Using Bloom Filters for Large Scale Gene Sequence Analysis in Haskell

Ketil Malde[1] and Bryan O'Sullivan[2]

[1] Institute of Marine Research, Bergen, Norway
ketil.malde@imr.no
[2] Serpentine Green Design, San Francisco, USA
bos@serpentine.com

Abstract. Analysis of biological data often involves large data sets and computationally expensive algorithms. Databases of biological data continue to grow, leading to an increasing demand for improved algorithms and data structures. Despite having many advantages over more traditional indexing structures, the Bloom filter is almost unused in bioinformatics. Here we present a robust and efficient Bloom filter implementation in Haskell, and implement a simple bioinformatics application for indexing and matching sequence data. We use this to index the chromosomes that make up the human genome, and map all available gene sequences to it. Our experiences with developing and tuning our application suggest that for bioinformatics applications, Haskell offers a compelling combination of rapid development, quality assurance, and high performance.

1 Introduction

A central part of bioinformatics involves work with biological sequences. These sequences represent molecules of DNA, RNA, and protein, all of which are structured as long chains of smaller building blocks. For computational purposes, these chains are usually represented as strings over fixed alphabets. For instance, the nucleotides of DNA are represented using the alphabet of A (for adenine), C (cytosine), G (guanine), and T (thymine).

Since the introduction of large-scale sequencing in the early 1990s, public sequence databases have doubled in size every 18 months. The U.S. National Center for Biotechnology Information's GenBank database now contains 110 million nucleotide sequences, totaling 200GB of data[1].

Over the past two decades, the cost of generating new sequences has dropped by three orders of magnitude. As this trend is likely to continue, the rate at which biological sequences are produced will increase dramatically. For instance, the newest generations of pyrosequencing technologies produce hundreds of megabytes of sequence data per run [19][23][7].

Much of bioinformatics research involves the development of "throwaway" code that integrates preexisting components to create focused analytic tools that

[1] http://www.nih.gov/news/health/apr2008/nlm-03.htm

A. Gill and T. Swift (Eds.): PADL 2009, LNCS 5418, pp. 183–194, 2009.

have short lifespans. For many tasks, such as accessing and manipulating data from the more than 1 000 known public databases [10], languages like Python and Perl are widely used, with performance-critical analysis delegated to code written in languages such as C and C++.

An ideal situation for bioinformaticians is to be able to develop new analytic tools rapidly, without sacrificing speed or correctness. With these goals in mind, we used Haskell to prototype some novel uses of Bloom filters for sequence analysis.

1.1 Sequence Similarity

The core of sequence analysis is the search for similarity between sequences. Similarity provides the basis for many important tasks, for example:

- Genes are usually identified based on their similarity to known proteins and gene transcripts.
- The sequencing process commonly produces only fragments of the true sequence. These fragments are clustered by similarity, and then assembled by joining fragments whose ends are similar.
- Identifying similar regions of genomes from different organisms can reveal evolutionary relationships between those organisms, and shed light on the mechanisms of evolution.

Applications like these are ubiquitous. They are usually computationally expensive due to the size of the data sets involved.

A commonly used metric for sequence similarity is the edit distance or Levenshtein distance, which is the number of edit operations needed to transform one sequence into another. The edit distance between two sequences n and m can be calculated using dynamic programming in $O(nm)$ time [11][22][20]. This approach quickly becomes impractical for large sequences, and heuristic methods are usually used instead.

1.2 Word-Based Approaches

Heuristic approaches typically start by identifying fixed-size exact matches, called k-words[2]. Once a sufficient number of matches is identified, they are used as a starting point (or *seed*) to construct a more accurate alignment or comparison score.

The choice of k-word size is influenced by several factors. Sequences often contain errors introduced by the sequencing process, or differ due to mutations. Words should therefore be short enough that the number of false negatives is reasonable. For instance, if the data have a (rather severe) error rate of 5%, a word size of less than 20 will ensure that hits can be found. On the other hand, shorter words are less likely to be unique in a data set, which increases the chance of false positives. The inherent non-randomness of genes and genomes amplifies this problem.

[2] These are also known as q-grams, or k-tuples.

