Solving the Minimum Common String Partition Problem with the Help of Ants

S.M. Ferdous and M. Sohel Rahman*

AℓEDA Group, Department of CSE, BUET, Dhaka-1000, Bangladesh

Abstract. In this paper, we consider the problem of finding minimum common partition of two strings (MCSP). The problem has its application in genome comparison. As it is an NP-hard, discrete combinatorial optimization problem, we employ a metaheuristic technique, namely, MAX-MIN ant system to solve this. The preliminary experimental results are found to be promising.

Keywords: Ant Colony Optimization, Stringology, Genome sequencing, Combinatorial Optimization, Swarm Intelligence, String partition.

1 Introduction

String comparison is one of the important problems in Computer Science with diverse applications in different areas including genome sequencing, text processing and compressions. In this paper, we address the problem of finding minimum common partition (MCSP) of two strings. MCSP is closely related to genome arrangement which is an important field in computational biology. More detailed study of the application of MCSP can be found at [5], [6] and [8].

In MCSP problem, we are given two *related* strings (X,Y). Two strings are related if every letter appears the same number of times in each of them. Clearly, two strings have a common partition if and only if they are related. So, the length of the two strings are also the same (say, n). A partition of a string X is a sequence $P = (B_1, B_2, \cdots, B_c)$ of strings whose concatenation is equal to X, that is $B_1 B_2 \cdots B_c = X$. The strings B_i are called the blocks of P. Given a partition P of a string X and a partition Q of a string Y, we say that the pair $\pi = < P, Q >$ is a common partition of X and Y if Q is a permutation of P. The minimum common string partition problem is to find a common partition of X, Y with the minimum number of blocks, that is to minimize c. For example, if (X,Y) = {“ababcab”,“abcabab”}, then one of the minimum common partition sets is π ={“ab”,“abc”,“ab”} and the minimum common partition size is 3. The restricted version of MCSP where each letter occurs at most k times in each input string, is denoted by k-MCSP.

1.1 Related Works

In [9], the authors investigated k-MCSP along with two other variants: $MCSP^c$, where the alphabet size is at most c; and x-balanced MCSP, which requires that the length of

* Partially supported by a Commonwealth Fellowship and an ACU Titular Award.

the blocks must be witnin the range $(n/d-x, n/d+x)$, where d is the number of blocks in the optimal common partition and x is a constant integer. They showed that $MCSP^c$ is NP-hard when $c \geq 2$. As for k-MCSP, they presented an FPT algorithm which runs in $O^*((d!)^{2k})$ time.

Chrobak et al. [8] analyzed a natural greedy heuristic for MCSP: iteratively, at each step, it extracts a longest common substring from the input strings. They showed that for 2-MCSP, the approximation ratio (for the greedy heuristic) is exactly 3, for 4-MCSP, $\log n$ and for the general MCSP, between $\Omega(n^{0.43})$ and $O(n^{0.67})$.

In this paper, we apply an Ant Colony Optimization (ACO) algorithm to solve the MCSP problem. We conduct experiments on both random and real data to compare our algorithm with the state of the art algorithm in the literature and achieve promising results.

2 Preliminaries

In this section, we present some defitnitions and notations that are used throughout the paper. Two strings (X,Y), each of length n, over an alphabet $\sum$ are called *related* if every letter appears the same number of times in each of them. A block $B = ([id,i,j])$, $0 \leq i \leq j < n$, of a string S is a data structure having three fields: id is an identifier of S and the starting and ending positions of the block in S are represented by i and j, respectively. Naturally, the *length* of a block $[id,i,j]$ is $(j-i+1)$. We use $substring([id,i,j])$ to denote a substring of S induced by the block $[id,i,j]$. Throughout the paper we will use 0 and 1 as the identifiers of X and Y respectively. We use [] to denote the empty block.

For example, if we have two strings (X,Y) = {“abcdab”,“bcdaba”}, then $[0,0,1]$ and $[0,4,5]$ both represent the substring “ab” of X. In other words, $substring([0,0,1]) = substring([0,4,5]) =$ “ab”.

