Math.Comput.Sci. (2017) 11:233–249
DOI 10.1007/s11786-017-0293-5

Mathematics in Computer Science

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

S. M. Ferdous · M. Sohel Rahman

Received: 22 April 2014 / Revised: 14 March 2016 / Accepted: 10 June 2016 / Published online: 24 February 2017

Abstract In this paper, we consider the problem of finding a minimum common partition of two strings. 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 problem. To achieve better efficiency we first map the problem instance into a special kind of graph. Subsequently, we employ a MAX–MIN ant system to achieve high quality solutions for the problem. Experimental results show the superiority of our algorithm in comparison with the state of art algorithms in the literature. The improvement achieved is also justified by a standard statistical test.

Keywords Ant Colony Optimization · Metaheuristics · Graph Algorithms · Computational Biology

Mathematics Subject Classification 68

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 data compression. In this paper, we address the problem of finding a minimum common partition (MCSP) of two strings. MCSP is closely related to genome arrangement which is an important topic in Computational Biology. Given two DNA sequences, the MCSP asks for the smallest set of the common building blocks of the sequences.

Our goal is to partition each string into c segments called *blocks*, so that the *blocks* in the partition of X and that of Y constitute the same multiset of substrings. The cardinality of the partition set, i.e., c is to be minimized. Formally, a partition of a string X is a sequence $P = (B_1, B_2, \ldots, B_m)$ of strings whose concatenation is equal to X, that is $B_1 B_2 \ldots B_m = X$. 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 $\pi = \langle P, Q \rangle$ is a common partition of X and Y if Q is a permutation of P. The minimum common string partition problem, i.e., MCSP problem, is to find a common partition of X, Y

S. M. Ferdous · M. S. Rahman (✉)
AℓEDA Group, Department of CSE, BUET, Dhaka 1000, Bangladesh
e-mail: msrahman@cse.buet.ac.bd

S. M. Ferdous
e-mail: ferdous.csebuet@gmail.com

with the minimum number of blocks. 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. Note that, in the MCSP problem, we consider 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 lengths of the two strings are also the same (say, n).

MCSP has applications in Comparative Genomics. Given two DNA strings, MCSP answers the possibilities of re-arrangement of one DNA string to another [7]. MCSP is also important in ortholog assignment. In [5], Chen et al. present a new approach to ortholog assignment that takes into account both sequence similarity and evolutionary events at a genomic level. In that approach, first, the problem is formulated as that of computing the signed reversal distance with duplicates between the two genomes of interest. Then, the problem is decomposed into two optimization problems, namely minimum common partition and maximum cycle decomposition problem. Thus MCSP plays an integral part in computing ortholog assignment of genes.

1.1 Our Contribution

In this paper, we consider metaheuristic approaches to solve the problem. Particularly we are interested in nature inspired algorithms. As the problem is discrete combinatorial optimization problem, the natural choice is Ant Colony Optimization (ACO). Before applying ACO, it is necessary to map the problem into a graph. We have developed this mapping. In this paper, we implement a variant of ACO algorithm namely MAX–MIN Ant System (MMAS) to solve the MCSP problem. We conduct experiments on both random and real data to compare our algorithm with the state of the art algorithms in the literature and achieve excellent results. Notably, a preliminary version of the paper appeared at [16].

2 Literature Review

MCSP is essentially the breakpoint distance problem [27] between two permutations which is to count the number of ordered pairs of symbols that are adjacent in the first string but not in the other; this problem is obviously solvable in polynomial time [19]. The 2-MCSP is proved to be NP-hard and moreover APX-hard in [19]. The authors in [19] also presented several approximation algorithms. Chen et al. [5] studied the problem, Signed Reversal Distance with Duplicates (SRDD), which is a generalization of MCSP. They gave a 1.5-approximation algorithm for 2-MCSP. In [7], the author analyzed the fixed-parameter tractability of MCSP considering different parameters. In [20], 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 the blocks must be within 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. [6] 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. They also proved that for 4-MCSP the ratio would be $\log n$ and for the general MCSP, between $\Omega(n^{0.43})$ and $O(n^{0.67})$.

Ant Colony Optimization (ACO) [10,11,13] was introduced by Dorigo and colleagues as a nature-inspired metaheuristic for the solution of hard Combinatorial Optimization (CO) problems. The inspiring source of ACO is the pheromone trail laying and following behavior of real ants which use pheromones as a communication medium. In analogy to the biological example, ACO is based on the indirect communication of a colony of simple agents, called (artificial) ants, mediated by (artificial) pheromone trails. The pheromone trails in ACO serve as a distributed, numerical information which the ants use to probabilistically construct solutions to the problem being solved and which the ants adapt during the algorithm's execution to reflect their search experience.

Different ACO algorithms have been proposed in the literature. The original algorithm is known as the Ant System (AS) [8,9,12]. The other variants are Elitist AS [8,12], ANT-Q [18], Ant Colony System (ACS) [11], MAX–MIN AS [23–25] etc.

Recently growing interest has been noticed towards ACO in the scientific community. There are now available several successful implementations of the ACO metaheuristic applied to a number of different discrete combinatorial optimization problems. In [10] the authors distinguished among two classes of applications of ACO: those to static combinatorial optimization problems, and those to the dynamic ones. Static combinatorial optimization problems are those in which the problem does not change while it is being solved. The authors list some static combinatorial optimization problems that are successfully solved by different variants of ACO. Some of the problems are, travelling salesperson, Quadratic Assignment, job-shop scheduling, vehicle routing, sequential ordering, graph coloring, etc. Dynamic problems are defined as a function of some quantities whose values are set by the dynamics of an underlying system. The problem changes therefore at run time and the optimization algorithm must be capable of adapting online to the changing environment. The authors listed connection-oriented network routing and connectionless network routing as the examples of dynamic problems those are successfully solved by ACO.

