情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2011-AL-137 No.2
2011/11/18

Spaced Seedの検索のための索引

小野寺　拓[†1]　渋谷哲朗[†1]

本稿では特定の位置に”don't care”を含むパターンの検索に対応した接尾辞配列の一般化を導入し、また、その構築アルゴリズムを３つ提案する。このようなパターンは計算生物学における spaced seed の検索において現れる。

An Index Structure for Spaced Seed Search

Taku Onodera[†1] and Tetsuo Shibuya[†1]

In this paper, we generalize suffix array to support the search of patterns with “don't care”s in predetermined positions and introduce three algorithms to construct it. Such patterns occur in the spaced seed search of computational biology.

1. Introduction

String searching algorithms can be classified into online search and index search. Though construction of indices is computationally expensive, it makes successive searches much faster. Hence, it is a good idea to prepare an index for the text when the text is large and static and many searches are likely to be performed later.

In this paper, we consider an index structure which supports search of patterns with “don't care”s. A “don't care” is a special character that can match any single character and it is also called a ”wild card”. Such queries occur, for example, in spaced seed search. In the research of computational biology, search of homologous regions between two sequences is a common task and to do that the following three steps are widely used: a)extract a short segment called a seed from one of the sequences; b)find the occurrences of the seed in the other; c)examine the surrounding region of each hit by alignment. A spaced seed is a seed including “don't care”s. Ma et. al.[10] found that it is possible to optimize the arrangement of “don't care”s in a spaced seed to make homology search faster and more sensitive. Since then, many researches about spaced seeds, both theoretical and practical, have been made[2),9),14)].

The gapped suffix array[3] is a generalization of the suffix array. A (g_0, g_1)-gapped suffix array of a text is the indices of the heads of suffixes, sorted ignoring i-th characters for $i \in [g_0, g_0 + g_1]$. If the (g_0, g_1)-gapped suffix array is given, patterns whose i-th characters are “don't care”s for $i \in [g_0, g_0+g_1]$ can be searched in $O((m-g_1)\log n)$-time where n and m are the length of the text and the pattern respectively. The gapped suffix array can be constructed in $O(n)$-time.

Because the gapped suffix array is not applicable to the search of spaced seeds as it is, we generalize it as follows. First, fix a binary string $b = b_1 b_2 \dots b_w \in \{0,1\}^w$. Then, we consider an array containing the indices of the heads of suffixes sorted ignoring i-th characters except $i \in \{1 \le j \le w | b_j = 1\}$. For example, if $b = 101$ suffixes are sorted according to their first and third characters while the second and those following the third are ignored. If this array is given, patterns whose i-th characters are “don't care”s for $i \in \{1 \le j \le w | b_j = 0\}$ can be searched in $O(k \log n)$-time where $k := \#\{1 \le i \le w | b_i = 1\}$.

The problems considered in the rest of the paper and our results for them are summarized as follows.

Problem 1. Given a text $T = T[1, 2, \dots, n]$ and a binary string $b = b_1 b_2 \dots b_w$, sort the suffixes of T ignoring i-th characters for $i \in \{1 \le j \le w | b_j = 0\}$ and construct an array $b\text{-}SA[1, 2, \dots, n]$ s.t. $b\text{-}SA[i]$ is the index of the head of the i-th suffix.

Problem 2. Given a text $T = T[1, 2, \dots, n]$, $b = b_1 b_2 \dots b_w$, $b\text{-}SA$ and the rank corresponding to $b\text{-}SA$ where $b\text{-}SA$ is the same as that in Problem 1, sort the suffixes of T ignoring i-th characters for $i \in \{1 \le j \le n | b_{j\%w} = 0\}$ where $j\%w$ is the remainder of j divided by w, and construct an array $b^*\text{-}SA[1, 2, \dots, n]$ s.t. $b^*\text{-}SA[i]$ is the index of

†1 東京大学医科学研究所ヒトゲノム解析センター
Human Genome Center, Institute of Medical Science, the University of Tokyo

the head of the i-th suffix.

Result for Problem 1. Given T and b, b-SA (and the corresponding rank) can be obtained either in $O(gn)$-time and $O(n)$-space where g is the number of runs of 1 in b or in $O(\frac{wn}{\epsilon \log w})$-time and $O(w^{\epsilon}n)$-space where ϵ can be any constant s.t. $0 < \epsilon < 1$.

