600.639 Computational Genomics
Final Project

Ashleigh Thomas and Jed Estep

Abstract

Easily searchable representations of genome strings are useful for many kinds of analysis, but in practice their usability is often limited on commodity hardware due to their high memory requirements. Suffix arrays are one of the least memory-intensive commonly used representations, but its space requirements may still be prohibitive in the case of indexing numerous genomes. In this paper we investigate the suffix array compression scheme described in [1]. We attempt to apply the compressed suffix arrays as a searchable database of multiple genomes with use in the context of metagenomics. The design of our database is similar to that of QUASAR [2].

1 Introduction

We arrived at this design while investigating multiple topics. From one end, we were interested in pursuing the applicability of compressed data structures to representing genomes. Many implementations of useful index structures like suffix trees are extremely memory intensive, so decreasing the size of their representation is paramount if they are to be used on commodity hardware. Literature on the topic of succinct data structures often neglects to discuss practical versions of their structures, and as such we explore how well the methods of [1] work in a real program.

From an alternative angle, we noted that most approaches to metagenomics rely on probabilistic methods, such as [3], and less attention is given to index search methods that are commonly used for read alignment. We wanted to explore the usefulness of string indexing in identifying the source of a read from an unknown genome. We operate under the following assumption: at least some reads from a particular random sample of DNA will come from one lone source, provided that the reads are short enough. In other words, the kind of DNA collected in metagenomics is not totally fragmented, but stays in clumps of some unknown length k. As long as we are searching for q-grams such that $q < k$, we should be able to identify some sources of a read using exact or approximate matching.

2 Prior Work

Burkhardt [2] points out that, in the operation of QUASAR, numerous special methods are necessary to accommodate suffix arrays which are too large to fit in main memory. As such, we attempted to apply the compression methods of Grossi and Vitter [1] to a search index similar to QUASAR. The compressed suffix array implementation of our choice has a few notable properties:

1. In comparison to compressed representations of permutations [6], it has a much faster time for random access, which is vitally important for our desired application, and
2. Its implementation has not been explored in any major public implementation, such as the SDSL library [7].

While [1] proposes a novel structure and retrieval algorithm for its suffix array, the primary technique of compression (decreasing the size of an array Ψ, whose definition is discussed both in [1] and section 3.1) can be swapped out for a more standard one for compressing vectors of integers, such as the methods of [5].

3 Methods and Software

3.1 Suffix Arrays

We implemented both a compressed and uncompressed suffix array representation, as well as a database that stores labels and genomes and can be queried with reads.

The uncompressed suffix array is implemented as an array of integers. The actual substrings are not saved in the class. In addition to standard functionality, each suffix array has the ability to create an array B. B is defined as follows [1]:

$B[i] = SA[i] \% 2$

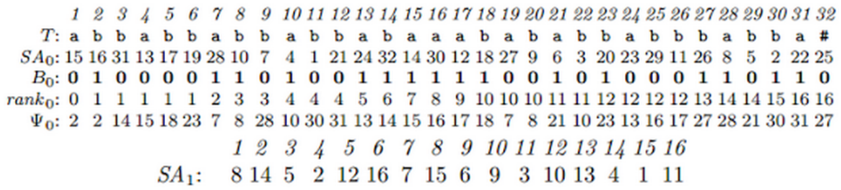
We are able to represent B as a simple `unsigned char` with each value being a single bit. Not only is this representation highly compact, but we obtain several useful properties, particularly in the computation of $rank(B, i)$. Using the SSE4.2 extension to the x86 architecture, we can compute the rank on 4-byte chunks using the `__builtin_popcount` function. This saves space over the original description, which used a rank-select dictionary to store *rank* explicitly.

The implementation of the compressed suffix array extends suffix array. Added class variables include the number of levels k, which is the maximum number of times the suffix array can be compressed; a vector of arrays B_i for $0 < i < k$, which we refer to as the *odd-even arrays*; and a vector of arrays Ψ_i, which we refer to as the *companion arrays*. Ψ is defined as follows [1]:

$\Psi[i] = i$ if $SA[i] \% 2 = 0$
$\Psi[i] = j$ where $SA[j] = SA[i] + 1$ otherwise

Additionally, a lookup method is implemented based on [1] with several small modifications; we describe the need for these in Section 5.

