Hybridizations of metaheuristics with branch & bound derivates

Christian Blum[1], Carlos Cotta[2], Antonio J. Fernández[2], José E. Gallardo[2], and Monaldo Mastrolilli[3]

[1] ALBCOM research group
Universitat Politècnica de Catalunya
cblum@lsi.upc.edu

[2] Dept. Lenguajes y Ciencias de la Computación
Universidad de Málaga
{ccottap,afdez,pepeg}@lcc.uma.es

[3] Istituto Dalle Molle di Studi sull'Intelligenza Artificiale (IDSIA)
monaldo@idsia.ch

One of the basic ingredients of an optimization technique is a mechanism for exploring the search space, that is, the space of valid solutions to the considered optimization problem. Algorithms belonging to the important class of *constructive optimization techniques* tackle an optimization problem by exploring the search space in form of a tree, a so-called *search tree*. The search tree is generally defined by an underlying solution construction mechanism. Each path from the root node of the search tree to one of the leaves corresponds to the process of constructing a candidate solution. Inner nodes of the tree can be seen as partial solutions. The process of moving from an inner node to one of its child nodes is called a solution construction step, or extension of a partial solution.

The class of constructive optimization techniques comprises approximate methods as well as complete methods. Recall that complete algorithms are guaranteed to find for every finite size instance of a combinatorial optimization problem an optimal solution in bounded time. This is in contrast to incomplete methods such as heuristics and metaheuristics where we sacrifice the guarantee of finding optimal solutions for the sake of getting good solutions in a significantly reduced amount of time. A prominent example of a deterministic constructive heuristic is a *greedy heuristic*. Greedy heuristics make use of a weighting function that gives weights to the child nodes of each inner node of the search tree. At each construction step the child node with the highest weight is chosen.

Apart from greedy heuristics, the class of constructive optimization techniques also includes metaheuristics such as ant colony optimization (ACO) [13] and greedy randomized adaptive search procedures (GRASP) [14].[4] They are iterative algorithms that employ repeated probabilistic (randomized) solution constructions at each iteration. For each child node of an inner node of the tree they compute the probability of performing the corresponding construction step. These probabilities may depend on weighting functions and/or the search history of the algorithm. They are sometimes called *transition probabilities* and define a probability distribution over the search space. In GRASP, this probability distribution does not change during run-time, while in ACO the probability distribution is changed during run-time with the aim of biasing the probabilistic construction of solutions towards areas of the search space containing high quality solutions.

In addition to the methods mentioned above, the class of constructive optimization techniques also includes complete algorithms such as *backtracking* [3] and *branch & bound* (B&B) [25]. A common backtracking scheme is implemented in the depth-first search (DFS) algorithm. The un-informed version of DFS starts from the root node of the search tree and progresses by always moving to the best unexplored child of the current node, going deeper and deeper until a leaf node is reached. Then the search backtracks, returning to the most recently visited node of which remain unexplored children, and so on. This systematic search method explicitly visits all possible solutions exactly once.

Branch & bound algorithms belong to the class of implicit enumeration techniques. The branch & bound view on the search tree is slightly different to that exhibited by the algorithms mentioned before. More specifically, the subtree rooted at an inner node of the search tree is seen as a subspace of the search space. Accordingly, the subspaces represented by the subtrees rooted at the children of an inner node consitute a partition of the subspace that is represented by the inner node itself. The partitioning of the search space is called branching. A branch & bound algorithm produces for each inner node of the search tree an upper bound as well as a lower bound of the objective function values of the solutions contained by the corresponding subspace. These bounds are used to decide if the whole subspace can be discarded, or if it has to be further partitioned. As in backtracking, there are different schemes such as depth-first search or breadth-first search for traversing over the search tree.

An interesting heuristic version of a breadth-first branch & bound is *beam search* [33]. Instead of considering all nodes of a certain level of the search tree, beam search restricts the search to a certain number of nodes based on the bounding information (lower bounds for minimization, and upper bounds

[4] See Chapter 1 of this book for a comprehensive introduction to metaheuristics.

for maximization).

Each of the algorithms mentioned above has advantages as well as disadvantages. Greedy heuristics, for example, are usually easy to implement and fast in execution. However, the obtained solution quality is often not sufficient. Metaheuristics, on the other side, can find good solutions in a reasonable amount of time, without providing performance guarantees. However, metaheuristics can generally not avoid visiting the same solution more than once, which might lead to a waste of computation time. Finally, complete techniques guarantee to find an optimal solution. However, a user might not be prepared to accept overly large running times. In recent years it has been shown that a hybridization of concepts originating from different types of algorithms can often result in more efficient techniques. For example, the use of backtracking in metaheuristics is relatively wide-spread. Examples of their use in construction-based metaheuristics are [2, 4, 10]. Backtracking is also used in evolutionary algorithms (see, for example, [11, 24]), and even in tabu search settings [32]. The hybridization of metaheuristics with branch & bound (respectively, beam search) concepts is rather recent. We distinguish between two different lines of hybridization. On one side, it is possible to use branch & bound concepts within construction-based metaheuristics in order to increase the efficiency of the metaheuristics search process. On the other side, metaheuristics can be used within branch & bound in order to reduce the space and time consumption of branch & bound. This chapter is dedicated to outline representative examples of both types of hybrid algorithms. The reader interested in a broader discussion on the combination of metaheuristics and exact techniques is referred to [34].

1 Using branch & bound concepts within construction-based metaheuristics

Recent hybrid metaheuristics include some important features that are inspired by deterministic branch & bound derivatives such as beam search:

1. Bounding information is used for evaluating partial solutions; sometimes also for choosing among different partial solutions, or discarding partial solutions.
2. The extension of partial solutions is allowed in more than one way. The number of nodes which can be selected at each search tree level is hereby limited from above by a parametric constraint, resulting in parallel and non-independent solution constructions.

This type of hybrid algorithm includes probabilistic beam search (PBS) [7], Beam-ACO algorithms [5, 6], and approximate and non-deterministic tree search (ANTS) procedures [27, 28, 29].[5] These works give empiricial evidence of the usefulness of including the two features mentioned above in the construction process of construction-based metaheuristics.

In the following we first give a motivation of why the above mentioned branch & bound features should be incorporated in construction-based metaheuristics. Afterwards, we present some representative approaches.

1.1 A tree search model

The following tree search model captures the essential elements common to all constructive procedures. In general, we are given an optimization problem $\mathcal{P}$ and an instance x of $\mathcal{P}$. Typically, the search space S_x is exponentially large in the size of the input x. Without loss of generality we intend to maximize the objective function $f : S_x \mapsto \mathbb{R}^+$. The optimization goal is to find a solution $y \in S_x$ to x with $f(y)$ as great as possible. Assume that each element $y \in S_x$ can be viewed as a composition of $l_{y,x} \in \mathbb{N}$ elements from a set Σ. From this point of view, S_x can be seen as a set of strings over an alphabet Σ. Any element $y \in S_x$ can be constructed by concatenating $l_{y,x}$ elements of Σ.

The following method for constructing elements of S_x is instructive: A solution construction starts with the empty string ϵ. The construction process consists of a sequence of construction steps. At each construction step, we select an element of Σ and append it to the current string t. The solution construction may end for two reasons. First, it may end in case t has no feasible extensions. This happens in case t is already a complete solution, or when no solution of S_x has prefix t. Second, a solution construction ends in case of available upper bound information that indicates that each solution with prefix t is worse than any solution that is already known. Henceforth we denote the upper bound value of a partial solution t by $\mathrm{UB}(t)$.

The application of such an algorithm can be equivalently described as a walk from the root v_0 of the corresponding search tree to a node at level $l_{y,x}$. The search tree has nodes for all $y \in S_x$ and for all prefixes of elements of S_x. The root of the tree is the empty string, that is, v_0 corresponds to ϵ. There is a directed arc from node v to node w if w can be obtained by appending an element of Σ to v. Note that henceforth we identify a node v of the search tree with its corresponding string t. We will use both notations interchangeably. The set of nodes that can be reached from a node v via directed arcs are called the children of v, denoted by $C(v)$. Note, that the nodes at level i correspond to strings of length i. If w is a node corresponding to a string of length $l > 0$ then the length $l-1$ prefix v of w is also a node, called the father of w denoted

[5] Note that the algorithms presented in [27, 28] only use the first one of the features mentioned above.

Algorithm 1 Solution construction: SC($\hat{f}$)

```
 1: input: the best known objective function value $\hat{f}$ (might be 0)
 2: initialization: $v := v_0$
 3: while $|C(v)| > 0$ and $v \neq$ NULL do
 4:   $w :=$ ChooseFrom($C(v)$)
 5:   if UB($w$) $> \hat{f}$ then
 6:     $v :=$ NULL
 7:   else
 8:     $v := w$
 9:   end if
10: end while
11: output: $v$ (which is either a complete solution, or NULL)
```

by $\mathcal{F}(w)$. Thus, every $y \in S_x$ corresponds to exactly one path of length $l_{y,x}$ from the root node of the search tree to a specific leaf. The above described solution construction process is pseudo-coded in Algorithm 1. In the following we assume function ChooseFrom($C(v)$) of this algorithm to be implemented as a probabilistic choice function.

1.2 Primal and dual problem knowledge

The analysis provided in the following assumes that there is a unique optimal solution, represented by leaf node v_d of the search tree, also referred to as the target node. Let us assume that — without loss of generality — the target node v_d is at the maximum level $d \geq 1$ of the search tree. A probabilistic constructive optimization algorithm is said to be *successful*, if it can find the target node v_d with high probability.

