Comparing Compressed Sequences for Faster Nucleotide BLAST Searches

Michael Cameron[1], Hugh E. Williams[2]

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

[2]Microsoft Corporation
One Microsoft Way, Redmond, Washington 98052, USA
hughw@microsoft.com

Abstract. Molecular biologists, geneticists, and other life scientists use the BLAST homology search package as their first step for discovery of information about unknown or poorly annotated genomic sequences. There are two main variants of BLAST: BLASTP for searching protein collections and BLASTN for nucleotide collections. Surprisingly, BLASTN has had very little attention; for example, the algorithms it uses do not follow those described in the 1997 BLAST paper (Altschul, Madden, Schaffer, Zhang, Zhang, Miller & Lipman 1997) and no exact description has been published. It is important that BLASTN is state-of-the-art: nucleotide collections such as GenBank dwarf the protein collections in size, they double in size almost yearly, and take many minutes to search on modern general-purpose workstations. This paper proposes significant improvements to the BLASTN algorithms. Each of our schemes is based on compressed *bytepacked* formats that allow queries and collection sequences to be compared four bases at a time, permitting very fast query evaluation using lookup tables and numeric comparisons. Our most significant innovations are two new, fast gapped alignment schemes that allow accurate sequence alignment without decompression of the collection sequences. Overall, our innovations more than double the speed of BLASTN with no effect on accuracy and have been integrated into our new version of BLAST that is freely available for download from http://www.fsa-blast.org/.

1 Introduction

The GenBank collection (Benson, Karsch-Mizrachi, Lipman, Ostell & Wheeler 2004) records information on over 59 billion nucleotide base pairs stored in around 54 million sequences[1]. Further, the collection doubles in size around every 16 months, and has roughly done so since the early 1980s[2]. Even downloading a copy requires significant resources: it is divided into over 900 files — each typically more than 250 megabytes in size — that require around 200 gigabytes of disk space to store in their uncompressed format. Given the scale of the collection and the well-known fact that homology search is computationally intensive, it is unsurprising that fast, accurate search is a difficult task.

The tool in most widespread use for nucleotide homology search is BLASTN, a member of the BLAST tool family (Altschul, Gish, Miller, Myers & Lipman 1990, Altschul et al. 1997). Similarly to the other BLAST tools, BLASTN is an exhaustive search tool: for each query, it compares that query to each sequence in the collection. The comparison is multi-stage, where the first step is necessarily computationally efficient and the later, slower stages are only executed when the similarity between the query and a collection sequence exceeds a threshold. On modern desktop hardware, a BLASTN search of the entire GenBank collection takes in the order of several minutes for a typical sequence query (Chen 2004). Perhaps because of this, the NCBI website[3] no longer supports complete BLASTN searches of the entire GenBank collection. Instead, a concise version of the collection — roughly one quarter of the size — that does not include "high-throughput, patent, genomic or sequence tagged cite (STS) sequences" is available for search (McGinnis & Madden 2004).

Other work has focused on improving the speed and scalability of nucleotide search. For example, Williams and Zobel have proposed and described refinements to the CAFE indexed-based homology search tool (2002). Like many schemes before and after it, it avoids exhaustive search through using a disk-based inverted index, adapted from text Information Retrieval. However, despite its promise, CAFE and other non-exhaustive methods are not in widespread use because of impractical query evaluation on very large collections.

More recently, several nucleotide search tools have been proposed that use *spaced seeds* to perform hit detection, including MEGABLAST (Zhang, Schwartz, Wagner & Miller 2000) and PATTERNHUNTER (Ma, Tromp & Li 2002, Li, Ma, Kisman & Tromp 2004). These schemes 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 (we refer the reader to Brown et al. (2004) for a detailed description of spaced seeds and other seeding variants). Spaced seed schemes have been shown to provide better sensitivity and faster search times for some types of nucleotide search, but they are not replacements for BLASTN because they are either impractical for searching large collections such as GenBank or are less sensitive than BLASTN. Therefore, BLASTN remains the only practical tool for accurately searching any significant fraction of the GenBank collection.

In this paper, we propose innovations in BLASTN searching. Each of our schemes is based on the simple, practical compression scheme proposed by Williams and Zobel (1997). In their approach, each of the four nucleotide bases is stored as a two-bit binary value, permitting four bases to be stored per byte; they store wildcards in a separate structure and optionally restore them before sequence comparison. This compression scheme allows for very fast searching: compressed sequences are faster to read from disk than uncompressed sequences, they require less main-memory to store, and they can be compared without decompression. This latter point is important and unique to our work: our innovations allow a compressed query sequence to be compared to a compressed collection sequence in each of the first three stages of the BLAST algorithm.

Our work on the hit detection and ungapped hit extension stages focuses largely on re-engineering the BLASTN approach. We propose practical improvements to the NCBI BLASTN algorithms and data structures, and show that our compression based schemes improve search times significantly without affecting accuracy. For the first stage — where exact subsequences, typically of length $N = 11$, are identified between the query and each collection sequence — our approach reduces search times by almost 50%. For the second stage — where hits are extended using an ungapped alignment algorithm that does not permit insertions or deletions — our approach reduces alignment times by around 43%. Together, since both stages consume around 90% of the entire search process, our schemes reduce total BLASTN search times by around 43%.

[1] GenBank Release 152 (February 2006). Available from `ftp://ftp.ncbi.nih.gov/genbank/`

[2] See: `http://www.ncbi.nih.gov/Genbank/genbankstats.html`

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

Our major innovations in this paper are two novel approaches to gapped alignment. The first scheme — which we refer to as *bytepacked alignment* — allows the computation of alignment scores between a compressed query sequence and a compressed collection sequence. To do this, we compress the query sequence into four compressed representations, one for each possible position of a two-bit nucleotide code in a byte. We are then able to compute alignments between two sequences based on an ungapped extension from the previous stage, beginning at any offset without decompression. The bytepacked alignment method is heuristic because each collection sequence is compressed only once, requiring that insertions and deletions only occur at one in four offset positions. (However, insertions and deletions are permitted anywhere in the query because it is compressed into four representations.) Regular gapped alignment is then performed for high-scoring alignments using uncompressed sequences to compute the optimal alignment. Overall, with very little modification to the underlying BLASTN parameters, our bytepacked alignment scheme reduces the time taken to align sequences by around 78% when compared to the NCBI BLASTN gapped alignment stage. Importantly, there is no significant difference in accuracy.

The second scheme — which we refer to as *table-driven alignment* — aligns sequences using a novel variant of the Four Russians (Wu, Manber & Myers 1996) approach. Using this approach, we are able to compare the query to sequences in the collection without decompression. When applied to BLAST, our table-driven alignment approach reduces the time taken to perform gapped alignment by 72%, and unlike bytepacked alignment, the approach is guaranteed to find the optimal alignment between two sequences.

Overall, our improvements to the hit detection, ungapped alignment, and gapped alignment stages more than double the speed of BLASTN with no significant effect on accuracy. Further, our improvements can be applied to other tools that use a two bits per base representation of nucleotide sequences.

This paper is structured as follows. We provide an overview of the BLAST algorithm and other approaches to homology search in Section 2. In Section 3, we describe our approaches for comparing compressed sequences. We present accuracy and performance results for our approach in Section 4. Finally, we provide concluding remarks in Section 5.

2 Background

In this section, we describe the stages of the BLASTN algorithm as implemented in a recent release of the NCBI-BLAST software[4]. In addition, we briefly describe other approaches to nucleotide search.

2.1 Homology Search

There are two distinct types of homology search. Amino-acid or protein search is preferred by biologists: the sequence collections are well-annotated, the collections are small and targeted, and rich tools and scoring schemes are available for the search process. Nucleotide search is less preferable, but often used: very large databases are available, many queries are from non-coding regions (and so do not have a protein equivalent), genome to genome comparisons are beginning to yield interesting results, and the queries return much broader result sets. Based on statistics provided by the US National Centre for Biotechnology Information (NCBI) for queries submitted on September 22, 2005, around 57% of searches conducted by users were BLASTN nucleotide-nucleotide searches.

To search GenBank and other nucleotide repositories, most users make use of the BLASTN search tool, a member of the BLAST family of homology search techniques (Altschul et al. 1997, Altschul et al. 1990); we discuss the BLASTN tool in detail in the next section. However, other tools have been proposed for nucleotide search and have found favour with small numbers of researchers. This section briefly describes a selection of these other tools.

When sufficient computing resources are available, the preference of all users is to use exhaustive Smith-Waterman local alignment (1981) to compare a query sequence to sequences in a nucleotide collection. The most popular implementation of this approach is SSEARCH, a space-efficient variant (Myers & Miller 1988) distributed with the FASTA homology search tools (Pearson & Lipman 1988). Smith-Waterman alignment is optimal and exhaustive, that is, it guarantees finding all optimal alignments with respect to a scoring

[4] BLAST version 2.2.10, 2005.

scheme. However, while it is accurate, it is not fast: on a general-purpose workstation, a search of GenBank would typically take around three days for a query that is only one thousand basepairs in length.

Most searchers make use of heuristic approaches to homology search. The first, well-known and widely-used such tool was FASTP (Pearson & Lipman 1985), which was later improved and re-released as FASTA (Pearson & Lipman 1988). The FASTA approach provides a framework for the approaches taken by all heuristic tools: for each query, a multi-step approach is taken in comparing it to each sequence in a sequence collection. First, an exact match of between $n = 4$ and $n = 6$ characters is required before a collection sequence is considered in further stages. Then, in later stages, alignments are rescored and successive stages are only applied when scores exceed a threshold. If a sequence passes through the first three stages, a banded Smith-Waterman alignment (Chao, Pearson & Miller 1992) is performed in the fourth stage and the result saved for possible display to the user. In practice, FASTA is 6.5 times faster than SSEARCH and almost as accurate (Brenner, Chothia & Hubbard 1998).

In 1996, Williams and Zobel proposed the CAFE index-based homology search tool (Williams & Zobel 1996, Williams & Zobel 2002). This scheme uses inverted indexing techniques employed in text retrieval (Witten, Moffat & Bell 1999), which are most well-known now for their use in web search engines. The significant difference between it and FASTA is in the first stage: it uses a large, disk-based structure to identify sequences that share similarity with a query, rather than exhaustively processing each sequence in response to the query. Results showed that CAFE was around eight times faster than BLASTN and over eighty times faster than FASTA (Williams & Zobel 2002). However, despite its promise, CAFE has not found widespread acceptance because of its high main-memory requirements for query evaluation. There have been other, similar index-based approaches attempted for homology search, such as FLASH (Califano & Rigoutsos 1993), SCAN (Orcutt & Barker 1984), and RAMDB (Fondrat & Dessen 1995). Similarly to CAFE, none have found widespread acceptance, probably for similar reasons.

