Rapid homology search with neighbor seeds*

Miklós Csűrös† Bin Ma‡

December 8, 2005

Abstract

Using a seed to rapidly "hit" possible homologies for further scrutiny is a common practice to speed up homology search in molecular sequences. It was shown that a collection of higher weight seeds have better sensitivity than a single lower weight seed at the same speed. However, huge memory requirements diminish the advantages of high weight seeds. This paper describes a two-stage extension method, which simulates high weight seeds with modest memory requirements. By drawing upon the previously studied ideas of vector seed and multiple seeds, we introduce *neighbor seeds*, which are implemented using two-stage extension. Neighbor seeds provide the flexibility to maximize the independence between the seeds, which is a well-known criterion for optimizing sensitivity. A major advantage of neighbor seeds is that they all rely on the same pre-built database index. Based on considerations of sensitivity and biological adequacy, different neighbor seeds can be used for different queries without rebuilding the index. The paper also discusses some other practical techniques to reduce memory usage.

Key words: bioinformatics; local alignment; hit-and-extend.

*This work was supported by grants from the Natural Sciences and Engineering Research Council of Canada, the *Fonds québécois de la recherche sur la nature et les technologies*, and a Canada Research Chairs program. A preliminary version [1] was presented at the COCOON conference, August 16–19, 2005, Kunming, China.

†Department of Computer Science and Operations Research, Université de Montréal, C.P. 6128, succ. Centre-Ville, Montréal, Qué., Canada, H3C 3J7. E-mail: csuros@iro.umontreal.ca.

‡Department of Computer Science, University of Western Ontario, London, Ont., Canada, N6A 5B7. E-mail: bma@csd.uwo.ca

1 Introduction

An important task in the analysis of molecular sequences is the search for *local alignments*, formed by pairs of substrings from two sequences, which score high according to some string similarity metric. Local alignments are the "unit operations" in comparative genomics [2], where sequence conservation and lack of it are used to reason about evolutionary relationships and biological function. For instance, even alignments between different species' genomes [3, 4] rely on anchors, which are local alignments between the genomes that restrict the search space for whole-genome alignments.

The importance of the local alignment problem led to a large body of research, starting in the early 1980s with the algorithm of Smith and Waterman [5], later improved by Gotoh [6]. The Smith-Waterman-Gotoh algorithm uses dynamic programming to find all local alignments scoring above a fixed threshold in $O(|S| \cdot |T|)$ time for two sequences S and T over a finite alphabet Σ. For genomic sequences, Σ is the DNA alphabet of size 4. While its speed can be improved by a logarithmic factor [7], a full sensitivity search that involves sequences with several million letters cannot be carried out in a reasonable time frame. For large alignment problems, other solutions are needed that may sacrifice some sensitivity for speed, i.e., that may miss some local alignments but run reasonably fast. Heuristic search programs such as FASTA [8] and BLAST [9] were introduced at the end of the 1980s. They rely on the so-called *hit-and-extend* heuristic, which can be implemented using hashing and lookup tables, introduced in this context by Wilbur and Lipman [10]. The majority of modern local alignment programs [11, 12, 13, 14, 15] exploit some variant of this idea. Some recent alternatives are based on suffix trees [16, 17].

This paper concentrates on hit-and-extend methods. Hit-and-extend methods rely on a hash function $h\colon \Sigma^\ell \to \{0, \ldots, H-1\}$. Local alignments are found by first identifying *hits*, which are pairs of positions (i, j) where $h(S[i..i+\ell-1]) = h(T[j..j+\ell-1])$. The most obvious choice for hashing is to use the identity function, when hits are defined by identical substrings of length ℓ, called ℓ*-mers*. In fact, this strategy is used by homology programs such as BLAST. All the hits can be found efficiently by using a lookup table that stores the *occurrence lists* $\mathsf{Occ}(g) = \{i\colon h(S[i..i+\ell-1]) = g\}$ for every key g. Subsequently, hits are detected and extended by consulting the occurrence list for $h(T[j..j+\ell-1])$ in each position j. Figure 1 outlines this concept. This strategy is often called "seeding" and the hash function or its representation is called as a seed. The sensitivity of a seed measures its ability to hit a homology, and the specificity of a seed characterizes its

Algorithm HIT-AND-EXTEND

Input sequences S, T; hash function $h\colon \Sigma^\ell \to \{0, \ldots, H-1\}$

H1 **for** $i = 1, \ldots, |S| - \ell + 1$ **do**

H2 set $g \leftarrow h(S[i..i+\ell-1])$

H3 add i to the list $\mathsf{Occ}(g)$

H4 **end for**

H5 **for** $j = 1, \ldots, |T| - \ell - 1$ **do**

H6 set $g \leftarrow h(T[j..j+\ell-1])$

H7 process hits $(i, j)\colon i \in \mathsf{Occ}(g)$

H8 **end for**

Figure 1: Basic hit-and-extend procedure. Algorithm HIT-AND-EXTEND outlines the method. Hits are extended in Line H7 by exploring the dynamic programming table around the hits.

ability to filter out a random region.

It was recently discovered [14] that *spaced seeds* provide very good sensitivity and specificity. A spaced seed is defined by a set $\mathcal{S} = \{s_1, \ldots, s_k\} \subseteq \{1, \ldots, \ell\}$. In practice, a spaced seed is often denoted by the characteristic vector for the set, defined as the length-ℓ binary string in which the bits at the positions specified by the seed have value 1. The corresponding hash function concatenates the characters in positions specified by the seed, and encodes the resulting string $u[s_1] \cdot u[s_2] \cdots u[s_k]$ by an integer in the range $\{0, \ldots, |\Sigma|^k - 1\}$. Such a seed is called an (ℓ, k)-seed, and has *weight* k. The initial discovery led to a number of results on selecting spaced seeds [18, 19, 20] in various statistical or empirical alignment models. Additional references with a thorough discussion are offered in [21]. Spaced seeds or similar indexing constructs have been studied also in the contexts of lossless filtration [22, 23] and sequencing by hybridization [24].

There exist several generalizations of spaced seeds, which include multiple seeds [15, 25], and vector seeds [12, 26]. Multiple seeds are a set of carefully selected spaced seeds $\mathcal{S}_1, \ldots, \mathcal{S}_M$. The set of hits for such a set is the union of the hits found by every single seed: $h_m(S[i..i+\ell-1]) = h_m(T[j..j+\ell-1])$ for at least one $m = 1, \ldots, M$. A vector seed is defined by a vector of non-negative weights $(w_1, \ldots, w_\ell)$ and a threshold t: there is a hit at (i, j) if $t \leq \sum_{\delta=1}^{\ell} w_\delta I\{S[i+\delta-1] = T[j+\delta-1]\}$, where $I\{C\}$ is 1 if

and only if condition C is true, otherwise it is 0. (The slightly more general definition of [26] allows for a scoring matrix.) In Section 3 we will show that a vector seed can be viewed as a well-structured set of multiple seeds.

