New Algorithms for the Spaced Seeds

Xin Gao[1], Shuai Cheng Li[1], and Yinan Lu[⋆1,2]

[1] David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 6P7
[2] College of Computer Science and Tecnology of Jilin University
10 Qianwei Road, Changchun, Jilin Province, China 130012
{x4gao, scli}@cs.uwaterloo.ca, luyinan@email.jlu.edu.cn

Abstract. The best known algorithm computes the sensitivity of a given spaced seed on a random region with running time $O((M+L)|B|)$, where M is the length of the seed, L is the length of the random region, and $|B|$ is the size of seed-compatible-suffix set, which is exponential to the number of 0's in the seed. We developed two algorithms to improve this running time: the first one improves the running time to $O(|B'|^2ML)$, where B' is a subset of B; the second one improves the running time to $O((M|B|)^{2.236}log(L/M))$, which will be much smaller than the original running time when L is large. We also developed a Monte Carlo algorithm which can guarantee to quickly find a near optimal seed with high probability.

keyword: homology search, spaced seed, bioinformatics.

1 Introduction

The goal of homology search is to find similar segments or local alignments between biological molecular sequences. Under the framework of match, mismatch, and gap scores, the Smith-Waterman algorithm guarantees to find the global optimal solution. However, the running time of the Smith-Waterman algorithm is too large to be used on real genome data.

Many programs have been developed to speed up the homology search, such as FASTA [11], BLAST [1, 2, 16, 14], MUMmer [8], QUASAR [5], and PatternHunter [12, 9]. BLAST (Basic Local Alignment Search Tool) is the most widely used program to do homology search. The basic idea of BLAST is that by using a length 11 seed, which requires that two sequences have locally 11 consecutive matches, local matches can be found, and a reasonably good alignment can be then generated by extending those local matches. However, Li *et al* [12] found that the homology search sensitivity can be largely improved if long "gapped" seeds are used instead of short "exact" seeds. PatternHunter is developed based on long "gapped" seeds. PatternHunter applies a dynamic programming (DP)

⋆ To whom correspondence should be addressed

based algorithm to compute the hit probability of a given seed. We refer the algorithm to compute the sensitivity of a seed in PatternHunter as the PH algorithm.

The running time of the PH algorithm is dominated by the product of the length of random region and the size of seed-compatible-suffix set. In [9], Li *et al* proved that computing the hit probability of multiple seeds is NP-hard. In [10], Li *et al* further proved that computing hit probability of a single seed in a uniform homologous region is NP-hard.

The problems of computing the sensitivity of a given spaced seed and finding the most sensitive pattern have been studied for a long time. Choi and Zhang and coworkers [7, 13] studied the problem of calculating sensitivity for spaced seeds from computational complexity point of view, and proposed an efficient heuristic algorithm for identifying optimal spaced seeds. Choi *et al.* [6] found that an optimal seed on one sequence similarity level may not be optimal on another similarity level. Yang *et al.* [15] proposed algorithms for finding optimal single and multiple spaced seeds. Brejova *et al.* [3] studied the problem of finding optimal seeds for sequences generated by a Hidden Markov model. Brown [4] formulated choosing multiple seeds as an integer programming problem, and gave a heuristic algorithm. So far, the PH algorithm is still the best running time algorithm for calculating sensitivity of a given spaced seed. And most algorithms for finding the optimal seed can not give any guarantee on the performance.

Here, we develop two algorithms to improve the PH algorithm for some cases. The first algorithm improves the PH algorithm when the size of seed-compatible-suffix set is large, while the second algorithm improves PH algorithm when the region length is large. We further develop a Monte Carlo algorithm which can guarantee to quickly find the optimal seed with high probability.

2 Preliminaries

2.1 Notations and Definitions

The notations are largely followed from those in [9].

Denote the i-th letter of a string s as $s[i-1]$. The length of s is denoted as $|s|$. A spaced seed a is a string over alphabet $\{1, 0\}$. Denote $M = |a|$. For a spaced seed a, we require $a[0] = 1$ and $a[M-1] = 1$. The number of 1's in a is called the weight of a, here denoted as W. A 1 position in a means "required match", while a 0 in a means "do not care".

