SAX: a new and efficient assembler for solving DNA Fragment Assembly Problem

Gabriela Minetti[1], Guillermo Leguizamón[2], and Enrique Alba[3]

[1] National University of La Pampa, La Pampa, Argentina,
minettig@ing.unlpam.edu.ar,
[2] National University of San Luis, San Luis, Argentina
legui@unsl.edu.ar,
[3] University of Málaga, Málaga, Spain
eat@lcc.uma.es

Abstract. In the past, the Fragment Assembly Problem has been solved efficiently by many metaheuristics. In this work, we propose a new one, called SAX, which consists in combining two metaheuristics: a trajectory method as Simulated Annealing and a population-based method as Genetic Algorithm. We also analyze the relative advantages of this hybridization against other assemblers from literature. From this analysis, we conclude that SAX improves the quality results found by other metaheuristic and non-metaheuristic assemblers for solving 100% of the largest instances for this problem.

1 Introduction

The Fragment Assembly Problem (FAP) consists in building a DNA sequence from several hundreds (or even, thousands) of fragments, which are obtained by biologists in the laboratory. The DNA fragment assembly is needed because current technology, such as gel electrophoresis, cannot directly and accurately sequence DNA molecules longer than 1000 bases. However, most genomes are much longer. FAP is the primary goal in any genome project and the remaining phases depend on the accuracy of the results at this stage. Therefore, we need accurate and efficient methods for solving this NP-complete problem.

Ant Colony Optimization (ACO), Genetic Algorithms (GA), Variable Neighborhood Search (VNS), and Artificial Neural Networks (ANN) are some metaheuristics, which have been recently used for solving the DNA Fragment Assembly Problem (FAP). As we can see in [2, 6–9], these methods methods outperform several algorithms that have been specially developed for this problem, but more efficacy and efficiency are necessary since smaller DNA sequences are more than 3000 bases long. To achieve these properties, efficacy and efficiency, new metaheuristic assemblers were proposed like ISA in [10, 14], PALS in [1], and GAG_{50} in [11].

ISA [10, 14] is a SA-based metaheuristic that incorporates the inversion procedure as strategy to generate neighbor solutions. This assembler obtains optimal

layouts for all the instances of different size tested but, the overlapping scores are decreased for larger instances.

PALS [1] is a metaheuristic that performs a trajectory in the search space by improving a single-solution using a specific local search. The PALS main strength is the inexpensive calculation of the variation in the overlap and in the number of contigs between the current solution and the resulting solution after applying a movement. This calculation is not computationally expensive since neither the fitness function nor the number of contigs are calculated in each iteration. Instead PALS estimates the variation of these values. However, its most important weakness is the quick convergence to local optima caused by the local search when PALS solves large instances, as reported in [12, 13].

GAG_{50} [11] is a GA that initializes 50% of the first population using a greedy strategy (which is also proposed in [11]). This metaheuristic uses a swap mutation and the order crossover (OX) as genetic operators. Although, GAG_{50} achieves very high overlapping scores, it can not find the optimal layout for the larger instances of FAP.

Our proposal is to design a metaheuristic that finds optimal layouts with high overlapping scores for the most complex instances of FAP, the larger ones. In order to do that, we propose to hybridize the metaheuristics that reach these targets separately. More specifically, we combine ISA with the order crossover. The behavior of this new assembler, called SAX, is compared with the performance of ISA, PALS, GAG_{50}, and two non-metaheuristic assemblers.

The rest of this article is organized as follows. The next section introduces the DNA Fragment Assembly Problem. Section 3 explains how the ISA is hybridized with OX to solve this problem. In Section 4 the experimental design is described. In the fifth section, we analyze the SAX behavior and make a comparison against ISA, PALS, and GAG_{50}. Further, we compare the performance of these four metaheuristics against other assemblers proposed in literature. Finally, we discuss the conclusions and hints to further research.

