Querying Highly Similar Structured Sequences via Binary Encoding and Word Level Operations

Ali Alatabbi[1], Carl Barton[1], Costas S. Iliopoulos[1,2,4], Laurent Mouchard[1,3]

[1] King's College London, Dept. of Informatics, London WC2R 2LS, UK
{ali.alatabbi,carl.barton,c.iliopoulos,laurent.mouchard}@kcl.ac.uk
[2] Curtin University, GPO Box U1987 Perth WA 6845, Australia
[3] University of Rouen, LITIS - EA4108, France
[4] University of Western Australia, Perth, Australia

Abstract. In the post-genomic era there has been an explosion in the amount of genomic data available and the primary research problems have moved from being able to produce interesting biological data to being able to efficiently process and store this information. In this paper we present efficient data structures and algorithms for the High Similarity Sequencing Problem. In the High Similarity Sequencing Problem we are given the sequences S_0, $S_1, \ldots, S_k$ where $S_j = e_{j_1} I_{\sigma_1} e_{j_2} I_{\sigma_2} e_{j_3} I_{\sigma_3}, \ldots, e_{j_\ell} I_{\sigma_\ell}$ and must perform pattern matching on the set of sequences. In this paper we present time and memory efficient datastructures by exploiting their extensive similarity, our solution leads to a query time of $O(m + vk \log \ell + \frac{mocc_v v}{w} + \frac{PSC(p)m}{w})$ with a memory usage of $O(N \log N + vk \log vk)$.

1 Introduction

The sequencing of the whole Human Genome was a celebrated breakthrough. The goal was to obtain a consensus sequence accounting for the common parts of the genomes of all humans. Storing genetic sequences of many individuals of the same species promises new discoveries for the whole field of biology, and the low cost of acquiring an individual human genome gives way to "personalized medicine", making use of one's individual genetic profile to tailor treatment to an individual. The human genome has about 3 billion DNA base pairs (bps), consisting of 23 chromosomes with lengths ranging from about 33 to 247 million bps. If a researcher or physician is dealing with many human genomes, then there is a challenge to store, communicate, and manipulate those genomes. Data structures, such as the one introduced in this paper, can address the storage and communication challenges. DNA sequences within the same species are highly repetitive and often will only have a few differences. Human genomes are identical to each other for example, only about 1%. of the $3GB$ human genome differs; the rest is common to all humans. This poses interesting challenges to efficiently store and access the data. Therefore, a delta (difference) representation that encodes the differences between two human genomes can be quite small. Although

a reference sequence is required to retrieve the information from delta representations. Most classic data compression techniques are not well prepared to deal with this tremendous redundancy. Flexible and efficient data analysis on such a typically huge collection is plausible using suffix trees. However, suffix trees occupies $O(N \log N)$ bits, which very soon inhibits in-memory analysis. Recent advances in full-text self-indexing reduce the space of suffix tree to $O(N \log \sigma)$ bits, where σ is the alphabet size. In practice, the space reduction on a Human Genome is more than 10-fold. However, this reduction factor remains constant when more sequences are added to the collection [5]. In this work, we propose a practical compressed index for the repetitive collection indexing problem. In particular, the proposed algorithm makes provision to accommodate variations that may occur in the target sequence with respect to the reference sequence. With the increasing knowledge of variants, one could simply align against all known genomes for similar species .

2 Related Work

Compression of biological sequences and querying compressed structures has been an active research area in recent times, much of this work focuses on developing indexing schemes to allow for the compression of the sequence whilst indexing them. Basic indexing techniques often construct one long string by concatenating all the sequences together and use auxiliary data structures such as suffix trees and BWT [1], to build an index of the entire set of sequences; however, when there are many sequences these are not appropriate due to the memory usage of a suffix tree $O(n \log n)$ bits. Using classical indexes in this way does not efficiently exploit the large similarities between the DNA sequences, meaning they often end up storing extra, unnecessary, information. More advanced data structures such as CSA [11] and FM-index [7] present an improvement over classical data structures and take into account the entropy of a sequence to produce a compressed index of the sequence using techniques such as block addressing.