An index can store k-words either directly, using tables, or in a sparse data structure. The simplest approach is to use each word as an index in a table of size α^k, where α is the alphabet size. This approach is used by e.g. BLAT [13], which by default indexes words of length 11. To reduce the density of the index, BLAT only indexes non-overlapping words and removes words that occur frequently in the data set. As the table grows exponentially with word size, available memory limits the possible word lengths. Although longer words are often desirable, to make efficient use of memory they require sparse data structures like hash tables or search trees. This incurs additional overheads in space and time.

1.3 Suffix Trees and Arrays

Suffix trees [25] and suffix arrays [18] provide interesting alternatives to word-oriented indexing, as they allow searching for words of arbitrary length. They form the basis of several tools for sequence analysis, e.g. [2][16][12]. While both suffix trees and suffix arrays can be constructed in linear time, and can perform lookups of a length-m string in $O(m)$ time, this comes at a cost of about $12m$ bytes per position with 32-bit pointers [1]. Suffix structures are thus memory intensive. Unlike the word-based approaches, it is not straightforward to reduce memory use by omitting frequent or overlapping words. In addition, while the sensitivity of word-based indexing can be improved using gapped words [24], it is not clear how to apply this approach to suffix structures.

1.4 Bloom Filters

The Bloom filter [4] is a set-like data structure that uses space efficiently. Unlike a normal set data structure, its query operation is probabilistic: it may report false positives. The error rate is tunable: an application that can tolerate a higher error rate will consume less memory than one with stricter needs.

For example, to represent a 400 000-element set with a 1% false positive rate, a Bloom filter will use 0.46MB of memory. If we reduce the false positive rate to 0.01%, the space consumption doubles, to 0.91MB. The size of a Bloom filter does not depend on the sizes of its elements. In our case, this property offers the prospect of efficiently indexing long sequences.

A Bloom filter is implemented as an m-bit array and a family of h distinct hash functions. The empty set is represented as a zeroed bit array. To add an element, we compute h hashes over it. We use each hash value as an offset into the array, and set each corresponding bit to 1. To query the set for membership, we compute h hashes over the input. If any corresponding bit is not 1, the element is not present in the array. False positives arise if distinct values hash to the same offsets for all h hash functions.

Although Bloom filters are widely used in networking [5] and formal methods [9], they are almost unknown in bioinformatics. In the sections that follow, we discuss their use to implement solutions to some typical bioinformatics problems, and investigate how they perform on massive data sets.

2 Methods

2.1 A Fast Bloom Filter in Haskell

We implemented a Bloom filter in Haskell. Our library is general purpose in nature[3], and provides typical Haskell interfaces to construct and query immutable Bloom filters:

```
fromList :: (a -> [Hash])       -- family of hash functions
         -> [a]                 -- elements to add
         -> Bloom a

elem :: a -> Bloom a -> Bool
```

To achieve a false positive rate of 0.1% for an input list of known size, we use a family of 10 hash functions. Building a Bloom filter requires many modifications to a bit array, in this case 10 per element added. We use the ST monad [15] to efficiently make in-place modifications to this bit array, then freeze it to present an immutable interface to consumers of the library.

We avoid developing many independent hash functions by using Dillinger and Manolios's technique of double hashing [9]. We compute two hashes over a value, and combine their results using cheap algebraic operations to produce further hash values on demand. Although the resulting hash values are not independent, analysis has shown them to provide good enough dispersion for practical use [14].

We double our hashing performance by computing both hashes in a single traversal of an element, by using Haskell's foreign function interface (FFI) to invoke Jenkins's `hashlittle2` implementation[4].

We also use a power-of-two table size, so that we can perform cheap bit-manipulation operations to turn a hash value into a valid array index.

2.2 Indexing Sequences with Bloom Filters

We used the Bloom filter to implement a simple indexing scheme for biological sequences. As with other indexing schemes, the sequences are cut into fixed-length overlapping fragments that can be stored in the Bloom filter.