In the following let us examine the success probability of repeated applications of Algorithm 1 in which function ChooseFrom($C(v)$) is implemented as a probabilisitc choice function. Such solution constructions are employed, for example, within the ACO metaheuristic. The value of the input $\hat{f}$ is not important for the following analysis. Given any node v_i at level i of the search tree, let $\mathbf{p}(v_i)$ be the probability that a solution construction process includes node v_i. Note that there is a single path from v_0, the root node, to v_i. We denote the corresponding sequence of nodes by $(v_0, v_1, v_2, ..., v_i)$. Clearly, $\mathbf{p}(v_0) = 1$ and $\mathbf{p}(v_i) = \prod_{j=0}^{i-1} \mathbf{p}(v_{j+1}|v_j)$. Let $Success(\rho)$ denote the event of finding the target node v_d within ρ applications of Algorithm 1. Note that the probability of $Success(\rho)$ is equal to $1 - (1 - \mathbf{p}(v_d))^\rho$, and it is easy to check that the following inequalities hold:

$$1 - e^{-\rho \mathbf{p}(v_d)} \leq 1 - (1 - \mathbf{p}(v_d))^\rho \leq \rho \mathbf{p}(v_d) \qquad (1)$$

By (1), it immediately follows that the chance of finding node v_d is *large* if and only if $\rho \mathbf{p}(v_d)$ is *large*, namely as soon as

$$\rho = O\left(1/\mathbf{p}(v_d)\right) \tag{2}$$

In the following, we will not assume anything about the exact form of the given probability distribution. However, let us assume that the transition probabilities are heuristically related to the *attractiveness* of child nodes. In other words, we assume that in a case in which a node v has two children, say w and q, and w is known (or believed) to be *more promising*, then $\mathbf{p}(w|v) > \mathbf{p}(q|v)$. This can be achieved, for example, by defining the transition probabilities proportional to the weights assigned by greedy functions.

Clearly, the probability distribution reflects the available knowledge on the problem, and it is composed of two types of knowledge. If the probability $\mathbf{p}(v_d)$ of reaching the target node v_d is "high", then we have a "good" problem knowledge. Let us call the knowledge that is responsible for the value of $\mathbf{p}(v_d)$ the **primal problem knowledge** (or just primal knowledge). From the dual point of view, we still have a "good" knowledge of the problem if for "most" of the wrong nodes (i.e. those that are not on the path from v_0 to v_d) the probability that they are reached is "low". We call this knowledge the **dual problem knowledge** (or just dual knowledge). Note that the quality of the dual knowledge grows with the value $\hat{f}$ that is provided as input to Algorithm 1. This means, the better the solution that we already know, the higher is the quality of the dual knowledge. Observe that the two types of problem knowledge outlined above are complementary, but not the same. Let us make an example to clarify these two concepts. Consider the search tree of Figure 1, where the target node is v_5. Let us analyze two different probability distributions:

Case (a) For each v and $w \in \mathcal{C}(v)$ let $\mathbf{p}(w|v) = 0.5$. Moreover, let us assume that no upper bound information is available. This means that each solution construction is performed until a leaf node is reached. When probabilistically constructing a solution the probability of each child is therefore the same at each construction step.

Case (b) In general, the transition probabilities are defined as in case (a), with one exception. Let us assume that the available upper bound indicates that the subspaces represented by the black nodes do not contain any better solutions than the ones we already know, that is, $\mathrm{UB}(v) \leq \hat{f}$, where v is a black node. Accordingly, the white children of the black nodes have probability 0 to be reached.

Note that in both cases the primal knowledge is "scarce", since the probability that the target node v_d is reached by a probabilistic solution construction decreases exponentially with d, that is, $\mathbf{p}(v_d) = 2^{-d}$. However, in **case (b)** the dual knowledge is "excellent", since for most of the wrong nodes (i.e. the white nodes), the probability that any of them is reached is zero. Viceversa, in **case (a)** the dual knowledge is "scarce", because there is a relatively "high"

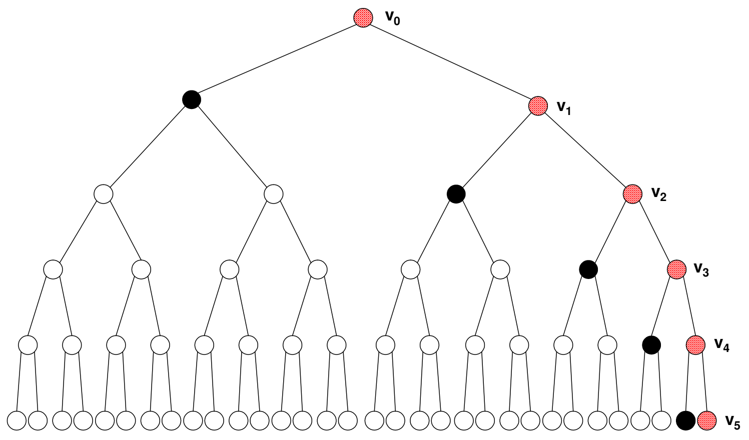


Fig. 1. Example of a search tree. v_5 is the unique optimal solution.

probability that a white node is reached.

By using the intuition given by the provided example, let us try to better quantify the quality of the available problem knowledge. Let V_i be the set of nodes at level i, and let

$$\ell(i) = \sum_{v \in V_i} \mathbf{p}(v), \qquad i = 1, \ldots, d \ . \tag{3}$$

Note that $\ell(i)$ is equal to the probability that the solution construction process reaches level i of the search tree. Observe that the use of the upper bound information makes the probabilities $\ell(i)$ smaller than one. **Case (b)** was obtained from **case (a)** by decreasing $\ell(i)$ (for $i = 1, \ldots, d$) down to 2^{i-1} (and without changing the probability $\mathbf{p}(v_i)$ of reaching the ancestor v_i of the target node at level i), whereas in **case (a)** it holds that $\ell(i) = 1$ (for $i = 1, \ldots, d$). In general, good dual knowledge is supposed to decrease $\ell(i)$ without decreasing the probability of reaching the ancestor v_i of the target node v_d. This discussion may suggest that a characterization of the available problem knowledge can be given by the following *knowledge ratio*:

$$K_{v_d} = \min_{1 \leq i \leq d} \frac{\mathbf{p}(v_i)}{\ell(i)} \tag{4}$$

The larger this ratio the better the knowledge we have on the target node v_d. In **case (a)** it is $K_{v_d} = 1/2^d$, whereas the knowledge ratio of **case (b)** is

$K_{v_d} = 1/2$, which is exponentially larger.

Finally, it is important to observe that the way of (repeatedly) constructing solutions in a probabilistic way does not exploit the dual problem knowledge. For example in **case (b)**, although the available knowledge is "excellent", the target node v_d is found after an expected number of runs that is proportional to $1/\mathbf{p}(v_d) = 2^d$ (see Equation (2)), which is the same as in **case (a)**. In other words, the number of necessary probabilistic solution constructions only depends on the primal knowledge.

1.3 How to exploit the dual knowledge?

The problem of Algorithm 1 is clearly the following one: When encountering a partial solution whose upper bound is less or equal to the value of the best solution found so far, the construction process is aborted, and the computation time invested in this construction is lost. Generally, this situation may occur very often. In fact, the probability for the abbortion of a solution construction is $1-\mathbf{p}(v_d)$ in the example outlined in the previous section, which is quite high.

In the following let us examine a first simple extension of Algorithm 1. The corresponding algorithm — henceforth denoted by PSC(α,$\hat{f}$) — is pseudo-coded in Algorithm 2. Hereby, α denotes the maximum number of allowed extensions of partial solutions at each construction step; in other words, α is the maximum number of solutions to be constructed in parallel. We use the following additional notation: For any given set S of search tree nodes let $\mathcal{C}(S)$ be the set of children of the nodes in S. Morever, B_i denotes the set of reached nodes of tree level i. Recall that the root node v_0 is the only node at level 0.

The algorithm works as follows. Given the selected nodes B_i of level i (with $|B_i| \leq \alpha$)), the algorithm probabilistically chooses at most α solutions from $C := \mathcal{C}(B_i)$, the children of the nodes in B_i. The probabilistic choice of a child is performed in function $\mathsf{ChooseFrom}(C)$ proportionally to the following probabilities:

$$\mathbf{p}(w|C) = \frac{\mathbf{p}(w|\mathcal{F}(w))}{\sum\limits_{v \in C} \mathbf{p}(v|\mathcal{F}(v))} \quad , \forall\, w \in C \tag{5}$$

Remember that $\mathcal{F}(w)$ denotes the father of node w. After choosing a node w it is first checked if w is a complete solution, or not. In case it is not a complete solution, it is checked if the available bounding information allows the further extension of this partial solution, in which case w is added to B_{i+1}. However, if w is already a complete solution, it is checked if its value is better than the value of the best solution found so far. The algorithm returns the best solution found, in case it is better than the $\hat{f}$ value that was provided as

Algorithm 2 Parallel solution construction: PSC(α,$\hat{f}$)

```
1: input: $\alpha \in \mathbb{Z}^+$, the best known objective function value $\hat{f}$
2: initialization: $i := 0$, $B_i := \{v_0\}$, $z :=$ NULL
3: while $B_i \neq \emptyset$ do
4:    $B_{i+1} := \emptyset$
5:    $C := \mathcal{C}(B_i)$
6:    for $k = 1, \ldots, \min\{\alpha, |\mathcal{C}(B_i)|\}$ do
7:       $w :=$ ChooseFrom($C$)
8:       if $|\mathcal{C}(w)| > 0$ then
9:          if UB($w$) $> \hat{f}$ then $B_{i+1} := B_{i+1} \cup \{w\}$ end if
10:      else
11:         if $f(w) > \hat{f}$ then $z := w$, $\hat{f} := f(z)$ end if
12:      end if
13:      $C := C \setminus \{w\}$
14:   end for
15:   $i := i + 1$
16: end while
17: output: $z$ (which might be NULL)
```

input. Otherwise the algorithm returns NULL.

Observe that when $\alpha = 1$, PSC(α,$\hat{f}$) is equivalent to SC($\hat{f}$). In contrast, when $\alpha > 1$ the algorithm constructs (maximally) α solutions non-independently in parallel. Concerning the example outlined in the previous section with the probability distribution as defined in **case(b)**, we can observe that algorithm PSC(α,$\hat{f}$) with $\alpha > 1$ solves this problem even within one application. At each step i, B_i will only contain the brother of the corresponding black node, because the upper bound information does not allow the inclusion of the black nodes in B_i. This shows that algorithm PSC(α,$\hat{f}$), in contrast to algorithm SC($\hat{f}$), benefically uses the dual problem knowledge.

1.4 Probabilistic beam search

For practical optimization, algorithm PSC(α,$\hat{f}$) has some drawbacks. First, in most cases algorithms for optimization are applied with the goal of finding a solution as good as possible, without having a clue beforehand about the value of good solutions. Second, the available upper bound function might not be very tight. For both reasons, solution constructions that lead to unsatisfying solutions are discarded only at very low levels of the search tree, that is, close to the leaves. Referring to the example of Section 1.2, this means that black nodes will only appear close to the leaves. In those cases, algorithm PSC(α,$\hat{f}$) will have practically no advantage over algorithm SC($\hat{f}$). It might even have a disadvantage due to the amount of computation time invested in choosing children from bigger sets.