A homologous region R with length L is defined as a binary string, in which a 1 means a match and a 0 means a mismatch. In this paper, we only focus on random homologous regions with uniform distribution. That is, $Pr(R[i] = 1) = p$, $0 \leq i \leq L-1$, where p is referred to as *similarity level* of R. For a spaced seed a and a homologous region R, if there $\exists\ j$, $0 \leq j \leq L - M$, such that whenever $a[i] = 1$, we have $R[j+i] = 1$, then we say that a *hits* region R.

This paper studies the following two problems:

1. **Seed Sensitivity**: Given a spaced seed a, and a homologous region R , what is the probability of a hitting R? This probability is referred to as the sensitivity of this seed on the region R. We just call it *sensitivity* of R if the context is clear.
2. **Optimal Seed**: Given a seed length M and weight W, and a homologous region R with similarity level p, what is the seed with the highest sensitivity? A seed with the highest sensitivity is called an *optimal seed*.

2.2 Reviews of the PatternHunter Algorithm for Seed Sensitivity

Li *et al* [9] developed a dynamic programming based algorithm to compute the sensitivity of a given seed.

For a seed a of length M and weight W, we call a string b *compatible* with a if $b[|b|-j]=1$ whenever $a[|a|-j]=1$ for $0<j\leq \min\{|a|,|b|\}$. Suppose the random region R has length L, and similarity level p. For a binary string b, let $f(i,b)$ be the probability that seed a hits region $R[0:i-1]$ which has b as the suffix of $R[0:i-1]$. Generally, we are only interested in the case $0\leq|b|\leq M$. There are two cases: 1) the position before b has value 0; and 2) the position before b has value 1. Thus, $f(i,b)$ can be recursively expressed as:

$$f(i,b)=(1-p)f(i,0b)+pf(i,1b) \tag{1}$$

This will generate a $O(L2^M)$ dynamic programming algorithm because length of b is at most M. However, since the only case seed a can hit the "tail" of a region R is that the suffix of the region R is compatible with seed a, instead of considering all possible suffixes, they only consider those suffixes which are compatible with a.

Define B to be the set of binary strings that are not hit by a but compatible with a. Let $B(x)$ denote the longest proper prefix of x that is in B. The PH algorithm thus uses the following recursion function:

$$f(i,b)=(1-p)f(i-|b|+|b'|,0b')+pf(i,1b)\ \ 0b'=B(0b). \tag{2}$$

It is clear that any entry in the dynamic programming table depends on two previously computed entries. Therefore, PH algorithm has to consider all the possible $b\in B$ for each i, $0\leq i\leq L-1$. The size of B is bounded by $M2^{M-W}$. The running time is thus $O((M+L)M2^{M-W})$. However, the size of B can be reduced.

In this paper, we improve the PH algorithm from two perspectives:

- Instead of considering all possible suffixes $b\in B$, we consider only a small subset of B, which will result in an algorithm with a better running time when $|B|$ is large. That is, following the similar idea applied by PH algorithm, we further reduce the number of suffixes needed.
- We reduce the factor L in the running time of PH algorithm to $\log L$, which will generate an algorithm with much better running time when L is large.

3 Suffix-recursion-tree Algorithm

3.1 Algorithmic Details

The basic idea of our suffix-recursion-tree (SRT) algorithm is that if we precompute more steps of the recursion function Eq. 2 instead of only one step, we may have a small suffix set B', s.t. any suffix $b \in B'$ is a recursive function of only these suffixes from B', which has the size much smaller than B. Recall that the sensitivity is stored in entry (L, ϵ). Thus, we require ϵ in B'. The SRT of a spaced seed is a tree with root node being labeled as ϵ, and each node of the tree is labeled with $b \in B$. The label of the left child of any node b' is $B(0b')$, where $B(0b')$ is the longest prefix of $0b'$ that belongs to B; and the label of the right child of b' is $1b'$. If $1b'$ is comparable with a and have the same length as seed a, then $1b'$ is "hit" by the seed a, and the corresponding node is labeled as "hit". The SRT is built by a depth first search. A node is a leaf only if 1) b is labeled as a "hit", or 2) the label b has been occurred before. Fig.1 shows an example of the SRT of the spaced seed 1101011.

