A Deterministic Finite Automaton for Faster Protein Hit Detection in BLAST

Michael Cameron[1], Hugh E. Williams[2], and Adam Cannane[1]

[1] School of Computer Science and Information Technology
RMIT University, GPO Box 2476V, Melbourne, Australia, 3001
{mcam, cannane}@cs.rmit.edu.au

[2] Microsoft Corporation
One Microsoft Way, Redmond, Washington 98052-6399 U.S.A
hughw@microsoft.com

February 8, 2006

Abstract

BLAST is the most popular bioinformatics tool and is used to run millions of queries each day. However, evaluating such queries is slow, taking typically minutes on modern workstations. Therefore, continuing evolution of BLAST — by improving its algorithms and optimisations — is essential to improve search times in the face of exponentially-increasing collection sizes. We present an optimisation to the first stage of the BLAST algorithm specifically designed for protein search. It produces the same results as NCBI-BLAST but in around 59% of the time on Intel-based platforms; we also present results for other popular architectures. Overall, this is a saving of around 15% of the total typical BLAST search time. Our approach uses a deterministic finite automaton (DFA), inspired by the original scheme used in the 1990 BLAST algorithm. The techniques are optimised for modern hardware, making careful use of cache-conscious approaches to improve speed. Our optimised DFA approach has been integrated into a new version of BLAST that is freely available for download at `http://www.fsa-blast.org/`

Keywords: BLAST, homology search, sequence alignment

1 Introduction

BLAST is staggeringly popular: it is used over 120,000 times each day (McGinnis and Madden, 2004) at the NCBI website[1], it is installed at hundreds of institutes, several hardware-dependent implementations are available and many variations of its algorithms have been published including PSI-BLAST (Altschul et al., 1997), MegaBLAST (Gotea et al., 2003), PHI-BLAST (Zhang et al., 1998) and RPS-BLAST (Schaffer et al., 1999). Indeed, the 1997 BLAST paper (Altschul et al., 1997) has been cited over 10,000 times[2] in electronically-available manuscripts.

While BLAST is popular, it it not perfect. A typical search of a fraction of the GenBank database requires several minutes on modern desktop hardware. With perhaps millions of queries each day being run, this represents a very significant investment in computing resources. Accordingly, there is enormous benefit to any improvement of the BLAST algorithm that can reduce typical runtimes without affecting accuracy. This paper proposes such an improvement to the first stage of the algorithm.

The first stage of BLAST — and many other homology search tools — involves identifying *hits*: short, high-scoring matches between the query sequence and the sequences from the collection being searched. The definition of a hit and how they are identified differs between protein and nucleotide searches, mainly because of the difference in alphabet sizes. For example, for BLAST nucleotide search, an exact match of length 11 is required. However, for protein search, it requires a match of length 3 and inexact matches are permitted. Indeed, BLAST uses an algorithmically different approach to hit detection for protein and nucleotide searches.

Several nucleotide search tools use *spaced seeds* to perform hit detection, including MegaBLAST (Gotea et al., 2003), BLAT (Kent, 2002), BLASTZ (Schwartz et al., 2002) and PatternHunter (Li et al., 2004; Ma et al., 2002). Spaced seeds use a binary mask string, such as `111010010100110111`, where matches in all of the `1` positions are required for a hit and `0` denotes allowed mismatches. PatternHunter uses spaced seeds to achieve better sensitivity and faster search times than BLAST for nucleotide queries but is not suitable for searching larger collections such as GenBank (Li et al., 2004). MegaBLAST, BLAT and BLASTZ are significantly less sensitive than BLAST. Brown (2005) presents a framework for using spaced seeds in protein search and demonstrates that spaced seeds can achieve comparable search accuracy to the continuous seed used in BLAST and produce roughly 25% as many hits. However, we observe in Section 2.2 that reducing the number of hits does not necessary translate to faster search times. Further, Brown does not present experimental results comparing the two approaches. Therefore, the original BLAST approach of scanning the entire database using a continuous seed remains, to our knowledge, the fastest and most sensitive method for performing protein search.

In this paper, we investigate algorithmic optimisations that aim to improve the speed of protein search. Specifically, we investigate the choice of data structure for the hit matching process, and experimentally compare an optimised implementation of the current NCBI-BLAST codeword lookup approach to the use of a deterministic finite automaton (DFA). Our DFA is highly optimised and carefully designed to take advantage of CPU cache on modern workstations. Our results show that the DFA approach reduces total search time by 6%–30% compared to codeword lookup, depending on platform and parameters. This represents an around 41% reduction in the time required by BLAST to perform hit detection; this is an important gain, given the millions of searches that are executed each day. Furthermore, the new approach can be applied to a range of similar protein search tools. We also explore the effect of varying the word length and neighbourhood threshold on the hit detection process.

[1] See: `http://www.ncbi.nlm.nih.gov/`

[2] See: `http://scholar.google.com/`

This paper is organised as follows. We describe the BLAST algorithm and experiment with varying the word size and threshold in Section 2. In Section 3 we discuss the data structures used for hit detection in BLAST. We compare our new DFA approach to the original codeword lookup scheme in Section 4. Finally, we provide concluding remarks and describe planned future work in Section 5.

2 Background

This section presents a background on the four stages of the BLAST algorithm, a detailed description of the first stage hit detection process used in NCBI-BLAST, and key BLAST parameters.

2.1 BLAST

The BLAST algorithm has four stages. The first identifies *hits* and is the focus of this paper. The second performs ungapped extensions, the third performs gapped alignment, and fourth computes tracebacks for result presentation to the user. A brief summary of stages two to four is presented here, and detail can be found elsewhere (Altschul et al., 1990, 1997).

Stage 1

In the first stage, BLAST carries out a comparison of a query sequence q to each subject sequence s from the collection of sequences being searched using the algorithm of Wilbur and Lipman (1983). To do this, fixed-length overlapping subsequences of length W are extracted from the query sequence q and the current subject sequence s. For example, suppose $W = 3$ and the following short sequence is processed: `ACDEFGHIKLMNPQ`. The *words* extracted from the sequence are as follows: `ABC`, `BCD`, `CDE`, `DEF`, `EFG`, `GHI`, `HIK`, and so on. In the first stage, all subject sequences are exhaustively, sequentially processed from the collection being searched, that is, each subject sequence is read from the database, parsed into words of length W, and matched against the query.

The matching process identifies words in the query that are hits with words in the subject. Hits may not be exact: a match between a pair of words is considered high scoring if the words are identical, or the match scores above a given threshold T when scored using a mutation data matrix. The T parameter is discussed in Section 2.2.