We allow a choice of word length and overlap (the distance between the beginnings of successive words). These parameters can be tuned to optimize the trade-off between sensitivity, specificity, and resulting index size. For instance, given the sequence *GATTACCA*, a word length of 3, and an overlap of 2, the index would store the three words *GAT*, *TTA*, and *ACC*. In our test application, we use a word size of 30 and an overlap of 6. The Bloom filter is configured to give a false positive rate of 0.005. As our implementation limits filter sizes to powers of two for efficiency, the observed false positive rate may be substantially lower in practice.

[3] `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/bloomfilter`
[4] `http://burtleburtle.net/bob/hash/`

To calculate a distance between a *query* sequence and a *target* sequence, we index the target using a Bloom filter, then score the query sequence against it. If the Bloom filter uses an overlap of 1—i.e. every word from the target sequence is used in the Bloom filter—the score is the number of words from the query that match the filter. With larger overlaps, we match every word from the query sequence against the filter, but remove matches that occur closer than the overlap. Typically, such matches arise from spurious similarities to unrelated parts of the target, or highly repetitive sequences.

We can also calculate the expected number of false positives introduced by the Bloom filter, to quantify their effect on result quality. Under the assumption that the probability of a false positive result is word-independent, we can model false positives using a binomial distribution. Given a number of lookups n and false positive rate p, the expected number of false positives is np, with standard deviation $\sqrt{np(1-p)}$.

We implemented a simple application that reads a set of FASTA-formatted files containing target sequences, and builds a Bloom filter for each. Query sequences are then read from standard input, and matched against the Bloom filters, and the best hit is reported.

To compare the efficiency of Bloom filter indexing to other approaches, we also implemented versions of the application that use a balanced binary tree (using the standard Haskell module Data.Set) with ByteString elements. Since comparison of strings requires time proportional to their lengths, this is not an optimal strategy, and we therefore also implemented a version using words encoded as integers [17].

2.3 Applications and Data

We benchmarked our program in two different settings. We began by filtering ESTs for contaminants. We then clustered ESTs by matching them to chromosomes.

Sets of expressed sequence tags, or *EST*s, are an important source of genomic information. These sequences are produced from messenger RNA gene transcripts. ESTs are usually incomplete, and thus represent fragments of genes. In addition, error rates are high—typically about 0.5–1% even in regions of relatively high quality.

The current release of GenBank contains over eight million human ESTs, representing 4.2 gigabytes of data. We downloaded these from the University of California, Santa Cruz web site[5].

The human genome is about 3 billion nucleotides in length, split into 23 chromosomes. We downloaded the set of sequences representing these chromosomes from UCSC[6].

An EST originates from a gene that resides on a chromosome. Knowing the location of each EST helps with a number of tasks, among which are identifying

[5] ftp://hgdownload.cse.ucsc.edu/goldenPath/hg18/bigZips/est.fa.gz

[6] `ftp://hgdownload.cse.ucsc.edu/goldenPath/hg18/bigZips/chromFaMasked.zip`

the gene; identifying its full extent and internal structure; and discovering or identifying surrounding patterns that regulate the expression of the gene. We thus used our application to cluster ESTs by identifying their chromosomes of origin. An EST is assigned to the chromosome with most matching k-words if the number of matches is statistically significant.

Our second application filtered sequences for contamination. As part of the sequencing process, the molecules to be sequenced are inserted into a host organism (typically the bacterium *E. coli*) for mass production. Occasionally, genomic DNA from the host organism is retrieved and sequenced instead of the desired sequence, thus contaminating the resulting sequence data with unwanted sequences. It is therefore necessary to screen sequence data by comparing it to the *E. coli* genome, and remove the offending sequences before further analysis.

While human chromosomes range up to 240 megabases (Mb) in size, the 5Mb *E. coli* genome is relatively small. To provide a smaller test case for comparing different indexing implementations, we also downloaded the genome for one strain of *E. coli* from GenBank[7] .

All tests were performed on a single core of a 2.4GHz Intel Core2 processor, using version 6.8.3 of the GHC Haskell compiler.

