Succinct Text Indexing with Wildcards

Alan Tam, Edward Wu, Tak-Wah Lam*, and Siu-Ming Yiu

Department of Computer Science, University of Hong Kong, Hong Kong
{sltam,mkewu,twlam,smyiu}@cs.hku.hk

Abstract. A succinct text index uses space proportional to the text itself, say, two times $n \log \sigma$ for a text of n characters over an alphabet of size σ. In the past few years, there were several exciting results leading to succinct indexes that support efficient pattern matching. In this paper we present the first succinct index for a text that contains wildcards. The space complexity of our index is $(3 + o(1))n \log \sigma + O(\ell \log n)$ bits, where ℓ is the number of wildcard groups in the text. Such an index finds applications in indexing genomic sequences that contain single-nucleotide polymorphisms (SNP), which could be modeled as wildcards.

In the course of deriving the above result, we also obtain an alternate succinct index of a set of d patterns for the purpose of dictionary matching. When compared with the succinct index in the literature, the new index doubles the size (precisely, from $n \log \sigma$ to $2n \log \sigma$, where n is the total length of all patterns), yet it reduces the matching time to $O(m \log \sigma + m \log d + occ)$, where m is the length of the query text. It is worth-mentioning that the time complexity no longer depends on the total dictionary size.

1 Introduction

Pattern matching is a fundamental problem. Consider a text T and a pattern P, the earliest work can solve the problem in $O(|T| + |P|)$ time. When the text remains relatively static (say, the text is the human genome), one would like to build an index of T so as to speed up pattern matching. Let n be the number of characters of T. The classical index suffix trees requires $O(n)$ words, or equivalently, $O(n \log n)$ bits, and can support pattern matching in $O(|P| + occ)$ time, where occ is the number of occurrences of P in T. Note that the space complexity has a natural lower bound of $n \log \sigma$ bits (i.e., worst-case text size), where σ is the alphabet size. Starting with the work of Ferragina and Manzini [6] and Grossi and Vitter [9], the past decade has witnessed a chain of works that make it feasible to build a succinct text index with size proportional to $n \log \sigma$ bits or even a compressed index (with size proportional to nH_k bits), while supporting efficient pattern matching, using $O(|P| + occ \log^{1+\epsilon} n)$ time for any $\epsilon > 0$ (see the survey by Navarro and Mäkinen [12] for a complete list of references).

This paper is concerned with pattern matching on text containing wildcards (or don't care characters). Specifically, a wildcard, denoted by ϕ, is a special character that matches any single character. Fischer and Paterson [8] were among

* Part of the work is supported by RGC Grant HKU 714006E.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 39–50, 2009.

the first to study wildcard matching. There are several results on text indexing for wildcard matching. In the simple setting where the text contains no wildcards, Rahman and Iliopoulos [15] and later Lam et al. [11] have each given an $O(n)$-word index for matching patterns with wildcards. Indexing a text containing wildcards is technically more challenging. It naturally arises in indexing genomic sequences, in which some base pairs are known to be single-nucleotide polymorphisms (SNP), that could be modeled as wildcards. The wildcard index by Cole et al. [5] uses $O(n \log^k n)$ words, where k is the number of wildcards. It takes $O(|P| + \log^k n \log\log n + occ)$ time to find the occurrences of a given pattern P without wildcards. Obviously, the size of the index implies a prohibitive amount of memory for applications involving more than a few wildcards. Lam et al. [11] have given another index, which requires only $O(n)$ words and also avoids a time complexity exponential in the number of wildcards. Precisely, the time required is $O(|P| \log n + \gamma + occ)$ time, where γ is defined as follows. Assume that the text T contains $\ell \geq 1$ groups of consecutive wildcards. I.e., $T = T_1\phi^{k_1}T_2\phi^{k_2}\ldots\phi^{k_\ell}T_{\ell+1}$, where $k_1, k_2, \ldots k_\ell \geq 1$, and each T_i contains no wildcards. Define γ to be the sum, over all T_i's, of the number of occurrences of T_i in P. Note that γ is upper bounded by $|P|(\ell + 1)$.[1] Both indexes can be extended to handle patterns with wildcards.

When we index long genomic sequences (e.g., the human genome which has about three billion characters), even an $O(n)$-word or $O(n \log n)$-bit data structure is still too large. In this paper, we give a succinct index for a text containing wildcard characters. Precisely, assume that T has $\ell \geq 1$ wildcard groups, the space complexity is $(3 + o(1))n \log \sigma + O(\sigma \log n) + O(\ell \log n)$ bits. For practical applications, the last two terms can often be absorbed into $o(n \log \sigma)$, and the pattern matching time of the new index compares favorably with the previous indexes. It is useful to define $\hat{\ell}$ to be the number of distinct wildcard-group lengths (i.e., the number distinct elements in the set $\{k_1, k_2, \ldots, k_\ell\}$; e.g., if $k_i = 1$ for all i, then $\hat{\ell} = 1$). Given a pattern P, our new index can find all occurrences of P in $O(|P|(\log \sigma + \min(|P|, \hat{\ell}) \log \ell) + \gamma \log_\sigma \ell + occ \log^{1+\epsilon} n)$ time for any $\epsilon > 0$.