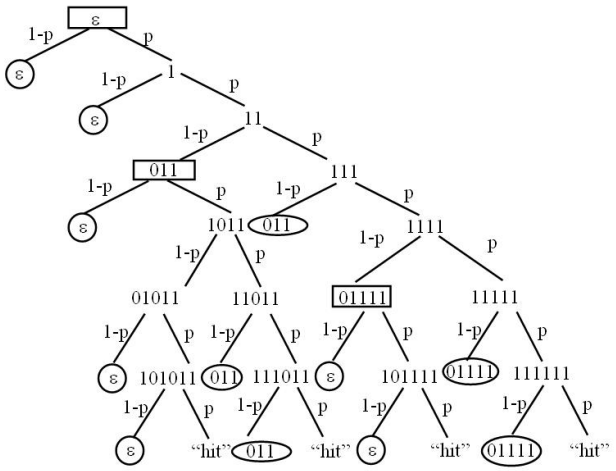

Fig. 1. An Illustration of Suffix-recursion-tree of seed 1101011.

As shown in Fig.1, there are only three suffixes that recur more than once, $\epsilon, 011, 01111$. Thus, $B' = \{\epsilon, 011, 01111\}$. Note that the depth of the suffix-recursion-tree is M. For any node in the tree, it depends on its left child with probability $(1-p)$, while p on its right child, if the children exist. Each path from a node v to its ancestor u represents the probability that u depends on v. Thus, $f(i, \epsilon)$ can be expressed only by those circled entries. Similarly, we can get a recursive relation for each internal node v on all the leaves of the subtree rooted at v. The dependency of b on all $b' \in B'$ is thus presented in the sub-tree rooted at b, and the depth of the sub-tree rooted at b is at most $M-1$. Thus,

by building this SRT, a set of new recursion functions can be found that only depends on a small suffix set B'.

By applying dynamic programming on this new recursion function, we have a new algorithm for computing the sensitivity of a single spaced seed on a random region. First, we construct the SRT for a given seed. Second, we deduce the recurrence relations. Third, we do a dynamic programming based on this set of new recurrence relations. Lastly we output the sensitivity.

Algorithm SRT-DP

Input Seed a, similarity level p, and region length L.
Output Sensitivilty of seed a on a random region of length L at similarity level p.
1. Construct the SRT of the seed a, and build set B'
2. Deduce all the recursive equations for every $b' \in B'$.
3. For $i \leftarrow 0 \dots L$
4. For $b' \in B'$ with decreasing lengths
5. Compute $f(i, b')$ by the recursive equations from Step 2.
6. Output $f(L, \epsilon)$

Fig. 2. Algorithm SRT-DP

Theorem 1. *Let a be a spaced seed and R be a random region. Algorithm SRT-DP computes $Pr(a$ hits $R)$ in $O(|B'|^2ML + |B|)$ time, where M is the length of seed a, and B' is the suffix subset determined by the suffix-recursion-tree of seed a.*

Proof. The correctness of the algorithm comes from the discussion before the theorem. Line 1 can be done by depth first search. Since each node is determined by its two direct children from Eq. 2, and once the node is traversed, the depth first search will stop if the node recurs somewhere else, the running time is thus $O(|B|)$. Line 2 can be done in $O(|B|)$, which is the size of the tree. After pre-computing, line 3 to line 5 takes $O(|B'|^2ML)$ running time because for each entry in the dynamic programming table, it depends on at most $M|B'|$ entries. Therefore, the total running time for the algorithm SRT-DP is $O(|B'|^2ML+|B|)$ □

3.2 A Concrete Example

In this section, we will give a concrete example to show our algorithm has a much better running time than the PH algorithm on some cases. First, we prove the following results.

Lemma 1. *Any suffix $b \in B'$ of any spaced seed can be either ϵ or a binary string starting with 0. That is, any $b \in B'$ can not start with 1.*

Proof. By contradiction. Suppose there is a $b \in B'$ that starts with 1, i.e. $b = 1b'$. That means 1) b has occurred more than once, and 2) b' is a suffix in B, and b' is the parent of b. Considering any two places where b occurs, at each place, b' is the parent of b. Thus, b' has also occurred at least twice in the suffix-recursion tree. For the definition of the suffix-recursion tree, the tree should stop at one b', which contradicts with b is a child of this b'.