Result for Problem 2. Given T, b, b-SA and corresponding rank, b^*-SA can be obtained in $O(n)$-time and $O(n)$-space.

Though it requires much memory, the second algorithm for Problem 1 can be faster than the first when g is large (For example, when there is $\exists c > 0$ s.t. $g > cn$). The algorithm for Problem 2 requires not only b-SA but also the rank. When the rank is not given, one can compute it by checking i-th characters of suffixes for $i \in \{1 \leq j \leq w | b_j = 1\}$, but this naive method takes $O(kn)$-time. The result for Problem 1 means one can provide both b-SA and the rank in $O(gn)$-time to the solver of Problem 2.

There are other types of indices which aid the search of patterns with "don't care"s[13),7),15)]. These are based on ordinary indices like the suffix tree. These indices support more flexible search than ours because they do not assume positions in patterns where "don't care"s occur. On the other hand, these indices work as a filter in general and, after referring to it, one needs to determine whether each candidate is actually a match. Thus, in terms of search time, our index is better for inputs to which these indices generate many false positive candidates.

In section 2, we set up basic definitions and notations. The usage of the index is also explained here. In sections 3 and 4, we describe the results for the problems 1 and 2 respectively. We conclude in section 5.

2. Preliminaries

2.1 Definitions and Notations

Let Σ be a finite totally ordered set. A string T is either an element of $\cup_{n=1}^{\infty}\Sigma^n$ or an empty string. That is, the set of strings is equal to $\Sigma^* := \cup_{n=0}^{\infty}\Sigma^n$ where Σ^0 is the set consists only of the empty string. The length of T is $n \geq 0$ s.t. $T \in \Sigma^n$, and is denoted by $|T|$. We denote the i-th character of a string T by $T[i]$. For string T of length n and $i, j \in \mathcal{N}$ s.t. $1 \leq i \leq j$, let $T[i,j]$ be $T[i]T[i+1]\dots T[j]$ if $j \leq n$, $T[i]T[i+1]\dots T[n]$ if $i \leq n < j$ and the empty string if $n < i$. Let T_i be $T[i,n]$ for $i \leq n$. The lexicographic order $\leq$ between $T1, T2 \in \Sigma^*$ is defined as follows: a) $\phi \leq T$ for $\forall T \in \Sigma^*$; b) $T1 \leq T2$ if $|T1| > 0, |T2| > 0$ and $T1[1] < T2[1]$; c) $T1 \leq T2$ if $|T1| > 0, |T2| > 0, T1[1] = T2[1]$ and $T1_2 \leq T2_2$. Throughout the paper we assume integer alphabets, i.e. the size of Σ is $O(|T|)$.

A "don't care" is a special character not included in Σ and we denote it by ?. A pattern P is an element of $\cup_{m=1}^{\infty}(\Sigma \cup \{?\})^m$. The length of P is $m \geq 1$ s.t. $P \in (\Sigma \cup \{?\})^m$ and denoted by $|P|$. The i-th character of a pattern P is denoted by $P[i]$. If a pattern P and a string T satisfy $P[j] = T[i+j-1]$ for $\forall j \in \{i | 1 \leq i \leq m, P[i] \neq ?\}$, we say P matches T at position i.

For $w \in \mathcal{N}$, a "don't care" position of length w, $b = b_1 b_2 \dots b_w$, is an element of $\{0,1\}^w$. Let $\mathbf{1}_b, \mathbf{0}_b$ and $|b|$ denote $\{1 \leq i \leq w | b_i = 1\}, \{1 \leq i \leq w | b_i = 0\}$ and $\#\mathbf{1}_b$ respectively. Let $b[i,j]$ denote $b_i b_{i+1} \dots b_j$ for $1 \leq i \leq j \leq w$. Let $b(T)$ denote $T[i_1]T[i_2]\dots T[i_k]$ (if $i_k \leq n$), $T[i_1]T[i_2]\dots T[i_j]$ (if $i_j \leq n < i_{j+1}, 1 \leq j < k$), ϕ (if $n < i_1$) where $\{i_j\}_{j=1}^{k}$ is the subsequence of $\{i\}_{i=1}^{w}$ s.t. $\{i_j\}_j$ equals to $\mathbf{1}_b$ as sets. The b-lexicographic order $\leq_b$ between $T1, T2 \in \Sigma^*$ is defined as $T1 \leq_b T2$ iff $b(T1) \leq b(T2)$. $\leq_b$ is a preorder. We write $T1 =_b T2$ if $T1 \leq_b T2$ and $T2 \leq_b T1$.

