Performing local similarity searches with variable length seeds*

Miklós Csűrös

Département d'informatique et de recherche opérationnelle, Université de Montréal
C.P. 6128 succ. Centre-Ville, Montréal, Québec, H3C 3J7, Canada
csuros@iro.umontreal.ca

Abstract. This paper describes a general method for controlling the running time of similarity search algorithms. Our method can be used in conjunction with the seed-and-extend paradigm employed by many search algorithms, including BLAST. We introduce the concept of a seed tree, and provide a seed tree-pruning algorithm that affects the specificity in a predictable manner. The algorithm uses a single parameter to control the speed of the similarity search. The parameter enables us to reach arbitrary levels between the exponential increases in running time that are typical of seed-and-extend methods.

1 Introduction

Finding similarities between sequences is one of the major preoccupations of bioinformatics [1]. All similarities, defined as local alignments scoring above a certain threshold, can be found with dynamic programming using the Smith-Waterman algorithm [2]. While there are many inventions that improve the speed of a full sensitivity search (e.g., [3]), a full-scale search that involves sequences with several million letters cannot be carried out in a reasonable time frame. In such cases, sensitivity is most often sacrificed for speed. This paper describes a method to adjust the running time of local similarity searches on a fine scale.

Most successful alignment heuristics rely on suffix trees [4], or hash tables. Hashing-based methods include FASTA [5], BLAST [6], BLAT [7], BLASTZ [8], and PatternHunter [9, 10]. In their simplest form, hashing-based, or *seed-and-extend*, methods identify common short substrings of a fixed length k between the input sequences S and T. Matching substrings, or hits, are the starting points ("seeds") for computing local alignments. The sensitivity depends on the seed length k: weaker similarities can be found by using smaller values. Unfortunately, by decreasing k, we increase the number of hits due to random matches that do not expand into significant alignments. Roughly speaking, the number of such spurious hits is $\frac{|S|\cdot|T|}{4^k}$ for DNA sequences. As a consequence, increased sensitivity levels are achieved at the cost of fourfold increases in the running time.

We propose a general technique to regulate the specificity of hashing-based methods. The key idea is to vary the seed lengths, which leads to a fine resolution

* Research supported by NSERC grant 250391-02.

in specificity. Shorter seeds mean better sensitivity, so we aim at a maximal increase in sensitivity by maximizing the number of shorter seeds. The procedure is controlled by a parameter that explicitly sets the permitted increase in running time. More precisely, the parameter limits the number of spurious hits caused by shorter seeds.

Seed-and-extend methods find matching substrings between S and T by using a lookup table. For every length-k word, the table stores a list of positions where it is seen in S. Our method creates shorter seeds by merging lists in the lookup table for words with common prefixes. We estimate the increase in spurious hits by using actual word counts in S and the letter frequencies observed in T. The list merging procedure uses the concept of a seed tree, described in Section 2. The technique is not restricted to fixed-length substrings but can be used in conjunction with spaced seeds, as reviewed in Section 3. The procedure modifies the lookup table, and thus can be incorporated in most seed-and-extend algorithms. Our proposed method is fast and requires only a small amount of additional memory. Specifically, it can be implemented for DNA sequences with the help of about $\frac{2}{3}4^k$ integer variables (the lookup table on its own uses $|S|+4^k$ integers), an increase of about 11 megabytes for $k=11$ over the basic method. Section 4 analyzes the method's space and time requirements in detail. Section 5 describes the results of a few experiments. Section 6 concludes with a discussion of related methods and future research.

1.1 Seed-and-extend methods

We use the following notation. We are interested in sequences over a finite alphabet Σ. The length of a sequence S is denoted by $|S|$. The set of all sequences with length m is denoted by Σ^m. The concatenation of two sequences S and T is denoted by $S \cdot T$. The notation $S[i]$ stands for the i-th character of S, and $S[i..j]$ means the substring of S starting with its i-th character and ending with its j-th character.

Seed-and-extend methods rely on a hashing function $h\colon \Sigma^\ell \mapsto \Sigma^k$. Pairs of positions (i,j) are identified in which $h(S[i..i+\ell-1]) = h(T[j..j+\ell-1])$. Such pairs are called *hits*. Hits are further extended into local alignments. In the simplest case, the similarity search uses the identity function for hashing (and thus $\ell = k$): hits are identified by identical substrings. Such fixed length substrings are called *k-mers*. The success of the seed-and-extend approach is due to the speed in which hits can be identified. The first step of the search constructs a hash table using a window of length ℓ that slides over S. For every position $i = 1, \ldots, |S|-\ell+1$, the hash key $h(S[i..i+\ell-1])$ is calculated: the table contains the $\left\langle h(S[i..i+\ell-1]), i \right\rangle$ pairs. In other words, the *key occurrence list* $\mathsf{Occ}_S(u)$ is calculated for every possible hash key $u \in \Sigma^k$, defined as

$$\mathsf{Occ}_S(u) = \Big\{ i \colon h(S[i..i+\ell-1]) = u \Big\}. \tag{1}$$

In the second step of the search, a window slides over T. In each position $j = 1, \dots, |T| - \ell + 1$, the hits (i, j) are considered for extension where i is in the occurrence list $\mathsf{Occ}_S\Big(h(T[j..j+\ell-1])\Big)$.

It was discovered recently [9] that *spaced seeds* yield better sensitivity than k-mers. A spaced seed is defined by an ordered set $\mathcal{S} = \{s_1, s_2, \dots, s_k\} \subseteq \{1, \dots, \ell\}$. Such a seed is referred to as an (ℓ, k)*-seed*. The corresponding hashing function is $h(u) = u[s_1] \cdot u[s_2] \cdots u[s_k]$, i.e., it samples the positions specified by $\mathcal{S}$. Parameter k is called the *weight* of the seed. The use of a (k, k)-seed corresponds to the case of using k-mers. From this point on we assume that the hash function h is defined by an (ℓ, k)-seed.

