Efficient solutions for the Far From Most String Problem

Paola Festa[*]   Panos M. Pardalos[†]

Abstract

Computational molecular biology has emerged as one of the most exciting interdisciplinary fields. It has currently benefited from concepts and theoretical results obtained by different scientific research communities, including genetics, biochemistry, and computer science. In the past few years it has been shown that a large number of molecular biology problems can be formulated as combinatorial optimization problems, including sequence alignment problems, genome rearrangement problems, string selection and comparison problems, and protein structure prediction and recognition. This paper provides a detailed description of string selection and string comparison problems. For finding good-quality solutions of a particular class of string comparison molecular biology problems, known as *the far from most string problem*, we propose new heuristics, including a Greedy Randomized Adaptive Search Procedure (GRASP) and a Genetic Algorithm (GA). Computational results indicate that these randomized heuristics find better quality solutions compared with results produced by the best state-of-the-art heuristic approach.

Keywords: Computational biology, Molecular structure prediction, Protein and sequences alignment, Combinatorial optimization, Metaheuristics.

1 String selection and comparison problems

String selection and comparison problems belong to the more general molecular biology class of problems known as *sequences consensus*, where a finite set of sequences is given and one is interested in finding their consensus, i.e. a new sequence that agrees as much as possible with all the given sequences. In other words, the objective is to determine a sequence called consensus, because it represents in some way all the given sequences.

The idea of representation and of being consensus can be related to several different objectives listed in the following:

[*]Dipartimento di Matematica e Applicazioni, Università degli Studi di Napoli *Federico II*, Italy, E-mail: paola.festa@unina.it

[†]Department of Industrial and Systems Engineering, University of Florida, USA, E-mail: pardalos@ufl.edu

i. the consensus is a new sequence whose total distance from all given sequences is minimum (*closest string problem*);

ii. the consensus is a new sequence whose total distance from all given sequences is maximum (*farthest string problem*);

iii. the consensus is a new sequence far from most of the given sequences (*far from most string problem*).

Many further different objectives can be defined, such as for example the consensus as a *distinguishing substring* that distinguishes the "bad genes sequences" from the "good genes sequences". In this case, a class of specular type of problems is defined as *Distinguishing (Sub)string Selection Problems*.

Linear coding of DNA and proteins. A fundamental remark made by researchers in molecular biology regards the abstraction of the real three-dimensional structure of DNA and its representation as a unidimensional sequence of characters from an alphabet of four symbols. The same type of assumption involves also the protein represented as a sequence of characters from an alphabet of twenty symbols. As a result of the linear coding of DNA and proteins, many molecular biology problems have been formulated as computational and optimization problems involving strings and sequences. Biological applications of computing distance/proximity among sequences occur mainly in two varieties. Some require that a region of similarity be discovered, while other applications use the reverse complement of the region, such as designing probes or primers.

Creating diagnostic probes for bacterial infections. One possible application arises in creating diagnostic probes for bacterial infections. Given a set of DNA sequences from a group of closely related pathogenic bacteria, the task is to find a substring that occurs in each of the bacterial sequences (as close as possible) without occurring in the host's DNA. Probes are then designed to hybridize to these target sequences, so that the detection of their presence indicates that at least one bacterial species is likely to be present in the host.

Discovering potential drug targets. Another biological application related to string selection and comparison problems is related to discovering potential drug targets. Given a set of sequences of orthologous genes from a group of closely related pathogens, and a host (such as human, crop, or livestock), the goal is to find a sequence fragment that is more conserved in all or most of the pathogens sequences but not as conserved in the host. Information encoded by this fragment can then be used for novel antibiotic development or to create a drug that harms several pathogens with minimal effect on the host. All these applications reduce to the task of finding a pattern that with some error occurs in one set of strings (closest string problem) and/or does not occur in another set (farthest string problem). The far from most string problem can help to identify a sequence fragment that distinguishes the pathogens from the host, so the potential exists to create a drug that harms several but not all pathogens.

Computational intractability of the general sequences consensus problem was first proved in 1997 by Frances and Litman [14] and in 1999 by Sim and Park [34]. For most consensus problems, Hamming distance (1) is used instead of any alternative measure (such as for example the editing distance) and biological reasons justifying this choice are very well described and motivated by Lanctot et al. in [24]. Apart from the distance definition, this class of problems are a further important example of molecular biology problems that are in essence combinatorial optimization problems. Subsequent subsections will be devoted to the description of their mathematical formulation and new heuristic approaches will be proposed for the far from the most string problem in Section 2. Finally, in Section 3, computational results are presented which demonstrate empirically the efficiency of the proposed algorithms.

Throughout the remainder of this paper the following notation is used:

- an *alphabet* $\Sigma = \{c_1, c_2, \ldots, c_k\}$ is a finite set of elements, called *characters*;
- $s^i = (s^i_1, s^i_2, \ldots, s^i_m)$ is a sequence of length m ($|s^i| = m$) on Σ ($s^i_j \in \Sigma,\ j = 1, 2, \ldots, m$);
- given two sequences s^i and s^l on Σ such that $|s^i| = |s^l|$, $d_H(s^i, s^l)$ denotes their Hamming distance and is given by

$$d_H(s^i, s^l) = \sum_{j=1}^{|s^i|} \Phi(s^i_j, s^l_j), \tag{1}$$

  where s^i_j and s^l_j are the characters in position j in s^i and s^l, respectively, and $\Phi : \Sigma \times \Sigma \to \{0, 1\}$ is the predicate function such that

$$\Phi(a, b) = \begin{cases} 0, & \text{if } a = b; \\ 1, & \text{otherwise;} \end{cases}$$

- given a set of sequences $\Omega = \{s^1, s^2, \ldots, s^n\}$ on Σ ($s^i \in \Sigma^m$, $i = 1, 2, \ldots, n$) d^{Ω}_H denotes the Hamming distance among the sequences in Ω and it is given by

$$0 \quad \leq \quad d^{\Omega}_H = \min_{i,l=1,\ldots,n \ | \ i<l} d_H(s^i, s^l) \quad \leq \quad m. \tag{2}$$

1.1 The closest string problem (CSP)

Given a finite set of sequences $\Omega = \{s^1, s^2, \ldots, s^n\}$ on Σ ($s^i \in \Sigma^m$, $i = 1, 2, \ldots, n$), the closest string problem (CSP) is to find a *center string* $t \in \Sigma^m$ such that the Hamming distance between t and all other sequences in Ω is minimal; in other words, t is a sequence to which a minimal value d corresponds such that

$$d_H(t, s^i) \leq d, \quad \forall\ s^i \in \Omega,$$

or, equivalently,

$$d^{\Omega \cup \{t\}}_H = d.$$

```
algorithm lanctot-et-al($\{s^i\}_{i=1,2,\ldots,n}$)
1 $\Omega := \{s^i\}_{i=1,2,\ldots,n}$;
2 return $(s^1)$;
end lanctot-et-al
```

Figure 1: 2-approximation algorithm for the CSP.

The CSP can be formulated as an integer program. In fact, let V_k be the set of characters appearing in position k in the sequences in Ω.

For each $k = 1, 2, \ldots, m$ and $j \in V_k$, let us define the following binary variables:

$$x_{jk} = \begin{cases} 1, & \text{if } j\text{-th character in } V_k \text{ is used in position } k; \\ 0, & \text{otherwise.} \end{cases}$$

Then, the CSP admits the following mathematical programming formulation:

$$\begin{array}{lll} \min & d & \\ \text{s.t.} & & \\ & \displaystyle\sum_{j \in V_k} x_{jk} = 1, & k = 1, 2, \ldots, m \qquad (1) \\ & \displaystyle m - \sum_{j=1}^{m} x_{s^i_j j} \le d, & i = 1, 2, \ldots, n \qquad (2) \\ & d \in N^+, \; x_{jk} \in \{0, 1\}, & k = 1, 2, \ldots, m, \; \forall\, j \in V_k. \qquad (3) \end{array}$$

Equalities (1) guarantee that only one character in each set V_k, $k = 1, 2, \ldots, m$, is selected. Inequalities (2) guarantee that if a character s^i_j is not in a solution t, then that character has to contribute to increasing the Hamming distance from s^i to t. Constraints (3) force d to a nonnegative integer value and decision variables to binary values.