2 The DNA Fragment Assembly Problem

The fragment assembly reconstructs an original DNA sequence from a set of separate fragments, which are obtained by a sequencing procedure. One of the most used sequencing procedure is the *shotgun sequencing*, which first cuts the DNA into small pieces, second individually sequences each of these fragments, and third puzzles together to create the original contiguous sequence, i.e., *the contig*. The main advantages of the shotgun sequencing method are its high level of automation and scalability. Specifically, the shotgun sequencing method consists in:

1. Several copies of the DNA are produced and each copy is broken into millions of random fragments.
2. Those fragments are read by a DNA sequencing machine.
3. An assembler pieces together the many overlapping fragments and reconstructs the original sequence.

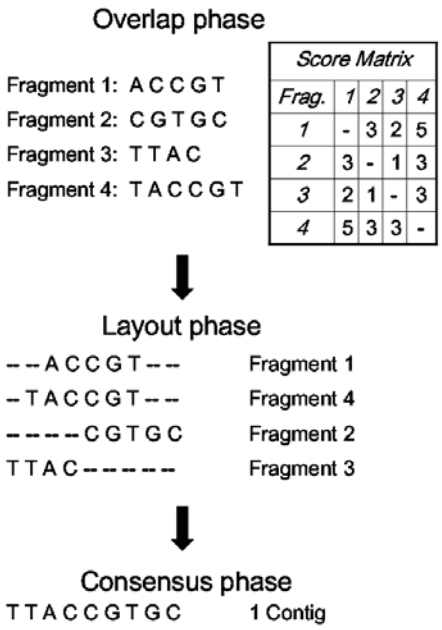

Fig. 1. DNA Fragment Assembly Process.

The assembling of DNA fragments is divided into three different phases (as show Figure 1):

1. *Overlap*: finding the overlapping among fragments -score-. This phase consists in finding the best or longest match between the suffix of every sequence and the prefix of another one. We compare all possible pairs of fragments to determine their similarity. Usually, the dynamic programming algorithm is used in this step to find semi-global alignments.
2. *Layout*: finding the order of fragments based on computed similarity scores. This is the most difficult step because true overlaps are hard to determine. After the order is determined, an alignment algorithm is applied to combine all the pairwise alignments obtained in the overlap phase.
3. *Consensus*: deriving the DNA sequence from the layout. The most common technique used in this phase is to apply the majority rule in building the consensus.

If no sequencing error was detected at the overlap phase, the process simply finds the longest suffix of one string that exactly matches the prefix of another one. However, when sequencing errors exist, the process finds the best matches but a small percentage of errors remains (1-3%). During the assembling process, the only information available is the sequence of bases, and thus the ordering

of the fragments must rely primarily on the similarity of fragments and on how much they overlap.

Once the fragments have been ordered (layout), the final consensus is generated. This means that a multiple alignment is computed to obtain a consensus sequence that will be used as the genomic sequence. To measure the quality of a consensus, we can look at the distribution of the coverage. The coverage at a base position is defined as the number of fragments at that position. It is a measure of the redundancy of the fragment data, and it denotes the number of fragments, on average, in which a given nucleotide in the target DNA is expected to appear. The coverage is computed as the number of bases read from fragments over the length of the target DNA:

$$Coverage = \frac{\sum_{i=1}^{n} length\ \ of\ \ the\ \ fragment\ \ i}{target\ \ sequence\ \ length} \tag{1}$$

where n is the number of fragments. The higher the coverage, the fewer the number of gaps, and the better the result. An incomplete coverage is provoked when the algorithm is not able to assemble a given set of fragments into a single *contig*. More specifically, a contig is defined as a layout consisting of contiguous overlapping fragments. The overlap between adjacent fragments must be greater or equal to a predefined threshold (cutoff parameter).

Particularly, the assembly of DNA fragments into a consensus sequence corresponding to the parent sequence constitutes the "fragment assembly problem". It is a permutation NP-hard problem [15].

3 SAX, a Hybrid Metaheuristic for Solving FAP

SAX is a low-level teamwork hybrid algorithm according the Talbi's classification [16], which combines ISA with a genetic crossover operator. This new metaheuristic unifies the good performance of ISA with the heuristic information obtained by using the OX crossover operator. This information guides the search toward regions of solution space where the high fitness values are combined with the optimal number of contigs.

