Information Processing Letters 146 (2019) 17–19

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Applying the Positional Burrows–Wheeler Transform to all-pairs Hamming distance

Veli Mäkinen [*], Tuukka Norri

Helsinki Institute for Information Technology & Department of Computer Science, University of Helsinki, PL 68 (Gustaf Hällströmin katu 2b), 00014 Helsingin yliopisto, Finland

ARTICLE INFO

Article history:
Received 16 March 2018
Received in revised form 22 December 2018
Accepted 5 February 2019
Available online 8 February 2019
Communicated by Marek Chrobak

Keywords:
Data structures
Hamming distance
Positional Burrows–Wheeler Transform
Phylogenetic inference

ABSTRACT

Crochemore et al. gave in WABI 2017 an algorithm that from a set of input strings finds all pairs of strings that have Hamming distance at most a given threshold. The proposed algorithm first finds all long enough exact matches between the strings, and sorts these into pairs whose coordinates also match. Then the remaining pairs are verified for the Hamming distance threshold. The algorithm was shown to work in average linear time, under some constraints and assumptions.

We show that one can use the Positional Burrows–Wheeler Transform (PBWT) by Durbin (Bioinformatics, 2014) to directly find all exact matches whose coordinates also match. The same structure also extends to verifying the pairs for the Hamming distance threshold. The same analysis as for the algorithm of Crochemore et al. applies.

As a side result, we show how to extend PBWT for non-binary alphabets. The new operations provided by PBWT find other applications in similar tasks as those considered here.

1. Introduction

Given a set $\mathcal{S}$ of n strings all of the same length m and drawn from alphabet Σ of size σ, we would like to compute the Hamming distance of all pairs $\{S_1, S_2\}$, where $S_1, S_2 \in \mathcal{S}$, up to a distance limit l. This is called the *All-Pairs Hamming Distance* problem. The problem was studied in [1] (first published in [2]) as a preprocessing step for building distance-based phylogenetic trees and for querying typing databases, where pairs with large distance are not required. In what follows, we give an alternative average linear time algorithm for this problem, by replacing the steps of the algorithm in [1] with the usage of the Positional Burrows–Wheeler transform [3].

[*] Corresponding author.
E-mail address: veli.makinen@helsinki.fi (V. Mäkinen).

2. Positional Burrows–Wheeler transform

The *Positional Burrows–Wheeler Transform* [3] is a method of sorting a set of strings of the same length at each character position in the lexicographic order of the reverse prefixes up to that position. One of its applications is to make it easy to identify longest matching substrings. The transformation is originally defined for the binary alphabet but is easily extended for the alphabet $[1..\sigma]$ as follows.

Following Durbin's notation, for each index i from 1 to n define $a_k[i]$ as the identifier or index of the string $S \in \mathcal{S}$ which is at the i-th position of the ordered reverse prefixes at string position $k-1$, that is, having sorted the strings by each column up to and including column index $k-1$, $a_k[i]$ shall have the identifier of the i-th string. As a result, a_1 shall contain the original order. Also define y_i^k to be the i-th string in this sorted order. In addition, the divergence arrays d_k for $k \in [2..m+1]$ are filled. They contain values for each row index i such that $d_k[i]$ is the smallest value j

