

# Highly Scalable and Accurate Seeds for Subsequence Alignment

Abhijit Pol

Tamer Kahveci

Department of Computer and Information Sciences and Engineering,  
University of Florida, Gainesville, FL, USA, 32611  
{apol, tamer}@cise.ufl.edu

## Abstract

*We propose a method for finding seeds for the local alignment of two nucleotide sequences. Our method uses randomized algorithms to find approximate seeds. We present a dynamic index to store the fingerprints of  $k$ -grams and a highly scalable and accurate (HSA) algorithm to incorporate randomization into process of seed generation. Experimental results show that our method produces better quality seeds with improved running time and memory usage compared to traditional non-spaced and spaced seeds. The presented algorithm scales very well with higher seed lengths while maintaining the quality and performance.*

## 1 Motivation

Locating *similar* subsequences between a query sequence and the sequences in a database is one of the most fundamental problems in bioinformatics. This is also known as the *local alignment* problem. Local alignment matches pairs of letters between two subsequences. A score is then assigned for each match. Every mismatch and gap are penalized with appropriate mismatch, gap-open, and gap-extend penalties. The score of the local alignment is then computed as the sum of all such scores and penalties.

One of the earliest algorithms to solve the local alignment problem is the Smith-Waterman algorithm [13] (SW). SW uses dynamic programming to find all local alignments. Both time and space complexity of the SW method is  $O(mn)$ , where  $m$  and  $n$  are the lengths of the two sequences aligned. Quadratic complexity makes the SW algorithm unpractical as the sequences get longer and as the sequence databases grow. A number of efficient heuristic methods were developed to solve the local alignment problem using less space and time. One of the popular seed-based approaches for local alignment is BLAST [20]. BLAST works in two phases: (i) search phase - it first finds the  $k$ -grams ( $k$  letter subsequence) in the target sequence that have a perfect match in the query sequence with the help of a hash table. The region is popularly referred as *seed*. (ii) alignment phase - it stitches and extends the seeds

found in the first phase into significant alignments. For example, BLASTN (BLAST for nucleotide sequences) first finds exact matches of seed length 11. It then extends these matches into local alignments, and accepts only those alignments which satisfy a user-defined score cutoff.

There is a tradeoff between execution speed and sensitivity for selected seed length,  $k$ . If  $k$  is large, the search phase may miss high scoring alignments that do not have  $k$  consecutive matches. On the other hand, increasing  $k$  will produce fewer seeds for the alignment phase. Thus, the overall computation time decreases. In other words,  $k$  provides a tradeoff between performance and quality. Several approaches including PatternHunter (PH) [7] and megaBLAST [18] address this problem by using spaced seeds or different length seeds. Although, these methods aim at improving the performance and seed quality, the underlying problem with BLAST's seed selection remains untouched. If  $k$  is very large, the amount of memory to store the hash table may exceed available resources. The space taken-up by hash data structure is exponential in  $k$ . Specifically, for a sequence of a nucleotides letters, the hash size calculates to  $4^k$ . For  $k = 24$ , the hash table size is one petabyte even if only four bytes are used per hash table entry. This is many orders of magnitude larger than the capacity of today's computers. Such large values of  $k$  are commonly used by tools that analyze very large datasets. For example  $k = 24$  and 28 for Arachne [6] and megaBLAST [18] respectively.

A simple solution to the seed length scalability problem is not to use the hash table data structure, but rather keep only those  $k$ -grams from query sequence which are unique. This method keeps the  $k$ -grams that appear in the query sequence in a sorted list, in alphabetical order. Thus, every  $k$ -gram in the query is stored only once, and every letter in the query is stored at most  $k$  times. On the other hand, using sorted list increases the cost of locating a  $k$ -gram since its position in the sorted list is not known. Binary search incurs logarithmic search time for locating  $k$ -grams (compared to constant time search cost of hash tables).

**Our contribution:** Although the techniques of randomized

algorithms have been extensively used for efficient string matching [15], to our knowledge, no one has applied them for alignment of biological sequences. In this paper, we describe a new, *highly scalable and accurate* (HSA) algorithm and a dynamic index structure for finding seeds. Our method employs randomization technique for choosing initial seeds. HSA is more efficient and accurate than the existing tools, such as BLAST, that use a fixed seed size. Unlike existing methods, HSA provides worst-case guarantees on either its efficiency or its accuracy or both. In particular HSA has the following advantages over existing tools:

- HSA is highly scalable in the sense that even if we choose large seed lengths for exact matching, we can guarantee to work with limited amount of memory and still produce high quality seeds.
- HSA provides guarantees on accuracy of our results. Specifically, it finds all exactly matching seeds. It also computes an upper bound, such as  $10^{-3}$ , on the probability of finding an approximate seed accidentally. This probability can be considered as the error probability.
- Given a desired upper bound on the error probability, HSA can also compute the amount of memory required to guarantee that quality. It dynamically adapts its data structure to fit into that memory.
- HSA also provides upper bounds on the running times of our algorithms to find seeds for given allowable error probability. The time needed to find the seeds for HSA is less than that of the competing methods.

