Information Processing Letters 139 (2018) 49–52

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

A fast algorithm for all-pairs Hamming distances

Abdullah N. Arslan

Department of Computer Science and Information Systems, Texas A & M University – Commerce, Commerce, TX 75428, USA

ARTICLE INFO

Article history:
Received 13 December 2017
Received in revised form 7 July 2018
Accepted 11 July 2018
Available online xxxx
Communicated by Kun-Mao Chao

Keywords:
Hamming distance
Algorithms
Clustering
Distance matrix
Matrix multiplication

1. Introduction

Computing a *distance matrix* is a major step in hierarchical clustering [11] and in creating a phylogenetic tree [2]. A *distance matrix* requires computing distances for all pairs of elements. Pairwise distances are also used in calculating cluster distances [13]. In many bioinformatics applications, the elements are biological sequences (amino-acid or nucleotide sequences). The *Unweighted Pair Group Method with Arithmetic Mean (UPGMA)* [11] builds a hierarchical tree bottom up based on pairwise distances between clusters which in turn are calculated by using pairwise distances for sequences in these clusters [4,6]. The *Hamming distance* is among the distances used for this purpose. The *Hamming distance* between two strings of equal length is the number of positions at which these strings differ.

Several recent applications use distance matrices based on Hamming distances [5,8,13].

Hierarchical clustering with Hamming distance yields clonal grouping of immune cells in human. This is used in understanding the micro-evolutionary dynamics that drive successful immune responses and the dysregulation that occurs with aging or disease [5]. Single nucleotide polymorphisms (SNPs) are the most common type of genetic variation among people. A two-stage method for disease association was proposed in [13]. The first stage constructs SNP-sets by a clustering algorithm which employs Hamming distance to measure the similarity between strings of SNP genotypes and evaluates whether the given SNPs or SNP-sets should be clustered. With the resulting SNP-sets, the second stage develops an association test to examine susceptibility to the disease of interest. BugMat [8] is a program which generates a distance matrix based on Hamming distance for bacterial genomes. It is deployed as part of the Public Health England solution for M.tuberculosis genomic processing and detecting possible disease transmission. Mycobacterium tuberculosis has a genome size of about 4.4×10^6 bases. Computing Hamming distances for many pairs of long sequences is a time-consuming step. For example, this step takes about 26.5 minutes (including clustering) for 940 SNPs from 4864 subjects [13], and 3 hours for 4000 sequences [8]. In this paper we propose an algorithm for this step. The main step of our algorithm is based on matrix multiplication.

E-mail address: Abdullah.Arslan@tamuc.edu.

https://doi.org/10.1016/j.ipl.2018.07.006

Definition 1. For two given sets S_1 and S_2, *AllPairsHamDist* is the problem of computing Hamming distances for all pairs (e_1, e_2) of strings, where $e_1 \in S_1, e_2 \in S_2$.

We show that Problem *AllPairsHamDist* can be solved by performing a series of matrix multiplications. In these multiplications, the matrix elements are from a finite alphabet. We translate each of these multiplications to a multiplication of $(0,1)$-matrices in which each matrix-element is either a zero or a one. The pairwise distance matrix is calculated as the following: First, for every element of the alphabet, one matrix multiplication is performed. All such resulting product matrices are added and a summation matrix is obtained. Second, the matrix that contains Hamming distances for all pairs is calculated by subtracting this summation matrix from a constant matrix.

We also consider a special case of Problem *AllPairsHamDist* in which the input sets S_1 and S_2 are equal. That is, in this case, pairwise Hamming distances are calculated for all pairs within the same set. Our algorithm applies to this case without any change except that the matrix multiplications are of the form AA^T for matrices A obtained from $S_1 = S_2$.

The outline of this paper is the following: We describe our notation in Section 2. We present a method for a special case of matrix multiplication in Section 3. We present our algorithm for all pairs Hamming distance computations in Section 4. We conclude in Section 5.

2. Notation

Let Σ be a fixed finite alphabet for matrix elements. Let $A = (A_{i,j})_{n\times r}$ denote a matrix of size $n \times r$ whose elements are $A_{i,j}$ in Σ. For a given A and e, let A_e denote the matrix in which each element e in A is replaced by a one, and all other elements are replaced by zeros. That is,

$$A_e[i,j] = \begin{cases} 1, & \text{if } A[i,j] = e; \\ 0, & \text{otherwise.} \end{cases}$$

Let A^T denote the transpose of matrix A. For matrices A, B, let $A[i,*]$ denote the ith row of A, and $B[*,j]$ denote the jth column of B. Define a $(0,1)$-matrix as a matrix in which each element is a zero or a one.