1.2 Spurious hits

Using a simple model of random sequences, we estimate the number of spurious hits that spaced seeds generate. Assume that S is a random sequence of independent and identically distributed (i.i.d.) characters, specified by the distribution $\mathbb{P}\{S[i] = \sigma\} = p_\sigma$ for all $i = 1, \dots, |S|$ and $\sigma \in \Sigma$. Similarly, assume that T is a sequence of i.i.d. characters, specified by the distribution $\mathbb{P}\{T[j] = \sigma\} = q_\sigma$ for all $j = 1, \dots, |T|$ and $\sigma \in \Sigma$. Let $\mathcal{S} = \{s_1, s_2, \dots, s_k\}$ be the spaced seed that defines the hash function h. The number of window positions in which a specific key u is encountered in S equals

$$n_S(u) = \Big|\mathsf{Occ}_S(u)\Big| = \sum_{i=1}^{|S|-\ell+1} \prod_{j=1}^{|u|} \Big\{S[i + s_j - 1] = u[j]\Big\},$$

where $\{\cdot\}$ stands for the indicator function. Hence the expected number of such positions is

$$\mathbb{E} n_S(u) = (|S| - \ell + 1) \prod_{j=1}^{|u|} p_{u[j]}. \tag{2}$$

The expected number of hits produced by u is

$$H_{S,T}(u) = \mathbb{E}[n_S(u) n_T(u)] = (|S| - \ell + 1)(|T| - \ell + 1)\left(\prod_{j=1}^{|u|} (p_{u[j]} q_{u[j]})\right), \tag{3}$$

where we used the independence of the two sequences. The expected total number of hits is

$$H_{S,T} = \sum_{u \in \Sigma^k} H_{S,T}(u) = (|S| - \ell + 1)(|T| - \ell + 1)\left(\sum_{\sigma \in \Sigma} p_\sigma q_\sigma\right)^k = NM\beta^k \tag{4}$$

with $N = |S| - \ell + 1$, $M = |T| - \ell + 1$, and $\beta = \sum_{\sigma \in \Sigma} p_\sigma q_\sigma$. If at least one of the sequences is uniform, i.e., if $\forall\sigma\colon p_\sigma = 1/|\Sigma|$ or $\forall\sigma\colon q_\sigma = 1/|\Sigma|$, then $\beta = 1/|\Sigma|$. However, genomic sequences often display biased nucleotide

frequencies. For instance, the mouse genome [11] has a a 42% ($\mathtt{G}+\mathtt{C}$) content, and ($\mathtt{G}+\mathtt{C}$) content varies between 38% and 48% on different human chromosomes. In order to model the effect of a given ($\mathtt{G}+\mathtt{C}$) content $f_{\mathtt{G+C}}$, let $p_{\mathtt{A}} = p_{\mathtt{T}} = (1 - f_{\mathtt{G+C}})/2$, $p_{\mathtt{C}} = p_{\mathtt{G}} = f_{\mathtt{G+C}}/2$, and $q_\sigma = p_\sigma$. Equation (4) then gives $H_{S,T} = NM\Big(\frac{1}{2} - f_{\mathtt{G+C}}(1 - f_{\mathtt{G+C}})\Big)^k$. For $f_{\mathtt{G+C}} = 40\%$, we obtain $H_{S,T} = NM \cdot 3.85^{-k}$ and for $f_{\mathtt{G+C}} = 30\%$, the formula gives $H_{S,T} = NM \cdot 3.45^{-k}$. The base of the exponential may actually be larger than 4, when e.g., S has $(\mathtt{G}+\mathtt{C}) > 50\%$ and T has $(\mathtt{G}+\mathtt{C}) < 50\%$.

2 Variable length seeds

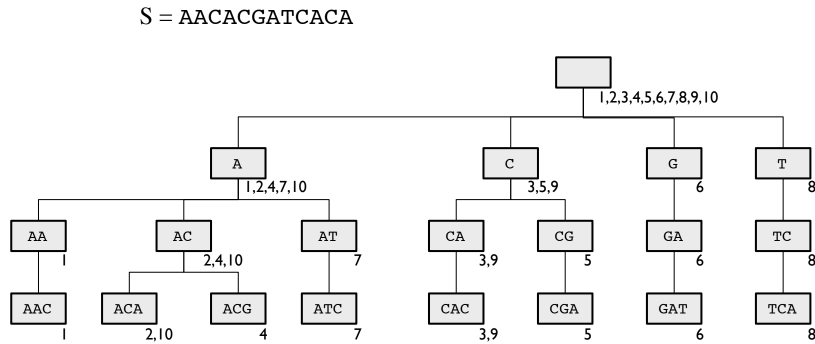

Fig. 1. Example of a seed tree built over 3-mers in the sequence S. Numbers at the nodes are the window positions at which the node's t-mer is seen. Only those nodes are shown that are sampled by at least one sequence position.

A *seed tree* is obtained by placing the hash keys in a trie data structure [12]. Specifically, the seed tree is a rooted tree defined as follows. Every non-leaf node has a child corresponding to every character $\sigma \in \Sigma$. The root is at level 0. Nodes at level $t = 0, \ldots, k$ correspond to t-mers. The children of node $u \in \Sigma^t$ are $\{u \cdot \sigma \colon \sigma \in \Sigma\}$. The leaves are (initially) at level k. Figure 1 shows an example of a seed tree. We now define the seeds corresponding to higher levels, and extend Equation (1) to inner nodes. Recall the definition of the occurrence list for a key u, $\mathsf{Occ}_S(u) = \Big\{i \colon h(S[i..i+\ell-1]) = u\Big\}$. The list Occ is defined recursively by $\mathsf{Occ}_S(v) = \cup_{\sigma\in\Sigma}\mathsf{Occ}_S(v\cdot\sigma)$ whenever $|v| < k$. It is not hard to see that in the case of hashing with k-mers, this definition corresponds to the idea that the lists Occ_S at level t are produced by the t-mers in S. Accordingly,

$\mathsf{Occ}_S(u) = \Big\{ i \colon i \leq |S| - k + 1, S[i..i+|u|-1] = u \Big\}$ holds for all nodes u in that case.