Therefore, any suffix $b \in B'$ can be either ϵ or a binary string starting with 0. □

Recall that the running time of PH algorithm is $O((M + L)|B|)$, in which $|B|$ can be as large as $O(M2^{M-W})$. From lemma 1, we know B' is a subset of B, the size of which is much smaller than B, because all suffixes starting with 1 in B will not be in B'. Furthermore, we construct a simple example illustrate that the algorithm SRT-DP is much better than PH algorithm. Better examples are available, but it is out of the scope.

$$10 \quad \underbrace{11\cdots11}_{m^3+(m-1)\ 1's} \quad 0\cdots\underbrace{11\cdots11}_{m^3+1\ 1's}\,0\,\underbrace{11\cdots11}_{m^3\ 1's} \tag{3}$$

For a seed as shown in Eq. 3, there are m 0's in a. The size of B for this case is:

$$\begin{aligned}|B| &= m^3 + 2 + (m^3+1)\times 2 + 4 + \cdots + (m^3+m-1)\times 2^{m-1} + 2^m + 2^m \\ &= \sum_{i=1}^{m} 2^i + \sum_{i=0}^{m-1} m^3 2^i + \sum_{i=0}^{m-1} i2^i + 2^m \\ &= (2^{m+1}-2) + m^3(2^m-1) + (m2^m - 2^{m+1} + 2) + 2^m \\ &= (m^3+m+1)2^m - m^3\end{aligned}$$

Lemma 2. *Any suffix $b \in B'$ of our seed a can be either ϵ or a binary string starting with 0 and followed by 1's and at most one 0.*

Proof. From lemma 1, we know that the only possible suffix $b \in B'$ of seed a can be either ε or a binary string starting with 0. By contradiction, suppose there is a suffix $b \in B'$ which starts with 0 and followed by at least two 0's.

From the definition of B', b should be compatible with the seed a. Thus, the 0's in b which follows the first 0 have to be matched to some 0's in a.

$$\begin{array}{lcccccc} a: & 10\cdots11\cdots11 & 011\cdots11 & 011\cdots11011\cdots11 & 011\cdots11 \\ & & & 1 & 2 \\ b: & & 011\cdots11 & 011\cdots11111\cdots11 & 011\cdots11 \end{array}$$

Recall that any suffix $b \in B'$ is generated by taking the longest compatible prefix of some $0b'$. Thus, b in the above figure is the result of cutting the tail of some binary string. Thus, there are two different pairs of 0's in a, which have the same distance between each other. Suppose the first pair of 0's (position 1 and position 2 in seed a) contains region $\underbrace{11\cdots11}_{l_1+(n_1-1)\ 1's}\ 0\ \underbrace{11\cdots11}_{l_1+(n_1-2)\ 1's}\ 0\cdots\underbrace{11\cdots11}_{l_1+1\ 1's}\,0\,\underbrace{11\cdots11}_{l_1\ 1's}$, and the second pair of 0's (position 1 and position 2 in suffix b) contains region

which corresponds to the region $\underbrace{11\cdots11}_{l_2+(n_2-1)\ 1's}\ 0\ \underbrace{11\cdots11}_{l_2+(n_2-2)\ 1's}\ 0\cdots\underbrace{11\cdots11}_{l_2+1\ 1's}0\underbrace{11\cdots11}_{l_2\ 1's}$ in seed a. Note here $n_1 \leq m$ and $n_2 \leq m$. Since the distances between these two pairs are the same. We have

$$l_1+(l_1+1)+\cdots+(l_1+n_1-1)+(n_1-1) = l_2+(l_2+1)+\cdots+(l_2+n_2-1)+(n_2-1)$$

From this equation, we have

$$\begin{aligned}(2l_1+n_1+1)n_1 &= (2l_2+n_2+1)n_2\\ 2(l_1n_1-l_2n_2) &= (n_2+1)n_2-(n_1+1)n_1\end{aligned}$$

From the construction of the seed a, for any i, we have $l_i = m^3 + j$, where $0 \leq j \leq m-1$. If n_1 and n_2 are different, without loss of generality, assume $n_1 > n_2$. Let $n_1 = n_2 + h$, where $h \geq 1$. The left part of the above equation is then:

$$\begin{aligned}&2(l_1n_1 - l_2n_2)\\ &= 2[(m^3+j_1)(n_2+h)-(m^3+j_2)n_2]\\ &= 2[m^3h + j_1(n_2+h) - j_2n_2]\\ &\geq 2(m^3h - m^2)\\ &\geq m^3 \qquad (when \quad m \geq 2)\end{aligned}$$

Thus, the absolute value of the left part of the above equation is at least m^3, while the right part is at most m^2. Thus, $n_1 = n_2$, and $l_1 = l_2$. This contradicts to the assumption that these two regions are different.

