Ant-CSP: an Ant Colony Optimization Algorithm for the Closest String Problem

Simone Faro and Elisa Pappalardo

Università di Catania, Dipartimento di Matematica e Informatica
Viale Andrea Doria 6, I-95125 Catania, Italy
{faro | epappalardo} @dmi.unict.it

Abstract. Algorithms for sequence analysis are of central importance in computational molecular biology and coding theory. A very interesting problem in this field is the Closest String Problem (CSP) which consists in finding a string t with minimum Hamming distance from all strings in a given finite set. To overcome the NP-hardness of the CSP problem, we propose a new algorithm, called Ant-CSP, based on the Ant Colony Optimization metaheuristic. To assess its effectiveness and robustness, we compared it with two state-of-the-art algorithms for the CSP problem, respectively based on the simulated annealing and the genetic algorithm approaches. Experimental results show that Ant-CSP outperforms both of them in terms of quality of solutions and convergence speed.

Keywords: closest string problem, string comparison problems, metaheuristic algorithms, ant colony optimization, NP-hard problems.

1 Introduction

The task of finding a string that is close to each of the strings in a given finite set is a combinatorial optimization problem particularly important in computational biology and coding theory. In molecular biology, one of the main issues related to DNA or protein sequences comparison is to find similar regions. Such problem finds applications, for instance, in genetic drug target and genetic probes design [16], in locating binding sites [25, 11], and in coding theory [6, 5], where one is interested in determining the best way to encode a set of messages [24].

A precise definition of the Closest String Problem (CSP, for short) is given next. To begin with, for a string s over a finite alphabet Σ, we denote by $|s|$ and $s[i]$ the length of s and the i-th character of s, respectively. The *Hamming distance* $H(s,t)$ between two strings s and t, of equal length, is the number of positions in which s and t differ.

Definition 1 (CSP problem). *Let $S = \{s_1, s_2, ..., s_n\}$ be a finite set of n strings, over an alphabet Σ, each of length m. The Closest String Problem for S is to find a string t over Σ, of length m, that minimizes the Hamming distance $H(t,S) =_{Def} max_{s \in S} H(t,s)$.*

Recently, the CSP problem has received much attention. Frances and Litman [5] have proved the NP-hardness of the problem for binary codes. Gramm *et al.* [9, 10] provided a fixed-parameter algorithm for the CSP problem with running time $O(nm + nd^{d+1})$, where d is the parameter. Plainly, for large values of d, such approach becomes prohibitive.

Several approximation algorithms have been proposed for the CSP problem: Gasieniec *et al.* [6] and Lanctot *et al.* [16] developed two $(4/3 + \epsilon)$-polynomial time approximation algorithms. Then, based on such results, Li *et al.* [17] and Ma and Sun [20] presented two new polynomial-time approximation algorithms. However, the running time of these algorithms make them of theoretical importance only. We mention also an approach based on the integer-programming formulation proposed by Meneses, Pardalos *et al.* in [21]: the CSP is reduced to an integer-programming problem, and then using a branch-and-bound algorithm to solve it. Despite the high quality of solutions, such algorithm is not always efficient and has an exponential time complexity. Moreover, the branch-and-bound tecnique leads easily to memory explosion.

Another approach to NP-hard problems consists in using heuristics. In 2005, Liu *et al.* [18] proposed to apply two heuristic algorithms to solve the CSP problem, based on the simulated annealing and the genetic algorithm approaches; then in [19], Liu and Holger presented a hybrid algorithm which combines both the genetic and the simulated annealing approaches, though limited only to binary alphabets. We mention also the approach in [8], based on a combination of a 2-approximation algorithm with local search strategies, consisting in taking a string s and modifying it until a local optimal solution is found. However, approximation algorithms sacrifice solution quality for speed [12].

In this paper we propose a new heuristic solution, Ant-CSP, based on the Ant Colony Optimization (ACO) metaheuristic, and we compare it to those proposed by Liu *et al.* [18], which are to date the fastest algorithms for the CSP problem. The ACO metaheuristic is inspired by the foraging behaviour of ant colonies [3]. Artificial ants implemented in ACO are stochastic procedures that, taking into account heuristic information and artificial pheromone trails, probabilistically construct solutions by iteratively adding new components to partial solutions.