There are several combinatorial problems related sequences consensus problems. For example, the *hitting string problem* [15] is similar to the CSP. Given a set S of strings of length n over the alphabet $\{0, 1, *\}$, the hitting string problem consists in finding a string over the alphabet $\{0, 1\}$ that has at least one match with each string in S. This problem was proved to be NP-complete in 1974 by Fagin [7] and is a special case of the CSP in which the Hamming distance bound is $n-1$ and the sought string lies over $\{0, 1\}$ rather than over the original alphabet $\{0, 1, *\}$.

The CSP was first studied in the area of coding theory [33] and recently has been independently proved computationally intractable by Frances and Litman [14] and Lanctot et al. [24, 25].

Optimal solutions have been found via integer programming [30], while the first approximation algorithm for the CSP was proposed by Lanctot et al. in [25] with a worst case performance ratio of 2. It is a simple algorithm that constructs an approximate feasible solution in a pure random fashion. Starting from an empty solution, the algorithm selects at random the next element to be added to the solution under construction (see Figure 1).

Better performance approximation algorithms proposed in the literature are based on linear programming relaxation of the previous integer programming model. The basic idea consists in formulating the problem as the integer program above described, solving its linear programming relaxation, and using the result of the relaxed problem to find an approximate solution to the original problem. Following this line, Lanctot et al. in [25] also proposed a $\frac{4}{3}(1+\epsilon)$-approximation algorithm (for any small $\epsilon > 0$) that uses the randomized rounding technique for obtaining an integer 0-1 solution from the continuous solution for the relaxed problem. The randomized rounding technique works by defining the value of a Boolean variable $x \in \{0,1\}$ to be $x = 1$ with a certain probability y, where y is the value of the continuous variable corresponding to x in the relaxation of the original integer programming problem. In 1999, Li et al. in [26] used the rounding idea to design a Polynomial Time Approximation Scheme (PTAS). Very few problems admit a PTAS and it has been shown that if $P \neq NP$, then there does not exist a PTAS for many problems, such as for example the minimum bin packing problem (see, among others, Chapter 3 of [2]). A PTAS is a polynomial time algorithm - or a family of such algorithms - that, for each fixed $\epsilon > 1$, can approximate the problem within a factor $1 + \frac{1}{\epsilon}$. The running time could depend upon ϵ, but for each fixed ϵ it has to be polynomial in the input size. The PTAS proposed in [26] is based on randomized rounding, that here is refined to check results for a large (but polynomially bounded) number of subsets of indices. However, since a large number of iterations involves the solution of a linear relaxation of an integer program, the algorithm becomes impractical for any instance with large strings.

Recent literature includes several parallel algorithms [8, 19], a network flow based algorithm for the longest common subsequence problem [29], and a specialized branching [5]. A specialized branching combines the good characteristics of dynamic programming and branch and bound in order to reduce the search space. In fact, in [5] the authors showed that their approach works quite well when $n \geq 6$ and outperforms both dynamic programming (that works well for $n = 2$ or $n = 3$) and standard branch and bound, that usually builds a binary branching tree instead of a $|\Sigma|$ branching tree.

1.2 The farthest string problem (FSP)

Given a finite set of sequences $\Omega = \{s^1, s^2, \dots, s^n\}$, a problem complementary to the CSP is the farthest string problem (FSP) that consists in finding a string $t \in \Sigma^m$ farthest from Ω. This type of problem can be useful in situations such as finding a genetic sequence that cannot be associated to a given number of species.

Like the CSP, the FSP can be formulated mathematically as the following integer program:

$$\begin{array}{ll} \max & d \\ \text{s.t.} & \\ & \displaystyle\sum_{j \in V_k} x_{jk} = 1, \qquad\qquad k = 1, 2, \ldots, m \\ & \displaystyle m - \sum_{j=1}^{m} x_{s^i_j j} \geq d, \qquad i = 1, 2, \ldots, n \\ & d \in N^+,\ x_{jk} \in \{0, 1\}, \quad k = 1, 2, \ldots, m,\ \forall\, j \in V_k. \end{array}$$

By polynomial time reduction from the 3-SAT problem, the computational intractability of the FSP was demonstrated in 2003 in [25] by Lanctot et al., who proved that the problem remains computationally intractable even for the simplest case where the alphabet has only two characters.

Despite its inherent computational intractability, it has be shown in [25] that there is a PTAS for the FSP. The algorithm is based on the randomized rounding of the relaxed solution of the above reported integer programming formulation and uses the randomized rounding technique together with probabilistic inequalities to determine the maximum error possible in the solution computed by the algorithm. Note that, the mathematical formulation of FSP is quite similar to the one used for the CSP, with only a change in the optimization objective, and the inequality sign in the constraint

$$m - \sum_{j=1}^{m} x_{s^i_j j} \geq d, \qquad i = 1, 2, \ldots, n.$$

Thus, to solve the problem using an integer programming formulation one can use similar techniques to that employed for the CSP.

1.3 The far from most string problem (FFMSP)

A problem closely related to the FSP is the far from most string problem (FFMSP). It consists in determining a string far from most of the strings in the input set Ω. This can be formalized by saying that given a threshold t, a string s must be found maximizing the variable x such that

$$d_H(s, s^i) \geq t, \text{ for } s^i \in P \subseteq \Omega \text{ and } |P| = x,$$

or, equivalently

$$d_H^{P \cup \{s\}} \geq t, \text{ for } P \subseteq \Omega \text{ and } |P| = x.$$

Despite the similarity with the FSP, it can be shown [25] that the FFMSP is much harder than FSP to approximate, due to the approximation preserving reduction to FFMSP from the Independent Set Problem, a classical and computationally intractable combinatorial optimization problem. In more detail, Lanctot et al. [25] proved that for strings over an alphabet Σ with $|\Sigma| \geq 3$, approximating FFMSP within a polynomial factor is NP-hard.

```
algorithm iter-impr ($c$,$x$,$\mathcal{N}$)
    $NoLocal := true$;
    while ($NoLocal$) do
        /* exploration of the neighborhood $N(x)$ */
        if ($\exists\ \bar{x} \in N(x):\ c'\bar{x} < c'x,\ c'\bar{x} \le c'y,\ \forall\ y \in N(x)$) then
            $x := \bar{x}$;
        else $NoLocal := false$;
        end;
    end;
    return ($x$);
end iter-impr
```

Figure 2: Local search procedure `iter-impr`.

Given theoretical computational hardness results, polynomial time algorithms can yield only solutions with no constant guarantee of approximation. In such cases, as discussed in the next section, heuristic methods can be useful in determining good solutions at least for a class of instances.

2 Heuristic algorithms for the FFMSP

A simple heuristic for the FFMSP has been recently proposed by Pardalos et al. in 2005 [28]. It consists of the following two phases.

- A *construction phase*, that iteratively builds a feasible solution s.

  Initially,

  - for each position j, compute the set V_j of characters appearing in that position in any of the strings in Ω;
  - for each character $c \in V_j$, compute the number of times that c appears in the input on position j.

  Then, for each position $j \in \{1, \ldots, m\}$ a string s is iteratively built by choosing the character in V_j that appears in the smallest number of strings.

  For each position $j > 1$, check the effect that assigning a character to this position will have on previous assignments.