Figure 1: *All information produced after a single level of compression. Figure courtesy of [1].*



	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*	*30*	*31*	*32*
T:	a	b	b	a	b	b	a	b	b	a	b	b	a	b	a	a	a	b	a	b	a	b	b	a	b	b	b	a	b	b	a	#
SA_0:	15	16	31	13	17	19	28	10	7	4	1	21	24	32	14	30	12	18	27	9	6	3	20	23	29	11	26	8	5	2	22	25
B_0:	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**0**
$rank_0$:	0	1	1	1	1	1	2	3	3	4	4	4	5	6	7	8	9	10	10	10	11	11	12	12	12	12	13	14	14	15	16	16
Ψ_0:	2	2	14	15	18	23	7	8	28	10	30	31	13	14	15	16	17	18	7	8	21	10	23	13	16	17	27	28	21	30	31	27

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
SA_1:	8	14	5	2	12	16	7	15	6	9	3	10	13	4	1	11

Each array Ψ_i is implemented as a standalone class, holding a reference to the array B_i and an integer array, which is a cache of the values represented by Ψ_i, and which we will call `values`. While [1] describes a succinct method for representing `values`, we chose to investigate a different approach. The paper mentions defining `values` as a total function, i.e. it contains the mapped value for every valid input. In practice, this is not necessary, as $\Psi_i[j] = j$ whenever $B_i[j] = 1$. Therefore, we store only half the possible values of Ψ_i in `values`, specifically those for which $B_i[j] = 0$. While only half the values in the codomain of Ψ_i are stored explicitly, storing them directly led to an increase in overall size of the data structure when compared to a standard suffix array. As such, we stored them in a compressed integer array from the SDSL-Lite library implementation by Simon Gog [7]. This vector class first encodes its integers as their deltas, then applies a self-delimiting code to the result. [4].

We made overall a number of decisions in this implementation which differed from the initial report substantially. These decisions were made in the context of ignoring theoretical bounds in places where they were not meaningful, in favor of building a real, practical implementation. The first major change is that *rank* is not represented explicitly at all. Each time we compute $rank(i)$, we perform an $O(|B_i|)$ calculation, which seems drastically unimproved from the constant bound proposed in the paper. But in reality, the `__builtin_popcount` instruction operates with incredible speed; computing the rank of a 100 million byte array took around 220ms. Thus, while the theoretical bound of lookup is increased, the practical speed is not seriously impacted and no extra bits are used to store *rank*.

We also tried multiple algorithms for building the uncompressed suffix array, which is a requirement for building a compressed suffix array. Our initial algorithm uses a sorted hashmap with key-value pairs `std::<string, int>`, holding a substring and its index. Once built, the suffix array is the list of values. Using the C++ class `std::map`, which has $O(\log n)$ inserts in general but amortized constant, we got a worst case bound for the build time of $O(n \log n)$. However, the map requires $O(n^2)$ space; while this is all freed once the array is built, it can be prohibitive on larger string inputs. We also used a more common method of sorting the strings in length-based blocks, increasing by a power of 2 each time. This approach runs in $O(n \log^2 n)$ and uses $O(n)$ extra memory. While we hoped that this method would run faster due to not requiring memory paging, in practice both implementations had impractically long build times for large strings. As such, our tests were run on smaller data, using the former $O(n \log n)$ method of construction.

3.2 Database

There is an implementation of a class that represents an entry in the database, called a *genome entry*. This consists of a label (the name of the species that the genome is from), the genome, and the associated compressed suffix array of this genome.

Finally, we implement the database in a straightforward manner as a `std::vector` of genome entries. Users can query the database with strings and receive the labels of the genomes that the read is contained in within the database. The database holds a vector of genome entries. The primary query API is the function `getGenomeLabel(const string&)`. This method calls a basic binary search on each genome entry in the database, which searches through the genome to find the correct index if one exists. When landing on long substrings, the binary search checks if the input read is a prefix of the substring; this leads our binary search to be $O(n \log m)$ (where $n = |genome|$ and $m = |input|$) in the worst case, though in practice it is often faster than that as most suffixes are much shorter than n.