A rigorous benchmark of the HSA structure demonstrates its superiority over other alternatives. According to our experiments, HSA produces better quality seeds with improved running time and memory usage compared to BLAST and spaced seeds. The presented algorithm scales very well with higher seed lengths while maintaining the quality and performance.

**Paper organization:** The rest of the paper is organized as follows. Section 2 presents background on randomized algorithms. Section 3 discusses our index structure and algorithms. Section 4, presents quality and performance results. Section 5 discusses the related work. We end with a brief discussion in Section 6.

## 2 Randomized algorithms and fingerprinting

Assume that sequences are defined over a fix set of alphabet  $\Sigma = \{\alpha_1, \alpha_2, \dots, \alpha_\sigma\}$ , where  $\sigma$  is the alphabet size. For nucleotide sequences  $\Sigma = \{A, C, G, T\}$ . Each letter in this alphabet can be represented using  $\log \sigma$  bits. For example, for nucleotides A = 00, C = 01, G = 10, and T = 11. Given a  $k$ -gram  $a = a_1 a_2 \dots a_n$ , the hash value for  $a$ ,  $h(a)$ , can be computed by concatenating the bit representations of its letters. Therefore,  $k \log \sigma$  bits are needed to store the hash value of a  $k$ -gram. For example, for  $a = ACGT$ ,  $h(a) = 00011011$ . Thus,  $h(a) = 27$ . The hash value of a  $k$ -gram varies between zero and  $\sigma^k$ . Two  $k$ -grams have the

same hash value if and only if they are equal.

Fingerprint of a  $k$ -gram  $a$  is defined as

$$F_p(a) = h(a) \pmod p,$$

where  $p$  is a given prime number. If  $p \geq \sigma^k$ , then the fingerprint of  $a$  will be equal to its hash value. Otherwise,  $F_p(a)$  may be smaller than  $h(a)$ . Fingerprinting suggests the use of fingerprints instead of hash values in order to find similar  $k$ -grams. Thus, we call two  $k$ -grams  $a$  and  $b$  similar if  $F_p(a) = F_p(b)$ . The advantage of using fingerprints instead of hash values is that only  $O(\min\{\log p, \sigma^k\})$  bits are needed to store a fingerprint. For small  $p$ , fingerprints require much less space than hash values.

One can show that if  $a = b$  then  $F_p(a) = F_p(b)$ . Therefore, fingerprints find all the true positive matches. On the other hand different  $k$ -grams may have the same fingerprint. That is,  $F_p(a) = F_p(b)$  does not imply the equality of  $a$  and  $b$ . Therefore, fingerprints may produce false positive matches. This is also known as the *adversary problem*: Given a fixed prime number  $p$  and a database sequence, an adversary can come up with a query sequence that will produce many false positives by choosing query  $k$ -grams such that they will have different hash values than the database  $k$ -grams but same fingerprints.

The adversary problem can be solved by choosing  $p$  at random for every query instead of having a fixed  $p$ . Next, we discuss how bad this strategy can go. Assume that  $c = |h(a) - h(b)|$ ,  $a$  is incorrectly classified as similar to  $b$  only when  $c > 0$  and  $p$  divides  $c$  (i.e.,  $a$  and  $b$  are different but they have the same fingerprint.) Then the question boils down to the number of prime numbers that can divide  $c$ . If  $c$  has a large number of prime divisors, then it is more likely that a randomly chosen  $p$  will divide  $c$ . *Prime Number Theorem* states that for any number  $\tau$ , the number of primes less than  $\tau$  is asymptotically  $\frac{\tau}{\ln \tau}$ . It is also known from number theory that, the number of *distinct prime divisors* of any number less than  $2^n$  is at most  $n$ . Thus, given an upper bound  $\tau$ , if the prime number  $p$  is chosen randomly among the primes less than  $\tau$ , then the probability of returning a false positive (i.e., error probability) can be computed as

$$Pr[F_p(a) = F_p(b) | a \neq b] \leq \frac{2k}{\tau / \ln \tau} \quad (1)$$

This can be explained as follows. The difference of two hash values  $c = |h(a) - h(b)|$  is at most  $2^k$  for  $k$ -grams composed of nucleotides since  $0 \leq h(a) < 4^k, \forall a$ . Thus  $c$  can have at most  $2k$  prime divisors. In order to have a false positive, one of these prime numbers need to be chosen among all the  $\frac{\tau}{\ln \tau}$  primes. If we choose  $\tau = t2k \log t2k$ , for some value  $t$ , then we have following error probability:

$$Pr[F_p(a) = F_p(b) | a \neq b] \leq O\left(\frac{1}{t}\right) \quad (2)$$

If we choose  $t = 2k$  the error probability becomes  $O(1/2k)$ .

We now extend our discussion on fingerprinting to the problem of subsequence matching. Let  $X = x_1x_2 \cdots x_{m+k-1}$  be nucleotide sequence in the database and  $b$  be a  $k$ -gram from query sequence.  $X$  contains  $m$   $k$ -grams. If we assume that the signatures of these  $k$ -grams are randomly distributed, then the probability that a false match occurs for any of them is  $O((m2k \log \tau)/\tau)$ . If we choose  $\tau = m^2 2k \log m^2 2k$ , that gives us:

$$Pr[\text{at least one false positive returns}] \leq O(1/m) \quad (3)$$

### 3 The HSA algorithm

In this section, we consider the problem of incorporating described randomized algorithm and fingerprinting technique to find seeds. The solution is discussed by way of presenting a new highly scalable and accurate algorithm and a dynamic index structure.