The survey [14] presents a non-exhaustive list of applications of ACO algorithms grouped by problem types. The authors categorized the problems into different types namely routing, assignment, scheduling, subset machine learning and bioinformatics. In each type they listed the problems that are successfully solved by some variants of ACO.

There are not too many string related problems solved by ACO in the literature. In [4], the authors addressed the reconstruction of DNA sequences from DNA fragments by ACO. Several ACO algorithms have been proposed for the longest common subsequence (LCS) problem in [1,21]. Recently minimum string cover problem is solved by ACO in [15]. Finally, we note that to the best of our knowledge, there were no attempt to solve this problem with metaheuristic approaches in the literature when this research work was conducted. However, during the rather long submission process of the current manuscript, a few works (e.g., [2,3]) have been reported in the literature inspired by our preliminary work presented at [16]. A follow up work of the current authors have also been published in [17] during this period.

3 Preliminaries

In this section, we present some definitions and notations that are used throughout the paper.

Definition 1 *Related string*: 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.

Example 1 X = "abacbd" and Y = "acbbad", then they are *related*. But if X_1 = "aeacbd" and Y = "acbbad", they are not *related*.

Definition 2 *Block*: 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 the substring of S induced by the block $[id, i, j]$. Throughout we will use 0 and 1 as the identifiers of X(i.e., $id(X)$) and Y(i.e., $id(Y)$) respectively. We use [] to denote an empty block.

Example 2 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 (with same ids) is a block that contains the common portion of the two.

Definition 3 *Intersection of blocks*: Formally, the intersection operation of $B_1 = [id, i, j]$ and $B_2 = [id, i', j']$ is defined as follows:

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

Example 3 If, $B_1 = [0, 1, 5]$ and $B_2 = [0, 3, 6]$, then $B_1 \cap B_2 = [0, 3, 5]$. On the other hand, if $B_1 = [0, 1, 5]$ and $B_2 = [0, 6, 8]$, then $B_1 \cap B_2 = []$.

Definition 4 *Union of blocks*: The union of two blocks (with the same ids) is either another block or an ordered (based on the starting position) set of blocks. Without loss of generality we suppose that $i \leq i'$ 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:

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

Example 4 If, $B_1 = [0, 1, 5]$ and $B_2 = [0, 3, 6]$, then $B_1 \cup B_2 = [0, 1, 6]$. On the other hand, if $B_1 = [0, 1, 5]$ and $B_2 = [0, 6, 8]$, then $B_1 \cup B_2 = \{[0, 1, 5], [0, 6, 8]\}$.

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 $B_k \cup B' \cup B_{k+1}$.

Example 5 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) match if $substring(B_1) = substring(B_2)$. If the two matched blocks are in two different strings then the matched substring is called a common substring of the two strings denoted by *cstring*(B_1, B_2).

Definition 5 *Span*: Given a list of blocks with the same id, the *span* of a block, $B = [id, i, j]$ in the list denoted by $span(B)$ is the length of the block (also in the list) that contains B and whose length is maximum over all such blocks in the list. Note that a block is assumed to contain itself. More formally, given a list of blocks, $list_b$, $span(B \in list_b) = \max\{\ell \mid \ell = length(B'), B \subseteq B', \forall B' \in list_b\}$.

Example 6 If $list_b = \{[0, 0, 0], [0, 0, 1], [0, 0, 2], [0, 4, 5]\}$ then $span([0, 0, 0]) = span([0, 0, 1]) = span([0, 0, 2]) = 3$ where as, $span([0, 4, 5]) = 2$. In other words, span of a block is the maximum length of the super string than contains the substring induced by the block.

Definition 6 *Partition*: A *partition* of a string X is a list of blocks all with $id(X)$ having the following two properties:

(a) Non overlapping: The blocks must be disjoint, i.e., no block should overlap with another block. So the intersection of any two blocks must be empty.
(b) Cover: The blocks must cover the whole string.

In other words, a *partition* of a string X is a sequence $P = (B_1, B_2, \ldots, B_m)$ of strings whose concatenation is equal to X, that is $B_1 B_2 \ldots B_m = X$, where B_i's are *blocks*.

3.1 Basics of ACO

In ACO, a Combinatorial Optimization (CO) problem is solved by iterating the following two steps. At first, solutions are constructed using a parameterized probability distribution over the solution space which is called the pheromone model. The second step is to modify the pheromone values using the solutions that were constructed in earlier iterations in a way that is deemed to bias the search towards the high quality solutions.

3.2 Ant Based Solutions Construction

The basic ingredient of an ACO algorithm is a constructive heuristic that constructs solutions probabilistically. Sequences of solution components taken from a finite set of solution components $C = \{c_1, c_2, \ldots, c_n\}$ is assembled by a constructive heuristic. Starting with an empty partial solution $s^p = \emptyset$ a solution is constructed. Then at each construction step the current partial solution s^p is extended by adding a feasible solution component from the solution space C. The definition of the feasible solution component is problem specific. Typically a problem is mapped into a construction Graph $G_c = (C, E)$ whose vertices are the solution components C and the set E are the connections (i.e., edges). The process of constructing solutions can be regarded as a walk (or a path) on the construction graph.

3.3 Heuristic Information

In most ACO algorithms the transition probabilities, i.e., the probabilities for choosing the next solution component, are defined as follows:

$$p\left(c_i|s^p\right) = \frac{{\tau_i}^{\alpha} \cdot \eta(c_i)^{\beta}}{\sum_{c_j \in N(s^p)} {\tau_j}^{\alpha} \cdot \eta(c_j)^{\beta}}, \quad \forall c_i \in N(s^p) \tag{3}$$

Here, c_i is a candidate component and s^p is the partial solution. The current partial solution s^p is extended by adding a feasible solution component from the set of feasible neighbors $N(s^p) \subseteq C$, η is a weight function that contains *heuristic information* and α, β are positive parameters whose values determine the relation between the *pheromone information* and the *heuristic information*. The pheromone deployed by the ants are denoted by τ.