More recently, other classes of heuristics have been considered for exhaustive nucleotide search. The popular MEGABLAST tool is now publically-accessible for online GenBank searches through the NCBI website[5]. It uses an efficient greedy alignment algorithm (Zhang et al. 2000), a longer word length than typical tools, and compares multiple query sequences to each collection sequence in a single scan of the collection (thereby reducing the costs when multiple queries are simultaneously posed by one or more users). It is not designed as a replacement for FASTA or BLASTN, but is instead designed for quickly finding very similar sequences. Specifically, its default word length is $n = 28$, meaning that the minimum number of contiguous identical bases is 28 before any subsequent processing stage is attempted.

The BLAT scheme (Kent 2002) is designed for nucleotide search, but for specific applications. It shares similarities with the index-based approaches we described previously, but constructs a main-memory search structure that limits its application to a class of small database search problems. Specifically, BLAT is designed for nucleotide alignments between messenger RNA (mRNA) and genomic DNA from the same species, and is staggeringly fast for the task of finding near identical matches between longer queries and whole genomes. While the BLAT scheme has been tested on whole genomes — which are typically one to two orders of magnitude smaller than GenBank — it could not be applied on current desktop hardware to larger scale search problems.

The PATTERNHUNTER approaches (Ma et al. 2002, Li et al. 2004) take a novel approach to matching in the first stage. Rather than requiring exact, contiguous matches of length n, they require matches between n bases within a window of length m, where $m > n$. In addition, the n matching bases must follow a template, that is, the offset of each of the n matches in the window of size m is important. However, similarly to BLAT, PATTERNHUNTER relies on a main-memory index of the collection and so is impractical for very large scale database search; the results presented in Ma et al. (2002) are for searches of collections in the order of a few tens of megabytes in size, using query sequences of similar size. The PATTERNHUNTER approach is patented and no openly-configurable versions are available.

2.2 BLASTN

The BLAST tools are ubiquitous. Since their emergence in 1990 (Altschul et al. 1990), they have been the key software tool for molecular biologists, geneticists, and other life scientists. The tools in use in 2006 are most

[5] See: `http://www.ncbi.nml.nih.gov/BLAST/`

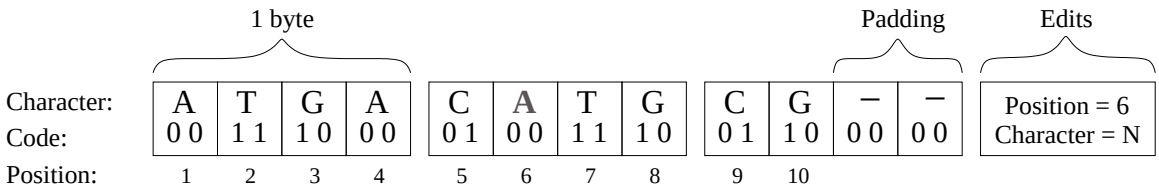

Fig. 1. *Bytepacked representation of the sequence* ATGACNTGCG*. The 6th character is a wildcard, therefore a randomly selected base (in this case A) is recored in its place, and a wildcard edit is stored at the end of the sequence.*

closely related to those described in the second major BLAST publication (Altschul et al. 1997), and include BLASTP (for protein search), BLASTN (for nucleotide search), and PSI-BLAST (for iterative, repeated search for distant homologs). While the former two tools are accessible through a common executable program, BLASTALL, the underlying source code and algorithms remain separate. This section describes the BLASTN algorithm implemented in the current release of BLASTALL; interestingly, this differs substantially from the description published in 1997 that focuses on improved protein search and, in fact, closely resembles the original 1990 approach.

The BLASTN algorithm is a four-stage process. Similarly to the FASTA scheme described in the previous section, it is exhaustive and multi-step, that is, the steps refine the number of matching sequences and each takes longer than the previous to process each sequence. This section describes the steps. However, before we begin this discussion, we describe the bytepacked representation used by BLAST to compress sequences in the collection.

Bytepacked representation

Before searching a collection with BLAST, the user is required to convert it to a new format using the *formatdb* tool provided with NCBI-BLAST. The *formatdb* tool converts a collection of nucleotide sequences from uncompressed FASTA format to a specially formatted file that encodes the sequences using bytepacked compression; sequences are stored using two bits per base, that is, each byte represents four nucleotide bases. This approach was shown by Williams and Zobel (1997) to yield storage space savings and, importantly, reduce retrieval times from disk when processing sequences.

Bytewise compression is an important innovation in BLASTN. A diagram illustrating the approach further is shown in Figure 1. The figure shows three bytes that store a sequence of ten bases in length: the first byte stores four two-bit codes for `atga`, the second four codes for `catg`, and the third two codes for `cg`. The sequence length is stored separately, so decoding of the final byte is unambiguous. Note that the original sequence includes a wildcard character N at position 6, which is replaced in the compressed representation by a random choice of the nucleotides represented by N. The additional structure shown to the right of the figure can optionally be decoded to restore the original sequence. (The frequency of wildcards in collections such as GenBank is extremely low: over 99% of character occurrences are one of the four nucleotide bases, and almost 98% of the wildcard occurrences are N (Williams & Zobel 1997).) In 1997, it was shown that retrieval times are reduced by over 70% when sequences are stored using the compressed representation rather than uncompressed regardless of whether wildcards are decoded (Williams & Zobel 1997). In addition to the retrieval benefits, as we have discussed, the compressed representation permits four bases to be compared through a single comparison of two bytes; an approach currently used in part for the first stage of BLAST, but not in the remaining stages. We describe the individual stages of the BLAST algorithm next.

Stage 1

The first stage of BLAST is executed for all sequences in the target collection, and identifies all matching substrings or *hits* of length W — where $W = 11$ is the default for nucleotide search — between the query sequence and each sequence in the collection. To identify matches quickly, two preliminary steps are performed for each query sequence. First, the query sequence is parsed into fixed-length overlapping subsequences or

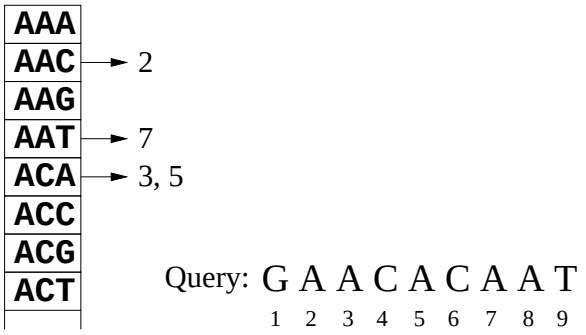

Fig. 2. *Part of the lookup structure used by* BLAST *to detect matches between the query and collection sequences. In this example* $W = 3$ *and the query sequence is* `gaacacaat`. *The first 8 of a total of* $4^3 = 64$ *entries in the lookup table are shown.*

words of length W. For example, suppose a word length of $W = 3$ and a short query of `gaacacaat`. When parsed, it has the following subsequences: `gaa`, `aac`, `aca`, `cac`, `aca`, `caa`, and `aat`. Second, a fast lookup structure is constructed that contains a^W entries, where $a = 4$ is the alphabet size for nucleotide excluding wildcards, with one entry for each possible subsequence of length W. For each subsequence in the query, its location is stored in the lookup structure at the corresponding entry. A partial lookup structure for the query `gaacacaat` is illustrated in Figure 2. The choice of lookup structure requires care and we have recently described what we believe is the best choice for protein search (Cameron, Williams & Cannane 2006). We discuss the lookup structure used for nucleotide search in more detail in Section 3.1.

Once the lookup structure has been created it is used to search the collection for hits between the query and collection sequences. To do this, each collection sequence is retrieved, and parsed into fixed-length overlapping subsequences of length n using the same process as the query. Each collection subsequence is searched for in the query table, and zero or more exact matches to query hits are identified. Finally, the position in the query and position in the collection sequence of the matching subsequences are passed on to Stage 2 of BLASTN for subsequent processing. The first stage process is illustrated to the left of Figure 3, where hits between a collection sequence and query are shown as short, dark lines. In this example, there are three hits between the query and collection sequences. The lookup and matching process we have described follows the technique of Wilbur and Lipman (1983).

The algorithm used by NCBI-BLAST for nucleotide search that we have just described is not exactly the same as described in the 1997 manuscript (Altschul et al. 1997). In particular, only one hit between a query and collection sequence triggers a second stage, ungapped alignment; the published description describes the BLASTP approach that requires two matches. This suggests that the two-hit mode of operation improves search times for protein searches only (see Cameron et al. (2006) for a more detailed comparison of one-hit and two-hit modes of BLASTP). In addition, neighbourhood words are not used for nucleotide search, that is, each hit must be an exact match between a query and collection subsequence; this is in contrast to both the 1990 and 1997 descriptions of the approach.

The algorithm we have described so far omits subtle, but important, details of the NCBI-BLAST implementation. In particular, a modified approach is used to search collections when sequences are compressed using the bytepacked representation. Instead of extracting each overlapping subsequence of length W from sequences in the collection — which would involve decompressing the sequences — BLASTN extracts every fourth subsequence of length n where n is the largest value such that $n \equiv 0, \text{modulo}\, 4$ and $n \leq W - 3$. For the default value of $W = 11$, $n = 8$ and so BLASTN extracts pairs of adjacent bytes from the collection. The query lookup structure is therefore for subsequences of length $n = 8$, and not length $W = 11$. When an initial match of length $n = 8$ is found between a collection sequence and query, the original sequences are inspected $W - n$ bases in both directions surrounding the match to identify if a contiguous match of length W exists and, if this is the case, a hit is detected. This approach, which is illustrated in Figure 4, has two speed advantages: it allows two bytes to be compared in compressed form (that is, two collection sequence bytes do not need to be decompressed), and it permits whole bytes to be retrieved from the collection rather than fractions of bytes. In practice, $W = 7, 11, 15$ are the optimal choices to take advantage of this bytewise

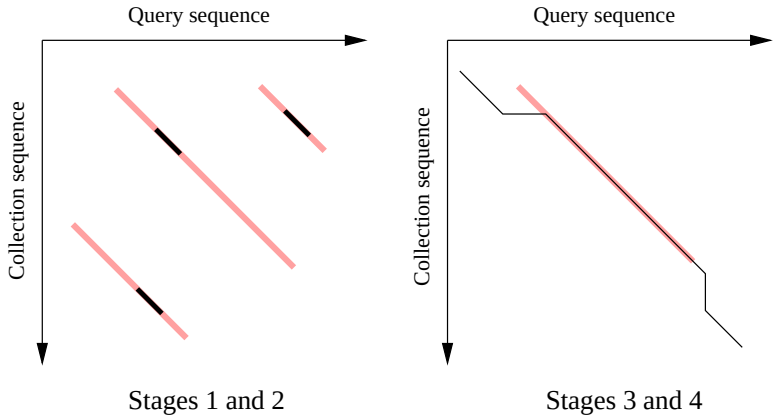

Fig. 3. *The four stages of the* BLAST *algorithm. In this example, three hits are identified, where each hit is illustrated as a short black line. Each hit triggers an ungapped alignment that is illustrated as a longer, grey line. Finally, one of the ungapped alignments scores above a threshold and triggers a gapped alignment that is illustrated as a thin, black line.*

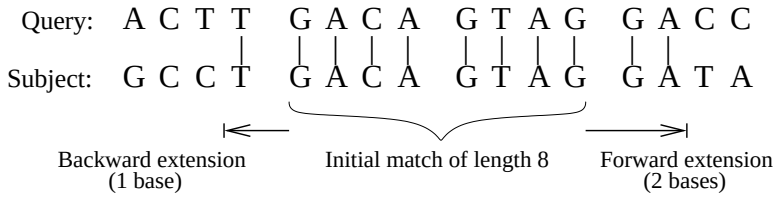

Fig. 4. *The* NCBI-BLAST *approach to detecting hits of length* $W = 11$*. A byte aligned initial match of length* $n = 8$ *is identified first, then the query and collection sequences are inspected each side of the match to check if it forms part of a longer hit of length* $W = 11$*.*

approach, since each must span at least 1, 2, or 3 bytes respectively. Since wildcard characters are not present in the bytepacked sequence, the randomly selected basepair that is recorded in its place is processed instead. This is unlikely to have any significant effect on search accuracy, since wildcard characters typically occur very infrequently in collections.