The paper is organized as follows. In Sections 2 and 3, we present in some detail the two heuristic algorithms by Liu *et al.* [18] for the CSP problem, respectively based on the simulated annealing and the genetic algorithm approaches. Then, in Section 4, after describing the Ant Colony Optimization metaheuristic, we illustrate our proposed solution, Ant-CSP. In Section 5, we discuss the results of an extensive experimental comparison of our algorithm with the ones by Liu *et al.* [18]. Finally, we report our conclusions in Section 6.

2 A simulated annealing algorithm for the CSP problem

Simulated Annealing (SA) is a generalization of Monte Carlo methods, originally proposed by Metropolis and Ulam [23, 22] as a means of finding the equilibrium configuration of a collection of atoms at a given temperature. The basic idea of

SA was taken from an analogy with the annealing process used in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. In the original Metropolis scheme, an initial state of a thermodynamic system is chosen, having energy E and temperature T. Keeping T constant, the initial configuration is perturbed, and the energy change ΔE is computed. If ΔE is negative, the new configuration is always accepted, otherwise it is accepted with a probability given by the Boltzmann factor $e^{-(\Delta E/T)}$. This process is repeated L times for the current temperature, then the temperature is decremented and the entire process is repeated until a frozen state is reached at $T = 0$. Due to such characteristics, methods based on the SA may accept not only transitions that lead to better solutions, but also transitions that lead to worse ones, though with probabilities which tend to 0: at the beginning of the search, when temperatures are high, the algorithm behaves as a random search and therefore bad solutions can be accepted; whereas for lower values of T, solutions are located in promising regions of the search space.

Kirkpatrick [15] first proposed to apply SA to solve combinatorial optimization problems. The SA algorithm for the CSP problem by Liu *et al.* [18] works much along the same lines as Kirkpatrick's algorithm. Initially, the temperature T is set to $m/2$, where m is the common string length. For each temperature value, a block of L iterations is performed. At each iteration, a new string u' of length m, over Σ, is constructed in the following way: the current string u is split around a random point and the two substrings are interchanged. Then the energy change $\Delta E = H(u', S) - H(u, S)$ is evaluated, where S is the input set of strings. If $\Delta E \leq 0$, u' becomes the new current solution, otherwise u' is choosen as current solution with probability $e^{-(\Delta E/T)}$ only. At the end of each block of iterations, the temperature value is multiplied by a reduction factor γ. Liu *et al.* [18] report experiments with the parameters $L = 100$ and $\gamma = 0.9$. The algorithm stops when a suitable termination criterion is met.

The pseudo-code of the algorithm SA for the CSP problem (SA-CSP) is shown below.

3 A genetic algorithm for the CSP problem

Genetic algorithms were first proposed by John Holland [13, 7, 14, 1] as an abstraction of the biological evolution of living organisms. Genetic algorithms are based on natural selection and sexual reproduction processes. The first mechanism determines which members of a population survive and are able to reproduce, the second one assures genic recombination among individuals of a same population. The principle of selection is based on a function, called *fitness*, that measures "how good is an individual": individuals with better fitness have higher probability to reproduce and survive.

In the genetic algorithm for the CSP problem (GA-CSP, for short) proposed by Liu *et al.* [18], an initial population P of random candidate solutions $ind_0, ..., ind_{popsize-1}$ is generated. An individual/solution is a string of length m

Algorithm 1 SA-CSP(S)

```
1: randomly generate an initial string u
2: set an initial T = T_max
3: set the number of iterations L
4: set γ
5: while not (TERMINATION_CRITERION) do
6:     for 0 ≤ I < L do
7:         u' ← mutate(u);
8:         Δ = evaluate_energy(u', S) − evaluate_energy(u, S);
9:         if ((Δ ≤ 0) or ((Δ > 0) and (e^{−Δ/T} > random(0, 1)))) then
10:            u ← u';
11:        end if
12:    end for
13:    T ← γ · T;
14: end while
```