3.4 Pheromone Update

The pheromone update consists of two parts. The first part is pheromone evaporation, which uniformly decreases all the pheromone values. From a practical point of view, pheromone evaporation prevents too rapid convergence of the algorithm toward a sub-optimal region. Thus it helps to avoid the local optimal solutions and favors the exploration of new areas in the search space. Then, one or more solutions from the current or from earlier iterations (the set is denoted by S_{upd}) are used to increase the values of pheromone trail parameters on solution components that are part of these solutions:

$$\tau_i \leftarrow (1 - \varepsilon) \times \tau_i + \varepsilon \times \sum_{s \in S_{upd} | c_i \in s} F(s), i = 1, 2, \ldots, n. \tag{4}$$

Let $W(.)$ be the cost function. In Eq. 4, S_{upd} is the set of local best or global best solution, $\varepsilon \in (0, 1]$ is a parameter called the *evaporation rate*, and $F : G \rightarrow \mathbb{R}^+$ is a function such that $W(s) < W(\acute{s}) \Rightarrow F(s) \geq F(\acute{s}), s \neq \acute{s}, \forall s \in G$. The function $F(.)$ is commonly called the *Fitness Function*.

In general, different versions of ACO algorithms differ in the way they update the pheromone values. This also holds for the two currently best-performing ACO variants in practice, namely, the Ant Colony System (ACS) [11] and the MAX–MIN Ant System (MMAS) [25]. Since in our algorithm we hybridize ACS with MMAS, below we give a brief description of MMAS.

3.5 MAX–MIN Ant System (MMAS)

MMAS algorithms are characterized as follows. First, the pheromone values are limited to an interval $[\tau_{MIN}, \tau_{MAX}]$ with $0 < \tau_{MIN} < \tau_{MAX}$. Pheromone trails are initialized to τ_{max} to favor the diversification during the early iterations so that premature convergence is prevented. Explicit limits on the pheromone values ensure that the chance

of finding a global optimum never becomes zero. Second, in case the algorithm detects that the search is too much confined to a certain area in the search space, a restart is performed. This is done by initializing all the pheromone values again. Third, the pheromone update is always performed with either the iteration-best solution, the restart-best solution (i.e., the best solution found since the last restart was performed), or the best-so-far solution.

4 Our Approach: MAX–MIN Ant System on the Common Substring Graph

4.1 Formulation of Common Substring Graph

We define a common substring graph, $G_{cs}(V, E, id(X))$ of a string X with respect to 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 an 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:
$(v_i, v_j) \in E \Leftrightarrow cstring([id(X), v_i, v_j], B')\ is\ not\ empty, \quad \exists B' \in Y$

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 $(X, Y) = \{$"abad","adab"$\}$. Now consider the corresponding common substring graph, $G_{cs}(V, E, id(X))$. Then, we have $V = \{0, 1, 2, 3\}$ and $E = \{[0, 0, 0], [0, 0, 1], [0, 1, 1], [0, 2, 2], [0, 2, 3]\}$. The construction steps are shown in Fig. 1.

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

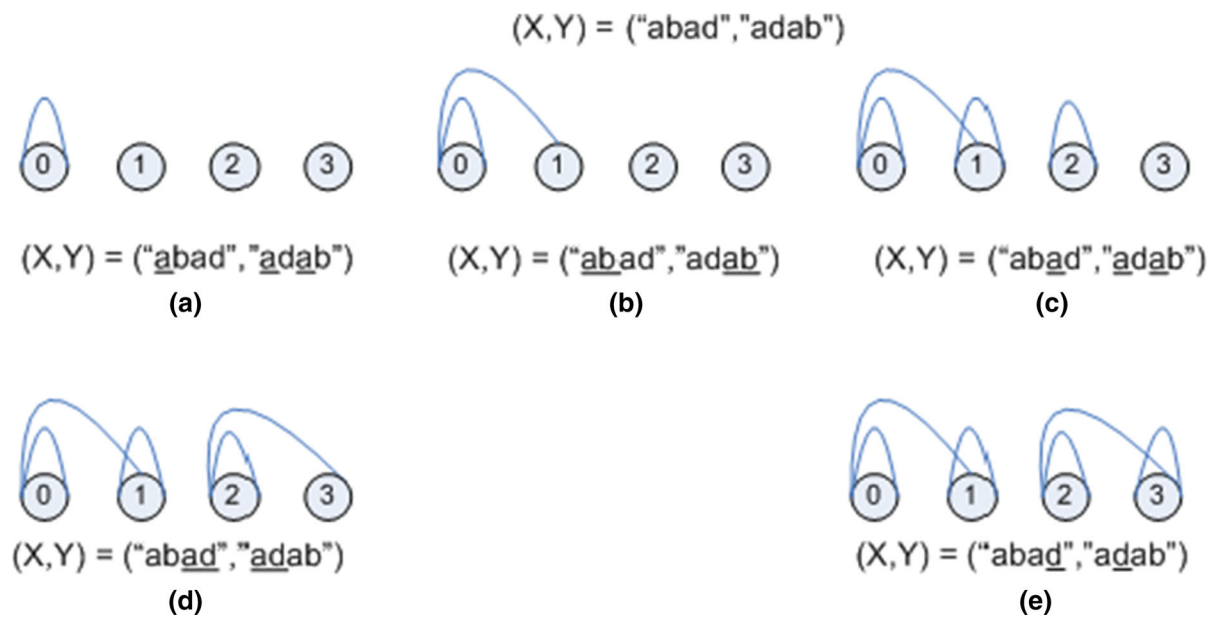

Fig. 1 Construction of $G_{cs}(V, E, id(X))$ of (X, Y). **a** Vertex 0 is connected with itself because "a" is common string of X and Y. **b** An edge exists between vertices 0 and 1 as "ab" is a common string of X and Y. **c** Vertex 1 is connected with itself. **d** Vertex 1 and vertex 2 are connected with vertex 0 and vertex 3, respectively. **e** Vertex 3 is connected with itself

