OptiM—a referential genome compression algorithm based on suffix array using an optimal strategy for seeking common substrings

Tingting Yi [1, a], Jianhua Chen [1, b]

[1] Information College, Yunnan University, Kunming, Yunnan 650500, China;

[a] ytt603135452@163.com, [b] chenjh@ynu.edu.cn

Abstract. In the field of referential genome compression research, how to select appropriate matching substrings, which are used to encode the target sequence, from the common substrings between the reference and the target sequences to improve the compression performance is a key issue. This study proposed the OptiM algorithm, which uses an optimal matching strategy that considers multiple factors to improve the compression performance. The candidate common substrings are reprsented as nodes and a directed graph is constructed. The storage overhead of method is measured in the form of node cost and edge cost. The nodes on the lowest cost path are selected as subsequent matching substrings through the shortest path algorithm. In the experiment, we used 8 sets of human genome as the data set for compression testing. The results show that compared with the existing algorithms, the OptiM algorithm improved the compression performance.

Keywords: gene compression; reference compression; suffix array; shortest path; optimal matching.

1. Introduction

With the advancement of genome sequencing technology, the speed of sequencing has been significantly improved, and the cost has been continuously reduced, and the volume of genomic data has increased exponentially[1], which has brought tremendous pressure on data storage and transmission. How to efficiently compress genome data has become an important research direction in the field of bioinformatics and data science[2]. A human genome sequence typically contains about 3 billion base pairs, but the genomic data between homologous species are highly similar[3]. Referential genome data compression algorithms mainly use this feature to find similar base substrings between the target and reference genome sequences and record them as common substrings. Then, valid matching substrings are selected from the common substrings, and the corresponding matching regions in the target sequence are encoded and compressed by the length of the matching substrings and their position information in the reference sequence. However, how to effectively select valid matching substrings is a difficult and important point of research. Researchers have proposed a variety of solutions. For example, the HiRGC[4] algorithm uses a greedy strategy to find common substrings in the target and reference sequences by extending them character by character, and gives priority to longer substrings to improve compression efficiency. In addition, the SCCG[5] algorithm constructs a distance relationship between the reference and the target sequences and gives priority to common substrings with smaller position increments, so that the final matching substrings can fit the position increment characteristics of the target sequence as closely as possible, thereby reducing the redundancy of the compressed data. In addition to the algorithms mentioned above, other studies have further explored matching strategies. For example, the LMSRGC[6] algorithm prioritizes the longest common substrings while retaining other common substrings that do not overlap with the longest common substrings to cover more areas of the target sequence. These methods provide a variety of solutions for substring matching strategies. However, factors such as the length, position, and increment of the common substring will affect the final compression performance, and existing algorithms usually rely on only a single factor, there is still a lack of a selection strategy that comprehensively considers multiple factors.

In view of the shortcomings of existing referential genome data compression algorithms, this study proposes the OptiM (Optimal Matching) algorithm. The algorithm uses suffix arrays to quickly find the common substrings between the reference and the target sequences. In terms of matching substrings selection strategy, the OptiM algorithm selects the matching substrings with the smallest compression storage requirement through the shortest path algorithm. First, the sorted candidate common substrings are abstracted as nodes and a directed graph is constructed. At the same time, the node cost and edge cost are defined by comprehensively considering the storage overhead of substring length, position increment, substring spacing, etc. From the starting point of the target sequence to the endpoint, the shortest path algorithm is used to traverse all nodes and calculate the path with the lowest total cost, the common substrings corresponding to the nodes along this path represent the optimal matching substrings, which will be used for subsequent. The experimental results show that compared with existing HiRGC[4], SCCG[5], and LMSRGC[6] algorithms, OptiM algorithm has better compression performance, proving the effectiveness of the matching strategy.

2. Mthod

The core goal of the OptiM algorithm is to reduce the storage requirements of intermediate files by accurately selecting the optimal matching substrings between the target and reference sequences by calculating the minimum path cost.

This study uses human genome data in FASTA format as the raw data. The first step is to preprocess the original data to improve the convenience of subsequent processing; the second step is to use the suffix array structure to find the common substrings between the reference and target sequences; The third step is to filter the common substrings to improve the running time of the next step; the fourth step, the optimal matching substrings are selected from the candidate common substrings through the optimal matching strategy. The final step is to encode and compress the target sequence with the information of the optimal matching substrings to obtain the final compression result.

2.1 Preprocessing

First, the genome sequence is modified by converting lowercase characters to uppercase, and "N" characters are removed. At the same time, in order to avoid information loss and ensure that subsequent decoding can accurately restore the original sequence, the modification information will also be stored. In addition, human genes have an antiparallel double helix structure, so the reverse complementary sequence[7] of the reference sequence can be generated according to the base pairing rule (A-T, G-C), and the reference sequence and the reverse complementary sequence can be concatenated together to form an extended sequence. Assuming that the reference sequence is 'ACCTCT', then the reverse complementary sequence of the reference sequence is 'AGAGGT', then the extend sequence is 'ACCTCTCTAGAGGT', so that more and longer common substrings can be found between the target and reference sequences, which helps to improve the matching efficiency.