**Notations and data structure:** The query sequence of size  $m$  is denoted by  $Q = q_1q_2 \cdots q_m$ , and a database sequence of size  $n$  is denoted by  $S = s_1s_2 \cdots s_n$ . The  $k$ -gram starting at the  $j$ th character of the query sequence is denoted by  $Q(j)$  and likewise for the database sequence by  $S(j)$ . The integer representation of the  $k$ -gram starting at the  $j$ th position in the query is denoted by  $h(Q(j))$  and likewise for the database sequence by  $h(S(j))$ .  $p$  is the prime number and  $\tau$  is the threshold for selection of  $p$ .  $F_p$  is the mod fingerprint function used and  $k$  is the length of the seed.

HSA index is essentially an array, size of which is dynamically decided in its initialization. Each entry in the array is referred as a bucket and stores the fingerprint of a query  $k$ -gram. Associated with each bucket is the list of integers indicating positions of the  $k$ -grams in query sequence with that fingerprint value.

**HSA method and seed generation:** We build an index structure to store different  $k$ -grams of an input query sequence. The size of the index structure is dynamically decided based on the prime number  $p$  selected for fingerprinting. If  $p$  is small, we use direct hashing and store the fingerprints of all possible  $k$ -grams in a hash table. Thus, the hash table has  $p$  entries. If  $p$  is too large to fit the hash table into allowed memory, we choose sorted list option for unique  $k$ -grams of a query sequence. In this case, the number of entries of the sorted list is equal to the number of unique fingerprint values for the  $k$ -grams in the query sequence. Both of these index structures store pointers to all the  $k$ -grams in the query sequence according to their fingerprint values. Our dynamic index structure supports various methods for efficient storage and searching of fingerprints. The methods are sketched in Figure 1.

In the *Initialize* method we mainly select the random prime number for fingerprinting and then dynamically set the size and type of our structure. We start by calculating the threshold  $\tau = m^2 2k \log m^2 2k$  for a given seed length

---

```

/* TYPE = how HSA is structured */
/* SIZE = number of buckets */
/* TABLE = an array of buckets */
/* LIST = an array of integers for each bucket */
/* k = k-gram size */
/* m = length of query sequence */
/* maxMem = maximum allowable memory */
Method Initialize ( $k, m, maxMem$ )
     $\tau := m^2 2k \log m^2 2k$ ;
    Randomly select a prime  $p < \tau$ ;
     $SIZE := \min(p, 4^k, maxMem)$ ;
    if  $SIZE \geq maxMem$  then  $TYPE := SORTED$ ;
    else  $TYPE := HASH$ ;
    return  $p$ ;

/* fp = fingerprint value of the k-gram */
/* pos = position of the k-gram */
Method Insert ( $fp, pos$ )
    if  $TYPE = HASH$  then
        Add  $pos$  to  $TABLE[fp \bmod SIZE].LIST$ ;
    else
         $ptr := Search(fp)$ ;
        Add  $pos$  to  $TABLE[ptr].LIST$ ;
        if  $TABLE[ptr].LIST = 1$  then
             $Sort(TABLE)$ ;

Method Search ( $fp$ )
    if  $TYPE = HASH$  then
        if  $TABLE[fp \bmod SIZE].LIST \neq \emptyset$  then
            return ( $fp \bmod SIZE$ );
    else  $ptr := Binary Search(fp)$ ;
    return  $ptr$ ;

```

---

**Figure 1.** HSA Methods. *Initialize* method sets the type of the index structure based on memory limitations. *Insert* method inserts a  $k$ -gram into an existing index structure. *Search* method locates the positions of the  $k$ -grams that have a certain fingerprint.

$k$  as described in Section 2. A prime  $p$  is then selected at random less than  $\tau$ . Next, we choose the number of buckets in the HSA structure as the smallest of three choices: First, prime  $p$  itself. Since the fingerprint is the *mod* function with respect to  $p$ , we are guaranteed that the fingerprint function will not exceed value  $p$ . The second choice is number of all possible  $k$ -grams, that is  $4^k$ . The last choice is simply the size of maximum allowable memory. If we choose from the first two options, we can structure our buckets for direct hashing. If we pick the third option we build sorted list on the unique query fingerprints. Note that, we assume that all the unique  $k$ -grams from the query sequence fits in allowable memory. If the query sequence is too large to fit in memory, we need to lower the value of  $\tau$  to stay within memory limitations.

The other methods, *Insert* and *Search*, directly follow from the type of selected structure. In case of direct hashing, simply insert the position of the  $k$ -gram into the bucket suggested by a hash function, whereas for the sorted type

we perform binary search to decide the target bucket.