These indexing algorithms are often called 'opportunistic datastructures', this is because they attempt to compress the text as the index is being built. This is done by considering fixed length segments of the text and detecting repetitions within them. These algorithms achieve good compression rates in terms of the entropy of the text; they have a space complexity in the order of $O(nH_k)$. Where H_k is the $k-$th order entropy of the sequence. Entropy has been shown to be a good measure of compression when you have a single repetitive text, but when you have a large number of sequences which are all highly similar the entropy of the set changes only very slightly as more sequences are added. This means that a space complexity of $O(nH_k)$ becomes unwieldy as the factor of n comes from the length of all the sequences. So unless H_k decreases proportionally to n, the memory usage will still inhibit in memory analysis. Ideally we would like to remove the dependency on n so that the space complexity is terms of the length of a single sequence and the number of differences.

Recent research has focused on exploiting the inherent similarity present in multiple DNA sequences to allow for in memory analysis of a large number of number of DNA sequences. For example, in [9], as well as the current article, a number of common segments are assumed with differences between them. This allows them to exploit the common segments for fast string searching and low memory usage. Other techniques sacrifice guaranteeing a match to improve the time complexity, these tend to use filtering techniques which reduce the number of potential matches (similar to BLAST [3] or FASTA [12], such as searching using q-grams[14] .

3 Preliminaries

An *alphabet* Σ is a finite non-empty set whose elements are called *symbols*. In this work, we consider the finite alphabet Σ for DNA sequences, where $\Sigma = \{A, C, G, T\}$. A *string* is a sequence of zero or more symbols from an alphabet Σ. The zero-symbol sequence is called the *empty string*, and is denoted by ε. The set of all the strings on the alphabet Σ is denoted by Σ^*. The set of all non-empty strings on the alphabet Σ is denoted by Σ^+. The length of a string x is denoted by $|x|$.

We denote by $x[i]$, for all $0 \leq i < |x|$, the symbol at index i of x. Each index i, for all $0 \leq i < |x|$, is a position in x when $x \neq \varepsilon$. It follows that the ith symbol of x is the symbol at position $i - 1$ in x, and that

$$x = x[0\,..\,|x| - 1]$$

A string x is a *factor* of a string y if there exist two strings u and v, such that $y = uxv$.

Let x be a non-empty string and y be a string. We say that there exists an *occurrence* of x in y, or, more simply, that x *occurs in* y, when x is a factor of y.

Every occurrence of x can be characterised by a position in y. Thus we say that x occurs at the *starting position* i in y when $y[i\,..\,i + |x| - 1] = x$. It is sometimes more suitable to consider the *ending position* $i + |x| - 1$.

Compression methods are undergoing rapid development making it more practical to store sequencing data for longer periods so that the data can be analysed with the latest techniques as they become available. This creates challenging research problems, the huge influx of data and rapidly improving analysis techniques have created the need to store and transfer very large volumes of data. More work is needed to select the perfect reference for huge data to improve difference compression generation, and to further investigate the use of the new method in practical genomic applications.

With the continuous increasing knowledge of variants between individuals, compression methods are undergoing rapid development making it tempting to store sequencing data for long periods of time so that the data can be analysed with the latest techniques. There exist many different programmes for this task. However, this procedure will come with the overhead of redundant alignments in conserved regions.

In this paper, we propose new difference compression algorithms that are suitable for storing highly repetitive collections of genomic sequences for the same species. We analysed the space-time complexity of query the proposed structure and show its improvement over existing solutions for this specific problem.

4 Problem Definition

High Similarity Sequencing: Intuitively the problem consists of performing pattern matching on a large set of sequences where each sequence in the set has a large degree of similarity and adheres to some type of structure. More formally the problem is as follows. We are given the sequences S_0, $S_1, \ldots, S_k$ where $S_j = e_{j_1} I_{\sigma_1} e_{j_2} I_{\sigma_2} e_{j_3} I_{\sigma_3}, \ldots, e_{j_\ell} I_{\sigma_\ell}$. In other words, each sequence is formed from a permutation of the identical blocks $I_1, I_2, \ldots, I_\ell$ with differences between them $e_1, e_2, \ldots, e_\ell$. We consider a model of the problem were the errors between identical segments have a size bounded by a constant factor and for all $i, j |S_i| = |S_j|$. We wish to find all occurrences of a pattern p in S_i where $0 \leq i \leq k$.

