Metaheuristics Assemblers of DNA strands: Noiseless and Noisy Cases

Gabriela Minetti and Enrique Alba

Abstract—The DNA fragment assembly problem is an NP-complete problem which has been solved efficiently by many metaheuristics. However, those techniques generally assemble fragments that belong to noiseless DNA sequences. But nowadays dealing with noisy instances is imperative. For that we analyse exhaustively how noiseless and noisy instances of this problem are dealt by three efficient algorithms (Problem aware local search, Simulated Annealing and Genetic Algorithms). This analysis includes a performance evaluation of those algorithms to assemble fragments and a study of the solution composition. From these analysis we observe that the GA is more robust in presence of noise than the other two searches, while it usually does not improve the accuracy of results for large instances (where Simulated Annealing is the more precise technique).

I. INTRODUCTION

The fragment assembly problem (FAP) consists of building the DNA sequence from several hundreds (or even, thousands) of fragments which are obtained by biologists in the laboratory. This is an important task 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 problem.

However this is not the only difficulty that we have to face. Another important obstacle is the noise which can be produced by the method used for sequencing long DNA strands in the laboratory. Thus the data that we use for assembling the fragments can contain errors.

Many metaheuristic assemblers like Simulated Annealing (SA), Variable Neighbourhood Search (VNS), Genetic Algorithms (GA), Ant Colony Optimization (ACO), and Problem Aware Local Search (PALS) solve the fragment assembly problem in a relatively efficient way [1]–[7]. Generally speaking, these past lines of research only reported on results using noiseless problem instances. On the contrary, there are only a few researches that have handled noisy in some stage of problem in the literature [8]–[10]. Since in a genome project most tasks depend on the accuracy (and efficiency) of the sequences which are obtained by the assembler, an appropriate processing of noisy instances is imperative, and this is our goal in this article.

For our purposes, we should simulate the noise over a set of known noiseless instances changing uniformly their score values. We include noise in the score matrix representing the overlapping between any pair of fragments; by doing so we represent and account in a single and simple way all the potential kinds of mutations, deletions, insertions, and errors found in Nature through their resulting numerical effects in the data of this problem. Hence we select three of the most used and efficient metaheuristics for solving the noisy FAP: SA, GA and PALS. In order to do that we compare the behaviour of those algorithms for noisy and noiseless cases at different stage of each algorithm. Thus we analyse and compare the found results (quality and execution time) for each algorithm in each case. But we also have to study how solutions are formed at different stages of the search. Therefore we need to know the relative frequency of a fragment $i$ precedes a fragment $j$ during the search process. This information allows us recognize how an algorithm behaves in noiseless and noisy situations.

The rest of this article is organized as follows. In the next section we describe the Fragment Assembly Problem. In Section 3, we introduce a theoretical base for GA, SA and PALS algorithms. In the fourth section we explain their implementations. We show the results from experiments performed and analyse them in the fifth section. Finally, we discuss the conclusions and provide hints to further research.

II. THE DNA FRAGMENT ASSEMBLY PROBLEM

Shotgun sequencing is a method used for sequencing long DNA strands, which consists in [11]:

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 reads and reconstructs the original sequence.

The assembling of DNA fragments is divided into three different phases: overlap phase (finding the overlapping among fragments -score-), layout phase (finding the order of fragments based on computed similarity scores), and consensus phase (deriving the DNA sequence from the layout). At the assembly stage, the only information available is the sequences of bases, and thus the ordering of the fragments must rely primarily on the similarity of fragments and on how they overlap. An important challenge of the general sequencing problem is to determine the relationship and orientation of the fragments. Another important issue is the incomplete coverage provoked when the algorithm is not able to assemble a given set of fragments into a single contig. A contig is a layout consisting of contiguous overlapping fragments.

Once the fragments have been ordered (layout), the final consensus is generated. This process includes a detailed alignment step that must consider the insertion and deletion...
errors potentially present in the data. 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. It is computed as the number of bases read from fragments over the length of the target DNA [12]:

\[
\text{Coverage} = \frac{\sum_{i=1}^{n} \text{length of the fragment } i}{\text{target sequence length}}
\]  

where \( n \) is the number of fragments. The higher the coverage, the fewer the number of the gaps, and the better the result.

Particularly, the assembly of DNA fragments into a consensus sequence corresponding to the parent sequence constitutes the “fragment assembly problem” [12]. It is a permutation NP-hard problem [13].