The time complexity increase of using multiple seeds can be offset by using higher-weighted seeds. It was shown that higher-weighted multiple seeds and vector seeds may offer superior sensitivity [15, 26] to that of a single seed at the same specificity. However, they can hardly reach their theoretical potential due to their memory requirements. In case of multiple seeds [15], a lookup table is constructed for every seed. Vector seeds rely on a hash table for the spaced seed defined by the positions with non-zero weights. As a consequence, memory usage is exponential in the number of positive weights. Vector seeds with widely varying weighting schemes proved prohibitive due to their demands on memory. In [26], vector seeds for genomic sequences are restricted to those with weights 0, 1, or 2, permitting at most two mismatches.

We first propose in Section 2 a novel two-stage extension procedure that improves the efficiency of hit-and-extend methods. Rather than being a trivial heuristic, extensive optimization is needed to maximize the sensitivity of the two-stage extension. The concept of neighbor seeds is introduced in Section 3. Neighbor seeds combine the advantages of the multiple seeds, vector seed and two-stage extension; and allow us to attain or surpass the sensitivity and speed of multiple and vector seeds, and pose only modest demands on memory. Section 4 briefly explores some practical techniques of space reduction, which include an implementation of 11-mer based hashing with 1.5 bytes per base pair for the purposes of comparing mammalian-sized genomes. Section 5 concludes the paper.

2 Two-stage extension

Since the introduction of spaced seeds, there has been much work on finding variants of hit conditions and hash functions to gain better sensitivity and specificity. In this section we scrutinize the extension instead.

2.1 Terminologies

Our goal is to detect *local alignments* [27] between two input sequences, S and T. A local alignment involves two substrings $S[i..i']$ and $T[j..j']$ which exhibit a large similarity, according to some similarity metric between strings. Often gaps are inserted to S and T to make the two strings similar. In practice, however, a gapped alignment usually consists of several

Algorithm X-DROP

Input S, T; start (i, j); allowed drop X

X1 set $s \leftarrow 0; \mathsf{max} \leftarrow 0$

X2 **while** $s > \mathsf{max} - X$, $i \leq |S|$, $j \leq |T|$ **do**

X3 **if** $S[i] = T[j]$ **then** $s \leftarrow s + 1$ **else** $s \leftarrow s - 1$

X4 **if** $s > \mathsf{max}$ **then** set $\mathsf{max} \leftarrow s$

X5 set $i \leftarrow i + 1; j \leftarrow j + 1$

X6 **end while**

X7 report max

Figure 2: X-DROP is a popular extension algorithm, used in BLAST [9, 11] and many other alignment programs. The extension is shown only in the forward direction. An analogous extension process is carried out in the backward direction where i and j decrease by 1 in every step.

long gapless alignment. Therefore, most homology search programs perform gapped alignment only if a high-scoring gapless alignment is found. For this reason, we restrict our attention to gapless local alignments in this study.

For simplicity, we consider the alignment scoring policy that rewards an identity with $+1$ and penalizes a mismatch with -1. In other words, the matching score of $S[i..i+k]$ and $T[j..j+k]$ is $\sum_{l=0}^{k} \mu(S[i+l], T[j+l])$ where μ is such that

$$\mu(a, b) = \begin{cases} 1 & \text{if } a = b; \\ -1 & \text{if } a \neq b. \end{cases}$$

Without loss of generality, each local alignment between $S[i..i + n - 1]$ and $T[j..j + n - 1]$ can be thus represented by a 0-1 string R of length n, where $R[k] = 1$ if and only if $S[i + k - 1]$ matches $T[i + k - 1]$. Let n' be the number of ones in R, called the *weight* of R. Then the score of the alignment is $(2n' - n)$; the *similarity* of the local alignment is the ratio n'/n.

If $S[i..i+n-1]$ and $T[j..j+n-1]$ are random unrelated sequences, then the similarity is expected to be $\beta = \sum_{a \in \Sigma} p(a)q(a)$, where $p(a)$ and $q(a)$ are the background frequencies for the letter a in the two sequences. For DNA sequences with alphabet size 4, $\beta = \frac{1}{4}$ if the letter occurrences are uniform random in at least one of S and T. For simplicity, we mostly focus on such a model of random sequences. Nonetheless, the analyses can be easily extended to arbitrary background frequencies.

A heuristic local alignment method can be assessed by evaluating its specificity and sensitivity. *Specificity* is measured by the average running time on random unrelated sequences. *Sensitivity* is measured by the probability of detecting a homology under a probabilistic model of homologies.

There are two commonly used probabilistic models of homologies. The first such model, called the *Bernoulli model* was introduced in [14]: it imposes that local alignments are created by independent equiprobable substitutions. In other words, R is a sequence of independent and identically distributed Bernoulli variables. Another random model of homology is the *uniform model*: R is uniformly drawn from binary sequences of equal length and equal number of ones.

For a given spaced seed, computing the probability that it generates a hit in a homology region is NP-hard in the Bernoulli model [28], as well as in the uniform model [29].

2.2 Time complexity of the classic hit-and-extend method

The usual extension method X-DROP extends a hit in each of the two directions along the diagonal until the score drops by a specified amount. In each direction, the position where the maximum score is reached is recorded and gives the boundary of the gapless local alignment. Figure 2 shows the X-DROP procedure in one direction.

If we ignore the border effects, the average running time of a hit-and-extend method for random sequences is $f \times t \times |S| \times |T|$, where f is the probability of a hit at a fixed position pair (i, j), and t is the average time spent on a hit extension. The probability f is called the *false positive rate* in [26]. In what follows, we analyze t more closely for the X-drop algorithm of Fig. 2. In Line X3, the score decreases with an expected value of $(1-2\beta)$ in each step. The extension should thus stop after around $X/(1-2\beta)$ steps. The following lemma formalizes this argument.

Lemma 1. *Suppose that $\beta < 1/2$ holds for the match probability, and* X-DROP *is invoked with a positive integer X. Let $N = \min\{|S|, |T|\}$. If τ denotes the number of times the loop body X3–X5 is executed, then*

$$\lim_{N\to\infty} \mathbb{E}\tau = \frac{X - \beta\Big(1 - \big(\frac{\beta}{1-\beta}\big)^X\Big)}{1-2\beta} = \frac{X - \sum_{t=1}^{X}\big(\frac{\beta}{1-\beta}\big)^t}{1-2\beta}. \tag{1}$$