Algorithm 3 Probabilistic beam search: PBS(α,$\mu\hat{f}$)

```
 1: input: $\alpha, \mu \in \mathbb{Z}^+$, the best known objective function value $\hat{f}$
 2: initialization: $i := 0$, $B_i := \{v_0\}$, $z :=$ NULL
 3: while $B_i \neq \emptyset$ do
 4:   $B_{i+1} := \emptyset$
 5:   $C := \mathcal{C}(B_i)$
 6:   for $k = 1, \ldots, \min\{\mu \cdot \alpha, |\mathcal{C}(B_i)|\}$ do
 7:     $w := \mathsf{ChooseFrom}(C)$
 8:     if $|\mathcal{C}(w)| > 0$ then
 9:       if $\mathrm{UB}(w) > \hat{f}$ then $B_{i+1} := B_{i+1} \cup \{w\}$ end if
10:     else
11:       if $f(w) > \hat{f}$ then $z := w$, $\hat{f} := f(z)$ end if
12:     end if
13:     $C := C \setminus \{w\}$
14:   end for
15:   Restrict $B_{i+1}$ to the (maximally) $\alpha$ best nodes w.r.t. their upper bound
16:   $i := i + 1$
17: end while
18: output: $z$ (which might be NULL)
```

The following simple extension can help in overcoming the drawbacks of algorithm PSC(α,$\hat{f}$). At each algorithm iteration we allow the choice of $\mu \cdot \alpha$ nodes from B_i, instead of α nodes. $\mu \geq 1$ is a parameter of the algorithm. Moreover, after the choice of the child nodes we restrict set B_{i+1} to the (maximally) α best solutions with respect to the upper bound information. This results in a so-called (probabilistic) beam search algorithm — henceforth denoted by PBS(α,μ,$\hat{f}$) — pseudo-coded in Algorithm 3. Note that algorithm PBS(α,μ,$\hat{f}$) is a generalization of algorithm PSC(α,$\hat{f}$), that is, when $\mu = 1$ both algorithms are equivalent. Algorithm PBS(α,μ,$\hat{f}$) is also a generalization of algorithm SC($\hat{f}$), which is obtained by $\alpha = \mu = 1$.

1.5 Adding a learning component: Beam-ACO

In general, algorithm PBS(α,μ,$\hat{f}$) can be expected to produce good solutions if (at least) two conditions are fullfilled: Neither the greedy function nor the upper bound function are misleading. If at least one of these two functions is misleading, the algorithm might not be able to find solutions above a certain threshold. One possibility of avoiding this drawback is to add a learning component to algorithm PBS(α,μ,$\hat{f}$), that is, adding a mechanism that is supposed to adapt the primal knowledge, the dual knowledge, or both, over time, based on accumulated search experience.

Ant colony optimization (ACO) [13] is the most prominent construction-based metaheuristics that attempts to learn the primal problem knowledge during run-time. ACO is inspired by the foraging behavior of ant colonies. At the core of this behavior is the indirect communication between the ants by means of chemical pheromone trails, which enables them to find short paths between their nest and food sources. This characteristic of real ant colonies is exploited in ACO algorithms in order to solve, for example, combinatorial optimization problems. In general, the ACO approach attempts to solve an optimization problem by iterating the following two steps:

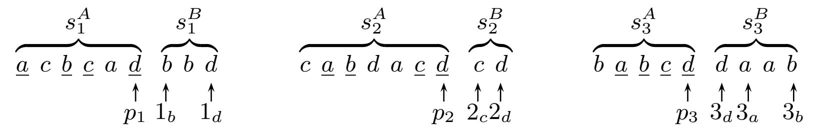
- At each iteration, a certain number of α candidate solutions is probabilistically constructed. The respective probability distribution is derived from an available greedy function and from the values of so-called pheromone trail parameters, the *pheromone values*. The set of pheromone trail parameters is denoted by $\mathcal{T}$.
- The constructed candidate solutions are used to modify the pheromone values in a way that is deemed to bias future solution constructions towards areas of the search space containing high quality solutions. Hereby, the greedy function can be seen as the a priori available primal knowledge, whereas the pheromone values are used to modify (ideally, to improve) this a priori given primal knowledge over time.

While standard ACO algorithms use α applications of algorithm SC($\hat{f}$) at each iteration for the probabilistic construction of solutions, the idea of Beam-ACO [5, 6] is to use one application of probabilistic beam search PBS(α,μ,$\hat{f}$) instead.

A related ACO approach is labelled ANTS (see [27, 28, 29]). The characterizing feature of ANTS is the use of upper bound information for defining the primal knowledge. The latest version of ANTS [29] uses at each iteration algorithm PSC(α,$\hat{f}$) to construct candidate solutions.

1.6 Example: Longest common subsequence (LCS) problem

The longest common subsequence (LCS) problem is one of the classical string problems. Given a problem instance $(\mathcal{S}, \Sigma)$, where $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ is a set of n strings over a finite alphabet Σ, the problem consists in finding a longest string t^* that is a subsequence of all the strings in $\mathcal{S}$. Such a string t^* is called a *longest common subsequence* of the strings in $\mathcal{S}$. Note that a string t is called a subsequence of a string s, if t can be produced from s by deleting characters. For example, dga is a subsequence of $adagtta$. If $n = 2$ the problem is polynomially solvable, for example, by dynamic programming [18]. However, when $n > 2$ the problem is in general NP-hard [26]. Traditional applications of this problem are in data compression, syntactic pattern recognition, and file comparison [1], whereas more recent applications also include computational biology [37].


s_1^A
s_1^B
$\underline{a}$ c $\underline{b}$ $\underline{c}$ a $\underline{d}$ b b d
p_1 1_b 1_d
s_2^A
s_2^B
c $\underline{a}$ $\underline{b}$ d a $\underline{c}$ $\underline{d}$ c d
p_2 2_c 2_d
s_3^A
s_3^B
b $\underline{a}$ $\underline{b}$ $\underline{c}$ $\underline{d}$ d a a b
p_3 3_d 3_a 3_b

(a) String s_1 (b) String s_2 (c) String s_3

Fig. 2. Given is the problem instance ($\mathcal{S} = \{s_1, s_2, s_3\}, \Sigma = \{a, b, c, d\}$) where $s_1 = acbcadbbd$, $s_2 = cabdacdcd$, and $s_3 = babcddaab$. Let us assume that $t = abcd$. (a), (b), and (c) show the corresponding division of s_i into s_i^A and s_i^B, as well as the setting of the pointers p_i and the next positions of the 4 letters in s_i^B. Note that in case a letter does not appear in s_i^B (for example, letter a does not appear in s_1^B), the corresponding pointer is set to ∞. For example, as letter a does not appear in s_1^B, we set $1_a := \infty$.

In order to apply algorithm PBS(α,μ,$\hat{f}$) to the LCS problem, we have to define the solution construction mechanism, the greedy function that defines the primal knowledge, and the upper bound function that defines the dual knowledge. We use the construction mechanism of the so-called Best-Next heuristic [16, 22] for our algorithm. Given a problem instance $(\mathcal{S}, \Sigma)$, this heuristic produces a common subsequence t sequentially by appending at each construction step a letter to t such that t maintains the property of being a common subsequence of all strings in $\mathcal{S}$. Given a common subsequence t of the strings in $\mathcal{S}$, we explain in the following how to derive the children of t. For that purpose we introduce the following notations:

1. Let $s_i = s_i^A \cdot s_i^B$ be the partition of s_i into substrings s_i^A and s_i^B such that t is a subsequence of s_i^A and s_i^B has maximal length. Given this partition, which is well-defined, we introduce position pointers $p_i := |s_i^A|$ for $i = 1, \ldots, n$ (see Figure 2 for an example).
2. The position of the first appearance of a letter $a \in \Sigma$ in a string $s_i \in \mathcal{S}$ after the position pointer p_i is well-defined and denoted by i_a. In case a letter $a \in \Sigma$ does not appear in s_i^B, i_a is set to ∞ (see Figure 2).
3. A letter $a \in \Sigma$ is called *dominated*, if exists at least one letter $b \in \Sigma$ such that $i_b < i_a$ for $i = 1, \ldots, n$;
4. $\Sigma_t^{\mathrm{nd}} \subseteq \Sigma$ henceforth denotes the set of non-dominated letters of the alphabet Σ with respect to a given t. Moreover, for all $a \in \Sigma_t^{\mathrm{nd}}$ it is required that $i_a < \infty$, $i = 1, \ldots, n$. Hence, we require that in each string s_i a letter $a \in \Sigma_t^{\mathrm{nd}}$ appears at least once after position pointer p_i.

The children $\mathcal{C}(t)$ of a node t are then determined as follows: $\mathcal{C}(t) := \{v = ta \mid a \in \Sigma_t^{\mathrm{nd}}\}$. The primal problem knowledge is derived from the greedy function $\eta(\cdot)$ that assigns to each child $v = ta \in \mathcal{C}(t)$ the following greedy weight:

$$\eta(v) = \min\{|s_i| - i_a \mid i = 1, \ldots, n\} \tag{6}$$

The child with the highest greedy weight is considered the *most promising one*. Instead of the greedy weights themselves, we will use the corresponding ranks. More in detail, the child $v = ta$ with the highest greedy weight will be assigned rank 1, denoted by $r(v) = 1$, the child $w = tb$ with the second-highest greedy weight will be assigned rank 2 (that is, $r(w) = 2$), and so on.