The leftmost rank of an element e in a finite totally ordered set S is $\#\{e' \in S | e' \leq e, e \not\leq e'\} + 1$.

The suffix array SA of a text T is the unique permutation of $\{1, \dots, n\}$ s.t. $T_{SA[i]}$ is lexicographically increasing. SA_h is a permutation s.t. $T_{SA_h[i]}[1,h]$ is lexicographically non-decreasing. The h-rank of T_i is the leftmost rank of $T_i[1,h]$ in $\{T_i[1,h]\}_{i=1}^{n}$ ordered lexicographically. Let $RSA_h[1,\dots,n]$ be an array s.t. $RSA_h[i]$ is the h-rank of T_i. The height array Hgt of a string T is an array s.t. $Hgt[i]$ is the length of the longest common prefix of $T_{SA[i-1]}$ and $T_{SA[i]}$ for $1 < i \leq n$.

The b-gapped suffix array b-SA of a string T of length n is the unique permutation of $\{1, \dots, n\}$ s.t. a) $T_{b\text{-}SA[i]} \leq_b T_{b\text{-}SA[i+1]} (1 \leq i < n)$; b) b-$SA[i] < b$-$SA[j]$ if $T_{b\text{-}SA[i]} =_b T_{b\text{-}SA[j]}$ and $i > j$.

Let b-$SA_h := b[1,h]$-SA. The b-rank of T_i is the leftmost rank of T_i in b-SA. Let (b,h)-rank be $b[1,h]$-rank. Let b-$RSA_h[1,\dots,n]$ be an array s.t. b-$RSA_h[i]$ is the (b,h)-

rank of T_i.

2.2 Search Method

The b-gapped suffix array can be applied to pattern matching in almost the same way as the suffix array. If b-gapped suffix array b-SA of a string T and $b \in \{0,1\}^w$ is given, a pattern P of length at most w s.t. $P[i] =?$ for $i \in \mathbf{0}_b$ and $P[i] \in \Sigma$ for $i \in \mathbf{1}_b$ can be located by binary search in $O(|b| \log n)$-time. All the occurrences of the pattern can be enumerated in $O(|b| \log n + Occ(P))$-time where $Occ(P)$ is the number of occurrences of P.

3. Constructing the b-Suffix Array for Given T and b

In this section, we consider constructing b-SA for given T and b. An obvious way to do this is to use ordinary radix sort, that is, to repeat sorting the suffixes by i-th character while i runs through all $i \in \mathbf{1}_b$ downwards from the largest to the smallest. It takes $O(|b|n)$-time and $O(n)$-space. We propose two more elaborate algorithms to construct b-suffix arrays. We first explain the underlying ideas shared in common. Then we describe each algorithm.

3.1 Underlying Ideas

In SA_h suffixes of the same h-rank neighbor each other. These groups are called h-groups. When sorted by first h' characters where $h < h'$, an h-group can split into several h'-groups but each suffix never move beyond the boundaries of the h-group it belongs to. Thus the members of an h-group are still located together in $SA_{h'}$. In particular, this is true for h-groups of $\forall h \leq n$ in $SA = SA_n$. In some suffix array construction algorithms, $SA = SA_n$ is obtained by repeatedly deriving SA_{2h} from SA_h[11),8)].

Similarly, in b-SA_h, suffixes of the same (b,h)-rank align consecutively, which we call a (b,h)-group. We consider constructing b-SA in a similar way as that for SA above. But unfortunately, the method to double h at a time no longer works for b-SA_h. We try other methods to increase h. In both of the following subsections we use a modification of radix sort to derive b-SA_h of larger h from that of smaller h. This method is based on the sort algorithms presented in[1)] and[11)].

3.2 $O(gn)$-Time, $O(n)$-Space Algorithm

For a "don't care" position b, we call the subregion from b_{i_1} to b_{i_2} a run of 1 iff a) $b_i = 1 (i_1 \leq \forall i \leq i_2)$; b) $i_1 = 1$ or $b_{i_1-1} = 0$; c) $i_2 = w$ or $b_{i_2+1} = 0$. A run of 1 of length r is denoted by 1^r. A run of 0 is defined similarly.

In this subsection, we prove the following theorem.