To match query and subject words, BLAST uses a lookup table — as illustrated in Figure 1 — constructed from the query sequence and alignment scoring matrix. The table contains an entry for every possible word, of which there are a^W for an alphabet of size a; the example in Figure 1 illustrates the table for an alphabet of size $a = 3$, with three symbols `A`, `B`, and `C`, and a word length $W = 2$. Associated with each word in the table is a list of query positions that denote the one or more offsets of a hit to that word in the query sequence. In the example table, the word `AB` has hits at query positions 1 and 3. The query sequence is shown at the base of the figure, where the exact word `AB` occurs at position 1 and a close-matching hit (`CB`) occurs at position 3.

The search process using the lookup table is straightforward. First, the subject sequence is parsed into words. Second, each subject word is searched for in the query lookup table. Third, for each matching position in the query, a hit is recorded. Last, when a hit is recorded, a pair of positions, i, j, that identify the match between the query and subject are passed to the second stage.

The first stage of NCBI-BLAST consumes on average 37% of the total search time (Cameron et al., 2004). In Section 3, we discuss the design of the lookup table in more detail, including how entries are accessed and query positions are stored within the structure.

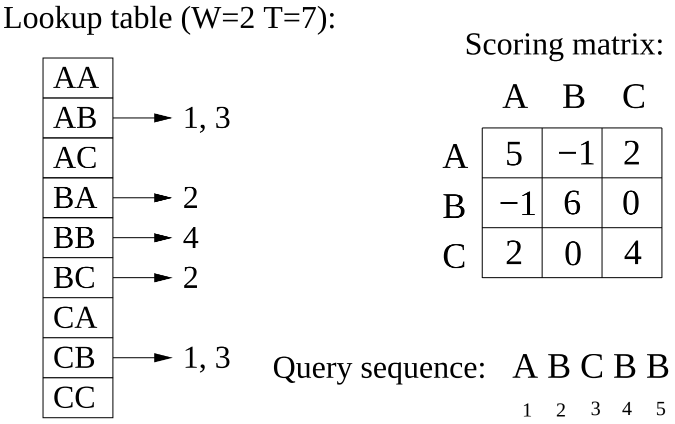

Figure 1: *The lookup table used to identify hits. In this example, the alphabet contains 3 characters:* `A`*,* `B`*, and* `C`*. The lookup table to the left is constructed using the scoring matrix and query sequence to the right with a word size $W = 2$ and threshold $T = 7$. The table contains $a^W = 3^2 = 9$ entries, one for each possible word, and each entry has an associated list of query positions. The query positions identify the starting position of words in the query that score above T when aligned to the subject word.*

Stage 2

The second stage determines the *diagonal* d of each hit by calculating the difference in the query and subject offsets, that is, $d = j - i$. If two hits $h_1[i_1, j_1]$ and $h_2[i_2, j_2]$ are located on the same diagonal and are within A residues, that is, $i_2 - i_1 \leq A$, an *ungapped extension* is performed. An ungapped extension links two hits by computing scores for matches or substitutions between the hits; it ignores insertion and deletion events, which are more computationally expensive to evaluate and calculated only in the third and fourth stages. After linking the hits, the ungapped extension is processed outward until the score decreases by a pre-defined threshold. If the resulting ungapped alignment scores above the threshold $S1$, it is passed to the third stage of the algorithm. The second stage of NCBI-BLAST consumes on average 31% of the total search time (Cameron et al., 2004).

The first two stages are illustrated on the left side of Figure 2. The short black lines represent high-scoring hits of length W between a query and subject sequence. There are eight hits in total and two cases where a pair of hits are located on the same diagonal less than A residues apart. For these two cases, an ungapped extension is performed as illustrated by a grey line. In this example, the longer, central ungapped alignment scores above the threshold $S1$ and is passed onto the third stage of the algorithm; the shorter alignment to the left does not.

BLAST can also be run in one hit mode, where a single hit rather than two hits is required to trigger an ungapped extension. This leads to an increase in the number of ungapped extensions performed, increasing runtimes and improving search accuracy. To reduce the number of hits, a larger value of the neighbour threshold T is typically used when BLAST is run in one hit mode. The original BLAST algorithm (Altschul et al., 1990) used the one hit mode of operation, and the approach of using two hits to trigger an ungapped extension was one of the main changes introduced in the 1997 BLAST paper (Altschul et al., 1997).

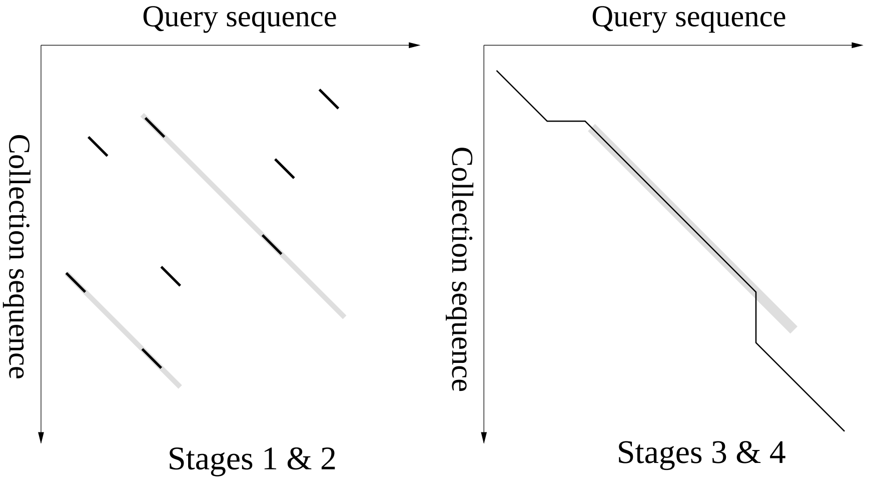

Figure 2: *The four stages of the BLAST algorithm. In stage 1, short hits between the query and subject are identified, shown as short black lines in the left figure. In stage 2, an ungapped extension is performed, as shown as grey lines in the left figure. The right figure illustrates gapped alignment, where the high-scoring ungapped alignment from stage 2 is explored in stages 3 and 4.*

Stages 3 and 4

In the third stage, dynamic programming is used in an attempt to find a gapped alignment that passes through the ungapped region. To do this, a start or *seed point* is chosen from the ungapped alignment, then dynamic programming is used to find the highest-scoring gapped alignment that passes through that point. If a high-scoring alignment is found, it is passed to the fourth stage where alignment traceback information is recorded for display to the user.