another corresponding edge block as before. We continue this process 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, and only if we have the following:*

$$B_i \cap B_j = [] \quad \forall B_i, B_j \in mappedList,\ i \neq j \tag{5}$$

and

$$B_1 \cup B_2 \cup \cdots \cup B_c = [id(Y), 0, |Y| - 1]. \tag{6}$$

Proof By construction, $partitionList$ is a *partition* of X. We need to prove that $mappedList$ is a partition of Y and with the one to one correspondence between $partitionList$ and $mappedList$ it is obvious that $partitionList$ would be the common partition of (X, Y). Equation 5 asserts the non overlapping property of $mappedList$ and Eq. 6 assures the cover property. So, $mappedList$ will be a partition of Y if Eqs. 5 and 6 are satisfied.

On the other hand, let $partitionList$ along with $mappedList$ be a common partition of (X, Y). According to construction, $partitionList$ satisfies the two properties of a partition. Let, $mappedList$ be a partition of Y. We assume $mappedList$ does not follow the Eqs. 5 or 6. So, there might be overlapping between the blocks or the blocks do not cover the string Y, a contradiction. This completes the proof. □

4.2 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.

4.2.1 The Static Heuristic for MCSP

We employ an 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 to 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:

$$\eta_s([id, i, j]) \propto length([id, i, j]). \tag{7}$$

4.2.2 The Dynamic Heuristic for MCSP

We observe that the static heuristic can sometimes lead us to very bad solutions. For example if (X, Y) = {"bce-abcd","abcdbec"} then according to the static heuristic much higher value will be assigned to *edge block* [0, 0, 1] 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 the 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 corresponding blocks in its $matchList$. More formally, $\eta_d(B)$ is defined as follows:

$$\eta_d(B) \propto \frac{1}{|length(B) - minSpan(B)| + 1}, \tag{8}$$

where

$$minSpan(B) = \min\{span(B') \mid B' \in matchList(B)\}. \tag{9}$$

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 the dynamic heuristic much higher numeral will be assigned to block [0, 0, 0] rather than to block [0, 0, 1].

We define the total heuristic (η) to 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

$$\eta(B) = a \cdot \eta_s(B) + b \cdot \eta_d(B) \tag{10}$$

where a, b are real valued constants. The algorithms of static and dynamic heuristics are shown in Algorithm (1–2)

Algorithm 1 addDynamicHeuristic(G_{cs})

E ← edge blocks of E
for all Block B in E **do**
  minspan ← find minimum free span of B by Eq. 9
  dynamicHeuristic(E) = $\frac{1}{(length(E)-minspan+1)}$
end for

Algorithm 2 addStaticHeuristic(G_{cs})

E ← edge blocks of G_{cs}
max ← maximum length edgeblock of G_{cs}
for all Block B in E **do**
  staticHeuristic(B) = length(B)/max
end for

Algorithm 3 addHeuristic(G_{cs},a,b)

E ← edge blocks of G_{cs}
addStaticHeuristic(G_{cs})
addDynamicHeuristic(G_{cs})
for all Block B in E **do**
  heuristic(B) ← a · staticHeuristic(B) + b · dynamicHeuristic(B)
end for

4.3 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 [25], 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; n is the length of the string; p_{best} is the probability of finding the best solution when the system converges and ε is the evaporation rate. 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 [25]. The steps of the initialization is shown in Algorithm 4.

Algorithm 4 initialize(G_{cs})

```
initialize L_LB
initialize L_GB
set Parameters
E ← edge blocks of G_cs
for all Block B in E do
  pheromone(B) ← initPheromone
end for
```

4.4 Construction of a Solution

Let, $nAnts$ denote the total number of ants in the colony. Each ant is deployed randomly to a vertex v_s of 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 Eq. 11. The probability for choosing edge block $[0, v_i, v_j]$ is:

$$p([0, v_i, v_j]) = \frac{\tau([0, v_i, v_j])^{\alpha} \cdot \eta([0, v_i, v_j])^{\beta}}{\sum_{\ell} \tau([0, v_i, v_\ell])^{\alpha} \cdot \eta([0, v_i, v_\ell])^{\beta}}, \quad \forall \ell \; such \; that [0, v_i, v_l] \; is \; an \; available \; block. \tag{11}$$

Step 2: Suppose, $[0, v_i, v_k]$ is chosen according to Eq. 11 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
$B_m \cap B \neq []$ $\forall B_i \in E, B \in matchList(B_i)$.
We add $[0, v_i, v_k]$ to the $partitionList$ and B_m to the $mappedList$.

Step 3: If $(v_k + 1) \; MOD \; length(X) = v_s$ and the $mappedList$ obeys Eq. 6, 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*.

The construction is shown in Algorithm 5.

4.5 Intelligent Positioning

For every edge block of G_{cs} in X, we have a $matchList$ that contains the matched block of string Y. In construction (step 1), when an edge block is chosen by the probability distribution, we take a block from the $matchList$ of the chosen edge block. We can choose the matched block randomly. But we observe that random choosing may lead to a very bad partition. For example, if $(X, Y) = \{$"ababc","abcab"$\}$ then the $matchList([0, 0, 1]) = \{[1, 0, 1], [1, 3, 4]\}$. If we choose the first match block then eventually we will get the partition as {"ab","ab","c"} but a smaller partition exists and that is {"ab","abc"}.

To overcome this problem, we have imposed a rule for choosing the matched block. We will select a block from the $matchList$ having the lowest possible span. Formally, for the edge block, B_i, a block $B' \in matchList(B_i)$ will be selected such that $span(B')$ is the minimum.