As algorithm 1 shows, SAX creates the initial solution (S_0) using the greedy strategy proposed in [11] then, it uses the inversion mutation to generate two new and different solutions (S_1 and S_2) from S_0. S_3 arises by recombining S_1 and S_2 using OX, besides S_3 replaces the current solution if it is better than S_0 or if it is accepted under the Boltzman distribution.

In other words, SAX extends the exploration to generate two solutions from S_0, but it also increases the exploitation to recombine them. For illustrating the SAX working, we show an operating scheme of this new hybrid metaheuristic in Figure 3. On one hand, the inversion mutation increases the structural diversity of the current solution by inverting the substring between two random positions. As we can see, it is clearly an exploration operator that helps to discover new and promising regions of the search space.

Algorithm 1 SAX

```
k = 0;
init S_0 and T; {initial solution and temperature}
evaluate S_0 in E_0;
repeat
   repeat
      k = k + 1;
      generate S_1 from S_0 by applying the inversion mutation;
      {Pc is the crossover probability}
      if (random(0,1)< Pc) then
         generate S_2 from S_0 by applying the inversion mutation;
         generate S_3 applying OX to S_1 and S_2;
         S_1 = S_3;
      end if
      evaluate(S_1) in E_1;
      {if the new solution, S_1, is better than the current one, then S_1 is accepted}
      if (E_1 − E_0) ≥ 0 then
         S_0 = S_1;
         E_0 = E_1;
         {if the new solution is worst than the current one,
         S_1 is accepted under the Boltzman probability}
      else
         if exp((E_1 − E_0)/T) > random[0, 1) then
            S_0 = S_1;
            E_0 = E_1;
         end if
      end if
   until (k mod Length of Markov Chain) == 0
   update T;
until stop criterion is satisfied
return S_0;
```

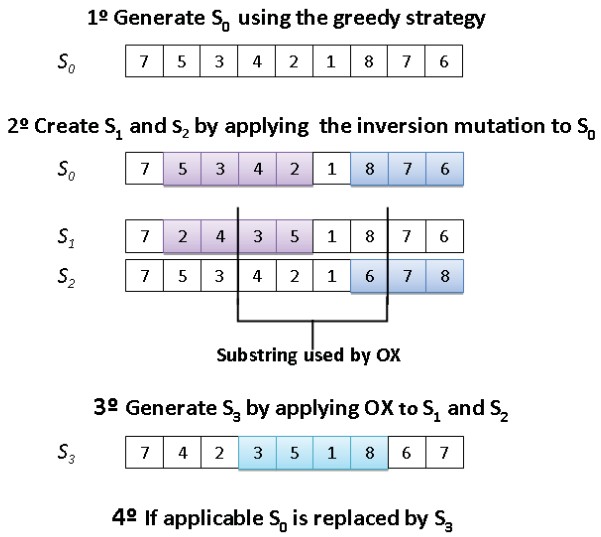

Fig. 2. SAX Operating Scheme.

On the other hand, SAX uses the OX operator to combine fragments of S_1 and S_2 and obtain an improved solution. The idea behind this information exchange is to implement a depth search or exploitation in order to improve the current solution.

In this way, SAX jumps to new and promising regions of the search space and carries out a very efficient exploitation. Consequently, unlike ISA, our assembler will be able to find optimal layouts with very high overlapping score. Besides, as SAX is a general purpose method, it can be used for solving other optimization problems with a permutation representation like the traveling salesman problem.

4 Data Sets and Experimentation Methodology

In this section, we describe the problem instances that have been used in the different experimentations, as well as the execution environment.

We have chosen four sequences from the NCBI web site[1]: a human MHC class II region DNA with fibronectin type II repeats HUMMHCFIB, with accession number X60189; a human apolopoprotein HUMAPOBF, with accession number M15421; the complete genome of bacteriophage lambda, with accession number J02459; and a sequence of Neurospora crassa BAC, with accession number BX842596 (GI38524243). We used GenFrag [4] to generate the different data sets from these sequences, shown in Table 1. GenFrag is a UNIX/C application created to accept a DNA sequence as input and generate a set of overlapping fragments as output that are used to test assemblers.

