BIOINFORMATICS APPLICATIONS NOTE Vol. 23 no. 21 2007, pages 2949–2951
doi:10.1093/bioinformatics/btm479

Sequence analysis

Improved BLAST searches using longer words for protein seeding

Sergey A. Shiryev, Jason S. Papadopoulos, Alejandro A. Schäffer and Richa Agarwala*

Department of Health and Human Services, National Center for Biotechnology Information, National Institutes of Health

Received on August 8, 2007; revised on September 13, 2007; accepted on September 19, 2007

Advance Access publication October 6, 2007

Associate Editor: Thomas Lengauer

ABSTRACT

Motivation: The `blastp` and `tblastn` modules of BLAST are widely used methods for searching protein queries against protein and nucleotide databases, respectively. One heuristic used in BLAST is to consider only database sequences that contain a high-scoring match of length at most 5 to the query. We implemented the capability to use words of length 6 or 7. We demonstrate an improved trade-off between running time and retrieval accuracy, controlled by the score threshold used for short word matches. For example, the running time can be reduced by 20-30% while achieving ROC (receiver operator characteristic) scores similar to those obtained with current default parameters.

Availability: The option to use long words is in the NCBI C and C++ toolkit code for BLAST, starting with version 2.2.16 of `blastall`. A Linux executable used to produce the results herein is available at: ftp://ftp.ncbi.nlm.nih.gov/pub/agarwala/protein_longwords

Contact: richa@helix.nih.gov

1 INTRODUCTION

BLAST is a widely used set of programs that produce local alignments for input *query* sequences by searching a database of *subject* sequences. In this note, we consider the `blastp` module where the query is a protein and the database also contains proteins, and the `tblastn` module where the query is a protein and the database contains DNA sequences that are hypothetically translated. The BLAST algorithm (Altschul *et al.*, 1990, 1997) consists of three major stages:

(1) Seeding stage: subject sequences are scanned to find locations that resemble the query sequence. The output is a list of offset pairs called *seeds*. Each seed represents coordinates on the query and subject sequences.

(2) Ungapped extension stage: an initial extension for each seed is performed by comparing the amino acids on both sides of the seed. A decision is made whether the location is of interest or not. At this stage, the majority (more than 99%) of seeds are typically discarded.

(3) Gapped extension stage: each high-enough-scoring ungapped alignment becomes the starting point for a local gapped alignment that attempts to improve the score further. Gapped alignments for which the calculated expect-value is below the threshold (default is 10) are considered for the final result set.

Each stage takes ~30% of the total running time.

This note presents improvements in the seeding stage. In BLAST, seeding is implemented via a lookup table approach. When we scan a sequence, for each location we group several consecutive letters into a *word*. A numerical representation of this word becomes the entry's offset in the lookup table. The lookup table was populated in advance using words from the query sequence, so when we scan the subject and find that the word being considered corresponds to a non-empty entry in the lookup table, we have a seed. An example of seeds of length 6 is shown below:

```
Query: 127 THRHMTEFTGLDMEMAF
           T RH+ E   +D EMAF
Sbjct: 117 TTRHLNEAWSIDSEMAF
            Seed1     Seed2
```

To improve lookup performance, BLAST utilizes CPU caches by employing a small bit array structure that is consulted prior to accessing the lookup table (Cameron *et al.*, 2006). The bit array is used to avoid unnecessary (and expensive) lookups in the lookup table for entries *known* to be empty.

2 METHODS

To improve the performance of the `blastp` and `tblastn` modules, we modified the implementation of the seeding stage to reduce the number of seeds generated, thus reducing time spent in later stages. The primary concern was not with the speed of the seeding stage (although we did spend some time optimizing it), but with the number and quality of seeds produced.

An obvious way to reduce the number of seeds is to increase the number of amino acids considered, which we call *word size*, when deciding if we have a seed or not. The *baseline* code from which we started supports word size up to 5, with the default of 3 and an additional requirement that there be two seeds in close proximity (called *two-hit* in BLAST). An early version used size 4 (Altschul *et al.*, 1990). A previous study showed that the two-hit requirement with word size 3 has a better time-sensitivity trade-off than using the older single-hit rule (Altschul *et al.*, 1997). Since the current code supports word size 5, it was natural to ask whether lookup tables for word sizes 6 and 7 could be designed, and if so, how they would perform.

A simple increase in word size is not practical because of exponential growth in the size of the lookup table, e.g. for word size 7 we would have at least $20^7 = 1.3e9$ lookup table entries! To tackle this explosion,

*To whom correspondence should be addressed.

we decided to use compressed alphabets that group similar letters together into disjoint sets (Edgar, 2004). Grouping letters decreases memory requirements, and if the compressed alphabet is chosen carefully, the loss of information is less than the smaller alphabet size would suggest.