https://doi.org/10.1016/j.ipl.2019.02.003

```
Input: $a_k$ and $d_k$, $d_k$ prepared for range maximum queries.
Output: $a_{k+1}$ and $d_{k+1}$.

    ▷ Count the instances of each character.
 1: Create empty array $C$ of size $\sigma + 1$.
 2: $C[1] \leftarrow 0$
 3: for $i \leftarrow 1$ to $n$ do                              ▷ Iterate the strings.
 4:     $c \leftarrow y_i^k[k]$                    ▷ Take the character in the current column.
 5:     $C[c+1] \leftarrow 1 + C[c+1]$                          ▷ Increment the count of $c$.
 6: end for
 7: for $i \leftarrow 1$ to $\sigma$ do
 8:     $C[i+1] \leftarrow C[i] + C[i+1]$                  ▷ Calculate the cumulative sum.
 9: end for
    ▷ Sort the strings by the $k$-th column and build the arrays.
10: Create empty arrays $a_{k+1}$ and $d_{k+1}$ of size $n$.
11: Create array $P$ of size $\sigma$, fill it with values 0.
12: for $j \leftarrow 1$ to $n$ do                              ▷ Iterate the strings.
13:     $c \leftarrow y_j^k[k]$                    ▷ Take the character in the current column.
14:     $j' \leftarrow C[c]$                          ▷ Find the next available index in $a_{k+1}$.
15:     $C[c] \leftarrow 1 + C[c]$                              ▷ Increment the count of $c$.
16:     $a_{k+1}[j'] \leftarrow a_k[j]$ ▷ Store the identifier of the current string to $a_{k+1}$.
    ▷ Next value for $d_{k+1}$.
17:     $i \leftarrow P[c]$                    ▷ Find the previous index $i$ where $c$ occurred.
18:     if $i = 0$ then
19:         $d_{k+1}[j'] = k + 2$ ▷ If this was the first instance of $c$, store $k + 2$.
20:     else
21:         $d_{k+1}[j'] = d_k.RMaxQ(i, j)$          ▷ Otherwise make use of Obs. 1.
22:     end if
23:     $P[c] \leftarrow j$                                     ▷ Store the current index.
24: end for
```

Algorithm 2.1: BuildPrefixAndDivergenceArrays a_{k+1} and d_{k+1}.

for which $y_i^k[j,k)$ matches $y_{i-1}^k[j,k)$, that is, the starting string position of the matching suffix compared to the previous string in the sorted order. If $y_i^k[k-1] \neq y_{i-1}^k[k-1]$, the value of $d_k[i]$ is set to $k+1$. To fill the divergence arrays, the following result is utilized.

Observation 1. *Given the j-th string at text position k in the order defined by a_k, suppose i is the greatest string index less than j where $y_i^k[k] = y_j^k[k]$. Suppose i' and $j' = i' + 1$ are the corresponding string indices in a_{k+1}, i.e. it holds that $a_k[i] = a_{k+1}[i']$ and $a_k[j] = a_{k+1}[j']$. Then the following holds.*

$$d_{k+1}[j'] = \max_{i<t\leq j} d_k[t]$$

The arrays a_k may be filled by applying counting sort [4]. After filling each d_k, it may be processed for constant time *range maximum queries* in $O(n)$ time [5], denoted $d_k.RMaxQ(i,j)$. Thus, it is possible to determine the value to be placed in d_{k+1} as part of executing the counting sort to fill a_{k+1} without affecting the overall time complexity as shown in Algorithm 2.1.

3. Positive filtering

To find candidates for Hamming distance calculation, the following result is utilized.

Lemma 2. *To have Hamming distance at most l from each other, a pair of strings S and T must have a matching substring the length of which is at least $\left\lfloor \frac{m}{l+1} \right\rfloor$.*

Proof. Suppose the Hamming distance of S and T is l or less. Split the strings S and T into $l+1$ parts such that the length of each part is at most $\left\lceil \frac{m}{l+1} \right\rceil$. The differing characters may be located in at most l such parts leaving one that may not have mismatches. □

Such pairs of candidate matches may be identified by utilizing the divergence arrays d_k calculated as part of the PBWT: Split the text into substrings such that the ending points are $t_u := \left\lceil \frac{u \cdot m}{l+1} \right\rceil$, $1 \leq u \leq l+1$ and $t_0 := 0$. If two strings have a matching substring, it must hold that $d_{1+t_u}[i] \leq 1 + t_{u-1}$ for some $1 \leq i \leq n$. If there is a run of more than two such adjacent rows, all combinations of two rows in the run are candidates.