Algorithm 5 constructSolution(i,G_{cs})

```
blockList = empty list of blocks
mappedList = empty list of blocks
startpos = ⌊n/m⌋ ∗ i
k = startpos
repeat
  addHeuristics(G_cs,a,b)
  constructPDF(k,G_cs) using Eq. 11
  B = choose an edge block from PDF
  M = choose a match block from matchList(B) {I}ntelligent Positioning
  Update matchList(B)
  add B to blockList
  add M to the mappedList
  k = B.j + 1
until k ≠ startpos
```

In our example $span([1, 0, 1]) = 3$ where as $span([1, 3, 4]) = 2$. So it is better to select the second block so that we do not miss the opportunity to match a larger block.

4.6 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 all the n iterations.

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 Eq. 4 after each iteration. The pheromone values are bounded within the range τ_{MIN} and τ_{MAX}. We update the pheromone values according to L_{LB} or L_{GB}. Initially for the first 50 iterations we update pheromone by only L_{LB} to favor the search exploration. After that we develop a scheduling where the frequency of updating with L_{LB} decreases and L_{GB} increases to facilitate exploitation. The pheromone update algorithm is listed in Algorithm 6.

4.7 The Pseudocode

The pseudocode of our approach for solving MCSP is given in Algorithm 7.

5 Experiments

We have conducted our experiments in a computer with Intel Core 2 Quad CPU 2.33 GHz. The available RAM was 4 GB. The operating system was Windows 7. The programming environment was Java. jre version is “1.7.0_15”. In our main experiments, the maximum allowed time for test case instances was set to 120 min.

5.1 Datasets

We have conducted our experiments on two types of data: randomly generated DNA sequences and real gene sequences.

Algorithm 6 updatePheromoneSchedule(iterationCounter,G_{cs},L_{LB},L_{GB})

```
E ← edge blocks of G_cs
decreasePheromone(E)
if iterationCounter ≤ 50 then
  increasePheromone(L_LB)
else if iterationCounter ≤ 100 then
  if iterationCounter MOD 5 == 0 then
    increasePheromone(L_LB)
  else
    increasePheromone(L_GB)
  end if
else if iterationCounter ≤ 200 then
  if iterationCounter MOD 4 == 0 then
    increasePheromone(L_LB)
  else
    increasePheromone(L_GB)
  end if
else if iterationCounter ≤ 400 then
  if iterationCounter MOD 3 == 0 then
    increasePheromone(L_LB)
  else
    increasePheromone(L_GB)
  end if
else if iterationCounter ≤ 800 then
  if iterationCounter MOD 2 == 0 then
    increasePheromone(L_LB)
  else
    increasePheromone(L_GB)
  end if
else
  increasePheromone(L_LB)
end if
Update tau_max and tau_min
for all Block B in E do
  Bound pheromone(B) between tau_max and tau_min
end for
```

Algorithm 7 MMAS(X,Y)

```
h!
G_cs ← construct common substring graph of string X and Y
for run = 1 → nRun do
  initialize(G_cs)
  interationCounter = 0
  repeat
    iterationCounter = iterationCounter + 1;
    Initialize local best
    for i = 1 → nAnts do
      constructSolution(i,G_cs)
      update localBest (L_LB)
    end for
    update globalBest (L_GB)
    updatePheromoneSchedule(iterationCounter,G_cs)
  until time reaches maxAllowedTime or No update found for maxAllowedIteration
end for
```

Table 1 List of parameters

Name	Symbol	Value set
Pheromone information	α	{1,2,3}
Heuristic information	β	{3,5,10}
Evaporation rate	ε	{0.02,0.04,.05}
Number of ants	*nAnts*	{20,60,100}
Probability of best solution	p_{best}	{0.005,0.05,0.5}

The first column represents the name, the second column represents the symbol of the parameter and the third column represent the set of values used for tuning

Table 2 Best found values of the parameters

Parameters	Value
α	2.0
β	10.0
Evaporation rate, ε	0.05
nAnts	100
p_{best}	0.05
initPheromone	10.0
Maximum allowed time	120 min

The first column is the symbol of the parameter and the second column is the best found value

5.1.1 Random DNA Sequences

We have generated 30 random DNA sequences each of length at most 600 using [22]. 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 [26]. 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 lengths within [100–200] bps (base-pairs), the next 10 have lengths within [201, 400] and the rest 10 have lengths within [401, 600] bps.

5.1.2 Real Gene Sequences

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

5.2 Parameter Tuning

There are several parameters which have to be carefully set to obtain good results. To obtain a good set of parameters we have done a preliminary experiment. In our experiment we have chosen 3 values for each of the parameters. so there are 243 possible permutations of the 5 parameters. The values of the parameters used in our experiment is listed in Table 1. We have chosen 2 input cases from each of the groups (group1, group2, group3 and realgene). The time limits are set to 10, 20, 30 and 20 min for the 4 groups, respectively. The algorithm is run for 4 times and the average result is recorded. Let the partition size of each of the case is denoted by A^i where $i \in [1, 8]$. With these settings, we find rank of a permutation by the following rule:

$$R_j = \sum_{i \in [1,8]} A^i_j / max(A^i) \quad \forall j \in [1, 243]$$

After computing the Rank, R, we find the permutation of the parameters for which the rank is minimum. The best found parameters are reported in Table 2.