To implement the seeding method described in this article, the following subproblems must be solved: (i) choose an alphabet size; (ii) given an alphabet size, determine the compressed alphabet; (iii) given a compressed alphabet, decide how to populate the lookup table; (iv) given a location, decide how to score a candidate seed; (v) decide how to evaluate performance of the algorithm.

2.1 Choice of compressed alphabets sizes

The size of the CPU cache was the primary factor in deciding the size of the alphabet to use for a given word size. On modern CPUs with traditional architecture, the size of the level 2 cache is usually somewhere between 512 KB and 4 MB, so 22–25 bits of address space is an estimate for the 'working set' of the bit array and this in turn gives us the number of entries in the lookup table.

Because the access pattern is highly non-uniform (due to the non-uniform frequency distribution of amino acids), the actual size of the bit array can be slightly larger than available cache while still being very effective. For example, words starting with many 'W' characters are extremely rare. Thus, we can rely on the CPU's cache control mechanism to adapt automatically to the incoming stream of letters to keep the most frequent combinations in cache. An additional benefit of a non-uniform access pattern is that performance changes gracefully as the amount of available CPU cache changes.

For word sizes 6 and 7, we chose 15-letter and 10-letter alphabets, respectively. This setup leads to a 1.4 MB array of bits for 6-letter words and a 1.2 MB bit array for 7-letter words.

2.2 Choice of compressed alphabets

Using compressed alphabets in (Edgar, 2004) as the starting point, we tested various compressed alphabets and selected the following ones:

IJLMV AST BDENZ KQR G FY P H C W (1)

ST IJV LM KR EQZ A G BD P N F Y H C W (2)

Each string of consecutive letters without spaces represents one *letter set* in the compressed alphabet. Alphabet (1) is the 'SE-V(10)' alphabet in Edgar (2004) with ambiguity characters added. For the 15-letter alphabet (2), however, we added one more letter set to Edgar's 'SE-B(14)'.

Because the bit array may not fit into CPU cache on some machines, using an alphabet's matrix entropy (Altschul, 1991) as the sole selection criterion is not sufficient. In practice, alphabet (2) demonstrated well-balanced performance. To further reduce the number of cache misses, we sorted letter sets within the alphabet in descending order of background probabilities.

2.3 Populating the lookup table

We can populate the lookup table with encoded words from the query, and then scan the subject looking for exact matches. Unfortunately, such an approach is not sensitive enough.

To increase sensitivity, we employed the same threshold-based mechanism as used in protein BLAST's lookup table population. Not only is the exact encoding of a word included in the lookup table, but also all of the word's *neighbors*, words that match the original word with a score not lower than a given threshold.

The matching score between two words is the sum of substitution scores for each pair of corresponding letters. The score for aligning one letter with another is taken from a *score matrix*.

Because substitution score usage is central to the algorithm, we cautiously decided not to use BLOSUM62 (Henikoff and Henikoff, 1992) at the standard precision because that precision may be too low. Instead we used a scaled version of the integer scores obtained by rounding $C \ln r_{ij} = C \ln(q_{ij}/(p_i p_j))$, where C is a constant, q_{ij} is the probability of aligning amino acid i with amino acid j and p_i, p_j are the background probabilities of i and j, respectively (Schäffer *et al.*, 2001).

Using a high-precision threshold allowed us much greater freedom in selecting the sensitivity of the BLAST algorithm. For example, in baseline `blastp`, going from threshold value 12 to 11 almost doubles the running time and there is no allowed threshold in between. Now, we can select any value and thereby adjust performance to meet time and computing power constraints. The new threshold is entered as a floating point number, treated on the same scale as the current low-precision threshold, and then internally converted to a scaled-up integer.

2.4 Calculating matching score

Converting the query and subject to the compressed alphabet and using a square score matrix (10×10 or 15×15) produced subpar results. Using a square matrix loses information in both query and subject sequences, leading to a high proportion of seeds that are discarded in later stages.

To prioritize potential seeds, we decided to weight the matching score by the probability of a match. For example, the matching score for 'A' to the 'AST' letter set is calculated based on the frequency ratio of 'A' to each one of the three letters weighted by the conditional probability of that letter. This additional weighting is needed because at the time of lookup table population, we do not know which exact letters we will encounter—the compressed alphabet obscures that information. We used background probabilities from Robinson and Robinson (1991) to calculate conditional probabilities, e.g. $p(\mathrm{A}|\mathrm{A}\vee\mathrm{S}\vee\mathrm{T})$, $p(\mathrm{T}|\mathrm{A}\vee\mathrm{S}\vee\mathrm{T})$. In practice, Robinson–Robinson background probabilities produced good results, sometimes even better than using the observed letter frequencies of the database.