Proof. Let $Y_1, Y_2, \ldots$ be the series of ± 1-valued random variables that track the changes of the score s in Line X3 so that $s = \sum_{\delta=1}^{t} Y_\delta$ after t comparisons. Formally, $Y_\delta = 1$ if $S[i+\delta-1] = T[j+\delta-1]$, otherwise $Y_\delta = -1$, and

let $s(t) = \sum_{\delta=1}^{t} Y_\delta$. Clearly, $\{Y_\delta\}$ are independent and identically distributed random variables with expected value $\mathbb{E}Y = -(1-2\beta)$, and $s(t)$ form a simple random walk. Since the number τ is a stopping time for $Y_1, Y_2, \ldots$, Wald's Equation applies [30]:

$$\mathbb{E}s(\tau) = (\mathbb{E}Y)(\mathbb{E}\tau). \tag{2}$$

We need therefore to determine $\mathbb{E}s(\tau)$, the expected final value of the score s when the condition fails in Line X2. The key idea is to consider *ladder points* [31] which are the places where max is updated in Line X4. Specifically, the ladder points $\tau_0 = 0, \tau_1, \tau_2, \ldots$ are defined in the following manner. For every $m > 0$, $\tau_m = \min\{t\colon s(t) = m\}$, with the convention that if $s(t)$ never reaches m, then $\tau_m = \infty$. Let L be the maximum value of $s(t)$ before the score drops by X for the first time: $L = \min\{m\colon s(t) = m - X \text{ for some } \tau_m < t < \tau_{m+1}\}$. The stopping time is $\tau = \min\{t\colon \tau_L < t, s(t) = L - X\}$. Consequently, $s(\tau) = L - X$ and the value $\mathsf{max} = L$ is returned at the end of the extension. We need to compute $\mathbb{E}L$ to obtain $\mathbb{E}\tau$ through Eq. (2). Consider the probability $\mathbb{P}\{L > 0\}$. It equals the probability that the random walk $s(t)$ attains the value 1 before $-X$. By standard results on random walks [30], this probability equals $\rho = \frac{\alpha^{-X}-1}{\alpha^{-X-1}-1}$ with $\alpha = \frac{\beta}{1-\beta}$. Since $\{Y_\delta\}$ are independent, the distribution of L is a (shifted) geometric distribution, and thus $\mathbb{P}\{L = m\} = (1-\rho)\rho^m$ for all $m \in \mathbb{N}$. Hence, $\mathbb{E}L = \frac{\rho}{1-\rho} = \frac{\alpha}{1-\alpha}(1-\alpha^X)$. By Eq. (2),

$$\mathbb{E}\tau = \frac{X - \mathbb{E}L}{1-2\beta} = \frac{X - \frac{\alpha}{1-\alpha}(1-\alpha^X)}{1-2\beta},$$

and Eq. (1) follows by plugging in the value of α. □

Corollary 1. *If the alphabet is of size 4 and one of the sequences is uniform random, then $\beta = \frac{1}{4}$, and the expected number of comparisons at a hit is $4X - 2 + o(1)$.*

Proof. Substituting $\frac{1}{4}$ for β in Lemma 1, $\mathbb{E}\tau = 2X - 1 + 3^{-X}$. Noticing that the extension is done in both directions, the corollary is proved. □

With a typical choice $X = 16$, a hit extension entails approximately 62 character comparisons on average.

2.3 Two-stage hit extension

In this section we propose the following two-stage extension process, and show that this seemingly simple idea is in fact nontrivial and makes a big

difference to the sensitivity. Let $\mathcal{S} = \{s_1, \ldots, s_k\}$ be an (ℓ, k)-seed, and let $\mathcal{S}' = \{s'_1, \ldots, s'_m\}$ be a set of positive integers with $\mathcal{S} \cap \mathcal{S}' = \emptyset$. Furthermore, let $0 < t \leq m$ be a threshold. The triple $(\mathcal{S}, \mathcal{S}', t)$ defines a *relaxed seed* employed in the following manner. Hits are found as if the spaced seed $\mathcal{S}$ were used. When a hit is found, the positions of $\mathcal{S}'$ are tested, and the full extension is performed if at least t matches are found. In particular, let (i, j) be a hit position. Full extension is performed only if $S[i+s'-1] = T[j+s'-1]$ for at least t choices of $s' \in \mathcal{S}'$. A relaxed seed may significantly increase the specificity, which can be seen in Theorem 1. As we will see in Table 1, the sensitivity of a relaxed seed varies very much for different choices of $\mathcal{S}'$ even with the same size and threshold. Therefore, the optimization of the positions in $\mathcal{S}'$ should be done together with $\mathcal{S}$.

Theorem 1. *Suppose that the two-stage extension method is employed with* $|\mathcal{S}'| = m$. *Let* $b(m,t) = \sum_{i=t}^{m} \binom{m}{i} (\frac{1}{4})^i (\frac{3}{4})^{m-i}$. *The average number of character comparisons performed during a bi-directional hit extension is* $C = \big(m + b(m,t)(4X - 2)\big) + o(1)$.

Proof. The preliminary test on $\mathcal{S}'$ compares m pairs of characters. A full extension is performed with probability $b(m,t) \leq 1$. A full extension performs an average of $4X - 2 + o(1)$ comparisons by Corollary 1. □

The sensitivity of a relaxed seed is assessed in the following manner. Let R be a binary representation of a homology region. In order to have a hit with the relaxed seed $(\mathcal{S}, \mathcal{S}', t)$, R has to contain a substring u such that $\forall s \in \mathcal{S}\colon u[s] = 1$ and $\sum_{j=1}^{m} u[s'_j] \geq t$. The sensitivity is defined to be the hit probability under a specific probabilistic model for homologies such as the Bernoulli or uniform model. Since spaced seeds can be considered as special cases of relaxed seeds, it is NP-hard to compute the sensitivity of a relaxed seed.

Theorem 2. *Computing the sensitivity of a relaxed seed is NP-hard in the Bernoulli and uniform models.*

Proof. Let $(\mathcal{S}, \mathcal{S}', t)$ be a relaxed seed with $t = |\mathcal{S}'|$. Then the relaxed seed is equivalent to the spaced seed $\mathcal{S} \cup \mathcal{S}'$. Computing the sensitivity is NP-hard in both models [28, 29]. □

We pose as an open problem to prove the NP-hardness of the problem for a fixed threshold $t < |\mathcal{S}'|$.

Known exponential-time algorithms for computing the sensitivity of spaced seeds in the uniform [32] and in the Bernoulli [18, 20] models can be readily

Seed & threshold		Sens. (u)	Sens. (B)	C	T
`111001001001010111`		0.618	0.594	62	4
`111010010100110111`		0.451	0.467	62	1
`1111111111`		0.391	0.412	62	4
`111xx1xx1x01010111x`	3	0.555	0.551	14.50	0.94
`x1110x10x10x1010111`	2	0.550	0.548	20.22	1.30
`1110010010010101111xxxx`	2	0.528	0.535	20.22	1.30

Table 1: Comparison of some relaxed and spaced seeds. Relaxed seeds are encoded by `0`–`1`–`x` strings: position i has `1` if it is in $\mathcal{S}$, and it has `x` if it is in $\mathcal{S}'$. Sensitivity (*Sens.*) is calculated for homology regions of length 64 at 70% similarity in the Bernoulli (B) model, and with 19 mismatches in the uniform (u) model. Column C shows the expected number of comparisons in hit extension, when $X = 16$, and column T lists the expected time spent on finding and extending hits, defined as C times the false positive rate. T is normalized for weight-11 seeds. The two spaced seeds are the most sensitive weight-10 and weight-11 seeds under Bernoulli models. (Notice that they are much better than a 10-mer.) The table also lists some relaxed seeds. Notice that the placement of relaxed positions has a non-negligible effect on sensitivity.

adapted to relaxed seeds. As an alternative, one can convert a relaxed seed to an equivalent set of multiple seeds and compute the sensitivity by employing the algorithm in [15] that calculates the sensitivity of an arbitrary set of seeds.