In the following we explain the implementation of function $\mathsf{ChooseFrom}(C)$ of algorithm PBS(α,μ,$\hat{f}$). Remember that C denotes the set of children obtained from the nodes that are contained in the beam B_i (that is, $C := \mathcal{C}(B_i)$). For evaluating a child $v \in C$ we use the sum of the ranks of the greedy weights that correspond to the construction steps performed to construct string v. Let us assume that v is on the i-th level of the search tree, and let us denote the sequence of characters that forms string v by $v_1 \ldots v_i$, that is, $v = v_1 \ldots v_i$. Then,

$$\nu(v) := \sum_{j=1}^{i} r(v_1 \ldots v_j), \tag{7}$$

where $v_1 \ldots v_j$ denotes the substring of v from position 1 to position j. With this definition, Equation 5 can be defined for the LCS problem as follows:

$$\mathbf{p}(v|C) = \frac{\nu(v)^{-1}}{\sum_{w \in C} \nu(w)^{-1}}, \quad \forall\, v \in C \tag{8}$$

Finally, we outline the upper bound function UB($\cdot$) that the PBS(α,μ,$\hat{f}$) algorithm requires. Remember that a given subsequence t splits each string $s_i \in \mathcal{S}$ into a first part s_i^A and into a second part s_i^B, that is, $s_i = s_i^A \cdot s_i^B$. Henceforth, $|s_i^B|_a$ denotes the number of occurrences of letter a in s_i^B for all $a \in \Sigma$. Then,

$$\mathrm{UB}(t) := |t| + \sum_{a \in \Sigma} \min\{|s_i^B|_a \mid i = 1, \ldots, n\} \tag{9}$$

i.e., for each letter $a \in \Sigma$ we take the minimum of the occurrences of a in s_i^B, $i = 1, \ldots, n$. Summing up these minima and adding the result to the length of t results in the upper bound. This completes the description of the implementation of the PBS(α,μ,$\hat{f}$) algorithm for the LCS problem.

In the following, we use algorithm PBS(α,μ,$\hat{f}$) in two different ways: First, we use PBS(α,μ,$\hat{f}$) in a multi-start fashion as shown in Algorithm 4, denoted by MS-PBS(α,μ). Second, we use PBS(α,μ,$\hat{f}$) within a Beam-ACO algorithm as explained in the following.

The first step of defining a Beam-ACO approach — and, in general, any ACO algorithm — consists in the specification of the set of pheromone values $\mathcal{T}$. In the case of the LCS problem $\mathcal{T}$ contains for each position j of a string $s_i \in S$ a pheromone value $0 \leq \tau_{ij} \leq 1$, that is, $\mathcal{T} = \{\tau_{ij} \mid i = 1, \ldots, n,\ j = 1, \ldots, |s_i|\}$. A value $\tau_{ij} \in \mathcal{T}$ indicates the desirability of adding the letter at position j of string i to a solution: the greater τ_{ij}, the greater is the desirability of adding the corresponding letter. In addition to the definition of the

Algorithm 4 Multi-start probabilistic beam search: MS-PBS(α,μ)

```
1: input: $\alpha, \mu \in \mathbb{Z}^+$
2: $z :=$ NULL
3: $\hat{f} := 0$
4: while CPU time limit not reached do
5:    $v :=$ PBS($\alpha$,$\mu$,$\hat{f}$) {see Algorithm 3}
6:    if $v \neq$ NULL then $z := v$, $\hat{f} := |z|$
7: end while
8: output: $z$
```

pheromone values, we also introduce a solution representation that is more suitable for ACO. Any common subsequence t of the strings in S can be translated into an ACO-solution $T = \{T_{ij} \in \{0,1\} \mid i = 1,\ldots,n,\ j = 1,\ldots,|s_i|\}$ where $T_{ij} = 0$ if the letter at position j of string i was *not* added to t during the solution construction, and $T_{ij} = 1$ otherwise. Note that the translation of t into T is well-defined due to the construction mechanism. For example, given solution $t = abcdd$ for the problem instance of Figure 2, the corresponding ACO-solution is $T_1 = 101101001$, $T_2 = 011001101$, and $T_3 = 011111000$, where T_i refers to the sequence $T_{i1} \ldots T_{i|s_i|}$. In the following, for each given solution, the lower case notation refers to its string representation, and the upper case notation refers to its binary representation.

The particular ACO framework that we used for our algorithm is the so-called $\mathcal{MAX}-\mathcal{MIN}$ Ant system ($\mathcal{MM}$AS) algorithm implemented in the hyper-cube framework (HCF); This algorithm consists of an ACO variant that performs very well and whose particularity is that applies a lower and an upper bound to the pheromone values with the aim of preventing convergence to a solution, see [8]. A high level description of the algorithm is given in Algorithm 5. The data structures used, in addition to counters and to the pheromone values, are: (1) the *best-so-far* solution T^{bs}, i.e., the best solution generated since the start of the algorithm; (2) the *restart-best* solution T^{rb}, that is, the best solution generated since the last restart of the algorithm; (3) the *convergence factor cf*, $0 \leq cf \leq 1$, which is a measure of how far the algorithm is from convergence; and (4) the Boolean variable *bs_update*, which becomes true when the algorithm reaches convergence.

Roughly, the algorithm works as follows. First, all the variables are initialized. In particular, the pheromone values are set to their initial value 0.5. Each algorithm iteration consists of the following steps. First, algorithm PBS(α,μ,$\hat{f}$) is applied with $\hat{f} = 0$ to generate a solution T^{pbs}. The setting of $\hat{f} = 0$ is chosen, because in ACO algorithms it is generally useful to learn also from solutions that are worse than the best solution found so far. The only change in algorithm PBS(α,μ,$\hat{f}$) occurs in the definition of the choice probabilities. Instead of using Equation 8, these probabilities are now defined as follows:

Algorithm 5 Beam-ACO for the LCS problem

```
 1: input: $\alpha, \mu \in \mathbb{Z}^+$
 2: $T^{bs} :=$ NULL, $T^{rb} :=$ NULL, $cf := 0$, $bs\_update :=$ FALSE
 3: $\tau_{ij} := 0.5$, $i = 1, \ldots, n$, $j = 1, \ldots, |s_i|$
 4: while CPU time limit not reached do
 5:    $T^{pbs} :=$ PBS($\alpha$,$\mu$,0) {see Algorithm 3}
 6:    if $|T^{pbs}| > |t^{rb}|$ then $T^{rb} := T^{pbs}$
 7:    if $|T^{pbs}| > |t^{bs}|$ then $T^{bs} := T^{pbs}$
 8:    ApplyPheromoneUpdate($cf$,$bs\_update$,$\mathcal{T}$,$T^{pbs}$,$T^{rb}$,$T^{bs}$)
 9:    $cf :=$ ComputeConvergenceFactor($\mathcal{T}$)
10:    if $cf > 0.99$ then
11:       if $bs\_update =$ TRUE then
12:          $\tau_{ij} := 0.5$, $i = 1, \ldots, n$, $j = 1, \ldots, |s_i|$
13:          $T^{rb} :=$ NULL
14:          $bs\_update :=$ FALSE
15:       else
16:          $bs\_update :=$ TRUE
17:       end if
18:    end if
19: end while
20: output: $t^{bs}$ (that is, the string version of ACO-solution $T^{bs}$)
```

$$\mathbf{p}(v = ta|C) = \frac{\left(\min\limits_{i=1,\ldots,n}\{\tau_{ii_a}\} \cdot \nu(v)^{-1}\right)}{\sum\limits_{w=tb\in C}\left(\min\limits_{i=1,\ldots,n}\{\tau_{ii_b}\} \cdot \nu(w)^{-1}\right)}, \quad \forall\, v = ta \in C \tag{10}$$

Remember in this context, that i_a was defined as the next position of letter a after position pointer p_i in string s_i. The intuition of choosing the minimum of the pheromone values corresponding to the next positions of a letter in the n given strings is as follows: If at least one of these pheromone values is low, the corresponding letter should not yet be appended to the string, because there is another letter that should be appended first.

The second action at each iteration concerns the pheromone update conducted in the ApplyPheromoneUpdate(cf, bs_update, $\mathcal{T}$, T^{pbs}, T^{rb}, T^{bs}) procedure. Third, a new value for the convergence factor cf is computed. Depending on this value, as well as on the value of the Boolean variable bs_update, a decision on whether to restart the algorithm or not is made. If the algorithm is restarted, all the pheromone values are reset to their initial value (that is, 0.5). The algorithm is iterated until the CPU time limit is reached. Once terminated, the algorithm returns the string version t^{bs} of the best-so-far ACO-solution T^{bs}. In the following we describe the two remaining procedures of Algorithm 5 in more detail.

Table 1. Setting of κ_{pbs}, κ_{rb}, κ_{bs}, and ρ depending on the convergence factor cf and the Boolean control variable *bs_update*

	bs_update = FALSE				bs_update
	$cf < 0.4$	$cf \in [0.4, 0.6)$	$cf \in [0.6, 0.8)$	$cf \geq 0.8$	= TRUE
κ_{ib}	1	2/3	1/3	0	0
κ_{rb}	0	1/3	2/3	1	0
κ_{bs}	0	0	0	0	1
ρ	0.2	0.2	0.2	0.15	0.15

ApplyPheromoneUpdate(cf,bs_update,$\mathcal{T}$,T^{pbs},T^{rb},T^{bs}): In general, three solutions are used for updating the pheromone values. These are the solution T^{pbs} generated by the PBS algorithm, the restart-best solution T^{rb}, and the best-so-far solution T^{bs}. The influence of each solution on the pheromone update depends on the state of convergence of the algorithm as measured by the convergence factor *cf*. Each pheromone value $\tau_{ij} \in \mathcal{T}$ is updated as follows:

$$\tau_{ij} := \tau_{ij} + \rho \cdot (\xi_{ij} - \tau_{ij}) \ , \tag{11}$$

where

$$\xi_{ij} := \kappa_{pbs} \cdot T^{pbs}_{ij} + \kappa_{rb} \cdot T^{rb}_{ij} + \kappa_{bs} \cdot T^{bs}_{ij} \ , \tag{12}$$

where κ_{pbs} is the weight (that is, the influence) of solution T^{pbs}, κ_{rb} is the weight of solution T^{rb}, κ_{bs} is the weight of solution T^{bs}, and $\kappa_{pbs} + \kappa_{rb} + \kappa_{bs} = 1$. After the pheromone update rule (Equation 11) is applied, pheromone values that exceed $\tau_{\max} = 0.99$ are set back to $\tau_{\max}$ (similarly for $\tau_{\min} = 0.001$). This is done in order to avoid a complete convergence of the algorithm, which is a situation that should be avoided. Equation 12 allows to choose how to schedule the relative influence of the three solutions used for updating the pheromone values. For our application we used a standard update schedule as shown in Table 1.