Two blocks can be intersected or unioned. The intersection of two blocks is a block that contains the common portion of the two. Formally, the intersection operation of B_1=$[id,i,j]$ and B_2=$[id,i',j']$ is defined as follows:

$$B_1 \cap B_2 = \begin{cases} [] & \text{if } i' > j \text{ or } i > j' \\ [id,i',j] & \text{if } i' \leq j \\ [id,i,j'] & \text{else} \end{cases} \tag{1}$$

Union of two blocks is either another block or an ordered (based on the starting position) set of blocks. Without the loss of generality we suppose that, $i <= i'$ for B_1=$[id,i,j]$ and B_2=$[id,i',j']$. Then, formally the union operation of B_1 and B_2 is defined as follows:

$$B_1 \cup B_2 = \begin{cases} [id,i,j] & \text{if } j' <= j \\ [id,i,j'] & \text{if } j' > j \text{ or } i' == j+1 \\ \{B_1, B_2\} & \text{else} \end{cases} \tag{2}$$

The union rule with an ordered set of blocks, B_{lst} and a block, B' can be defined as follows. We have to find the position where B' can be placed in B_{lst}, i.e., we have to find $B_k \in B_{lst}$ after which B' can be placed. Then, we have to replace the ordered subset $\{B_k, B_{k+1}\}$ with $B_k \cup B' \cup B_{k+1}$. As an example, suppose we have three blocks, namely, $B_1 = [0,5,7]$,$B_2 = [0,11,12]$ and $B_3 = [0,8,10]$. Then $B_1 \cup B_2 = B'_{lst} = \{[0,5,7], [0,11,12]\}$. On the other hand, $B'_{lst} \cup B_3 = [0,5,12]$, which is basically identical to $B_1 \cup B_2 \cup B_3$.

Two blocks B_1 and B_2 (in the same string or in two different strings) matches if $substring(B_1) = substring(B_2)$. If the two matched blocks are in two different strings then the the matched substring is called a common substring of the two strings denoted by *cstring*(B_1, B_2).

The span of a block, $B = [id,i,j]$, denoted by, $span(B)$ is the length of the maximum block that contains B. More formally, $span(B) = \max\{\ell \mid \ell = length(B'), B \subseteq B', \forall B'\}$. For example, if three blocks B_1, B_2 and B_3 are respectively $[0,0,0]$, $[0,0,1]$ and $[0,0,2]$, then $span(B_1) = span(B_2) = span(B_3) = 2$.

3 Our Approach: Max Min Ant System on the Common Substring Graph

3.1 Formulation of *Common Substring Graph*

We define a common substring graph, $G_{cs}(V,E,id(X))$ of two strings (X,Y) as follows. Here V is the vertex set of the graph and E is the edge set. Vertices are the positions of string X, i.e., for each $v \in V$, $v \in [0, |X|-1]$. Two vertices $v_i \leq v_j$ are connected with and edge, i.e, $(v_i, v_j) \in E$, if the substring induced by the block $[id(X), v_i, v_j]$ matches some substring of Y. More formally, we have:

$$(v_i, v_j) \in E \Leftrightarrow cstring([id(X), v_i, v_j], B') \textit{ is not empty } \; \exists B' \in Y$$

In other words, each edge in the edge set corresponds to a *block* satisfying the above condition. For convenience, we will denote the edges as *edge blocks* and use the list of edge blocks (instead of edges) to define the edgeset E. Notably, each *edge block* on the edge set of $G_{cs}(V,E,id(X))$ of string (X,Y) may match with more than one blocks of Y. For each *edge block B* a list is maintained containing all the matched blocks of string Y to that *edge block*. This list is called the $matchList(B)$.

For example, suppose (X,Y) = {"abcdba","abcdab"}. Now consider the corresponding common substring graph. Then, we have vertex set, $V = \{0,1,2,3,4,5\}$ and edge set, $E = \{[0,0,0],[0,0,1],[0,1,1],[0,2,2],[0,2,3],[0,3,3],[0,4,4],[0,5,5]\}$. The *matchList* of the second *edge block*, i.e., $matchList([0,0,1]) = \{[1,0,1],[1,4,5]\}$.

To find a common partition of two strings (X,Y) we first construct the common substring graph of (X,Y). Then from a vertex v_i on the graph we take an edge block $[id(X), v_i, v_j]$. Suppose M_i is the *matchList* of this block. We take a block B'_i from M_i. Then we advance to the next vertex that is $v_j + 1$ *MOD* $|X|$ and choose another correspoding edge block as before. We continue this until we come back to the starting vertex. Let *partitionList* and *mappedList* are two lists, each of length c, containing the traversed edge blocks and the corresponding matched blocks. Now we have the following lemma.