The right side of Figure 2 illustrates the gapped alignment stages. In this example, the high-scoring ungapped alignment identified in the second stage is considered as the basis of a gapped alignment. The black line shows the resulting high-scoring alignment identified in stage three. The third and fourth stages of NCBI-BLAST consume on average 32% of the total search time. We have recently described optimisations to these stages elsewhere (Cameron et al., 2004).

2.2 Varying the word size and threshold

The T parameter described in the previous section affects the speed and accuracy of BLAST. Consider the example mutation data matrix and query sequence shown in Figure 1 and a setting of $T = 7$. Observing the matrix, a match between the word BA and BC scores 8, since the intersection of the row and column labelled B is a score of 6 and between A and C is 2. Since the threshold is $T = 7$, a match between BA and BC is a hit, and so occurrences of BA or BC in the query match occurrences of BA or BC in the subject. High values of T restrict the number of *neighbourhood words* that match a word, resulting in less sensitive but fast search. Low values of T expand the number of neighbourhood words, with the opposite effects. Careful choice of T, for each W and scoring matrix pair, is crucial to BLAST performance.

NCBI-BLAST uses default values of $W = 3$ and $T = 11$ for protein search (Altschul et al., 1997). To investigate the effect of varying these two parameters we conducted the following experiment: for each value of $W = 2, 3, 4, 5$, we identified a value of T that provided a similar accuracy to the default of $W = 3$ and $T = 11$. We then used our own implementation of BLAST to perform 100 searches for each parameter pair between queries selected randomly from the GenBank non-redundant protein

Table 1: *A comparison of BLAST statistics for pairs of W and T that result in similar accuracy. The average number of hits, ungapped extensions, and gapped extensions and SCOP test ROC_{50} scores are shown.*

Settings	Total Hits	Total Extensions		ROC_{50}
		(Ungapped)	(Gapped)	score
W = 2, T = 10	812,500,081	80,571,433	53,063	0.382
W = 3, T = 11	428,902,038	17,785,578	47,264	0.380
W = 4, T = 13	219,446,068	6,113,409	43,682	0.380
W = 5, T = 15	111,574,045	3,670,768	40,116	0.378

database and the entire database. We recorded the average number of hits, ungapped extensions, and gapped extensions for each parameter pair. The results of this experiment are shown in Table 1.

Accuracy was measured using the SCOP test, a technique frequently used to assess retrieval performance (Brenner et al., 1998; Chen, 2003; Park et al., 1998). For the test, we used version 1.65 of the ASTRAL Compendium for Sequence and Structure Analysis (Chandonia et al., 2004). The test provides a *Receiver Operating Characteristic* (ROC) score between 0 and 1, where a higher score reflects better sensitivity and selectivity. The version of the GenBank database used throughout this paper was downloaded 17 November 2004 and contains 2,163,936 sequences in around 712 megabytes of sequence data. Our version of BLAST that was used for testing is referred to as FSA-BLAST (which stands for *Faster Search Algorithm - BLAST*) and is available for download from `http://www.fsa-blast.org/`. FSA-BLAST also uses the semi-gapped and gapped alignment algorithms described in our previous work (Cameron et al., 2004).

Table 1 shows that a longer word length W can be used to achieve comparable accuracy with far less computation. For example, the parameter settings $W = 5$ and $T = 15$ achieve similar accuracy to $W = 2$ and $T = 10$, while generating 86% fewer hits, 95% fewer ungapped extensions, and 14% fewer gapped extensions. Unfortunately, the reduction in computation for long word lengths does not necessarily translate to faster runtimes. This is because the data structures required for lookups with long word lengths are much larger and cannot be maintained in *CPU cache*.

CPU caches are typically no more than one megabyte in size, and store data that has been recently accessed or data that is located near recently-accessed data. CPU cache performance, design, and characteristics vary across platforms and architectures. However, in general, for memory-based tasks, data structures that are smaller and cluster related data make better use of cache and are faster on most architectures. Such data structures and algorithms are referred to as being *cache conscious*.

The next section discusses cache-conscious data structures and algorithms for the first stage of BLAST. Our aim in developing these structures is to permit the computational savings of longer word lengths to result in faster runtimes.

3 Data Structures for Fast Hit Matching

In this section, we discuss data structures for hit detection during the BLAST first stage. We describe in detail the codeword lookup table used by NCBI-BLAST, and then our new cache-conscious deterministic finite automaton. Results are presented in Section 4.

3.1 NCBI-BLAST Codeword Lookup

We have shown a schematic of the lookup table used by NCBI-BLAST in Figure 1. This section describes the implementation in more detail.

In NCBI-BLAST, each collection to be searched is pre-processed once using the *formatdb* tool and each amino-acid coded as a 5-bit binary value. The representation is 5 bits because there are 24 symbols — 20 amino-acids and 4 IUPAC-IUBMB amino-acid wildcard character substitutions — and $24 < 2^5$. For fast table lookup, W 5-bit representations are read and concatenated together, then used as an offset into the codeword lookup table; this provides unambiguous, perfect hashing, that is, a guaranteed $O(1)$ lookup for matching query positions to each subject sequence word. The table has a^W slots for an alphabet of size a and word length W. For $a = 24$ symbols and $W = 3$, a word requires 15 bits and the table has $24^3 = 13,824$ slots.

During search, collection sequences are read code-by-code, that is, 5-bit binary values between 0 and 23 decimal inclusive are read into main-memory. Since codewords overlap, each codeword (except the first two in each sequence) shares two symbols with the previous codeword. Therefore, to construct the current codeword, 5 bits are read from the sequence, and binary mask (&), bit-shift (<<), and OR (|) operations applied. Consider an example. After reading codes for the letters `T`, `Q`, and `A`, the current codeword contains a 15-bit representation of the three-letter codeword `TQA`. Suppose the code for `C` is read next, and the codeword needs to be updated to represent the codeword `QAC`. To achieve this, three operations are performed: first, a binary masking operation is used to remove the first 5 bits from the codeword, resulting in a 10-bit binary representation of `QA`; second, a left binary shift of 5 places is performed; and, last, a binary OR operation is used to insert the code for `C` at the end of the codeword, resulting in a binary representation of the new codeword `QAC`. The new codeword is then used to look up an entry in the table.