Therefore, any suffix $b \in B'$ of seed a can be either ϵ or a binary region starting with 0 and followed by 1's and at most one 0.

□

Combining lemma 1 and lemma 2, the number of suffixes $b \in B'$ for seed a is at most:

$$\begin{aligned}&m + \tbinom{m}{2}\\ &= O(m^2)\end{aligned}$$

In this seed a, the total length M is $m^4 + \frac{m^2}{2} + \frac{m}{2} + 1$

Therefore, the total running time for the algorithm SRT-DP is $O(|B'|^2ML + |B|) = O(m^8L + m^32^m)$, while the running time for PH algorithm is $O((M + L)|B|) = O(m^72^m + Lm^32^m)$. The dominant term here is L or 2^m. Thus, the SRT-DP algorithm is much faster than PH algorithm because it is the sum of the two dominant terms instead of the product of them.

4 Block-matrix Algorithm

Now, we develop another algorithm to solve the problem of calculating the sensitivity of a given seed, which mainly handle the case when the length of the homology region is long.

Recall Eq. 2, in which $0b'$ is the longest prefix of $0b$ in B. Any entry in the dynamic programming table depends on only two previously computed entries.

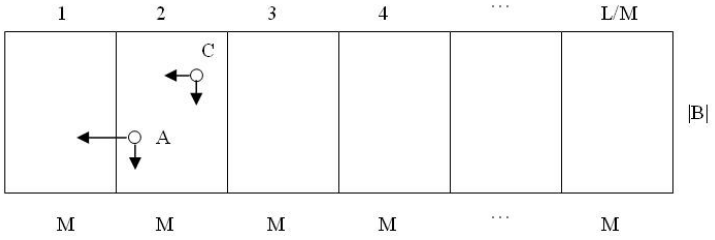

Fig. 3. An Illustration

For any $b \in B$, $|b| - |b'|$ is bounded by M. Thus, if we divide random region R into blocks, each of which has length M, all entries in one block will only depend on entries from itself or entries from another block. Fig. 3 shows an illustration of dividing region R into blocks.

For any entry in block 1, it depends on two previously computed entries in block 1. Thus, all entries in block 1 can be pre-computed by only using entries from itself. For an entry in block 2, it depends on only two entries, one of which must be in the same column as this entry in the same block (corresponding to $f(i, 1b)$). For the other dependent entry, there are two cases: 1) the entry is also in block 2 (as point C in Fig.3); and 2) the entry is in block 1 (as point A in Fig.3). In either case, the entry has been computed already. Let $F(i, j)$ denote the dependency relationship of block i on block j and i. From previous discussion, we can easily compute $F(1, 1)$ and $F(2, 1)$. Thus, for any entry in block 3, we can consider block 3 as block 2, and block 2 as block 1, then use $F(2, 1)$ on the entry. Clearly, $F(3, 2)$ is the same as $F(2, 1)$. If we further substitute any entry in block 2 used in $F(3, 2)$ by its $F(2, 1)$ relationship, we can have a dependency relationship between any entry in block 3 and entries in block 1 and block 3, i.e. $F(3, 1)$. For any entry in block 5, we can apply $F(3, 1)$ twice, which will result in $F(5, 1)$. Generally speaking, assume $L = (1 + 2^r)M$, we can apply this idea to reduce the region length dimension of the dynamic programming, L, to $\log L$ by using the algorithm shown in Fig.4.

We now analyze the running time of this algorithm. Line 1 is just for illustration purpose, in practice, we don't need to divide R into blocks, instead we just need to compute $[i/M] + 1$ for a given index i; line 2, 3, and 4 can be done in $O(M|B|)$; line 5 can be done in $O(M|B|)$ because each entry in block 2 depends on only 2 other entries from block 1 or block 2.