Table 1 compares relaxed and spaced seeds. It turns out that the sensitivity of a $(k-1)$-weight seed can be approached while the running time stays close to that of a weight-k spaced seed. It is noteworthy that the last two seeds, `x1110x10x10x1010111` and `1110010010010101111xxxx` have the same $\mathcal{S}$, same size $|\mathcal{S}'|$ and same threshold t. At the same time, they have very different sensitivities. The example demonstrates that the two-stage extension is not a trivial extension heuristic. Indeed, it can be fully profited of only after a meticulous optimization step, in which the threshold and the relaxed positions are selected. This observation is epitomized by the extreme case of a relaxed seed $(\mathcal{S}, \mathcal{S}', t)$ where $t = |\mathcal{S}'|$. This relaxed seed is equivalent to the spaced seed $\mathcal{S} \cup \mathcal{S}'$, and the necessity to optimize the spaced seed is well-known [14].

Although no efficient algorithm is known to find the optimal spaced seed, it was argued in [28] that finding one optimal spaced seed in the Bernoulli model is unlikely to be NP-hard, because the language that defines the

problem is sparse. The same argument can show that finding the optimal relaxed seed $(\mathcal{S}, \mathcal{S}', t)$ for Bernoulli or uniform homology regions is unlikely to be NP-hard. However, the problem of finding the optimal $\mathcal{S}'$ for given $\mathcal{S}$, $|\mathcal{S}'|$ and t does not correspond to a sparse language anymore and could be intractable.

2.4 Implementation and memory usage

The data structure for the basic algorithm of Fig. 1 has to support the operation $\textsc{Add}(g, i)$ that records the position i as one belonging to $\mathsf{Occ}(g)$, and the operation $\textsc{reportAll}(g)$ that returns the list $\mathsf{Occ}(g)$. For a spaced seed with weight k, a rather straightforward implementation was introduced in [14]. An integer array, *head*, of length 4^k was used to record the first occurrence of each hash value. Then another integer array, *next*, of length $|S|$ is used to retrieve all the other occurrences. $next[i]$ records the next occurrence of the same hash value as position i. The two arrays *head* and *next* form a hash table that requires memory for $(4^k + |S|)$ integers. In a direct manner, a relaxed seed $(\mathcal{S}, \mathcal{S}', t)$ can be implemented by relying on the hash table for the spaced seed $\mathcal{S}$.

3 Neighbor seeds

In this section we propose the neighbor seeds idea which elegantly combines the advantages of the multiple seeds [15], vector seeds [12, 26], and the two-stage extension introduced in Section 2.

It has been known that a spaced seed performs better than a consecutive seed with the same weight because the hit positions are more independent for the spaced seed [14, 28]. That is, for spaced seeds, different positions do not easily hit together ("wasting" hits) or fail together (missing the region entirely). Multiple spaced seeds perform even better by adding another level of independence between the hits of different seeds [15]. While significant improvement on sensitivity was obtained, the necessity of producing multiple hash tables and/or parsing the sequence multiple times greatly limited the number of seeds a search can use.

Vector seeds were proposed in [26], independently from multiple seeds. They are also very effective for improving sensitivity. Although the sensitivity cannot compare with multiple seeds at the same specificity, vectors seeds enjoy the advantage that only one hash table is needed.

We make the observation that every vector seed corresponds to a particular set of ordinary spaced seeds defined as follows. Let $(w_1, \ldots, w_\ell)$ be

the weights and t be the threshold of the vector seed. Let $\mathcal{P} = \{\delta : w_\delta > 0\}$ be the set of positively weighted positions. The vector seed is equivalent to the set of seeds $\{\mathcal{S}_1, \ldots, \mathcal{S}_M\}$ where $\mathcal{S}_i$ are the subsets of $\mathcal{P}$ in which $t \leq \sum_{\delta \in \mathcal{S}_i} w_\delta$. When all vector weights are 0 or 1, the equivalent multiple seeds are generated by removing up to $(|\mathcal{P}| - t)$ elements from $\mathcal{P}$. In fact, an equivalent set can be created by removing exactly that many positions. (In the preliminary version of this paper [1], the seeds in the equivalent set are called daughter seeds.) We also observe that some seeds in the equivalent set are closer to each other than some other seeds in the set. In order to maximize the independence between hits of different seeds, we should primarily rely on more distant seeds in the set. Our third observation is that the independence between neighbor seeds can be further increased using two-stage extension introduced in Section 2. Different neighbor seeds should be allowed to have different check points (the $\mathcal{S}'$) in the first extension stage.

These three observations lead to the following powerful idea. In what follows, a *parent seed* is a spaced seed $\mathcal{P}$. For a fixed integer Δ, a *neighbor seed* $\mathcal{D}$ of $\mathcal{P}$ is such that $|\mathcal{D} \backslash \mathcal{P}| = |\mathcal{P} \backslash \mathcal{D}| = \Delta$. When the seeds are represented in the binary string form, the equations imply that each neighbor seed has the same weight as the parent and is within Hamming distance 2Δ from the parent.

It is easy to see that two neighbor seeds may differ at as many as 4Δ different positions. In practice, when $\Delta = 2$, the neighbor seeds are already very different (and hopefully very independent) to each other. Selecting multiple seeds from the neighbors of the same parent may achieve similar performance as selecting from the complete seed space. Even if neighbor seeds cannot outperform general multiple seed sets, the beauty of the neighbor seeds is that they can share the same hash table in the following way. Let $\mathcal{P} = \{p_1, p_2, \ldots, p_k\}$ be a parent seed with $0 < p_j \leq \ell$ for all j. Given a database DNA sequence T, a hash table $\mathcal{H}$ is built using $\mathcal{P}$ as described in Section 2.4 to store the occurrence lists $\mathsf{Occ}(g) = \{i : h(T[i..i+\ell-1]) = g\}$. Recall that for any length-ℓ strings s and t, the hash function h is such that $h(s) = h(t)$ if and only if $s[j] = t[j]$ for $j \in \mathcal{P}$. In order to facilitate the ensuing discussion, write $\mathsf{Hits}_{\mathcal{S}}(q) = \{i : q[j] = T[i+j-1] \text{ for } j \in \mathcal{S}\}$ where $q \in \Sigma^\ell$ is an arbitrary query string and $\mathcal{S}$ is any spaced seed. Using this notation, $\mathsf{Hits}_{\mathcal{P}}(q) = \mathsf{Occ}(h(q))$, and $\mathsf{Hits}_{\mathcal{P}}(q)$ is efficiently retrieved using the hash table $\mathcal{H}$.