ComputeConvergenceFactor($\mathcal{T}$): The convergence factor cf, which is a function of the current pheromone values, is computed as follows:

$$cf := 2\left(\left(\frac{\sum\limits_{\tau_{ij} \in \mathcal{T}} \max\{\tau_{\max} - \tau_{ij}, \tau_{ij} - \tau_{\min}\}}{|\mathcal{T}| \cdot (\tau_{\max} - \tau_{\min})}\right) - 0.5\right)$$

In this way, $cf = 0$ when the algorithm is initialized (or reset), that is, when all pheromone values are set to 0.5. On the other side, when the algorithm has converged, then $cf = 1$. In all other cases, cf has a value in $(0, 1)$. This completes the description of our Beam-ACO approach for the LCS problem.

Experimental results

We implemented algorithms MS-PBS(α,μ) and Beam-ACO in ANSI C++ using GCC 3.2.2 for compiling the software. The experimental results that we outline in the following were obtained on a PC with an AMD64X2 4400 processor and 4 Gb of memory. We applied algorithm MS-PBS(α,μ) with three different settings:

1. $\alpha = \mu = 1$: The resulting algorithm corresponds to a multi-start version of algorithm SC($\hat{f}$); see Algorithm 1. In the following we refer to this algorithm by **MS-SC**.
2. $\alpha = 10$, $\mu = 1$: This setting corresponds to a multi-start version of algorithm PSC(α,$\hat{f}$); see Algorithm 2. We refer henceforth to this algorithm by **MS-PSC**.
3. $\alpha = 10$, $\mu > 1$: These settings generate a multi-start version of algorithm PBS(α,μ,$\hat{f}$); see Algorithm 3. This algorithm version is referred to simply by **MS-PBS**. Note that we made the setting of μ depended on the alphabet size, that is, the number of expected children of a partial solution.

In addition we applied Beam-ACO with $\alpha = 10$ and with the same settings for μ as chosen for MS-PBS.

For the experimentation we used a set of benchmark instances that was generated as follows. Given $h \in \{100, 200, \ldots, 1000\}$ and Σ (where $|\Sigma| \in \{2, 4, 8, 24\}$), an instance is produced as follows. First, a string s of length h is produced randomly from the alphabet Σ. String s is in the following called *base string*. Each instance contains 10 strings. Each of these strings is produced from the base string s by traversing s and by deciding for each letter with a probabilitiy of 0.1 whether to remove it, or not. Note that the 10 strings of such an instance are not necessarily of the same length. As we produced 10 instances for each combination of h and $|\Sigma|$, 400 instances were generated in total. Note that the values of optimal solutions of these instances are unknown. However, a lower bound is obtained as follows. While producing the 10 strings of an instance, we record for each position of the base string s, whether the letter at that position was removed for the generation of at least one of the 10 strings. The number of positions in s that were never removed constitutes the lower bound value henceforth denoted by LB_I with respect to an instance I.

We applied each of the 4 algorithms exactly once for $h/10$ seconds to each problem instance. We present the results averaged over the 10 instances for each combination of h (the length of the base string that was used to produce an instance), and the alphabet size $|\Sigma|$. Two measures are presented:

1. The (average) length of the solutions expressed in deviation (percentage) from the respective lower bounds, which is computed as follows:

$$\left(\frac{f}{\mathrm{LB}_I} - 1\right) \cdot 100 \ , \tag{13}$$

where f is the length of the solution achieved by the respective algorithm.

2. The computation time of the algorithms, which refers to the time the best solution was found within given CPU time (averaged over the 10 instances of each type).

The results are shown graphically in Figure 3. The graphics on the left hand side show the algorithm performance (in percentage of deviation from the lower bound), and the graphics on the right hand side show the computation times. The following observations are of interest. First, while having a comparable computation time, algorithm MS-PBS is always clearly better than algorithms MS-PSC and MS-SC. Second, algorithm Beam-ACO is consistently the best algorithm of the comparison. This shows that it can pay off adding a learning component to algorithm (MS-)PBS. The advantage of Beam-ACO over MS-PBS grows with growing alphabet size, that is, with growing problem complexity. This advantage of Beam-ACO comes with a slight increase in computational cost. However, this is natural: due to the learning component, Beam-ACO has a higher probability than MS-PBS of improving on the best solution found even at late stages of a run. Finally, a last interesting remark concerns the comparison of MS-PSC with MS-SC. Despite of the construction of solutions in parallel, MS-PSC is always slightly beaten by MS-SC. This is due to fact that the used upper bound function is not tight at all, which results in the fact that constructing solutions in parallel in the way of algorithm (MS-)PSC is rather a waste of computation time.

2 Using metaheuristics concepts within branch & bound

In this section, we present a collaborative technique that integrates a population based metaheuristics, a memetic algorithm (MA) [31, 20, 23], with the beam search variant of the branch & bound procedure. MAs are based on the systematic exploitation of knowledge about the problem being solved, and the synergistic combination of ideas taken from both population-based techniques and trajectory-based metaheuristics. A very common way to achieve this combination is using the template of an evolutionary algorithm, endowing it with local search add-ons. A general sketch of this kind of MA is shown in Algorithm 6. Several things must be noted: firstly, initialization is very often done by means of problem-dependant constructive heuristics, thus ensuring that a good starting point is used for the evolutionary search. Local search can be also used in this initialization stage (to supplement the lack of an adequate constructive heuristic, or to complement the latter). The remaining components of the algorithm are typically chosen so that they incorporate problem-knowledge (if possible) as well.

Algorithm 6 Pseudocode of a memetic algorithm.

```
1: for $i := 1$ to $popsize$ do
2:    $pop[i]$ := HEURISTIC SOLUTION($ProblemData$)
3:    $pop[i]$ := LOCAL SEARCH($pop[i]$)
4:    EVALUATE($pop[i]$)
5: end for
6: while allowed runtime not exceeded do
7:    for $i := 1$ to $offsize$ do
8:       if recombination is performed then
9:          $parent_1$ := SELECT($pop$)
10:         $parent_2$ := SELECT($pop$)
11:         $offspring[i]$ := RECOMBINE($parent_1$,$parent_2$)
12:      else
13:         $offspring[i]$ := SELECT($pop$)
14:      end if
15:      if mutation is performed then
16:         $offspring[i]$ := MUTATE($offspring[i]$)
17:      end if
18:      $offspring[i]$ := LOCAL SEARCH($offspring[i]$)
19:      Evaluate($offspring[i]$)
20:   end for
21:   $pop$ := REPLACE($pop$, $offspring$)
22: end while
```

According to the previous description, it is clear that MAs are specifically concerned with exploiting as much problem-knowledge as available. This makes MAs specifically suited for taking part in hybrid approaches, either integrative or collaborative [34]. In this case, we have considered an approach in which the control flows of BS and MA are intertwined: phases of BS and MA alternate, and both processes share the best known solution. The technique provides the following benefits:

- The best known solution can be used by the beam search part to purge its problem queue, by not expanding partial nodes whose upper bound is worse than the one obtained by the MA.
- The beam search can guide the search of the MA by injecting information about more promising regions of the search space into the MA population.

The resulting algorithm for a minimization problem is pseudo-coded in Algorithm 7. The procedure performs a standard beam search procedure (B_i is used to maintain the *beam* at level i of the search tree and α is the *beam width*, i.e., the maximum number of partial solutions to be expanded at each level). After spreading out each level, if a level dependent problem specific condition is fulfilled (represented in the pseudocode by the *runMA* variable), the MA is run with a population that is initialized using the best nodes (w.r.t some criteria) in the current beam. Note that nodes in the beam are partial solutions, whereas the MA population consists of complete solutions, so a

Algorithm 7 Beam Search and MA Hybrid: Hybrid(α,$\hat{f}$)

```
1: input: $\alpha \in \mathbb{Z}^+$, the best known objective function value $\hat{f}$
2: initialization: $i := 0$, $B_i := \{\epsilon\}$, $z :=$ NULL
3: while $B_i \neq \emptyset$ do
4:   $B_{i+1} := \emptyset$
5:   for $w \in \mathcal{C}(B_i)$ do
6:     if $|\mathcal{C}(w)| > 0$ then
7:       if $\mathrm{LB}(w) < \hat{f}$ then $B_{i+1} := B_{i+1} \cup \{w\}$ end if
8:     else
9:       if $f(w) < \hat{f}$ then $z := w$, $\hat{f} := f(z)$ end if
10:    end if
11:  end for
12:  Restrict $B_{i+1}$ to the (maximally) $\alpha$ best nodes
13:  if $runMA$ then
14:    $pop :=$ select $popsize$ best nodes from $B_{i+1}$
15:    for $j = 1, \ldots, popsize$ do
16:      complete partial solution $pop_j$
17:    end for
18:    $sol :=$ run $\mathrm{MA}_{pop}$
19:    if $f(sol) < \hat{f}$ then $z := sol$, $\hat{f} := f(z)$ end if
20:  end if
21:  $i := i + 1$
22: end while
23: output: $z$ (which might be NULL)
```

problem specific procedure must be used to complete them. After the MA stabilizes, if the solution it provides improves the incumbent one, this one is updated.

Example: Shortest common supersequence (SCS) problem

The Shortest Common Supersequence Problem (SCSP) is a well-known problem in the area of string analysis. Essentially, given a certain alphabet Σ and a set L of strings from Σ, the aim is to find a minimal-length sequence s, such that all strings in the given set L can be *embedded* in s. The SCSP can be shown to be NP$-$hard, even if strong constraints are posed on L, or on Σ. For example, it is NP$-$hard in general when all s_i have length two [38], or when the alphabet size $|\Sigma|$ is two [30]. This combinatorial problem is interesting as it constitutes a formalization of different real-world problems. For example, it has many implications in bioinformatics [19]: it is a problem with a close relationship to multiple sequence alignment [36], and to probe synthesis during microarray production [35]. Besides this, it also has applications in planning [15] and data compression [38], among other fields.

Formally, the notion of embedding can be described as follows. Let s and r be two strings of symbols taken from Σ. String s can be said to embed string

r (denoted as $s \succ r$) using the following recursive definition:

$$\begin{aligned}
s \succ \epsilon \quad &= \text{True} \\
\epsilon \succ r \quad &= \text{False}, \quad \text{if } r \neq \epsilon \\
\alpha s \succ \alpha r \quad &= s \succ r \\
\alpha s \succ \beta r \quad &= s \succ \beta r, \quad \text{if } \alpha \neq \beta
\end{aligned} \tag{14}$$

Plainly, $s \succ r$ means that all symbols in r are present in s in the very same order (although not necessarily consecutive).