Another speed related optimisation used by NCBI-BLAST is the use of an auxiliary lookup table that contains one bit for each possible subsequence; for example, for $n = 8$ there are $4^8 = 65,536$ bits in this table. The single bit indicates, for each possible collection subsequence, if it occurs in the query. Hit detection does not proceed directly to the query lookup table. Instead, after a collection subsequence of length $n = 8$ is read, it is searched for in the auxiliary lookup table. If a one bit is found, there is at least one matching offset for this subsequence in the query and the main lookup structure is consulted to find the query positions. If a zero bit is found, no match exists in the query and the next collection subsequence is read. The auxiliary lookup table is considerably smaller than the main lookup structure and is fast to access because it is compact and can be cached at the CPU. However, it is unclear if the extra computation required to perform a second lookup for some words is worthwhile and we discuss this further in Section 3.1.

To measure the percentage of time spent by BLAST performing the first stage of nucleotide search we conducted 50 searches using the test collection and randomly-selected queries described in Section 4.1, and a recent version of NCBI-BLAST[6]. The results of this experiment are shown in Table 1. We found that the first stage consumes on average 85% of the total search time and that an average of 3.5 hits were found per collection sequence; this contrasts with protein search where the first stage consumes 37% of search time and on average 229 hits are found per collection sequence (Cameron, Williams & Cannane 2004).

[6] BLAST version 2.2.10, 2005.

Stage	Task	Average number per sequence	Percentage overall time
1	Identify short, high-scoring matches	3.5	85%
2	Perform ungapped alignment	3.1	5%
3	Perform gapped alignment	0.07	9%
4	Perform gapped alignment with traceback	0.0001	1%

Table 1. Performance characteristics of each stage of BLASTN. Results were measured by searching 50 randomly selected queries against a portion of the GenBank non-redundant database using NCBI-BLAST.

Stage 2

In the second stage, hits detected in the first stage are considered as the basis of ungapped alignments, that is, sequences are aligned without considering insertion or deletion events. The left side of Figure 3 illustrates this step: short hits identified in the first stage (shown as dark lines) are extended in this stage to identify longer regions of alignment between a collection sequence and query sequence.

The ungapped extension process is conceptually straightforward. Starting at the match of length W between the two sequences, ungapped extension sequentially considers bases from each sequence that are adjacent to the hit, beginning by proceeding in one direction. The total score increases when bases are identical, and decreases when they are not. Extension is abandoned when the total score decreases by more than a threshold, and the highest scoring point of the extension is recorded. The extension process is repeated in the other direction.

Since collection sequences are stored in bytepacked form, they must be decompressed during the ungapped extension process. Instead of decompressing the entire collection sequence, NCBI-BLAST extracts each nucleotide base from the sequence on demand, that is, when the base is to be compared to the query sequence during the ungapped extension process.

We measured the performance of the second stage using the same 50 randomly-selected queries and collection used in the previous section. On average, the second stage consumes 5% of the total search time and 3.1 ungapped extensions are performed per sequence; not all hits trigger an ungapped extension because, in some cases, multiple hits are contained within a single ungapped alignment.

Stage 3

In the third stage, BLASTN uses a dynamic programming algorithm to find gapped alignments that augment the high-scoring ungapped alignments identified in the second stage. To do this, a seed point is chosen partway along the ungapped alignment and a gapped extension is performed in both the forwards and backwards direction from that seed. Using this approach, only gapped alignments that pass through the seed are considered; this reduces the area of the dynamic programming matrix that needs to be processed.

BLASTN uses affine gap costs to score alignments; the cost c of a gap of length k is defined as $c(k) = k \times e + o$ where o is the cost of opening a gap, e is the cost of extending a gap and $e > 0$, $o > 0$, $k > 0$. We define d as the cost of opening a gap plus the first insertion in that gap, that is $d = o + e$. BLASTN uses the popular dynamic programming algorithm by Gotoh (1982) to find gapped alignments using affine gap costs. The algorithm uses a matrix of size $|q| \times |s|$ where each cell $[i, j]$ in the matrix represents the highest scoring alignment between q and s ending at the i^{th} character in q and the j^{th} character in s. At each cell in the matrix three operations are considered; the bases q_i and s_j are aligned (either they are identical and increase the alignment score, or differ and decrease the score), an insertion is made in q, or an insertion is made in s. Therefore, each cell $[i, j]$ is dependent on three immediate neighbours at positions $[i-1, j-1]$, $[i, j-1]$ and $[i-1, j]$. This dependence is illustrated in Figure 5 where diagonal arrows represent the alignment of two bases, vertical arrows represent an insertion with respect to q, and horizontal arrows represent an insertion with respect to s.

The Gotoh algorithm records three values at each cell in the matrix; $B(i, j)$ is the best score for an alignment ending at $[i, j]$, $I_q(i, j)$ is the best score for an alignment ending at $[i, j]$ with an insertion in q

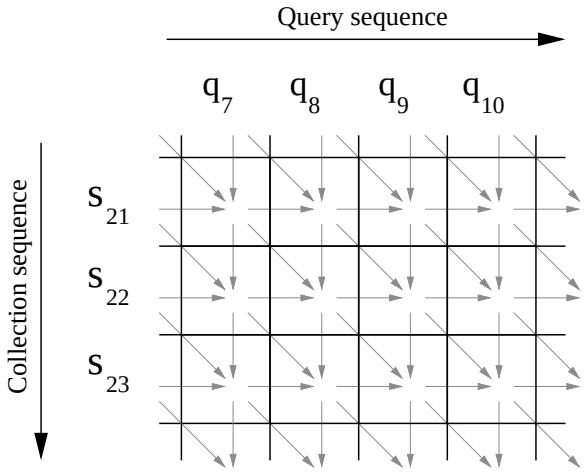

Fig. 5. *Illustration of the alignment matrix used by* BLASTN *to perform gapped alignment. The values at each cell* $[i, j]$ *in the matrix are dependent on three immediate neighbours at coordinates* $[i-1, j-1]$, $[i, j-1]$ *and* $[i-1, j]$. *Diagonal arrows represent the alignment of a query and collection sequence base, and horizontal and vertical arrows represent insertion in the collection sequence and query.*

and $I_s(i, j)$ is the best score for an alignment ending at $[i, j]$ with an insertion in s. Using these values, the following recurrence relations are employed to compute the score of the optimal alignment between q and s:

$$M(i,j) = B(i-1, j-1) + s(q_i, s_j)$$

$$B(i,j) = \max \begin{cases} I_q(i-1,j) \\ I_s(i,j-1) \\ M(i,j) \end{cases}$$

$$I_q(i,j) = \max \begin{cases} M(i,j) - d \\ I_q(i-1,j) - e \end{cases}$$

$$I_s(i,j) = \max \begin{cases} M(i,j) - d \\ I_s(i,j-1) - e \end{cases}$$

where $s(q_i, s_j)$ is the score resulting from the alignment of the i^{th} base in q and the j^{th} base in s, and $M(i, j)$ is a scalar value that represents the best score for an alignment ending at $[i, j]$ with a match. For nucleotide sequence alignments a positive score $r > 0$ results from the alignment of matching bases, that is $s(q_i, s_j) = r \mid q_i = s_j$ and a negative score $p < 0$ results from the alignment of mismatching bases, $s(q_i, s_j) = p \mid q_i \neq s_j$. Boundary conditions are used to initialize the matrix; all cells where $i = 0$ or $j = 0$ are initialized to $-\infty$ except for the alignment starting point $[0, 0]$ which is initialized to 0. These recurrence relations are suitable for finding the best score for an alignment that starts at a given position in the query and collection sequences; for global alignment this is the beginning of the sequences and for BLASTN gapped alignments this is the seed point; a point chosen partway along the ungapped extension that triggered the gapped alignment. The algorithm we have described calculates the gapped alignment score only; additional data structures are required to record the optimal alignment. The algorithm requires $O(|q||s|)$ time and $O(|q|)$ space; only the previous row in the matrix needs to be retained to compute the next row.

The recurrence relations we have described are those used by NCBI-BLAST. In previous work (Cameron et al. 2004), we showed that a rearrangement of the relations can be used to reduce computation without affecting results and obtain a speedup of around 20% for protein searches. In the same paper, we describe an approach called *restricted insertion* that does not permit adjacent gaps, where a gap in the query sequence is immediately followed by a gap in the collection sequence or vice-versa, and results in an 8% speed up to gapped alignment. We have used both the improved rearrangement and the restricted insertion approach for our implementation of BLASTN.

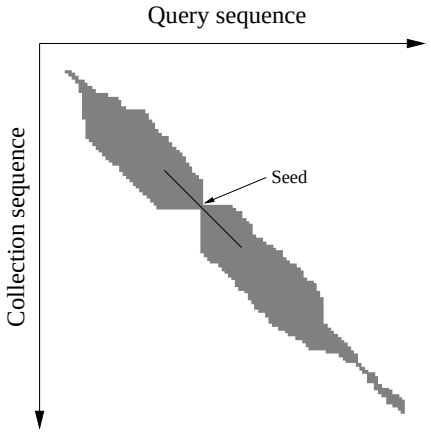

Fig. 6. *Illustration of* BLAST *gapped alignment using the dropoff technique. The short black line represents a high-scoring ungapped alignment between the query and collection sequences. The grey area represents the region of the alignment matrix that is processed when gapped alignment is performed in each direction from an initial seed point.*

BLASTN uses a dropoff heuristic (Zhang, Berman & Miller 1998) to limit the region of the alignment matrix processed during gapped alignment. During alignment, it only considers cells in the matrix with a best score $B(i, j)$ that is greater than the best alignment score observed so far minus the value of the dropoff parameter, X. Figure 6 illustrates how this approach limits the region of the alignment matrix processed during gapped alignment. The approach is highly effective at reducing the number of cells in the matrix that are processed without reducing sensitivity to high-scoring alignments.

To perform the gapped alignment on collection sequences, NCBI-BLAST must first decompress the sequence. Decompressing the entire sequence can be costly; some sequences are several million base pairs in length and in most cases only parts of the sequence are processed during alignment. Therefore, instead of unpacking the entire sequence, NCBI-BLAST extracts each nucleotide base as required; this is the same approach used during the ungapped alignment stage of BLASTN.