over the alphabet Σ of the input string set S. The fitness function f is evaluated for each string in the current population, where $f = m - H_{max}$ and H_{max} is the maximum Hamming distance of s from all strings in S: therefore the larger is f, the better will be the solution represented by the string. A crossover step allows to generate new individuals from members of the current population. More specifically, firstly two "parental" individual ind_x and ind_y, are randomly selected according to their crossover probablity p_c, which is proportional to the fitness value of each individual. Then the crossover operator simply exchanges a randomly selected segment in the pair of "parents" so that two new strings are generated, each inheriting a part from each of the parents. At this intermediate stage, there are two populations, namely, parents and offsprings. To create the next generation, an elitist strategy is applied, i.e., the best individuals from both populations are selected, based on their fitness. Finally, a mutation operator is applied to each individual, depending on a probability p_m: this consists in exchanging two random positions in the string. Reproduction and mutation steps are repeated until a termination criterion is met. We report the pseudo-code of GA-CSP as Algorithm 2 below.

4 Ant Colony Optimization

We present now a new algorithm for the CSP problem based on the Ant Colony Optimization (ACO) metaheuristic.

ACO is a multi-agent metaheuristic approach particularly suited for hard combinatorial optimization problems. ACO was firstly proposed by Dorigo [3] as an innovative approach to the Traveling Salesman problem. It has been inspired by the real behaviour of ant colonies, where the behaviour of single ants is directed to the survival of the whole colony. In particular, in his analysis Dorigo observed the foraging behaviour of colonies: when a new food source is found, ants search the shortest and easiest way to return to nest. While walking from

Algorithm 2 GA-CSP(S)

1: $t \leftarrow 0$
2: initialize $P(t) = \{ind_i \in P(t), i = 0, 1, ...popSize - 1\}$
3: evaluate $P(t)$ to get the fitness of each individual in S
4: calculate the probability of each individual, $p_i \propto ind_i.fitness$
5: $currentBest$ = best_ind($P(t)$);
6: $bestInd = ind_{currBest}$;
7: **while** $t < TERMINATION\_CRITERION$ **do**
8: $i = 0$;
9: **while** $i < popSize/2$ **do**
10: select ($ind_x ind_y$) from $P(t)$ according to their p_{ind}
11: $\{chd_{(2i)}, chd_{(2i+1)}\}$ = crossover(ind_x, ind_y);
12: **end while**
13: $i = 0$;
14: **while** $i < popSize$ **do**
15: $r \leftarrow random(0, 1)$;
16: **if** $(r < p_m)$ **then**
17: mutate(chd_i);
18: **end if**
19: $P(t + 1) \leftarrow P(t + 1) \bigcup chd_i$
20: **end while**
21: evaluate $P(t + 1)$ to get the fitness of each individual in S
22: calculate the probability of each individual, $p_i \propto ind_i.fitness$
23: $worst = worst\_ind(P(t + 1))$;
24: $ind_{worst} \leftarrow bestInd$;
25: $currBest = best\_ind(P(t + 1))$;
26: **if** $(ind_{currBest}.fitness > bestInd.fitness)$ **then**
27: $bestInd = ind_{currBest}$;
28: **end if**
29: $t \leftarrow t + 1$;
30: **end while**

food sources to the nest, and vice versa, ants deposit on the ground a substance called pheromone [4]. Ants can smell pheromones and, when choosing their way, they select, with higher probability, paths marked by strong pheromone concentrations. It has been proved that pheromone trails make shortest paths to emerge over other paths [2], due to the fact that pheromone density tends to be higher on shortest paths.

In analogy with the real behaviour of ant colonies, ACO applies pheromone trails and social behaviour concepts to solve hard optimization problems. In short, ACO algorithms work as follows: a set of asynchronous and concurrent agents, a colony of ants, moves through the states of the problem. To determine the next state, ants apply stochastic local decisions based on pheromone trails. Ants can release (additional) pheromone into a state, while building a solution, and/or after a solution has been built, by moving back to all visited states.

In an elitist strategy, only the ant that has produced the best solution is allowed to update pheromone trails. In general, the amount of pheromone deposited is proportional to the quality of the solution built.