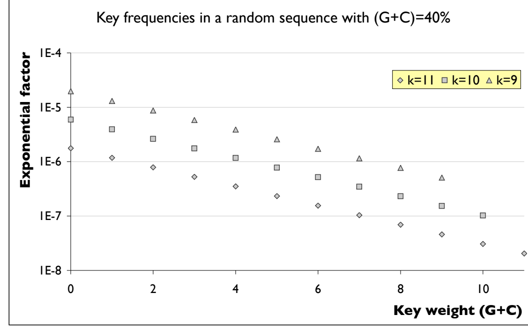

Fig. 2. Frequency of k-mers in the case of non-uniform nucleotide distribution. Under the assumption that S is a sequence of i.i.d. nucleotides with frequencies $p_{\mathtt{A}} = p_{\mathtt{T}} = 0.3$ and $p_{\mathtt{C}} = p_{\mathtt{G}} = f_{\mathtt{G+C}}/2 = 0.2$, Equation (2) predicts that a k-mer with t $(\mathtt{G}+\mathtt{C})$ content is encountered $(|S| - \ell + 1)\alpha$ times where $\alpha = \frac{f_{\mathtt{G+C}}^t (1-f_{\mathtt{G+C}})^{k-t}}{2^k}$. The plot shows α as a function of t for $k = 9, 10, 11$. For example, a 9-mer with eight $(\mathtt{G}+\mathtt{C})$ is seen about as often as a 10-mer with five $(\mathtt{G}+\mathtt{C})$ or an 11-mer with two $(\mathtt{G}+\mathtt{C})$.

Equation (3) is particularly interesting when the nucleotide frequencies are not uniform. Figure 2 shows the frequency of different k-mers in a sequence with 40% $(\mathtt{G}+\mathtt{C})$ content. The frequencies may vary widely for a given k. Given the fact that some shorter k-mers are rarer than others, we prune the seed tree and use shorter or longer k-mers depending on the number of hits they generate. In every pruning operation, a node is selected whose children are leaves, and the children are removed. Our aim is to balance the number of hits generated by the leaves. After a certain amount of pruning, we stop, and use the modified seed tree for finding hits. Using a sliding window over T, instead of looking for the key $h(T[j..j+\ell-1])$ in every position j, we find its longest prefix u that is in the pruned seed tree. The corresponding hits are $\mathsf{Occ}_S(u) \times \{j\}$.

The key to the pruning procedure is the criterion by which nodes are selected. Our criterion relies on predicting the increase in the number of hits at each pruning. First, the sequence S is analyzed to build the key occurrence lists at the leaf level. At the same time $n_S(u) = |\mathsf{Occ}_S(u)|$ is calculated for all $u \in \Sigma^k$. For inner nodes we rely on the recursions

$$\mathsf{Occ}_S(v) = \bigcup_{\sigma \in \Sigma} \mathsf{Occ}_S(v \cdot \sigma); \qquad \text{and} \qquad n_S(v) = \sum_{\sigma \in \Sigma} n_S(v \cdot \sigma). \tag{5}$$

The number of hits a node $v \in \Sigma^t$ generates is predicted as

$$\mathsf{hits}(v) = M n_S(v) \prod_{j=1}^{|v|} \hat{q}_{v[j]},$$

where $M = |T| - \ell + 1$ and $\hat{q}_\sigma$ are the nucleotide frequencies observed in T. Notice that while Equation (3) calculates the number of hits from letter frequencies in S and T, $\mathsf{hits}(v)$ is the expected number of hits, given observed key frequencies in S, and observed letter frequencies in T. The increase in the number of hits when pruning at node v is predicted as

$$\mathsf{hits}^+(v) = \mathsf{hits}(v) - \sum_{\sigma \in \Sigma} \mathsf{hits}(v \cdot \sigma) = M \left(\prod_{j=1}^{|v|} \hat{q}_{v[j]} \right) \sum_{\sigma \in \Sigma} (1 - \hat{q}_\sigma) n_S(v \cdot \sigma), \quad (6)$$

where we employed the recursion of (5). We keep on pruning the tree, always selecting a node that increases the number of hits by the least amount, until the total predicted increase surpasses a threshold parameter.

Procedure EstimateHits
Input: hash key size k, occurrence lists $\mathsf{Occ}_S(\cdot)$, character frequencies $\hat{q}$, sequence length M
Output: places every $u \in \Sigma^{<k}$ in the appropriate bin by setting up the linked lists BinList

H1 set total $\leftarrow 0$; **for** all $b \leftarrow 0, \ldots, B$ **do** set BinList $\leftarrow \emptyset$
H2 **for** all $u \in \Sigma^{k-1}$ **do**
H3 set $\mathsf{n}_\sigma \leftarrow |\mathsf{Occ}_S(u \cdot \sigma)|$ for all $\sigma \in \Sigma$ // $\mathsf{n}_\sigma = n_S(u \cdot \sigma)$
H4 set $\mathsf{h} \leftarrow M \left(\prod_{i=1}^{|u|} \hat{q}_{u[i]} \right)$
H5 set $\mathsf{total} \leftarrow \mathsf{total} + \mathsf{h} \sum_{\sigma \in \Sigma} \hat{q}_\sigma \mathsf{n}_\sigma$
H6 set $n_S(u) \leftarrow \sum_{\sigma \in \Sigma} \mathsf{n}_\sigma$
H7 set $\mathsf{H} \leftarrow \mathsf{h} \sum_{\sigma \in \Sigma} (1 - \hat{q}_\sigma) \mathsf{n}_\sigma$ // $\mathsf{H} = \mathsf{hits}^+(u)$
H8 add u to the end of $\mathsf{BinList}[b(\mathsf{H})]$
H9 **for** $t \leftarrow k-2, \ldots, 0$ **do**
H10 **for** all $u \in \Sigma^t$ **do**
H11 set $n_S(u) \leftarrow \sum_{\sigma \in \Sigma} n_S(u \cdot \sigma)$
H12 set $\mathsf{H} \leftarrow M \left(\prod_{i=1}^{|u|} \hat{q}_{u[i]} \right) \sum_{\sigma \in \Sigma} (1 - \hat{q}_\sigma) n_S(u \cdot \sigma)$
H13 add u to the end of $\mathsf{BinList}[b(\mathsf{H})]$
H14 **return** total. // (*returns number of hits without pruning*)

Fig. 3. Procedure EstimateHits. EstimateHits places the inner nodes of the seed tree in the appropriate bins, based on the number of hits they produce. Notice that $n_S(\cdot)$ is not needed outside this procedure.