Figure 2 presents how the seeds are generated. The algorithm starts by reading the query sequence. A random prime  $p$  is selected based on input parameters. The query is then scanned and the index structure is created on the fingerprints of its  $k$ -grams. Next, the database sequences are scanned and a fingerprint is obtained for each  $k$ -gram from the database sequences. The same fingerprint function is used for the database sequences as the query sequence. Each database fingerprint is then searched in the index structure for a match. For every match, two positions, one in the database sequence and one in the query sequence are returned as a seed.

---

```

/* k = seed length */
/* maxMem = max allowable memory */
/* Q = a query sequence */
/* S = a database sequence */
/* m = |Q|, n = |S| */
/* HSA = HSA Data structure */

Procedure Seed Generation ( $k, Q, S, maxMem$ )
   $m := LOAD(qry)$ ;
   $p := HSA.Initialize(k, m, maxMem)$ ;

  /* Scan the query */
  for  $i = 1$  to  $m - k + 1$ ; do
     $F_p(Q(i)) := h(Q(i)) \bmod p$ ;
     $HSA.Insert(F_p(Q(i)), i)$ ;

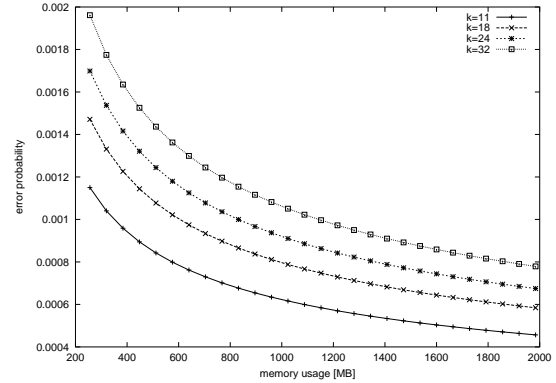
  /* Scan the database */
   $n := LOAD(db)$ ;
  for  $j = 1$  to  $n - k + 1$ ; do
     $F_p(S(j)) := h(S(j)) \bmod p$ ;
     $idx := HSA.Search(F_p(S(j)))$ ;
    if  $idx > 0$  then
       $HSA.Output(idx, j)$ ;

```

---

**Figure 2.** HSA Algorithm - Seed Generation. The algorithm takes a query sequence, a database sequence, seed size, and allowed memory size as input. It returns the positions of the matching  $k$ -grams.

**Of primality and performance hiccups:** It is important for the correctness of proposed randomized algorithm that we randomly select a prime number less than the threshold  $\tau$ . However, we should remark that the task of picking a random prime is non-trivial, primarily because verifying the primality of a number is difficult. One can again make use of randomized algorithms to time efficiently choose a random prime [15]. Also recently, the primality problem has

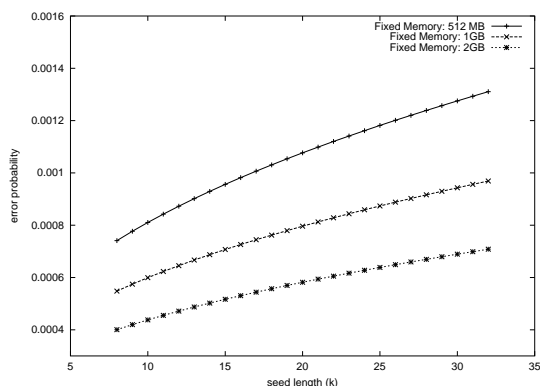


**Figure 3.** Error Probability vs Memory Used

been shown to be in P class [23], and the proposed fast algorithm can be used to speed-up primality testing. In our context, since we generate a random prime once for each query, the prime selection penalty is not overwhelming. One can avoid computing a prime number by keeping the set of all the primes up to a large number. For a given query, a prime number is then selected randomly from this set.

Another important parameter for our algorithm is  $\tau$ . The value of  $\tau$  upper bounds the selected prime. As stated in the algorithm description, we have three choices for the index structure size: prime  $p$ ,  $4^k$ , and  $maxMem$ , i.e., memory limit. Clearly, every time we get a prime smaller than the last two values, we not only build a smaller structure, but also speed-up fingerprint search. Thus choosing a small value for  $\tau$  improves both performance and memory usage. On the other hand, as mentioned in the Section 2, error probability directly depends on the value of  $\tau$ . The smaller the value of  $\tau$ , the more we are prone to an error, and vice versa. In our algorithm, we used  $\tau = m^2 2k \log m^2 2k$ , to bound our error probability to  $O(1/m)$ . Clearly, the  $\tau$  and thus the error probability depends of value of seed length,  $k$  and query length  $m$ . Since we are assuming that our query always fits into memory, we can see memory size as an upper bound for  $m$ . Thus, we can conclude that for given fixed  $k$ , the error probability decreases with more available memory. Also, for a given fixed memory size, the error probability of HSA increases with increasing  $k$ . The theoretical plots supporting this argument are shown in Figures 3 and 4.

Figure 3 shows that most of the improvement on the error probability is obtained by increasing the memory usage by less than 200 MB. Although the error probability drops by further increasing the size of the index structure, the improvement is smaller. Figure 4 shows that for a fixed memory size, the error probability of HSA increases gradually as seed length increases. This is a very important observation since static hash table methods used by tools such as BLAST are too large for practical purposes for seed size greater than 15. Another important observation that follows



**Figure 4.** Error Probability vs Seed Length

these figures is that the error probability in all these graphs are too low. Thus, for a given  $k$ -gram, it is not very likely that HSA will produce many erroneous results.