Each slot in the lookup table contains four integers. The first integer specifies the number of hits in the query sequence for the codeword, and the remaining three integers specify the query positions of up to three of these hits. If there are more than three hits, the first query position is stored in the table and the remaining query positions are stored outside the table, with their location recorded in the entry in place of the second query position. Figure 3 illustrates a fraction of the lookup table: the example shows two codewords that have two and zero query positions, and one codeword that uses the external structure to store a total of five word positions.

Storing up to three query positions in the lookup table improves spatial locality and, therefore, takes advantage of CPU caching effects. A codeword with less than three hits can be accessed without jumping to an external main-memory address. Assuming slots are within cache, the query positions can be retrieved without accessing main-memory. However, storing query positions in the table increases the table size: as the size increases, less slots are stored in the CPU cache.

In addition to the primary lookup table, NCBI-BLAST uses a secondary lookup table to reduce search times. For each word, the secondary table contains one bit that indicates if there are any hits for that word. If no hits exist, the search advances immediately to the next word without the need to search the larger primary table. This potentially results in faster search times because the smaller secondary table is compact (it stores a^W bits) and can be accessed faster than the larger primary table.

The average sizes of the NCBI-BLAST codeword lookup tables for the values of W and T used previously are shown in the left column of Table 2. As W increases, so does the table size: when $W = 4$, the table is around 12 Mb in size, much larger than the available cache on most modern hardware. This is probably why word sizes of $W = 4$ or larger are disabled by default in NCBI-BLAST. The secondary table is considerably smaller than the primary table across the range of word lengths, so that codewords that do not generate a hit are less likely to result in a cache miss.

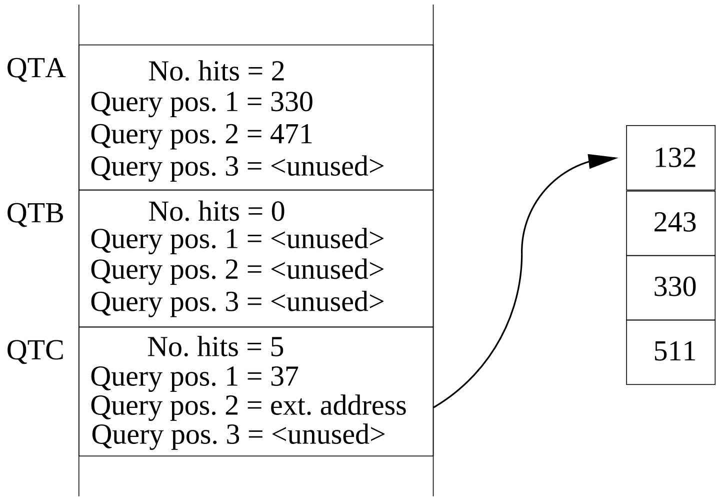

Figure 3: *Three slots from the lookup table structure used by NCBI-BLAST. In this example, $W = 3$ and the words* `QTA`*,* `QTB`*, and* `QTC` *have two, zero, and five associated query positions respectively. Since* `QTC` *has more than three query positions, the first is stored in the table and the remaining positions are stored at an external address.*

Table 2: *Average size of the codeword lookup and DFA lookup table structures for 100 queries randomly selected from the GenBank non-redundant protein database. Values of W and T with comparable accuracy were used. Experiments were conducted using our own implementation of the DFA structure and the codeword lookup structure used by NCBI-BLAST version 2.2.10 with minor modifications to allow word sizes 4 and greater.*

BLAST Parameters		Codeword lookup		Deterministic finite
		Primary	Secondary	automaton
W = 2	T = 10	13 Kb	$\ll$ 1 Kb	2 Kb
W = 3	T = 11	392 Kb	3 Kb	22 Kb
W = 4	T = 13	12,329 Kb	96 Kb	257 Kb
W = 5	T = 15	393,374 Kb	3,072 Kb	3,011 Kb

3.2 Deterministic finite automaton

The original version of NCBI-BLAST (Altschul et al., 1990) used a deterministic finite automaton (DFA) (Sudkamp, 1997) similar to the one we describe here, but it was abandoned in 1997 for the lookup table approach described in the previous section. In this section, we propose a new cache-conscious DFA design for fast codeword lookup. We first describe the original DFA structure used by BLAST, and then describe our new cache-conscious DFA design and several optimisations that can be applied to the new structure.

The DFA approach is fundamentally different to the lookup table scheme. Rather than generating a unique numeric value for each codeword, the lookup structure represents the query as *states* and *transitions* between those states. The DFA structure employed by the original BLAST is as follows: each possible prefix of length $W-1$ of a word is represented as a state, and each state has transitions to a possible next states. Associated with each transition is a list of zero or more query positions.

Figure 4 shows a portion of an example DFA that has been constructed using a simplified

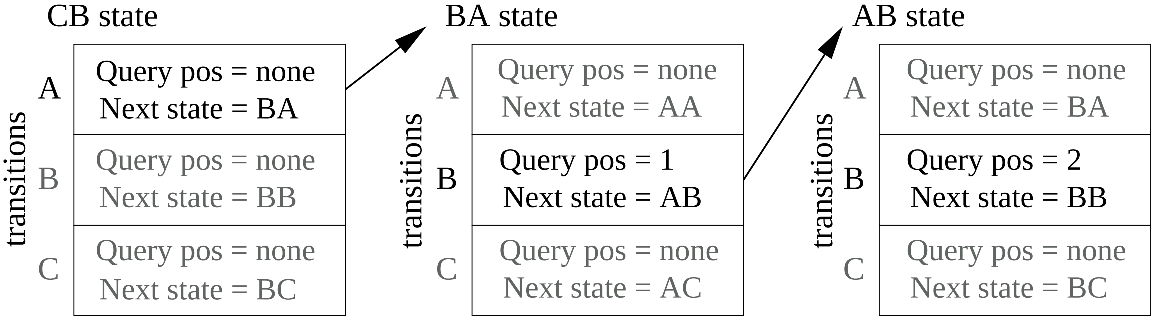

Figure 4: *The original DFA structure used by BLAST. In this example $a = 3$ and the query sequence is* `BABBC`*. The three states shown are those visited when the subject sequence* `CBABB` *is processed.*

alphabet with size $a = 3$ and an example query sequence `BABBC`. Three states with a total of nine transitions are shown. The `B` transition from the `BA` state contains the single query position 1 because the word `BAB` appears in the query beginning at the first character. Similarly, the `B` transition from the `AB` state contains the single query position 2 because the word `ABB` appears in the query beginning at the second character.