In order to carry out the pruning, we use a binning procedure. Proceeding from the leaves towards the root, we calculate $n_S(v)$ using Equation (5), and

$\mathsf{hits}^+(v)$ using Equation (6). Using a monotone binning function $b\colon [0,\infty) \mapsto \{0,1,\ldots,B\}$, we place every tree node v into the bin $b(\mathsf{hits}^+(v))$. The pruning is carried out by selecting nodes first from bin 0, then from bin 1, and so on, until the threshold for the number of hits is reached, or there are no more nodes available. Lemma 1 shows that the increase in the number of hits is larger at parents than it is at the children unless the sequence T consists mostly of the same character. The importance of this lemma is that it suggests how nodes should be stored: using a linked list at each bin, nodes are added at the list tail as we calculate hits^+ from the leaves upwards. Pruning proceeds by traversing the list in each bin. By Lemma 1, whenever a node is considered for pruning, all of its descendants are pruned already.

Lemma 1. *If $\hat{q}_\sigma \le 1/2$ for all $\sigma \in \Sigma$, then for all v with $|v| < k-1$ and $x \in \Sigma$, $\mathsf{hits}^+(v) \ge \mathsf{hits}^+(v \cdot x)$.*

Proof. By Equation (6), $\mathsf{hits}^+(v) \ge \mathsf{hits}^+(v \cdot x)$ if and only if

$$\sum_{\sigma\in\Sigma}(1-\hat{q}_\sigma)n_S(v\cdot\sigma) \ge \hat{q}_x \sum_{\sigma'\in\Sigma}(1-\hat{q}_{\sigma'})n_S(v\cdot x\cdot\sigma'). \qquad (*)$$

We prove that the term for $\sigma = x$ on the left-hand side already ensures the inequality, i.e., that

$$(1-\hat{q}_x)n_S(v\cdot x) \ge \hat{q}_x \sum_{\sigma'\in\Sigma}(1-\hat{q}_{\sigma'})n_S(v\cdot x\cdot\sigma'), \qquad (**)$$

which implies (*) and thus the lemma. By Equation (5), $n_S(v\cdot x) = \sum_{\sigma'} n_S(v\cdot x\cdot \sigma')$. Consequently, Equation (**) holds if $(1-\hat{q}_x)n_S(v\cdot x\cdot\sigma') \ge \hat{q}_x(1-\hat{q}_{\sigma'})n_S(v\cdot x\cdot\sigma')$, that is, if $(1-\hat{q}_x) \ge \hat{q}_x(1-\hat{q}_{\sigma'})$ for all σ'. Since $\hat{q}_x \le \frac{1}{2}$, $1-\hat{q}_x \ge \hat{q}_x$, and Equation (**) follows. □

Figure 3 sketches the procedure used to estimate the increased number of hits, and to place the nodes in their appropriate bins. Figure 4 sketches the pruning itself. It is important to notice that the seed tree is not used later when hit extension is performed, but rather that Line P5 updates the lookup table entry $\mathsf{Occ}_S(u)$ for every leaf in the subtree rooted at an inner node v when pruning at v. Line P4 calculates the parent's list by merging the lists at the children to preserve the ordering of positions. (It is often exploited during hit extension that positions are listed in decreasing order for each key.)When T is finally processed, we can use the lists $\mathsf{Occ}_S(u)$ directly with every hash key u. This technique allows for interfacing with other lookup table-based algorithms. Procedures EstimateHits and Prune update the hash table after it is constructed, and the search can proceed as usual without any additional data structures.

Figure 5 sketches a possible way to incorporate the seed tree pruning procedures in a local similarity search algorithm. It follows the logic of such tools as BLAST [6] and PatternHunter [9]. The difference is in Lines A4–A6, where the hash table is updated based on the seed tree pruning procedure. Lines A1–A3 build the hash table as usual. Lines A7–A10 find hits and extend just as if the pruning never happened.

	Procedure Prune
	Input: total increase in hits max, list BinList of nodes in every bin
	Output: pruned seed tree
P1	set increase $\leftarrow 0$ and bin $\leftarrow 0$
P2	**while** increase $<$ max and bin $\leq B$ **do**
P3	**for** all $u \in$ BinList[bin] starting at the list head, **do**
P4	set $\mathsf{Occ}_S(u) \leftarrow \cup_{\sigma\in\Sigma}\mathsf{Occ}_S(u\cdot\sigma)$
P5	set $\mathsf{Occ}_S(v) \leftarrow \mathsf{Occ}_S(u)$ for every leaf v in the subtree of u
P6	set increase $\leftarrow$ increase $+\, b^{-1}$(bin)
P7	**if** increase $\geq$ max **then return**
P8	set bin $\leftarrow$ bin $+ 1$

Fig. 4. Procedure Prune. Line P5 implements the pruning: the lists at the leaves are updated to accelerate the ensuing extension process when scanning T. (Accordingly, for every inner node u, $\mathsf{Occ}_S(u)$ is stored physically at the leftmost leaf in its subtree.) Notice that BinList is not needed after the pruning. The expression b^{-1}(bin) in Line P6 denotes the average value of hits^+ in bin.