Theorem 1. *Given a text T of length n and a "don't care" position b of length w, the b-suffix array b-SA can be constructed in $O(gn)$-time and $O(n)$-space where g is the number of runs of 1 in b where ϵ can be any constant s.t. $0 < \epsilon < 1$.*

b can be divided into runs as $0^{r'_1}1^{r_1}\ldots 0^{r'_g}1^{r_g}0^{r'_{g+1}}$. Let i_j be the index of the head of the j-th run of 1. Suppose b-$SA_{i_{j-1}+r_{j-1}-1}$ and b-$RSA_{i_{j-1}+r_{j-1}-1}$ are given and we compute b-$SA_{i_j+r_j-1}$ and b-$RSA_{i_j+r_j-1}$. Because $b_i = 0$ for $i \in [i_{j-1}+r_{j-1}, i_j-1]$, b-$SA_{i_{j-1}+r_{j-1}-1} = b$-$SA_{i_j-1}$ and b-$RSA_{i_{j-1}+r_{j-1}-1} = b$-$RSA_{i_j-1}$. Hence, the (b,i_j-1)-groups are already collected and the next thing to do is to sort each elements of a (b,i_j-1)-group according to characters between the i_j-th and the i_j+r_j-1-th. $T_i[i_j, i_j+r_j-1] = T_{i+i_j-1}[1,r_j]$ and the right hand sides for different is can be compared in constant time if we have RSA_{r_j}, the array of r_j-ranks.

RSA_{r_j} is easily computed when we have SA and Hgt. The boundaries of h-groups are those places where suffixes with different $h+1$-th characters lie next to each other. They can be located by scanning Hgt once marking is s.t. $Hgt[i] < h$. Thus we calculate SA and Hgt in advance.

We prepare queues for each r_j-rank. We enumerate b-SA_{i_j-1} in ascending order and put suffixes into the queues corresponding to $RSA_{r_j}[i+i_j-1]$. Then we enumerate queues from the one corresponding to the lowest r_j-rank to the one corresponding to the highest. From each queue, we pop all the suffixes out aligning them by the following rule: a) Suffixes of lower (b,i_j-1)-rank are aligned before suffixes of higher (b,i_j-1)-rank; b) suffixes of the same (b,i_j-1)-rank are aligned according to the order they are popped out. We maintain b-RSA_{i_j-1} to find the (b,i_j-1)-rank of each suffix in constant time. After that, suffixes in a (b,i_j-1)-group are arranged according to the i-th characters for $i \in [i_j, i_j+r_j-1] \cap \mathbf{1}_b$, which means we got b-$SA_{i_j+r_j-1}$. b-SA is obtained by repeating this procedure for $j = 1$ to g,

Next, we explain the detail. The following data are used[⋆1]: $S1$: b-SA_{i_j-1}, $R1$: b-RSA_{i_j-1}, $C1$: counter for $S1$, B: marker for the heads of $(b, i_j + r_j - 1)$-groups, $S2$: buckets, $R2$: RSA_{r_j}, $C2$: counter for $S2$, SA: suffix array, Hgt: height array.

We show how to update $S1$ and $S2$ from b-SA_{i_j-1} and b-RSA_{i_j-1} to b-$SA_{i_j+r_j-1}$ and b-$RSA_{i_j+r_j-1}$ respectively. First, we assign RSA_{r_j} to $R2$ using SA and Hgt. Next, we put suffixes in $S1$, say $T_{S1[i]}$ into buckets in $S2$ corresponding to the r_j-rank of $T_{S1[i]+r_j-1}$, that is $R2[i1]$ where $i1 := S1[i] + i_j - 1$. Because there can be several suffixes of the same rank, we use $C2$ to record offsets as $S2[R2[i1] + C2[R2[i1]]] \leftarrow S1[i]$. Suffixes of $(b, i_j - 1)$-groups of only 1 member and T_is for $i > n - i_j + 1$ are already in their final position and can be skipped. Then we return the suffixes in $S2$ to the tail of the $(b, i_j - 1)$-groups they belong. When i is the index of the head of an r_j-group in $S2$, the index of the head of the $(b, i_j - 1)$-group suffix $S2[i + j]$ belongs to is $k := R1[S2[i + j]]$. Again, there can be several elements to return to a $(b, i_j - 1)$-group and we use $C1$ to record offsets as $S1[k + C1[k]] \leftarrow S2[i + j]$. We set $B[i]$ to 1 every time we push back a suffix to $S1[i]$. At this time, the suffixes returned to the same $(b, i_j - 1)$-group from the same r_j-group make up a $(b, i_j + r_j - 1)$-group. Each time we finish returning suffixes from one bucket, we traverse the same bucket again setting $B[i]$ to 0 except the heads of $(b, i_j + r_j - 1)$-groups. After returning every suffixes to $S1$, we update $S2$ to b-$RSA_{i_j+r_j-1}$ referring to B. The whole process in this paragraph is repeated while j runs from 1 to g.