Let $\mathcal{D} = \{d_1, d_2, \ldots, d_k\}$ be a neighbor seed and $|\mathcal{P} \setminus \mathcal{D}| = |\mathcal{D} \setminus \mathcal{P}| = \Delta$. Since the hash table $\mathcal{H}$ does not impose any limitation on the lower and upper bounds of p_j, we assume $0 < d_j \leq \ell$ for all j, without loss of generality. We assume furthermore that $d_j = p_j$ for all $j = \Delta + 1, \ldots, k$. We need to

show that $\mathsf{Hits}_{\mathcal{D}}(q)$ can be efficiently computed using the hash table $\mathcal{H}$ for the parent seed. It is easy to see that

$$\mathsf{Hits}_{\mathcal{D}\cap\mathcal{P}}(q)$$
$$= \bigcup_{t_1,\ldots,t_\Delta\in\Sigma} \Big\{ i\colon T[i+p_j-1] = t_j \text{ for } j \le \Delta, T[i+p_j-1] = q[p_j] \text{ for } j > \Delta \Big\}$$

Consequently, $\mathsf{Hits}_{\mathcal{D}\cap\mathcal{P}}(q)$ can be computed by forming the union of $|\Sigma|^\Delta$ occurrence lists stored in $\mathcal{H}$. For each $i \in \mathsf{Hits}_{\mathcal{D}\cap\mathcal{P}}(q)$, we further use two-stage extension to check whether $T[i+j-1] = q[j]$ holds for all $j \in \mathcal{D} \setminus \mathcal{P}$. If so, then we proceed to full hit extension; otherwise, we discard i. In this way, $\mathsf{Hits}_{\mathcal{D}}$ is obtained using $\mathsf{Hits}_{\mathcal{P}}$.

When the database sequence T is large and the parent seed has a reasonable weight k (such as $|T| \ge 64 \times 10^6$ and $k = 13$), all or almost all $\mathsf{Occ}(g)$ are non-empty. In such a situation, the time of generating $\mathsf{Hits}_{\mathcal{D}\cap\mathcal{P}}$ is insignificant.

If Δ is reasonably small ($\Delta = 2$ in our examples to follow), the time of generating $\mathsf{Hits}_{\mathcal{D}}$ from $\mathsf{Hits}_{\mathcal{D}\cap\mathcal{P}}$ is quite modest compared to the full extension of a hit. For each $i \in \mathsf{Hits}_{\mathcal{D}\cap\mathcal{P}}$, up to Δ more positions are examined to produce $\mathsf{Hits}_{\mathcal{D}}$. When these positions are checked one by one in random DNA homology regions, the average number of comparisons is $1+4^{-1}+\ldots+4^{-\Delta} = \frac{4}{3}\times(1-4^{-\Delta})$. Because $\mathsf{Hits}_{\mathcal{D}\cap\mathcal{P}}$ is 4^Δ times of the size of $\mathsf{Hits}_{\mathcal{D}}$, this approach of generating $\mathsf{Hits}_{\mathcal{D}}$ will cost $\frac{4}{3} \times (1 - 4^{-\Delta}) \times 4^\Delta = \frac{4}{3} \times (4^\Delta - 1)$ extra comparisons for each successful hit in $\mathsf{Hits}_{\mathcal{D}}$ (compared to generating hits in $\mathsf{Hits}_{\mathcal{P}}$). As discussed in Section 2.2, these extra comparisons are relatively inexpensive in a typical setting, in view of the time spent in fully extending a successful hit. For instance, when $\Delta = 2$, each hit of $\mathsf{Hits}_{\mathcal{D}}$ costs 20 extra comparisons, whereas a typical X-drop extension would incur 62 comparisons already. The number of comparisons increases only by $20/62 \approx 32\%$.

When multiple neighbor seeds are used, the hits for each of them can be generated using the above mentioned procedure. Subsequently, usual extension is carried out for every hit in order to generate the local alignment. Notice that as long as the neighbor seeds are from the same parent seed $\mathcal{P}$, the hash table $\mathcal{H}$ needs to be built only once and can be reused by every neighbor seed. In contrast to general multiple seeds [15], not only do the neighbor seeds require only one hash table, but also the number and the choices of neighbor seeds can be decided after the hash table $\mathcal{H}$ is built. This property is of great interest in practice because a program can index the DNA database first. Only much later need the users decide on the

#	seed	sensitivity
parent	1110110010110101111	-
1	11100110110010101111	0.2066
2	1101110110000110100111	0.3311
3	1011110010110111011	0.4234
4	11001110000010110101111	0.4913
5	1011011010110001111	0.5472
6	10101010110010100101111	0.5921
7	1110110001111101101	0.6279
8	11001110110010010001111	0.6590

Table 2: The first eight greedily selected neighbor seeds of 1110110010110101111. The optimization is done on 5×10^6 uniformly drawn regions of length 64 with 45 matches. The accumulated sensitivity is estimated by using 10^7 such regions.

actual neighbor seeds, depending on factors such as speed and sensitivity, and whether the queries are protein-coding or non-coding sequences.

Similarly to the multiple seeds proposed in [15], the neighbor seed sets need to be optimized in order to maximize sensitivity. The neighbor seeds used in this paper are selected by first fixing a parent seed, and then using the same greedy algorithm as in [15] to add neighbor seeds one by one, which maximize the sensitivity of the set on target homology regions. Table 2 shows the first eight neighbor seeds selected by the greedy algorithm for the parent seed 1110110010110101111 and $\Delta = 2$, along with their accumulated sensitivities in uniform regions of length 64 with 45 matches (about 70% similarity level).

Figures 3–5 plot the sensitivity of neighbor seeds in function of the similarity, along with some other seeding techniques for comparison. In particular, single spaced seeds, a vector seed, and multiple seeds are compared to weight-13 neighbor seeds.

Figure 3 compares multiple weight-13 neighbor seeds with optimized single spaced seed of different weights. We note that sixteen neighbor seeds have the same specificity as a single weight-11 spaced seed, while have much higher sensitivity. Similarly, 64 neighbor seeds are comparable in specificity to a single weight-10 spaced seed, while they achieve an outstanding sensitivity. The plots also illustrate that the difference in sensitivity between neighbor seeds and spaced seeds is about the same as the one between spaced seeds and contiguous seeds (i.e., k-mers). For instance, at 45 matches, the sensitivity for the 10-mer is 0.39; for the weight-10 spaced seed it is 0.62;