We measured the performance of the third stage using the same 50 randomly-selected queries and test collection used in the previous sections. We found that the third stage consumes 9% of the total nucleotide search time, and on average 1 gapped alignment is performed for every 14 sequences in the collection.

Stage 4

In the final stage of BLAST, high-scoring alignments are recorded and displayed to the user. The gapped alignment algorithm described in the previous section records only the score of the optimal alignment between the query and collection sequences, and not the alignment itself. Therefore, a modified version of the algorithm that also records traceback information is used for the fourth stage.

To perform gapped alignment with traceback, each collection sequence is fully decompressed by NCBI-BLAST before the alignment is performed and, unlike in the first three stages, wildcards are re-inserted into the sequence to improve the accuracy of the final alignment. Only alignments that are considered statistically significant and score above the E-value cutoff (Karlin & Altschul 1990, Altschul & Gish 1996) are processed during the fourth stage. Further, a maximum of V alignments are processed and displayed to the user. In our experiments, we found that 0.01% of collection sequences produced an alignment that was reported to the user and that the final stage consumes only 1% of the total search time.

An alternative version of BLAST called WU-BLAST has been developed by researchers at Washington University[7]. The authors claim better sensitivity and faster search times than NCBI-BLAST, however the precise details of the algorithm and its source code are not publicly available. As a result, we have only considered the NCBI version in our descriptions and experimental evaluations.

[7] See: `http://blast.wustl.edu/`

3 Novel bytepacked approaches

In this section, we describe new approaches to each stage of BLAST that permit sequences in the collection to be processed in their compressed form.

3.1 Stage 1: Hit detection

As described previously, BLASTN performs nucleotide hit detection by searching for occurrences of n matching characters between the query and collection sequences, then extending this initial match in each direction to determine if it forms part of an W-basepair *hit*. To identify matches quickly, BLASTN uses a lookup table to find byte-aligned exact matches of length n and then examines four bases from the collection sequence on each side of the match. This involves reading the compressed or *packed* bytes on each side of the initial match, unpacking them, and aligning the adjacent nucleotide bases with the respective query characters. If $W - n$ adjacent bases are found to match then a W-base hit has been identified; the coordinates of the hit are then passed onto the second stage and an ungapped extension is performed.

We have optimised the hit detection process by developing a faster method for extending the initial hit of length n in each direction to check for a W-basepair hit. Rather than decompress bytes from the collection sequence that are adjacent to the initial match, our approach uses a pair of fast lookup tables and a special representation of the query to extend the hit without decompressing the collection sequence and in fewer operations.

Let us define a special bytepacked representation of the query sequence, where each overlapping quadruplet from the query is packed into a byte, that is $Q = \{q_{[1:4]}, q_{[2:5]}, ..., s_{[|q|-3:|q|]}\}$. This representation can be used to extract a portion of the query starting at any offset in compressed form by selecting every fourth byte. For example, consider the query sequence `ACTTGACAGTAGGACC`. The special representation of this sequence is $Q = \{ACTT, CTTG, TTGA, TGAC, ..., GACC\}$. To extract a substring from the query of length 12 starting from the second character in compressed form, we extract the second, sixth, and tenth bytes from Q; this provides the string `CTTGACAGTAGG` in a bytepacked representation.

We can use the special bytepacked representation of the query to perform fast comparisons between four adjacent characters from the query sequence and four characters from a collection sequence. Let us define $\overrightarrow{M}(q_{[i:i+3]}, s_{[j:j+3]})$ as the number of matching characters between the two bytes $q_{[i:i+3]}$ and $s_{[j:j+3]}$ going from the first character to the last before the first mismatch. Similarly, we define $\overleftarrow{M}(q_{[i:i+3]}, s_{[j:j+3]})$ as the number of matching characters between the bytes going from last character to the first before the first mismatch. For example, $\overrightarrow{M}(ACTT, ACAT) = 2$ and $\overleftarrow{M}(ACTT, ACAT) = 1$. Both functions can be computed quickly by performing a binary XOR operation between the pair of bytes; this provides a single value between 0 and 255 that specifies which character positions within the bytes differ. The result can then be used to fetch pre-computed values for $\overrightarrow{M}$ and $\overleftarrow{M}$ in a specially designed lookup table.

Given an initial hit of length n between bases $q_i \dots q_{i+n-1}$ from the query and bases $s_j \dots s_{j+n-1}$ from the collection sequence we use the new query representation Q and lookup tables for $\overrightarrow{M}$ and $\overleftarrow{M}$ to extend the hit. The new length of the hit N is calculated using the equation:

$$N = n + \overleftarrow{M}\left(q_{[i-4:i-1]}, s_{[j-4:j-1]}\right) + \overrightarrow{M}\left(q_{[i+n:i+n+3]}, s_{[j+n:j+n+3]}\right)$$

If $N \geq W$ then the location of the hit is passed on to the second stage where an ungapped extension is performed.

We have also optimised the lookup table used to identify the initial n-character match. The NCBI-BLAST lookup table contains four 32-bit integer fields for each entry; one to store the number of hits, and the remaining to store up to three query positions in the table entry itself. If a word produces more than three hits then a list of query positions is stored outside of the table, and the table entry instead contains a pointer to the external list. As a result, if a word produces three hits or less then the list of query positions can be accessed without the cache penalty associated with jumping to an external address. This is the same table design used by BLAST for protein searches and is described in more detail elsewhere (Cameron et al. 2006).

The table design is reasonably efficient for the first stage of protein searches, where BLAST considers inexact but high-scoring matches as well as exact matches between words as hits. For protein data, BLAST identifies on average 229 hits per collection sequence (Cameron et al. 2004) and many words produce at least

one hit. This contrasts with nucleotide searches where, as described previously, BLAST identifies on average 3.5 hits per collection sequences and the lookup table is considerably more sparse. We conducted a simple experiment to illustrate this by constructing a BLAST word lookup table using default parameters for 100 nucleotide queries chosen randomly from the GenBank NR database. We found that on average 97.1% of words generated zero hits, 2.6% generated a single hit, and less than 0.3% generated more than one hit.

To address the differing characteristics of nucleotide searches, we have designed a new lookup table that is optimised for nucleotide search. The table records a single integer for each entry; a positive value provides the query position of a single hit, a negative value provides the location of an external, zero-terminated list of query positions, and a zero value indicates that there are no hits for that word. We use 16-bit integers when sufficient, that is, when the query is less than 32,768 bases in length, otherwise 32-bit integers are used. We have also found that the auxiliary lookup table described in Section 2.2 does not improve nucleotide search times and we do not employ the auxiliary table in our own implementation.

3.2 Stage 2: Ungapped alignment

As described in Section 2.2, NCBI-BLAST performs an ungapped extension in each direction from the initial hit until the score decreases by more than the dropoff parameter. During the extension process, BLAST keeps track of the best alignment score observed so far. To perform ungapped extensions on compressed nucleotide sequences, each byte from the collection sequence is unpacked as required, and the sequence is aligned one basepair at a time. Figure 7 provides a pseudo-code description of the ungapped extension algorithm used by NCBI-BLAST for aligning uncompressed collection sequences. Starting at the location of the hit that triggered the extension $[i, j]$, pairs of bases from the query and collection sequences are progressively aligned. The variable *bestscore* records the best alignment score so far, and the alignment process terminates at the end of either sequence or if the alignment score decreases by more than *dropoff*. Note that the psuedo-code given here performs the forward extension of an alignment only; some minor variations are required to perform the backwards extension of a hit.

We have developed a new ungapped alignment algorithm for BLASTN that permits alignment of compressed collection sequences. Using our approach, ungapped alignment is performed four bases at a time. The pseudo-code description of our new algorithm is shown in Figure 8. In addition to the $\overrightarrow{M}$ and $\overleftarrow{M}$ lookup tables described previously, the algorithm uses a third table to compute $\overline{\overline{M}}(q_{[i:i+3]}, s_{[j:j+3]})$, which we define as the total score for matching bases between a pair of bytes. For example, $\overline{\overline{M}}(ACTT, ACAT) = 2$ if a match score of 1 and mismatch score of -1 is used.

The new algorithm aligns collection sequences one byte at a time using the $\overline{\overline{M}}$ lookup table. When the alignment of individual bases may result in an optimal alignment score, that is, when $score > bestscore - (3 \times matchscore)$, a partial alignment of the next byte is also considered by consulting the $\overrightarrow{M}$ lookup table. The variable $finescore$ records the score resulting from the partial alignment. Again, minor variations are required to perform the backwards extension including the use of the $\overleftarrow{M}$ instead of the $\overrightarrow{M}$ lookup function.

Our approach is similar to the ungapped alignment method used by SENSEI (States & Agarwal 1996), which also aligns sequences one byte at a time with the aid of a lookup table. However, their approach only considers alignments that start and end on the byte boundary and so may result in suboptimal alignments. Unreported experiments found that the SENSEI approach leads to a loss in search performance or accuracy when used to perform the second stage of BLAST.

We have employed another minor optimisation to the ungapped extension process that reduces the number of bytes processed. The region covered by the initial match detected in Stage 1 is already known to contain $\frac{n}{4}$ matching bytes and we avoid realigning this region when performing an ungapped alignment.

3.3 Stage 3: Gapped alignment

Computing optimal gapped alignments between a query sequence and compressed collection sequences is significantly more complicated than computing ungapped alignments; there are several ways to align a packed byte from the collection with the query sequence when gaps are also considered. For example, optimal gapped alignments may include insertions or deletions in the middle of the packed byte. To illustrate this, consider the example gapped alignment between query sequence `ATGCAGTT` and collection sequence `ATGGTT` at the top-left of Figure 9. The first four characters of the collection sequence, `ATGG`, are represented by a single

```
UngappedExtension(q, s, i, j, dropoff)
int score = 0
int bestscore = 0
int i_best = 0, j_best = 0

while i ≤ |q| and j ≤ |s|
    if q_i = s_j then
        score ⟵ score + matchscore
    else
        score ⟵ score − mismatchscore
    if score > bestscore then
        bestscore ⟵ score
        i_best ⟵ i
        j_best ⟵ j
    else if bestscore − score > dropoff
        then stop
    increment i
    increment j
```

Fig. 7. Original ungapped extension algorithm used by NCBI-BLAST for aligning collection sequences.

```
BytePackedExtension(q, s, i, j, dropoff)
int score = 0
int bestscore = 0
int i_best = 0, j_best = 0

while i ≤ |q| and j ≤ |s|
    if score > bestscore − (3 × matchscore) then
        finescore ⟵ score + M⃗(q_[i:i+3], s_[j:j+3])
        if finescore > bestscore then
            bestscore ⟵ finescore
            i_best ⟵ i
            j_best ⟵ j
    else if bestscore − score > dropoff then
        stop
    score ⟵ score + M⃡(q_[i:i+3], s_[j:j+3])
    increment i by 4
    increment j by 4
```

Fig. 8. Improved ungapped extension algorithm for aligning compressed collection sequences four bases at a time.

packed byte and the optimal alignment involves two insertions in the middle of the byte. A scheme that aligns sequences one byte at a time without considering insertions in the middle of a byte does not produce this optimal alignment.