Furthermore, we have selected other sequences from the NCBI web site; they correspond to a human microbion bacterium ATCC 49176 with accession numbers from ACIN02000001 to ACIN02000026. Particularly, we have used the longer sequences from this genome, and we have fragmented them with DNAgen application. These new instances[2] are shown in the second part of the Table 1. The cutoff, which we have set to thirty (a very high value), provides one filter for spurious overlaps introduced by experimental error.

We use the MALLBA library [3] to implement each algorithm. The results are obtained after performing 30 independent runs on a AMD Phenom 8450-Triple core processors at 2 GHz and 2 GB RAM. The Operating System is Slackware Linux with 2.6.27 kernel version.

In order to get concluding results from the comparison made, we have performed some statistical tests on the results. Before performing the statistical tests, we first check whether the data follow a normal distribution by applying the Shapiro-Wilks test. When the data are normally distributed, we perform an ANOVA test. Otherwise, the tool we use is the Kruskal-Wallis test. This statistical study allows us to asses if there are meaningful differences among the compared algorithms with 95% confidence level.

The main goal of this research is to study if the proposed algorithm solves efficiently FAP, thus demonstrating its usefulness to act as a practical assembler.

[1] http://www.ncbi.nlm.nih.gov/
[2] http://mdk.ing.unlpam.edu.ar/ lisi/

In order to do that, we compare the SAX results with those obtained ones by other practical and relevant assemblers like ISA, PALS, and GAG_{50}.

Given the features from those four assemblers, we need to find a same stop criterion for the four algorithms to make a right comparison. For example, the numbers of iterations is not an adequate criterion to measure and compare the computational effort between trajectory methods like ISA, PALS, and SAX and population-based methods like GAG_{50}. The reason for this lies in the first ones works with only one solution by iteration, while the second one uses several solutions in each iteration. Otherwise, the number of fitness evaluations is another inappropriate termination condition because PALS does not calculate this value on the search.

Consequently, we choose the runtime as stop criterion. Thus, ISA, PALS, GAG_{50}, and SAX search their best solution during 60 seconds. In this way, we can measure the behavior of each assembler in a given period and compare with others under the same conditions. In Table 2 we resume the parametric settings of each one of these assemblers.

Table 1. Information of datasets. Accession numbers are used as instance names.

Instances	***Coverage***	***Mean Fragment Length***	***Number of Fragments***	***Original Sequence Length***
GenFrag Instances				
x60189_4	4	395	39	3835
x60189_5	5	386	48	
x60189_6	6	343	66	
x60189_7	7	387	68	
m15421_5	5	398	127	10089
m15421_6	6	350	173	
m15421_7	7	383	177	
j02459_7	7	405	352	20000
bx842596_4	4	708	442	77292
bx842596_7	7	703	773	
DNAgen Instances				
acin1	26	182	307	2170
acin2	3	1002	451	147200
acin3	3	1001	601	200741
acin5	2	1003	751	329958
acin7	2	1003	901	426840
acin9	7	1003	1049	156305

Table 2. Parametric values used by ISA, PALS, GAG_{50}, and SAX.

	Parameter	***Value***
ISA and SAX	Length of Markov Chain	10
	Initial temperature	0.99
PALS	Movement Selection	The Best Movements
GAG_{50}	μ	256
	λ	256
	Mutation Op. and its probability	*Swap* 0.2
	Parents Selection	Binary Tournament
	Replacement	The best μ solutions of $(\mu + \lambda)$
GAG_{50} *and SAX*	Recombination Op. and its probability	Order Crossover (OX) 0.7 for GAG_{50} and 1.0 for SAX
ISA, PALS, GAG_{50}*, and SAX*	Stop Criterion	60”