Let $X(m)$ denote the complexity of matrix multiplication for square matrices of size $m \times m$. Let $X(w,r,c)$ denote the time complexity of multiplying two matrices of sizes $w \times r$ and $r \times c$.

3. Computing AA^T based on square matrix multiplication

Let A be an $n \times r$ matrix. We decompose matrices A and $B = A^T$ into square matrices. As a result, the multiplication AA^T is expressed using multiplications of square matrices.

The naive matrix multiplication algorithm takes $\Theta(n^3)$ time to multiply two $n \times n$ matrices. Strassen's $O(n^{2.81})$-time algorithm [12] for this problem was the first subcubic time algorithm. Following this achievement, even faster algorithms for matrix multiplication have been proposed in the literature. Let k^* denote the power k of m in fastest $O(m^k)$-time multiplication algorithm for $(0,1)$-matrices of size $m \times m$. Currently, the fastest such algorithm generalizes the Coppersmith–Winograd algorithm [3], and it has $k^* \leq 2.3728639$ [7]. Naturally, $k^* \geq 2$ because any matrix multiplication algorithm takes $\Omega(m^2)$ time for multiplying two $m \times m$ matrices. A less-obvious lower bound is that the size of any arithmetic circuit for generating the product of two matrices under certain conditions is $\Omega(n^2 \log n)$ [10].

In decomposing matrices $A = (A_{ij})_{n\times r}$ and $B = A^T$ into square matrices, there are three cases.

When $n = r$, no decomposition is necessary. The multiplication AA^T is a square matrix multiplication which can be done in $O(n^{k^*})$ time by fast matrix multiplication algorithms (e.g. [7]).

When $n > r$, the matrices and the multiplication can be decomposed in the following way: If n is not a multiple of r, then let x be the smallest number that needs to be added to make n a multiple of r. Note that $x \leq r-1$. We add x rows of zeros to A, and x columns of zeros to B. We note that this does not change the product matrix. We then decompose A and B into submatrices (blocks) of size $r \times r$ as can be seen in the writing of matrices in Eq. (1).

$$\begin{pmatrix} \left(C'_{1,1}\right)_{r\times r} & \cdots & \left(C'_{1,r}\right)_{r\times r} & \\ & \cdots & \cdots & \cdots \\ \left(C'_{n,1}\right)_{r\times r} & \cdots & \left(C'_{n,r}\right)_{r\times r} & \end{pmatrix}_{\frac{n}{r}\times\frac{n}{r}} = \begin{pmatrix} (A'_1)_{r\times r} \\ \cdots \\ (A'_{\frac{n}{r}})_{r\times r} \end{pmatrix}_{\frac{n}{r}\times 1} \left((B'_1)_{r\times r} \ldots (B'_{\frac{n}{r}})_{r\times r}\right)_{1\times\frac{n}{r}}, \tag{1}$$

where (A'_i) is the i'th block of size $r \times r$ from the top in A, and (B'_i) is the i'th block of size $r \times r$ from the left in B. In terms of such $r \times r$ blocks, A is of size $\frac{n}{r} \times 1$, and B is of size $1 \times \frac{n}{r}$. Similarly, C is composed of blocks $C'_{i,j}$ of size $r \times r$. For all $i, j \in [1, \frac{n}{r}]$, $C'_{i,j} = A'_i B'_j$. The time complexity of computing C in this case is $O(\frac{n}{r}X(r)) = O(\frac{n}{r}r^{k^*}) = O(nr^{k^*-1})$.

When $n < r$, the matrices and the multiplication can be decomposed in the following way: If r is not a multiple of n, then let x be the smallest number that needs to be added to make r a multiple of n. Note that $x \leq n-1$. We add x columns of zeros to A, and x rows of zeros to B. We note that this does not change the product matrix. We then decompose A and B into submatrices of size $n \times n$ as seen in the writing of matrices in Eq. (2).

$$C = \left(\left(A''_1\right)_{n\times n} \cdots \left(A''_{\frac{r}{n}}\right)_{n\times n}\right)_{1\times\frac{r}{n}} \begin{pmatrix} \left(B''_1\right)_{n\times n} \\ \left(B''_2\right)_{n\times n} \\ \cdots \\ \left(B''_{\frac{r}{n}}\right)_{n\times n} \end{pmatrix}_{\frac{r}{n}\times 1}, \tag{2}$$

where (A''_i) is the i'th block of size $n \times n$ from the left in A, and (B''_i) is the i'th block of size $n \times n$ from the top in B. In terms of such $n \times n$ blocks, A is of size $1 \times \frac{n}{r}$, and B is of size $\frac{n}{r} \times 1$. $C = \sum_{\ell=1}^{r/n} A''_\ell B''_\ell$. This requires r/n matrix