2.2 Suffix array(*SA*) and longest prefix array(*LCP*)

The suffix array index structure can quickly find the common substring between the reference and target sequences, and store the substring information for subsequent use. In order to better understand this efficient search method, we need to understand the structure and working principle of suffix array.

First, the suffix array is an index structure that sorts all the suffixes of an input string in lexicographic order and records their starting positions. It is a one-dimensional array and can be written as *SA[1], SA[2], SA[3]. . . SA[n]. SA[i]*=m means that the i-th suffix of the input sequence *S* after all the suffixes are sorted in lexicographic order. It is the suffix starting from the m-th character of the input sequence S. For example, we input a string: *S* = "banana", and its suffix array

is *SA = [5, 3, 1, 0, 4, 2]*, where *SA[1]=3* means that the second suffix after sorting, which is the suffix starting from the third character in *S*, is: "ana".

Another key concept we need to know is the longest common prefix (*LCP*). The *LCP* array records the length of the longest common prefix between adjacent suffixes in the suffix array, *LCP[i] = x* means that there are at most x identical prefix characters between the *SA[i]* and *SA[i-1]*. It further narrows the calculation range of substring matching based on the suffix array, helping us find common substrings faster. For the input sequence S = "banana" and the suffix array *SA = [5, 3, 1, 0, 4, 2]*, the corresponding longest common prefix is *LCP = [0, 1, 3, 0, 0, 2]*; where *LCP[2]=3* means that the length of the longest common prefix is 3 between the two adjacent suffixes *SA[2]*:"anana" and *SA[1]*:"ana".

By the *LCP* array and its corresponding indexs of suffix array, we can extract common substrings from the string and simultaneously record their length and position. In this step, we use GPU multithreading[8] to quickly construct suffix arrays and *LCP*s to find common substrings.

2.3 Filtering common substrings

The algorithm further screens the common substrings. Firstly, we use the filtering strategy of the LMSRGC[6] algorithm to filter out the public substrings whose length is less than 21. Second, we remove the substrings that are completely covered by longer substrings. For the remaining substrings, if there is an overlap between two common substrings and the length of the non-overlapping portion exceeds the set threshold, the shorter substring is truncated and the truncated part is saved. Specifically, the common substrings are sorted from the longest to the shortest to obtain *Substring[2]*, *Substring[43]*, and *Substring[21]*.

Taking the sequence in Fig. 1 as an example, suppose that three substrings are found from the sequence, namely *Substring [2]*, *Substring [43]* and *Substring [21]*. The longest length represented by *Substring[2]* is first recorded as the common substring between the target and extended reference sequences; since the common substring represented by *Substring[43]* does not overlap with other substrings, it is also retained. Finally, *Substring [21]* is processed. As can be seen from the Fig. 1, there are 5 characters overlapping between the matching substring represented by *Substring[21]* and the previously saved common *Substring[2]*. The saved common *Substring[2]* is not changed. It is determined whether the remaining length of *Substring[21*] after removing the 5 overlapping characters meets the set threshold. If so, *Substring[21]* is truncated to form a new common substring and the corresponding *Substring[26]* is saved. The truncated *substring[26]* is then re-sorted with the remaining common substrings.

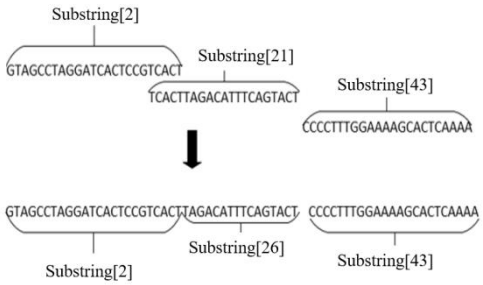

Fig. 1 Example of common substring filtering

2.4 Selection of the optimal matching substrings

In terms of matching selection strategy, OptiM aims to minimize the compression storage requirements as far as possible. After sorting the candidate common substrings in ascending order according to their starting position in the target sequence, these common substrings are represented sequentially as nodes and then, a directed graph is constructed with the nodes and edges linking them. The cost of nodes and edges is set by weighting the storage usage of the relevant information representing the common substrings (e.g, the length of a common substring, the position of a

common substring in the target sequence, the difference between the positions of two adjacent common substrings in the reference sequence). The detailed steps are described below.