5 Analysis of Results

In this section we compare the performance of SAX against other metaheuristic assemblers like ISA, PALS, and GAG_{50}. Our aim is to offer an analysis from the accuracy (quality) and performance points of view. Hence, tables 3 and 4 show the best and the average values for the fitness and the percentage of optimal contigs found by the algorithm, and the average run time. The best results for each instance are in **boldface**. Additionally, we show the results of the statistical tests in the comparison of the different algorithms for each tested instance. Symbol + means that significant differences where found at the 95% confidentiality, and cells with distinct backgrounds in the same row means the respective assemblers behaves statically different for the given instance. For example, in Table 3, we see that the ISA and PALS behaviors are different from the GAG_{50} and SAX behaviors for instance *38524243_7*, while there are no significant differences between ISA and PALS and between GAG_{50} and SAX.

Table 3. Best and average fitness found by ISA, PALS, GAG_{50}, and SAX.

Instances	*Best fitness*				*Average Fitness*				*KW*
	ISA	PALS	GAG_{50}	SAX	ISA	PALS	GAG_{50}	SAX	*Test*
x60189_4	**11478**	**11478**	**11478**	**11478**	11332.90	11416.17	**11478.00**	11380.47	+
x60189_5	**14027**	14021	13553	**14027**	13872.40	13758.47	13275.50	**13999.20**	+
x60189_6	**18301**	**18301**	17866	**18301**	**18147.93**	17890.43	17414.90	18131.33	+
x60189_7	**21271**	21210	20884	21268	20913.10	20832.83	20737.23	**21070.37**	-
m15421_5	38583	38526	37932	**38726**	38474.17	38402.30	37506.83	**38557.97**	+
m15421_6	**48048**	**48048**	47152	**48048**	47891.70	**47925.13**	46788.87	47894.03	+
m15421_7	55048	55067	52702	**55072**	54702.47	54525.20	52277.93	**54789.90**	+
j02459_7	**116257**	115320	110869	115301	**115164.87**	114575.13	110223.87	114160.87	+
38524243_4	**226538**	225783	218250	223029	**225647.07**	224833.10	217186.55	221677.70	+
38524243_7	436739	**438215**	417702	417680	433482.17	**436645.40**	416011.13	415455.13	+
acin1	44511	**46876**	45565	46865	44258.94	**46758.20**	45377.03	46636.97	+
acin2	138699	**144634**	143444	144567	136719.33	**144112.00**	142499.33	143972.87	+
acin3	152177	**156776**	154947	155789	150193.90	**156507.50**	153980.90	154991.57	+
acin5	143864	**146591**	145332	145880	142770.50	**146563.80**	145193.87	145309.40	+
acin7	155117	**158004**	155873	157032	154977.53	**157972.80**	155801.40	156939.40	+
acin9	311035	**325930**	313203	314354	308603.93	**324620.30**	312005.55	311863.23	+

In the first place, we study the behavior of these assemblers taking into account the quality of solutions. In this sense, we analyze the fitness values and the percentage of optimal contigs for each instance (see tables 3 and 4) and we can notice several facts. ISA and SAX get the optimal layout (only one contig) in all instances, regardless of their sizes. But, for the larger instances (Acin group), the ISA fitness values are lesser than the obtained ones by PALS, GAG_{50}, and SAX. The difference between these two groups of algorithms is statistically significant as we can see in Table 3 (different backgrounds on the Average Fitness columns).

Table 4. Percentage of optimal layouts found by ISA, PALS, GAG_{50}, and SAX and the average time spent by these algorithms to find the best solution.