III. OPTIMIZING WITH SA, GA AND PALS

In this section, we introduce the theoretical base corresponding to SA, GA and PALS algorithms.

A. Simulated Annealing

Simulated Annealing is a simple and general purpose Monte-Carlo method which was developed for combinatorial optimization in the early 1980s [14]. This metaheuristic algorithm is based on an analogy to physical systems. Annealing is a thermal process for obtaining low energy states of a solid in a heat bath. Simulated Annealing can be executed at decreasing values of a control parameter (temperature). Where the solutions for a combinatorial optimization problem are considered as states of the physical systems and the cost of a solution is equivalent to the energy of a state.

In other words, at the first steps (at high temperature), SA accepts solutions with higher costs under a certain probability in order to explore the search space and to escape from local optimaums. During the annealing process this probability decreases according to temperature cooling; intensifying the search and reducing the exploration in order to exploit a restricted area of a search space. In Algorithm 1, we show a pseudo-code of a basic Simulated Annealing Algorithm.

Simulated annealing evolves by a sequence of transitions between states and these transitions are generated by transition probabilities. Consequently SA can be mathematically modelled by Markov chains, where a sequence of chains is generated by a transition probability which calculation involves the current temperature.

B. Genetic Algorithms

Genetic Algorithms were first introduced by Holland [15] in 1975. They simulate the evolution of individual structures via the Darwinian natural selection process.

As the Algorithm 2 shows, a GA maintains a population of multiple tentative solutions (individuals) which evolve throughout generations by reproduction of the fittest ones.

![Algorithm 1 Pseudo-code of Simulated Annealing Algorithm](image1)

![Algorithm 2 Pseudo-code of Genetic Algorithm](image2)

Selection, recombination, and mutation are the main operators used for modifying individual features. So, it is expected that evolved generations provide better and better individuals (tentative solutions in the problem space).

Selection mechanisms favor reproduction of better individuals imposing a direction on the search process. This process does not create new individuals, it selects comparatively good individuals from a population for mating. The idea is to generate a competition among individuals with higher fitness because they have a higher probability to be selected for mating. The fitness of an individual gives a measure of its goodness.

The crossover operation tries to recombine good characteristics from different parents selected in order to yield a new individual. The main idea is the crossovers ability to combine and/or disrupting pieces of information.

The mutation operator is used for incorporating diversity into the population. Thus the GA can avoid a premature convergence to a local optimum. The simplest form of mutation is called swap, and consists of random interchanges on alleles.

After applying the crossover and mutation operators, a new group of individuals is obtained, which are assessed to form the population of the new generation. Unlike selection methods, where individuals are selected to be recombined, the replacement methods choose individuals for the next
Algorithm 3 Pseudo-code of Problem Aware Local Search

```
initialize S0; \{generate the initial solution\}
repeat
    L = Ø;
    for i = 0 to N do
        for j = 0 to N do
            \(\Delta_c, \Delta_f = \text{CalculaeDelta}(s,i,j);\) \{see Alg. 4\}
            if \(\Delta_c \leq 0\) then
                \(L = L \cup \langle i, j, \Delta_c, \Delta_f \rangle;\) \{Add candidate movements to L\}
            end if
        end for
    end for
    if \(L < \emptyset\) then
        \(<i, j, \Delta_c, \Delta_f> = \text{Select}(L);\) \{Select a movement from L\}
        \(\text{ApplyMovement}(s,i,j);\) \{Modify the solution\}
    end if
    until no changes
return s;
```

Algorithm 4 CalculaeDelta function

```
\(\Delta_c = 0;\)
\(\Delta_f = 0;\)
(Calculate the variation in the overlap)
\(\Delta_f = w_{[i,j],s[i,j]} + w_{s[i,j],s[j+1]};\) \{Add the overlap of the modified solution\}
\(\Delta_f = \Delta_f - w_{s[i,j],s[i]} - w_{s[i,j],s[j+1]};\) \{Remove the overlap of the current solution\}
(Test if a contig is broken, and if so, it increases the number of contigs)
if \((w_{[i,j],s[i,j]} > \text{cutoff}) || (w_{s[i,j],s[j+1]} > \text{cutoff})\) then
    \(\Delta_c = \Delta_c + 1;\)
end if
if \((w_{s[i,j],s[i]} > \text{cutoff}) || (w_{s[i,j],s[j+1]} > \text{cutoff})\) then
    \(\Delta_c = \Delta_c + 1;\)
end if
return \(\Delta_c, \Delta_f;\)
```

population.

