# Highly Scalable and Accurate Seeds for Subsequence Alignment [*]

Abhijit Pol Tamer Kahveci
Department of Computer and Information Science 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 known as the *local alignment* problem. It matches pairs of letters between two subsequences and an appropriate score is then assigned for each match and mismatch. The score of the local alignment is then computed as the sum of all such scores.

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 and has, both, time and space complexity of is $O(mn)$, where $m$ and $n$ are the lengths of the two sequences aligned. A number of efficient heuristics that use less space and time have been developed. One such tool, BLAST [1], 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.

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 address this problem by using spaced seeds or different length seeds [12, 14]. 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 as the space taken-up by it is exponential in $k$. Such large values of $k$ are commonly used by tools that analyze very large datasets. For example $k = 24$ and 28 for Arachne [2] and megaBLAST [14] respectively.

A simple solution to the seed length scalability problem is to keep unique $k$-grams from query sequence in a alphabetically sorted list. Though, 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 compared to constant time search cost of hash tables.

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.

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 experimental 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 $\Sigma = \{\alpha_1, \alpha_2, \cdots, \alpha_\sigma\}$, 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 $= 11$.

[*]Work supported partially by NSF under grant 0347408.

Given a $k$-gram $a = a_1 a_2 \cdots a_n$, 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. 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

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

where $p$ is a given prime number [9]. 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 $\mathsf{O}(\min\{\log p, \sigma^k\})$ 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

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

If we choose $\tau = 4k^2 \log 4k^2$, for some value $t$, the error probability becomes $\mathsf{O}(1/2k)$.

We now extend our discussion on fingerprinting to the problem of subsequence matching. Let $X = x_1 x_2 \cdots x_{m+k-1}$ 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 $\mathsf{O}((m2k \log \tau)/\tau)$. If we choose $\tau = m^2 2k \log m^2 2k$, that gives us:

$$Pr[\text{at least one false positive returns}] \leq \mathsf{O}(1/m) \qquad (2)$$

## 3 The HSA algorithm

In this section, we consider the problem of incorporating described randomized algorithm and fingerprinting technique to find seeds 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 $Q = q_1 q_2 \cdots q_m$, and a database sequence of size $n$ is denoted by $S = s_1 s_2 \cdots s_n$. 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:** 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. 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 to a sorted list or a hash table. *Insert* and *Search* methods 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.

Seed generation algorithm that first reads query sequence and the index structure is created on the fingerprints of its $k$-grams. Next, the database sequences are scanned to obtain fingerprints of its $k$-grams. Each database fingerprint is

```
/* TYPE = how HSA is structured, SIZE = number of buckets */
/* TABLE = array of buckets, LIST = 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, $pos$ = position of the $k$-gram */
Method Insert ($fp$, $pos$)
    if TYPE = $HASH$ then
       Add $pos$ to TABLE [$fp$ mod 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 [$fp$ mod SIZE].LIST $\neq \emptyset$ then
          return ($fp$ mod SIZE);
    else $ptr$ := Binary Search ($fp$) ;
    return $ptr$;
```

**Figure 1.** HSA Methods. *Initialize* sets the type of the index structure. *Insert* inserts a $k$-gram into an existing structure. *Search* locates the positions of the $k$-grams given certain fingerprint.

searched in the index structure for a match. Every matched positions in the database and query sequence are returned as seeds. For the complete algorithm see [**?**].

One important parameter for our algorithm is $\tau$. The value of $\tau$ upper bounds the selected prime. As stated before, 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 $k$.

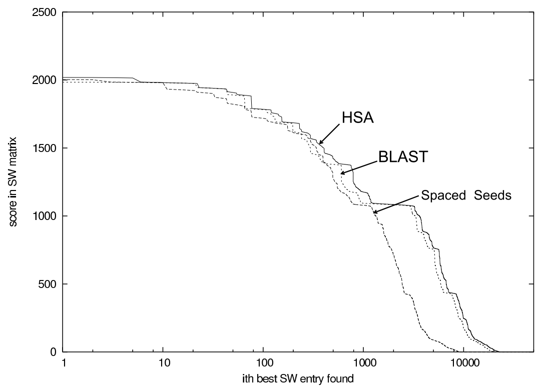

**Figure 2.** Scores for dissimilar seq. ($k = 11$) before extending.

## 4 Experimental evaluation

In this section, we present a set of experiments aimed at benchmarking the performance of our algorithm. Using the C++ programming language we have implemented and tested a suit of query sequences for seeds used by BLAST, spaced seeds [5], 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:* Same as *Set18a*, but is mutated with 10 % probability.

- *Set22:* Same as *Set18a*, but for chromosome 22.

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 sequence giving total 200 runs. The experiments were performed on a Linux workstation with 2.4 Ghz clock speed and a 2GB of RAM.