Instances	*% Optimal contigs*				*KW Test*	*Average time to find the best solution*				*KW Test*
	ISA	PALS	GAG_{50}	SAX		ISA	PALS	GAG_{50}	SAX	
x60189_4	**100.00%**	**100.00%**	**100.00%**	**100.00%**	-	0.01	**0.00**	0.22	0.21	-
x60189_5	**100.00%**	**100.00%**	**100.00%**	**100.00%**	-	0.04	**0.00**	0.41	0.99	-
x60189_6	**100.00%**	**100.00%**	**100.00%**	**100.00%**	-	0.10	**0.01**	0.69	1.46	+
x60189_7	**100.00%**	**100.00%**	**100.00%**	**100.00%**	-	0.07	**0.00**	0.62	5.32	+
m15421_5	**100.00%**	96.67%	90.00%	**100.00%**	-	0.54	**0.03**	2.57	8.17	+
m15421_6	**100.00%**	0.00%	0.00%	**100.00%**	+	0.79	**0.07**	6.24	12.99	+
m15421_7	**100.00%**	0.00%	0.00%	**100.00%**	+	1.15	**0.09**	6.08	14.07	+
j02459_7	**100.00%**	86.67%	10.00%	**100.00%**	+	8.77	**0.69**	25.94	58.17	+
38524243_4	**100.00%**	0.00%	0.00%	**100.00%**	+	16.88	**1.23**	46.04	58.96	+
38524243_7	**100.00%**	0.00%	0.00%	**100.00%**	+	30.46	**1.64**	47.36	59.25	+
acin1	96.77%	0.00%	0.00%	**100.00%**	+	6.29	**0.39**	29.56	55.38	+
acin2	**100.00%**	0.00%	0.00%	**100.00%**	+	15.44	**2.42**	58.30	59.42	+
acin3	**100.00%**	0.00%	0.00%	**100.00%**	+	17.73	**6.50**	59.42	59.49	+
acin5	**100.00%**	0.00%	0.00%	**100.00%**	+	26.11	**19.30**	59.82	59.73	+
acin7	**100.00%**	0.00%	0.00%	**100.00%**	+	35.57	**33.16**	59.98	59.63	+
acin9	**100.00%**	0.00%	0.00%	**100.00%**	+	43.78	39.84	**2.27**	59.77	+

Furthermore, as we can see in Table 4, PALS and GAG_{50} cannot find the optimal layout for the largest instances, although PALS can achieve similar (and sometimes better) fitness values than ISA and SAX for those instances. These results confirm that ISA and SAX are better than PALS and GAG_{50} for solving all instances, regardless of their sizes.

Given the aforementioned facts, we infer that the optimal number of contigs is not necessarily associated with a high overlapping score. ISA needs to sacrifice this score to obtain the optimal layout. This happen when ISA accepts worse solutions than the current one depending on an accepting probability. In this way, ISA escapes from local optima, explores new regions of the search space, and finds the optimal layout.

However, SAX, which also accepts worse solutions than the current one, achieves both objectives through the OX operator. This crossover operator enables SAX to make a better exploitation of the new-found regions. In other words, SAX achieves the optimal layout of fragments with a high overlapping score.

In a second place, we analyze the assembler behavior considering the execution time. If we look at the average time that is necessary to find the best solution in Table 4, we can see that ISA highlights a direct relationship between the instance size and the time to find the best solution.

Moreover, PALS is the least consuming-time assembler. However, considering the results obtained in the experimentation, we infer that PALS converges quickly a local optima. Besides, GAG_{50} spends the total time allocated when it solves the larger instances. This means that GAG_{50} needs more time to find a

better solution. Although SAX also requires almost 60 seconds for solving the larger instances, it can find optimal layouts with very high fitness values for these instances.

As to the resulting ranking according to the solution quality and the execution time, SAX is in the first place because it outperforms ISA, PALS, and GAG_{50} in 100% of the largest instances. SAX is followed by ISA, after that we can find PALS and finally GAG_{50}.

6 Comparison Against Existing Assemblers

In this section we compare the performance of our approaches against other assembler algorithms proposed in the literature: CAP3 [5] and PHRAP (http://www.phrap.org). These packages automate fragment assembly using a variety of techniques that are greedy-based. In the next paragraphs we briefly describe these assembler programs.