In the course of deriving the above solution for indexing wildcards, we have also obtained a succinct index for the dictionary matching problem, which is another classical matching problem not involving wildcards. In this problem, we are required to index a set of patterns $P_1, P_2, \ldots, P_d$ with total length n. Given a query text T, the index is required to locate the occurrences of all P_i in T. Aho and Corasick [1] were the first to give an $O(n)$-word index for the dictionary matching problem. Chan et al. [3] have improved the space complexity to $O(n\sigma)$ bits, and recently Hon et al. [10] gave a succinct index using $(1 + o(1))n \log \sigma + O(d \log n)$ bits. The matching time for any text T is $O(|T|(\log^\epsilon n + \log d) + occ)$. In this paper we present a different way to derive a succinct index for the dictionary

[1] [11] has given a more practical upper bound of γ. Define the prefix complexity of the T_i's to be the maximum number of T_j's that are prefixes of the same T_i. Then γ is at most $|P|$ times the prefix complexity. In practice, wildcards are sparse and the prefix complexity is often a small constant.

matching problem. The new index increases the space to $(2 + o(1))n \log \sigma + O(d \log n)$ bits, but reducing the matching time to $O(|T|(\log \sigma + \log d) + occ)$.

Organization of the paper. In Section 2, we will review several data structures in the literature for indexing text (without wildcards), as well as for indexing geometric data on a two-dimensional plane. In Section 3, we describe the core elements of our succinct index, which include BWT and a new solution to the dictionary matching problem. In Section 4, we present the details of matching with wildcards in the text.

2 Preliminaries

Throughout this paper, we consider texts and patterns with characters chosen from an alphabet Σ of size σ. The text can contain one or more wildcard character ϕ, which is a special character not in Σ, and which can match any character in Σ. Our data structures would make use of two additional symbols \$ and # not in Σ. We assume that \$ is lexicographically smaller than all characters in Σ, and # greater than all characters in Σ. Below we review several data structures for text indexing (without wildcards), as well as points and rectangles in a two dimensional plane.

2.1 Suffix Array

Let $T[1..n]$ be a text that does not contain wildcard character and ends with a special character \$. A suffix of T is a substring $T[j..n]$ where $1 \leq j \leq n$. We sort all suffixes of T in lexicographical order and store their starting positions in an integer array $SA[1, n]$. Intuitively, $SA[i]$ gives the starting position of the i-th smallest suffix of T, or equivalently, the suffix with rank i.

Consider a pattern X. Inside SA, all the suffixes of T that contain X as a prefix appear in consecutive entries. We define the SA range X to be $[s, r]$ if there are $s' = s - 1$ suffixes lexicographically smaller than X, and r suffixes smaller than or equal to X. If X does not appear in T, then $s - 1 = r$ and the SA range has a right boundary (r) smaller than the left boundary (s). In this case, we say that the SA range of X is *empty*.

2.2 Burrows-Wheeler Transform (BWT)

Burrows-Wheeler Transform (BWT) was first proposed as a compression technique [2]. Later it was found that BWT can support pattern matching efficiently when equipped with auxiliary data structures. Let $T[1..n]$ be a text (containing no wildcard). Assume $T[n] =$ \$. The BWT of T is a sequence of n characters such that the i-th character is the character in T just preceding the rank-i suffix of T. Precisely, $BWT[i] = T[j - 1]$ where $j = SA[i]$ and $SA[i] \neq 1$. If $SA[i] = 1$, $BWT[i] =$ \$.

BWT can be used to compute the SA range of any pattern if it is equipped with auxiliary data structures to compute the functions $Count(c)$ and $Appear(i, c)$. For

any character c, $Count(c)$ gives the number of characters in T that are lexicographically smaller than c, and $Appear(i, c)$ returns the number of times c appears in the prefix $BWT[1..i]$. Suppose that the SA range $[s, r]$ of a string X is given. Then, for any character c, we can find the SA range of cX as $[Count(c) + Appear(s-1, c) + 1, Count(c) + Appear(r, c)]$ [6].

A straightforward implementation of the *Appear* function requires $O(n\sigma \log n)$ bits. To reduce the space requirement, we use the *wavelet tree* implementation proposed by Ferragina et al. [7]. It only uses $n \log \sigma + o(n \log \sigma)$ bits, but it is slower, taking $O(\log \sigma)$ time to serve each function call. On the other hand, with the wavelet tree implementation, we no longer need to store T or BWT explicitly, since it supports retrieving any single character of BWT in $O(\log \sigma)$ time. In summary, BWT together the auxiliary data structures occupy $n \log \sigma + o(n \log \sigma) + O(\sigma \log n)$ bits and can support pattern matching efficiently, as stated in the following lemma.