Formally, an instance $I = (\Sigma, L)$ for the SCSP is given by a finite alphabet Σ and a set L of m strings $\{s_1, \cdots, s_m\}$, $s_i \in \Sigma^*$. The problem consists of finding a string s of minimal length that embeds each string in L ($s \succ s_i, \forall s_i \in L$ and $|s|$ is minimal).

A branch & bound algorithm to solve an instance $I = (\Sigma, L)$ of the SCSP can start from a single node containing as tentative solution ϵ. In order to implement function $\mathcal{C}(w)$, $|\Sigma|$ subproblems are generated, each of them obtained by appending a symbol from Σ to partial solution w. Nodes with unproductive characters (i.e., not contributing to embedding any string in L) are pruned from the search tree. To obtain a lower bound for a node s^t, the set of remaining strings in L not embedded by s^t must first be calculated as follows:

$$R = \{r_i \mid (s_i^e, r_i) = s^t \gg s_i, s_i \in L\} \tag{15}$$

where $s \gg r = (r^e, r^r)$ if r^e is the longest initial segment of string r embedded by s and r^r is the remaining part of r not embedded by s. Let $M(\alpha, R)$ be the maximum number of occurrences of symbol α in any string in R:[6]

$$M(\alpha, R) = \max\{|r_i|_\alpha \mid r_i \in R\} \tag{16}$$

Clearly, every common supersequence for the remaining strings must contain at least $M(\alpha, R)$ copies of the symbol α. Thus a lower bound can be obtained by summing the length of the tentative solution and the maximum number of occurrences in any string in R of each symbol of the alphabet:

$$LB(s^t) = |s^t| + \sum_{\alpha \in \Sigma} M(\alpha, R) \tag{17}$$

In order to rank nodes in the branch & bound queue, the following quality function was used for each node:

$$\mathit{quality}\ (s^t, L) \ = \ \sum_{s_i \in L} \{\, |s_i^e| \mid (s_i^e, r_i) = s^t \gg s_i\} \tag{18}$$

so that tentative solutions embedding more symbols in L are selected. As all tentative solutions in the same level of the search tree have the same length, the algorithm selects nodes that provide good initial segments for constructing

[6] As in Section 1.6, $|r_i|_\alpha$ denotes the number of occurrences of symbol α in r_i.

Algorithm 8 Majority Merge algorithm.

```
1: input: $L = \{s_1, \cdots, s_m\}$
2: $s := \epsilon$
3: repeat
4:    for $\alpha \in \Sigma$ do
5:       $\nu(\alpha) := \sum_{s_i \in L, s_i = \alpha s'_i} 1$
6:    end for
7:    $\beta \leftarrow \max^{-1}\{\nu(\alpha) \mid \alpha \in \Sigma\}$
8:    for $s_i \in L, s_i = \beta s'_i$ do
9:       $s_i := s'_i$
10:   end for
11:   $s := s\beta$
12: until $\sum_{s_i \in L} |s_i| = 0$
13: output: $s$
```

a short supersequence. Before being injected into the MA population, solutions were randomly completed and repaired using the following function:

$$\begin{array}{lcll} \rho(s, L) & = & s, & \text{if } \forall i : s_i = \epsilon \\ \rho(\alpha s', L) & = & \rho(s', L), & \text{if } \nexists i : s_i = \alpha s'_i \\ \rho(\alpha s', L) & = & \alpha\rho(s', L|_\alpha), & \text{if } \exists i : s_i = \alpha s'_i \\ \rho(\epsilon, L) & = & \mathrm{MM}(L), & \text{if } \exists i : s_i \neq \epsilon \end{array} \tag{19}$$

where MM is the *Majority Merge* algorithm (see Algorithm 8) described in [9]. This is a greedy algorithm that constructs a supersequence incrementally by adding the symbol most frequently found at the front of the strings in L, and removing these symbols from the corresponding strings.

Note that, apart from completing a string in order to have a valid supersequence, this function also removes unproductive steps from the repaired string, acting thus as a local searcher.

Preliminary tests show that partial good solutions were only obtained after descending a substantial number of levels in the beam search tree. This led us to the following strategy for interleaving the MA and the branch & bound in the hybrid algorithm: start by running in isolation the branch & bound part of the algorithm for a initial number of levels, and then periodically interleave both algorithms afterwards. To be precise, an estimation for the SCSP solution s_0 was calculated using the *Weighted Majority Merge* (WMM) algorithm [9] and its length was used to set $l_0 = 0.7 \cdot |s_0|$. The condition for running the MA was $(i > l_0)$ **and** $(i$ **mod** $l = 0)$, where variable i (see Algorithm 7) is the current level explored by the beam search part of the algorithm, and parameter l controls the balance between the MA and beam search, i.e., a execution of the MA is performed every l iterations of the beam search. A sensitivity analysis of the parameters was done in a similar way to that described in [17] and, based on it, the following values were used for the different parameters of the algorithm: $\alpha = 10000$ and $l = 10$.

Algorithm 9 Local search for MA(s, L)

```
1: input: $s \in \Sigma^*$, $L = \{s_1, \cdots, s_m\}$
2: initialization: $k := 1$
3: while $k < |s|$ do
4:     $r := \textsc{Del}_k(s, L)$
5:     if $fit(r, L) < fit(s, L)$ then
6:         $s := r$
7:         $k := 1$
8:     else
9:         $k := k + 1$
10:    end if
11: end while
12: output: $s$
```

As to the MA used, it evolves sequences in $|\Sigma|^\lambda$, where $\lambda = \sum_{s_i \in L} |s_i|$. Before being evaluated, sequences in the population are repaired using the ρ function. After this repairing, raw fitness (to be minimized) is simply computed as:

$$\begin{array}{lcll} fit\,(s, L) & = & 0, & \text{if } \forall i : s_i = \epsilon \\ fit\,(\alpha s', L) & = & 1 + fit(s', L|_\alpha), & \text{if } \exists i : s_i \neq \epsilon \end{array} \qquad (20)$$

An additional local-improvement level is considered. To do so, we have considered the neighborhood defined by the $\textsc{Del}_k : \Sigma^* \times (\Sigma^*)^m \to \Sigma^*$ operation [35]. The functioning of this procedure is as follows:

$$\begin{array}{lcll} \textsc{Del}_k\,(\alpha s, L) & = & \rho(s, L), & \text{if } k = 1 \\ \textsc{Del}_k\,(\alpha s, L) & = & \alpha \textsc{Del}_{k-1}(s, L|_\alpha), & \text{if } k > 1 \end{array} \qquad (21)$$

This operation thus removes the k-th symbol from a string, and then submits it to the repair function so that all strings in L can be embedded. Notice that the repairing function can actually find that the sequence is feasible, hence resulting in a reduction of length by one symbol. A full local-search scheme is defined by iterating this operation until no single deletion results in length reduction (see Algorithm 9). The improvement in solution quality attainable via the application of this LS operator comes obviously at the expenses of an increased computational cost. This additional cost might be too high if LS were massively applied. On the other hand, the extreme option of simply removing LS handicaps the search capabilities of the algorithm. A pragmatic solution can be found in the use of partial lamarckism [21], namely using LS but with some intermediate probability. Preliminary experiments were conducted with probabilities to apply local search in $\{0, 0.01, 0.1, 0.5, 1\}$ (see [12]), and setting this parameter to 0.01 provided a better tradeoff between the attainable improvement, and the additional computational cost implied.

Experimental results

In this section, we do a experimental comparison of the beam search and MA hybrid algorithm with respect to the probabilistic beam search (PBS) algorithm for the SCSP described in [7]. For this purpose, two sets of benchmark instances have been used:

The first one — henceforth referred to as RandomSet — consists of random strings with different alphabet lengths. To be precise, each instance is composed of eight strings, four of them of length 40, and the other four of length 80. Each of these strings is randomly generated, using an alphabet Σ. The benchmark set consists of 5 classes of each 5 instances characterized by different alphabet sizes, namely $|\Sigma| = 2$, 4, 8, 16, and 24. Thus, the benchmark set consists of 25 different problem instances.

A second set of instances — henceforth referred to as RealSet — is composed of strings obtained from molecular sequences, comprising both DNA sequences ($|\Sigma| = 4$) and protein sequences ($|\Sigma| = 20$). In the first case, we have taken two DNA sequences of the SARS coronavirus from a genomic database[7]; these sequences are 158 and 1269 nucleotides long. As to the protein sequences, we have considered four of them, extracted from Swiss-Prot[8]:

- *Oxytocin*: quite important in pregnant women, this protein causes contraction of the smooth muscle of the uterus and of the mammary gland. The sequence is 125-aminoacid long.
- *p53*: this protein is involved in the cell cycle, and acts as tumor suppressor in many tumor types; the sequence is 393-aminoacid long.
- *Estrogen*: involved in the regulation of eukaryotic gene expression, this protein affects cellular proliferation and differentiation; the sequence is 595-aminoacid long.
- *Myelin*: this sequence correspond to a transcription factor of myelin, and is associated with neuronal differentiation. The sequence is 1186-aminoacid long.

Problem instances in RealSet are obtained from the target sequence by removing symbols from the latter with a certain probability $p\%$ ($p \in \{10\%, 15\%, 20\%\}$ in our experiments).

Figure 4 shows results for RandomSet. Results are averaged over 5 independent runs for each problem instance and further averaged over 5 different problem instances with the same alphabet length. For the beam search and MA hybrid, executions were performed on a Pentium IV PC (2400MHz and 512MB of main memory), and a time limit of 600 seconds per execution was imposed. As to the PBS, tests were performed on a AMD64X2 4400 processor and 4 Gb of memory. The time limit was set to 350 seconds, that roughly corresponds to the time given to other algorithm on a different machine. Results show that PBS performs better for this instance set, as it finds better solutions

[7] `http://gel.ym.edu.tw/sars/genomes.html`
[8] `http://www.expasy.org/sprot/`

except for $|\Sigma| = 2$. Note that PBS performs several iterations of the beam search part of the algorithm, and thus exhausts the allowed time, whereas the beam search and MA hybrid only performs a beam search execution, and does not necessarily use all the permitted time. We also studied the performance of a variation of the beam search and MA hybrid (labelled MA-BS 2 in Figure 4) that exhausts the allowed time by performing several iterations of the beam search. In order to introduce more randomness in the algorithm, each time the MA was executed, its population was initialized by selecting nodes from the beam using binary tournament selection. Results show that MA-BS 2 outperforms PBS for $|\Sigma| \in \{2, 24\}$, and is slightly worse for $|\Sigma| = 4$.