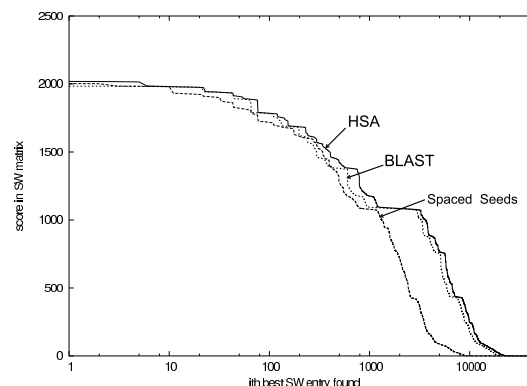
## 4 Experimental evaluation

In this section, we present a set of experiments aimed at benchmarking the performance of our algorithm to find initial exact match seeds. Using the C++ programming language we have implemented and tested a suit of query sequences for non-spaced seeds used by BLAST, spaced seeds [8], and the HSA algorithm. For first two cases we executed each query sequence as follows: We build a hash table of size  $4^k$  to store the  $k$ -grams from query if it fits in available memory. Otherwise, we build a sorted list similar to one described by the HSA algorithm.

We tested all three scenarios on two categories: comparison of similar sequences and dissimilar sequences. For both categories we performed 100 pairwise sequence comparisons as follows. We created three sets of sequences.

- *Set18a*: This dataset consists of 100 sequences of human chromosome 18, chopped into size of 4000 letters each. The sequences are then mutated (i.e., insert, delete, or modify) with 5 % probability.
- *Set18b*: This dataset consists of the same 100 sequences as *Set18a*. However, these sequences are mutated with 10 % probability.
- *Set22*: This dataset consists of 100 sequences of human chromosome 22, chopped into size of 4000 letters each. The sequences are mutated 5 % probability.

The  $i$ th sequences from *Set18a* and *Set18b* differ by at most 15 % of their letters. For comparison of similar sequences, we used these two datasets. Since the sequences from *Set18a* and *Set22* are selected from different chromosomes, there is no upper bound on the difference between the  $i$ th query sequence and  $i$ th database sequence. We used these two datasets for comparison of dissimilar sequences. In each scenario, a run consisted of  $i$ th database sequence against  $i$ th similar query sequence and  $i$ th dissimilar query



**Figure 5.** Scores found for dissimilar sequences before extending seeds when  $k = 11$ .

sequence. Thus, total 200 runs were performed for measuring quality of seeds generated and time-space performance, for each: BLAST, Spaced Seed, and HSA approach. The experiments were performed on a Linux workstation having an Intel Xeon dual processor with 2.4 Ghz clock speed and a 2GB of RAM. All the datasets and source code we implemented are available from the authors upon request.

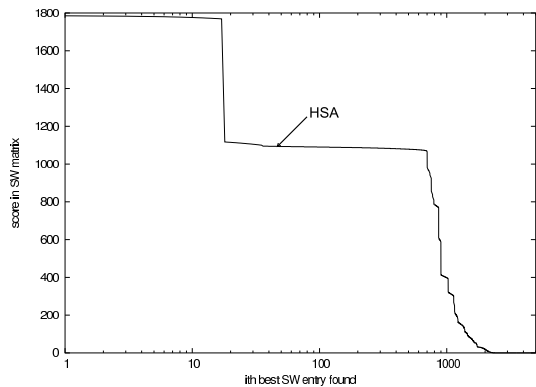
### 4.1 Quality comparison results

Our first experiment set inspects the quality of the seeds found in each of three scenarios. In order to have fair comparison, we first constructed a SW score matrix using Smith-Waterman algorithm [13] for a given database and a query sequence. We used a score of +1 for every matching letter and a penalty of -1 for every mismatch, insertion, or deletion. A seed found by any of the three algorithms maps to a set of  $k$  entries in the SW matrix. These entries essentially form a diagonal for BLAST and HSA, and a gapped diagonal for spaced seeds. On SW score matrix, each entry corresponds to a pair of letters, one from the query sequence and one from the database sequence. The value of this entry shows the best score obtained by aligning the input sequences up to and including these letters. For each of the three strategies, we find all the seeds generated by that strategy. We then plot the score of the SW matrix entries generated by these seeds in decreasing order. We performed this set of experiment on similar as well as dissimilar dataset, first with seed length  $k = 11$  and then with seed length  $k = 18$ . The plots are shown in Figures 5 to 7.