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

Table 3 Comparison between Greedy approach [6] and MMAS on random DNA sequences (Group 1, [100–200] bps)

Greedy	MMAS(Avg.)	Worst	Best	Difference	SD (MMAS)	Time (s) (MMAS)	tstat	p value	Significance
46	42.8667	43	42	−3.1333	0.3519	114.6243	34.4886	0.0000	+
56	51.8667	52	51	−4.1333	0.5164	100.823	31	0.0000	+
62	57	58	55	−5	0.6547	207.5253	29.5804	0.0000	+
46	43.3333	43	43	−2.6667	0.488	168.3098	21.166	0.0000	+
44	42.9333	43	43	−1.0667	0.2582	42.7058	16	0.0000	+
48	42.8	43	42	−5.2	0.414	75.2033	48.6415	0.0000	+
65	60.6	60	60	−4.4	0.5071	131.9478	33.6056	0.0000	+
51	46.9333	47	47	−4.0667	0.4577	201.2292	34.4086	0.0000	+
46	45.5333	46	45	−0.4667	0.5164	172.6809	3.5	0.0016	+
63	59.7333	60	59	−3.2667	0.7037	288.4226	17.9781	0.0000	+

Here, Difference = MMAS(Avg.) − Greedy. Best and Worst report the maximum and minimum partition size among 15 runs using MMAS

Table 4 Comparison between Greedy approach [6] and MAX–MIN on random DNA sequences (Group 2, [201–400] bps)

Greedy	MMAS	Worst	Best	Difference	SD (MMAS)	Time (s) (MMAS)	tstat	p value	Significance
119	113.9333	116	111	−5.0667	1.3345	1534.1015	14.7042	0.0000	+
122	118.9333	121	117	−3.0667	0.9612	1683.1146	12.3572	0.0000	+
114	112.5333	114	111	−1.4667	0.8338	1398.5315	6.8126	0.0000	+
116	116.4	117	115	0.4	0.7368	1739.3478	−2.1026	0.0446	−
135	132.2	135	130	−2.8	1.3202	1814.7264	8.2143	0.0000	+
108	106.0667	107	105	−1.9333	0.8837	1480.2378	8.4731	0.0000	+
108	98.4	101	96	−9.6	1.2421	1295.2485	29.9333	0.0000	+
123	118.4	120	117	−4.6	0.7368	1125.2353	24.1802	0.0000	+
124	119.4667	121	117	−4.5333	1.0601	1044.4141	16.5622	0.0000	+
105	101.8667	103	101	−3.1333	0.7432	1360.1529	16.328	0.0000	+

Here, Difference = MMAS(Avg.) − Greedy. Best and Worst report the maximum and minimum partition size among 15 runs using MMAS

5.3 Results and Analysis

We have compared our approach with the greedy algorithm of Chrobak [6] 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. As it is expected the greedy algorithm runs very fast. All of the results by greedy algorithm presented in this paper outputs within 2 min.

5.3.1 Random DNA Sequence

Tables 3, 4 and 5 present the comparison between our approach and the greedy approach [6] for the random DNA sequences. For a particular DNA sequence, the experiment was run 15 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 MMAS, the third and fourth column report the worst and best results among 15 runs, the fifth column represents the difference between the two approaches. A positive (negative) difference indicates that the greedy result is better (worse) than the MMAS result by that amount. The sixth column reports the standard deviation of 15 runs of MMAS, the seventh column is the average time in second by which the reported partition size is achieved. The first 3 columns summarize the t-statistic result for greedy versus MMAS. The first column reports the t-value of two sample t-test. A positive t-value indicate significant improvement. The second column presents the p value. A lower p value represent higher significant improvement and the third column reports whether the null hypothesis is rejected or accepted. Here the null hypothesis is that the two random populations (partition sizes from greedy and MMAS) have equal means. We have used $+, -, \approx$ to denote improvement, deterioration and almost

Table 5 Comparison between Greedy approach [6] and MAX–MIN on random DNA sequences (Group 3, [401–600] bps)

Greedy	MMAS	Worst	Best	Difference	SD (MMAS)	Time (s) (MMAS)	tstat	p value	Significance
182	179.9333	181	177	−2.0667	1.7099	1773.0398	4.6810	0.0001	+
175	176.2000	177	175	1.2000	0.8619	3966.8293	−5.3923	0.0000	−
196	187.8667	189	187	−8.1333	0.7432	1589.2953	42.3833	0.0000	+
192	184.2667	185	184	−7.7333	0.4577	2431.1580	65.4328	0.0000	+
176	171.5333	173	171	−4.4667	0.9155	1224.8943	18.8965	0.0000	+
170	163.4667	165	160	−6.5333	1.8465	1826.1438	13.7036	0.0000	+
173	168.4667	170	167	−4.5333	1.1872	1802.1655	14.7886	0.0000	+
185	176.3333	177	175	−8.6667	0.8165	1838.5603	41.1096	0.0000	+
174	172.8000	175	172	−1.2000	1.5675	4897.4688	2.9649	0.0061	+
171	167.2000	168	167	−3.8000	0.5606	1886.2098	26.2523	0.0000	+

Here, Difference = MMAS(Avg.) − Greedy. Best and Worst report the maximum and minimum partition size among 15 runs using MMAS

Table 6 Comparison between MMAS with and without dynamic heuristic on random dna sequence

Group 1 (200 bps)			Group 2 (400 bps)			Group 3 (600 bps)		
MMAS	MMAS (w/o heuristic)	Difference	MMAS	MMAS (w/o heuristic)	Difference	MMAS	MMAS (w/o heuristic)	Difference
42.7500	43.2500	0.5000	114.2500	115.5000	1.2500	180.0000	183.2500	3.2500
51.5000	50.7500	−0.7500	119.0000	121.0000	2.0000	176.2500	183.2500	7.0000
56.7500	56.5000	−0.2500	112.2500	113.5000	1.2500	188.0000	193.7500	5.7500
43.0000	44.0000	1.0000	116.2500	120.5000	4.2500	184.2500	189.2500	5.0000
43.0000	42.7500	−0.2500	132.2500	134.0000	1.7500	171.7500	173.5000	1.7500
42.2500	42.5000	0.2500	105.5000	107.7500	2.2500	163.2500	168.0000	4.7500
60.0000	60.5000	0.5000	99.0000	99.7500	0.7500	168.5000	170.5000	2.0000
47.0000	47.5000	0.5000	118.0000	121.7500	3.7500	176.2500	178.7500	2.5000
45.7500	46.0000	0.2500	119.5000	120.7500	1.2500	172.7500	179.2500	6.5000
59.2500	61.5000	2.2500	101.7500	103.7500	2.0000	167.2500	172.2500	5.0000

equal respectively. According to t-statistic value with 5% significance value we have found better solution in 28 cases for MMAS. For the other 2 case we got worse result in 5% significance level.