3 Results

We randomly selected ESTs in sets of various sizes, and benchmarked the three different indexing implementations by matching the ESTs against the *E. coli* genome. The times are shown in Figure 1, we see that while integer matching is faster than strings, the Bloom filter substantially outperforms both. A linear regression shows that the Bloom filter indexing stage takes only 1.7 seconds, compared to 20.2 for the Integer-encoded and 11.9 for the string-based indexing. Similarly, the Bloom filter matches 1718 sequences per second, compared to 589 and 310 for the Integer and string based indexes, respectively.

Perhaps more important than time spent is memory consumption. Time affects how long we must wait for a result, but excessive memory consumption prevents us from successfully processing large sequences. While the set based implementations allocate 160–190MB of memory (as measured by `top`) for this test, the Bloom filter application runs in a mere 20MB, of which the Bloom filter itself uses only 2MB.

By comparing the outputs from the set-based and the Bloom-filter-based implementations, we can measure the number of actual false positives generated. The results from the 10K data set are displayed in Figure 2. Here, 76.5% of the sequences generated no false matches, and only 370 sequences had two or more false matches.

Figure 2 also shows the expected number of false matches, calculated separately for each sequence. Here, we see clearly that due to the power of two rounding of the Bloom filter size, the observed false positive rate is lower than the requested rate.

[7] `http://www.ncbi.nlm.nih.gov/entrez/viewer.fcgi?db=nuccore\&id=56384585`

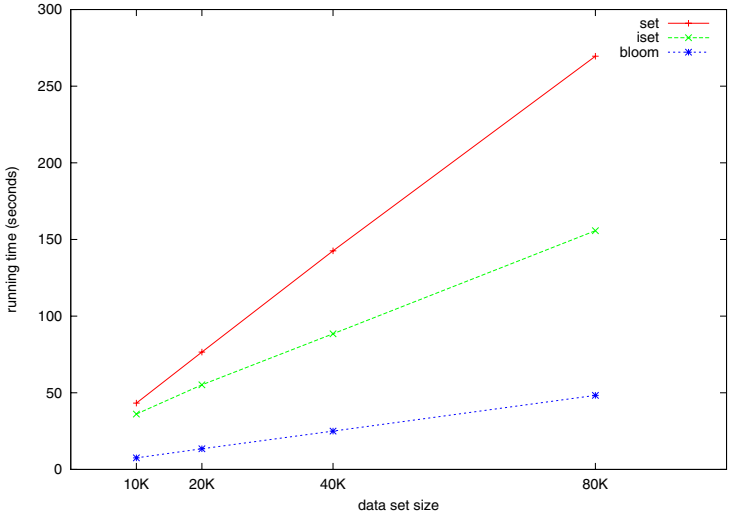

Fig. 1. Times (in seconds) using the Bloom filter, sets of ByteStrings, and sets of integers to index the *E. coli* genome, and match sets of sequences against it

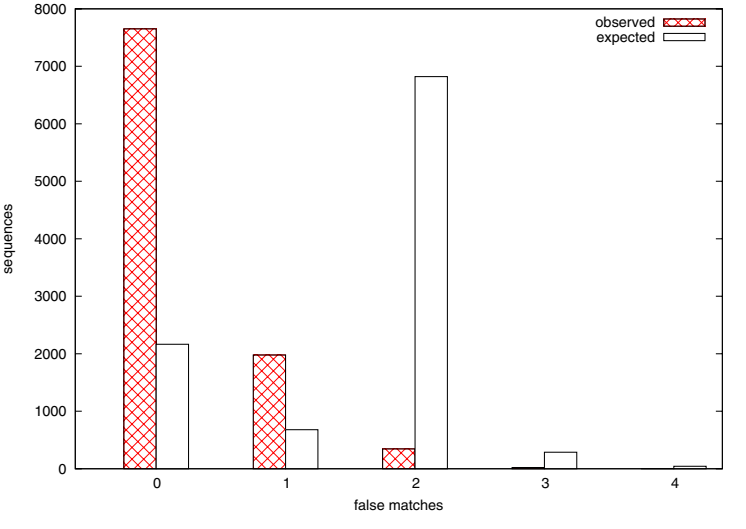

Fig. 2. False positives introduced by the Bloom filter from the 10K data set. 7 652 of the sequences have no false matches, 1 978 have one false match, 345 have two, 22 have three, and 3 sequences have four false matches. Also the expected false positives, calculated as described in Section 2.2.