Lemma 1. *Let P be a pattern of m characters. The SA ranges of all suffixes of P can be computed in $O(m \log \sigma)$ time.*

2.3 Orthogonal Range Search

Consider a set G of ℓ points on a two-dimensional plane. Given a rectangle $R = (x_1, y_1) \times (x_2, y_2)$, we want to find all the points in G that are enclosed by R.

Lemma 2. *[13] Given ℓ points with coordinates in $[1..n]$, we can build an $O(\ell \log n)$-bit data structure such that given a query rectangle R, all the points enclosed by R can be reported in $O(\log \ell + t \log^{\epsilon} \ell)$ time, where t is the number of answers and $\epsilon > 0$.*

2.4 Point Enclosure Problem

Consider a set H of ℓ rectangles on a two-dimensional plane. Given a query point $q = (x, y)$, we want to find efficiently all the rectangles in H that enclose q.

Lemma 3. *[4] Given ℓ rectangles on a 2-D plane, we can build an $O(\ell)$-word data structure such that given a query point q, all the rectangles enclosing q can be reported in $O(log\ell + t)$ time, where t is the number of answers.*

3 Succinct Representation of Non-wildcard Characters

Consider a text T of n characters. Suppose $T = T_1 \phi^{k_1} T_2 \phi^{k_2} \ldots T_\ell \phi^{k_\ell} T_{\ell+1}$, where ϕ^{k_i} denotes a group of k_i consecutive wildcards, and each T_i does not contain any wildcard. Below, each T_i is called a text segment. In this section we show how to index the T_i's. We make use of BWT and the point enclosure data structure. The former allows us to determine whether each T_i is a prefix of a given pattern X in constant time. This prefix matching capability, together with the point enclosure data structure, allow us to have a faster index for dictionary matching, i.e., to find out the occurrences of every T_i in a given pattern X. In Section 4, we will show how to make use of these indexes to perform wildcard matching.

3.1 BWT and Prefix Matching

Define $TS = T_1\$T_1\#T_2\$T_2\#\ldots T_{\ell+1}\$T_{\ell+1}\#$, where \$ and # are new symbols assumed to be lexicographically smaller than and greater than all symbols in T, respectively. We construct the BWT index (including the necessary auxiliary data structures) to support pattern matching for TS. We denote this index as *BWT-TS*. Note that with *BWT-TS*, we no longer need to store the text TS explicitly as the index can support pattern matching TS. *BWT-TS* uses $(2 + o(1))n \log \sigma + O(\sigma \log n)$ bits. Furthermore, we explicitly store the SA range of each T_i (with respect to the suffixes of TS), using $(\ell + 1) \log n$ bits.

Below, an SA range always makes reference to the suffixes of TS. By Lemma 1, for any pattern $P[1..m]$, we can use *BWT-TS* to find the SA ranges of the suffixes $P[m..m], P[m-1..m], \ldots, P[1..m]$ in $O(m \log \sigma)$ time. In the rest of this section, we show how to exploit the SA ranges of a suffix $X = P[j..m]$ and a text segment T_i to determine whether X is a prefix of T_i, and more importantly, whether T_i is a prefix of X.

Note that X may or may not appear in any T_i, and the SA range $[s, r]$ of X may be empty ($s - 1 = r$) or non-empty ($s \leq r$). When X has a non-empty SA range, it is straightforward to determine whether X is a prefix of a text segment T_i, or vice versa. See the following lemma. The duplicate structure of TS is needed to handle the case when X has an empty SA range.

Lemma 4. *Suppose that the text segment T_i has SA range $[p, q]$. For any string X, if the SA range $[s, r]$ of X is non-empty, then* **(i)** *X is a prefix of T_i if and only if $s \leq p \leq q \leq r$; and* **(ii)** *T_i is a prefix of X if and only if $p \leq s \leq r \leq q$. Both conditions can be determined in constant time.*

Proof. We only prove (i), as (ii) is symmetric. Suppose X is a prefix of T_i. The SA range of X encloses all suffixes with prefix X, so the SA range of T_i must be enclosed by the SA range of X. Hence, $s \leq p \leq q \leq r$. Conversely, suppose $s \leq p \leq q \leq r$. The SA range of X encloses all suffixes with the prefix X. Since the SA range of T_i is a subrange of $[s, r]$, all suffixes with the prefix T_i must also have X as the prefix. Thus, X is a prefix of T_i.

It remains to consider the case when X has an empty SA range. In this case, X does not occur anywhere in TS, and X is not a prefix of any text segment T_i. However, T_i can still be a prefix of X. To determine this case is no longer straightforward. The following lemma exploits the duplicate structure of each T_i in TS to derive a simple condition.