5.3.2 Effects of Dynamic Heuristics

In Sect. 4.2.2, we discussed the dynamic heuristic we employ in our algorithm. We conducted experiments to check and verify the effect of this dynamic heuristic. We conducted experiments with two versions of our algorithm—with and without applying the dynamic heuristic. The effect is presented in Table 6, where for each group the average partition size with dynamic heuristic and without dynamic heuristic is reported. The positive difference depicts the improvement using dynamic heuristic. Out of 30 cases we found positive differences on 27 cases. This clearly shows the significant improvement using dynamic heuristics. It can also be observed that with the increase in length, the positive differences are increased. Figures 2, 3, and 4 show the case by case results. The blue bars represent the partition size using dynamic heuristic and the red bars represent the partition size without the dynamic heuristic.

5.3.3 Real Gene Sequence

Table 7 shows the minimum common partition size found by our approach and the greedy approach for the real gene sequences. Out of 15 cases positive improvement is found in 10 cases in 5% significance level.

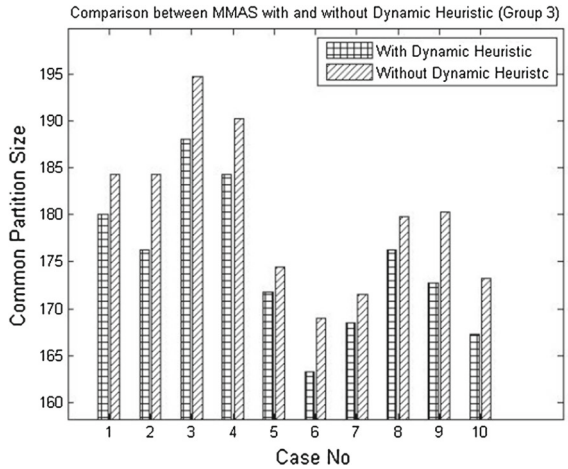

Fig. 2 Comparison between MMAS with and without dynamic heuristic (Group 1)

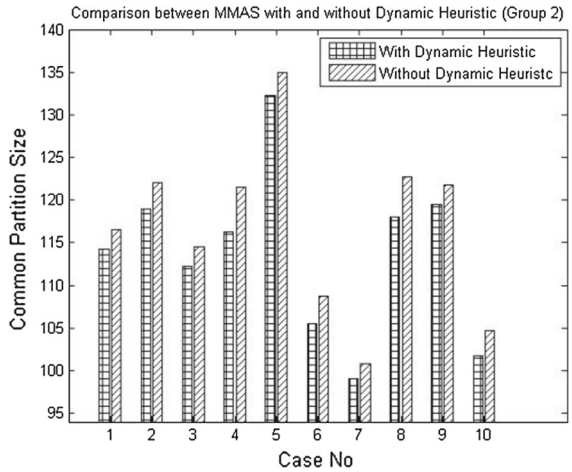

Fig. 3 Comparison between MMAS with and without dynamic heuristic (Group 2)

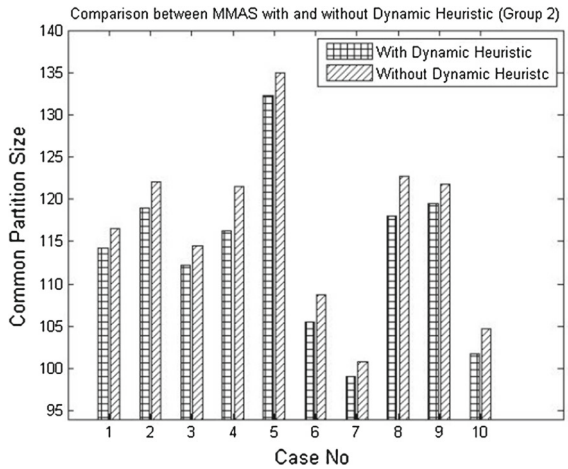

Fig. 4 Comparison between MMAS with and without dynamic heuristic (Group 3)

Table 7 Comparison between Greedy approach [6] and MMAS on real gene sequence

Greedy	MMAS	Worst	Best	Difference	SD (MMAS)	Time (s) (MMAS)	tstat	p value	Significance
95	87.66666667	88	87	−7.333333333	0.487950036	863.8083333	58.2065	0.0000	+
161	156.3333333	162	154	−4.666666667	2.350278606	1748.34	7.6901	0.0000	+
121	117.0666667	118	116	−3.933333333	0.883715102	1823.4922	17.2383	0.0000	+
173	164.8666667	167	163	−8.133333333	1.187233679	1823.012533	26.5325	0.0000	+
172	170.3333	172	169	1.2	1.207121724	2210.153533	3.8501	0.0006	+
153	146	148	143	−7	1.309307341	1953.838267	20.7063	0.0000	+
140	141	142	140	1	0.755928946	2439.0346	−5.1235	0.0000	−
134	133.1333333	136	130	−0.866666667	1.807392228	1406.804533	1.8571	0.0738	≈
149	147.5333333	150	145	−1.466666667	1.505545305	2547.519267	3.7730	0.0008	+
151	150.5333333	152	148	−0.466666667	1.597617273	1619.6364	1.1313	0.2675	≈
126	125	127	123	−1	1	1873.3868	3.8730	0.0006	+
143	139.1333333	141	137	−3.866666667	1.245945806	2473.249067	12.0194	0.0000	+
180	181.5333333	184	179	1.533333333	1.35576371	2931.665333	−4.3802	0.0002	−
152	149.3333333	151	147	−2.666666667	1.290994449	2224.403733	8.0000	0.0000	+
157	161.6	164	160	4.6	1.242118007	1739.612133	–14.3430	0.0000	−

Here, Difference = MMAS(Avg.) − Greedy. Best and Worst report the maximum and minimum partition size among 15 runs using MMAS

6 Conclusion

The 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 with intelligent positioning. The simulation is conducted on random DNA sequences and real gene sequences. The results are significantly better than the previous results. The t-test result also shows significant improvement. As future works other metaheuristic techniques may be applied to present better solutions to the problem.