In this section, we propose two new approaches to gapped alignment that address this problem, *bytepacked alignment* and *table-driven alignment*. Bytepacked alignment restricts the location of gaps in an alignment so that they only occur on the collection sequence byte boundary. As a result, collection sequences can be aligned a byte at a time, however the approach generates suboptimal alignments. Table-driven alignment considers all possible alignments between a single base from the query sequence and a packed byte from the collection sequence through the use of a specially designed lookup table. Although table-driven alignment is lossless, our approach is only suitable for aligning sequences using non-affine gap costs. As we show later, both techniques provide a good approximation of the gapped alignment score and work well when employed as a filtering step between the ungapped and gapped alignment stages of BLAST.

The advantages of our new approaches are two-fold. First, both techniques process the collection sequence one byte at a time, rather than one base at a time, providing a significant reduction in processing when compared to gapped alignment. Second, collection sequences do not need to be decompressed to perform bytepacked or table-driven alignment. Because both techniques provide an additional filtering step between ungapped alignment and gapped alignment, fewer collection sequences need to be decompressed before being realigned using gapped alignment.

Bytepacked alignment

In this section we propose our novel bytepacked alignment scheme. With restrictions, it enables bytepacked alignment to be performed on compressed collection sequences. Specifically, gaps can only start and end on the collection sequence byte boundary, that is where $j \equiv 0, \text{modulo } 4$, and as a result bytepacked alignment provides an approximation of the optimal gapped alignment score.

Bytepacked alignment is performed in a similar manner to gapped alignment. The key difference is that bytepacked alignment uses one row in the dynamic programming matrix for each packed byte in the collection sequence, rather than one row for each individual base. By restricting the start and end of gaps to lie on the byte boundary, it is possible to consider four bases at a time during alignment, as illustrated in Figure 10. The y-axis of the figure represents the sequence in its compressed form and the query is represented as a series of overlapping quadruplets along the x-axis; this is the same representation of the query, Q, described

previously. Three events are considered for each cell in the matrix; a match of four bases — or a single byte — from the query sequence and collection sequence, a single insertion in the collection sequence, or a series of four insertions in the query sequence.

Figure 9 illustrates bytepacked alignments for two example sequence pairs, with the optimal gapped alignments and corresponding bytepacked alignments shown at the top and bottom of the figure respectively. The example on the left-hand side illustrates the case where the alignment contains insertions in the collection sequence. In Figure 9 (a) the optimal gapped alignment contains two insertions in the collection sequence that move the alignment from one diagonal to another. Bytepacked alignment still permits the insertions, however they must occur on the sequence byte boundary as illustrated in Figure 9 (b). The example on the right-hand side illustrates the case where the optimal alignment contains insertions in the query sequence. In Figure 9 (c) the optimal alignment contains two insertions in the query. However, gaps can only start and end on the collection sequence byte boundary in a bytepacked alignment and, as a result, only insertions in the query of length i where $i \equiv 0, \text{modulo } 4$ are permitted. Therefore, the bytepacked alignment must contain a pair of *adjacent gaps* — one in the query of length four followed immediately by another in the collection sequence of length two — to move the alignment to the new diagonal in Figure 9 (d).

To minimise the scoring penalty associated with adjacent gaps in bytepacked alignments we have employed the two state variation of the Gotoh gapped alignment algorithm (Durbin 1998). The two state variation produces identical alignment scores to the original approach except for alignments that contain adjacent gaps. In the two state variation a single open gap penalty is incurred for the pair of adjacent gaps, whereas two open gap penalties are applied in the original approach. By reducing the scoring penalty for adjacent gaps, bytepacked alignment provides a better approximation of gapped alignment scores.

The gapped alignment algorithm described in Section 2.2 records three values for each cell in the dynamic programming matrix; $B(i,j)$ is the highest score for any alignment ending at $[i,j]$, $I_q(i,j)$ is the highest score for any alignment ending at $[i,j]$ with an insertion in the query, and $I_s(i,j)$ is the highest score for an alignment ending at $[i,j]$ with an insertion in the collection sequence. The two state variation combines $I_q(i,j)$ and $I_s(i,j)$ into a single maximum value, $I(i,j)$, that represents the best score for an alignment ending at $[i,j]$ with an insertion in either direction. The following recurrence relations are used to perform bytepacked alignment between a compressed collection sequence s and the query sequence q that is represented using the special overlapping quadruplet representation Q:

$$M(i,j) = B(i-4, j-4) + \overline{\overline{=}}_M(q_{[i-3:i]}, s_{[j-3:j]})$$

$$I(i,j) = \max \begin{cases} M(i-1,j) - d \\ I(i-1,j) - e \\ M(i,j-4) - d - 3e \\ I(i,j-4) - 4e \end{cases}$$

$$B(i,j) = \max \begin{cases} I(i,j) \\ M(i,j) \end{cases}$$

where the temporary scalar $M(i,j)$ represents the best score for any alignment ending at $[i,j]$ with four matching bases. In addition to the recurrence relations, initialisation rules are used to handle boundary conditions in the matrix; all cells where $j = 0$ or $-3 \geq i \geq 0$ are initialized to $-\infty$, except for the starting point $[0,0]$ which is initialised to zero.

Bytepacked alignment places restrictions on the location of gaps in the alignment. However, we expect the restrictions to have a minor effect on alignment scores for two reasons related to those discussed in our previous work on protein search (Cameron et al. 2004):

1. Bytepacked alignment still permits gaps, but forces them to occur in a suboptimal location or with a different arrangement. In cases where the optimal alignment contains a gap in the collection sequence, bytepacked alignment shifts the start and end location by no more than 2 bases to ensure the gap occurs on the collection sequence byte boundary. Unless the optimal gap is adjacent to high-scoring bases, the shift will not significantly change the alignment score. In cases where the optimal alignment contains a gap of length G in the query sequence, the corresponding bytepacked alignment will contain a gap of length $\lceil \frac{G}{4} \rceil \times 4$ in the query and a gap of length $4 - G$ modulo 4 in the collection sequence. Assuming gap

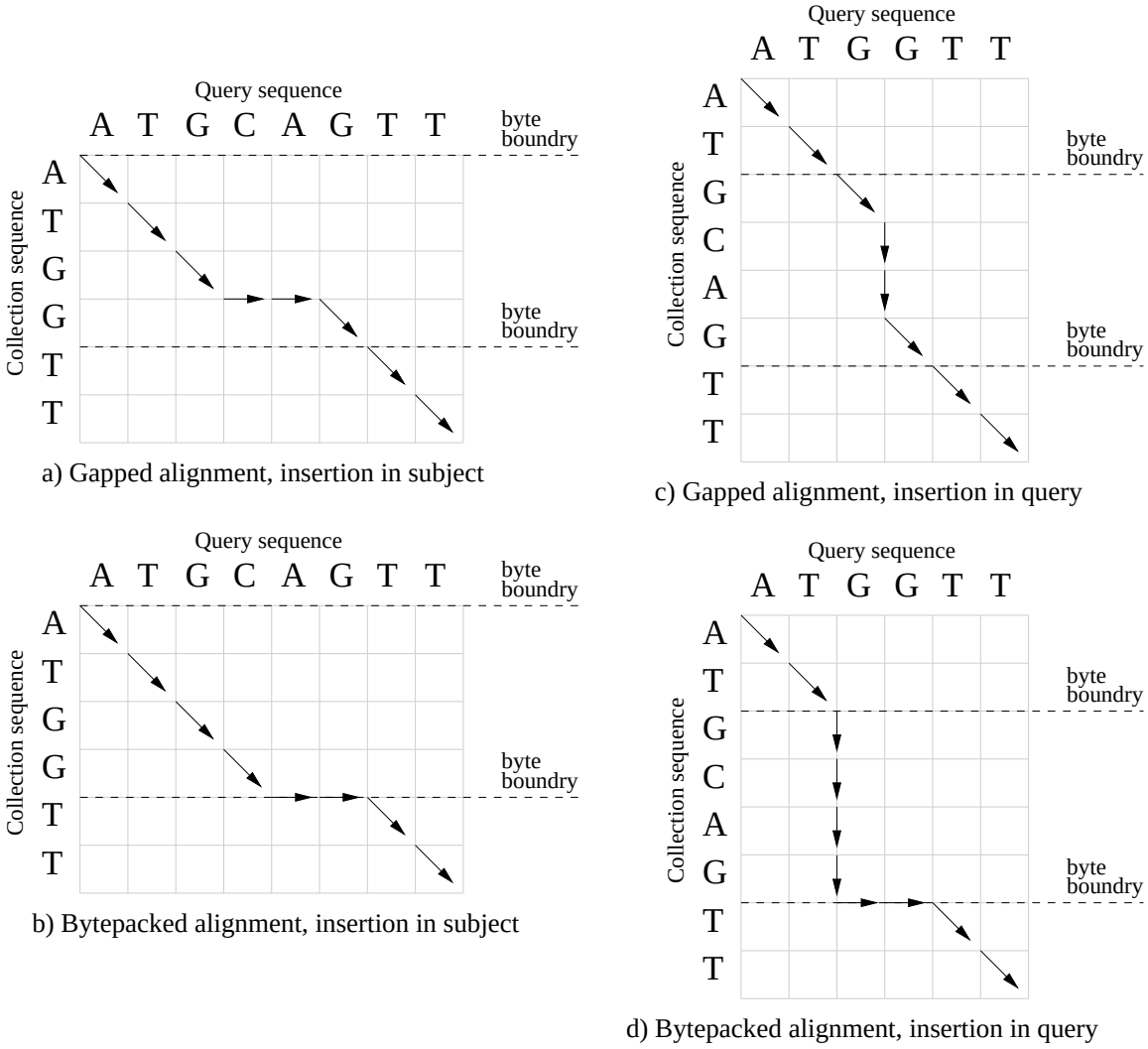

Fig. 9. *Illustration of the effect of bytepacked alignment constraints on example alignments. a) illustrates a gapped alignment containing insertions in the collection sequence and b) shows the equivalent bytepacked alignment with the insertions shifted to the byte boundary. c) illustrates a gapped alignment containing insertions in the query and d) shows the equivalent bytepacked alignment that uses adjacent gaps to change diagonal.*

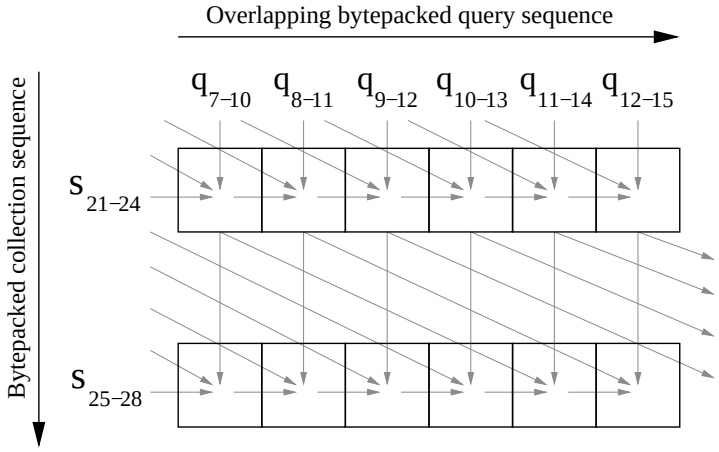

Fig. 10. *Portion of the dynamic programming matrix used to perform bytepacked alignment. A diagonal arrow represents a match between four bases, or a single byte, from the query and collection sequences, a horizontal arrow represents a single insertion in the collection sequence, and a vertical arrow represents four insertions in the query.*

lengths are uniformly distributed, this results in an average additional gap penalty of $3e$ when the two state algorithm is used. If the recurrence relations described in Section 2 were used instead, the average increase in gap penalty would be $d + 2e$, illustrating the advantage of the two state variation, since d is usually much larger than e.