CAP3 is an extended version of CAP2 and CAP. CAP performs three different tasks: fragment overlap detection, contig formation, and consensus sequence generation. During the first task, this assembler computes the overlaps between every pair of input fragments. After that, contigs are assembled in a greedy fashion, by adding the best overlapping fragment (found in the previous fragment overlap detection phase) one at a time. Finally, CAP generates consensus sequences from the contigs, CAP merges each pairwise alignment into a multiple alignment of fragments. CAP2 improves CAP since it filters out potentially non-overlapping fragments called singlets, identifies chimeric fragments, and handles repetitive sequences. CAP3, a pretty specialized assembler, has a capability to clip 5' and 3' low-quality regions of reads, uses base quality values in computation of overlaps between reads, and uses clone-mate information (forward-reverse constraints) to correct assembly errors and link contigs.

PHRAP is an assembly program that performs comparison, alignment, and assembly of large sets of DNA sequences. PHRAP compares sequences by searching for pairs of perfectly matching words or sequence regions that meet certain criteria. If a match is found, PHRAP then tries to extend the alignment into contigs. PHRAP uses quality values produced by the PHRED base-caller that reads DNA sequence chromatogram files and analyzes the peaks in it to call bases, assigning quality scores to each base call.

In order to make that comparison, we show in Table 5 the best number of contigs achieved by ISA, PALS, GAG_{50}, SAX, and CAP3 for instances of Table 1. We also show the results reached by PHRAP only for some instances: *x60189*, *m15421*, *j02459* and *38524243*. This happens because the process to generate the noisy instances studied in this work does not consider the generation of the associated chromatograms which are necessary to apply PHRAP.

If we analyze Table 5, we can see that ISA and SAX outperforms PALS, GAG_{50}, CAP3, and PHRAP for solving all instances, regardless their sizes. Besides, we can detect similar behaviors among PALS, GAG_{50}, CAP3, and PHRAP given that they can not find optimal layouts for the same instances,

Table 5. Best final number of contigs for of ISA, PALS, GAG_{50}, SAX, CAP3, and PHRAP. Symbol - indicates that this information can not be computed.

Instances	*ISA*	*PALS*	*GAG_{50}*	*SAX*	*CAP3*	*PHRAP*
x60189_4	**1**	**1**	**1**	**1**	**1**	**1**
x60189_5	**1**	**1**	**1**	**1**	**1**	**1**
x60189_6	**1**	**1**	**1**	**1**	**1**	**1**
x60189_7	**1**	**1**	**1**	**1**	**1**	**1**
m15421_5	**1**	**1**	**1**	**1**	3	**1**
m15421_6	**1**	2	2	**1**	2	-
m15421_7	**1**	2	2	**1**	**1**	2
j02459_7	**1**	**1**	**1**	**1**	**1**	**1**
38524243_4	**1**	6	6	**1**	8	6
38524243_7	**1**	3	3	**1**	2	2
acin1	**1**	4	5	**1**	9	-
acin2	**1**	236	236	**1**	239	-
acin3	**1**	358	358	**1**	361	-
acin5	**1**	552	552	**1**	556	-
acin7	**1**	722	722	**1**	727	-
acin9	**1**	552	552	**1**	552	-

e.g., *m15421_6*, *m15421_7*, *38524243_4*, and the whole Acin group. Furthermore, we also should put special emphasis in making a comparison between CAP3 and GAG_{50}, since the number of contigs obtained by CAP3 is equal or larger than the obtained ones by GAG_{50}. This indicates that GAG_{50} designs better layouts of fragments than CAP3. As to the resulting ranking according to the numbers of contigs, ISA and SAX are in the first place, followed by PALS, GAG_{50}, CAP3, and finally PHRAP.

7 Conclusions

In this paper we present a new hybrid metaheuristic for solving FAP based on Simulated Annealing, called SAX. This new development was born to hybridize a SA-based assembler (ISA) with a genetic crossover operator (OX). The ISA component helps SAX to escape from local optima and OX expresses the ability of the algorithm to reach the best local solutions within the search space. The quality of the results found by SAX clearly outperforms ISA, PALS, GAG_{50}, CAP3, and PHRAP.