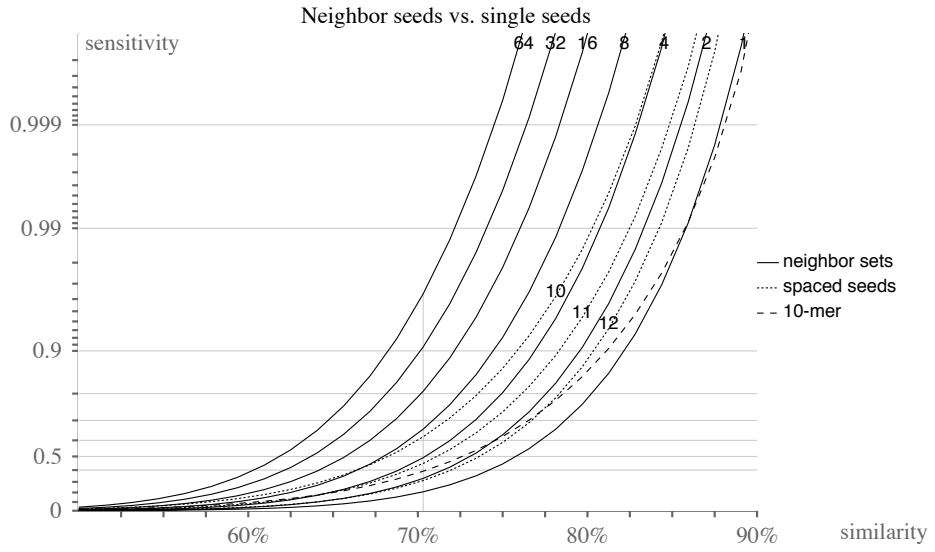

Figure 3: Neighbor seeds compared to single spaced seeds. Sensitivity (Y axis) is computed in the uniform model on regions of length 64 by Monte Carlo approximation. (The Y axis is mapped using $\sqrt{1 - \log(1 - \text{sensitivity})}$ onto graphical coordinates for better visibility.) Numbers on the neighbor set curves denote the set size; numbers on the single seed curves denote the seed weight. The spaced seeds have maximum sensitivity among spaced seeds of equal weight in the uniform model at 45 matches (cca. 70% similarity, shown by the pale vertical line).

and for 64 neighbor seeds it is 0.96.

Figure 4 compares neighbor seeds with the vector seed VS-12-13 which requires twelve matches in thirteen positions [26]. The vector seed is optimized for uniform regions with 45 matches out of 64. The specificity of such a vector seed is $4^{-13} + 13 \times 3 \times 4^{-13} = 40 \times 4^{-13}$. This is identical to the specificity of forty weight-13 neighbor seeds. However, twenty neighbor seeds already have a better sensitivity than the vector seed.

Figure 5 compares neighbor seeds with multiple seeds. Because neighbor seeds are selected from a special subset of the complete seed space, whereas multiple seeds are selected freely, the performance of neighbor seeds cannot be better. The curves in Figure 5 demonstrate this observation. However, it is also clear that the neighbor seed set's sensitivity is close to that of equal-sized multiple seed sets. Considering the convenience that neighbor

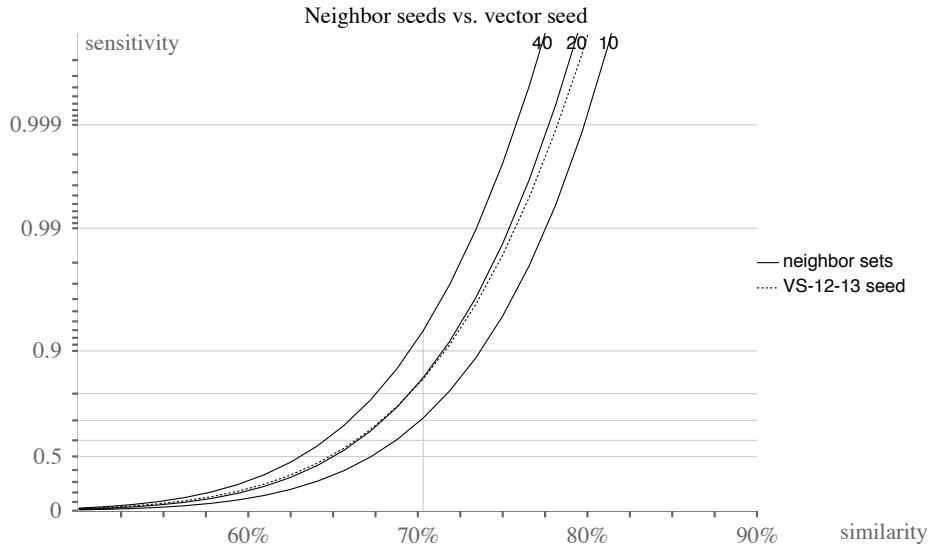

Figure 4: Neighbor seeds compared to the vector seed VS-12-13. Sensitivity (Y axis) is computed in the uniform model on regions of length 64 by Monte Carlo approximation. Numbers on the neighbor set curves denote the set size.

seeds can provide, i.e., the use of one hash table and the selection of seeds after hashing, neighbor seeds are clearly a better choice in practice.

We also point out that according to our simulation on ten million regions, 64 weight-13 neighbor seeds can achieve better than 99.99% sensitivity in a uniform region of 49 matches out of length 64 (76.5% similarity). It is a theoretically interesting question to ask for the minimum number of neighbor seeds or multiple seeds to achieve 100% sensitivity; alternatively, one can ask the minimum of similarity for a group of neighbor seeds or multiple seeds to achieve 100% sensitivity.

4 Two ideas for reducing memory usage

Neighbor seeds rely on a single hash table for the parent seed, and avoid this way the impractical memory requirements of general seed sets. Further memory reductions can be achieved by storing the hash table in a more compact fashion. We need a data structure that supports the operation REPORTALL. If k-mers are used, then a suffix array can provide the function-

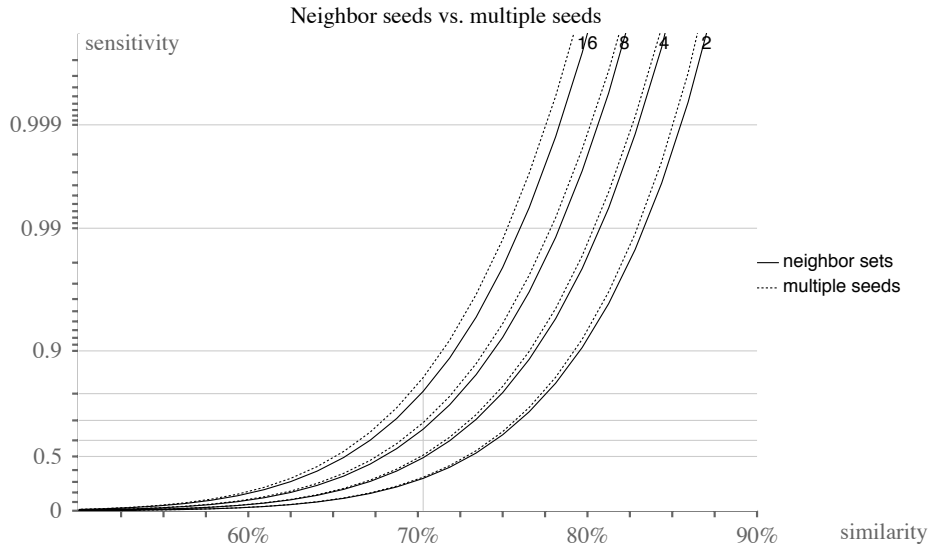

Figure 5: Neighbor seeds compared to multiple seeds. Sensitivity (Y axis) is computed in the uniform model on regions of length 64 by Monte Carlo approximation. Numbers on the curves denote the set size.

ality, which can be implemented using $O(|S|)$ bits [33] in addition to storing the sequence S. Various self-indexing methods [34, 35] promise even better compression by storing S and the indexing structure together. These latter, however, are still impractical for genomic DNA sequence comparisons, since the amount of time they spend on retrieving each hit is measured in milliseconds [35]. More importantly, the suffix array approach cannot support spaced seeds and suffer from the low sensitivity of using k-mers.