Mathematically, the implemented scoring matrix is 20 (plus ambiguity characters) rows by 10 or 15 columns. The score for matching amino acid i to letter set g_k, denoted by s_{ik}, is calculated using the formula:

$$s_{ik} = \frac{1}{\lambda} \ln \sum_{j \in g_k} r_{ij} p(j \mid g_k) \qquad (3)$$

where j is an amino acid that belongs to the letter set g_k, r_{ij} is the frequency ratio defined above, λ is a scaling factor and $p(j \mid g_k)$ is the group conditional probability of amino acid j and is given by $p_j/(\sum_{a \in g_k} p_a)$.

The example in the Introduction section has two seeds. The query is `d1eova2` and the subject is `d1b8aa2` from ASTRAL (Chandonia *et al.*, 2004). Seed1, a 6-letter seed with mismatch at position 1 and 5, illustrates that we can have multiple mismatches at arbitrary positions. Seed1 has a scaled score of 19.1 exceeding the recommended threshold of 18.8.

2.5 Performance evaluation

To evaluate result quality, we used SCOP/ASTRAL (Chandonia *et al.*, 2004). The SCOP/ASTRAL dataset provides a gold standard from which it is possible to identify true and false positives in the output and compute 'receiver operator characteristic' (ROC) scores (Gribskov and Robinson, 1996). We report ROC_{10000} scores, meaning that matches for all queries are combined and ranked by expect-value up to the first 10 000 false positives. Program configuration A is considered to have higher quality output than configuration B if the ROC score for A is higher than for B. We used sequences of $<40\%$ identity in the current ASTRAL version 1.71. True positives are query-subject pairs in the same superfamily. We ignored self-hits and we used as queries, the 7218 sequences that have at least two sequences in the same superfamily.

Unfortunately, ASTRAL is not large enough to adequately represent the more commonly used nr and Swiss-Prot databases. We therefore decided to use a hybrid measurement approach. We measured ROC scores produced by using different values of the threshold. We then ran

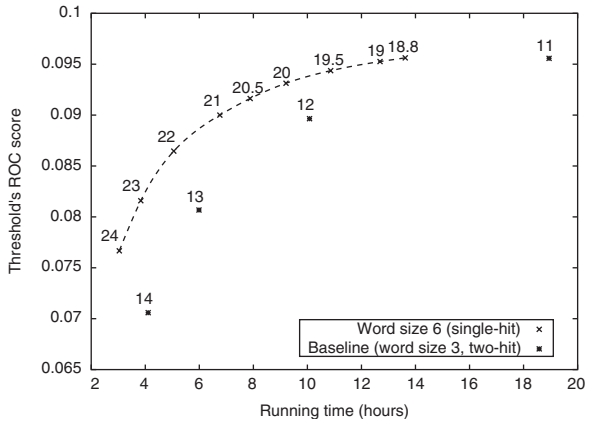

Fig. 1. Trade-off between `blastp` running time and ASTRAL 1.71 ROC_{10000} scores. Points for word size 3 are not connected as high-precision thresholds are allowed only for word sizes 6 and 7. The numbers next to the points in the graph are the score thresholds.

our software with these thresholds, ASTRAL data as the set of queries, and the first volume of nr (4 August 2007 with 2 671 244 sequences and 903 233 208 total letters) as the database to measure running time. We assert that the overall sensitivity of the algorithm is set by the threshold and as such does not depend on the size of the database. The assertion held when we used different ROC score datasets and when we compared output quality on nr using different seeding strategies, but set to the same sensitivity level.

In particular, the high precision for the threshold value allowed us to match (by trial and error) the ROC score of our algorithm to the ROC score of the baseline BLAST. This allowed direct comparison of running times between these two programs. The performance evaluation method described here has proven to be quite sensitive—it allowed us to see minute differences in speed between different alphabets and word sizes.

Results of the performance evaluation of `blastp` are shown in Figure 1. It can be seen that using word size 6 improves performance of the program. Baseline `blastp` with default score threshold 11 took 18 h 56 min, while `blastp` with word size 6 and threshold 18.8 took 13 h 37 min. Performance results for `tblastn` are qualitatively similar to `blastp`, although the ROC score test set we used is small (Gertz *et al.*, 2006).