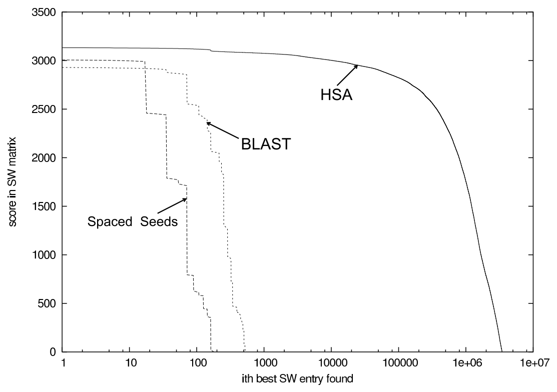

**Figure 3.** Scores for similar seq. ($k = 18$) before extending.

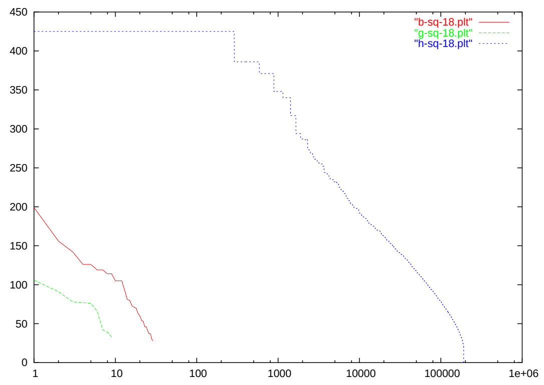

**Figure 4.** Scores for similar seq. ($k = 18$) after extending.

### 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.

In Figure 2 that shows the scores for comparison of dissimilar sequences when $k = 11$, 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. Similar results are observed for the similar datasets with the gap between HSA, BLAST, and spaced seed negligible (plots not shown).

For similar sequences with $k = 18$ (Figures 3) 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. For dissimilar sequences (plots not shown), BLAST and spaced seed did not find any seeds while HSA finds many seeds out of which many have significant score.

One important observation that follows from these experiments is that spaced seeds miss many high scoring entries 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 drops below by 20. We used +1 score for each match and -3 penalty for each mismatch. Figure 4 shows the scores obtained after extending the seeds for the experiments in Figure 3. 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 seed lengths ranging from 12 to 28. In case of the spaced seed scenario the execution is done only for available spaced seeds of length 9 to 18. It can be seen from Table 1 that seed creation time increases slowly until $k = 13$ and suddenly there after. This is because, at $k = 13$, 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 because if it picks a small enough prime number it uses hash table instead of sorted list even for large $k$.

Table 1 also shows that the memory usage of all three methods increase exponentially until $k = 13$ as 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 which can not be fit 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. More experimental results are further described in [**?**].

**Table 1.** Performace comparison results

| | k | 12 | 18 | 22 | 28 |
|---|---|---|---|---|---|
| Time (Sec) | HSA | 0.005 | 0.38 | 0.45 | 0.52 |
| | Blast | 0.006 | 8.06 | 8.07 | 8.05 |
| | S.Seed | 0.005 | 8.08 | - | - |
| Mem. Usage (KB) | HSA | 44097 | 14779 | 12215 | 14017 |
| | Blast | 67124 | 32 | 32 | 32 |
| | S.Seed | 67124 | 32 | - | - |

## 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 [11, 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. [10] 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 [12], BLAST [1], PatternHunter [8]. 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 [6], QUASAR [4], REPuter [7], and AVID [3]. 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.

## 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. We proposed to employ randomized algorithms that are suitable for pattern matching to find initial approximate seeds. Our main contribution 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. Altschul, W. Gish, W. Miller, E. W. Meyers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[2] S. Batzoglou, D.B. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J.P. Mesirov, and E.S. Lander. ARACHNE: A Whole-Genome Shotgun Assembler. *Genome Research*, 12(1):177–189, 2002.

[3] N. Bray, I. Dubchak, and L. Pachter. AVID: A Global Alignment Program. *Genome Research*, 13(1):97–102, 2003.

[4] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, 'E. Rivalsd, 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.

[5] K.P. Choi, F. Zeng, and L. Zhang. Good Spaced Seeds for Homology Search. *Bioinformatics*, 20:1053–1059, 2004.

[6] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. Whited, and D.L. Salzberg. Alignment of Whole Genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[7] S. Kurtz and C. Schleiermacher. REPuter: Fast Computation of Maximal Repeats in Complete Genomes. *Bioinformatics*, 15(5):426–427, May 1999.

[8] M. Ma, J. Tromp, and M. Li. PatternHunter: Faster and More Sensitive Homology Search. *Bioinformatics*, 18(0):1–6, 2002.

[9] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 2000, 2nd Edition.

[10] E. Myers. An O(ND) Difference Algorithm and Its Variations. *Algorithmica*, pages 251–266, 1986.

[11] 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.

[12] 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.

[13] T.F. Smith and M.S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, March 1981.

[14] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A Greedy Algorithm for Aligning DNA Sequences. *Journal of Computational Biology*, 7(1-2):203–214, 2000.