- A *local search phase*, that starting from s explores a suitably defined *neighborhood* of s (a set of feasible solutions "close" to s) until a local optimum is found.

The simplest local search procedure is known as *iterative improvement* and its pseudo-code for a minimization problem is shown in Figure 2. Given a cost vector c and a defined a neighborhood function N, starting from an initial solution x the iterative improvement procedure explores $N(x)$ looking for a better solution $\overline{x}$. If such a solution exists, then the search continues from $\overline{x}$;

```
algorithm GRASP(MaxIterations)
1    for $i = 1, \ldots,$ MaxIterations do
2        Build a greedy randomized solution $x$;
3        $x :=$ LocalSearch($x$);
4        if $i = 1$ then $x^* := x$;
5        else if $c'x < c'x^*$ then $x^* := x$;
6    end;
7    return ($x^*$);
end GRASP
```

Figure 3: Pseudo-code of a generic GRASP for a minimization problem.

otherwise, the procedure provides as output the current solution x which is locally optimal with respect to the defined neighborhood.

Pardalos et al. [28] proposed a 2-*exchange* local search procedure, whose basic step consists in randomly selecting a position j and changing it to another character in V_j (at random). The authors show that it empirically improves the solution by making it a local optimum relative to this type of transformation.

2.1 A GRASP for the FFMSP

Recently, in [11] a GRASP (Greedy Randomized Adaptive Search Procedure) have been proposed to find suboptimal solutions for the FFMSP.

GRASP is an iterative multi-start heuristic algorithm, successfully applied to find good quality solutions to several computationally intractable combinatorial problems and originally proposed in the literature by Feo and Resende [9, 10]. For a comprehensive study of GRASP strategies and variants, the reader is referred to the survey chapter by Resende and Ribeiro [31], as well as to the annotated bibliography of Festa and Resende [12] for a survey of applications.

Generally speaking, GRASP is a randomized heuristic method that for a certain number of iterations realizes two phases: a construction phase and a local search phase. The construction phase iteratively adds one element at a time to a set that ends up with a representation of a feasible solution. At each iteration, an element is randomly selected from a *restricted candidate list* (RCL), whose elements are among the best ordered, according to some greedy function that measures the (myopic) benefit of selecting each element. Once a feasible solution is obtained, the local search procedure attempts to improve it by producing a locally optimal solution with respect to some suitably defined neighborhood structure. Construction and local search phases are repeatedly applied until stopping criterion is met and the best solution found is returned as output. Figure 3 depicts the pseudo-code of a generic GRASP heuristic for a minimization problem.

The construction phase makes use of an adaptive greedy function, a construction mechanism for the restricted candidate list, and a probabilistic selection criterion. The greedy function takes into account the contribution to the objective function achieved by selecting a particular element. In the case of the far from most string problem, it is intuitive to relate the greedy function to the occurrence of each character in a given position. In fact, as in [28], for each position j compute the set V_j of characters appearing in that position in any of the strings in Ω and then, for each character $c \in V_j$, compute $g_j(c)$ as the number of times that c appears in the input on position j. Starting from an empty solution, at each construction iteration the choice of the next element to be added to the partial solution is determined by ordering all candidate characters in a candidate list C with respect to the above defined greedy function. The probabilistic component of the GRASP here proposed is characterized by *randomly* choosing one of the best candidates in the list, but not necessarily the top candidate. As in any GRASP heuristic, our construction procedure is *adaptive*, because the benefits associated with every element are updated at each iteration of the construction phase to reflect the changes brought on by the selection of the previous element.

There are several different mechanisms to build the RCL. Typically, it can be limited by the number of elements (cardinality-based criterion) or by their quality (value-based criterion). If the cardinality-based criterion is chosen, then the cardinality of RCL is a priori fixed to some p and the RCL is made up of those elements having the p best greedy function values, while in the value-based case, the cardinality of RCL depends on a threshold parameter $0 \leq \gamma \leq 1$. In our GRASP, at each iteration j of the construction procedure, RCL is formed by all possible candidates y whose greedy function value $g_j(y)$ is better or equal to $\gamma \cdot g_j^*$, where g_j^* is the best greedy function value. Note that, the extreme case $\gamma = 0$ corresponds to a pure greedy strategy, while the extreme case $\gamma = 1$ is equivalent to a completely random strategy.

To realize the local search phase the 2-exchange procedure is used as in [28]. The procedure takes as input the solution $s = (s_1, \ldots, s_m) \in \Sigma^m$ built at the end of the GRASP construction phase and $\{V_j\}_{j=1,\ldots,m}$, where V_j is the set of characters appearing in position j in any of the strings in Ω. Then, for each position $j = 1, \ldots, m$ and for each character $c \in V_j$, $c \neq s_j$, the 2-exchange procedure checks if the solution $\overline{s} = (s_1, \ldots, s_{j-1}, c, s_{j+1}, \ldots, s_m)$ obtained from s exchanging the character in position j is better than s in terms of objective function value. Note that $\overline{s}$ is a neighbor of solution s such that $d_H(s, \overline{s}) = 1$.

A local search may be implemented using either a *best improving* or a *first improving strategy*. The first improving strategy stops the current iteration as soon as an improving neighbor is found, while in the best improving strategy all neighbors are evaluated and the best among them is kept as new current solution from which to start the next iteration. As reported in Section 3, we implemented both strategies and noticed that the best improving produces better quality solutions for most of the problem instances but in a higher amount of time as compared to the first improving strategy.

```
algorithm GA
1   $t := 0$;
2   Initialize population $P(0)$;
3   Evaluate $P(0)$;
4   while not termination-criteria do
5       $t := t + 1$;
6       Select $P(t)$ from $P(t-1)$;
7       Alter $P(t)$;
8       Evaluate $P(t)$;
9   end;
end GA
```

Figure 4: Pseudo-code of a generic genetic algorithm.

2.2 A genetic algorithm for the FFMSP

Rooting in the mechanisms of evolution and natural genetics and deriving from the principles of natural selection and Darwin's evolutionary theory, the interest in heuristic search algorithms began in the 1970s, when Holland [21] first proposed genetic algorithms. This type of evolutionary technique is meta-heuristic algorithm that is not guaranteed to find an optimal solution. Nevertheless, in many research papers and text book (see, for example [6, 17, 18]) the authors define the evolution belonging to the algorithm as a Markov chain and find conditions implying that the evolution finds an optimum with high probability.

In nature, competition among individuals results in the fittest individuals surviving and reproducing. This is a natural phenomenon called *the survival of the fittest*: the genes of the fittest survive, while the genes of weaker individuals die out. The reproduction process generates diversity in the gene pool. Evolution is initiated when the genetic material (chromosomes) from two parents recombines during reproduction. The exchange of genetic material among chromosomes is called *crossover* and can generate good combination of genes for better individuals. Another natural phenomenon called *mutation* causes regenerating lost genetic material. Repeated selection, mutation, and crossover cause the continuous evolution of the gene pool and the generation of individuals that survive better in a competitive environment.

In complete analogy with nature, once encoded each possible point in the search space of the problem into a suitable representation, a GA transforms a population of individual solutions, each with an associated *fitness* (or objective function value), into a new generation of the population. By applying genetic operators, such as crossover and mutation [23], a GA successively tries to produce better approximations to the solution. At each iteration, a new generation of approximations is created by the process of selection and reproduction. In Figure 4 a simple genetic algorithm is described by the pseudo-code, where $P(t)$ is the population at iteration t.

The GA framework has been applied to find good quality solutions to several computationally intractable combinatorial problems (see, for example [4, 13, 20, 27, 35, 36]). In solving a given optimization problem $\mathcal{P}$, a GA consists of the following basic steps.