Theorem 2. *The Block-matrix Algorithm has a running time* $O((M|B|)^{2.236} \log(L/M))$.

Proof. If an entry in block $(1+2^i)$ depends only on previously computed entries in the same block, the algorithm takes $O(M|B|)$ running time. Thus, we can assume an entry in block $(1+2^i)$ depends on entries in block $(1+2^{i-1})$ by using previously computed $F(1 + 2^{i-1}, 1)$. Let $a_i, i = 1 \ldots M|B|$ denote all entries in block 1 with the condition that an entry with smaller line index is indexed before any entry with larger line index, and an entry with smaller column index

Block-matrix Algorithm

Input Seed a, similarity level p, and region length L.
Output Sensitivilty of seed a on a random region R of length L at similarity level p.

1. Divide length L into blocks, each of which has length M. Index these blocks as block 1, block 2, ..., block $1 + 2^r$.
2. For $i \leftarrow 0 \dots M-1$
3. For $b \in B$ with decreasing lengths
4. Compute $f(i, b)$ by the recursive function Eq. 2.
5. Compute $F(2, 1)$.
6. For $i \leftarrow 1 \dots r$
7. Compute $F(1 + 2^i, 1)$ by using $F(1 + 2^{i-1}, 1)$.
8. Output $f(L, \epsilon)$

Fig. 4. Block-matrix Algorithm

is indexed before any entry with larger column index and the same line index. Let $b_i, i = 1 \dots M|B|$ and $c_i, i = 1 \dots M|B|$ denote all entries in block $(1 + 2^{i-1})$ and $(1 + 2^i)$, respectively, with the same indexing rule as a_i. Thus, any b_j is a linear combination of all a_i, and any c_k is a linear combination of all b_j. That means:

$$b_j = \sum_{i=1}^{M|B|} w_{ji} a_i$$

$$c_k = \sum_{j=1}^{M|B|} w'_{kjb_j}$$

Note that $F(1 + 2^i, 1)$ is the same as $F(1 + 2^{i-1}, 1)$, because $(1 + 2^i) - (1 + 2^{i-1}) = (1 + 2^{i-1}) - 1 = 2^{i-1}$. Thus, $w_{ij} = w'_{ij}$.

Thus,

$$c_k = \sum_{j=1}^{M|B|} w_{kj} b_j = \sum_{j=1}^{M|B|} w_{kj} \sum_{i=1}^{M|B|} w_{ji} a_i = \sum_{j=1}^{M|B|} \sum_{i=1}^{M|B|} w_{kj} w_{ji} a_i.$$

This means that the dependency relationship between any c_k and all a_i can be calculated by matrix multiplication. Thus, the running time for line 6 and 7 is bounded by $O((M|B|)^{2.236})$.

Therefore, the total running time of Block-matrix algorithm is $O((M|B|)^{2.236} \log(L/M))$. □

It's not difficult to extend our algorithm when L is not in $(1 + 2^r)M$ format. Recall the running time of PH algorithm is $O((M + L)|B|)$, our Block-matrix Algorithm can provide a much better running time if L is large.

5 Monte Carlo Algorithm for Finding Optimal Seed

In this section, we propose a Monte Carlo algorithm to solve the optimal seed problem. Given seed length M, weight W, a random region R of length L with

distribution $Pr(R[i] = 1) = p$, $0 \leq i \leq L-1$, we want to find a seed a of good enough sensitivity with high probability. The deterministic algorithm for finding the optimal seed has a running time of $O(\binom{M}{W}(M+L)|B|)$. Here, we provide a $O((k\binom{M}{W}M + t|B|)L)$ Monte Carlo algorithm that can find a near-optimal seed with high probability, where k and t are parameters to adjust the errors of the sensitivity and to control the probability.

The intuition behind our algorithm is that if we randomly generate k binary regions of length L with similarity level p, the number of hits for a seed a on these regions will be relevant to the sensitivity of a. The higher the sensitivity is, the more expected hits are. The outline of the algorithm is displayed in Fig.5. First, we randomly generate k regions. Then we count the occurrence of each pattern. If a seed occurs multiple times in a random region, it counts only once. After that, we select top t patterns with the largest numbers of occurrences. Lastly, we employ an deterministic algorithm to compute the sensitivity of the t seeds, and output the seed with the highest sensitivity.