The structure is used as follows. Suppose the subject sequence `CBABB` is processed. After the first two symbols `C` and `B` are read, the current state is `CB`. Next, the symbol `A` is read and the transition `A` from state `CB` is followed. The transition does not contains any query positions and the search advances to the `BA` state. The next symbol read is `B` and the `B` transition is followed from state `BA` to state `AB`. The transition contains the query position 1, which is used to record a hit. Finally, the symbol `B` is read and this produces a single hit because the `B` transition out of state `AB` contains the query position value 2.

The structure we have described is as used in the original version of BLAST. The original DFA implementation used a linked list to store query positions and stored redundant information, making it unnecessarily complex and not cache-conscious. This is not surprising: in 1990, general-purpose workstations did not have onboard caches and genomic collections were considerably smaller, making the design suitable for its time. However, the general DFA approach has several advantages over the lookup table approach, which we have exploited in our own DFA design:

1. It is more compact. The lookup table structure has unused slots, since eight of the 5-bit codes are unused
2. Flexibility is possible in where query positions are stored, without affecting caching behavior. We take advantage of this in the DFA structure we propose: we store query positions outside of the matching structure, immediately preceding the states; this reduces the likelihood of a cache miss when an external list is accessed. We terminate query position lists with a zero value
3. Words do not map to codewords, that is, to offsets within the structure. Therefore, data can be organised to cluster frequently-accessed states, maximising the chances that frequently-accessed data is cached. In the DFA we propose, we store the most-frequently accessed states clustered at the centre of the structure; to do this we use the Robinson and Robinson background amino-acid frequencies (Robinson and Robinson, 1991)
4. Transitions within each state can be arranged from most- to least-frequently occurring amino-acid residue, improving caching effects and minimising the size of the offset to the query positions. We do this in our DFA by assigning binary values to amino-acids in descending order of frequency

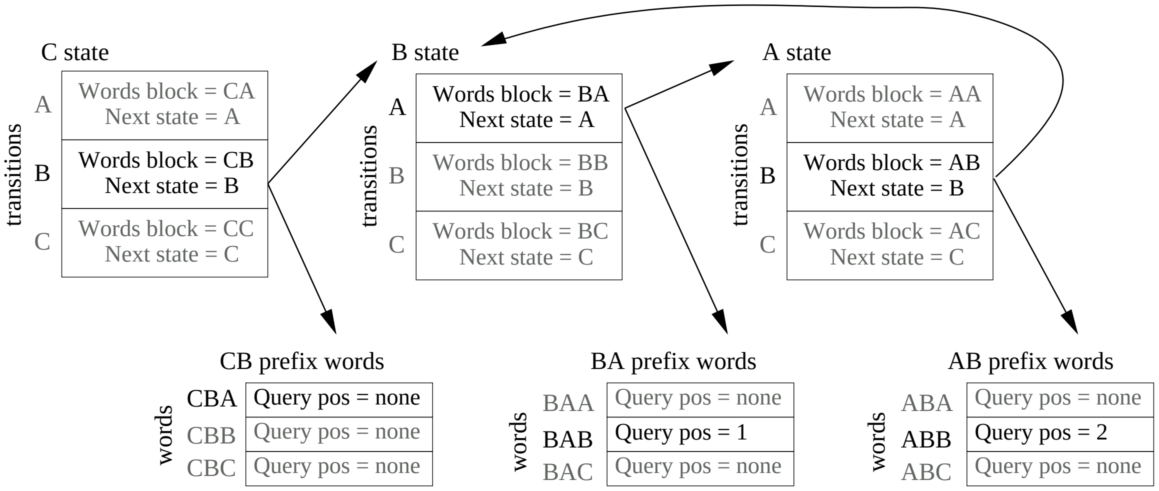

Figure 5: *The new DFA structure. In this example $a = 3$ and the query sequence is* `BABBC`*. The figure shows the portion of the new structure that is traversed when the example subject sequence* `CBABB` *is processed.*

A drawback of the original DFA is the requirement of an additional pointer for each word, resulting in a significant increase in the size of the lookup structure. To examine this effect, we experimented with 100 query sequences randomly selected from the GenBank database and found that pointers consume on average 61% of the total size of the structure when default parameters are used. However, it is possible to reduce the number of pointers as we discuss next.

The next state in the automaton is dependent only on the suffix of length $W - 1$ of the current word and not the entire word. We can therefore optimise the structure further: each state corresponds to the suffix of length $W - 2$ of a word, and each transition corresponds to the suffix of length $W - 1$. Each transition has two pointers: one to the next state, and one to a collection of words that share a common prefix. The words are represented by entries that contain a reference to a list of query positions. As each symbol is read, both pointers are followed: one to locate the query positions for the new word and the other to locate the next state in the structure.

This new DFA is illustrated in Figure 5. Again, we use the example query sequence `BABBC` and the diagram only shows the portion of the structure that is traversed when the subject sequence `CBABB` is processed. Consider now the processing of the subject sequence. After the first symbol `C` is read, the current state is `C`. Next, the symbol `B` is read and the `B` transition is considered. It provides two pointers: the first to the new current state, `B`, and the second to the `CB` prefix words. The next symbol read is `A` and two events occur: first, the word `CBA` from the `CB` prefix words is checked to obtain the query positions for the word, of which there are none in this example; and, second, the current state advances to `A`. Next, the symbol `B` is read and the word `BAB` in the collection of `BA` prefix words accessed and a single hit is recorded, with query position 1. The current state then advances to `B`. Finally, the symbol `B` is read and the word `ABB` of the `AB` prefix words is consulted and a single hit is recorded, with query position 2.

This new DFA structure requires more computation to traverse than the original: as each symbol is read, two pointers are followed and two array lookups are performed. In contrast, the original DFA structure requires one pointer to be followed and one array lookup to be performed. However, the new DFA is considerably smaller and more cache-conscious, since the structure contains significantly fewer pointers. Despite each state containing two pointers, there are $a^{(W-1)}$ instead of a^W states. Further, there are additional optimisations that have been applied to the new DFA, as we describe