To avoid a too rapid convergence towards suboptimal regions, ACO algorithms include another mechanism for the updating of pheromone trails, namely, pheromone evaporation. This consists in decreasing over time the intensity of the components of pheromone trails, as pheromone density tends to increase on shortest paths. Thus, the evaporation mechanism limits premature stagnation, namely situations in which all ants repeatedly construct the same solutions, which would prevent further explorations of the search space.

The ACO metaheuristic has two main application fields: NP-hard problems, whose best known solutions have exponential time worst-case complexity, and shortest path problems, in which the properties of the problem's graph representation can change over time, concurrently with the optimization process. As the CSP problem is NP-hard, and searching a closest string can be viewed as finding a minimum path into the graph of all feasible solutions, it is natural to apply the ACO heuristic to the CSP problem. This is what we do next.

4.1 The Ant-CSP algorithm

We describe now our proposed ACO algorithm for the CSP problem, called Ant-CSP. Given an input set S of n strings of length m, over an alphabet Σ, at each iteration of the Ant-CSP algorithm, a $COLONY$ consisting of u ants is generated. Each of the artificial ant, say $COLONY_i$, searches for a solution by means of the $find\_solution$ procedure, by building a string while it moves character by character on a table T, represented as a $|\Sigma| \times m$ matrix. The location $T[i, j]$, with $1 \leq i \leq |\Sigma|$ and $0 \leq j \leq m - 1$, mantains the pheromone trail for the i-th character at the j-th position of the string.

Ants choose "their way" probabilistically, using a probability depending on the value $T[i, j]$ of the local pheromone trail: the normalized probability for each character is computed, depending on the pheromone value deposited on it. So, the algorithm probabilistically chooses a character. Initially, $T[i, j] = 1/|\Sigma|$; when each ant has built and evaluated its own solution, respectively by means of the $find\_solution()$ and $evaluate\_solution()$ procedures, pheromone trails are updated. We adopted an elitist strategy, so that only the ant that has found the best solution, say $COLONY_{best}$, updates the pheromone trails, by depositing on the characters that appear in the best solution an amount of pheromone proportional to the quality of the solution itself. In particular:

$$T^{(t+1)}[i, j] = T^{(t)}[i, j] + \left(1 - \frac{HD}{m}\right),$$

where HD is the maximum Hamming distance of the current string from all strings in S. Thus, the larger is the pheromone trail for the i-th character, the higher will be the probability that this character will be chosen in the next iteration.

Pheromone values are normalized and are used as probabilities. After additional pheromone trail on the best string has been released, the evaporation procedure is applied: this consists in decrementing each value $T[i, j]$ by a constant factor ρ; in our experiments, we put $\rho = 0.03$.

The pseudo-code of the Ant-CSP algorithm is shown as Algorithm 3 below.

Algorithm 3 Ant-CSP(S)

```
 1: initialize table T
 2: for i ← 1 to m do
 3:     for j ← 1 to |Σ| do
 4:         T_ij ← 1/|Σ|
 5:     end for
 6: end for
 7: initialize COLONY
 8: while not (TERMINATION_CRITERION) do
 9:     for i ← 1 to u do
10:         COLONY_i ← new_ant()
11:         COLONY_i.find_solution()
12:         COLONY_i.evaluate_solution()
13:     end for
14:     for i ← 1 to m do                          ▷ start pheromone evaporation
15:         for j ← 1 to |Σ| do
16:             T_ij ← (1 − ρ) · T_ij;
17:         end for
18:     end for                                    ▷ end pheromone evaporation
19:     COLONY_best.update_trails()
20: end while
```

5 Experimental results

We have tested the SA-CSP, the GA-CSP, and the ACO-CSP algorithms using the azotated compounds alphabet $\Sigma = \{A, C, G, T\}$ of the fundamental components of nucleic acids.
In our test platform, we considered a number of input strings $n \in \{10, 20, 30, 40, 50\}$, and string length $m \in \{10, 20, ..., 50\} \cup \{100, 200, ..., 1000\}$. For each of a randomly generated problem instances, all algorithms were run 20 times.