1. Randomly create an initial population $P(0)$ of individuals, i.e. solutions for $\mathcal{P}$.
2. Iteratively perform the following substeps on the current generation of the population until the termination criterion has been satisfied.
   (a) Assign fitness value to each individual using the fitness function.
   (b) Select parents to mate.
   (c) Create children from selected parents by crossover and mutation.
   (d) Identify the best-so-far individual for this iteration of the GA.

Next, we describe how the above principles were tailored to produce a genetic algorithm for the FFMSP.

Representation. In designing a GA it is generally not easy to choose a suitable representation of a solution that leads the genetic operators produce feasible offsprings. Fortunately, for the FFMSP it can be easily established a 1-to-1 correspondence between the alphabet $\Sigma = \{c_1, c_2, \ldots, c_k\}$ and the discrete interval $\hat{\Sigma} = [1; k]$. A solution sequence $s = (s_1, s_2, \ldots, s_m)$ is represented by a point in the discrete search space $[1; k]^m$ and all points in the search space represent feasible solutions.

Initial population. The initial population is generated by randomly choosing feasible points in the search space $[1; k]^m$, represented as integer vectors. In more detail, we have divided the continuous interval $[0; 1]$ in k subintervals and established a 1-to-1 correspondence between the set of subintervals and $\hat{\Sigma}$. Then, for generating each component s^i_j $(j = 1, \ldots, m)$ of each individual $s^i \in P(0)$ we pick one of the k interval at random. For example, let us suppose that $\Sigma = \{a, b, c, d\}$, $\hat{\Sigma} = \{1, 2, 3, 4\}$, and that $\lambda_1 = [0, 0.25)$, $\lambda_2 = [0.25, 0.50)$, $\lambda_3 = [0.50, 0.75)$, $\lambda_4 = [0.75, 1]$ are the four subintervals corresponding to $\hat{\Sigma}$. Then, for each $s^i \in P(0)$ and for each $j = 1, \ldots, m$, we pick at random $\xi \in [0; 1]$ and set $s^i_j = q$, if $\xi \in \lambda_q$.

Evaluation function. The association of each solution to a fitness value is done through the fitness function. We associate a cost to each individual through the Hamming distance cost function (1). The evaluation function is computationally cheap, as the Hamming distance is evaluated in $O(m)$, if in the worst case we need to compute it from scratch.

```
procedure GA-crossover($m$, $r$, $e$, $ne$, $K$)
1    for each $i = 1, \ldots, m$ do
2        if $r_i < K$ then $c_i := e_i$;
3        else $c_i := ne_i$;
4        end;
5    end;
6    return ($c$);
end GA-crossover
```

Figure 5: Pseudo-code of the GA crossover procedure.

Population partitioning. After sorting the individuals according to their fitness values, the population is divided into three classes. The top $\alpha \times 100\%$ (class A) is called the upper class, or elite. The next $\beta \times 100\%$ (class B) is called the middle class. The remaining population (class C) constitutes the lower class.

Parent selection. Our GA uses a combined selection method from Hollstein's selection methods [22]. It combines family selection and individual selection, so that it randomly chooses one parent from the elite class (class A) and the other parent from a non-elite class (either class B or C). Using the family size control parameters α and β, one can control the size of the classes and therefore the balance of convergence and diversity as well. To create the next generation, our GA promotes all class A solutions without change, replaces all class C solutions by randomly generated solutions (as done for generating the initial population), and chooses $\beta \times PopSize$ pairs of parents, as described above, where $PopSize$ is the size of the population.

As done by Koza et al. in [23], the selection method reflects the following intuitive principles.

- Better individuals are more likely to reproduce.
- Re-selection is allowed as better individuals can be selected for breeding more than once.
- Selection is probabilistic. This way, the selection process will do a significant amount of hill climbing, but it is not entirely greedy.

Crossover. The target of crossover is to cross and combine two parent solutions e (elite) and ne (non-elite). Our GA applies the *random keys* procedure proposed in 1994 by Bean [3].

It generates a random m-vector r of real numbers between 0 and 1 and chooses a cutoff real number $K \in [0.5; 1]$, which determines if a gene of a child solution c is inherited from e or ne, as depicted in Figure 5.

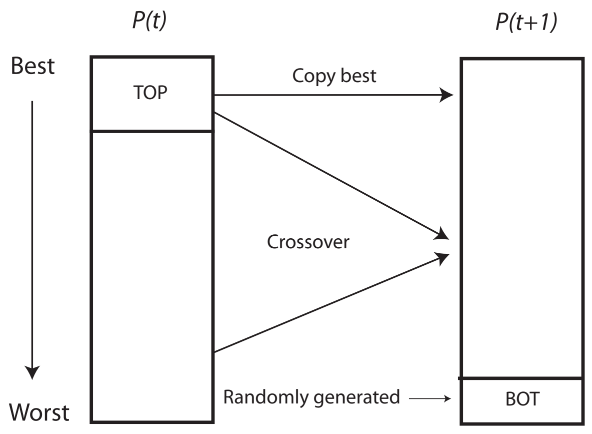


Figure 6: Selection of population $P(t+1)$ from population $P(t)$.

Next population selection. Population $P(t)$ at iteration t is sorted from best to worst individual (see Figure 6). Top individuals (TOP) from $P(t)$ are copied unchanged to next population $P(t+1)$. Bottom individuals (BOT) in $P(t)$ are replaced by randomly generated individuals in $P(t+1)$. The remaining individuals of $P(t+1)$ are generated by applying crossover operator to randomly selected individual from TOP individuals of current population $P(t)$ and randomly selected individual from entire current population.

3 Computational Results

In this section computational results are presented which demonstrate empirically that the proposed GRASP and Genetic Algorithm result in better quality solutions, compared with the heuristic of Pardalos et al. [28]. All the algorithms have been implemented in the C language, using the gcc compiler, and run on Linux (Kubuntu 7.10), Intel Core2 Duo T520 1.50 GHz RAM 2GB. Problem instances used in the computational tests were generated at random, following the computational experience realized by Pardalos et al. [28].

The primary structure of a protein is a linear chain of 20 amino acids (denoted by A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, and V), while DNA molecules are chains of 4 different types of nucleotides (denoted by A, T, G, C). Therefore, since both proteins and DNA molecules can be represented as strings of letters from relatively small alphabets, for our random problem instances we used an alphabet of 4 characters. The general characteristics of the test problems are listed in the following:

- the alphabet Σ is a set of four characters;
- the sequence length m ranges from 300 to 800;

- the number n of sequences in Ω ranges from 100 to 300;
- the threshold t varies from $75\% \times m$ to $85\% \times m$ for each problem size.

As underlined in Section 2.1, any local search may be implemented using either a best improving or a first improving strategy. We implemented both a GRASP making use in the local search of the best improving strategy and a GRASP using the first improving strategy. The GRASP with best improving implementation is called `Grasp Best Improvement`, while the GRASP with first improving implementation is called `Grasp First Improvement`. The GRASP's stopping criterion is related to the parameter `MaxIterations` denoting the number of iteration to be performed. In our experiments, we set `MaxIterations` = 2000.

About the Genetic Algorithm, we called its implementation `Genetic Algorithm`. It requires that the following five parameters are specified:

- *PopSize* denotes the size of the population and strongly affects the running time of the Genetic Algorithm. We ran tests with high population size setting $PopSize = n$ for each problem size.
- $MAXGEN$ denotes the number of generations and Genetic Algorithm uses this parameter as a stopping criterion. In the experiments, we set $MAXGEN = 2000$ coherently with GRASP implementation. Nevertheless, as we note later in this section, Genetic Algorithm finds good solutions after fewer generations.
- The parameter α is the proportion of the population of solutions that are elite, or upper class. In the experiments, we set $\alpha = 0.15$.
- The parameter β is the proportion of the population of solutions that are middle class. In the experiments, we set $\beta = 0.8$. Consequently, the proportion of lower class solutions is $1 - \alpha - \beta = 0.05$.
- K is the crossover cutoff value. This is the probability that the child solution inherits the gene from its elite parent. We found that a cutoff value of 0.7 resulted in a good balance between convergence and diversity.