Lemma 1. *partitionList is a common partition of length c if the blocks of mappedList obeys,*

$$B_i \cap B_j = [] \; \forall B_i, B_j \in mappedList, \; i \neq j \tag{3}$$

and

$$B_1 \cup B_2 \cup \cdots \cup B_c = [id(Y), 0, |Y| - 1] \tag{4}$$

3.2 Heuristics

Heuristics (η) contain the problem specific information. We propose two different (types of) heuristics for MCSP. Firstly, we propose a static heuristic that does not change during the runs of algorithm. The other heuristic we propose is dynamic in the sense that it changes between the runs.

The Static Heuristic for MCSP. We employ a very naive and intuitive idea. It is obvious that the larger is the size of the blocks the smaller is the partition set. To capture this phenomenon, we assign on each edge of the common substring graph a numerical value that is proportional to the length of the substring corresponding to the edge block. Formally, the static heuristic (η_s) of an edge block $[id,i,j]$ is defined as follows:

$$\eta_s([id,i,j]) \propto length([id,i,j]) \tag{5}$$

The Dynamic Heuristic for MCSP. We observe that the static heuristic can sometimes lead us to very bad solutions. For example if $(X,Y) = \{$“bceabcd”,“abcdbec”$\}$ then according to the static heuristic much higher value will be assigned to *edge block* $[0,0,1]$ rather than to $[0,0,0]$. But if we take $[0,0,1]$, we must match it to the block $[1,1,2]$ and we further miss the opportunity to take $[0,3,6]$ later. The resultant partition will be $\{$“bc”,“e”,“a”,“b”,“c”,“d”$\}$ but if we would take $[0,0,0]$ at first step, then one of the resultant partitions would be $\{$“b”,“c”,“e”,“abcd”$\}$. To overcome this shortcoming of the static heuristic we define a dynamic heuristic as follows. The dynamic heuristic (η_d) of an edge block ($B = [id,i,j]$) is inversely proportional to the difference between the length of the block and the minimum span of its correspoding blocks in *matchList*. More formally, $\eta_d(B)$ is defined as follows:

$$\eta_d(B) \propto \frac{1}{|length(B) - minSpan(B)| + 1}, \tag{6}$$

where

$$minSpan(B) = \min\{span(B') \mid B' \in matchList(B)\} \tag{7}$$

In the example, $minSpan([0,0,0])$ is 1 as follows: $matchList([0,0,0]) = \{[1,1,1], [1,4,4]\}$. $span([1,1,1]) = 4$ and $span([1,4,4] = 1)$. On the other hand, $minSpan([0,0,1])$ is 4. So, according to dynamic heuristic much higher numeral will be assigned to block $[0,0,0]$ rather than block $[0,0,1]$.

We define the total heuristic (η) is the linear combination of the static heuristic (η_s) and the dynamic heuristic (η_d). Formally, the total heuristic of an edge block B is, $\eta(B) = a \cdot \eta_s(B) + b \cdot \eta_d(B)$, where a, b are any real valued constant.

3.3 Initialization and Configuration

Given two strings (X,Y), we first construct the common substring graph $G_{cs} = (V,E,id(X))$. We use the following notations. *Local best solution* (L_{LB}) is the best solution found in each iteration. *Global best solution* (L_{GB}) is the best solution found so far among all iterations. The pheromone of the edge block is bounded between τ_{max} and τ_{min}. Like [3], we use the following values for τ_{max} and τ_{min}: $\tau_{max} = \frac{1}{\varepsilon \cdot cost(L_{GB})}$, and $\tau_{min} = \frac{\tau_{max}(1-\sqrt[n]{p_{best}})}{(avg-1)\sqrt[n]{p_{best}}}$. Here, *avg* is the average number of choices an ant has in the construction phase. Initially, the pheromone values of all edge blocks (substring) are initialized to *initPheromone* which is a large value to favor the exploration at the first iteration [3].