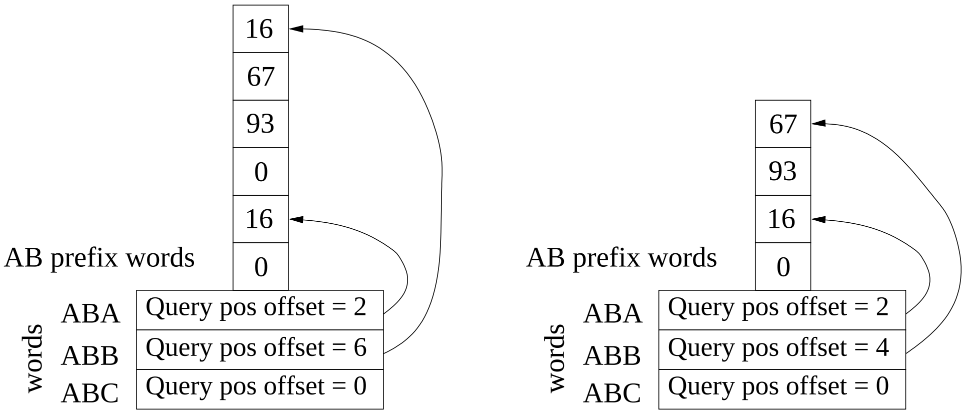

Figure 6: *Example of query position reuse. The word* ABB *produces a hit at query positions 16, 67, and 93, the word* ABA *produces a hit at query position 16 and the word* ABC *does not produce any hits. A simple arrangement is shown on the left. A more space efficient arrangement that reuses query positions is shown on the right.*

next.

The new structure offers the ability to reuse collections of words with a common prefix, such as those shown in Figure 5. If two collections share the same set of query positions, for the same respective suffix symbols, a single collection can be used for both. In this event, the *words block* field of the two transitions point to a single collection of words. This reuse leads to a further reduction in the overall size of the lookup structure.

In the new DFA, query positions are stored immediately preceding each collection of words with a common prefix. The distance between the first word in the collection and the start of the query positions list is recorded for each word, where a zero value indicates an empty list. As part of this new structure, we have developed a technique for reusing query positions between words with a common prefix. An existing list of query positions $M = m_1, ..., m_{|M|}$ can be reused to store a second list $N = n_1, ..., n_{|N|}$ in memory if $|N| \leq |M|$ and N is the suffix of M, that is, $M_{|M|-i} = N_{|N|-i}$ for all $0 \leq i \leq |N|$. Because the order of the query positions within an entry is unimportant, the lists can be rearranged to allow more reuse. We have developed a greedy algorithm that produces a near-optimal arrangement for minimising memory usage. The algorithm processes the new lists from shortest to longest, and considers reusing existing lists in order from longest to shortest. A more detailed study of the query position rearrangement and reuse problem is planned for future work. However, in practice, our results show that our current approach is efficient and effective. An example of query position reuse is shown in Figure 6. A simple and less space efficient arrangement is shown on the left-hand side. On the right-hand side of the figure the query positions are rearranged to permit reuse between the words ABA and ABB. The strategy further reduces the table size without any computational penalty.

To further optimise the new structure for longer word lengths, we have reduced the alphabet size from $a = 24$ to $a = 20$ during construction of the structure by excluding the four wildcard characters: V, B, Z, and X. These four characters are highly infrequent, contributing a total of less than 0.1% of symbol occurrences in the GenBank non-redundant database. To do this, we replace each wildcard character with an amino-acid symbol that is used only during the first phase of BLAST. This has a negligible effect on accuracy: Table 3 shows there is no perceivable change in ROC score for the SCOP test, despite a small change in total hits between the queries and subject sequences. The approach of replacing wildcard characters with bases is already employed

Table 3: *Effect on search accuracy of substituting non-wild characters for wildcards during the first stage of BLAST search. ROC_{50} scores were measured using the SCOP database and number of hits was measure by searching 100 randomly selected queries against the entire GenBank non-redundant database. Our BLAST implementation was used.*

	No wildcard substitution		Wildcard substitution	
	Total hits	ROC_{50}	Total hits	ROC_{50}
W = 2, T = 10	812,500,081	0.382	815,203,770	0.382
W = 3, T = 11	428,902,038	0.380	429,296,279	0.380
W = 4, T = 13	219,446,068	0.380	219,705,636	0.380
W = 5, T = 15	111,574,045	0.378	111,785,042	0.378

by BLAST for nucleotide searches, as originally proposed by Williams and Zobel (1997).

Our final optimisation is to store query positions as 16-bit integers where possible, instead of 32-bit integers as used in NCBI-BLAST. In the case where 32-bit integers are required — because the query exceeds 65,536 symbols in length — we use those instead.

The size of this optimised deterministic finite automaton for values of $W = 2, 3, 4, 5$ is shown in Table 2. The new structure is considerably smaller than the NCBI-BLAST codeword table: for example, when $W = 4$ it is roughly 2% of the size of the codeword lookup table, and small enough to fit into the available cache on most modern processors.

4 Results

This section presents the results of our experiments with various implementations of the first stage of BLAST.

We present a comparison of the NCBI-BLAST codeword lookup approach, our optimised implementation of the NCBI-BLAST approach, and an implementation of our optimised DFA scheme. All code was compiled using the same compiler flags as NCBI-BLAST. The NCBI-BLAST implementation is version 2.2.10 with a minor modification to permit experimentation with word lengths $W = 4$ and greater. All experiments used default SEG filtering (Wootton and Federhen, 1993) of the query sequences and NCBI-BLAST default parameters except where noted.

Collections and queries are as described in Section 2.2. The best elapsed time of three runs was recorded for each query, and then query times averaged across 100 queries. All experiments were carried out on machines under light load, with no other significant processes running. Four modern workstations were used in the experiments: an Intel Pentium 4 2.8 GHz with 16Kb L1 cache, 1Mb L2 cache and 2 Gb of RAM; an Intel Xeon 2.8GHz with 16Kb L1 cache, 512Kb L2 cache, 1Mb L3 cache and 2 Gb of RAM; an Apple PowerMac G5 dual processor 2.5GHz with 64Kb L1 cache, 512Kb L2 cache and 1.5 Gb of RAM; and, a Sun UltraSPARC-IIIi 1280 MHz with 96Kb L1 cache, 1Mb L2 cache and 1 Gb of RAM.

Table 4 shows a comparison of our implementation of the NCBI table-based approach to the original NCBI-BLAST version. With our optimisations — including minor changes to reduce the computation involved in constructing codewords and accessing the primary and secondary lookup tables — elapsed query times for stage one are typically around 70% to 80% of the NCBI-BLAST times. The exception is the PowerMac G5, where our optimisations result in an 8% speed up. We believe the different performance on the PowerMac is due to a relative difference in fundamental costs — of shifts, additions, binary OR, and increments — between it and the other platforms. We use our implementation as a baseline in the experiments reported in the remainder of this section.