We next present the computational results for the test problems and the four heuristics. For each problem size, the algorithms were run for 200 random instances and an average solution value was computed. The results obtained are summarized in Tables 1–3, where for each problem type, in the first column the instance size (m, n, and t) is reported, the remaining columns report average objective function values obtained by applying Pardalos et al.'s heuristic, `Grasp First Improvement`, `Grasp Best Improvement`, and `Genetic Algorithm`, respectively. The last column shows the percentage improvement of the average last generation fitness values produced by `Genetic Algorithm` to the average objective function values obtained with `Grasp Best Improvement`, i.e.

$$100 \times \frac{|z(\texttt{Genetic Algorithm}) - z(\texttt{Grasp Best Improvement})|}{z(\texttt{Grasp Best Improvement})}.$$

Table 1: For each problem of size n =100 and $m \in \{300, 600, 800\}$, the table lists instance size, average objective function values z found by the tested heuristics, and average running times (in seconds). The last column shows the percentage improvement of the average last generation fitness values produced by `Genetic Algorithm` to the average objective function values obtained with `Grasp Best Improvement`.

	Pardalos		Grasp First		Grasp Best		Genetic alg.		Impr. (%)
Instances	z	time	z	time	z	time	z	time	
n =100, m =300, t =225	99	0.002	100	3.538	100	3.564	100	11.594	0
n =100, m =300, t =234	85	0	86	3.542	87	3.548	90	13.714	3,448
n =100, m =300, t =240	59	0	59	3.542	59	3.536	64	14.198	8,474
n =100, m =300, t =249	16	0	16	3.52	16	3.526	24	18.148	50
n =100, m =300, t =252	5	0	8	3.532	8	3.544	16	20.074	100
n =100, m =300, t =255	3	0	4	3.524	4	3.54	11	21.532	175
n =100, m =600, t =468	92	0.01	95	7.092	95	7.152	97	28.414	2,107
n =100, m =600, t =480	57	0.01	60	7.072	60	7.116	62	30.012	3,333
n =100, m =600, t =486	33	0.01	34	7.064	35	7.082	40	35.37	14,285
n =100, m =600, t =492	16	0.01	19	7.046	18	7.058	24	40.556	33,333
n =100, m =600, t =498	3	0.006	7	7.054	7	7.044	14	46.852	100
n =100, m =600, t =504	1	0.008	3	7.042	3	7.07	7	62.092	133,333
n =100, m =800, t =624	93	0.01	96	9.468	96	9.588	98	37.946	2,083
n =100, m =800, t =656	10	0.012	14	9.342	14	9.348	20	62.134	42,857
n =100, m =800, t =672	0	0.01	1	9.288	2	9.364	4	132.742	100
n =100, m =800, t =680	0	0.01	0	9.31	0	9.382	1	196.448	100

To make readable the results, for some test problem size in terms of n and m, we plot for each algorithm in Figures 7–9 objective function values (in logarithmic scale) as t varies from $75\% \times m$ to $85\% \times m$. We make the following remarks regarding the computational experiments. First, we observe that the Genetic Algorithm always finds much better quality solutions compared with the other three competitor algorithms. Looking at the last column of Tables 1–3, the percentage improvement is always high, except for the first problem instance in Table 1. As expected, the fitness obtained gets greater, and is thus better, as the instance size increases. In fact, the average percentage improvement is as large as about 54%, 158%, and 257% for the instances in Table 1–3, respectively.

The Genetic Algorithm requires much higher running times. This drawback is due to the high value for the parameter $PopSize$ that we set equal to n for each problem size and to the high value for the parameter $MAXGEN$ that we set $MAXGEN = 2000$ coherently with GRASP implementation. Clearly, Genetic Algorithm needs some further investigation to learn more about its behavior for example running it with moderate population sizes. Nevertheless, during the experiments we noticed that often Genetic Algorithm even in its current implementation finds good solutions after fewer generations than 2000.

To show this characteristic of GA, we plot in Figures 10–12 the empirical distributions of the random variable *time-to-target-solution-value* considering the following three random instances:

1. $n = 100$, $m = 300$, $t = 240$, and target value $\hat{z} = 0.57 \times n$ (Figure 10);

2. $n = 200$, $m = 300$, $t = 240$, and target value $\hat{z} = 0.33 \times n$ (Figure 11);

3. $n = 300$, $m = 300$, $t = 240$, and target value $\hat{z} = 0.25 \times n$ (Figure 12).

We performed 200 independent runs of each heuristic using random number generator seeds 270001, 270002, . . . , and 270200 and recorded the time

Table 2: For each problem of size n =200 and $m \in \{300, 600, 800\}$, the table lists instance size, average objective function values z found by the tested heuristics, and average running times (in seconds). The last column shows the percentage improvement of the average last generation fitness values produced by `Genetic Algorithm` to the average objective function values obtained with `Grasp Best Improvement`.

Instances	Pardalos		Grasp First		Grasp Best		Genetic alg.		Impr. (%)
	z	time	z	time	z	time	z	time	
n =200, m =300, t =225	192	0.012	194	7.084	195	7.094	200	46.67	2,564
n =200, m =300, t =228	184	0.01	185	7.1	185	6.978	196	51.208	5,945
n =200, m =300, t =249	10	0.008	12	6.972	12	6.956	28	73.62	133,333
n =200, m =300, t =252	2	0.008	5	6.954	5	6.954	19	83.592	280
n =200, m =300, t =255	1	0.006	3	6.932	3	6.976	11	90.898	266,333
n =200, m =600, t =462	175	0.02	176	14.188	177	14.074	191	115.506	7,909
n =200, m =600, t =480	48	0.02	51	13.91	52	13.982	73	127.36	40,384
n =200, m =600, t =486	20	0.02	21	14.046	22	13.91	42	153.746	90,909
n =200, m =600, t =492	6	0.02	8	13.914	9	13.948	24	174.186	166,333
n =200, m =600, t =498	1	0.02	3	13.864	4	13.898	13	208.786	225
n =200, m =800, t =608	193	0.024	196	18.81	197	18.948	200	141.638	1,522
n =200, m =800, t =640	40	0.03	42	18.528	43	18.572	64	197.196	48,837
n =200, m =800, t =648	12	0.02	14	18.414	14	18.448	34	227.49	142,857
n =200, m =800, t =656	3	0.024	5	18.37	5	18.408	16	276.522	220
n =200, m =800, t =664	0	0.024	1	18.742	1	18.644	7	399.018	600
n =200, m =800, t =680	0	0.022	1	18.74	2	18.626	8	990.02	300

Table 3: For each problem of size n =300 and $m \in \{300, 600, 800\}$, the table lists instance size, average objective function values z found by the tested heuristics, and average running times (in seconds). The last column shows the percentage improvement of the average last generation fitness values produced by `Genetic Algorithm` to the average objective function values obtained with `Grasp Best Improvement`.