Lemma 5. *Suppose that the text segment T_i has SA range $[p, q]$. For any string X, if the SA range $[s, r]$ X is empty (i.e., $s - 1 = r$), then T_i is a prefix of X if and only if $p \leq r < s \leq q$. This can be determined in constant time.*

Proof. Suppose that T_i is a prefix of X. Since X has an empty SA range and T_i has a non-empty one, T_i is a proper prefix of X. Recall that \$ is smaller than any character in Σ, and hence $T_i\$$ is lexicograpically smaller than X. Similarly, $T_i\#$ is lexicograpically greater than X. If $[s, r]$ is an empty range, $s - 1 = r$ and $r < s$. It remains to prove the other two inequalities: (1) $p \leq r$; (2) $s \leq q$.

(1) By definition of $[p, q]$, the p-th smallest suffix of TS contains $T_i\$$ as a prefix. This prefix is smaller than X, and hence there are at least p suffixes of TS smaller than X. Therefore, $s - 1 \geq p$ and $r = s - 1 \geq p$.
(2) By definition of $[p, q]$ and $\#$, the q-th smallest suffix of TS contains $T_i\#$ as a prefix, and this prefix is greater than X. There are at most $q - 1$ suffixes of TS smaller than or equal to X. Therefore, $r \leq q - 1$ and $s = r + 1 \leq q$.

Conversely, if $p \leq r < s \leq q$, we can prove that T_i is a prefix of X. Let X' be the prefix comprising the first $|T_i|$ characters of X (or equal to X if X is shorter than T_i). For the sake of contradiction, we assume that T_i is not a prefix of X and consider the scenarios when X' is larger than T_i or smaller than T_i. If $X' > T_i$, TS contains at least q suffixes smaller than X, and $s - 1 \geq q$. It contradicts that $s \leq q$. If $T_i > X'$, then there are at most $p - 1$ suffices that are smaller than X, and $s - 1 \leq p - 1$. It contradicts that $r = s - 1 \geq p$.

3.2 Dictionary Matching

Given the text segments $T_1, T_2, \ldots, T_{\ell+1}$ and a pattern $P[1..m]$, the dictionary matching problem is to report the occurrences of all T_i that appear in P. In this section, we show how to make use of $BWT\text{-}TS$ (defined in the previous section) and a point enclosure index to perform dictionary matching in a more efficient way than the existing indexes in the literature. The overall space requirement is $(2 + o(1))n \log \sigma + O(\sigma \log n) + O(\ell \log n)$ bits, and the dictionary query can be answered in $O(m \log \sigma + m \log \ell + \gamma)$ time, where γ denotes the total number of occurrences. This result, when compared with the work of Hon et al. [10], doubles the space requirement, but improves the dominating term of the time complexity from $m \log^{\epsilon} n$ to $m \log \sigma$.

Suppose that a text segment T_i appears in P. Then T_i must be a prefix of some suffix of P. To find out such occurrences, we consider each suffix $P[j..m]$ of P separately and find all T_i's that are a prefix of $P[j..m]$. First of all, we use Lemma 1 to compute the SA ranges (with respect to TS) of every suffix $P[j..m]$. Using Lemmas 4(ii) and 5, we can check whether T_i, for all i in $[1, \ell + 1]$, is a prefix of $P[j..m]$ in $O(\ell)$ time. We can speed up this checking process for each $P[j..m]$ to $O(\log \ell)$ time by a reduction to a point enclosure problem defined as follows.

> For each T_i with SA range $[p, q]$, we consider the rectangle $(p, p) \times (q, q)$ in the two-dimensional plane. Let H be the set of all the $\ell + 1$ rectangles associated with the T_i's. We build an $O(\ell \log n)$-bit index for point enclosure query. For each $P[j..m]$, we transform its SA range $[s, r]$ to a query point $x_j = (s, r)$. By Lemmas 4(ii) and 5, T_i is a prefix of $P[j..m]$ if and only if the rectangle of T_i encloses x_j.

Lemma 6. *We can build an index for $T_1, T_2, \ldots, T_{\ell+1}$ using $(2 + o(1))n \log \sigma + O(\sigma \log n) + O(\ell \log n)$ bits. Then, given a pattern P, the occurrences of all T_i in P can be computed in $O(m \log \sigma + m \log \ell + \gamma)$ time, where γ is the total number of occurrences.*

Proof. First, we can find the SA ranges of all suffixes of P. By Lemma 1, it takes $O(m \log \sigma)$ time. H contains $\ell + 1$ rectangles. By Lemma 3, we can build an $O(\ell \log n)$-bit data structure to answer the point enclosure query of each suffix of P in $O(\log \ell + t)$ time, where t is the number of answers. In summary, the total time required to find the occurrences of all T_i in P is $O(m \log \sigma + m \log \ell + \gamma)$.

Repeated Dictionary Matching. Given a pattern P, after we have computed the γ occurrences of the text segments in P, we want to store these results in a compact way so that they can be retrieved altogether in $O(\gamma)$ time. It is indeed relatively simple to derive a scheme using only $O(m \log \ell)$ bits, i.e., independent of the size of γ. Details are as follows.