Table 4: *A comparison of BLAST stage one times between NCBI-BLAST and our own implementation of table-based codeword lookup hit detection, using default parameters of $W = 3$ and $T = 11$. The percentage of the NCBI-BLAST runtime is also shown.*

Machine	NCBI-BLAST	Optimised Codeword	
	(secs)	(secs)	(%)
Intel Pentium 4	10.92	8.63	79%
Intel Xeon	10.60	8.26	78%
Sun UltraSPARC	18.58	13.27	71%
PowerMac G5	9.56	8.75	92%

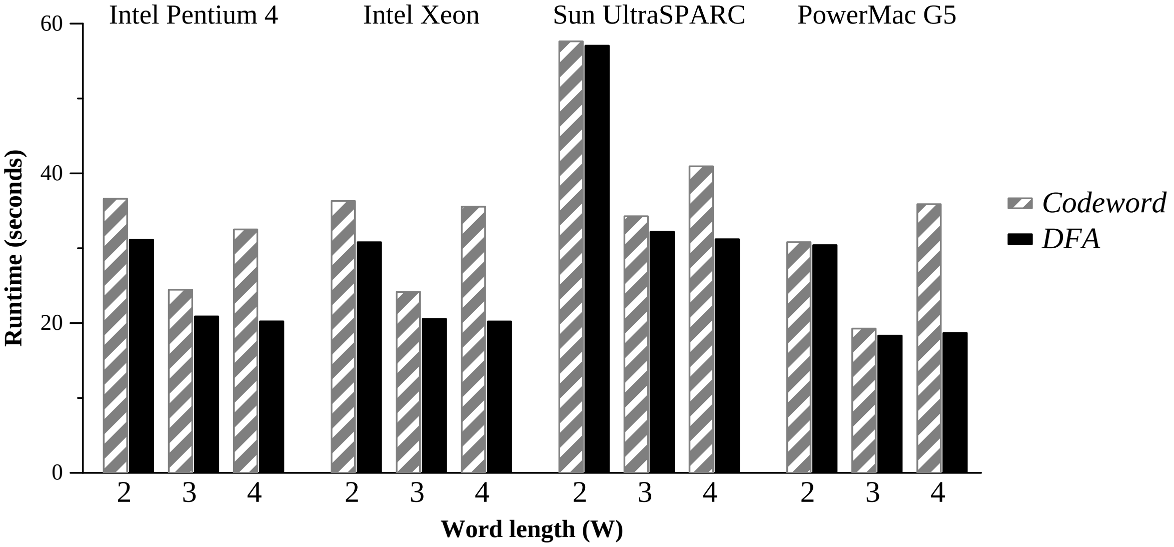

Figure 7: *A comparison of BLAST total search times for table-based codeword lookup and optimised deterministic finite automaton (DFA) designs. Experiments were conducted using W and T parameters pairs with similar accuracy on four different hardware architectures.*

Figure 7 shows a comparison of the optimised table-based and DFA schemes using our own implementation of BLAST. These results show overall BLAST search times for each of the W and T parameters pairs with similar accuracy, except for $W = 5$ and $T = 15$ which did not produce a runtime below 50 seconds on any of the architectures tested. The DFA is significantly faster: for the default $W = 3$ setting, it results in 10% faster average runtimes, and is around 15% faster on the commonly-used Intel platforms; this is equivalent to a 41% speedup in the first stage of BLAST. Importantly, because the compact DFA structure caches effectively, it is practical for $W = 4$, where the elapsed query times are almost identical to $W = 3$ and 35%–50% faster than the table-based scheme for the same setting; the DFA is therefore a practical structure for $W = 4$, a parameter disabled by default in NCBI-BLAST.

Figure 8 shows a comparison between the one hit and two hit modes of operation using our new implementation of BLAST and our optimised DFA structure. The results show overall BLAST search times for values of W and T with comparable accuracy. For one hit mode we used parameters $W = 3$, $T = 13$ which result in a ROC_{50} score of 0.384, and parameters $W = 4$, $T = 15$ which result in a ROC_{50} score of 0.383. Results for values of $W = 2$ and $W = 5$ are not shown because they did

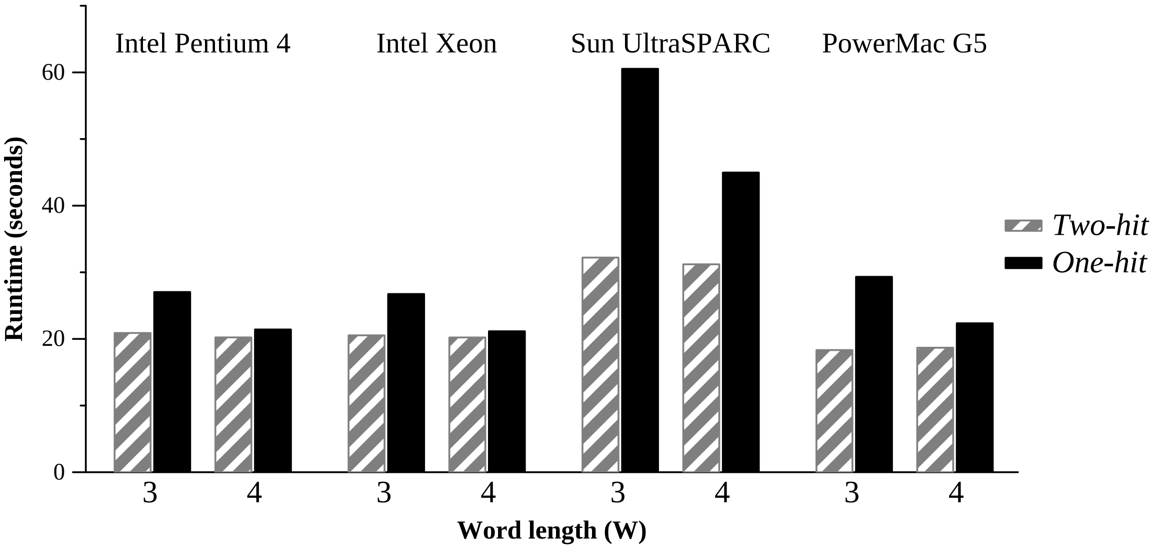

Figure 8: *A comparison of BLAST total search times for one hit and two hit modes of operation, using our implementation of BLAST and the optimised DFA structure. Experiments were conducted using W and T parameters pairs with similar accuracy on four different hardware architectures.*

Table 5: *Runtime comparison between NCBI-BLAST and FSA-BLAST using default parameters. The percentage of NCBI-BLAST runtime is also shown.*

	NCBI-BLAST	FSA-BLAST	
	(secs)	(secs)	(%)
Intel Pentium 4	30.58	20.89	68%
Intel Xeon	30.22	20.54	68%
Sun UltraSPARC-IIIi	46.42	32.21	69%
PowerMac G5	22.83	18.33	80%

not produce runtimes below 60 seconds for one hit mode of operation on any of the architectures tested. The results confirm that BLAST is faster when run in two hit mode, in agreement with the 1997 BLAST paper (Altschul et al., 1997). However, we note that for $W = 4$ the speed difference between one hit and two hit is significantly smaller than for $W = 3$.