Figure 5 shows the scores found for comparison of dissimilar sequences when  $k = 11$ . The scores found by HSA seeds are always higher than both BLAST and spaced seeds. BLAST results are very close to HSA results. This is mainly because the error probability is very low for  $k = 11$ . On the other hand, HSA finds a number of high scoring seeds that BLAST fails to find. We observed similar results for the similar datasets. However the gap between HSA, BLAST,

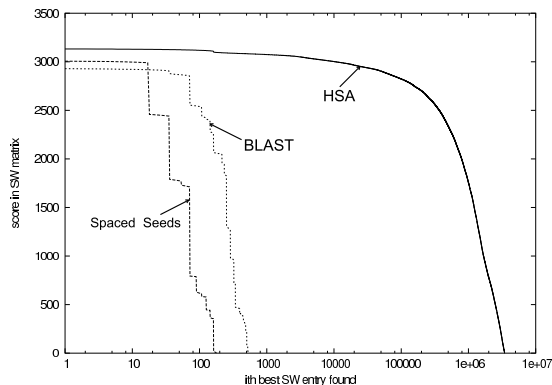
and spaced seed was negligible. We omit this graph due to space limitations.

Figures 6 and 7 show the scores found for comparison of dissimilar and similar sequences respectively when  $k = 18$ . For dissimilar sequences, BLAST and spaced seed did not find any seeds. On the other hand HSA finds many seeds. Many of the entries that HSA finds have significant score. For similar sequences, although BLAST and spaced seeds produce some seeds, HSA finds many matches that they miss. This is because as  $k$  increases, the probability of having two exactly same  $k$ -grams decreases exponentially.

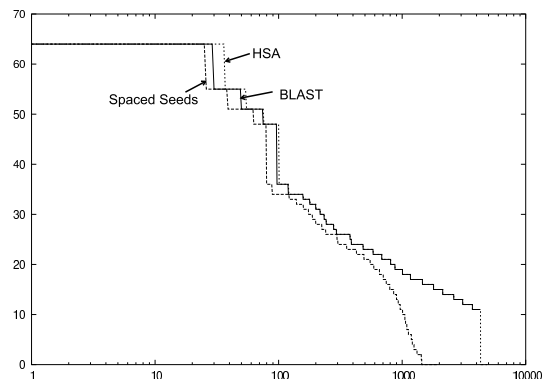


**Figure 6.** Scores found for dissimilar sequences before extending seeds when  $k = 18$ .

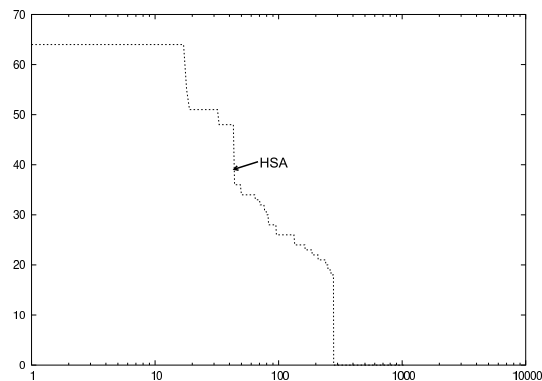
One important observation that follows from these experiments is that spaced seeds miss many high scoring entries. This is contrary to the results of spaced seed papers. We therefore performed another experiment to verify these results as follows. Instead of intersecting the seeds with SW matrix, we extended the seeds produced by each of the methods to right and left until the alignment score



**Figure 7.** Scores found for similar sequences before extending seeds when  $k = 18$ .



**Figure 8.** Scores found for dissimilar sequences after extending seeds when  $k = 18$ .

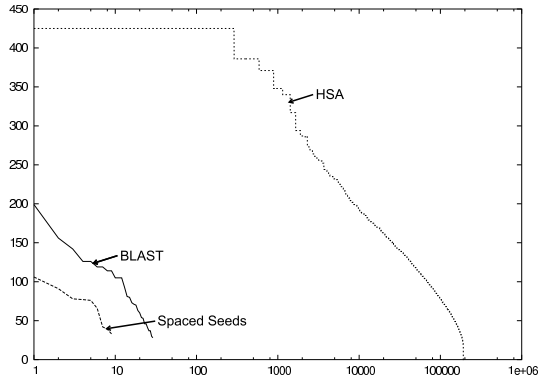


**Figure 9.** Scores found for dissimilar sequences after extending seeds when  $k = 18$ .

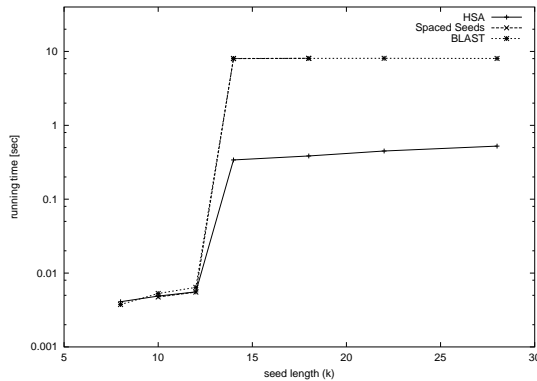
drops below by 20. We used +1 score for each match and -3 penalty for each mismatch. These are the default values BLAST uses for seed extension. We then reported the resulting score in decreasing order. Figures 8 to 10 show the scores obtained after extending the seeds for the experiments in Figures 5 to 7. These results concur with the earlier ones. Therefore, we conclude that 1) HSA can find at least as good alignments as BLAST and spaced seeds (usually much better than BLAST and spaced seeds), and 2) spaced seeds do not necessarily produce better results than non-spaced seeds. Note that comparison of spaced and non-spaced seeds is not within the scope of this paper. The fingerprinting and randomization ideas of HSA can be applied to spaced seeds as well.