The data structure for the hashing is typically implemented using 32-bit integers [14]. Consequently, a table for a k-weight key occupies $4(4^k + |S|)$ bytes. We describe a way of saving space without much sacrifice in either speed or ease of implementation. In particular, we show how to replace 32-bit integers with $(2k)$-bit integers. For seeds of weights 10–13, this means a memory reduction of 37.5–18.75%. The idea is fairly simple: choose a large integer Q and store the modulo Q remainders in both head and next. The integer value Q is reserved for marking ends of lists, so $\lceil \log_2(Q+1) \rceil$-bit integers suffice. Figure 6 shows the data structure. Since $\textsc{Add}(g, i)$ is called in increasing order of i (cf. Fig. 1), key occurrences are restored correctly.

Theorem 3. *(a)* reportAll *of Fig. 6 correctly enumerates the occur-*

Initialization

I1 allocate $\mathsf{head}[0..H-1]$

I2 **for** all g set $\mathsf{head}[g] \leftarrow Q$

I3 allocate $\mathsf{next}[1..|S|-\ell+1]$

Add(g, i)

A1 $\mathsf{next}[i] \leftarrow \mathsf{head}[g]$

A2 $\mathsf{head}[g] \leftarrow i \bmod Q$

reportAll(g)

R1 set $\mathsf{Occ} \leftarrow \emptyset$; $i \leftarrow \mathsf{head}[g]$

R2 **if** $i = Q$ **then return** Occ

R3 set $q \leftarrow \left\lfloor \frac{|S|-\ell-i+1}{Q} \right\rfloor$

R4 **while** $i \neq Q$ **do**

R5 **while** $h(S[qQ+i..qQ+i+\ell-1]) \neq g$ **do** $q \leftarrow q-1$

R6 $\mathsf{Occ} \leftarrow \mathsf{Occ} \cup \{qQ+i\}$

R7 $j \leftarrow \mathsf{next}[qQ+i]$; **if** $j \geq i$ **then** $q \leftarrow q-1$

R8 $i \leftarrow j$

R9 **return** Occ

Figure 6: Data structure for occurrence lists that uses integers in the range $\{0, \ldots, Q\}$. The value Q represents a null pointer.

rences of a key g, provided that the calls $\textsc{Add}(g, i)$ *were made in increasing order of* i.

(b) Suppose that S is a uniform random string, and the hash function is such that all keys occur with equal probability. If $Q \gg 1$ and $|S| \to \infty$, then the hash function is evaluated in the loop of Line R5 $(1-e^{-Q/H})^{-1}$ times on average. If h is defined by a weight-k spaced seed, then, for each occurrence of a key g, $\textsc{ReportAll}(g)$ *performs an expected number of $k + \frac{4/3}{e^{Q/H}-1}$ character comparisons.*

Proof. Claim (a) holds since g occurs in positions $q_1Q + \mathsf{head}[g]$, $q_2Q + \mathsf{next}[\mathsf{head}[g]]$, $q_3Q + \mathsf{next}[\mathsf{next}[\mathsf{head}[g]]]$, ... with $q_1 \geq q_2 \geq \cdots$. The variable q always stores the current value of q_i until all the occurrences are enumerated. In order to prove Claim (b), we use the fact that the set positions in which a particular key occurs can be modeled using a Poisson process [27]. Let Δ be the number of times the **while** condition is evaluated in Line R5 before continuing with Line R6. Let X be the distance between the previously found occurrence and the one the loop is looking for. Then $\mathbb{P}\{\Delta = q\} = \mathbb{P}\{X \in [1 + (q-1)Q, qQ]\}$ for all $q = 1, 2, \ldots$. Using the Poisson process approximation, $\mathbb{P}\{\Delta = q\} = (1-\gamma)\gamma^{q-1}$ with $\gamma \approx (1 - H^{-1})^Q \approx e^{-Q/H}$. Consequently, $\mathbb{E}\Delta = \frac{1}{1-\gamma}$ as claimed. For spaced seeds in particular, when the loop condition evaluates to true, an expected number of $4/3$ positions are looked at, and when the condition finally fails, k comparisons are made. The expected number of tested positions is therefore $k + \frac{4}{3}(\mathbb{E}\Delta - 1)$, as claimed. □

By Theorem 3, using $Q = 4^k - 1$ with a weight-k seed entails an expected number of $(k + 0.77)$ character comparisons. (One can even get away with not comparing all k positions in Line R5 but only some $k' < k$ of them. There is a small chance $(0.25^{k'})$ that we switch to enumerating occurrences of a different hash key g'. The key g', however, matches key g in k' positions, and so the generated hits are not completely arbitrary. The advantage is the lower number of comparisons per hit.) As an alternative to the (mod Q) representation, one can avoid the character comparisons by using run-length encoding [36] of the distances between consecutive occurrences, which reduces the space equivalently at the price of having to handle bit vectors of varying length.

Suffix trees or arrays can be employed to enumerate occurrences of k-mers. To our knowledge, there is no efficient way of retrieving occurrences of spaced seeds from a suffix array, and thus their use is limited to k-mers. At

table type	chromosome		genome	
	int32	mod$Q$	int32	mod$Q$
11mers	33	22.69	32.07	22.04
every 2^{nd} 12mer	20	14.375	16.25	11.68
every 4^{th} 14mer	136	110.5	12	9.75
12mers	36	27	32.5	24.38
every 2^{nd} 13mer	32	25	17	13.28

Table 3: Number of bits used per character when storing a k-mer table. The traditional implementation uses 32-bit integers; the implementation of Fig. 6 (mod$Q$) uses $2k$-bit integers. Note that $(2k-1)$ or $(2k-2)$ bits suffice for storing occurrences restricted to every second or fourth position, respectively. Sequence lengths are $|S| = 2^{27}$ for a chromosome, and $|S| = 2^{31}$ for a genome, based on the human genome.

the same time, suffix tree-based local alignment methods use at least 12.5–15.6 bytes [17] per base pair. Here we describe a simple method of reducing storage for hashing with k-mers in genome-size local alignments. The idea is to use a hash table for longer $(k+d)$-mers sampled in every $(d+1)$-th position of S. The occurrences of a key g can be retrieved by listing the occurrences of the keys $a_1a_2\cdots a_d \cdot g$, $a_1\cdots a_{d-1}ga_d$, $\ldots$, $ga_1a_2\cdots a_d$ for all choices of $a_1,\ldots,a_d \in \Sigma$. With a judicious choice of d, the running time remains essentially the same, while the memory usage is reduced. Table 3 shows some numerical values, for a typical mammalian chromosome or genome. For instance, about 1.5 bytes/nucleotide suffice for 11-mer based alignment of a whole mammalian genome, if the sequence is stored in 2 bits/nucleotide and the table is stored in less than 10 bits/nucleotide. This memory usage is better than that of the currently most space-efficient suffix array representation [37], which uses 12 bits per nucleotide in addition to the sequence storage. At the same time, the hash table takes considerably less effort to implement.