3.4 Construction of a Solution

Let, *nAnts* denotes the total number of ants in the colony. Each ant is deployed randomly to a vertex v_s of the G_{cs}. A solution for an ant starting at a vertex v_s is constructed by the following steps.

step 1: Let $v_i = v_s$. Choose an *available* edge block starting from v_i by the discrete probability distribution defined below. An edge block is available if its *MatchList* is not empty and inclusion of it to the *partitionList* and *mappedList* obeys Equations 3. The probability for choosing edge block $[0,v_i,v_j]$ is:

$$p([0,v_i,v_j]) = \frac{\tau([0,v_i,v_j])^{\alpha} \cdot \eta([0,v_i,v_j])^{\beta}}{\sum_{\ell} \tau([0,v_i,v_{\ell}])^{\alpha} \cdot \eta([0,v_i,v_{\ell}])^{\beta}}, \forall \ell \textit{ such that } [0,v_i,v_l] \textit{ is an available block.} \quad (8)$$

step 2: Suppose, $[0,v_i,v_k]$ is chosen according to Equation 8 above. We choose a match block B_m from the *matchList* of $[0,v_i,v_k]$ and delete B_m from the *matchList*. We also delete every block from every *matchList* of every edge block that overlaps with B_m. Formally we delete a block B if

$$B_m \cap B \neq [] \quad \forall B_i \in E, B \in matchList(B_i)$$

We add $[0,v_i,v_k]$ to the *partitionList* and B_m to the *mappedList*.

step 3: If (v_k+1) *MOD* $|X| = v_s$ and the *mappedList* obeys 4, then we have found a common partition of X and Y. The size of the partition is the length of the *partitionList*. Otherwise, we jump to the *step 1*.

3.5 Pheromone Update

When each of the ants in the colony construct a solution (i.e., a common partition), an iteration completes. We set the local best solution as the best partition that is the minimum length partition in an iteration. The global best solution for n iterations is defined as the minimum length common partition over first n iteration.

We define the fitness $F(L)$ of a solution L as the reciprocal of the length of L. The pheromone of each interval of each target string is computed according to:

$$\tau_i \leftarrow (1-\varepsilon) \cdot \tau_i + \tau_i \cdot \sum_{s \in G_{iter} | c_i \in s} F(s) \cdot \varepsilon, i = 1,2,...,n \quad (9)$$

The pheromone are bounded within the range τ_{MIN} and τ_{MAX}. We have updated the pheromone values according to L_{LB} or L_{GB}.

4 Experiments

We have conducted our experiments in a computer with Intel Core 2 Quad CPU 2.33 GHz. The available RAM was 4.00 GB. The operating system was Windows 7. The programming environment was java. The maximum allowed time for each instance was 120 minutes.

4.1 Dataset

We have taken two types of data into consideration: randomly generated DNA sequence and real gene sequence.

Random DNA Sequence: We have generated 30 DNA sequences of length at most 600 randomly using [10]. The fraction of bases A, T, G and C is assumed to be 0.25 each. For each DNA sequence we shuffle it to create a new DNA sequence. The shuffling is done using the online toolbox [11]. The original random DNA sequence and its shuffled pair constitute a single input (X,Y) in our experiment. This dataset is divided into 3 classes. The first 10 have length less than or equal 200 bps (base-pairs), the next 10 have length within $[201,400]$ and the rest 10 have length within $[401,600]$ bps.

Real Gene Sequence: We collected the gene sequence data from the NCBI GenBank[1]. For simulation we have taken Bacterial Sequencing (part 14). We have taken the first 15 gene sequences whose lengths are within $[200,600]$.

4.2 Parameters

The settings of parameters for which we achieved the results is described in Table 1.

Table 1. Parameters

Parameters	Value
α	2.0
β	5.0
Evaporation rate, ε	0.02
$nAnts$	$\lvert X \rvert$
p_{best}	0.09
$initPheromone$	10.0
Maximum Allowed Time	120 min
Coeff. of η_s, a	0.5
Coeff. of η_d, b	0.5

[1] `http://www.ncbi.nlm.nih.gov`

4.3 Results and Analysis

We have compared our approach with the greedy algorithm of [8] because none of the other algorithms in the literature are for general MCSP: each of the other approximation algorithms put some restrictions on the parameters.

Random DNA Sequence: Table 2 presents the comparison between our approach and the greedy approach [8] for the random DNA sequences. For a particular DNA sequence, the experiment was run 4 times and the average result is reported. The first column under any group reports the partition size computed by the greedy approach, the second column is the average partition size found by MAX-MIN and the third column represents the difference between the two approaches. A positive (negative) difference indicates that the greedy result is better (worse) than the MAX-MIN result by that amount. From the table, we can see that out of 30 instances our approach gets better partition size for 28 cases.