2. The scoring penalty associated with a gap typically outweighs any additional penalty incurred by opening the gap at a suboptimal location or increasing the length of the gap. Further, the additional penalty can be compensated for by decreasing the open gap penalty, as shown in our previous work with protein alignments (Cameron et al. 2004). We report experiments with varying the open gap penalty used for bytepacked alignment in Section 4.

In terms of computational cost, bytepacked alignment represents a significant saving when compared to gapped alignment. The number of cells in the dynamic programming matrix is reduced by roughly a factor of four and the amount of computation per cell is almost unchanged. Some additional computation is required to match a pair of bytes, instead of a pair of characters, when calculating $\overline{\overline{s}}_M(q_{[i-3:i]}, s_{[j-3:j]})$ for each cell. However, $\overline{\overline{s}}_M$ can be calculated quickly by performing a binary XOR and using a specially designed lookup table to calculate the score. Further, our results in Section 4 show that bytepacked alignment offers a substantial speed gain.

We have found that bytepacked alignment is best employed as an additional filtering step between the ungapped and gapped alignment stages of BLASTN. Therefore, we need a method for deciding which bytepacked alignments should be passed on to the gapped alignment stage. We have chosen to perform gapped alignment on collection sequences with a bytepacked alignment score above $R \times S2$, where S2 is the nominal score required to achieve the E-value cutoff and $0 < R \leq 1$. Figure 11 illustrates the new process for scoring sequences using bytepacked alignment. In the example, an ungapped extension is performed first, followed by a bytepacked alignment where gaps can start and end only on the byte boundary. The bytepacked alignment scores above $R \times S2$ and the collection sequence is unpacked and realigned using gapped alignment. The choice of R affects the speed and sensitivity of BLAST, and we report experiments with varying values of R in Section 4.

Table-driven alignment

In this section, we propose our novel table-driven alignment, an alternative to the bytepacked alignment approach. The approach is based on the Four Russians concept (Wu et al. 1996) that involves dividing a problem into subsections and solving each subsection using precomputed answers found in a lookup table.

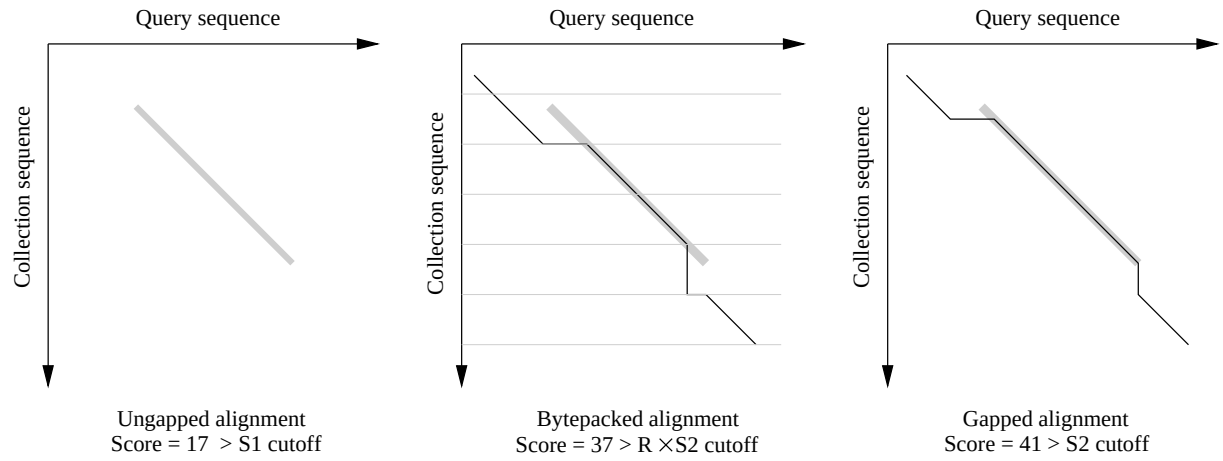

Fig. 11. *Process for scoring sequences using bytepacked alignment. First, an ungapped extension is performed and the resulting alignment scores above the S1 cutoff. Next, a bytepacked alignment is performed where gaps can start and end only on byte boundaries, illustrated by grey horizontal lines. The resulting alignment scores above $R \times S2$ and is passed onto the last stage, where an unrestricted gapped alignment is performed.*

In the context of sequence alignment, the dynamic programming matrix is divided into blocks of adjacent cells that are processed simultaneously. The values of proceeding cells are used as input into the table and alignment scores for each cell in the current block are provided as output.

The Four Russians concept has previously been applied to general string matching problems (Wu et al. 1996), regular expression pattern matching (Myers 1992), protein sequence alignment (Myers & Durbin 2003), and the alignment of nucleotide sequences compressed using the Lempel-Ziv compression scheme (Crochemore, Landau & Ziv-Ukelson 2002). The latter application is closely related to our approach, however Lempel-Ziv compression is not suitable for use with the first two stages of BLAST. Instead, we have applied the Four Russians approach to the alignment of bytepacked nucleotide sequences permitting four rows of the alignment matrix to be processed at a time. This is achieved by dividing the alignment matrix into subsections of size 1×4 that correspond to the alignment of one base from the query and one packed byte from the collection sequence. Similar to the other techniques discussed in this paper, this provides two major advantages over gapped alignment; first, the collection sequence does not need to be decompressed for the alignment to be performed, and second, a significant reduction in computation is achieved by using a lookup table to process four characters from the collection sequence at a time.

Figure 12 illustrates the table-driven alignment approach to processing four cells in the matrix at a time with consideration of all possible alignments, including gaps, between a single query base q_{a+1} and a packed byte $s_{[b+1:b+4]}$ from the collection sequence. A bytepacked collection sequence and uncompressed query sequence are used as input. On the left-hand side of Figure 12, the alignment matrix is divided into blocks of adjacent cells. Values for the proceeding grey cells have already been computed, and values for the four empty cells shown on the right-hand side of the figure are calculated by consulting a lookup table. The input into the table consists of values for the six neighbouring grey cells and a *match vector* that specifies which bases in the packed byte $s_{[b+1:b+4]}$ match the query base q_{a+1}. The arrows in the right-hand side of the figure represent the match, mismatch and insertion events that are considered for each block.

A significant problem in implementing the Four Russians approach is keeping the lookup table small; a large table will not fit into CPU cache, leading to poor performance due to an increase in latency associated with accessing main memory. The alignment algorithm described in Section 2 records three values, $B(i, j)$, $I_x(i, j)$ and $I_y(i, j)$, for each cell. The values for six cells are used as input into the table, as illustrated in Figure 12, resulting in a total of 18 distinct inputs and a prohibitively large table. Therefore, we have made two optimisations to reduce the size of the table.

Our first optimisation is to use the following simplified recurrence relations that do not allow for affine gap costs when performing the alignment:

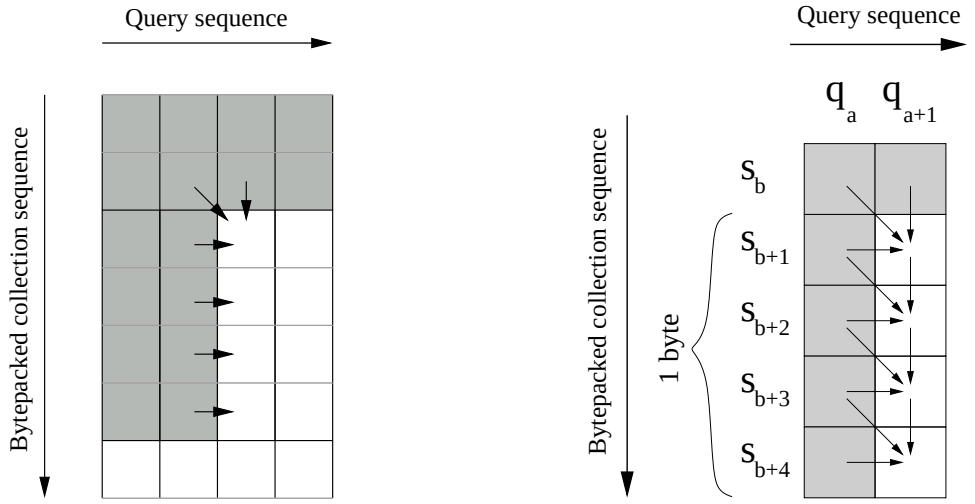

Fig. 12. *Illustration of table-driven alignment. The alignment matrix is divided into blocks of four adjacent cells, and alignment scores for each block are calculated using a lookup table. Previously computed values in the six neighbouring grey cells and a* match vector *that specifies whether the query base q_{a+1} matches each base in the collection sequence byte $s_{[b+1:b+4]}$ are used as input into the table, which provides values for the four white cells.*

$$B(i,j) = \max \begin{cases} B(i,j) + s(q_i, s_j) \\ B(i-1,j) - e \\ B(i,j-1) - e \end{cases}$$

where e is the cost of each insertion, and no penalty is applied for the opening of a gap. As a result, only one value is recorded for each cell in the matrix; $B(i,j)$ represents the highest score for any alignment ending at $[i,j]$. This reduces the number of scores used as input into the lookup table from 18 to 6. Our results in Section 4 show this approach still provides a good approximation of affine gap cost alignment scores, and works well in practice.

Our second optimisation is to use the difference between neighbouring values rather than absolute values as input into the table, an approach best described by Jones and Pevzner (2004). Encoding differences rather than absolute values reduces the size of the lookup table, because the differences are limited to a smaller range. Let us define the difference in score $\Delta B(i_C, j_C, i_D, j_D) = B(i_D, j_D) - B(i_C, j_C)$ where $[i_C, j_C]$ and $[i_D, j_D]$ are immediately neighbouring cells such that $i_D = i_C, j_D = j_C + 1$ or $j_D = j_C, i_D = i_C + 1$. When the above recurrence relations are used, the value of ΔB is limited such that $-e \leq \Delta B(i_C, j_C, i_D, j_D) \leq e + r$ where r is the alignment score for two matching bases, $r > 0$. When the default BLAST values of $m = 1$ and $e = 2$ are used, the difference between any cell and its immediately proceeding neighbour is no less than -2 and no more than 3, a range of 6 possible values.