References

1. Blum, C.: Beam-ACO for the longest common subsequence problem. In: IEEE Congress on Evolutionary Computation, pp. 1–8. IEEE (2010)
2. Blum, C., Lozano, J., Davidson, P.P.: Iterative probabilistic tree search for the minimum common string partition problem. In: Blesa, M., Blum, C., Voß, S. (eds.) Hybrid Metaheuristics, pp. 145–154. Springer, Berlin (2014)
3. Blum, C., Lozano, J., Davidson, P.P.: Mathematical programming strategies for solving the minimum common string partition problem. Eur. J. Oper. Res. **242**, 769–777 (2015)
4. Blum, C., Vallès, M.Y., Blesa, M.J.: An ant colony optimization algorithm for dna sequencing by hybridization. Comput. Oper. Res. **35**(11), 3620–3635 (2008)
5. Chen, X., Zheng, J., Fu, Z., Nan, P., Zhong, Y., Lonardi, S., Jiang, T.: Assignment of orthologous genes via genome rearrangement. IEEE/ACM Trans. Comput. Biol. Bioinform. **2**(4), 302–315 (2005)
6. Chrobak, M., Kolman, P., Sgall, J.: The greedy algorithm for the minimum common string partition problem. ACM Trans. Algorithms **1**(2), 350–366 (2005)
7. Damaschke, P.: Minimum common string partition parameterized. In: Crandall, K., Lagergren, J. (eds.) Algorithms in Bioinformatics. Lecture Notes in Computer Science, vol. 5251, pp. 87–98. Springer, Berlin (2008)
8. Dorigo, M.: Optimization, learning and natural algorithms. PhD thesis, Politecnico di Milano, Milan (1992)
9. Dorigo, M., Colorni, A., Maniezzo, V.: Positive feedback as a search strategy. Technical report 91-016, Dipartimento di Elettronica, Politecnico di Milano, Milan (1991)
10. Dorigo, M., Di Caro, G., Gambardella, L.M.: Ant algorithms for discrete optimization. Artif. Life **5**(2), 137–172 (1999)
11. Dorigo, M., Gambardella, L.M.: Ant colony system: a cooperative learning approach to the traveling salesman problem. Trans. Evol. Comput. **1**(1), 53–66 (1997)
12. Dorigo, M., Maniezzo, V., Colorni, A.: The ant system: optimization by a colony of cooperating agents. IEEE Trans. Syst. Man Cybern. B **26**(1), 29–41 (1996)
13. Dorigo, M., Stützle, T.: Ant Colony Optimization. Bradford Company, Scituate (2004)

14. Dorigo, M., Stützle, T.: Ant colony optimization: overview and recent advances. In: Gendreau, M., Potvin, J.-Y. (eds.) Handbook of Metaheuristics. International Series in Operations Research & Management Science, vol. 146, pp. 227–263. Springer, New York (2010)
15. Ferdous, S., Das, A., Rahman, M., Rahman, M.: Ant colony optimization approach to solve the minimum string cover problem. In: International Conference on Informatics, Electronics and Vision (ICIEV), pp. 741–746. IEEE (2012)
16. Ferdous, S., Rahman, M.: Solving the minimum common string partition problem with the help of ants. In: Tan, Y., Shi, Y., Mo, H. (eds.) Advances in Swarm Intelligence. Lecture Notes in Computer Science, vol. 7928, pp. 306–313. Springer, Berlin (2013)
17. Ferdous, S.M., Rahman, M.S.: An integer programming formulation of the minimum common string partition problem. PLoS ONE **10**, e0130266 (2015)
18. Gambardella, L., Dorigo, M.: Ant-q: a reinforcement learning approach to the traveling salesman problem, pp. 252–260. Morgan Kaufmann, Los Altos (1995)
19. Goldstein, A., Kolman, P., Zheng, J.: Minimum common string partitioning problem: hardness and approximations. Electron. J. Comb. **12**(R50) (2005)
20. Jiang, H., Zhu, B., Zhu, D., Zhu, H.: Minimum common string partition revisited. In: Proceedings of the 4th International Conference on Frontiers in Algorithmics, FAW'10, pp. 45–52. Springer, Berlin (2010)
21. Shyu, S.J., Tsai, C.-Y.: Finding the longest common subsequence for multiple biological sequences by ant colony optimization. Comput. Oper. Res. **36**(1), 73–91 (2009)
22. Stothard, P.: The sequence manipulation suite: Javascript programs for analyzing and formatting protein and DNA sequences. Biotechniques **28**(6), 1102 (2000)
23. Stützle, T., Hoos, H.: Improving the ant system: a detailed report on the max–min ant system. Technical report (1996)
24. Stützle, T., Hoos, H.: Max–min ant system and local search for the traveling salesman problem. In: IEEE International Conference on Evolutionary Computation (ICEC'97), pp. 309–314. IEEE Press (1997)
25. Stützle, T., Hoos, H.H.: Max–min ant system. Future Gener. Comput. Syst. **16**(9), 889–914 (2000)
26. Villesen, P.: Fabox: an online FASTA sequence toolbox (2007)
27. Watterson, G., Ewens, W., Hall, T., Morgan, A.: The chromosome inversion problem. J. Theor. Biol. **99**(1), 1–7 (1982)