The total colony size for the Ant-CSP algorithm as well as the population size for the GA-CSP algorithm have been set to 10, whereas the number of generations has been set to 1500. In the case of the SA-CSP algorithm, we fixed the number of function evaluations in 15,000, making the number of function evaluations comparable to the computational work performed by the former two algorithms.

Our tests have been performed on an Intel Pentium M 750, 1.86 GHz, 1 GB RAM, running Ubuntu Linux.

We report the results of our tests in the five tables below: *HD* indicates the Hamming distance value, that we want to minimize, *Time* is the running time in milliseconds. For each length, we computed the average (AVG) of the closest string scores found in the 20 runs and the standard deviation σ. Also, we computed the average of the running time over the 20 runs (AVG). Best results are reported in bold.

Experimental results show that almost always the Ant-CSP outperforms both the GA-CSP and the SA-CSP algorithms both in terms of solution quality and efficiency. As a matter of fact, the tables below show that our algorithm is much faster than both the GA-CSP and SA-CSP algorithms. In particular, in the case of short instances, i.e. for $10 \leq m \leq 50$, the Ant-CSP algorithm is from 5 to 36 times faster than GA-CSP.
Furthermore, it turns out that as n increases, the gap between the running time of the Ant-CSP and the SA-CSP algorithms becomes considerable. In Figure 1 we report the running times of the algorithms for some sets of strings.

We also remark that the Ant-CSP provides results of a better quality than the other two algorithms in terms of Hamming distance. In fact, the cooperation among the colony ants and the pheromone trails tend to orient the search towards optimal solutions, allowing to explore and modify local optima. On the other hand, the SA-CSP algorithm behaves as a random search, as it simply modifies a string without considering local promising solutions. Likewise, the GA-CSP algorithm performs random mutations and crossovers, whereas the Ant-CSP probabilistically selects each character for the solution.

We note also that the Ant-CSP algorithm is quite robust, as its standard deviation σ remains low.

The above considerations show that our algorithm represents a valid and innovative alternative to the SA-CSP and GA-CSP algorithms.

Size (m)	SA-CSP			GA-CSP			Ant-CSP		
	HD		Time	HD		Time	HD		Time
	AVG	σ	AVG	AVG	σ	AVG	AVG	σ	AVG
10	8.45	0.497	67.5	**6.9**	0.3	1840	7.05	0.218	50.5
20	15.9	0.384	112	13.3	0.714	1860	**13.1**	0.589	97
30	23.6	0.663	216	19.6	0.583	2700	**19.3**	0.557	200
40	31.4	0.589	313	25.3	0.714	3040	**25.1**	0.654	281
50	38.8	0.678	428	31.8	0.994	3220	**31.6**	0.805	386
100	75.9	0.943	465	63.4	1.31	2060	**62.2**	0.766	433
200	151	1.04	901	129	1.43	2290	**124**	1.58	855
300	226	1.18	1350	195	2.19	2540	**188**	1.57	1290
400	301	2.01	1780	262	2.52	2720	**252**	1.68	1700
500	375	2.05	2190	330	2.52	2940	**317**	2.15	2110
600	450	1.87	2740	400	3.71	3800	**385**	2.5	2920
700	525	1.68	3980	470	3.43	4860	**451**	2.95	4270
800	600	1.51	3720	540	4.04	4370	**517**	2.11	3860
900	675	1.19	5670	610	4.01	6110	**585**	4.05	5690
1000	750	1.53	7720	680	4.12	7850	**652**	3.72	7850

Table 1. Results for inputset of 10 strings of length m.