First, we observe a relationship between all text segments T_i that are a prefix of a particular suffix $P[j..m]$ of P. For any $1 \leq j \leq m$, let D_j be the set containing all such T_i's. Let $Longest(D_j)$ denote the longest T_i in the set D_j. Note that a text segment T_i is in D_j if and only if T_i is a prefix of $Longest(D_j)$. Therefore, for each T_i, we maintain a set of text segments that are each a prefix of T_i. Then, for each $P[j..m]$, we only need to store $Longest(D_j)$. The space required to store all $Longest(D_j)$ for all j is $O(m \log \ell)$ bits. To re-generate the γ answers of the dictionary matching for P, we report all T_i's that are each a prefix of $Longest(D_j)$ for all j.

It remains to show how to maintain the list of prefix text segments for each T_i. There are several possible ways. Below we make use of a compact trie, which requires $O(\ell \log \ell)$ bits. First, we build a compact trie CT for all text segments $\{T_1\$, T_2\$, .., T_\ell\$\}$. Each T_i is associated with a leaf in CT. If text segments are identical, they are associated with the same leaf. Consider any node u in CT, we denote $path(u)$ as the concatenation of all edge labels from the root to u. For each T_i, we mark the node v of CT such that $path(v) = T_i$. Then, for all nodes, we store a link to its closest marked ancestor. The space required by CT is $O(\ell \log \ell)$ bits. Given any T_i, we can recover the text segments that are a prefix of T_i by traversing the marked nodes from the leaf associated with T_i towards the root. To conclude, the space requirement is dominated by $BWT\text{-}TS$ and the SA ranges of all T_i's, which is $(2 + o(1))n \log \sigma + O(\sigma \log n) + O(\ell \log n)$ bits.

Lemma 7. *Using CT, we can retrieve, for any T_i, all the text segments that are each a prefix of T_i in $O(t)$ time, where t is the number of results.*

4 Matching with Wildcards

Finally we come to the discussion of matching a text T containing wildcards. Assume $T = T_1 \phi^{k_1} T_2 \phi^{k_2} .. T_\ell \phi^{k_\ell} T_{\ell+1}$, where $k_1, k_2, \ldots, k_\ell \geq 1$, and each T_i contains no wildcards. The basic data structure is $BWT\text{-}TS$ (as defined in Section 3.1), which indexes all the text segments T_i of T. Furthermore, we index the reverse of each T_i, which is denoted $\overleftarrow{T_i}$ below. Let $TP = \overleftarrow{T_1}\$\overleftarrow{T_2}\$ \ldots \overleftarrow{T_\ell}\$$, and denote $BWT\text{-}TP$ as the index comprising the BWT of TP and the required auxiliary data structures (as stated in Section 2.2). $BWT\text{-}TS$ and $BWT\text{-}TP$ together occupy $(3 + o(1))n \log \sigma + O(\sigma \log n)$ bits. Note that TP doesn't have the

duplicate structure of TS. We only need $BWT\text{-}TP$ to support constant-time checking whether a string X is a prefix of some $\overleftarrow{T_i}$, but not vice versa. We also store the SA ranges of all T_i's with respect to TS, as well as the SA ranges of all $\overleftarrow{T_i}$'s with respect to TP. They require $O(\ell \log n)$ bits.

Additional auxiliary data structures (such as for indexing the SA ranges of the T_i's) will be given in the discussion below; they only use $o(n \log \sigma) + O(\ell \log n)$ bits.

Let P be a given pattern of m characters. Following Lam et al. [11], we divide the problem of matching P with T into three cases.

Type 1: P is a substring of some T_i, where $1 \leq i \leq \ell + 1$.
Type 2: P occurs in $T[u..v]$ which contains exactly one wildcard group ϕ^{k_j}.
Type 3: P occurs in $T[u..v]$ which contains two or more wildcard groups.

Below we show how to make use of $BWT\text{-}TS$, $BWT\text{-}TP$ and some auxiliary data structures to match the pattern efficiently in each case.

4.1 Type 1 Matching

This is the simplest case and it does not involve any wildcards. We simply search for P in $BWT\text{-}TS$. The required SA range can be computed in $O(n \log \sigma)$ time. The only technical difficulty is how to retrieve the occurrences of P given the SA range of P with respect to TS. The problem becomes trivial if we can keep a suffix array of TS, which requires $O(n \log n)$ bits. Below we show that with a suitable sampling of the suffix array, we can reduce the space to $o(n \log \sigma)$, while allowing each occurrence to be retrieved in $O(\log^{1+\epsilon} n)$ time for any $\epsilon > 0$.

Lemma 8. *We can build an $o(n \log \sigma)$-bit auxiliary data structure such that, given the SA range of a pattern P, the occurrences of P in TS can be reported in $O(occ_1 \log^{\epsilon+1} n)$ time, where occ_1 is the number of type-1 occurrences.*

Proof. Let β be the sampling factor. We show that an index of $O(\frac{n}{\beta} \log n)$ bits would allow us to access an value in the suffix array of TS in $O(\beta \log \sigma)$ time.