4 Results

We have produced significant compression in our implementation. We ran our implementation on a randomly generated read of 60,000 nucleotides. The suffix array took up 240,048 bytes, while the compressed suffix array took up 106,998 bytes. This is a compression of 56%. There is a time performance penalty in constructing the compressed suffix array; while the uncompressed version built in 281ms, the compressed version built in 1352ms. We found that indexing performance on the two of them was nearly identical.

While the above results suggest that build times are reasonable, we found that for strings of a length appropriate for an assembled chromosome, the time required to build even an uncompressed suffix array was not practical (at least on `ugrad12`). Not only did this bottleneck prevent us from testing on large datasets, as even after 15 minutes or more the suffix arrays were still building, but it revealed an important underlying assumption of [1]: that a complete, uncompressed suffix array is always available. For the purpose of a database, which will be computing the compressed suffix arrays only once (and, in a more sophisticated implementation, caching them on disk when possible), it still precludes the idea of using the compressed suffix array on commodity hardware. Even if the amount of RAM required to store the compressed array is small enough to be practical for an average computer, the uncompressed suffix array still needs to be built and stored in RAM as an intermediate result. This diminishes the usefulness of a compressed suffix array for an application of this kind.

5 Conclusions

While we have made significant improvements on the space that a suffix array takes up, there are several key points to consider about this implementation.

First, it is important to note that the method described by Grossi and Vitter actually increases the size of the suffix array, unless the array Ψ is bit-compressed (using either their method or another). Therefore the size of the compressed suffix array becomes larger than that of the original suffix array. This is repaired by using the `enc_vector` encoding [5, 7]. This utilizes deltas in order to compress the compressed suffix array further, contributing to the 56% compression.

Grossi and Vitter use one-indexed arrays in their algorithm. This brings up a few problems, as the algorithm will work in only two cases. The first is that the arrays are one-indexed and the genome string is of even length. This is because if we are using a one-indexed array and an odd length genome, the largest index of the suffix array is odd but has no companion. This breaks the calculation of Ψ. The second case is when the arrays are zero-indexed and the genome string is of odd length. In [1], the authors explore building compressed suffix arrays on strings whose length is a power of 2, and do not appear to consider the case of odd length strings. While it is easy in theory to ignore this case, in practice it is unreasonable to assume all input strings have even length.

Although the explored method is both efficient in practical search applications once built and offers substantial data compression, we feel overall that the method is not necessarily practical. The construction of an initial suffix array may be prohibitive in its running time if the suffix array is to be built on commodity hardware; in the case of running on a computing cluster, it is not clear that compression is especially necessary for holding multiple structures concurrently in RAM.

References

[1] Roberto Grossi and Jeffrey Vitter, *Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching.* Society for Industrial and Applied Mathematics Journal of Computing, Vol. 35, No. 2, pp. 378-407, 2005.

[2] Stefan Burkhardt, et al., *q-gram Based Database Searching Using a Suffix Array (QUASAR).* Proceedings of the third annual international conference on Computational molecular biology, pp. 77-83, 1999.

[3] Arthur Brady and Steven Salzberg, *Phymm and PhymmBL: Metagenomic Phylogenetic Classification with Interpolated Markov Models.* Nature Methods, 2009.

[4] Daisuke Okanohara and Kunihiko Sadakane, *Practical Entropy-Compressed Rank/Select Dictionary.* Proceedings of the 9th Workshop on Algorithm Engineering and Experiments, ALENEX, 2007.

[5] Peter Elias, *Universal Codeword Sets and Representation of the Integers.* IEEE Transactions on Information Theory, Vol. 21, No. 2, 1975.

[6] Jeremy Barbay and Gonzalo Navarro, *Compressed Representations of Permutations, and Applications.* Technical Report TD/DCC-2008-18, University of Chile, 2008.

[7] Simon Gog, *Succinct Data Structures Library lite.* https://github.com/simongog/sdsl-lite, 2008.

[8] —, *Suffix Arrays.* http://codeforces.com/blog/entry/4025, 2011.