The primary reason for this improvement is the much smaller number of seeds generated in the first stage of the program. Compared to the baseline configuration: for threshold 11 and its equivalent, for query and subject used for the timing tests above, we have 1.3e11 seeds (word size 6) versus. 2.3e12 seeds (baseline), i.e. 17 times fewer seeds to be examined. In our tests of `blastp`, word size 6 worked better than 7.

3 IMPLEMENTATION

The results reflect performance on a 32-bit gcc 3.4.2 compiled application (single-threaded) run on a 64-bit 3 GHz Intel Xeon 5160, 4 MB L2 cache and 8 GB RAM. We compiled on a 32-bit machine for portability; compiling on the 64-bit machine improves running time. The following command line parameters were used to run baseline program: '`blastall -p blastp -C 2 -m 8`'; for word size 6 runs: '`blastall -p blastp -C 2 -m 8 -W 6 -P 1 -f 18.8`'. Option `-C 2` is for compositionally adjusted statistics (Altschul *et al.*, 2005; Gertz *et al.*, 2006); `-m 8` is the output format; `-W 6` is the word size (default is 3); `-P 1` to force one-hit extensions (default is two-hit); `-f 18.8` is the threshold.

4 DISCUSSION

Longer seeds would be expected to improve sensitivity of BLAST searches, but memory limitations make longer seeds difficult to implement with a good running time. Our implementation shows that using compressed alphabets can overcome the memory limitations for word sizes 6 and 7, while retaining overall sensitivity of the algorithm. For output quality comparable to the baseline BLAST, the running time decreases by 20–30%. The implementation enables a fine-scale trade-off between running time and output quality. Preliminary experimental results for larger (eight letters or more) word sizes were not promising.

BLAST can take as input multiple queries (e.g. in a single FASTA-formatted file) and can scan the database for several queries simultaneously to save time. Therefore, for any usage with many queries, the effective query size should be large. However, when the effective size of the query or the database is small, our approach cannot yet compete with baseline BLAST or deterministic finite automaton seeding (Cameron *et al.*, 2006) due to additional setup time required for building the lookup table and to slightly slower scanning speed. Our method is targeted for 'bulk-processing' applications that align large queries to large databases, for which our implementation outperforms both baseline `blastall` and the implementation of Cameron *et al.* (2006).

ACKNOWLEDGEMENTS

This research was supported by the Intramural Research Program of the National Institutes of Health, National Library of Medicine. Thanks to Stephen Altschul, E. Michael Gertz and Aleksandr Morgulis for helpful comments.

Conflict of Interest: none declared.

REFERENCES

Altschul,S.F. (1991) Amino acid substitution matrices from an information theoretic perspective. *J. Mol. Biol.*, **219**, 555–565.

Altschul,S.F. *et al.* (1990) Basic local alignment search tool. *J. Mol. Biol.*, **215**, 403–410.

Altschul,S.F. *et al.* (1997) Gapped BLAST and PSI-BLAST. a new generation of protein database search programs. *Nucleic Acids Res.*, **25**, 3389–3402.

Altschul,S.F. *et al.* (2005) Protein database searches using compositionally adjusted substitution matrices. *FEBS J.*, **272**, 5101–5109.

Cameron,M. *et al.* (2006) A deterministic nite automation for faster protein hit detection in BLAST. *J. Comput. Biol.*, **13**, 965–978.

Chandonia,J.-M. *et al.* (2004) The ASTRAL compendium in 2004. *Nucleic Acids Res.*, **32**, D189–D192.

Edgar,R.C. (2004) Local homology recognition and distance measures in linear time using compressed amino acid alphabets. *Nucleic Acids Res.*, **32**, 380–385.

Gertz,E.M. *et al.* (2006) Composition-based statistics and translated nucleotide searches: improving the TBLASTN module of BLAST. *BMC Biol.*, **4**, 41.

Gribskov,M. and Robinson,N.L. (1996) Use of receiver operating characteristic (ROC) analysis to evaluate sequence matching. *Comput. Chem.*, **20**, 25–33.

Henikoff,S. and Henikoff,J.G. (1992) Amino acid substitution matrices from protein blocks. *Proc. Natl Acad. Sci. USA*, **89**, 10915–10919.

Robinson,A.B. and Robinson,L.R. (1991) Distribution of glutamine and asparagine residues and their near neighbors in peptides and proteins. *Proc. Natl Acad. Sci. USA*, **88**, 8880–8884.

Schäffer,A.A. *et al.* (2001) Improving the accuracy of PSI-BLAST protein database searches with composition-based statistics and other refinements. *Nucleic Acids Res.*, **29**, 2994–3005.