Monte Carlo Algorithm for Finding Optimal Seed	
Input	Integer M, W, similarity level p, and random region R of length L.
Output	A seed a of length M, weight W with the highest sensitivity on R.
1.	Randomly generate k binary regions with similarity level p.
2.	For each possible binary pattern a' of length M and weight W.
	2.1 Let $cnt[a']$ be the number of random regions that contain a'.
3.	Let C be the set of t patterns with largest cnt values, ties break randomly.
4.	Compute the sensitivity of the patterns in C by a deterministic algorithm.
5.	Output the pattern with the highest sensitivity in C.

Fig. 5. Monte Carlo Algorithm for Finding Optimal Seed

Theorem 3. *The running time for the algorithm in Fig. 5 takes time* $O((k\binom{M}{W}M + t|B|)L)$

Proof. Step 1 takes $O(kL)$. The running time of step 2 is $(kML\binom{M}{W})$. The running time for step 3 is dominated by step 2. Step 4 takes $O(t(M+L)|B|)$ if HP algorithm is used to compute the sensitivity of a single seed. Thus, the overall running time is: $O((k\binom{M}{W}M + t|B|)L)$. □

We further estimate the probability that the Monte Carlo algorithm can find out the optimal seed. Let s_o be the sensitivity of an optimal seed on the random region R, s be the sensitivity of the seed on R found by our Monte Carlo algorithm. We have the following results:

Theorem 4. *The Monte Carlo algorithm ensures* $s_o - s \leq \epsilon$ *with high probability* $1 - e^{-\frac{\epsilon^2}{3(1-\epsilon)}kt}$. *If* $kt \geq \frac{3(1-\epsilon)}{\epsilon^2}\log\frac{1}{\sigma}$, *the Monte Carlo algorithm can guarantee to find out an optimal seed with probability* $1-\sigma$.

Proof. Define three random variables: C, C_o, and X. C is the sum of C_i which is defined to be the number of hits of a seed found by our algorithm on random region i, (values can be 0 or 1 with probability $1-s$ and s, respectively)); C_o is the sum of C_{oi} which is defined to be the number of hits of an optimal seed on random region i, (values can be 0 or 1 with probability $1-s_o$ and s_o, respectively); X is the sum of X_i which is defined to be $C_i - C_{oi} + 1$, (values can be 0, 1, 2 with probability $(1-s)s_o$, $ss_o + (1-s)(1-s_o)$, and $s(1-s_o)$, respectively). Since the number of random regions is k, $X > k$ means the real number of hits of the optimal seed is smaller than the number of hits of an arbitrary seed. Let T denote the set of the indices of the top t seeds chosen by our Monte Carlo algorithm.

Since both s_o and s lay in $[0,1]$, we can assume that when k and t are large enough, our algorithm can find out seeds with $s_o - s < 0.5$. Let C^i be random variable C for seed i. Thus,

$$\begin{aligned}&\Pr(\textit{an optimal seed is in top } t \textit{ seeds}) = 1 - \Pr(\textit{no optimal seed is in top } t \textit{ seeds})\\ &= 1 - \Pr(C_o \textit{ is smaller than } C^i,\ i \in T)\\ &= 1 - \prod_{i \in T} \Pr(C_o \textit{ is smaller than } C^i)\ \ (\textit{because of independency})\end{aligned}$$

We now compute $\Pr(C_o \textit{ is smaller than } C^i)$ by Chernoff bounds. For $0 < \delta \leq 1$, $\Pr(X \geq (1+\delta)\mu) \leq e^{-\mu\delta^2/3}$, where X is the sum of independent Poisson trials, and $\mu = E[X]$. It is obvious that $X_i = C_i - C_{oi} + 1$ is independent Poisson trials. Thus,

$$\begin{aligned}&\mu = E[X] = k\{0 \times (1-s)s_0 + 1 \times [ss_0 + (1-s)(1-s_o)] + 2 \times (1-s_0)s\}\\ &= (1 + s - s_o)k\end{aligned}$$