Instances	Pardalos		Grasp First		Grasp Best		Genetic alg.		Impr. (%)
	z	time	z	time	z	time	z	time	
n =300, m =300, t =249	7	0.012	10	10.45	11	10.36	34	172.958	209,090
n =300, m =300, t =252	3	0.012	5	10.458	6	10.368	23	196.322	283,333
n =300, m =300, t =255	0	0.01	2	10.45	2	10.386	13	215.518	550
n =300, m =600, t =450	292	0.03	294	21.036	295	20.91	300	215.844	1,694
n =300, m =600, t =456	275	0.03	276	20.938	277	20.996	295	252.15	6,498
n =300, m =600, t =486	16	0.03	17	20.864	18	20.938	47	361.02	161,111
n =300, m =600, t =492	3	0.03	6	20.966	6	21.056	25	412.222	316,666
n =300, m =600, t =498	0	0.026	2	20.55	2	20.668	11	486.694	450
n =300, m =600, t =504	0	0.03	1	20.76	1	20.932	6	778.254	500
n =300, m =600, t =510	0	0.03	0	20.748	0	20.772	1	1475.852	100

taken to find a solution at least as good as the target value $\hat{z}$. As in [1], to plot the empirical distribution we associate with the i^{th} sorted running time (t_i) a probability $p_i = \frac{i-1/2}{200}$, and plot the points $z_i = (t_i, p_i)$, for $i = 1, \ldots, 200$. About these further experiments we observe that the relative position of the curves implies that, given a fixed amount of computing time greater than or equal to 2 seconds in the first two cases (Figures 10 and 11) and greater than 10 seconds otherwise (Figure 12), Genetic Algorithm has a higher probability than all competitors of finding a solution whose objective function value is at least as good as the target objective function value.

4 Conclusions and future directions

A large number of problems in molecular biology can be formulated as combinatorial optimization problems. In fact, the applicability of operations research models and methods in molecular biology is a rapidly growing and fruitful area in the cross section of several different scientific disciplines.

This paper describes some interesting biological system problems that can be formulated as combinatorial optimization problems, known as string selection and comparison problems. Special emphasis has been given to their recently proposed mathematical formulation and to efficient mathematical programming methods that can be applied to solve them.

We have proposed a GRASP and a Genetic Algorithm since these problems are computationally intractable and it is unlikely that a practical and efficient algorithm that guarantees an optimal solution within a realistic and acceptable amount of time will be ever proposed. The algorithms were tested on several random instances and compared with results produced by the best state-of-the-art heuristic approach proposed by Pardalos et al. in 2005 [28]. The results show that the algorithms produces good-quality solutions for all instances. In particular, the Genetic Algorithm always finds much better quality solutions compared with the other three competitor algorithms, but with much higher running times. In the following, we summarize our observations about our computational experience:

- The processing time for Pardalos et al.'s algorithm was the smallest but the objective function values found by the algorithm were the worst.
- For the three metaheuristics GRASP First Improvement, GRASP Best Improvement, and Genetic Algorithm, objective function values increased with processing times.
- Overall, Genetic Algorithm found the best solutions, followed by GRASP Best Improvement, and GRASP First Improvement.
- On the smaller set of problems (Table 1), runs for 2000 iterations of GRASP Best Improvement were about 3 to 21 times faster than 2000 iterations of the Genetic Algorithm. On the larger set of problems (Table 2 and Table 3), runs for 2000 iterations of GRASP Best Improvement were about 6.5 to 71 times faster than 2000 iterations of the Genetic Algorithm.
- During the experiments we noticed that often Genetic Algorithm finds good solutions after fewer generations than 2000. To show this characteristic of GA, we plot in Figures 10–12 the empirical distributions of the random variable *time-to-target-solution-value* considering three different random instances. Our conclusion after this further investigation is that, given a fixed amount of computing time - greater than or equal to 2 seconds in the first two cases (Figures 10 and 11) and greater than 10 seconds otherwise (Figure 12) - Genetic Algorithm has a higher probability than

all competitors of finding a solution whose objective function value is at least as good as the target objective function value.

The results show that the Genetic Algorithm produces better quality solutions. Each iteration of the algorithm requires more running time, but it has been empirically shown that those better solutions can be produced with no need of longer runs of the algorithm. Moreover, since this drawback of Genetic Algorithm related to its running time is due to the high value for the parameter $PopSize = n$ for each problem size, it could be worth to better investigate its behavior for example running it with moderate population sizes.

Furthermore, it would be also interesting to design some variants of the approaches proposed in this paper. A natural extension of proposed meta-heuristic algorithms would be their hybridization with some intensification and post-optimization procedures, such as path-relinking [16, 32]. Such a procedure could be applied at each GRASP and Genetic Algorithm iteration as intensification and/or at the end of the computation as post-optimization.

5 Acknowledgements

The authors gratefully acknowledge Daniele Ferone for his help in the implementation and experimentation phase and the anonymous referees for their comments and suggestions which have been revealed useful to improve both quality and readability of the manuscript.

References

[1] R.M. Aiex, M.G.C. Resende, and C.C. Ribeiro. Probability distribution of solution time in grasp: an experimental investigation. *Journal of Heuristics*, 8:343–373, 2002.

[2] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation.* Springer, 1999.

[3] J.C. Bean. Genetic algorithms and random keys for sequencing and optimization. *ORSA J. on Computing*, 6:154–160, 1994.

[4] P. Domínguez-Marín, S. Nickel, P. Hansen, and N. Mladenović. Heuristic procedures for solving the discrete ordered median problem. *Annals of Operations Research*, 136(1):145-173, 2005.

[5] T. Easton and A. Singireddy. A specialized branching and fathoming technique for the longest common subsequence problem. *International Journal of Operations Research*, 4(2):98–104, 2007.

[6] A.E. Eiben, E.H.L. Aarts, and K.M. Van Hee. Global convergence of genetic algorithms: A Markov chain analysis. In *Proceedings of 1st Workshop*

on Parallel Problem Solving from Nature, volume 496 of Lecture Notes in Computer Science, pages 3–12, 1991.

[7] R. Fagin. Generalized first-order spectra and polynomial time recognizable sets. In R. Karp, editor, *Complexity of Computation*, pages 43–73. AMS, Providence, 1974.

[8] C.N. Meneses F.C. Gomes, P.M. Pardalos, and G.V.R. Viana. Parallel algorithm for the closest string problem. In R. Mondaini, editor, *Proceedings of the Fourth Brazilian Symposium on Mathematical and Computational Biology / First International Symposium on Mathematical and Computational Biology*, volume 2, pages 326–332, 2005.

[9] T.A. Feo and M.G.C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8:67–71, 1989.

[10] T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[11] P. Festa. On some optimization problems in mulecolar biology. *Mathematical Bioscience*, 207(2):219–234, 2007.

[12] P. Festa and M.G.C. Resende. GRASP: An annotated bibliography. In C.C. Ribeiro and P. Hansen, editors, *Essays and Surveys on Metaheuristics*, pages 325–367. Kluwer Academic Publishers, 2002.

[13] C. Fleurent and J.A. Ferland. Genetic and hybrid algorithms for graph coloring. *Annals of Operations Research*, 63:437-461, 1996.

[14] M. Frances and A. Litman. On covering problems of codes. *Theory of Computing Systems*, 30(2):113–119, 1997.

[15] M. Garey and D. Johnson. *Computers and Intractability: a guide to the theory of NP-completeness.* W.H. Freeman, San Francisco, 1979.

[16] F. Glover. Tabu search and adaptive memory programming: Advances, applications and challenges. In R. Barr, R. Helgason, and J. Kennington, editors, *Interfaces in Computer Science and Operations Research*, pages 1–75. Kluwer, 1996.

[17] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[18] D.E. Goldberg and P. Segrest. Finite markov chain analysis of genetic algorithms. In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms (Lawrence Erlbaum Associates)*, pages 1–8, 1987.

[19] F.C. Gomes, C.N. Meneses, P.M. Pardalos, and G.V.R. Viana. A parallel multistart algorithm for the closest string problem. *Computers & Operations Research*, 35(11):3636–3643, 2008.

[20] H.A. Guvenir and E. Erel. Multicriteria inventory classification using a genetic algorithm. *European Journal of Operational Research*, 105(1):29-37, 1998.