#### 4.2 Performance comparison results

We compared the runtime performance as well as the memory usage of HSA, BLAST, and spaced seeds for various seed lengths ranging from 8 to 28. In case of the spaced seed scenario the execution is done only for available spaced



**Figure 10.** Scores found for similar sequences after extending seeds when  $k = 18$ .

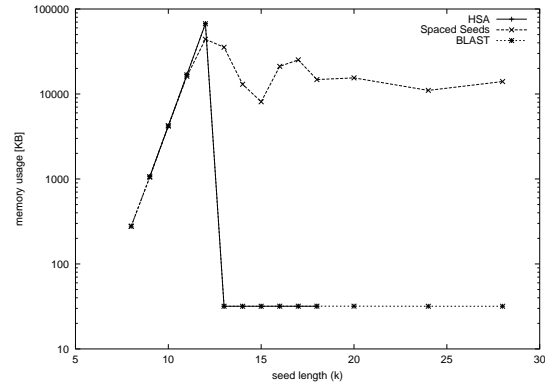


**Figure 11.** Running time of HSA, spaced seed, and BLAST to create seeds for different values of  $k$ .

seeds of length 9 to 18. We used similar datasets in this experiment.

Figure 11 shows the time to create the seeds for the three methods for various values of  $k$ . It can be seen that seed creation time increases slowly until  $k = 13$ . It then suddenly increases. This is because, at this point, the hash table becomes too large to fit in available memory. Therefore, a sorted list is maintained to create seeds. For all values of  $k$ , HSA has the lowest seed creation time. This is because if it picks a small enough prime number it uses hash table instead of sorted list even for large  $k$ .

Figure 12 shows the memory required by the index structures of the three methods for various values of  $k$ . The memory usage of all three methods increase exponentially until  $k = 13$ . This is because the size of the hash table increases exponentially with  $k$ . At  $k = 13$ , the memory usage suddenly drops for BLAST and spaced seed and increases gradually after that. This is because, they use sorted list instead of hash table after this point, since the hash table gets



**Figure 12.** Space usage of HSA, spaced seed, and BLAST for different values of  $k$ .

too large to keep in main memory. On the other hand, the memory usage of HSA remains almost same. This is because HSA efficiently uses the allowable memory to keep a hash table if it selects an appropriate prime number.

## 5 Related work

The dynamic programming solution to the problem of finding the best alignment between two strings of lengths  $m$  and  $n$  runs in  $O(mn)$  time and space [3, 13]. For large data and query strings, this technique is infeasible in terms of both time and space. Myers improved the time and space complexity to  $O(rn)$ , where  $r$  is the amount of allowed error, by maintaining only the required part of the distance matrix. [25] However, for large error rates,  $r$  is  $O(m)$ , so the complexity is still  $O(mn)$ .

Many heuristic-based search tools have been developed to align strings faster. They fall into two categories: hash-table-based tools and suffix-tree-based tools.

Some of the important hash table based tools are FASTA [7], BLAST [20], BL2SEQ [19], SENSEI [14], FLASH [10], BLASTZ [21], PatternHunter [11], BLAT [22]. The main difference between the tools is that they use different seed lengths and types, and they have different seed extensions strategies.

A number of homology search tools are based on suffix trees and derivatives. These include MUMmer [16], QUASAR [1], REPuter [4], and AVID [9]. There are two significant problems with the suffix-tree approach: (1) Suffix trees manage mismatches inefficiently. They are good for highly similar strings, but fail to recognize more distant homologies. (2) Suffix trees have a high space overhead. The suffix tree in MUMmer, for example, uses  $37n$  bytes of memory, where  $n$  is the input length [16], although with careful implementation, this can be reduced to  $8n$ . AVID [9] handles mismatches and gaps by using a variant of the Smith-Waterman algorithm once the anchors have been selected with the help of suffix trees.

CHAOS [2] indexes  $k$ -grams using a *threaded trie*, which is a cross between a suffix tree and a hash table in spirit. LAGAN [5] and DIALIGN [24, 12] employ CHAOS to find anchors for global alignment. Brudno et al. [17] also use CHAOS to find a *glocal alignment*, which is a combination of global and local alignments.

## 6 Conclusion and future work

The problem of local alignment of two biological sequences is one of the most fundamental problems in bioinformatics. Finding initial, fixed size seeds first and then stitching and extending them to obtain significant alignments remains the most popular heuristic. A seed can be obtained either by exact match or a pattern match of  $k$ -grams. In this paper, we considered the problem of detecting seeds for local alignment. We proposed to employ randomized algorithms that are suitable for pattern matching to find initial approximate seeds. The main contribution of this paper is the development of a highly scalable and accurate (HSA) algorithm and a dynamic index structure. Our method outperforms competing algorithms in terms of quality of the seeds, running time, and memory usage. Unlike existing methods, HSA scales very well with higher seed lengths, maintaining its quality as well as performance. Finally, HSA also provides guarantees in form of error probabilities (compared to exact match) and upper bound the running time and space usage during the execution.