Let $(1+\delta)\mu = k$, we get $\delta = \frac{k}{\mu} - 1 = \frac{s_o - s}{1+s-s_o}$.
By Chernoff bounds, we have

$$\begin{aligned}&\Pr(C_o \textit{ is smaller than } C^i) = \Pr(X \geq k)\\ &\leq e^{-(1+s-s_o)k\frac{(s_o-s)^2}{3(1+s-s_o)^2}} = e^{-\frac{(s_o-s)^2}{3(1+s-s_o)}k}\end{aligned}$$

Let $\epsilon = s_o - s$, we have $\Pr(C_o \textit{ is smaller than } C^i) \leq e^{-\frac{\varepsilon^2}{3(1-\varepsilon)}k}$.
Thus, the probability that our Monte Carlo algorithm guarantees to find out an optimal seed is:

$$\begin{aligned}&\Pr(\textit{an optimal seed is in top } t \textit{ seeds}) = 1 - \prod_{i \in T} \Pr(C_o \textit{ is smaller than } C^i)\\ &\geq 1 - (e^{-\frac{\epsilon^2}{3(1-\epsilon)}k})^t = 1 - e^{-\frac{\varepsilon^2}{3(1-\varepsilon)}kt}\end{aligned}$$

Thus, when k and t increases, the probability increases quickly. If we require the probability that our Monte Carlo algorithm fails to find out an optimal seed is smaller than σ, $0 < \sigma < 1$, we can have the requirement on k and t: $kt \geq \frac{3(1-\varepsilon)}{\varepsilon^2} \log \frac{1}{\sigma}$.

□

Acknowledgements

We are grateful to Dongbo Bu for his thought provoking discussion and comments. This work was supported by the Application Foundation Project of Technology Development of Jilin Province, Grant 20040531.

References

1. S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *J.Mol.Biol.*, 215:403–410, 1990.
2. S. Altschul, T. Madden, A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.
3. B. Brejova, D. Brown, and T. Vinar. Optimal spaced seeds for hidden markov models, with application to homologous coding regions. In *CPM2003: The 14th Annual Symposium on Combinatorial Pattern Matching*, pages 42–54, Washington, DC, USA, 2003. IEEE Computer Society.
4. D. Brown. Optimizing multiple seeds for protein homology search. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 2(1):29–38, 2005.
5. S. Burkhardt, A. Crauser, H. Lenhof, E. Rivals, P. Ferragina, and M. Vingron. q-gram based databse searching using a suffix array. In *Third Annual International Conference on Computational Molecular Biology*, pages 11–14, 1999.
6. K. Choi, F. Zeng, and L. Zhang. Good spaced seeds for homology search. *Bioinformatics*, 20(7):1053–1059, 2004.
7. K. Choi and L. Zhang. Sensitivity analysis and efficient method for identifying optimal spaced seeds. *Journal of Computer and System Sciences*, 68:22–40, 2004.
8. A. Delcher, S. Kasif, R. Fleischmann, J. Peterson, O. White, and S. Salzberg. Alignment of whole genomes. *Nucleic Acids Res.*, 27:2369–2376, 1999.
9. M. Li, B. Ma, D. Kisman, and J. Tromp. Patternhunter ii: highly sensitive and fast homology search. *JBCB*, 2(3):417–439, 2004.
10. M. Li, B. Ma, and L. Zhang. Superiority and complexity of the spaced seeds. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithms (SODA 2006)*, pages 444–453, 2006.
11. D. Lipman and W. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227:1435–1441, 1985.
12. B. Ma, J. Tromp, and M. Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.
13. F. Preparata, L. Zhang, and K. Choi. Quick, practical selection of effective seeds for homology search. *JCB*, 12(9):1137–1152, 2005.
14. T. Tatusova and T. Madden. Blast 2 sequences - a new tool for comparing protein and nucleotide sequences. *FEMS Microbiol. Lett.*, 174:247–250, 1999.
15. I. Yang, S. Wang, Y. Chen, and P. Huang. Efficient methods for generating optimal single and multiple spaced seeds. In *BIBE 2004: Proceedings of the 4th IEEE Symposium on Bioinformatics and Bioengineering*, page 411, Washington, DC, USA, 2004. IEEE Computer Society.
16. Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning dna sequences. *J.Comput.Biol.*, 7:203–214, 2000.