Finally, we built Bloom filters for the 23 chromosomes constituting the human genome, and matched all ESTs against them. Indexing the chromosomes took 26 minutes, and the resulting Bloom filters consumed from 16 to 64MB of

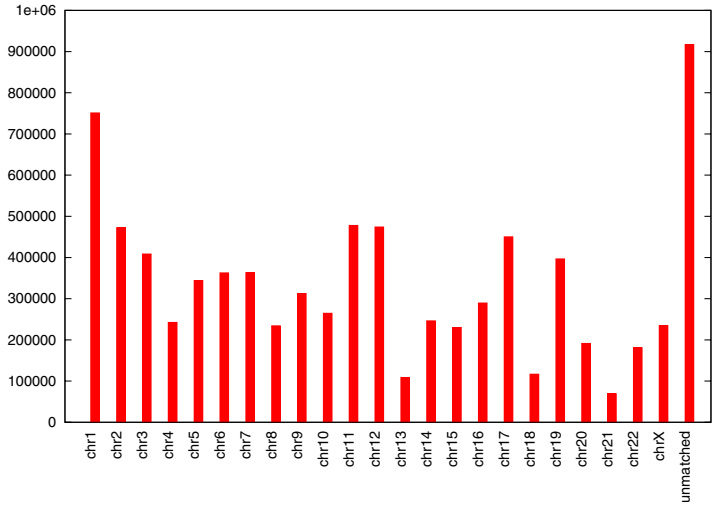

Fig. 3. Matching sequences by chromosome. As the chromosome sequences used have had repetitive regions masked, a large fraction of sequences are left unmatched.

memory, depending on chromosome size. The total memory used for the entire genome was 864MB. Matching the set of eight million ESTs against the Bloom filters took 49 hours. The resulting distribution of sequence locations is shown in Figure 3.

The chromosome sequences used here were masked, where repetitive regions were erased to avoid false positives. Many genes reside in masked regions, which seems to be the most likely explanation for 916 583 sequences that failed to match any of the chromosome. Manual checks have confirmed this for a small sample. Other possible explanations can be low quality sequence causing false negatives, or contamination.

4 Discussion and Conclusion

4.1 Performance Tuning Experiences

Early profiling of our application's performance indicated that the Bloom filter library accounted for over 70% of run time, even though we had addressed performance early on by double hashing in a single pass and using power-of-two sizes for bit arrays. Further investigation caused us to make a number of substantial changes. These all remained internal to the Bloom filter implementation, and did not affect its public interface.

By default, the GHC compiler checks the bounds of array accesses at run time, and we had somehow missed this early on. Switching to the alternative "unsafe" interfaces doubled our performance, by eliminating branches from an inner loop.

The lazy variant of the ByteString data type represents strings in chunked form [8], so a sequence can straddle multiple chunk boundaries. The Jenkins

hash functions operate over contiguous C strings. To address this, we began by concatenating chunks into one contiguous string. In addition, the ByteString library by default makes a defensive copy of data that must remain immutable, to protect it from modification by native code. We were thus copying every ByteString at least once, and those that straddled a chunk boundary twice. We reimplemented the Jenkins hash code to operate incrementally over ByteString chunks and eliminated the defensive copying from the ByteString library, thereby doubling our performance.

GHC performs runtime safety checks on the bounds of bit-shifting operations. Even given constant shift values, we were unable to predict the circumstances under which GHC would eliminate those checks from our code. To ensure uniform branch-free performance, we implemented our own bit-shifting functions using GHC's word-level primitives. The branches thereby eliminated netted us a performance gain of perhaps 20%.

Many of our low-level optimizations were motivated by reading dumps from GHC's simplifier phase, using Stewart's `ghc-core` tool[8]. Although simplifier output is challenging to read, with some experience it gives a clear picture of when unnecessary memory allocations, or unboxing and reboxing operations, are occurring.