Figure 5 shows results for RealSet. In this case, the beam search and MA hybrid performs better, as it always finds the presumed optimal solution in all runs (except for the MYELIN instance with $p\%$=20%). Note that in this latter instance, PBS finds a non-optimal better result. We also make note that the second version of the MA-BS hybrid does not improve these results because the allowed time is exhausted in the first iteration of the beam search; for this reason the results obtained by MA-BS 2 are not shown in Figure 5.

3 Conclusions

In this paper we have dealt with hybridizations of branch & bound derivatives (i.e., beam search) and metaheuristics techniques and have shown that the resulting hybrid algorithms provide better results than their counterparts working alone. Particularly, we have highlighted two different proposals: in the first one, a construction-based metaheuristics is enriched with branch & bound features. Here, we start from a (parallel) solution construction method with a probabilistic component in the election of the next step to execute (i.e., the set of nodes that can be reached from the current state). Then we improve it by incorporating first a beam search component in the election, resulting in a probabilistic beam search algorithm, and second adding a learning component to adjust the knowledge acquired from the accumulated experience.

Our second proposal consists of a branch & bound technique that collaborates in an interleaved way with a metaheuristics, namely a memetic algorithm. Here, the branch & bound technique is used to identify the promising regions of the search space in which the optimal solution can be found. The metaheuristics is then used to exploit this knowledge in order to improve the bounds employed by the branch & bound technique to force further branch pruning.

Our hybrid algorithms have been first described in detail and then applied on practical problems to show their effectiveness. This paper clearly shows that both exact techniques such as branch & bound (including non-complete derivatives such as beam search) and metaheuristics can clearly benefit one from each other.

Acknowledgements

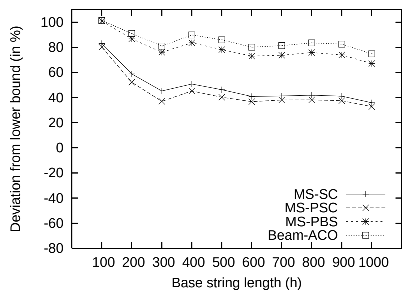
The work of second, third and fourth authors was partially supported by Spanish MCyT under contracts TIC2002-04498-C05-02 and TIN2004-7943-C04-01.

References

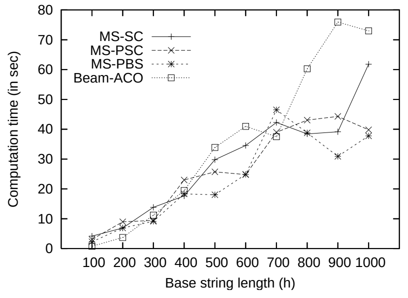
1. A. Aho, J. Hopcroft, and J. Ullman. *Data structures and algorithms.* Addison-Wesley, Reading, MA, 1983.
2. S. Al-Shihabi. Backtracking ant system for the traveling salesman problem. In M. Dorigo, M. Birattari, C. Blum, L. M. Gambardella, F. Mondada, and T. Stützle, editors, *Proceedings of ANTS 2004 – 4th International Workshop on Ant Colony Optimization and Swarm Intelligence*, volume 3172 of *Lecture Notes in Computer Science*, pages 318–325. Springer Verlag, Berlin, 2004.
3. A. Barr and E. Feigenbaum. *Handbook of Artificial Intelligence.* Morgan Kaufmann, New York, 1981.
4. M. J. Blesa and C. Blum. Ant colony optimization for the maximum edge-disjoint paths problem. In G. R. Raidl et al., editor, *Applications of Evolutionary Computing, Proceedings of EvoWorkshops 2004*, volume 3005 of *Lecture Notes in Computer Science*, pages 160–169. Springer Verlag, Berlin, 2004.
5. C. Blum. Beam-ACO–hybridizing ant colony optimization with beam search: an application to open shop scheduling. *Computers and Operations Research*, 32:1565–1591, 2005.
6. C. Blum, J. Bautista, and J. Pereira. Beam-ACO applied to assembly line balancing. In M. Dorigo, L. M. Gambardella, A. Martinoli, R. Poli, and T. Stützle, editors, *Proceedings of ANTS 2006 – Fifth International Workshop on Swarm Intelligence and Ant Algorithms*, volume 2463 of *Lecture Notes in Computer Science*, pages 14–27. Springer Verlag, Berlin, Germany, 2006.
7. C. Blum, C. Cotta, A. J. Fernández, and J. E. Gallardo. A probabilistic beam search algorithm for the shortest common supersequence problem. In C. Cotta et al., editor, *Proceedings of EvoCOP 2007 – Seventh European Conference on Evolutionary Computation in Combinatorial Optimisation*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, 2007. In press.
8. C. Blum and M. Dorigo. The hyper-cube framework for ant colony optimization. *IEEE Transactions on Systems, Man, and Cybernetics – Part B*, 34(2):1161–1172, 2004.
9. J. Branke, M. Middendorf, and F. Schneider. Improved heuristics and a genetic algorithm for finding short supersequences. *OR-Spektrum*, 20:39–45, 1998.
10. S. Casey and J. Thompson. GRASPing the examination scheduling problem. In E. K. Burke and P. De Causmaecker, editors, *Proceedings of PATAT 2002 – 4th International Conference on Practice and Theory of Automated Timetabling*, volume 2740 of *Lecture Notes in Computer Science*, pages 232–246. Springer Verlag, Berlin, 2003.
11. C. Cotta. Protein structure prediction using evolutionary algorithms hybridized with backtracking. In J. Mira and J. R. Álvarez, editors, *Proceedings of the 7th International Work-Conference on Artificial and Natural Neural Networks (IWANN 2003)*, volume 2687 of *Lecture Notes in Computer Science*, pages 321–328. Springer Verlag, Berlin, 2003.

12. C. Cotta. Memetic algorithms with partial lamarckism for the shortest common supersequence problem. In J. Mira and J.R. Álvarez, editors, *Artificial Intelligence and Knowledge Engineering Applications: a Bioinspired Approach*, number 3562 in Lecture Notes in Computer Science, pages 84–91, Berlin Heidelberg, 2005. Springer-Verlag.
13. M. Dorigo and T. Stuetzle. *Ant Colony Optimization*. MIT Press, 2004.
14. T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.
15. D.E. Foulser, M. Li, and Q. Yang. Theory and algorithms for plan merging. *Artificial Intelligence*, 57(2-3):143–181, 1992.
16. C. B. Fraser. *Subsequences and supersequences of strings*. PhD thesis, University of Glasgow, 1995.
17. J.E. Gallardo, C. Cotta, and A.J. Fernández. On the hybridization of memetic algorithms with branch-and-bound techniques. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 37(1):77–83, 2007.
18. D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Computer Science and Computational Biology. Cambridge University Press, Cambridge, 1997.
19. M.T. Hallet. *An integrated complexity analysis of problems from computational biology*. PhD thesis, University of Victoria, 1996.
20. W.E. Hart, N. Krasnogor, and J.E. Smith. *Recent Advances in Memetic Algorithms*. Springer-Verlag, Berlin Heidelberg, 2005.
21. C. Houck, J.A. Joines, M.G. Kay, and J.R. Wilson. Empirical investigation of the benefits of partial lamarckianism. *Evolutionary Computation*, 5(1):31–60, 1997.
22. K. Huang, C. Yang, and K. Tseng. Fast algorithms for finding the common subsequences of multiple sequences. In *Proceedings of the International Computer Symposium*, pages 1006–1011. IEEE press, 2004.
23. N. Krasnogor and J.E. Smith. A tutorial for competent memetic algorithms: model, taxonomy, and design issues. *IEEE Transactions on Evolutionary Computation*, 9(5):474–488, 2005.
24. G. B. Lamont, S. M. Brown, and G. H. Gates Jr. Evolutionary algorithms combined with deterministic search. In V. W. Porto, N. Saravanan, D. E. Waagen, and A. E. Eiben, editors, *Proceedings of Evolutionary Programming VII, 7th International Conference*, volume 1447 of *Lecture Notes in Computer Science*, pages 517–526. Springer Verlag, Berlin, 1998.
25. E. Lawler and D. Wood. Branch and bound methods: A survey. *Operations Research*, 4(4):669–719, 1966.
26. D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25:322–336, 1978.
27. V. Maniezzo. Exact and Approximate Nondeterministic Tree-Search Procedures for the Quadratic Assignment Problem. *INFORMS Journal on Computing*, 11(4):358–369, 1999.
28. V. Maniezzo and A. Carbonaro. An ANTS heuristic for the frequency assignment problem. *Future Generation Computer Systems*, 16:927–935, 2000.
29. V. Maniezzo and M. Milandri. An ant-based framework for very strongly constrained problems. In M. Dorigo, G. Di Caro, and M. Sampels, editors, *Proceedings of ANTS 2002: 3rd International Workshop on Ant Algorithms*, volume 2463 of *Lecture Notes in Computer Science*, pages 222–227. Springer Verlag, Berlin, 2002.