3 Spaced seeds

A seed tree can be used immediately in conjunction with a spaced seed $\mathcal{S} = \{s_1, \cdots, s_k\}$, since the pruning works regardless of how the hash keys are obtained. It is advantageous, however, to select a good permutation $s_{i_1}, \ldots, s_{i_k}$ of the sampled positions that maximizes sensitivity. For a permutation $i_1, \ldots, i_k$, define the series of seeds $\mathcal{S}_t$ for $t = 0, \ldots, k$ by $\mathcal{S}_0 = \emptyset$ and $\mathcal{S}_t = \mathcal{S}_{t-1} \cup \{s_{i_t}\}$ for $t = 1, \ldots, k$. The keys at level t of the seed tree correspond to hashing with the spaced seed $\mathcal{S}_t$. An optimal permutation is obtained by iteratively selecting $i_k, i_{k-1}, \ldots, i_1$. Starting with $\mathcal{S}_k = \mathcal{S}$, $\mathcal{S}_{t-1}$ is computed by selecting the element of $\mathcal{S}_t$ that can be removed to obtain the highest increase in sensitivity. Sensitivity is measured as the probability of detecting a region of a given length and similarity. The probability is calculated under the assumption that every position of the region mutates independently with the same probability. Formally, the following model is used. Assume that a region of length $L \geq \ell$ of similarity g is "hidden" in the two sequences: there exist i_0, j_0 such that $S[i_0..i_0 + L - 1]$ and $T[j_0..j_0 + L - 1]$ are identical in about gL positions. More precisely, the corresponding substrings have such a distribution that for all $\sigma, \sigma' \in \Sigma$, and position offsets $j = 1, \ldots, L$,

$$\mathbb{P}\Big\{T[j_0 + j - 1] = \sigma' \;\Big|\; S[i_0 + j - 1] = \sigma\Big\} = \begin{cases} g & \text{if } \sigma = \sigma'; \\ \frac{1-g}{|\Sigma|-1} & \text{if } \sigma \neq \sigma', \end{cases}$$

and that different positions are independent. Sensitivity of a hashing function is measured as the probability of producing a hit within this region, i.e., the probability that there exists $i \in \{-s_1+1, -s_1+2, \ldots, L-s_k\}$ for which $h(S[i_0+i..i_0+i+\ell-1]) = h(T[j_0+i..j_0+i+\ell-1])$. The probability of detecting such a homology region by a given seed can be calculated explicitly using an appropriately defined Markov chain [13, 14].

Algorithm CompareSequences
Input: sequences S, T, hashing function $h\colon \Sigma^\ell \mapsto \Sigma^k$, relative increase in running time R
A1 initialize $\mathsf{Occ}_S(u) \leftarrow \emptyset$ for all $u \in \Sigma^k$
A2 **for** all $i \leftarrow 1, \ldots, |S| - \ell + 1$ **do**
A3 set $u \leftarrow h(S[i..i + \ell - 1])$ and add i to $\mathsf{Occ}_S(u)$
A4 calculate $M = |T| - \ell + 1$, and $\hat{q}_\sigma = \frac{\sum_{j=1}^{|T|} \{T[j]=\sigma\}}{|T|}$ for all $\sigma \in \Sigma$
A5 set $\mathsf{T} \leftarrow \mathsf{EstimateHits}(k, \mathsf{Occ}_S, \hat{q}, M)$ // T *is the number of spurious hits without pruning*
A6 do $\mathsf{Prune}(R\mathsf{T})$
A7 **for** $j \leftarrow 1, \ldots, |T| - \ell + 1$ **do**
A8 set $u \leftarrow h(T[j..j + \ell - 1])$
A9 **for** all $i \in \mathsf{Occ}_S(u)$ **do**
A10 extend the hit at (i, j) and report the alignment if significant

Fig. 5. The comparison algorithm.

We implemented the Markov chain method and calculated the optimal permutation for a number of seeds. Table 1 shows the seeds used in our experiments. At first sight it may seem that it is best to remove the leftmost or rightmost positions from the seed, since that increases the expected number of hits within the similarity region (which is $(L - (\max \mathcal{S} - \min \mathcal{S} + 1))g^{|\mathcal{S}|}$). Interestingly, this is not always the case: for instance, the sensitivity of the $(18, 12)$-seed increases the most by removing a sampling position in the middle.

Clearly, the principle of choosing a permutation based on sensitivity is not restricted to the case of independent mutations and constant mutation rates. In particular, it can be applied in conjunction of other, more sophisticated models for assessing sensitivity, such as the Markov models for coding regions employed in [14] and [15].

4 Performance analysis

Theorem 1. *The procedures* EstimateHits *and* Prune *can be implemented using* $2\frac{|\Sigma|^k-1}{|\Sigma|-1} + O(1)$ *integer variables.*

Proof. Procedure EstimateHits has to keep track of $n_S(u)$ for every non-leaf node u. The number of non-leaf nodes in the seed tree equals $\sum_{t=0}^{k-1} |\Sigma|^t = \frac{|\Sigma|^k-1}{|\Sigma|-1}$. Procedure Prune relies on the BinList array of linked lists. The chaining within the lists can be implemented by using an array that gives the next element after each non-leaf node u in the bin that contains u. Therefore, $\frac{|\Sigma|^k-1}{|\Sigma|-1}$ integers can be used for the chaining if every node address is encoded by an integer value. An additional array of $(B + 1)$ elements specifies the first element in each bin. The total number of integer variables is therefore $2\frac{|\Sigma|^k-1}{|\Sigma|-1} + (B+1) + O(1)$, where the $O(1)$ term accounts for auxiliary local variables. □

As discussed in [9], the lookup table can be implemented so that every occurrence list $\mathsf{Occ}_S(u)$ is a linked list of positions in decreasing order, with the aid of $(|S|+|\Sigma|^k)$ integers. (In an array of size $|\Sigma|^k$, the last position is given where each hash key is seen in S. Another array of size $|S|$ gives the previous position for every i where the hash key $h(S[i..i+\ell-1])$ is seen.) Theorem 1 thus implies that the DNA similarity search algorithm of Figure 5 can be implemented with $(4|S|+\frac{5}{3}4^{k+1}+\epsilon)$ bytes (allowing four bytes per integer) in addition to the storage of the input sequences. The term ϵ accounts for the runtime environment, all local variables, and the data structure tracking recent extensions in Line A10. As an illustration of space efficiency, if S and T are around 150 Mbp in length, if every nucleotide is stored in one byte, and if $k=11$, then the algorithm uses less than 1 Gbytes of memory. In our experiments with trees over the $(18,12)$-seed comparing human and rat X chromosomes (152 and 163 million letters), the maximum memory usage was about 990MB.