After the previous filtering steps, we obtained a series of common substrings. At this point, each common substring contains three pieces of information, namely: length (*len*), starting position in the target sequence (*taridx*), and starting position in the reference sequence *(refidx*). Firstly, we sort all common substrings in ascending order according to *taridx*, and then represent all sorted candidate common substrings as nodes. Each node also contains three pieces of information {*len, taridx, refidx*}, and each node can find the corresponding common substring through its index. In addition, we construct two virtual nodes before and after the beginning and end of all nodes, namely: the starting node *Node(0):{0,0,0}* and the ending node *Node(n+1){0,tarlen,0}*, where *tarlen* represents the total length of the target sequence. The role of the starting node and the ending node is to determine the starting point and boundary of the routes.

Now we need to build directed edges between nodes to construct a directed graph, which requires judging the reachability and direction between each node. In the subsequent processing, a matching substring corresponds to a region in the target sequence, which means that there will be no overlap between two matching substrings. In other words, if we want to establish a directed edge from node i to node j, two conditions must be met: 1. The starting position of node i in the target sequence is less than that of the node j. 2. The starting position of node j in the target sequence must be located after the end position of node a in the target sequence. The evaluation formula is as follows.

$$j.taridx > i.taridx \;\&\&\; j.taridx > (i.taridx + i.len) \tag{1}$$

After building the edges of the nodes, we also need to set the cost of the nodes and edges. The region in the target sequence corresponding to the matching substrings will be represented by the length of the matching substring and its position in the reference sequence and the difference between the positions of two adjacent common substrings in the reference sequence(subtracting the position of the previous common substring from the position of the current common substring in the reference sequence). the number of characters in the strings used to represent these three quantities will be used as the encoding cost. Set the $num\{\}$ function to calculate the number of characters required to store the strings. For ease of demonstration, let the previous node be i and the next node be j. The cost H between nodes i and j is calculated as the sum of *Node_cost* and *Edge_cost*.

$$Node\_cost = num\{i.len\} + num(j.refisx - i.redidx\} \tag{2}$$

$$Edge\_cost = j.taridx - (i.taridx + i.len\} \tag{3}$$

After completing the above steps, the directed graph can be constructed, and then the path with the minimum cost can be obtained through the shortest path algorithm, the nodes on this path are the optimal matching substrings. To show the whole process more intuitively, a specific example is given below and we will demonstrate how to construct a directed graph and calculate the path with the minimum cost. Assume that as shown in Fig. 2, the target sequence length is 76, and its corresponding three candidate common substrings are *Substring[26]*, *Substring[40]*,and *Substring[75]*.Firstly, sort the candidate common substrings in ascending order of their positions in the target sequence, Then they are represented as nodes in turn: *Substring[26]* represents *Node[1]: {30,2,50}, Substring[40]* represents *Node[2]: {40,15,100}*, *Substring[75]* represents *Node[3]: {30,40,200}*. In addition, the virtual start node *Node[0]: {0,0,0}* and the end node *Node[4]: {76,0,0}* are added for the starting and ending points of the target sequence, respectively, as shown in Fig. 2. After that we need to check the reachability and direction between each node separately, then a directed graph is constructed according to the results, as shown in Fig. 3.

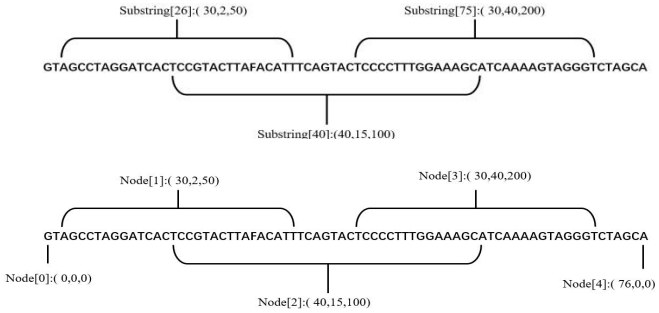

Fig. 2 Construction of nodes

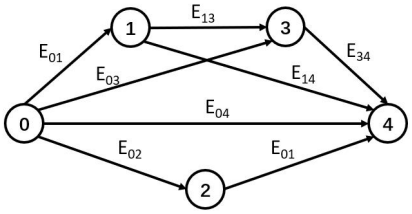

Fig. 3 Construction of the directed graph

Finally, we calculate the cost of each path from node 0 to node 4. The calculation results of all paths are shown in Table 1. After calculation, the path I through node 0-1-3-4 is the path with the lowest cost (*H[I]=24*) of all paths. Returns the index values of all nodes on this path: 0, 1, 3, 4. Delete virtual index 0 and 4, keep 1 and 3, and then find the corresponding common substrings *Substring[26] and Substring[75]* according to the index value: These two common substrings are the best matched substrings for subsequent compression.