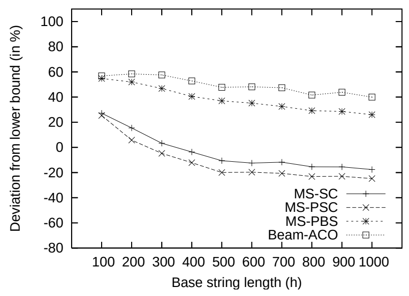
30. M. Middendorf. More on the complexity of common superstring and supersequence problems. *Theoretical Computer Science*, 125:205–228, 1994.
31. P. Moscato and C. Cotta. A gentle introduction to memetic algorithms. In *Handbook of Metaheuristics*, pages 105–144. Kluwer Academic Press, Boston, Massachusetts, USA, 2003.
32. E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job-shop problem. *Management Science*, 42(2):797–813, 1996.
33. P. S. Ow and T. E. Morton. Filtered beam search in scheduling. *International Journal of Production Research*, 26:297–307, 1988.
34. J. Puchinger and G.R. Raidl. Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification. In J. Mira and J.R. Álvarez, editors, *Artificial Intelligence and Knowledge Engineering Applications: a Bioinspired Approach*, number 3562 in Lecture Notes in Computer Science, pages 41–53, Berlin Heidelberg, 2005. Springer-Verlag.
35. S. Rahmann. The shortest common supersequence problem in a microarray production setting. *Bioinformatics*, 19(Suppl. 2):ii156–ii161, 2003.
36. J.S. Sim and K. Park. The consensus string problem for a metric is NP-complete. *Journal of Discrete Algorithms*, 1(1):111–117, 2003.
37. T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.
38. V.G. Timkovsky. Complexity of common subsequence and supersequence problems and related problems. *Cybernetics*, 25:565–580, 1990.



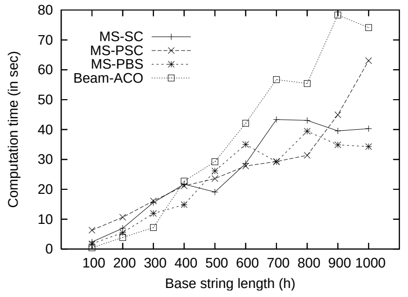
(a) $|\Sigma| = 2$, results



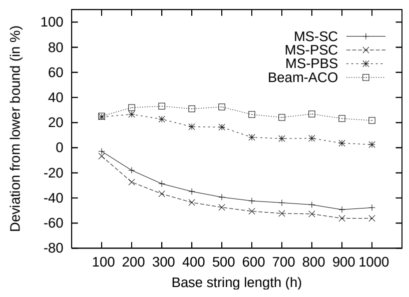
(b) $|\Sigma| = 2$, computation times



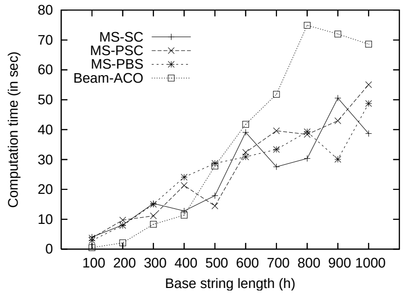
(c) $|\Sigma| = 4$, results



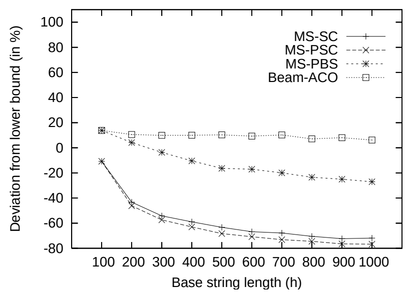
(d) $|\Sigma| = 4$, computation times



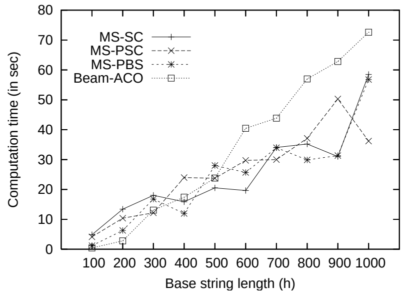
(e) $|\Sigma| = 8$, results



(f) $|\Sigma| = 8$, computation times

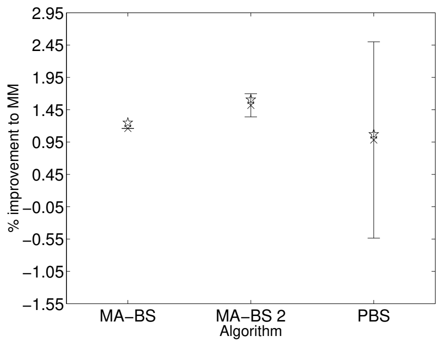


(g) $|\Sigma| = 24$, results

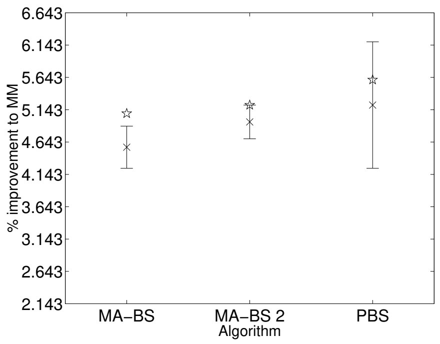


(h) $|\Sigma| = 24$, computation times

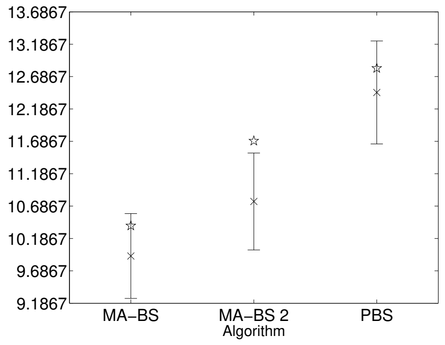
Fig. 3. Results and computation times of algorithms MS-SC, MS-PSC, MS-PBS, and Beam-ACO



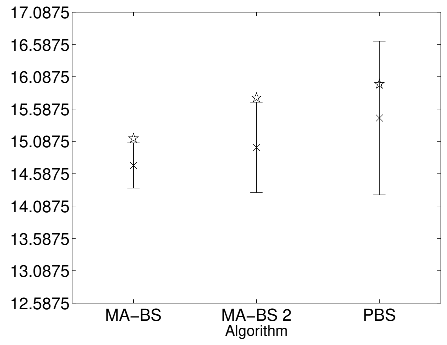
(a) $|\Sigma| = 2$



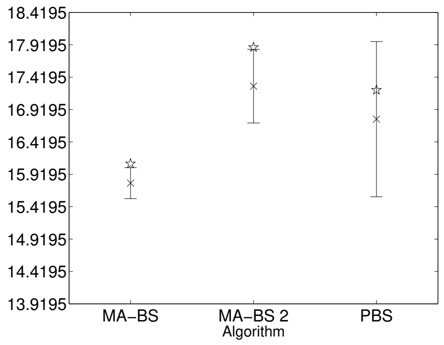
(b) $|\Sigma| = 4$



(c) $|\Sigma| = 8$

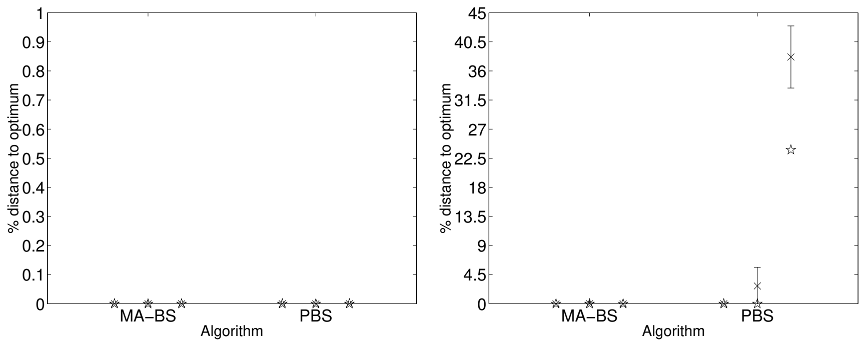


(d) $|\Sigma| = 16$

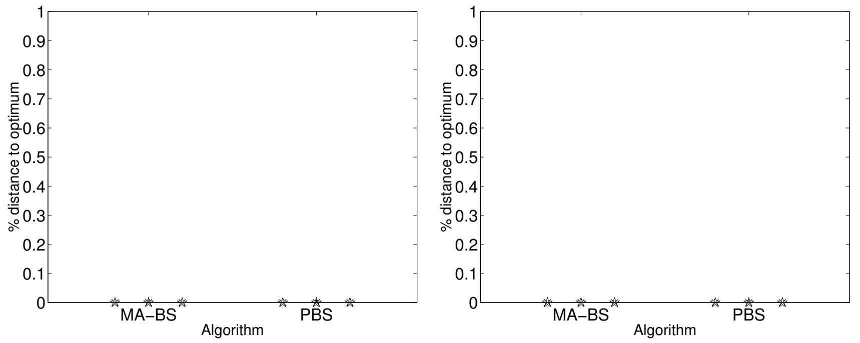


(e) $|\Sigma| = 24$

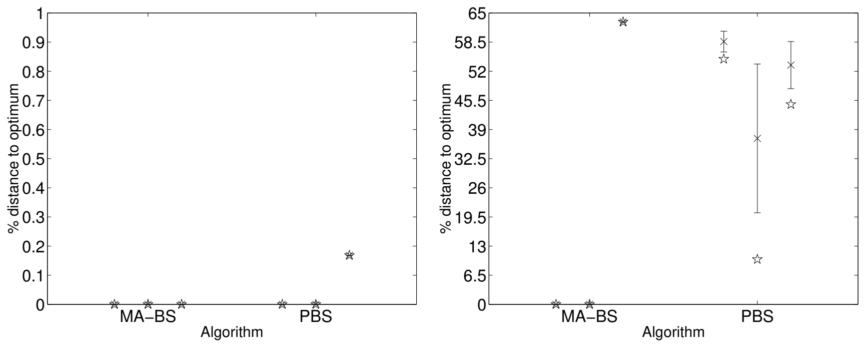
Fig. 4. Comparison of two versions of MA-BS Hybrid Algorithm and PBS on random instances for different alphabet sizes. Figures show relative improvements with respect to solutions provided by MM. A $\times$ sign indicates the mean solution, whereas a $\star$ marks the best solution. Standard deviations of distribution are also depicted.



(a) 158-Nucleotide SARS (b) 1269-Nucleotide SARS



(c) 125-Aminoacid OXYTOCIN (d) 393-Aminoacid P53



(e) 595-Aminoacid ESTROGEN (f) 1186-Aminoacid MYELIN

Fig. 5. Comparison of MA-BS Hybrid Algorithm and PBS on different real instances and gap $\in \{10\%, 15\%, 20\%\}$ (from left to right for each algorithm). Figures show relative distances to optimal solutions. A $\times$ sign indicates the mean solution, whereas a $\star$ marks the best solution. Standard deviations of distribution are also depicted.