Let M be a bit vector of length $|TS|$. Initially, $M[i] = 0$ for all i. Then we mark every $M[i] = 1$ where $SA[i] = k\beta$ and $0 \leq k \leq \left\lceil \frac{n}{\beta} \right\rceil$. We store the tuple $(i, SA[i])$ where $M[i]$ is marked with 1 in ascending order of i. Suppose we want to retrieve $SA[j]$ which has not been stored up. Let $j_0 = j$. We will have to find an index j_y such that the tuple $(j_y, SA[j_y])$ is stored and $SA[j_0] - SA[j_y] < \beta$. In general, we can find the index j_x by backward searching $BWT\text{-}TS$ with character $BWT\text{-}TS[j_{x-1}]$. We recurively obtain $j_1, j_2, j_3..$ until we find j_y such that the tuple $(j_y, SA[j_y])$ is stored. Tuple can be retrieved in constant time if a rank and select data structure has been built on M. Then, we report $SA[j] = SA[j_y] + y$. The searching time for a character in $BWT\text{-}TS$ is $O(\log \sigma)$. Since $y < \beta$, we can compute $SA[j]$ in $O(\beta \log \sigma)$ time.

Let $\beta = \lceil \log^{\epsilon} n \log_{\sigma} n \rceil$ for some $\epsilon > 0$. The space requirement of the sampled SA plus the rank and select index is $o(n \log \sigma)$. The access time of an entry in the suffix array becomes $O(\log^{1+\epsilon} n)$.

4.2 Type 2 Matching

For type 2 matching, we are interested in matching a given pattern $P[1..m]$ with $T_i\phi^{k_i}T_{i+1}$ for all $1 \leq i \leq \ell$. More specifically, we want to find out whether, for some $1 \leq a \leq m$, $P[1..a]$ is a suffix of T_i, and $P[a + k_i + 1..m]$ is a prefix of T_{i+1}. The first condition can be rewritten as $\overleftarrow{P[1..a]}$ is a prefix of $\overleftarrow{T_i}$. In other words, both conditions involve prefix matching, so we can exploit $BWT\text{-}TP$ and the SA ranges of $\overleftarrow{T_i}$'s, as well as and $BWT\text{-}TS$ and the SA ranges of T_i's. By Lemma 4(i), we would first compute the SA ranges of the suffixes of P and $\overleftarrow{P}$; then for any fixed i and a, it takes constant time to check whether $\overleftarrow{P[1..a]}$ is a prefix of $\overleftarrow{T_i}$, and $P[a + k_i + 1..m]$ is a prefix of T_{i+1}. Finding all the SA ranges requires $O(m \log \sigma)$ time, and then the naive implementation of type-2 matching requires $O(m\ell)$ time.

For genomic sequences, we observe that the number of wildcard groups (i.e., ℓ) is usually not a small constant, but the number of distinct wildcard group sizes k_i's is a small constant. Recall that the latter is denoted by $\hat{\ell}$. In fact, it is often the case that most groups contain only one wildcard. This motivates us to further improve the time complexity to something depending on $\hat{\ell}$ instead of ℓ. Below we show how to index the SA ranges of the T_i's using an orthogonal range search index. Then the time complexity can be reduced to to $O(m(\hat{\ell}) \log \ell + occ_2 \log^{\epsilon} \ell)$ time, where occ_2 is the number of type-2 occurrences of P.

Consider any integer b which is equal to some wildcard group size k_i. Let $W(b)$ denote all the wildcard groups that have size b, i.e., $W(b) = \{i \mid k_i = b \text{ and } 1 \leq e \leq \ell\}$. We want to conduct type-2 matching for all the wildcard groups in $W(b)$ together. Given a position a of P, we want to find, for all i in $W(b)$, whether $P[1..a]$ is a suffix of T_i and $P[a + b + 1..m]$ is a prefix of T_{i+1}.

Lemma 9. *We can build an $O(\ell \log n)$-bit data structure to store the SA ranges of $\overleftarrow{T_i}$'s and the SA ranges of T_i's. Then, for any wildcard group size b, given a pattern $P[1..m]$ and a position $1 \leq a \leq m$, we can find in $O(\log \ell)$ time the number of $i \in W(b)$ such that $\overleftarrow{P[1..a]}$ is a prefix of $\overleftarrow{T_i}$ and $P[a + b + 1..m]$ is a prefix of T_{i+1}. Furthermore, if there are t such i's, we can report them in $O(t \log^{\epsilon} \ell)$ time for some $\epsilon > 0$.*