Among the main future research lines, we propose the fine tune of the SAX parametric configuration to reduce the computational time of our algorithm and in order to face larger instances with true guarantee of success. Besides, as SAX is a metaheuristic, an interesting work is to analyze its application to other NP-complete problems with a permutation representation.

Acknowledgements

This work has been partially funded by the Spanish Ministry of Science and Innovation and FEDER under contracts TIN2008-06491-C04-01 (the MSTAR project) and TIN2011-28194 (the roadME project), and by the Andalusian Government under contract P07-TIC-03044 (the DIRICOM project). Gabriela

Minetti and Guillermo Leguizamón are supported by the Universidad Nacional de La Pampa, Universidad Nacional de San Luis, and the ANPCYT.

References

1. E. Alba and G. Luque. A New Local Search Algorithm for the DNA Fragment Assembly Problem. In *Evolutionary Computation in Combinatorial Optimization, EvoCOP'07*, volume 4446 of *Lecture Notes in Computer Science*, pages 1–12. Springer, Valencia, Spain, 2007.
2. E. Alba, G. Luque, and G. Minetti. Variable neighborhood search for solving the DNA fragment assembly problem. In *Anales del XIII Congreso Argentino de Ciencias de la Computacin (CACIC)*, pages 1359 – 1370, Corrientes y Resistencia, Argentina, October 2007.
3. E. Alba, G. Luque, J.M. Nieto, G. Ordónez, and G. Leguizamón. MALLBA: A Software Library To Design Efficient Optimization Algorithms. *International Journal of Innovative Computing and Applications*, 1(1):74–85, 2007.
4. M.L. Engle and C. Burks. Artificially generated data sets for testing DNA FA algorithms. *Genomics*, 16, 1996.
5. W. Huang and A. Madan. CAP3: A DNA Sequence Assembly Program. *Genome Research*, 9(9):868–877, 1999.
6. L. Li and S. Khuri. A Comparison of DNA Fragment Assembly Algorithms. In *Proc. of the 2004 International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences*, pages 329–335, Las Vegas, 2004.
7. G. Luque and E. Alba. Metaheuristics for the DNA Fragment Assembly Problem. *International Journal of Computational Intelligence Research*, 1(2):98–108, 2005.
8. G. Luque, E. Alba, and S. Khuri. Chapter 16: Assembling DNA Fragments with a Distributed Genetic Algorithm. In *Parallel Algorithms for Bioinformatics*. Wiley, 2006.
9. P. Meksangsouy and N. Chaiyaratana. DNA fragment assembly using an ant colony system algorithm. In *The 2003 Congress on Evolutionary Computation, 2003. CEC03*, volume 3, pages 1756– 1763. IEEE. ISBN: 0-7803-7804-0, 2003.
10. G. Minetti. *Problema de ensamblado de fragmentos de ADN resuelto mediante metaheurísticas y paralelismo*. PhD thesis, Universidad Nacional de San Luis, November 2011.
11. G. Minetti, E. Alba, and G. Luque. Seeding strategies and recombination operators for solving the DNA fragment assembly problem. *Information Processing Letters*, 108(3):94–100, October 2008.
12. G. Minetti, G. Leguizamón, and E. Alba. A new Parallel and Hybrid Metaheuristic for Solving Noisy DNA Strands. *Journal of Information Sciences, Elsevier (in evaluation)*, 2011.
13. G. Minetti, G. Leguizamón, and E. Alba. Assembling DNA Sequences Containing Noisy Information With Metaheuristic Algorithms. *Journal of Information Sciences, Elsevier (in evaluation)*, 2011.
14. G. Minetti, G. Luque, G. Leguizamón, and E. Alba. A new Hybrid SA for Solving the DNA Fragment Assembly Problem. In *XXVIII Internacional Conference of the Chilean Computing Science Society (SCCC)*, pages 109 – 116, Santiago, Chile, November 2009.
15. P. Pevzner. *Computational molecular biology: An algorithmic approach*. The MIT Press, 2000.
16. El-Ghazali Talbi. *Metaheuristics - From Design to Implementation*. Wiley, 2009.