5 Preprocessing

We need to perform some preprocessing in order to reduce the memory usage of our algorithm. We use a combination of techniques used in [4] and [8] by some of the authors. The general scheme is to store the equivalent of a single sequence, named S_r, along with the differences between S_r and each S_i. Along with this we also need to store the permutations of the identical sequences in each sequence. We then use a binary encoding of the sequences in a 2 bits per base encoding (2bpb) allowing us to exploit word level operations on binary vectors to speed up pattern matching. Due to the structure of each sequence we need to store all the identical sequences and all the differentiating segments.

More formally, we store for the entire sequence

$$I = I_1 I_2 \ldots I_\ell$$

and for each sequence

$$S_i = e_1^{(i)} I_{\sigma_1} e_2^{(i)} I_{\sigma_2} \ldots e_\ell^{(i)} I_{\sigma_\ell}$$

We will store

$$\tau_i = \sigma_{i_{(1)}} \sigma_{i_{(2)}} \ldots \sigma_{I_{(\ell)}}$$

and

$$\epsilon_i = e_1^{(i)} e_2^{(i)} \ldots e_\ell^{(i)}$$

For each $e_j^{(i)}$ the size is bounded by a constant and will be $< \frac{w}{2}$, where w is the length of a computer word. We will use a 2bpb encoding scheme with the

following mapping, as used in REAL [8] so that we can map each differentiating sequence to a computer word.

- $A = 00$
- $C = 01$
- $G = 10$
- $T = 11$

In addition we store the identical sequences with the same binary encoding as this will allow for quick verification of the prefix and suffix of candidate matches later on in the algorithm.

6 Our Algorithm

The general approach we use is to exploit word level operations to perform efficient string matching on the set of sequences. More specifically we make use the following binary operation.

bitop($\sigma(x), \sigma(y)$): a word level operation that given two strings x and y, with $|x| = |y| = \gamma$ and $2\gamma \leq w$, where w is the size of the computer word, it returns $\delta_H(x, y)$, in constant time. [8]

We distinguish 4 cases in which an occurrence of the pattern could occur and outline them below:

- *Simple Matches*: p occurs entirely within an identical block I_i or entirely within a differentiating segment e_i.
- *Border Matches*: p occurs as a suffix of an identical block and either ends within an error segment or as the prefix of another identical block
- *Complex Matches*: p is of the form $j_1 I_{p_1} j_2 I_{p_2} j_3 I_{p_3}, \ldots, j_n I_{p_v}$.
- *Complex Border Matches*: p is of the form $\alpha j_1 I_{p_1} j_2 I_{p_2} j_3 I_{p_3}, \ldots, j_n I_{p_v} \beta$. Where α is a prefix of the pattern which occurs as the suffix of an identical block and β is a suffix of the pattern which occurs as a prefix of an identical block.

It is clear from the 4 cases defined above that the *complex* matches are going to be the most computationally expensive and in the rest of the paper we focus most of our attention on this case. We proceed as follows, we first cover how to find the *non-complex* cases and cover, in more detail, how to find the *complex* matches.

Non-Complex Matches The approach used to find the 2 non-complex cases is outlined below.

Simple Matches can be solved simply by using a suffix array[10] of the identical sequences to find all occurrences of the pattern in $O(m + occ)$. For a survey on suffix arrays see [13]. In the case of short patterns where the pattern could

occur entirely within a differentiating section we can check these in $O(lk)$ using the bitop operation on the binary encoded differentiating segments.

Border Matches can be solved by finding all prefixes of the pattern which occur as a suffix of an identical block using the suffix array, we denoted this in the same way as [9] by $PSC(p)$. Once we have all of these we can then use the binary encoding of the identical blocks and differentiating segment to verify these in a total of $O(\frac{PSC(p)m}{w})$ for all possible occurrences.

Complex Matches First we introduce the idea of a *valid factorisation* of the pattern with respect to the identical sequences. For a pattern p we call a factorisation of the pattern valid if takes the form $p = \mu_1 I_{p_1} \mu_2 I_{p_2} \mu_3 I_{p_3}, \ldots, \mu_v I_{p_v}$ such that $|\mu_j| \leq \frac{w}{2}$ for $1 \leq j \leq v$.

The efficient computation of all valid factorisations plays an important part in our treatment of complex matches. The approach we use for more complicated matches is as follows.