Size (m)	SA-CSP			GA-CSP			Ant-CSP		
	HD		Time	HD		Time	HD		Time
	AVG	σ	AVG	AVG	σ	AVG	AVG	σ	AVG
10	8.95	0.384	211	**7.95**	0.218	3560	**7.95**	0.218	132
20	17.1	0.589	342	**14.8**	0.4	3460	**14.8**	0.4	258
30	24.8	0.536	502	21.6	0.497	3300	**21.4**	0.49	370
40	32.5	0.497	602	28.1	0.477	3220	**28**	0.632	452
50	40.1	0.726	735	35	0.589	3300	**34.8**	0.536	546
100	78.4	0.663	874	69.5	0.921	2250	**67.7**	0.853	646
200	154	0.917	2070	140	1.74	3370	**135**	0.963	1460
300	229	1.16	2300	210	2.09	2970	**203**	1.95	1810
400	305	1.18	4460	281	1.95	4980	**272**	1.56	3090
500	380	1.25	5270	353	2.52	4930	**341**	1.65	3510
600	456	1.46	4610	426	1.89	4180	**411**	1.68	3660
700	531	1.16	6280	499	3.51	4770	**482**	1.95	4350
800	607	1.32	11300	572	1.88	9370	**553**	2.84	7780
900	682	1.49	13700	645	2.58	10800	**623**	2.51	10400
1000	757	1.69	15700	720	2.79	11800	**695**	2.49	11800

Table 2. Results for inputset of 20 strings of length m.

Size (m)	SA-CSP			GA-CSP			Ant-CSP		
	HD		Time	HD		Time	HD		Time
	AVG	σ	AVG	AVG	σ	AVG	AVG	σ	AVG
10	9	0	245	8.25	0.433	2830	**8.15**	0.357	148
20	17.3	0.458	518	15.3	0.458	3460	**15.2**	0.4	341
30	25.1	0.357	772	22.7	0.458	3520	**22.4**	0.477	508
40	33	0.316	985	29.5	0.5	3720	**29.1**	0.357	638
50	40.9	0.539	1230	36.9	0.357	4180	**36.1**	0.436	814
100	79.3	0.557	1280	72.2	0.726	2450	**70.8**	0.536	850
200	156	0.829	4760	144	1.08	5800	**140**	0.975	2750
300	232	0.831	6640	216	1.77	6610	**209**	1.27	4260
400	308	0.829	9160	290	2.93	8160	**280**	1.28	5550
500	383	0.963	11110	362	1.66	8830	**351**	1.79	6760
600	459	1.24	12500	436	2.14	9800	**423**	1.95	7610
700	534	1.03	14500	510	2.57	10900	**495**	2.01	9430
800	610	1.14	17700	583	2.57	12600	**568**	2.36	10300
900	686	1.69	19800	658	3.42	13200	**640**	2.09	11400
1000	760	2.24	19800	731	2.97	12400	**713**	2.29	10700

Table 3. Results for inputset of 30 strings of length m.

6 Conclusions

In this paper, we proposed a new promising method for the CSP problem and we presented and commented some experimental results.

We compared our approach, Ant-CSP, to two heuristic algorithms proposed by Liu *et al.* [18]: the SA-CSP algorithm, based on the simulated annealing approach, and the GA-CSP algorithm, based on the genetic algorithm approach. Experimental results show that our algorithm computes almost always better

Size (m)	SA-CSP			GA-CSP			Ant-CSP		
	HD		Time	HD		Time	HD		Time
	AVG	σ	AVG	AVG	σ	AVG	AVG	σ	AVG
10	9.4	0.49	428	8.9	0.3	4000	**8.55**	0.497	252
20	17.6	0.477	742	15.9	0.218	3990	**15.8**	0.433	471
30	25.6	0.49	1210	23.1	0.384	4690	**22.9**	0.384	754
40	33.3	0.458	1540	30.4	0.572	4640	**30.1**	0.218	962
50	41.2	0.433	1940	37.5	0.497	5070	**37**	0.589	1220
100	80	0.669	2080	73.6	0.663	3420	**71.7**	0.477	1260
200	157	0.889	5740	146	1.24	5570	**142**	0.669	3230
300	233	0.889	8760	219	0.954	8640	**214**	1.05	5550
400	309	0.831	10090	293	1.87	9510	**285**	1.16	6560
500	385	0.748	14800	368	2.07	11000	**358**	1.24	7330
600	461	1.01	17800	441	1.69	13100	**431**	1.91	7940
700	536	1.05	21700	515	2.1	14300	**503**	1.01	11700
800	612	1.1	23500	590	2.34	14300	**577**	1.93	11300
900	688	1.34	26700	664	2.52	17200	**649**	2.31	15600
1000	763	1.43	30900	738	2.62	15900	**722**	1.91	16000

Table 4. Results for inputset of 40 strings of length m.