Theorem 2. *$\mathsf{EstimateHits}$ and Prune update the lookup table in $O\Big(k|S|+k|\Sigma|^k\Big)$ time.*

Proof. $\mathsf{EstimateHits}$ executes Line H3 for every leaf node. The total time spent counting the size of every list Occ in Line H3 is thus $O(|\Sigma|^k+|S|)$. Lines H11–H13 take $O(|\Sigma|+t)$ time for every node at level $t=0,\ldots,k-2$, and thus the total time of executing the loop is of $O(k|\Sigma|^{k-2}+|\Sigma|^{k-1})$. $\mathsf{EstimateHits}$ thus runs in $O(k|\Sigma|^{k-2}+|\Sigma|^k+|S|)$ time. In an extreme scenario, procedure Prune is called with $\mathsf{max}=\infty$, and all nodes are pruned. Since list merging in Line P4 takes linear time in the sum of the list lengths, pruning all nodes at level t takes $O(|S|)$ time. Hence the total time spent on merging in Line P4 is $O(|S|k)$. Line P5 takes $O(1)$ time for every leaf v, and thus $O(|\Sigma|^k)$ time is spent in Line P5 for every tree level. Consequently, Prune runs in $O(|\Sigma|^k k+|S|k)$ time. □

It is worth pointing out that in contrast to the worst-case analysis of Theorem 2, pruning should be carried out with a parameter $R=O(\beta^{-1})$ where β is defined in Equation (4). Otherwise, too many nodes of the seed tree are pruned, resulting in more hits than a shorter seed would produce. Since the shorter seed would require less memory and might provide better sensitivity, it is better to avoid large R values. For a reasonable choice of R, procedure Prune takes a time of $O(|\Sigma|^k+|S|)$, rather than $O(k)$ times as much, as pruning takes place only on one or a few tree levels.

5 Experiments

We implemented the algorithm in Figure 5 to test its performance on DNA sequences. We carried out a number of experiments with our prototype implementation with two goals in mind. First, we wanted to assess how valid the estimation of spurious hits is for biological sequences, and whether the fine-tuning of specificity vs. sensitivity is truly achieved by using a single parameter.

Seed	Ordered positions $s_1, s_2, \ldots, s_k$	Remark
$(20, 13)$-seed	$10, 9, 6, 4, 3, 14, 16, 18, 19, 2, 13, 1, 20$	Best weight-13 seed
$(18, 12)$-seed	$2, 3, 8, 10, 13, 14, 5, 1, 16, 17, 18, 7$	Best weight-12 seed
$(18, 11)$-seed	$2, 3, 8, 10, 13, 14, 5, 1, 16, 17, 18$	Best weight-11 seed (PatternHunter)
$(18, 10)$-seed	$1, 2, 3, 6, 9, 12, 14, 16, 17, 18$	Best (18,10)-seed
$(16, 10)$-seed	$1, 2, 4, 5, 9, 10, 12, 14, 15, 16$	Best weight-10 seed

Table 1. Seeds used in our experiments. "Best" refers to highest sensitivity, measured by the probability of detecting a 70%-similarity region of length 64. Sampled positions are ordered by maximizing the sensitivity of shortened seeds as described. The weight-10 seeds were not used in conjunction with tree pruning, and thus their permutation is arbitrary.

Secondly, we wanted to measure how much time the tree pruning takes with respect to other steps in the search. We implemented only gapless seed extension since our aim was not to develop another similarity search tool, but to produce a testbed for seed tree pruning. The experiments rely on a simple scoring function with a match score of 1 and mismatch penalty of 1. The extension calculates high-scoring segment pairs (HSPs) for every hit (i, j). The extensions are found by exploring the diagonal of (i, j) in two directions until the score drops below a threshold. The highest scoring segment is selected in both directions to obtain the HSP. The significance of the alignment is assessed using standard techniques [16]. We keep track of the longest extension along each diagonal, and attempt the extension only if the hit does not overlap with a previously explored region. In all experiments we used a cutoff of $E = 0.1$ for defining HSPs, i.e., we considered local alignments that have a score that is expected to occur 0.1 times in the sequence comparison.

The seeds we used in the experiments are shown in Table 1. In a first set of experiments, we compared the genome sequence of *H. influenzae* (1.8 million base pairs) to that of *E. coli* (4.6 Mbp). Figure 6a shows the accuracy of predicting the number of spurious hits using two seed trees, one built over the $(18, 12)$-seed and the other over the $(18, 11)$-seed. The parameter R predicts the increase in number of spurious hits very well[1]. Figure 6b plots the number of HSPs in function of the number of hits. The plot shows that the seed trees attain intermediate levels of sensitivity. In fact, the seed trees perform better than the weight-10 seeds.

We also compared the concatenated genome sequence of the budding yeast *S. cerevisiae* (12 Mbp) to the concatenated genome sequence of the fission yeast *S. pombe* (12.5 Mbp). Figure 7a shows the accuracy of predicting the number of spurious hits. Again, the predicted and measured values of specificity are very close. Figure 7b plots the number of HSPs for different levels of specificity: the

[1] Using the seed tree over the $(18, 11)$-seed as an example, Equation (4) predicts $H \approx 1.9 \cdot 10^6$ spurious hits (*H. influenzae* genome has a 38% ($\mathtt{G} + \mathtt{C}$) content, while that of *E. coli* is close to 50%). The $(18, 11)$-seed finds $2.6 \cdot 10^6$ hits without pruning. We expect therefore that the number of hits grows as $H(R) = 2.6 \cdot 10^6 + 1.9 \cdot 10^6 R$.

seed tree over the $(18, 12)$-seed falls behind the $(18, 11)$-seed, but the seed tree over the latter slightly outperforms a weight-10 seed.