Table 1 .The cost（H）of all paths

Path	I	II	III	IV	V
Nodes	0-1-3-4	0-1-4	0-2-4	0-3-4	0-4
H	29	40	36	56	78

2.5 Encoding and compression

After obtaining the matching substrings, the relevant information of the substrings is used to encode the corresponding matching region in the target sequence to obtain an intermediate file, finally the BSC (http://libbsc.com） compressor is used to compress the intermediate files to obtain the final compressed file. Compressed files can be used for transmission, saving, or decoding through reverse operations to restore the original data.

3. Result

In our experiment , eight human genome data sets are selected as the test, including Hg17, Hg18, Hg19, Hg39, KO131, KO224, YH, and HuRef. First, we use two commonly used human genes KO131 and KO224 as mutual references, compress each other to obtain intermediate files, and compare them with LMSRGC algorithm. The number of characters in the intermediate files obtained by each algorithm is as Table 2:

Table 2. Number of characters in the intermediate files obtained by different algorithms

Algorithm	LMSRGC	OptiM
KO131-KO224	4,512,454	3,879,968
KO224-KO131	4,649,476	4,648,070

From the two sets of data in the Table 2, we can see that compared with LMSRGC algorithm, the size of intermediate file obtained by OptimM algorithm is smaller, which means that the number of characters required after encoding the target sequence with the best matching substrings selected by the matching strategy of OptimM algorithm is reduced. The smaller storage occupation of intermediate files makes the subsequent compressed files smaller, which is also proved by the subsequent results.

The experiment is divided into eight groups. Each group uses one genome dataset from the above eight datasets as the reference genome, and the remaining seven datasets are compressed as the target genome. The corresponding seven compression results are accumulated as the final compression results of the group. The compression results obtained by OptimM algorithm are compared with HiRGC, SCCG and LMSRGC algorithms. The comparison results are shown in

Table 3. Comparison of compression results of different algorithms

Ref	Tar	HiRGC	SCCG	LMSRGC	OptiM
hg17	20685	107.7	92.5	63.1	62.5
hg18	20681	102.5	85.6	57.6	56.7
hg19	20666	100.5	85.6	56.7	55.6
hg38	20674	100.3	85.6	57.2	56.7
KO131	20691	130.0	113.8	100.5	99.9
KO224	20691	130.8	114.6	100.5	99.6
HuRef	20965	263.5	243.7	200.6	200.7
YH	20691	135.1	119.1	104.6	104.6

Table 3.

In Table 3, the tar column represents the sum of the data sizes of the seven target genomes, and the file size is in megabytes (MB). The results in bold represent the best compression results in the experiment. From the results, it can be seen that in the eight experiments, the proposed OptiM algorithm produces the smallest compressed file for groups with Hg17, Hg18, Hg19, Hg39, KO131 and KO224 as the reference genomes. This shows that the matching strategy proposed by OptiM is effective and can actually improve the compression performance.

The above results show that the matching strategy proposed by OptiM can choose the optimal matching substrings which represent the target sequence with fewer characters, and the generated intermediate files take up less storage space, thereby OptiM improve the final compression performance.

4. Disscussion

The experimental results show that the compression strategy used by OptiM algorithm improves the final compression performance and is competitive with the-state-of -the-art algorithms.

However, it is worth mentioning that the compression time of OptiM algorithm is not ideal. The main reason is that the traditional shortest path algorithm is used to filter the best match instead of an optimized algorithm. We plan to optimize the shortest path algorithm in the future, reduce the running time, and increase the number of candidate common substrings to obtain better results.

References

[1] Zhang X, Meyerson M. Illuminating the noncoding genome in cancer[J]. Nature Cancer, 2020, 1(9): 864-872.

[2] Wandelt S, Rheinländer A, Bux M, et al. Data management challenges in next generation sequencing[J]. Datenbank-Spektrum, 2012, 12(3): 161-171.

[3] Lesk A M. Introduction to genomics[M]. Oxford University Press, 2017.

[4] Liu Y S, Peng H, Wong L S, et al. High-speed and high-ratio referential genome compression[J]. Bioinformatics, 2017, 33(21): 3364-3372.

[5] Shi W, Chen J H, Luo M, et al. High efficiency referential genome compression algorithm[J]. Bioinformatics, 2019, 35(12): 2058-2065

[6] Lu, Z., Guo, L., Chen, J. et al. Reference-based genome compression using the longest matched substrings with parallelization consideration. BMC Bioinformatics 24, 369 (2023).

[7] Gupta R, Mittal A, Gupta S. An efficient algorithm to detect palindromes in DNA sequences using periodicity transform[J]. Signal Processing, 2006, 86(8): 2067-2073.

[8] Büren F, Jünger D, Kobus R, Hundt C, Schmidt B (2019) Suffix array construction on multi-GPU systems. In: The 28th international symposium.