Table 2. Comparison between Greedy approach [8] and MAX-MIN on random DNA sequences

Test No.	Group 1 (200 bps)			Group 2 (400 bps)			Group 3 (600 bps)		
	Greedy	**MAX-MIN**	**Difference**	**Greedy**	**MAX-MIN**	**Difference**	**Greedy**	**MAX-MIN**	**Difference**
1.	46	42.75	-3.25	119	114.25	-4.75	182	180.00	-2.00
2.	56	51.50	-4.50	122	119.00	-3.00	175	176.25	1.25
3.	62	56.75	-5.25	114	112.25	-1.75	196	188.00	-8.00
4.	46	43.00	-3.00	116	116.25	0.25	192	184.25	-7.75
5.	44	43.00	-1.00	135	132.25	-2.75	176	171.75	-4.25
6.	48	42.25	-5.75	108	105.5	-2.50	170	163.25	-6.75
7.	65	60.00	-5.00	108	99.00	-9.00	173	168.50	-4.50
8.	51	47.00	-4.00	123	118.00	-5.00	185	176.25	-8.75
9.	46	45.75	-0.25	124	119.50	-4.50	174	172.75	-1.25
10.	63	59.25	-3.75	105	101.75	-3.25	171	167.25	-3.75

Table 3. Comparison between Greedy approach [8] and MAX-MIN on real gene sequence

Test No.	Greedy	MAX-MIN	Difference
1.	95.0000	87.7500	-7.2500
2.	161.0000	158.5000	-2.5000
3.	121.0000	116.5000	-4.5000
4.	172.0000	171.7500	-0.2500
5.	153.0000	146.0000	-7.0000
6.	140.0000	140.7500	0.7500
7.	134.0000	131.0000	-3.0000
8.	149.0000	148.5000	-0.5000
9.	151.0000	149.0000	-2.0000
10.	126.0000	124.5000	-1.5000
11.	143.0000	138.2500	-4.7500
12.	180.0000	181.0000	1.0000
13.	152.0000	147.7500	-4.2500
14.	157.0000	161.2500	4.2500
15.	157.0000	158.7500	1.7500

Real Gene Sequence: Table 3 shows the minimum common partition size found by our approach and the greedy approach for the real gene sequences. Out of the 15 instances we get better results on 11 instances.

5 Conclusion

Minimum Common String Partition problem has important applications in computational biology. In this paper, we have described a metaheuristic approach to solve the problem. We have used static and dynamic heuristic information in this approach. Simulating this algorithm on long DNA sequences would be challenging future improvement.

References

1. Dorigo, M., Di Caro, G., Gambardella, M.L.: Ant algorithms for discrete optimization. J. of Artificial Life 5(2), 137–172 (1999)
2. Dorigo, M., Gambardella, M.L.: Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. J. of IEEE Transactions on Evolutionary Computation 1 (1996)
3. Stuetzle, T., Hoos, H.H.: MAX-MIN Ant System. J. of Future Gener. Comput. System 16(9), 889–914 (2000)
4. Watterson, G.A., Ewens, W.J., Hall, T.E., Morgan, A.: The chromosome inversion problem. Journal of Theoretical Biology 99, 7 (1982)
5. Goldstein, A., Kolman, P., Zheng, J.: Minimum common string partition problem: Hardness and approximations. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 484–495. Springer, Heidelberg (2004)
6. Chen, X., Zheng, J., Fu, Z., Nan, P., Zhong, Y., Lonardi, S., Jiang, T.: Assignment of orthologous genes via genome rearrangement (2004) (submitted)
7. Damaschke, P.: Minimum Common String Partition Parameterized. In: Crandall, K.A., Lagergren, J. (eds.) WABI 2008. LNCS (LNBI), vol. 5251, pp. 87–98. Springer, Heidelberg (2008)
8. Chrobak, M., Kolman, P., Sgall, J.: The greedy algorithm for the minimum common string partition problem. In: Jansen, K., Khanna, S., Rolim, J.D.P., Ron, D. (eds.) APPROX and RANDOM 2004. LNCS, vol. 3122, pp. 84–95. Springer, Heidelberg (2004)
9. Jiang, H., Zhu, B., Zhu, D., Zhu, H.: Minimum common string partition revisited. In: Lee, D.-T., Chen, D.Z., Ying, S. (eds.) FAW 2010. LNCS, vol. 6213, pp. 45–52. Springer, Heidelberg (2010)
10. Stothard, P.: The Sequence Manipulation Suite: JavaScript programs for analyzing and formatting protein and DNA sequences. Biotechniques 28, 1102–1104 (2000)
11. Villesen, P.: FaBox: an online fasta sequence toolbox (2007), `http://www.birc.au.dk/software/fabox`