4. Verifying a pair of candidate matches

Given a pair of candidate matches S and T, we would like to verify that their Hamming distance is at most l. To this end, we start from the divergence array for string position $m+1$, that is, the position after the last character. a_{m+1} determines the order of the strings sorted by the m-th position. The positions of S and T in d_{m+1} are determined by utilizing the inverse of the a_{m+1} array defined as a_{m+1}^{-1}. Suppose the positions are i and j respectively. We determine the starting index of the final matching part $t := d_{m+1}.RMaxQ(i,j)$, which gives the maximum value in range $[i..j]$ of array d_{m+1}. Since we know that the strings will not match at position $t-1$, we consider suffixes that end at position $t-2$. Again, by determining the indices $i' := a_{t-1}^{-1}[a_{m+1}[i]]$ and $j' := a_{t-1}^{-1}[a_{m+1}[j]]$ of S and T respectively, and utilizing d_{t-1}, the starting position of the next matching part is determined. If the start of the strings is reached by doing $l+1$ or less range maximum queries, there are at most l mismatches, in which case the pair $\{S, T\}$ may be reported.

Since each position may be processed in constant time, an $O(l)$ time complexity for verifying a pair of candidate matches is achieved, matching the bound of [1] that uses very similar mechanism with different data structures. The arrays a_k^{-1} may be filled for all m columns in $O(mn)$ time in total.

5. Analysis

Since we execute the same steps in the same time complexity as in [1], we can verbatim use their analysis as an upper bound for our algorithm. Thus, our algorithm also works in average $O(mn)$ time and space (linear in the input size) under the constraint that $l < \frac{(m-l-1)\log\sigma}{\log mn}$ and assumption that the input strings are randomly generated from an independent and identically distributed source. The constraint is derived from the expected number of pairs to check [1], which is the same in both algorithms.

6. Final remarks

After the submission of this article, we implemented the described PBWT[1] and applied it on founder sequence reconstruction [6]. We conducted extensive experiments on that application and PBWT worked efficiently in practice. For the construction of phylogenetic trees this PBWT approach should have much smaller constant factors than the suffix array-based approach of [1]: The integration of PBWT to the various analysis tasks considered in [1] is an interesting direction to study.

Acknowledgements

This work was supported in part by the Academy of Finland, grant 309048.

References

[1] J.A. Carriço, M. Crochemore, A.P. Francisco, S.P. Pissis, B. Ribeiro-Gonçalves, C. Vaz, Fast phylogenetic inference from typing data, Algorithms Mol. Biol. 13 (1) (2018) 4.

[2] M. Crochemore, A.P. Francisco, S.P. Pissis, C. Vaz, Towards distance-based phylogenetic inference in average-case linear-time, in: 17th International Workshop on Algorithms in Bioinformatics, WABI 2017, in: LIPIcs, vol. 88, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2017, pp. 9:1–9:14.

[3] R. Durbin, Efficient haplotype matching and storage using the Positional Burrows–Wheeler Transform (PBWT), Bioinformatics 30 (9) (2014) 1266–1272.

[4] H.H. Seward, Information Sorting in the Application of Electronic Digital Computers to Business Operations, Tech. Rep. R-232, Thesis, Massachusetts Institute of Technology, Digital Computer Laboratory, 1954.

[5] M.A. Bender, M. Farach-Colton, The LCA problem revisited, in: Latin American Symposium on Theoretical Informatics, Springer, 2000, pp. 88–94.

[6] T. Norri, B. Cazaux, D. Kosolobov, V. Mäkinen, Minimum segmentation for pan-genomic founder reconstruction in linear time, in: 18th International Workshop on Algorithms in Bioinformatics, WABI 2018, August 20–22, 2018, Helsinki, Finland, in: LIPIcs, vol. 113, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2018, pp. 15:1–15:15.

[1] See https://github.com/tsnorri/libbio/.