Size (m)	SA-CSP			GA-CSP			Ant-CSP		
	HD		Time	HD		Time	HD		Time
	AVG	σ	AVG	AVG	σ	AVG	AVG	σ	AVG
10	9.45	0.497	574	9	0	4390	**8.85**	0.357	334
20	17.8	0.433	1030	16.2	0.4	4620	**16.1**	0.218	620
30	25.9	0.3	1490	23.5	0.5	4820	**23.2**	0.4	899
40	33.5	0.497	1960	30.9	0.357	5070	**30.6**	0.497	1180
50	41.7	0.458	2410	38.2	0.433	5270	**37.8**	0.433	1450
100	80.6	0.49	2970	74.7	0.64	3970	**73.3**	0.64	1750
200	158	0.671	9090	148	0.91	8530	**144**	0.698	5550
300	234	0.678	14000	222	0.91	10900	**216**	0.889	8320
400	310	0.792	18500	297	1.65	13100	**289**	1.41	11100
500	386	1.16	21900	369	1.69	14800	**362**	1.24	12900
600	462	1.13	21200	444	1.5	14500	**434**	1.74	12200
700	538	1.14	26800	519	1.9	17300	**508**	1.7	15500
800	614	1.43	28900	594	2.9	14000	**582**	2.29	13900
900	689	1.1	33500	667	1.64	19700	**656**	2.11	18800
1000	765	1.19	36600	742	3.09	21000	**729**	1.68	18300

Table 5. Results for inputset of 50 strings of length m.

solutions and is much faster than the GA-CSP and SA-CSP algorithms, regardless the number and the length of input strings.

Future works will be focused on two fronts: performance improvements and search for heuristic information to improve quality of solutions and convergence speeds. Additionally, we plan to extend our algorithm to the Closest Substring Problem.

Acknowledgments

The authors thank the referees for their helpful comments.

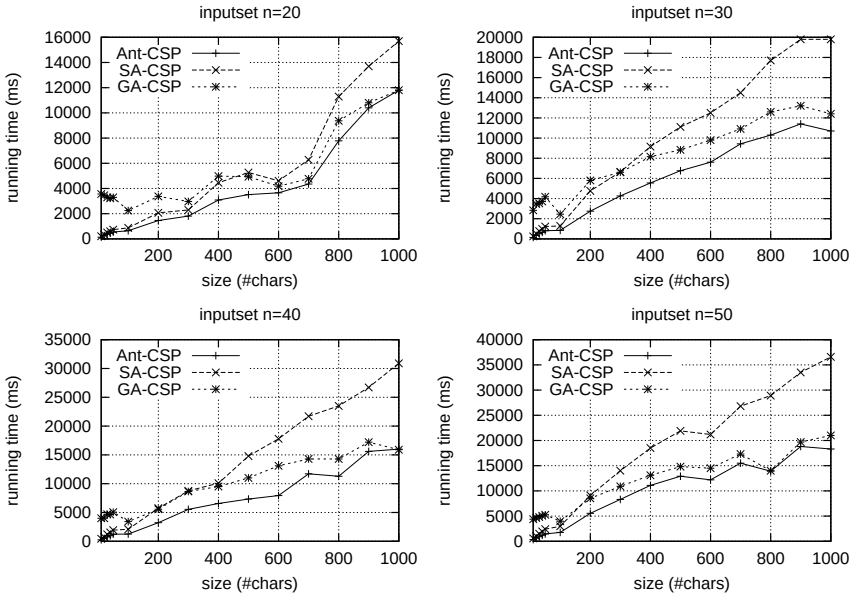

Fig. 1. Running times plot for $n = 20, 30, 40, 50$. Notice that, as n increases, the gap between Ant-CSP and the other two algorithms becomes more noticeable.