Proof. We make use of orthogonal range search on a two-dimensional plane. Consider any wildcard group size b. We define a set G_b of points as follows. For each wildcard group $i \in W(b)$, let the SA range of $\overleftarrow{T_i}$ on TP be (s', r') and the SA range of T_{i+1} on TS be (s, r). We add the point (s', s) into G_b. Given any position a on P, let R_a be the rectangle $(x_1, y_1) \times (x_2, y_2)$ where (x_1, y_1) and (x_2, y_2) are the SA range of $\overleftarrow{P[1..a]}$ on TP and the SA range of $P[a + b + 1..m]$ on TS, respectively. We find all the points on G_b that is enclosed by the query range R_a. A point in G_b represents a wildcard group k_i, it is enclosed by R_a if and only if the SA range of $\overleftarrow{P[1..a]}$ encloses the SA range of $\overleftarrow{T_i}$ and the SA range of $P[a + b + 1..m]$ encloses the SA range of T_{i+1}. By Lemma 2, an $O(\ell \log n)$-bit data structure can be built for all $\hat{\ell}$ distinct wildcard group sizes, then we can

determine the number of points enclosed by R_a in $O(\hat{\ell} + \log \ell)$ time, and retrieve each point in $O(\log^{\epsilon} \ell)$ time.

There are ℓ wildcard groups. The total number of points in all $\hat{\ell}$ orthogonal range search indexes is ℓ. Therefore, the total space required by the orthogonal range search indexes is $O(\ell \log n)$ bits.

Let us summarize the computation for type-2 matching for any given pattern P of m characters. We compute the SA ranges of all $\overleftarrow{P[1..a]}$ with respect to TP and the SA ranges of all $P[a..m]$ with respect to TS in $O(m \log \sigma)$ time. There are $\min(m, \hat{\ell})$ different b's for P. For each b, we consider every position a on P. All type-2 matches can be found in $O(m \log \ell + t \log^{\epsilon} \ell)$ time, where t is the number of occurrences. Summing over possible wildcard group size, we obtain the following lemma.

Lemma 10. *Given a pattern of m characters, all type-2 matches can be located in $O(m(\log \sigma + \min(m, \hat{\ell}) \log \ell) + occ_2 \log^{\epsilon} \ell)$ time, where occ_2 is the number of type-2 occurrences and ϵ is an arbitrary positive constant.*

4.3 Type 3 Matching

Type-3 matching occurs when a pattern P matches with a substring $T[j..j+m]$ which contains at least two groups of wildcards. In this case, P contains at least a whole text segment T_i. Therefore, we will first find out all T_i completely included in P, and then verify whether each such T_i can be extended to form a type-3 matching.

The first step is equivalent to performing a dictionary matching to report all T_i that occurs in P. By Lemma 6, we could find all T_i that occurs on P in $O(m \log \sigma + m \log \ell + \gamma)$ time, where γ is the total number of occurrences of the T_i's in P. If T_i occurs in P with starting position x, then it is possible that P occurs in T with starting position $y = t_i - x + 1$, where t_i is the starting position of T_i in T. Using $BWT\text{-}TS$ and $BWT\text{-}TP$, we can apply Lemma 4(i) to verify each candidate position y in constant time. Details are as follows.

First of all, we collect all the γ candidate positions y in an array $A[1..n]$ as follows. Initially, all entries of A are set to zero. We employ the constant time initialization technique[14] on A. The access time to any cell in A remains constant. Each time we find a candidate position y of P, we increment $A[y]$ by 1. The working space required by A is $O(n \log \ell)$ bits.

Consider each y with $A[y] > 0$. We want to verify whether P matches $T[y..y+m-1]$. Let T_f be the first text segment whose starting position $t_f \geq y$. Let T_g be the last text segment that ends at or before $y+m-1$ (i.e., $t_g \leq y+m-|T_i|$). Note that $g \geq f$. A position y defines a type-3 matching of P if and only if the following three conditions hold.

(1) $A[j] = g - f + 1$.
(2) If $y < t_f - k_{f-1}$ (i.e., the wildcard group $\phi^{k_{f-1}}$ starts after $T[y]$), then $P[1..t_f - k_{f-1} - y]$ is a suffix of T_{f-1}, or equivalently, $\overleftarrow{P[1..t_f - k_{f-1} - y]}$ is a prefix of $\overleftarrow{T_{f-1}}$.
(3) If $t_{g+1} \leq y + m - 1$ (i.e., the wildcard group ϕ^{k_g} ends before $T[y+m-1]$), then $P[t_{g+1} - y + 1..m]$ is a prefix of T_{g+1}.

Suppose that we have computed the SA ranges of all suffixes of P with respect to TS, as well as the SA ranges of all the suffixes of $\overleftarrow{P}$ with respect to TP. Then, by Lemma 4(i), we can make use of $BWT\text{-}TP$ and $BWT\text{-}TS$ to verify condition (2) and condition (3) in constant time. We conclude with the following lemma.