One obvious direction for the future work is extending and incorporating randomization in the entire process of local alignment and thereby providing some probabilistic guarantees on quality of a local alignment score. It might be possible in the future to self-tune the  $\tau$  by learning from the type of target sequences. For example, in case of similar sequences a smaller  $\tau$  can be selected to potentially speed-up the execution without much affecting the quality of the seeds. Finally, the technique of randomization can also be used to find the anchors for multiple sequence alignment.

## References

- [1] S. Burkhardt and A. Crauser and P. Ferragina and H.-P. Lenhof and 'E. Rivals and M. Vingron.  $q$ -gram Based Database Searching Using a Suffix Array (QUASAR). In *International Conference on Research in Computational Molecular Biology (RECOMB)*, Lyon, April 1999.
- [2] M. Brudno and B. Morgenstern. Fast and sensitive alignment of large genomic sequences. In *Computational Systems Bioinformatics Conference (CSB)*, 2002.
- [3] S. B. Needleman and C. D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology*, 48:443–53, 1970.
- [4] S. Kurtz and C. Schleiermacher. REPuter: Fast Computation of Maximal Repeats in Complete Genomes. *Bioinformatics*, 15(5):426–427, May 1999.
- [5] M. Brudno and C.B. Do and G.M. Cooper and M.F. Kim and E. Davydov and NISC Comparative Sequencing Program and E.D. Green and A. Sidow and S. Batzoglou. LAGAN and Multi-LAGAN: Efficient Tools for Large-Scale Multiple Alignment of Genomic DNA. *Genome Research*, 13(4):721–731, 2003.
- [6] S. Batzoglou and D.B. Jaffe and K. Stanley and J. Butler and S. Gnere and E. Mauceli and B. Berger and J.P. Mesirov and E.S. Lander. ARACHNE: A Whole-Genome Shotgun Assembler. *Genome Research*, 12(1):177–189, 2002.
- [7] W.R. Pearson and D.J. Lipman. Improved Tools for Biological Sequence Comparison. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 85:2444–2448, April 1988.
- [8] K.P. Choi and F. Zeng and L. Zhang. Good Spaced Seeds for Homology Search. *Bioinformatics*, 20:1053–1059, 2004.
- [9] N. Bray and I. Dubchak and L. Pachter. AVID: A Global Alignment Program. *Genome Research*, 13(1):97–102, 2003.
- [10] A. Califano and I. Rigoutsos. FLASH: Fast Look-up Algorithm for String Homology. In *Intelligent Systems for Molecular Biology (ISMB)*, 1993.
- [11] M. Ma and J. Tromp and M. Li. PatternHunter: Faster and More Sensitive Homology Search. *Bioinformatics*, 18(0):1–6, 2002.
- [12] B. Morgenstern and K. Frech and A. Dress and T. Werner. DIALIGN: Finding Local Similarities by Multiple Sequence Alignment. *Bioinformatics*, 14(3):290–294, 1998.
- [13] T.F. Smith and M.S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, March 1981.
- [14] D.J. States and P. Agarwal. Compact Encoding Strategies for DNA Sequence Similarity Search. In *Intelligent Systems for Molecular Biology (ISMB)*, 1996.
- [15] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 2000, 2nd Edition.
- [16] A.L. Delcher and S. Kasif and R.D. Fleischmann and J. Peterson and O. Whited and D.L. Salzberg. Alignment of Whole Genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.
- [17] M. Brudno and S. Malde and A. Poliakov and C.B. Do and O. Couronne and I. Dubchak and S. Batzoglou. Glocal Alignment: Finding Rearrangements During Alignment. *Bioinformatics*, 19(90001):54i–62, 2003.
- [18] Z. Zhang and S. Schwartz and L. Wagner and W. Miller. A Greedy Algorithm for Aligning DNA Sequences. *Journal of Computational Biology*, 7(1-2):203–214, 2000.
- [19] T.A. Tatusova and T.L. Madden. BLAST 2 Sequences, A New Tool for Comparing Protein and Nucleotide Sequences. *FEMS Microbiology Letters*, pages 247–250, 1999.
- [20] S. Altschul and W. Gish and W. Miller and E. W. Meyers and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.
- [21] S. Schwartz and Z. Zhang and K.A. Frazer and A. Smit and C. Riemer and J. Bouck and R. Gibbs and R. Hardison and W. Miller. PipMaker—A Web Server for Aligning Two Genomic DNA Sequences. *Genome Research*, 10(4):577–586, April 2000.
- [22] W. J. Kent. BLAT—The BLAST-Like Alignment Tool. *Genome Research*, 12(4):656–664, 2002.
- [23] Neeraj Kayal and Nitin Saxena Manindra Agrawal. PRIMES is in P. *IIT Kanpur*, 2002.
- [24] B. Morgenstern. DIALIGN 2: Improvement of the Segment-to-Segment Approach to Multiple Sequence Alignment. *Bioinformatics*, 15(3):211–218, 1999.
- [25] E. Myers. An O(ND) Difference Algorithm and Its Variations. *Algorithmica*, pages 251–266, 1986.