References

1. L.B. Booker, D.E. Goldberg, and J.H. Holland. Classifier systems and genetic algorithms. *Artif. Intell.*, 40(1-3):235–282, 1989.
2. J.L. Deneubourg, S. Aron, S. Goss, and J.M. Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behavior*, 3(2):159–168, 1990.
3. M. Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Dipartimento di Elettronica, Politecnico di Milano, Italy, 1992.
4. M. Dorigo, G.D. Caro, and L.M. Gambardella. Ant Algorithms for Discrete Optimization. *Artificial Life*, 5(2):137–172, 1999.
5. M. Frances and A. Litman. On covering problems of codes. *Theory of Computing Systems*, 30(2):113–119, 1997.
6. L. Gasieniec, J. Jansson, and A. Lingas. Efficient approximation algorithms for the Hamming center problem. In *Proceedings of the 10th annual ACM-SIAM Symposium on Discrete algorithms*, pages 905–906. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 1999.
7. D.E. Goldberg and J.H. Holland. Genetic Algorithms and Machine Learning. *Machine Learning*, 3(2):95–99, 1988.

8. F.C. Gomes, C.N. Meneses, P.M. Pardalos, and G.V.R. Viana. A parallel multistart algorithm for the closest string problem. *Computers and Operations Research*, 35(11):3636–3643, 2008.
9. J. Gramm, R. Niedermeier, and P. Rossmanith. Exact solutions for Closest String and related problems. In *ISAAC '01: Proceedings of the 12th International Symposium on Algorithms and Computation*, pages 441–453, London, UK, 2001. Springer-Verlag.
10. J. Gramm, R. Niedermeier, and P. Rossmanith. Fixed-parameter algorithms for closest string and related problems. *Algorithmica*, 37(1):25–42, 2003.
11. G.Z. Hertz, G.W. Hartzell, and G.D. Stormo. Identification of consensus patterns in unaligned DNA sequences known to be functionally related. *Bioinformatics*, 6(2):81–92, 1990.
12. D.S. Hochba. *Approximation algorithms for NP-hard problems*, volume 28. ACM New York, NY, USA, 1997.
13. J.H. Holland. *Adaptation in Natural and Artificial Systems*. The MIT Press, 1992.
14. J.H. Holland. Genetic algorithms computer programs that "evolve" in ways that resemble natural selection can solve complex problems even their creators do not fully understand. *Scientific American*, 267:62–72, 1992.
15. S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.
16. J.K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang. Distinguishing string selection problems. In *Proceedings of the 10th annual ACM-SIAM Symposium on Discrete algorithms*, pages 633–642. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 1999.
17. M. Li, B. Ma, and L. Wang. On the closest string and substring problems. *Journal of the ACM*, 49(2):157–171, 2002.
18. X. Liu, H. He, and O. Sykora. Parallel Genetic Algorithm and Parallel Simulated Annealing Algorithm for the Closest String Problem. In *Advanced Data Mining and Applications*, volume 3584 of *Lecture Notes in Computer Science*, pages 591–597. Springer Berlin/Heidelberg, 2005.
19. X. Liu, M. Holger, Z. Hao, and G. Wu. A Compounded Genetic and Simulated Annealing Algorithm for the Closest String Problem. In *Bioinformatics and Biomedical Engineering, 2008. ICBBE 2008. The 2nd International Conference on*, pages 702–705, 2008.
20. B. Ma and X. Sun. More Efficient Algorithms for Closest String and Substring Problems. *Lecture Notes in Computer Science*, 4955:396, 2008.
21. C.N. Meneses, Z. Lu, C.A.S. Oliveira, and P.M. Pardalos. Optimal Solutions for the Closest-String Problem via Integer Programming. *Informs Journal on Computing*, 16(4):419–429, 2004.
22. N. Metropolis, A.E. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Perspective on "Equation of state calculations by fast computing machines". *J. Chem. Phys*, 21:1087–1092, 1953.
23. N. Metropolis and S. Ulam. The Monte Carlo method. *Journal of the American Statistical Association*, 44(247):335–341, 1949.
24. S. Roman. *Coding and information theory*. Springer, 1992.
25. G.D. Stormo and G.W. Hartzell. Identifying protein-binding sites from unaligned DNA fragments. In *Proc. Natl. Acad. Sci. USA*, volume 86, pages 1183–1187, 1989.