The final example of the method's application is the comparison of human and rat X chromosomes (152 and 163 million base pairs, respectively). We concatenated the sequence contigs for each chromosome, using sequences that were masked for repeat and low-complexity regions. We ignored masked regions for hit generation but included them for hit extension. The experimental results are summarized in Figure 8. Again, the parameter accurately predicts the increases in running time, and specificity levels between those of spaces seeds are reached.

The time it takes to carry out the pruning is comparable to the time of building the lookup table. Table 2 shows measured running times. (The hit extension in our implementation is not optimized, and only gapless extensions are found, but comparable running times were reported for PatternHunter [9], which does construct gapped extensions.) The time increments for the X chromosomes are particularly instructive, as the chromosomes are of average size in the two genomes. Comparison of the entire human and rat genomes would take about four hundred times more. Using weight-13, 12, and 11 seeds, the chromosome comparison takes $\frac{1}{2}$, 1.25, and 4 hours without pruning. Multiplying by 400, the genome comparisons would take 8, 21, or 67 CPU days. Seed trees provide a flexible way to perform the genome comparisons at various levels of specificity in addition to these three very different choices.

	Table	Pruning	Extensions
H. influenzae–*E. coli*, (18,12)-seed $R = 0$	2s	0s	8s
H. influenzae–*E. coli*, (18,12)-seed $R = 3.5$	2s	9s	15s
H. influenzae–*E. coli*, (18,11)-seed $R = 3.5$	2s	3s	31s
S. cerevisiae–*S. pombe*, (18,12)-seed $R = 0$	13s	0s	3m 31s
S. cerevisiae–*S. pombe*, (18,12)-seed $R = 3.5$	14s	21s	6m 27s
S. cerevisiae–*S. pombe*, (18,11)-seed $R = 3.5$	12s	13s	32m 24s
Human chrX–rat chrX, (20,13)-seed $R = 0$	82s	0s	31m
Human chrX–rat chrX, (20,13)-seed $R = 3.5$	76s	6m 34s	78m
Human chrX–rat chrX, (18,12)-seed $R = 3.5$	119s	105s	4h 21m

Table 2. Measured running times for different parts of the comparison algorithm, excluding input-output. Column "Table" shows the time it takes to build the initial hash table after reading the first sequence. Column "Pruning" gives the running time for EstimateHits and Prune procedures. Column "Extensions" gives the running time for processing all hits and calculating HSPs. Timing was done on an Apple PowerBook 1.25GHz G4 with 1 GBytes of memory running MacOS X, using Java's `System.currentTimeMillis()` method.

6 Discussion

Seed-and-extend methods, exemplified by the NCBI BLAST suite, have been thoroughly successful in providing a way to perform highly sensitive similarity searches between long molecular sequences in reasonable time. Traditional seed-and-extend methods use fixed length hash keys. Our proposed method creates a hashing function with varying key lengths, based on statistics gathered from the input sequences. Shorter keys are introduced to maximize the sensitivity with a predictable compromise of specificity. We introduced the concept of the seed tree, which guides the selection of shorter seeds. After gathering statistics about the hash keys, and pruning, the seed tree is not needed anymore. The second sequence in the comparison is processed using the updated lookup table, and thus hash keys and their occurrence lists are still found in $O(1)$ time.

There are many variations on the basic seed-and-extend idea that affect the sensitivity of the search. We consider two such variations: multiple spaced seeds [10, 14] and vector seeds [17]. Li *et al.* [10] explore the use of multiple spaced seeds in homology searches. In order to use a set of m seeds, one needs to construct a lookup table for every seed, requiring enough memory for $(m|S|+m|\Sigma|^k)$ integers. As a consequence, long sequences may need to be split into smaller segments. The number of spurious hits grows linearly with m, allowing for no fine-tuning between consecutive values of m. Our method can be used instead of multiple seeds to control the specificity on a fine scale. It can also be used in conjunction with multiple lookup tables in the same way as with a single one: a seed tree is pruned for every table. In order to track the number of hits created, the procedure Prune would have to be modified to consider the seed trees built for the different seeds simultaneously.

Brejová *et al.* [17] consider relaxing the definition of a match between hash keys by introducing the concept of vector seeds. Vector seeds permit weighted mismatches between hash keys for producing hits. Relaxing or tightening the hit criteria allows for reaching different levels of sensitivity and specificity. Our approach of pruning a seed tree creates new hits in a data-dependent manner, and offers a way of setting different levels of specificity for different keys. Brudno *et al.* [18] also use a trie to manipulate hash keys, but do not consider either variable length keys, or pruning.

While our implementation focused on nucleotide sequences, we attempted to present the idea of pruning seed trees independently of the underlying alphabet. In particular, our algorithms in Figures 3 and 4 can be implemented for protein sequences, with consequences for space and time requirements as analyzed in Section 4. In the case of proteins, hits are usually defined by similarity rather than identity, so the ramifications of pruning are more complex. We are currently exploring the practicality of our method on other than DNA sequences.