[21] J.H. Holland. *Adaptation in Natural and Artificial Systems.* Univ. of Michigan Press, Ann Arbor, Mich., USA, 1975.

[22] R.B. Hollstein. *Artificial genetic adaptatiion in computer control systems.* PhD thesis, 1971.

[23] J.R. Koza, F.H. Bennett III, D. Andre, and M.A. Keane. *Genetic Programming III, Darwinian Invention and Problem Solving.* Morgan Kaufmann Publishers, 1999.

[24] J. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang. Distinguishing string selection problems. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete ALgorithms (SODA)*, pages 633–642, 1999.

[25] J. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang. Distinguishing string selection problems. *Information and Computation*, 185(1):41–55, 2003.

[26] M. Li, B. Ma, and L. Wang. Finding similar regions in many strings. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 473–482, 1999.

[27] G.E. Liepins and M.R. Hilliard. Genetic algorithms: Foundations and applications. *Annals of Operations Research*, 21(1–4):31-58, 1989.

[28] C.N. Meneses, C.A.S. Oliveira, and P.M. Pardalos. Optimization techniques for string selection and comparison problems in genomics. *IEEE Engineering in Medicine and Biology Magazine*, 24(3):81–87, 2005.

[29] C.A.S. Oliveira and P.M. Pardalos. Network flow algorithm for the longest common subsequence problem. In R. Mondaini, editor, *Proceedings of the Fourth Brazilian Symposium on Mathematical and Computational Biology / First International Symposium on Mathematical and Computational Biology*, volume 2, pages 300–313, 2005.

[30] P.M. Pardalos, C.A.S. Oliveira, Z. Lu, and C.N. Meneses. Optimal solutions for the closest string problem via integer programming. *INFORMS Journal on Computing*, 16:419–429, 2004.

[31] M.G.C. Resende and C.C. Ribeiro. Greedy randomized adaptive search procedures. In F. Glover and G. Kochenberger, editors, *State-of-the-Art Handbook of Metaheuristics.* Kluwer, 2002.

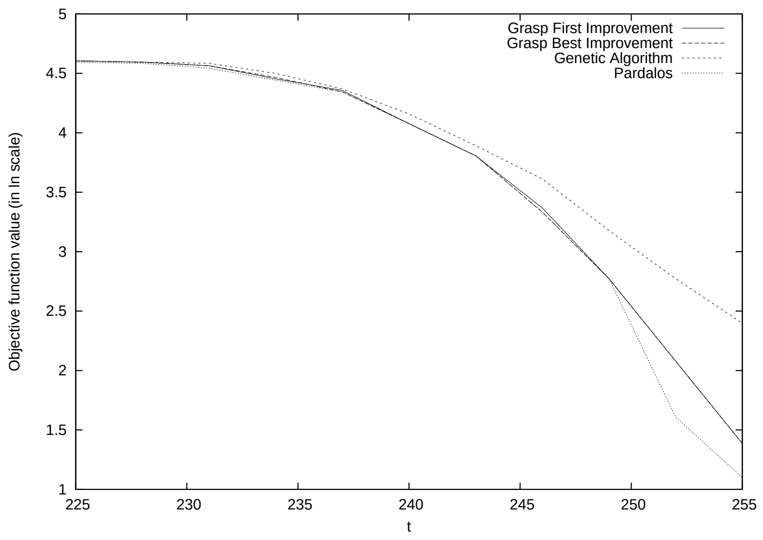
[32] M.G.C. Resende and C.C. Ribeiro. GRASP and path-relinking: Recent advances and applications. In T. Ibaraki, K. Nonobe, and M. Yagiura, editors, *Metaheuristics: Progress as Real Problem Solvers*, pages 29–63. Springer, 2005.

[33] S. Roman. *Coding and Information Theory.* volume 134 of Graduate Texts in Mathematics, Springer-Verlag, 1992.

[34] J.S. Sim and K. Park. The consensus string problem for a metric is NP-complete. In *Proceedings of the Annual Australiasian Workshop on Combinatorial Algorithms (AWOCA)*, pages 107–113, 1999.

[35] D. Wang and S.-C. Fang. A semi-infinite programming model for earliness/tardiness production planning with a genetic algorithm. *Computers & Mathematics with Applications*, 31(8):95106, 1996.

[36] A.R.P. White, J.W. Mann, and G.D. Smith. Genetic algorithms and network ring design. *Annals of Operations Research*, 86:347-374, 1996.



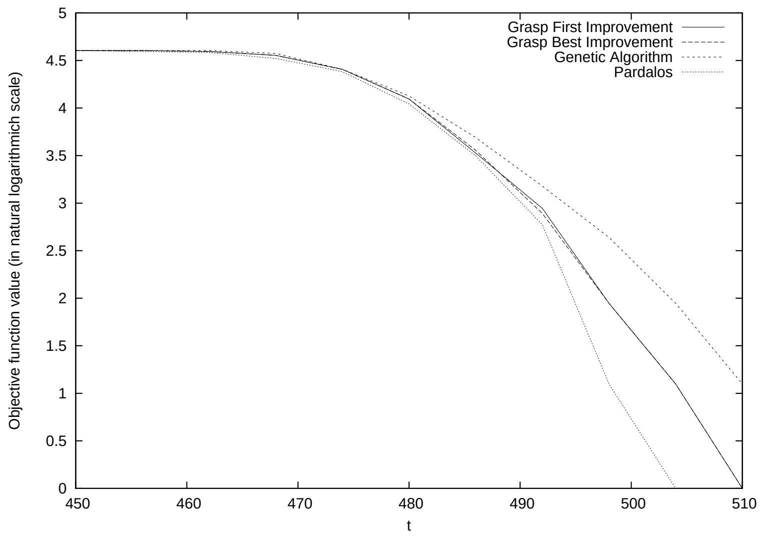
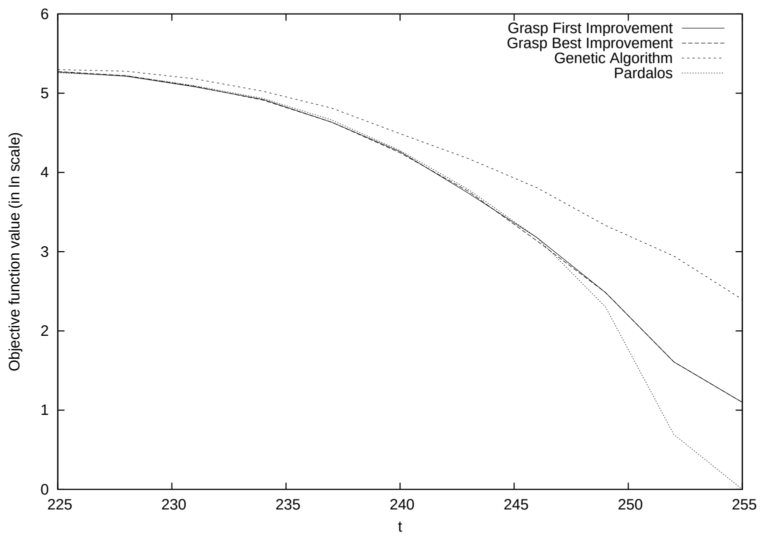


Figure 7: Mean objective function values comparing the four algorithms on random instances with $n = 100$ while $m = 300$ and $m = 600$, respectively.



