        count = 0  
        for j in range(n+1):  
            newall[0] = (j-1,0)  
            for i in range(1, m+1):  
                if j == 0:  
                    newall[i] = (0 ,i*gap)  
                else:  
                    optimumall = oldall[i][1] + gap  
                    source = 0  
                    value = newall[i-1][1] + gap  
                    if optimumall < value:  
                        optimumall = value  
                        source = 2  
                    value = oldall[i-1][1] + matrixscore(text[j-1],pattern[i-1])  
                    if optimumall < value:  
                        optimumall = value  
                        source = 1  
                    if source == 0:  
                        newall[i] = (oldall[i][0], optimumall)  
                    elif source == 1:  
                        newall[i] = (oldall[i-1][0], optimumall)  
                    elif source == 2:  
                        newall[i] = (newall[i-1][0], optimumall)  
                    if i == m and optimumall >= m*matrix[0][0]+gap+matrix[0][1]:  
                        matches.append((newall[i][0], j, newall[i][1]))  
                        count = count + 1  
            (newall, oldall) = (oldall,newall)  
    print "Count is", count  
    return matches
```

```
    s = score(text, pattern)
```

```
    return s
```

```
f = open(sys.argv[1], "r")  
text = f.readline()
```

```
f.close()  
text = text.replace("N", "")  
result = alignment(text, sys.argv[2])
```

```
print result[1]  
print result[0]  
print result[2]
```

Position Weight Matrices (with inversions)

```
#!/bin/env python
```

```
import sys
from math import *
import re

def pwm(string, pattern = [[13,13,18,9],[0,5,48,0],[52,0,1,0],[0,0,0,53],[25,7,15,6]]):
    dic = {'A':0, 'C':1, 'G':2, 'T':3}
    convertstring = []
    for e in string:
        convertstring.append(dic[e])
    n = len(string)
    m = len(pattern)
    smallnumber = 0.000001
    count = 0
    logpattern = []

    def match(convertstring, pattern, y):
        prob = 0
        for a in range(m):
            prob += (logpattern[a][convertstring[y+a]])
        return (prob-lo)/(hi-lo)

    def matchinversion(inversionstring, pattern, z):
        prob = 0
        for b in range(m):
            prob += (logpattern[m-1-b][3-convertstring[z+b]])
        return (prob-lo)/(hi-lo)

    for x in range(m):
        logpattern.append([""]*len(dic))

    for i in range(m):
        sumlist = sum(pattern[i])
        for j in range(len(dic)):
            logpattern[i][j] = log(float(pattern[i][j])/sumlist+smallnumber)
    print logpattern

    lo = 0
    hi = 0

    for j in range(m):
        lo += min(logpattern[j])
        hi += max(logpattern[j])

    final = []
    for y in range(n-m+1):
        p = match(convertstring, pattern, y)
        pr = matchinversion(convertstring, pattern, y)
        if p > 0.98:
            final.append((y, string[y:y+m], p, "+"))
        if pr > 0.98:
            final.append((y, string[y:y+m], pr, "-"))

    #otherwise number is float and does not work
    def signum(x):
        if x > 0:
            return 1
        if x < 0: return -1
        return 0

    for x in sorted(final, lambda x, y: signum(y[2]-x[2])):
        for y in x:
            print y,
        print
    print "Count is", len(final)
```

```

f=open(sys.argv[1],'r')
str=f.readline()
f.close()
str = str.replace("\n", "")

g=open(sys.argv[2],'r')

pat = []
h = g.readlines()

#PROBLEM opening file. sometimes 2 spaces, sometimes 1. leading and trailing characters.
#string converted into list of integers
for x in range(len(h)):
    list = [int(y) for y in re.split(" +", h[x].strip(" "))]
    if len(pat) == 0:
        for y in range(len(list)):
            pat.append([""]*4)
        for i in range(len(list)):
            pat[i][x] = list[i]
print pat

pwm(str, pat)

```

Suffix Trees

```
#!/usr/bin/env python
```

```
import sys
```

```
#Longest string which can be found twice
```

```
maxl = (0, 0)
```

```
def SuffixTree(string):
```

```
    n = len(string)
```

```
    all = {}
```

```
    def addsuffix(suffix, i):
```

```
        P = all
```

```
        length = 0
```

```
        position = i
```

```
        global maxl
```

```
        for j in range(len(suffix)):
```

```
            if suffix[j] in P:
```

```
                length = length + 1
```

```
                if length > maxl[0]:
```

```
                    maxl = (length, P[suffix[j]][0], position)
```

```
                P = P[suffix[j]][1]
```

```
            else:
```

```
                P[suffix[j]] = (i, {})
```

```
                P = P[suffix[j]][1]
```

```
    for i in range(n-1):
```

```
        addsuffix(string[i:n], i)
```

```
    print "Length is", maxl[0]
```

```
    print "First position is", maxl[1]
```

```
    print "Second position is", maxl[2]
```

```
SuffixTree(sys.argv[1])
```

Random DNA Generator

```
#!/bin/env python
```

```
import sys
```

```
import random
```

```
#Random Data Set generator of a set length n
```

```
def RandomDNA(n):
```

```
    list = []
```

```
    for x in range(n):
```

```
        list.append('ACGT'[random.randint(0,3)])
```

```
    print ''.join(list)
```

```
RandomDNA(int(sys.argv[1]))
```

References

T.H. Corman, C.E. Leiserson and R.L Rivest. *Introduction to Algorithms*. The MIT Press, 1994. Algorithms

R. Durbin, S.Eddy, A.Krogh, G. Mitchison. *Biological sequence analysis* Cambridge University Press, 2004

<http://jaspar.genereg.net>

<http://www.python.org>

<http://marknelson.us/1996/08/01/suffix-trees>

http://mila.cs.technion.ac.il/~yona/suffix_tree/

http://en.wikipedia.org/wiki/Suffix_tree

<http://en.wikipedia.org/wiki/FOXC1>

Acknowledgements

Thank you very much to Professor Jotun Hein, Dr Rune Bang Lyngsø and Dr Adam Novak for their invaluable help and support throughout the project. I would also like to thank Abi Linton who worked with me and Garrett Hellenthal and Joanna Davies for their advise.

Appreciation should also go to my family and friends for their encouragement, Dr Rushton for helping me obtain a Nuffield Bursary and to Jo Lewis and the Nuffield Foundation for allowing me to carry out this project.

I found the project really fascinating and hugely enjoyable. I definitely learnt a lot and it has inspired me to carry on with this type of research at university.