References

1. Miller, W.: Comparison of genomic DNA sequences: solved and unsolved problems. Bioinformatics **17** (2001) 391–397

2. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. J. Mol. Biol. **147** (1981) 195–197
3. Myers, G., Durbin, R.: A table-driven full sensitivity similarity search algorithm. J. Comput. Biol. **10** (2003) 103–117
4. Delcher, A.L., Phillippy, A., Carlton, J., Salzberg, S.L.: Fast algorithms for large-scale genome alignment and comparison. Nucleic Acids Res. **30** (2002) 2478–2483
5. Pearson, W.R., Lipman, D.J.: Improved tools for biological sequence comparison. Proc. Natl. Acad. Sci. USA **85** (1988) 2444–2448
6. 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 Res. **25** (1997) 3389–3402
7. Kent, W.J.: BLAT — the BLAST-like alignment tool. Genome Res. **12** (2002) 656–664
8. Schwartz, S., Kent, W.J., Smit, A., Zhang, Z., Baertsch, R., Hardison, R.C., Haussler, D., Miller, W.: Human-mouse alignments with BLASTZ. Genome Res. **13** (2003) 103–107
9. Ma, B., Tromp, J., Li, M.: PatternHunter: faster and more sensitive homology search. Bioinformatics **18** (2002) 440–445
10. Li, M., Ma, B., Kisman, D., Tromp, J.: PatternHunter II: highly sensitive and fast homology search. Journal of Bioinformatics and Computational Biology (2004) To appear.
11. MGSC: Initial sequencing and comparative analysis of the mouse genome. Nature **420** (2002) 520–562
12. Friedkin, E.: Trie memory. Comm. ACM **3** (1960) 490–500
13. Nicodème, P., Salvy, B., Flajolet, P.: Motif statistics. In Nešetřil, J., ed.: Algorithms — ESA'99: 7th Annual European Symposium. Volume 1643 of LNCS., Heidelberg, Springer-Verlag (1999) 194–211
14. Buhler, J., Keich, U., Sun, Y.: Designing seeds for similarity search in genomic DNA. In Vingron, M., Istrail, S., Pevzner, P., Waterman, M., eds.: Proc. 7th Annual International Conference on Computational Molecular Biology (RECOMB), New York, NY, ACM Press (2003) 67–75
15. Brejová, B., Brown, D., Vinař, T.: Optimal spaced seeds for homologous coding regions. Journal of Bioinformatics and Computational Biology **1** (2004) 595–610
16. Karlin, S., Altschul, S.F.: Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. Proc. Natl. Acad. Sci. USA **87** (1990) 2264–2268
17. Brejová, B., Brown, D., Vinař, T.: Vector seeds: an extension to spaced seeds allows substantial improvements in sensitivity and specificity. In Benson, G., Page, R., eds.: Algorithms and Bioinformatics: 3rd International Workshop (WABI). Volume 2812 of LNCS., Heidelberg, Springer-Verlag (2003) 39–54
18. Brudno, M., Chapman, M.A., Gottgens, B., Batzoglou, S., Morgenstern, B.: Fast and sensitive multiple alignment of large genomic sequences. BMC Bioinformatics **4** (2003) 66

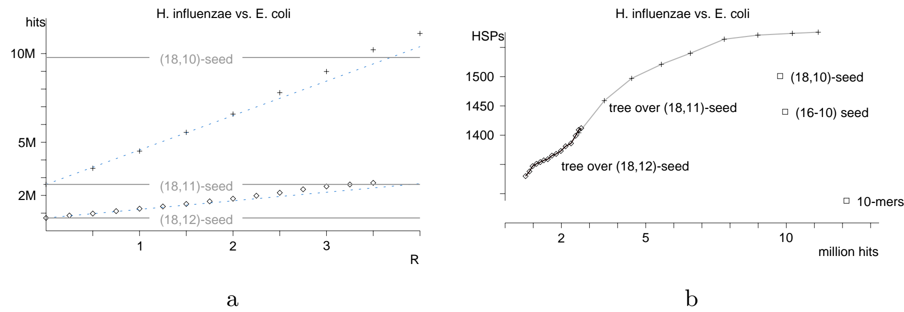

Fig. 6. Comparison of *H. influenzae* and *E. coli* genomes. The left-hand side plots the number of hits in function of the algorithm's parameter R for two seed trees. Each dotted line shows the expected slope of the function. The right-hand side shows the specificity-sensitivity trade-off for HSPs with E-value 10^{-1}. Both curves start at $R = 0$, i.e., where hits are produced by spaced seeds without tree pruning.

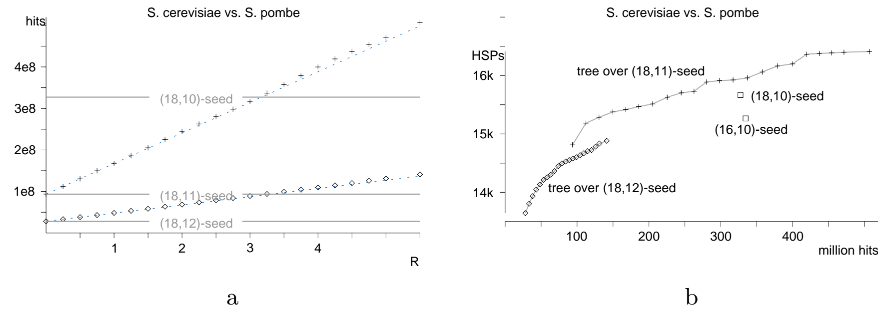

Fig. 7. Comparison of two yeast genomes. The left-hand side plots the number of hits in function of the algorithm's parameter R for two trees. Dotted lines show the expected slopes. The right-hand side shows the specificity-sensitivity trade-off for HSPs with E-value 10^{-1}. Both curves start at $R = 0$.

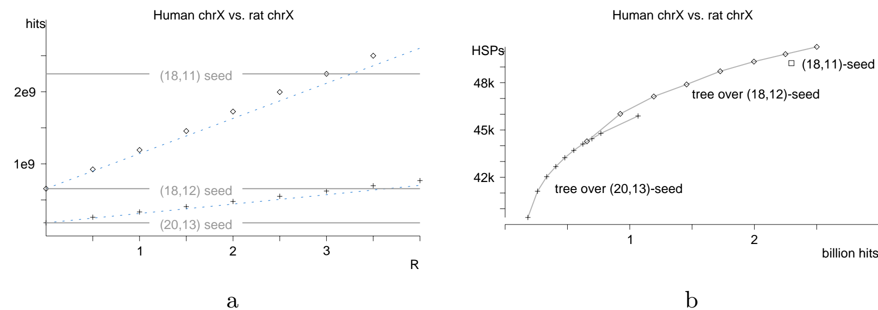

Fig. 8. Comparison of two mammalian X chromosomes. The figure on the left-hand side plots the number of hits for different R values; dotted lines indicate the expected slopes. The right-hand side illustrates the sensitivity-specificity trade-off. Both curves start at $R = 0$.