Step 1 The first step of the algorithm is to compute the valid factorisations of the pattern, p. To efficiently compute the factorisations of the pattern into this form we build an Aho-Corasick[2] automaton of all the identical blocks, $I_1 \ldots I_\ell$. We then feed the pattern into the automaton, this will give us all occurrences of the identical blocks that occur in the pattern. Once we have all the occurrences we can efficiently determine all the factorisations of the string as there are only a constant number of possible start positions to check for any valid factorisation. That is to say a valid factorisation can either start with an identical sequence or a differentiating segment and if it starts with a differentiating sequence then this differentiating sequence has a length bounded by a constant factor such that its length is $< \frac{w}{2}$.

Step 2 For each valid factorisation $p = \mu_1 I_{p_1} \mu_2 I_{p_2} \mu_3 I_{p_3}, \ldots, \mu_v I_{p_v}$ we have a sequence $\tau_p = p_1 p_2 p_3 \ldots p_v$ which is the order in which the identical sequences occur in this factorisation. We must find all the sequences, S_i, where this permutation of identical sequences occur, so we must find where τ_p occurs in a tau_i. To do this we can build another Aho-Corasick automaton of the set of valid factorisations and then feed each τ_i into the automaton.

Step 3 For each occurrence of a τ_p we must verify the differences between each I_{p_j}. To do this we can use bit parallelism techniques. Due to the short length of each of these differentiating sequences we can map them to a computer word and verify a differentiating sequence in constant time using the bitop operation.

Step 4 It now remains for us to validate the leading and trailing part of the sequence of each candidate. This is not the same as verifying the errors between sequences as this part of the sequence could be the suffix or prefix respectively of

another identical sequence. So in this instance we use the same technique used to verify the border matches and use a binary vector to represent the pattern and verify the pattern by using binary operations.

6.1 Analysis

Preprocessing For the preprocessing we need to build a Aho-Corasick automaton of I along with a suffix array which will take $O(|I|)$ for both. In addition to this we must convert every e_i into a binary word using the 2 bits per base binary encoding scheme which will take time proportional to the number of differentiating segments as the size of each differentiating segment is bounded by a constant factor, this will be $O(k\ell)$. Additionally we store each I_j in the binary encoding to allow us to quickly verify complex matches.

Algorithm In the analysis of the algorithm we focus on the complexity of finding and validating the complex and complex border matches, as the time taken to find these matches is greater than for all other cases. We continue the analysis by considering the cost of each step in the algorithm as defined above.

Step 1 This requires us to feed the pattern into the automaton and report all the occurrences of the identical sequences that occur in the pattern so that we can form all valid factorisations of the pattern. This will take a total of $O(m + occ)$, this is $O(m + occ)$ from feeding the pattern into the automaton. To form the set of valid factorisations we can store one string for each factorisation, of which there will be at most $O(m)$ and an occurrence can be added directly as the occurrence is reported. We need not worry about checking it is copied it into the correct factorisation as each occurrence will be in at most two factorisations, for an index i it may be in factorisation i and $i - |I|$. Although we may have up to $O(m)$, in practise this will only occur for extremely long patterns and the number of valid combinations will generally be far smaller.

Step 2 To find all the sequences where the order of the identical blocks is the same as that in the pattern we can build another Aho-Corasick of all the valid factorisations. The time taken to search using this will take slightly longer than linear time as we do not have a constant size alphabet. We make use of an Aho-Corasick automaton for integer alphabets as mentioned in [6]. This will take $O(vk)$ time to build as there will be $O(v)$ sequences each of length up to $O(k)$. In general the time taken to search in this type of automaton is $O(m \log |\Sigma| + occ_v)$. In our case Σ is the number of identical segments, ℓ so the total query time becomes $O(vk \log \ell + occ_v)$.

Step 3 In our preprocessing stage every e_i was encoded as a unique binary string allowing us to exploit the bit op operation to verify each possible segment in $O(1)$ time. It will take $O(v)$ time for each valid factorisation as there are up to $O(v)$ differentiating segments per match, however it should be noted that $v < m$.

In total this gives us a time of $O(occ_v v)$ to verify all valid factorisations.

STEP 4 Finally we need to check α and β for the complex matches. This is more challenging as these sections may also be a prefix or suffix of an identical segment. As we have verified all but α and β we essentially have a border match and can use the same technique that we used previously for border matches. We have the required part of each identical sequence in the 2 bits per base encoding we used previously due to the preprocessing step; we can simply match the segments using the bitop operation used previously and verify each possible segment in $O(\frac{m}{w})$.