To illustrate how our second optimisation is applied to table-driven alignment, let us return to the illustration in Figure 12. The cell $[a, b]$ acts as a reference, with other inputs and outputs encoded relative to its value. The remaining values in the q_a column are encoded as $\Delta B(a, j-1, a, j)$ where $b + 1 \geq j \geq b + 4$. Similarly, the value for cell $[a+1, b]$ is encoded as $\Delta B(a, b, a+1, b)$. This provides a total of 5 inputs into the table, each with $2e + r + 1$ possible values. The only remaining input necessary for calculating scores for the empty cells is the match vector, the 4-bit value specifying if the query base q_a matches or mismatches each of the bases $s_{b+1} \ldots s_{b+4}$. Using this optimised design, the lookup table contains a total of $(2e + r + 1)^5 \times 2^4$ or 124,416 entries when default BLASTN parameters are used.

We have carefully implemented the Four Russians approach to minimise the amount of computation per block. The match vector is generated efficiently by performing a binary XOR between the packed byte $s_{[b+1:b+4]}$ from the collection sequence and a special bytepacked representation of q_{a+1} that contains the

single query base repeated four times, $q_{[a+1,a+1,a+1,a+1]}$. The result of the binary XOR operation is then used to fetch the match vector from a small, specially designed lookup table. We have also used the *carry in*, *carry out* method when implementing our Four Russians approach (Wu et al. 1996). A *row carry in* codeword specifies the four values $\Delta B(a, j-1, a, j) \mid b+1 \geq j \geq b+4$ and is used as input into the lookup table. The output of the table, which we refer to as *row carry out*, includes a codeword formatted in the same manner as row carry in and specifies the four values $\Delta B(a+1, j-1, a+1, j) \mid b+1 \geq j \geq b+4$. The row carry out codeword can then be used as a row carry in when processing the next column in the dynamic programming matrix without the need to decode it; this provides a significant computational saving. Similarly, the value $\Delta B(a, b, a+1, b)$ is the *column carry in* used as input into the table, and the table outputs a *column carry out* specifying the value of $\Delta B(a, b+4, a+1, b+4)$. The column carry out can then be used as the column carry in when processing the next four rows in the current column.

Unlike our bytepacked alignment scheme, table-driven alignment considers all possible paths through the alignment matrix without any restriction on the locations of gaps. Although the table-driven approach does not support affine gaps costs, we set the table-driven alignment insertion penalty e_t to equal the gapped alignment extension penalty $e_t = e_g$. Assuming the dropoff regions explored by gapped alignment and table-driven alignment are the same, the latter produces alignment scores equal to or greater than those produced by the former. Any table-driven alignment that scores above the nominal score required to achieve cutoff $S2$ is rescored using gapped alignment, guaranteeing no loss in accuracy when table-driven alignment is used. The main drawback with table-driven alignment is its reliance on fast access to main-memory to consult the lookup table when calculating values for each subsection of the alignment matrix. This relies upon the lookup table fitting into CPU cache for reasonable performance.

4 Results

In this section we compare NCBI-BLAST to FSA-BLAST, our own freely available version of BLAST that incorporates the improvements to nucleotide search described in this paper. First, we describe the collections and performance and accuracy measures used for our experiments. We then present overall results for our approaches, including bytepacked and table-driven alignment. Finally, we present results for varying parameter choices for our schemes.

4.1 Test collection, environment and measurements

Annotated protein sequence collections such as the SCOP (Murzin, Brenner, Hubbard & Chothia 1995, Andreeva, Howorth, Brenner, Hubbard, Chothia & Murzin 2004) and PIR (Barker, Garavelli, Hou, Huang, Ledley, McGarvey, Mewes, Orcutt, Pfeiffer, Tsugita, Vinayaka, Xiao, Yeh & Wu 2001) databases provide sequence classifications that are ideal for evaluating the retrieval effectiveness of protein homology search tools. Unfortunately, no similarly annotated collections exist for measuring the accuracy of nucleotide search tools and their ability to identify related sequences. Therefore, we have employed the approach described by Li et al. (2004) for measuring retrieval effectiveness by comparing search results to the complete set of alignments identified by the exhaustive Smith-Waterman algorithm (1981).

A reduced version of the GenBank non-redundant (NR) nucleotide database[8] is the largest collection available for search through the NCBI online BLAST interface, and the default collection for search. For our evaluation, we used a copy of the NR database downloaded on 4 April 2005. To minimise the time spent generating Smith-Waterman alignments for our experiments, we used only half of the NR database by randomly extracting sequences from the collection and creating our own test collection, which we refer to as NR/2. The NR/2 collection contains 6,862,797,036 basepairs in 1,511,546 sequences, ranging in length from 6 to 36,192,742 basepairs. We use the NR/2 database to measure both search runtime and accuracy in our experiments.

A set of 50 test queries were randomly extracted from the NR database. Queries longer than 10,000 basepairs — typically entire genomes or chromosomes — were excluded from the selection process; BLAST searches with longer queries are too slow to be practical and less sensitive genome search tools such as BLAT and MEGABLAST are better suited to such searches. Queries were pre-filtered using our own implementation

[8] This does not include EST, STS, and GSS sequences, environmental samples, or phase 0, 1, or 2 HTGS sequences.

of the DUST filter, identical in output to the version of the filter used by NCBI-BLAST. Each query was searched against the entire NR/2 database using our own implementation of the Smith-Waterman algorithm that is distributed with FSA-BLAST. We use our implementation of Smith-Waterman instead of the SSEARCH tool distributed with FASTA to ensure identical scoring and statistical calculations between our baseline and BLAST (Karlin & Altschul 1990, Altschul & Gish 1996).

To evaluate the accuracy of BLAST, we used each of our 50 test queries to search the NR/2 collection using default parameters, including $v = 250$ where a maximum of v alignments are displayed to the user. We compared the alignments returned by BLAST to the best L alignments returned by Smith-Waterman, ranked by nominal score. We used $L = v$ by default, except for queries where the set of v best alignments was ambiguous because alignments had identical scores. For these queries we set L to the smallest value such that $L \geq v$ and $a_L > a_{L+1}$ where a_i is the score of the i^{th} alignment.

For each query we measured search accuracy with the commonly-used *Receiver Operating Characteristic* (ROC) (Gribskov & Robinson 1996), using the best L Smith-Waterman alignments as the set of true positives. The ROC score is calculated as:

$$ROC = \frac{1}{nL} \sum_{1 \leq F \leq n} u^F$$

where F is the ranked position of the F^{th} false positive in the list of alignments returned by BLAST, u^F is the number of true positives that are ranked ahead of the F^{th} false positive, and n is the number of false positive alignments returned by BLAST. We only consider an alignment to be positive if the score returned by BLAST is at least half the optimal Smith-Waterman alignment score, an approach also used by Li et al. (2004). The final reported ROC score is calculated by taking the average across all queries where $L > 1$. This provides a good measure of the sensitivity of BLAST and the level of accuracy for identifying the best v alignments. A higher ROC score indicates better search accuracy.

For our timing experiments, we compared each of the 50 test queries to the NR/2 database and recorded the best elapsed time of three runs. We report the average search time for 50 queries throughout this section. Experiments were carried out on a Pentium 4 2.8GHz workstation with 2 Gb of main-memory while the machine was under light load. We used version 2.2.10 of NCBI-BLAST as our baseline and all code was compiled with default NCBI-BLAST compiler flags and optimisations. All experiments were conducted using default BLAST parameters including a gapped alignment trigger score of 25.0 bits, an ungapped extension dropoff of 20.0 bits, a gapped extension dropoff of 30.0 bits, and an E-value cutoff of 10.0. Stage 3 search times include table-driven and bytepacked alignment times when these approaches were used.

4.2 Overall results

An overall comparison between NCBI-BLAST and our own implementation of BLAST based on the improvements described in this paper is presented in Table 2. Our results show that FSA-BLAST is more than twice as fast as NCBI-BLAST regardless of whether table-driven alignment or bytepacked alignment is employed. Our new method for extending hits without decompressing collection sequences and our new lookup table design reduce average search times by 10.63 seconds, equivalent to a 49% reduction in Stage 1 search time. Our new algorithm for performing ungapped alignment using compressed collection sequences results in a further saving of 0.53 seconds per query, with Stage 2 times reduced by 43%. Our bytepacked alignment scheme results in a 78% reduction in time for the gapped alignment stages of BLAST, and our table-driven alignment scheme results in an 72% reduction in time for the final two stages. Although both schemes align collection sequences one byte at a time, bytepacked alignment provides slightly better performance than table-driven alignment; this is likely due to the latency incurred by table-driven alignment when accessing the in-memory lookup table. Importantly, there is no significant change in accuracy between NCBI-BLAST and FSA-BLAST when either scheme is employed.

Further experiments reveal that table-driven and bytepacked alignment are indeed accurate approximations of gapped alignment and effective for detecting potentially high-scoring alignments. When bytepacked alignment is employed on average only 1,039 out of 107,076 bytepacked alignments score above $R \times S2$ and are realigned using gapped alignment. Roughly 99% of high-scoring ungapped alignments are discarded by the bytepacked approach. Since the bytepacked approach does not lead to a loss in search accuracy (Table 2),

	NCBI-BLAST	FSA-BLAST	
		Bytepacked alignment	Table-driven alignment
Stage 1 (secs)	21.85	11.22	11.22
Stage 2 (secs)	1.23	0.70	0.70
Stage 3 (secs)	2.25	0.37	0.46
Stage 4 (secs)	0.34	0.19	0.26
Total (secs)	25.67	12.48	12.64
ROC	0.973	0.974	0.973

Table 2. Average runtime in seconds for each stage of BLAST and ROC search accuracy when searching the NR/2 database. All alignment techniques use default parameters.

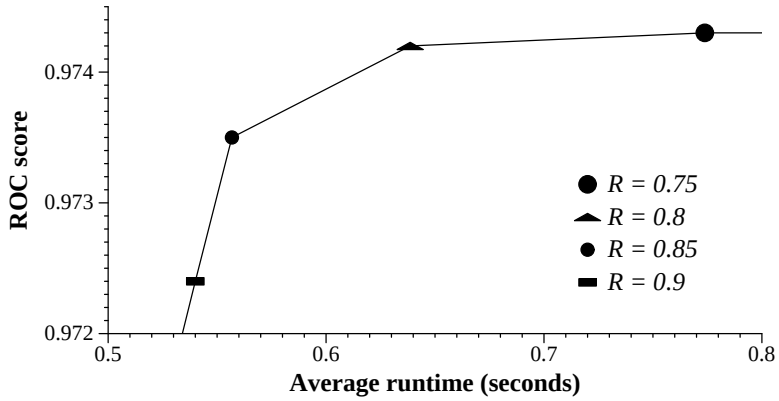

Fig. 13. *Average runtime and ROC search accuracy for searches against NR/2 database using bytepacked alignment and varying values of R.*

this indicates that it provides a good approximation of optimal alignment scores. When table-driven alignment is employed on average only 322 out of the 107,076 table-driven alignments score above $S2$, with less than 0.5% of table-driven alignments rescored using the gapped alignment approach. The result indicates that alignment using non-affine gap penalties can provide a good approximation of the optimal alignment score with affine gap costs.