Faced with a number of potentially unsafe code transformations, we used the QuickCheck testing tool [6] to give ourselves statistical confidence that our code remained correct. We found its ability to provide us with a test case when a test failed to be invaluable in quickly directing us to the sources of bugs. For instance, when we rewrote the Jenkins hashing code to consume chunks incrementally, we wrote a QuickCheck property to ensure that the hash of a contiguous string was the same as the hash of a chunked string. Checking this property over successively larger random inputs exposed three subtle errors in our handling of boundary conditions during chunk traversal.

The final speed of our Bloom filter was approximately five times better than when we began, and came within 8% of a C program that we had written to offer a point of comparison. This experience suggests that low-level Haskell performance tuning can be highly profitable. With extensive use of QuickCheck, we keep the risk of unsafe changes low, and create working code more quickly.

While we have begun writing about the practicalities of Haskell performance analysis and tuning in [21], there remains plenty of scope for further compiler improvements; more experience reports; and better tool support to assist programmers in writing faster (but still safe!) Haskell code.

4.2 Bloom Filters, Bioinformatics and Haskell

The Bloom filter is a tremendously useful data structure: in settings such as ours, it holds substantial advantages over traditional indexing schemes by allowing indexing of large data sets with long word sizes quickly, and with low memory consumption.

[8] `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/ghc-core`

The industry-standard alignment tool BLAST [3] aligns approximately 40 sequences per second when aligning the 10K data set against the *E. coli* genome. As BLAST identifies the alignments while our indexing merely detect the *presence* of a similarity, this result is not directly comparable to the results reported above. Nonetheless, it illustrates the potential benefit of using the Bloom filter as a preprocessing stage to eliminate unlikely candidates for a match.

To turn our sample implementation into an industry strength tool, there are many options to be explored: The impact of false positives could be reduced by requiring two or more consecutive (that is, spaced apart by exactly the overlap length) matches. Instead of counting matches along the length of the sequence, we could count matches within a fixed-size region (window). We could improve sensitivity by using gapped word indices [24]. Sequence quality should be taken into account. Our objective here has been to demonstrate the efficacy of Bloom filters in this application, and we therefore defer exploring these possibilities for the future.

We developed our prototype over the course of a few days, using preexisting Haskell libraries to parse sequence data and manipulate Bloom filters. The application-specific code amounted to 75 lines.

The field of bioinformatics is, in a sense, divided in two. On one side are standard algorithms and data structures, which must be highly optimized to deal with large data sets. These are typically implemented in C. On the other side are analysis pipelines, often project-specific, which combining such tools to perform a complete analysis. Here, rapid development is important, and scripting languages are often used.

We have demonstrated that we can address both sides of this divide with Haskell. The efficient implementation of crucial data structures such as ByteStrings and Bloom filters allows the application programmer to implement pipelines of functions, and from there entire tools, in a straightforward, even naïve, way, and still achieve both excellent performance and a high degree of confidence in results.

Acknowledgments

The authors wish to thank Shannon Engelbrecht for her extensive comments on drafts of this manuscript.

KM is supported by grant NFR 183640/S10 from the national Functional Genomics Programme (FUGE) of the Research Council of Norway.

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing Suffix Trees with Enhanced Suffix Arrays. Journal of Discrete Algorithms 2(1), 53–86 (2004)
2. Abouelhoda, M.I., Ohlebusch, E., Kurtz, S.: Optimal Exact String Matching Based on Suffix Arrays. In: Laender, A.H.F., Oliveira, A.L. (eds.) SPIRE 2002. LNCS, vol. 2476, pp. 31–43. Springer, Heidelberg (2002)
3. Altschul, S., Gish, W., Miller, W., Myers, E., Lipman, D.: A basic local alignment search tool. Journal of Molecular Biology 215(3), 403–410 (1990)

4. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)
5. Broder, A., Mitzenmacher, M.: Network applications of Bloom filters: A survey. Internet Mathematics 1(4), 636–646 (2003)
6. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: ACM SIGPLAN Notices, pp. 268–279. ACM Press, New York (2000)
7. Cloonan, N., Forrest, A.R.R., Kolle, G., Gardiner, B.B.A., Faulkner, G.J., Brown, M.K., Taylor, D.F., Steptoe, A.L., Wani, S., Bethel, G., Robertson, A.J., Perkins, A.C., Bruce, S.J., Lee, C.C., Ranade, S.S., Peckham, H.E., Manning, J.M., McKernan, K.J., Grimmond, S.M.: Stem cell transcriptome profiling via massive-scale mRNA sequencing. Nature Methods 5(7), 613–619 (2008)
8. Coutts, D., Stewart, D., Leshchinskiy, R.: Rewriting haskell strings. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 50–64. Springer, Heidelberg (2006)
9. Dillinger, P.C., Manolios, P.: Bloom filters in probabilistic verification. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 367–381. Springer, Heidelberg (2004)
10. Galperin, M.Y.: The molecular biology database collection: 2008 update. Nucleic Acids Research 36, D2–D4 (2008)
11. Gotoh, O.: An improved algorithm for matching biological sequences. Journal of Molecular Biology 162, 705–708 (1982)
12. Kalyanaraman, A., Aluru, S., Brendel, V., Kothari, S.: Space and time efficient parallel algorithms and software for EST clustering. IEEE Transactions on Parallel and Distributed Systems 14(12), 1209–1221 (2003)
13. Kent, W.J.: BLAT—the BLAST-like alignment tool. Genome Research 12(4), 656–664 (2002)
14. Kirsch, A., Mitzenmacher, M.: Less hashing, same performance: Building a better bloom filter. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 456–467. Springer, Heidelberg (2006)
15. Launchbury, J., Jones, S.L.P.: Lazy functional state threads. In: Programming Languages Design and Implementation, pp. 24–35. ACM Press, New York (1994)
16. Malde, K., Coward, E., Jonassen, I.: Fast sequence clustering using a suffix array algorithm. Bioinformatics 19(10), 1221–1226 (2003)
17. Malde, K., Schneeberger, K., Coward, E., Jonassen, I.: RBR: Library-less repeat detection for ESTs. Bioinformatics 22(18), 2232–2236 (2006)
18. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
19. Margulies, M., Egholm, M., Altman, W.E., Attiya, S., Bader, J.S., Berka, L.A.B.J., Braverman, M.S., Chen, Y.-J., Chen, Z., Dewell, S.B., Du, L., Fierro, J.M., Gomes, X.V., Godwin, B.C., He, W., Helgesen, S., Ho, C.H., Irzyk, G.P., Jando, S.C., Alenquer, M.L.I., Jarvie, T.P., Jirage, K.B., Kim, J.-B., Knight, J.R., Lanza, J.R., Leamon, J.H., Lefkowitz, S.M., Lei, M., Li, J., Lohman, K.L., Lu, H., Makhijani, V.B., McDade, K.E., McKenna, M.P., Myers2, E.W., Nickerson, E., Nobile, J.R., Plant, R., Puc, B.P., Ronan, M.T., Roth, G.T., Sarkis, G.J., Simons, J.F., Simpson, J.W., Srinivasan, M., Tartaro, K.R., Tomasz3, A., Vogt, K.A., Volkmer, G.A., Wang, S.H., Wang, Y., Weiner4, M.P., Yu, P., Begley, R.F., Rothberg, J.M.: Genome sequencing in microfabricated high-density picolitre reactors. Nature 437(7057), 376–380 (2005)

20. Needleman, S., Wunsch, C.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology 48(3), 443–453 (1970)
21. O'Sullivan, B., Stewart, D., Goerzen, J.: Real World Haskell. In: Profiling and optimization, ch. 25. O'Reilly Media, Sebastopol (2008)
22. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. Journal of Molecular Biology 147, 195–197 (1981)
23. Steemers, F.J., Gunderson, K.L.: Illumina profile: technology and assays. Pharmacogenomics 6(7), 777–782 (2005)
24. Valle, G.: Discover 1: a new program to search for unusually represented DNA motifs. Nucleic Acids Research 21(22), 5152–5156 (1993)
25. Weiner, P.: Linear pattern matching algorithms. In: Proceedings of 14th IEEE Symposium on Foundations of Computer Science (FOCS), pp. 1–11 (1973)