SA and Hgt can be made in $O(n)$-time[4),6),12),5)]. In each update from b-SA_{i_j-1} to b-$SAi_j + r_j - 1$, we initialize $R2, C1, C2, B$, move suffixes from $S1$ to $S2$, return suffixes from $S2$ to $S1$ and update $R2$. All of these are done in $O(n)$-time. Therefore the total time complexity is $O(gn)$. Because we only use constant number of data of $O(n)$-size, the total space complexity is $O(n)$. Therefore theorem 1 follows.

3.3 $O(\frac{wn}{\epsilon \log w})$-Time and $O(w^\epsilon n)$-Space Algorithm

Below we prove the following theorem.

Theorem 2. *Given a text T of length n and a "don't care" position b of length w, the b-suffix array b-SA can be constructed in $O(\frac{wn}{\epsilon \log w})$-time and $O(w^\epsilon n)$-space. where ϵ can be any constant s.t. $0 < \epsilon < 1$.*

Fix v s.t. $1 \le v \le w$. For brevity, we assume v divides w evenly. Suppose b-$SA_{(t-1)v}$ and b-$RSA_{(t-1)v}$ are given and we compute b-SA_{tv} and b-RSA_{tv}. $(b, (t-1)v)$-groups are already collected and we need to sort elements in a $(b, (t-1)v)$-group according to i-th characters for $i \in [(t-1)v + 1, tv] \cap \mathbf{1}_b$. By the same method as that of subsection 3.2, this sorting is done in $O(n)$-time if $b[(t-1)v+1, tv]$-RSA is given. Thus, we calculate all b'-RSAs for all "don't care" positions b' of length v in advance. By computing b'-SAs and b'-RSAs from those with smaller $|b'|$s to larger ones, this preprocessing is done in $O(2^v n)$-time.

The total time complexity is $O(2^v n + wn/v)$ and the total space complexity is $O(2^v n)$. In particular, if we choose v to be $\epsilon \log w$ where ϵ is a constant s.t. $0 < \epsilon < 1$, the time complexity is $O(\frac{wn}{\epsilon \log w})$ and the space complexity is $O(w^\epsilon n)$. Therefore theorem 2 follows. This value of v is chosen so that the time needed for preprocessing and those for main process are close to each other. [⋆2]

This algorithm is suitable for the case of multiple "don't care" positions and a single text. Since b'-SAs and b'-RSAs depend only on T, it is possible to share these data among the calculations of different b-STs for the same text and different "don't care" positions. [⋆3] In particular, when you have a text and are going to be given multiple "don't care" positions of the same length w, each b-suffix array can be constructed in $O(wn/\epsilon \log w)$-time for $1 \le \forall\epsilon$. [⋆4]

4. Constructing b-Suffix Arrays for Periodic b

In this section we prove the following theorem.

Theorem 3. *Given a text T of length n, a "don't care" position b of length p, the b-suffix array b-SA and corresponding rank b-$RSA = b$-RSA_p, b^*-$SA[i]$ is obtained in*

⋆1 It is possible to implement without $C1$ and $C2$ but in that case, we need to scan $S1$ and $S2$ to find the appropriate positions to put suffixes increasing the time complexity.

⋆2 The main process takes longer. The v that balances them is expressed by using the Lambert W function.

⋆3 SA and Hgt in the algorithm of subsection 3.2 can also be shared. But in terms of time complexity, the fact makes no difference.

⋆4 $1 \le \epsilon$ because it should be taken so that the preprocessing takes longer than the construction of each b-suffix array.

$O(n)$-time and $O(n)$-space where b^* is a "don't care" position obtained by repeating b enough times to be longer than T.