Lemma 11. *All type-3 matches can be located in $O(m \log \sigma + m \log \ell + \gamma)$ time. The working space required is $O(n \log \ell + m \log n)$ bits.*

Theorem 1. *Combining the results on type-1, type-2 and type-3 matching, we can find all occurrences of a given pattern P of m characters in $O(m(\log \sigma + \min(m, \hat{\ell}) \log \ell) + occ_1 \log^{\epsilon+1} n + occ_2 \log^{\epsilon} \ell + \gamma)$ time, where occ_1 and occ_2 denote the number of type-1 and 2 occurrences respectively, and γ is the occurrences of all text segments in P. The index space required is $(3+o(1))n \log \sigma + O(\sigma \log n) + O(\ell \log n)$ bits. The working space required is $O(n \log \ell + m \log n)$ bits.*

Reducing the Working Space. The solution to the type-3 matching demands a working space $O(n \log \ell + m \log n)$ bits. The first term is way too much. Below we show how to trade the running time for a solution that requires less working space. At the end, we obtain a solution that requires only $O(n \log \sigma + m \log n)$-bit working space, but the verification time for the γ candidates would increase to $O(\gamma \log_\sigma \ell)$. Intuitively, the idea is to split the array A into a number of subarrays. Then, we parse the γ dictionary matching results several times to cover all candidate positions.

By Lemma 7, we could retrieve the γ matching results for multiple times. Precisely, we could retrieve the γ matching results for d times in $O(d\gamma)$ time. Now, we split the entries in the array A into a number of groups. In each group, there are $\rho = \lfloor \frac{n \log \sigma}{\lceil \log \ell \rceil} \rfloor$ consecutive entries of array A. No entry in A is contained in more than one group. Therefore, there are $O(\log_\sigma \ell)$ groups of entries in total. Each group corresponds to a range of entries in A.

Let $B[1..\rho]$ be an array of integers. The space required by B is $O(\rho \log \ell) = O(n \log \sigma)$ bits. We repeat the process to mark the candidate positions, however, we mark the candidate positions on array B instead. We set $b = 1, \rho + 1, 2\rho + 1, \ldots, \rho \log_\sigma \ell + 1$. For each b, we mark on the array B by increasing the entry $B[j']$ by one if the candidate position $j = t_i - k + 1$ falls between b and $b + \rho - 1$, where $j' = j - b + 1$. We ignore all candidate positions that do not fall between b and $b + \rho - 1$. After we have marked array B for all γ dictionary matching occurrences, for each $B[j'] > 0$, it indicates an candidate position $j = j' + b - 1$. Then, we verify the candidate position j as mentioned in previous section. We repeat the marking process for another b until all positions on T are covered. The process marks the array for $\log_\sigma \ell$ times.

Lemma 12. *Type 3 matches can be located in $O(m \log \sigma + m \log \ell + \gamma \log_\sigma \ell)$ time. The working space required is $O(m \log n + n \log \sigma)$ bits.*

References

1. Corasick, M., Aho, A.: Efficient string matching: An aid to bibliographic search. Commun. ACM 18(6), 333–340 (1975)
2. Burrow, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, California (1994)
3. Chan, H.L., Hon, W.K., Lam, T.W., Sadakane, K.: Compressed indexes for dynamic text collections. ACM Transactions on Algorithms 3(2) (2007)
4. Chazelle, B.: Filtering search: a new approach to query answering. SIAM J. Comput. 15(3), 703–724 (1986)
5. Cole, R., Gottlieb, L.A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Proceedings of Symposium on Theory of Computing, pp. 91–100 (2004)
6. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings of Symposium on Foundations of Computer Science, pp. 390–398 (2000)
7. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: An alphabet-friendly FM-index. In: Apostolico, A., Melucci, M. (eds.) SPIRE 2004. LNCS, vol. 3246, pp. 150–160. Springer, Heidelberg (2004)
8. Fischer, M.J., Paterson, M.S.: String matching and other products. Technical Report MAC TM 41, Massachusetts Institute of Technology, Cambridge, MA, USA (January 1974)
9. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proceedings of Symposium on Theory of Computing, pp. 397–406 (2000)
10. Hon, W.K., Shah, R., Vitter, J.S., Lam, T.W., Tam, S.L.: Compressed index for dictionary matching. In: IEEE Data Compression Conference, pp. 23–32 (2008)
11. Lam, T.-W., Sung, W.-K., Tam, S.-L., Yiu, S.-M.: Space efficient indexes for string matching with don't cares. In: Tokuyama, T. (ed.) ISAAC 2007. LNCS, vol. 4835, pp. 846–857. Springer, Heidelberg (2007)
12. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comput. Surv. 39(1) (2007)
13. Nekrich, Y.: Orthogonal range searching in linear and almost-linear space. Computational Geometry: Theory and Applications 42(4), 342–351 (2009)
14. Torczon, L., Briggs, P.: An efficient representation for sparse sets. In: ACM Letters on Programming Languages and Systems 2, pp. 59–69 (1993)
15. Rahman, M.S., Iliopoulos, C.S.: Pattern matching algorithms with don't cares. In: Proceedings of 34th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), vol. 2, pp. 116–126 (2007)