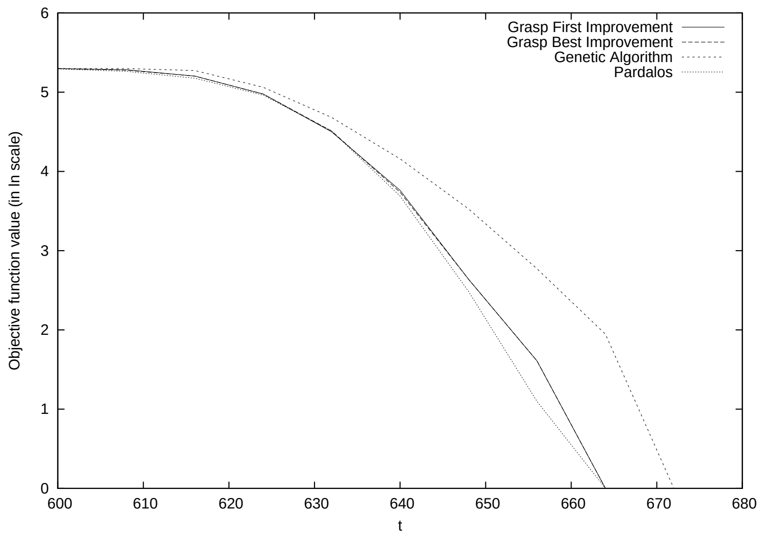
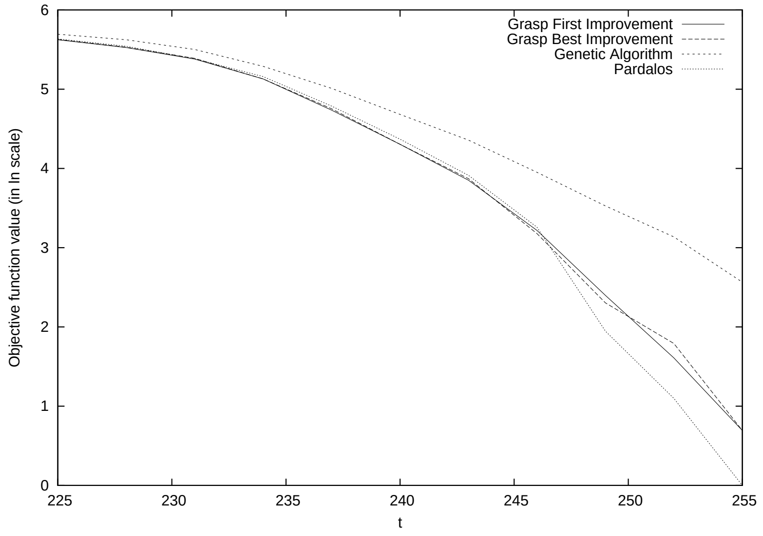


Figure 8: Mean objective function values comparing the four algorithms on random instances while $n = 200$ and $m = 300$ and $m = 800$, respectively.



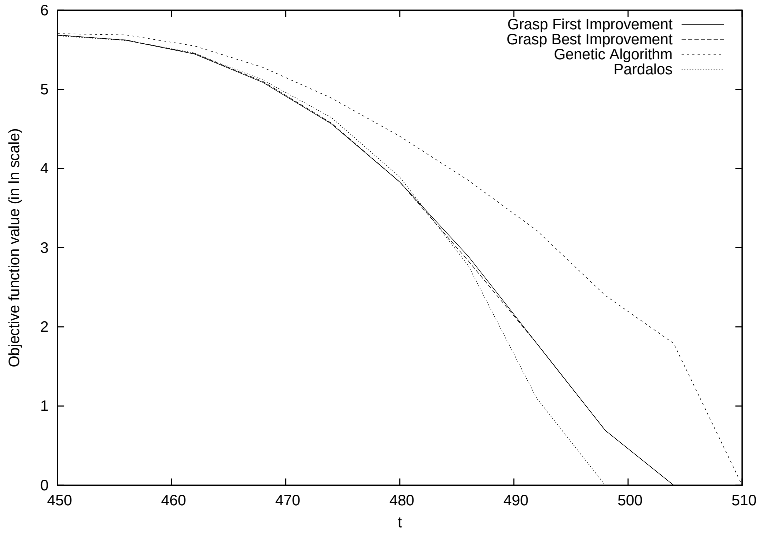


Figure 9: Mean objective function values comparing the four algorithms on random instances while $n = 300$ and $m = 300$ and $m = 600$, respectively.

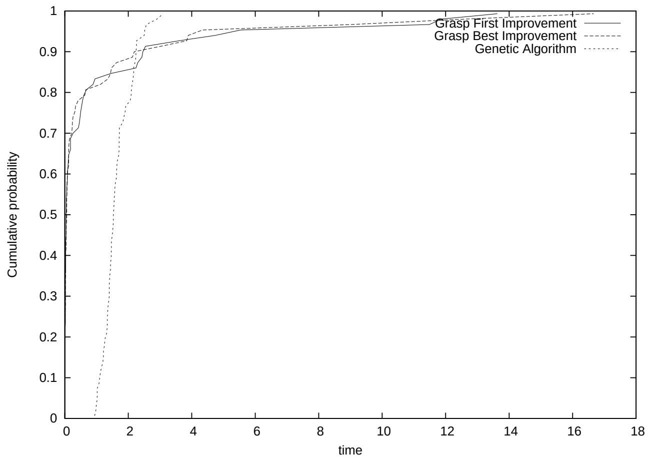


Figure 10: Time to target distributions comparing `Grasp First Improvement`, `Grasp Best Improvement`, and `Genetic Algorithm` on random instance with $n = 100$, $m = 300$, $t = 240$, and target value $\hat{z} = 0.57 \times n$.

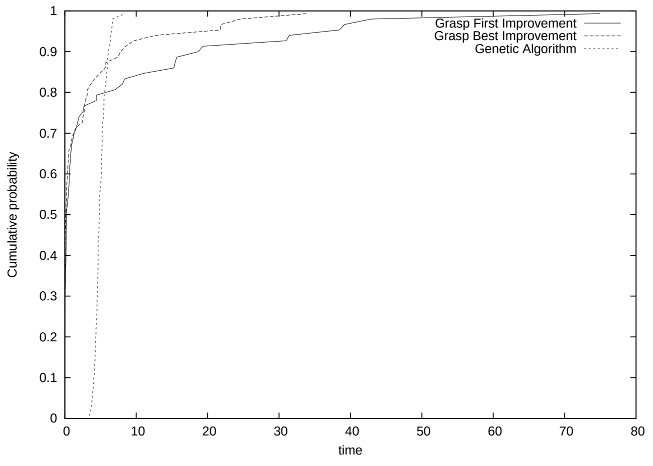


Figure 11: Time to target distributions comparing `Grasp First Improvement`, `Grasp Best Improvement`, and `Genetic Algorithm` on random instance with $n = 100$, $m = 300$, $t = 240$, and target value $\hat{z} = 0.33 \times n$.

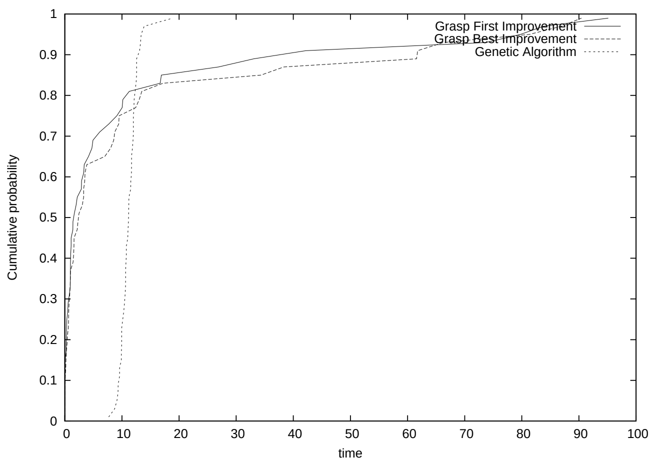


Figure 12: Time to target distributions comparing `Grasp First Improvement`, `Grasp Best Improvement`, and `Genetic Algorithm` on random instance with $n = 100$, $m = 300$, $t = 240$, and target value $\hat{z} = 0.25 \times n$.