All together this is $O(m + vk \log \ell + \frac{occ_v vm}{w} + \frac{PSC(p)m}{w})$. It should be noted that both v, k and ℓ will be very small in practise.

Theorem 1 *The algorithm correctly computes occurrences of p in the set of sequences. With query time $O(m + vk \log \ell k + v\frac{occ_v m}{w} + \frac{PSC(p)m}{w})$.*

6.2 Memory Usage

In terms of memory usage our algorithm requires $O(N \log N + \ell k)$ bits to store the sequences and to query we will use $O(N \log N + \ell k + vk \log vk)$ bits. Where v is the number of valid factorisations each of length k, N is the size of the identical segments, ℓ the number of differentiating segments, and k the number of sequences. It is clear that both $N \log N$ and $vk \log vk$ are due to the use of the Aho-Corasick automaton. This offers an improvement over simply indexing the entire set of sequences with a suffix tree as the factor of $N \log N$ will not grow as the number of sequences in the set is increased. However, it should be noted that depending on the type of sequences being considered $N \log N$ may be quite large to begin with. All other additional memory requirements are for storing the bit vectors we use in the preprocessing and for the suffix array, which all take $O(n)$ bits. We have provided a solution to a problem similar to that considered in [9], however, we take into account permutations of the identical sequences. Taking this into consideration we believe that the time-space trade-off of our solution is reasonable. For future work we aim to further reduce the memory requirements, specifically we would like to reduce the $\log N$ factor by removing the Aho-Corasick automaton for a more memory efficient alternative.

Theorem 2 *The algorithm uses $O(N \log N + \ell k)$ bits of space to store and $O(N \log N + \ell + vk \log vk)$ bits of space while querying.*

7 Conclusion and Future Work

In this paper we have presented a solution to the highly similar sequencing problem based on the use of word level operations on bit vectors. We still wish to improve the memory requirements of our approach as the use of Aho-Corasick

automaton for parts of the algorithm requires $O(n \log n)$ bits. Although we are not naively building an index of all the sequences such as a generalised suffix tree. The memory requirement will still inhibit in memory analysis at a point. Ideally we would like to reduce the memory requirements whilst maintaining the querying speed, or with only a small slowdown. We have achieved similar query times to [9] and would like to further extend out solution to take into account k-mismatches and k-differences.

References

1. D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, 1 edition, July 2008.
2. A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.
3. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, October 1990.
4. C. Barton, M. Giraud, C. Iliopoulos, T. Lecroq, L. Mouchard, and S. P. Pissis. Querying highly similar sequences. *International Journal of Computational Biology and Drug Design*, 2012. (Accepted).
5. F. Claude, A. Farina, M. A. Martínez-Prieto, and G. Navarro. Compressed q-gram indexing for highly repetitive biological sequences. In *Proceedings of the 2010 IEEE International Conference on Bioinformatics and Bioengineering*, BIBE '10, pages 86–91, Washington, DC, USA, 2010. IEEE Computer Society.
6. S. Dori and G. M. Landau. Construction of Aho Corasick automaton in linear time for integer alphabets. *Inf. Process. Lett.*, 98(2):66–72, April 2006.
7. P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, July 2005.
8. K. Frousios, C. S. Iliopoulos, L. Mouchard, S. P. Pissis, and G. Tischler. REAL: an efficient read aligner for next generation sequencing reads. In *Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology*, BCB '10, pages 154–159, New York, NY, USA, 2010. ACM.
9. S. Huang, T. W. Lam, W. K. Sung, S. L. Tam, and S. M. Yiu. Indexing similar DNA sequences. In *Proceedings of the 6th international conference on Algorithmic aspects in information and management*, AAIM'10, pages 180–190, Berlin, Heidelberg, 2010. Springer-Verlag.
10. J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, November 2006.
11. R. Lippert. Space-efficient whole genome comparisons with Burrows Wheeler Transforms. *Journal of Computational Biology*, 12(4):407–415, 2005.
12. W. R. Pearson. Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods in enzymology*, 183:63–98, 1990.
13. S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2), July 2007.
14. C. Xia, C. L. Shuai, and K. H. T. Anthony. Indexing DNA sequences using q-grams. In *In Proceedings of the 10th International Conference on Database Systems for Advanced Applications*, pages 4–16, 2005.