Table 5 shows an overall comparison between NCBI-BLAST and FSA-BLAST: our own implementation of the BLAST algorithm. FSA-BLAST uses the new optimised DFA and our improvements to the gapped alignment stages of BLAST described previously (Cameron et al., 2004). Our implementation is around 30% faster on Intel and Sun platforms, and 20% faster on the Apple PowerMac G5. Around 15% of this speedup can be attributed to our improvements to the gapped alignment stages, while the remainder is due to the work described in this paper. Importantly, there is no significant effect on accuracy: the ROC$_{50}$ score for NCBI-BLAST is 0.379 compared to 0.380 for our implementation.

5 Conclusion

We have explained, optimised, and proposed structures for the first, hit detection phase in BLAST with the aim of improving overall BLAST runtimes for protein search. Hit detection matches words extracted from subject sequences against a structure derived from the query sequence. To do this, NCBI-BLAST uses a table-based lookup approach, a structure we have investigated and optimised for our experiments.

We have proposed using a deterministic finite automaton (DFA) for fast, cache-conscious matching. Our scheme is optimised based on properties of the matching process and designed to make effective use of modern hardware. Our experiments show this approach works well: it typically improves overall BLAST search times by around 15% compared to our implementation of the NCBI-BLAST approach; our implementation of the table-based scheme is in turn around 20% faster than the NCBI implementation. Further, our scheme is practical for BLAST searches with a word length of four, a parameter value not supported by NCBI-BLAST. Our improvements to hit detection can also be applied to other variants of BLAST, including PSI-BLAST and BLASTX, and to other protein search tools that use word matches to trigger alignment.

We have integrated the deterministic finite automaton into a new open-source version of BLAST, available for download at `http://www.fsa-blast.org/`. The new tool, called FSA-BLAST, is 20-30% faster than NCBI-BLAST for protein searches with no significant effect on accuracy.

We are currently investigating other optimisations to BLAST. These include clustering of highly-similar collection sequences to further improve BLAST search speed and optimisations specifically for the relatively-unoptimised nucleotide search process.

Acknowledgements

This work was supported by the Australian Research Council. We thank Isaac Balbin for access to his Apple PowerMac.

References

Altschul, S., Gish, W., Miller, W., Myers, E., and Lipman, D. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410.

Altschul, S., Madden, T., Schaffer, A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D. (1997). Gapped BLAST and PSI–BLAST: A new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402.

Brenner, S., Chothia, C., and Hubbard, T. (1998). Assessing sequence comparison methods with reliable structurally identified distant evolutionary relationships. *Proceedings of the National Academy of Sciences USA*, 95(11):6073–6078.

Brown, D. (2005). Optimizing multiple seeds for protein homology search. *IEEE Transactions on Computational Biology and Bioinformatics*, 2(1):29–38.

Cameron, M., Williams, H. E., and Cannane, A. (2004). Improved gapped alignment in BLAST. *IEEE Transactions on Computational Biology and Bioinformatics*, 1(3):116–129.

Chandonia, J., Hon, G., Walker, N., Conte, L. L., Koehl, P., Levitt, M., and Brenner, S. (2004). The ASTRAL compendium in 2004. *Nucleic Acids Research*, 32:D189–D192.

Chen, Z. (2003). Assessing sequence comparison methods with the average precision criterion. *Bioinformatics*, 19(18):2456–2460.

Gotea, V., Veeramachaneni, V., and Makalowski, W. (2003). Mastering seeds for genomic size nucleotide BLAST searches. *Nucleic Acids Research*, 31(23):6935–6941.

Kent, W. (2002). BLAT–the BLAST-like alignment tool. *Genome Research*, 12(4):656–664.

Li, M., Ma, B., Kisman, D., and Tromp, J. (2004). PatternHunter II: Highly sensitive and fast homology search. *Journal of Bioinformatics and Computational Biology*, 2(3):417–439.

Ma, B., Tromp, J., and Li, M. (2002). PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445.

McGinnis, S. and Madden, T. (2004). Blast: at the core of a powerful and diverse set of sequence analysis tools. *Nucleic Acids Research*, 32:W20–W25.

Park, J., Karplus, K., Barrett, C., Hughey, R., Haussler, D., Hubbard, T., and Chothia, C. (1998). Sequence comparisons using multiple sequences detect three times as many remote homologues as pairwise methods. *Journal of Molecular Biology*, 284(4):1201–1210.

Robinson, A. and Robinson, L. (1991). Distribution of glutamine and asparagine residues and their near neighbors in peptides and proteins. *Proceedings of the National Academy of Sciences USA*, 88(20):8880–8884.

Schaffer, A., Wolf, Y., Ponting, C., Koonin, E., Aravind, L., and Altschul, S. (1999). IMPALA: matching a protein sequence against a collection of psi-blast-constructed position-specific score matrices. *Bioinformatics*, 15(12):1000–1011.

Schwartz, S., Kent, W. J., Smit, A., Zhang, Z., Baertsch, R., Hardison, R. C., Haussler, D., and Miller, W. (2002). Human-mouse alignments with BLASTZ. *Genome Research*, 13(1):103–107.

Sudkamp, T. A. (1997). *Languages and machines: an introduction to the theory of computer science*. Addison-Wesley Longman Publishing Co., Inc.

Wilbur, W. and Lipman, D. (1983). Rapid similarity searches of nucleic acid and protein data banks. *Proceedings of the National Academy of Sciences USA*, 80(3):726–730.

Williams, H. E. and Zobel, J. (1997). Compression of nucleotide databases for fast searching. *Computer Applications in the Biosciences*, 13(5):549–554.

Wootton, J. and Federhen, S. (1993). Statistics of local complexity in amino acid sequences and sequence databases. *Computers in Chemistry*, 17:149–163.

Zhang, Z., Schaffer, A., Miller, W., Madden, T., Lipman, D., Koonin, E., and Altschul, S. (1998). Protein sequence similarity searches using patterns as seeds. *Nucleic Acids Research*, 26(17):3986–3990.