$T_i \leq_{b^*} T_j$ iff a) $T_i[1,p] \leq_b T_j[1,p]$ and $T_j[1,p] \not\leq_b T_i[1,p]$ ($\Leftrightarrow$ b-$RSA[i] < b$-$RSA[j]$) or b) $T_i[1,p] =_b T_j[1,p]$($\Leftrightarrow$ b-$RSA[i] = b$-$RSA[j]$) and $T_{i+p} \leq_{b^*} T_{j+p}$. Thus, $\leq_{b*}$ on $\{T_i\}_i$ is equivalent to the lexicographic order on $\{b\text{-}RSA[i]b\text{-}RSA[i+p]\ldots b\text{-}RSA[i+\lfloor(n-i)/p\rfloor p]\}_i$ seen as strings. To sort $\{T_i\}_i$ by $\leq_{b^*}$, it suffices to sort $\{rs(i)\}_i$ by lexicographic order where $rs(i) := b\text{-}RSA[i]b\text{-}RSA[i+p]\ldots b\text{-}RSA[i+\lfloor(n-i)/p\rfloor p]$. For i s.t. $1 \leq i \leq p$ and $0 \leq k \leq \lfloor(n-i)/p\rfloor$, $rs(i+kp)$ is a suffix of $rs(i)$. We consider the string $rs := rs(1)0rs(2)0\ldots0rs(p)$. $\{rs(i)\}_i$ can be sorted by sorting the suffixes of rs. It is easy to ignore the suffixes starting from 0 because they gather to the head when sorted. Because the comparison of two suffixes ends before or at the time when either of them reaches the end, comparison of corresponding $rs(i)$s is not affected by inserting 0s.

Almost all of the work is reduced to suffix sorting of rs, which can be done in $O(n)$-time. Therefore, the total time complexity is $O(n)$. Space complexity is also $O(n)$. Therefore theorem 3 follows.

An argumment similar to this is made in[4].

5. Conclusion

We introduced an index structure based on the suffix array, which supports efficient search of patterns with "don't care"s in predetermined positions. We designed three algorithms to construct the index. Two of them are for general inputs and the other is for the case of patterns with periodic "don't care"s.

参 考 文 献

1) A.Andersson and S.Nilsson. A new efficient radix sort. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 714–721, Washington, DC, USA, 1994. IEEE Computer Society.
2) DanielG. Brown. *A Survey of Seeding for Sequence Alignment*, pages 117–142. John Wiley & Sons, Inc., 2007.
3) Maxime Crochemore and German Tischler. The gapped suffix array: A new index structure for fast approximate matching. In Edgar Chávez and Stefano Lonardi, editors, *SPIRE*, volume 6393 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2010.
4) Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In Jos C.M. Baeten, JanKarel Lenstra, Joachim Parrow, and GerhardJ. Woeginger, editors, *ICALP*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003.
5) Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In Amihood Amir and GadM. Landau, editors, *CPM*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2001.
6) Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In RicardoA. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *CPM*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2003.
7) TakWah Lam, Wing-Kin Sung, Siu-Lung Tam, and Siu-Ming Yiu. Space efficient indexes for string matching with don't cares. In Takeshi Tokuyama, editor, *ISAAC*, volume 4835 of *Lecture Notes in Computer Science*, pages 846–857. Springer, 2007.
8) N.Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden, May 1999.
9) Hao Lin, Zefeng Zhang, MichaelQ. Zhang, Bin Ma, and Ming Li. Zoom! zillions of oligos mapped. *Bioinformatics*, 24(21):2431–2437, 2008.
10) Bin Ma, John Tromp, and Ming Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.
11) Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, SODA '90, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.
12) GeNong, Sen Zhang, and WaiHong Chan. Linear suffix array construction by almost pure induced-sorting. In JamesA. Storer and MichaelW. Marcellin, editors, *DCC*, pages 193–202. IEEE Computer Society, 2009.
13) M.Sohel Rahman and CostasS. Iliopoulos. Pattern matching algorithms with don't cares. In Jan van Leeuwen, GiuseppeF. Italiano, Wiebe vander Hoek, Christoph Meinel, Harald Sack, Frantisek Plasil, and Mária Bieliková, editors, *SOFSEM (2)*, pages 116–126. Institute of Computer Science AS CR, Prague, 2007.
14) StephenM. Rumble, Phil Lacroute, AdrianV. Dalca, Marc Fiume, Arend Sidow, and Michael Brudno. Shrimp: Accurate mapping of short color-space reads. *PLoS Comput Biol*, 5(5):e1000386, 05 2009.

Vol.2011-AL-137 No.2
2011/11/18

15) Alan Tam, Edward Wu, Tak-Wah Lam, and Siu-Ming Yiu. Succinct text indexing with wildcards. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval*, SPIRE '09, pages 39–50, Berlin, Heidelberg, 2009. Springer-Verlag.