5 Conclusion

We introduced novel ideas on selecting a structured set of spaced seeds to gain superior sensitivity and speed in hit-and-extend methods of local alignment. The neighbor seeds we proposed remarkably outperform spaced seeds and vector seeds, and approach the performance of multiple spaced

seeds. At the same time, neighbor seeds use the same hash table of the same parent seed, and therefore only require moderate amount of memory. Our examples of 32 or 64 neighbor seeds have outstanding sensitivity: a comparable multiple seed set would pose unrealistic memory demands for current desktop computers. A very important feature of neighbor seeds is that their selection can be done after the hash table of the database sequence is built with the parent seed. This is of great interest in practical homology search systems, as the same hash table can support a wide variety of applications through different neighbor seeds which emphasize different factors such as speed, sensitivity, and coding/non-coding regions.

We additionally described some easily implementable ways to lower memory demands. Memory usage is a key factor in the efficiency of homology search algorithms, and is likely to become even more important in the future. Both the number and total length of DNA sequences in Genbank has doubled about every 17 months (`http://www.ncbi.nih.gov/Genbank/genbankstats.html`) since 1983. This rate of increase is comparable to the popular version of Moore's law about computing power doubling every 18 months, and thus powerful heuristics are likely to remain highly valued in the comparison of molecular sequences. Our methods are memory efficient and offer practical solutions for the alignment of large genomic sequences in terms of speed and sensitivity.

References

[1] Csűrös, M., Ma, B.: Rapid homology search with two-stage extension and daughter seeds. In: Proc. 11th Int. Computing and Combinatorics Conf. (COCOON). Volume 3595 of LNCS., Springer-Verlag (2005) 104–114

[2] Miller, W., Makova, K.D., Nekrutenko, A., Hardison, R.C.: Comparative genomics. Annual Review of Genomics and Human Genetics **5** (2004) 15–56

[3] Brudno, M., Do, C.B., Cooper, G.M., Kim, M.F., Davydov, E., Program, N.C.S., Green, E.D., Sidow, A., Batzoglou, S.: LAGAN and Multi-LAGAN: efficient tools for large-scale multiple alignment of genomic DNA. Genome Research **13** (2003) 721–731

[4] Couronne, O., Poliakov, A., Bray, N., Iskhanov, T., Ryaboy, D., Rubin, E., Pachter, L., Dubchak, I.: Strategies and tools for whole-genome alignments. Genome Research **13** (2003) 73–80

[5] Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. Journal of Molecular Biology **147** (1981) 195–197

[6] Gotoh, O.: An improved algorithm for matching biological sequences. Journal of Molecular Biology **162** (1982) 708–708

[7] Crochemore, M., Landau, G.M., Ziv-Ukelson, M.: A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. In: Proc. 11th ACM-SIAM Symposium on Discrete Algorithms (SODA). (2002) 679–688

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

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

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

[11] Altschul, S.F., Madden, T.L., Schäffer, A.A., Zhang, J., Zhang, Z., Miller, W., Lipman, D.J.: Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Research **25** (1997) 3389–3402

[12] Schwartz, S., Kent, W.J., Smit, A., Zhang, Z., Baertsch, R., Hardison, R.C., Haussler, D., Miller, W.: Human-mouse alignments with BLASTZ. Genome Research **13** (2003) 103–107

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

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

[15] Li, M., Ma, B., Kisman, D., Tromp, J.: PatternHunter II: highly sensitive and fast homology search. Journal of Bioinformatics and Computational Biology **2** (2004) 411–439

[16] Delcher, A.L., Kasif, S., Fleischmann, R.D., Peterson, J., White, O., Salzberg, S.L.: Alignment of whole genomes. Nucleic Acids Research **27** (1999) 2369–2376

[17] Kurtz, S., Phillippy, A., Delcher, A.L., Smoot, M., Shumway, M., Antonescu, C., Salzberg, S.L.: Versatile and open software for comparing large genomes. Genome Biology **5** (2004) R12

[18] Buhler, J., Keich, U., Sun, Y.: Designing seeds for similarity search in genomic DNA. Journal of Computer and System Sciences **70** (2005) 342–363

[19] Choi, K.P., Zhang, L.: Sensitivity analysis and efficient method for identifying optimal spaced seeds. Journal of Computer and System Sciences **68** (2004) 22–40

[20] Keich, U., Li, M., Ma, B., Tromp, J.: On spaced seeds for similarity search. Discrete Applied Mathematics **138** (2004) 253–263

[21] Brown, D.G., Li, M., Ma, B.: A tutorial of recent developments in the seeding of local alignment. Journal of Bioinformatics and Computational Biology **2** (2004) 819–842

[22] Pevzner, P., Waterman, M.S.: Multiple filtration and approximate pattern matching. Algorithmica **13** (1995) 135–154

[23] Burkhardt, S., Kärkkäinen, J.: Better filtering with gapped q-grams. Fundamenta Informaticae **23** (2003) 1001–1008

[24] Frieze, A.M., Preparata, F.P., Upfal, E.: Optimal reconstruction of a sequence from its probes. Journal of Computational Biology **6** (1999) 361–368

[25] Sun, Y., Buhler, J.: Designing multiple simultaneous seeds for DNA similarity search. In: Proc. 8th Annual International Conference on Computational Molecular Biology (RECOMB). (2004) 76–84

[26] Brejová, B., Brown, D., Vinař, T.: Vector seeds: An extension to spaced seeds. Journal of Computer and System Sciences **70** (2005) 364–380

[27] Waterman, M.S.: Introduction to Computational Biology: Maps, Sequences and Genomes. CRC Press (1995)

[28] Li, M., Ma, B., Zhang, L.: Superiority and complexity of spaced seeds. In: Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA). (2006) To appear.

[29] Nicolas, F., Rivals, E.: Hardness of optimal spaced seed design. In: Combinatorial Pattern Matching: 16th Annual Symposium. Volume 3537 of LNCS., Springer-Verlag (2005) 144–155

[30] Ross, S.M.: Stochastic Processes. Second edn. Wiley & Sons (1996)

[31] Ewens, W.J., Grant, G.R.: Statistical Methods in Bioinformatics: An Introduction. Springer-Verlag (2001)

[32] Kucherov, G., Noé, L., Ponty, Y.: Estimating seed sensitivity on homogeneous alignments. In: Proc. 4th IEEE Symposium on Bioinformatics and Bioengineering (BIBE). (2004) 387–394

[33] Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proc. 32nd ACM Symposium on Theory of Computing (STOC). (2000) 397–406

[34] Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS). (2000) 390–398

[35] Mäkinen, V., Navarro, G.: Compressed compact suffix arrays. In: Combinatorial Pattern Matching: 15th Annual Symposium. Volume 3109 of LNCS., Springer-Verlag (2004) 421–433

[36] Golomb, S.W.: Run-length encodings. IEEE Transactions on Information Theory **12** (1966) 399–401

[37] Hon, W.K., Sadakane, K.: Space-economical algorithms for finding maximal unique matches. In: Combinatorial Pattern Matching: 13th Annual Symposium. Volume 2373 of LNCS. (2002) 144–152