Experiments were conducted using default bytepacked alignment parameters $o_b = 0$ and $R = 0.85$, where o_b is the open gap penalty used for bytepacked alignment and R controls the rescoring of collection sequences using gapped alignment as discussed in Section 3.3. We present the results of our experiments with varying values of R and o_b next.

4.3 Varying R

The bytepacked alignment scheme is parameterised by the constant R that provides a trade-off between alignment speed and accuracy. As discussed in Section 3.3, only collection sequences with a bytepacked alignment score above $R \times S2$, where S2 is the nominal score required to achieve cutoff and $0 < R \leq 1$, are realigned using gapped alignment. Figure 13 shows the effect of varying R on the runtime and accuracy of FSA-BLAST when bytepacked alignment is employed. As R decreases, more collection sequences are realigned using gapped alignment resulting in improved accuracy and longer search times. A larger value of R results in fewer gapped alignments and reduced search times at the expense of search accuracy, with BLAST more likely to miss high-scoring alignments. The highlighted value $R = 0.85$ provides a good compromise between search speed and accuracy, and we have chosen this value as our default setting.

4.4 Varying the Open Gap Penalty

Bytepacked alignment places restrictions on the location of gaps, resulting in suboptimal alignments as discussed in Section 3.3. Therefore, in most cases where the optimal alignment contains insertions or deletions,

Open gap penalty (o_b)	0	1	2	3	4	5
R	0.85	0.80	0.75	0.75	0.55	0.55
ROC	0.974	0.974	0.973	0.973	0.973	0.973
Time (seconds)	0.28	0.30	0.32	0.32	0.33	0.34

Table 3. Average query evaluation time (stages 3 and 4 only) for varying bytepacked alignment open gap penalty. For each penalty, and value of R that produces an equivalent *ROC* score was chosen.

the bytepacked alignment score and gapped alignment score differ. To compensate for this, we consider lowering the open gap penalty used for bytepacked alignment, with the aim of providing a better approximation of the gapped alignment score. Let us define o_b and e_b as the open and extend gap penalties used for bytepacked alignment. Because gap length has little effect on the constraints imposed by bytepacked alignment, we let $e_b = e$. Next, we select pairs of values for o_b and R with similar accuracy and measure the average search time. The results for this experiment are shown in Table 3. The results indicate that a lower open gap penalty provides faster average search times with comparable accuracy, and that the minimum penalty $o_b = 0$ provides best results; we use this value as our default. Note that an open gap penalty of zero implies a non-affine gap cost, with the added advantage of simplifying the alignment algorithm and reducing the computation per cell in the alignment matrix. We have exploited this in our implementation of bytepacked alignment.

5 Conclusion

Despite the emergence of several new homology search algorithms, BLAST remains the most versatile and popular bioinformatics tool and it is used to conduct millions of searches each day. With such a large number of nucleotide searches being performed each day, it is important that BLASTN is both efficient and accurate, especially given the rapid grown of sequence databanks such as GenBank. In this paper, we have described several improvements to BLASTN that permit faster nucleotide searches. Our schemes rely on comparing nucleotide sequences using a simple bytepacked representation, leading to reduced search times in two ways: first, collection sequences can be processed using their on-disk representation without the need to decompress them, and second, sequences can be aligned much faster by comparing four nucleotide bases at a time.

We have presented optimisations to the first two stages of BLAST that result in a 43% reduction in average search time. We have also described two new approaches to gapped alignment that allow the alignment of compressed collection sequences. Our first alignment algorithm, bytepacked alignment, places restrictions on the location of gaps permitting four bases to be aligned at a time. Although the scheme is a heuristic, our results indicate that when carefully applied to BLAST, bytepacked alignment reduces the time taken to perform gapped alignment by 78% with no significant effect on accuracy. Our second alignment algorithm, table-driven alignment, uses the Four Russians approach of dividing the alignment matrix into sections and solving each section using precomputing values from a lookup table. Table-driven alignment is lossless, and reduces the time taken by BLAST to perform gapped alignment by 72%.

Our improvements to BLASTN, along with our previously described improvements to BLASTP, have been integrated into a new, open source version of the tool that is freely available for download from `http://www.fsa-blast.org/`. However, the ideas described in this paper are not only applicable to BLAST. Indeed, several other search tools use bytepacked compression to store collection sequences including PATTERNHUNTER (Li et al. 2004), BLAT (Kent 2002), MEGABLAST (Zhang et al. 2000), SENSEI (States & Agarwal 1996), and SSAHA (Ning, Cox & Mullikin 2001). We expect that many of the schemes presented in this paper would yield similar speed improvements when applied to these algorithms.

Acknowledgements

This work was supported by the Australian Research Council. We thank Tom Madden and David Lipman for providing NCBI search statistics.

References

Altschul, S. & Gish, W. (1996), "Local alignment statistics", *Methods in Enzymology* **266**, 460–480.

Altschul, S., Gish, W., Miller, W., Myers, E. & 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. & Lipman, D. (1997), "Gapped BLAST and PSI–BLAST: A new generation of protein database search programs", *Nucleic Acids Research* **25**(17), 3389–3402.

Andreeva, A., Howorth, D., Brenner, S., Hubbard, T., Chothia, C. & Murzin, A. (2004), "SCOP database in 2004: refinements integrate structure and sequence family data", *Nucleic Acids Research* **32**, D226–D229.

Barker, W., Garavelli, J., Hou, Z., Huang, H., Ledley, R., McGarvey, P., Mewes, H., Orcutt, B., Pfeiffer, F., Tsugita, A., Vinayaka, C., Xiao, C., Yeh, L. & Wu, C. (2001), "Protein information resource: a community resource for expert annotation of protein data", *Nucleic Acids Research* **29**(1), 29–32.

Benson, D., Karsch-Mizrachi, I., Lipman, D., Ostell, J. & Wheeler, D. (2004), "Genbank", *Nucleic Acids Research* **32**, D23–D26.

Brenner, S., Chothia, C. & 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., Li, M. & Ma, B. (2004), "A tutorial of recent developments in the seeding of local alignment", *Journal of Bioinformatics and Computational Biology* **2**(4), 819–842.

Califano, A. & Rigoutsos, I. (1993), FLASH: a fast look-up algorithm for string homology, *in* "Proceedings of the Fourth International Conference on Intelligent Systems for Molecular Biology", Vol. 1, pp. 56–64.

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

Cameron, M., Williams, H. E. & Cannane, A. (2006), "A deterministic finite automaton for faster protein hit detection in BLAST", *Journal of Computational Biology* **13**(4), 965–978.

Chao, K., Pearson, W. & Miller, W. (1992), "Aligning two sequences within a specified diagonal band", *Computer Applications in the Biosciences* **8**(5), 481–487.

Chen, X. L. (2004), A framework for comparing homology search techniques, Master's thesis, School of Computer Science and Information Technology, RMIT University.

Crochemore, M., Landau, G. M. & Ziv-Ukelson, M. (2002), A sub-quadratic sequence alignment algorithm for unrestricted cost matrices, *in* "SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms", Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, pp. 679–688.

Durbin, R. (1998), *Biological sequence analysis: probabilistic models of proteins and nucleic acids*, Cambridge University Press.

Fondrat, C. & Dessen, P. (1995), "A rapid access motif database (RAMdb) with a search algorithm for the retrieval patterns in nucleic acids or protein databanks", *Computer Applications in the Biosciences* **11**(3), 273–279.

Gotoh, O. (1982), "An improved algorithm for matching biological sequences", *Journal of Molecular Biology* **162**(3), 705–708.

Gribskov, M. & Robinson, N. (1996), "Use of receiver operating characteristic (ROC) analysis to evaluate sequence matching", *Computers & Chemistry* **20**, 25–33.

Jones, N. C. & Pevzner, P. A. (2004), *An Introduction to Bioinformatics Algorithms*, MIT Press.

Karlin, S. & Altschul, S. (1990), "Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes", *Proceedings of the National Academy of Sciences USA* **87**(6), 2264–2268.

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

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

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

Murzin, A., Brenner, S., Hubbard, T. & Chothia, C. (1995), "SCOP: a structural classification of proteins database for the investigation of sequences and structures", *Journal of Molecular Biology* **247**(4), 536–540.

Myers, E. & Miller, W. (1988), "Optimal alignments in linear space", *Computer Applications in the Biosciences* **4**(1), 11–17.

Myers, G. (1992), "A four russians algorithm for regular expression pattern matching", *J. ACM* **39**(2), 432–448.

Myers, G. & Durbin, R. (2003), "A table-driven, full-sensitivity similarity search algorithm", *Journal of Computational Biology* **10**(2), 103–117.

Ning, Z., Cox, A. J. & Mullikin, J. C. (2001), "SSAHA: a fast search method for large DNA databases", *Genome Research* **11**(10), 1725–1729.

Orcutt, B. & Barker, W. (1984), "Searching the protein database", *Bulletin of Mathematical Biology* **46**, 545–552.

Pearson, W. & Lipman, D. (1985), "Rapid and sensitive protein similarity searches", *Science* **227**(4693), 1435–1441.

Pearson, W. & Lipman, D. (1988), "Improved tools for biological sequence comparison", *Proceedings of the National Academy of Sciences USA* **85**(8), 2444–2448.

Smith, T. & Waterman, M. (1981), "Identification of common molecular subsequences", *Journal of Molecular Biology* **147**(1), 195–197.

States, D. J. & Agarwal, P. (1996), Compact encoding strategies for dna sequence similarity search, *in* "Proceedings of the Fourth International Conference on Intelligent Systems for Molecular Biology", AAAI Press, pp. 211–217.

Wilbur, W. & 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. & Zobel, J. (1996), Indexing nucleotide databases for fast query evaluation, *in* "EDBT '96: Proceedings of the 5th International Conference on Extending Database Technology", Springer-Verlag, London, UK, pp. 275–288.

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

Williams, H. E. & Zobel, J. (2002), "Indexing and retrieval for genomic databases", *IEEE Transactions on Knowledge and Data Engineering* **14**(1), 63–78.

Witten, I. H., Moffat, A. & Bell, T. C. (1999), *Managing Gigabytes: Compressing and Indexing Documents and Images*, Morgan Kaufmann Publishers, San Francisco, CA.

Wu, S., Manber, U. & Myers, E. W. (1996), "A subquadratic algorithm for approximate limited expression matching", *Algorithmica* **15**(1), 50–67.

Zhang, Z., Berman, P. & Miller, W. (1998), "Alignments without low-scoring regions", *Journal of Computational Biology* **5**(2), 197–210.

Zhang, Z., Schwartz, S., Wagner, L. & Miller, W. (2000), "A greedy algorithm for aligning DNA sequences", *Journal of Computational Biology* **7**(1–2), 203–214.