C. Problem Aware Local Search

Problem Aware Local Search is a variation of Lin’s 2-opt which is proposed by Luque et. Al in [7]. This heuristic does not only use the overlap among the fragments, but it also takes into account the number of contigs that have been created or destroyed. The pseudo-code of PALS is shown in Algorithm 3.

The main advantage in PALS is the calculation of the variation in the overlap \(\Delta_f\) and in the number of contigs \(\Delta_c\) between the current solution and the resulting solution of applying a movement (see Algorithm 4). This calculation is not computationally expensive since neither the fitness function nor the number of contigs are calculated, but instead the variation of these values are estimated.

IV. IMPLEMENTATION DETAILS

In this section, we describe the implementation details for these three algorithms (SA, PALS, GA) to solve the DNA fragment Assembly Problem. In a first place we present the characteristics which are present in all algorithms and then we describe the particularities of each algorithmic option.

A. Common Issues

1) Solution Representation: In our proposal, the solution representation is a permutation with integer number encoding. This permutation represents a sequence of fragment numbers, where successive fragments overlap. Consequently each fragment is represented by an unique integer ID. The permutation representation requires special operators to make sure that we always get legal (feasible) solutions. In order to maintain a legal solution, the two conditions that must be satisfied are all fragments must be presented in the ordering, and no duplicate fragments are allowed in the ordering.

2) Initial Solution: Since using seeding strategies to generate initial solutions can help exploit promising regions of the search space, we use a constructive heuristic, developed in [16], to create initial solutions for FAP. The idea behind this greedy approach is to generate solutions by adding appropriately selected solution components (fragments) to an initially empty partial solution.

3) Fitness Function: The used fitness function, \(F(l)\), is proposed by Parsons, Forrest, and Burks in [2]. This function sums the overlap score for adjacent fragments in a given solution. This function maximizes the score. It means that the best individual will have the highest score.

\[
F(l) = \sum_{i=0}^{n-2} w(f[i], f[i+1])
\]

where \(w(i,j)\) is the pairwise overlap strength of fragments \(i\) and \(j\). The overlap score \(w\) in \(F\) is computed using the semiglobal alignment algorithm, a classical dynamic programming technique to calculate the pairwise alignments.

4) Stop criterion: We define as the stop condition for the compared algorithms to run for the same real execution time. This means that SA, GA, and PALS, all of them, will have 60 seconds of execution time and then we report on their attained accuracy. This is mandatory since some algorithms (like PALS) have no any concept similar to an evaluation (this also happens with constructive algorithms like ACO, not used in this work). Then, using as the stop criterion the run time is the only fair way to test these techniques. In fact, since Bioinformatics is dealing with that large sets of data in general, real time is a critical restriction in present laboratories. This stopping time (60 s) and the remaining parameters have been previously “optimized” through a preliminary hand-tuning process comparing it against other different values.

B. Simulated Annealing Details

1) Markov chain length \(k\): determines the length of a state sequence in which each state only depends on the previous one. Particularly, we use a length of 10.

2) Cooling schedule: defines how the temperature is decreased at each stage. We use the proportional cooling scheme:

\[
T_{k+1} = \alpha \ast T_k
\]

where \(\alpha\) is a constant close to, but smaller than, 1. Particularly, we calculate \(\alpha\) as follows:

\[
\alpha = \frac{k}{k+1}
\]
C. Genetic Algorithm Details

1) Population Initialization: Since having “good” solutions (seeds) in the first steps of the method help exploit promising regions of the search space, we use a seeding strategy (see section IV-A.2) to initialize the fifty percent (128 individuals) of the population. Meanwhile the rest of population is generated in a random way.

2) Recombination Operator: We work with a specially designed recombination operator for permutation representation to avoid infeasible solutions. The operator considered in this study is Order crossover [17] and its probability of application is 0.7.

3) Mutation Operator: In this work, we use the swap mutation operator with an application probability of 2%. This operator works in the same way than the SA neighborhood function.

4) Selection Operators: We use two different selection methods. The binary tournament selection is used to choose the individuals to be recomposed. While, for building the population for the next generation we select the best 256 individuals from parent and offspring populations.

D. Problem Aware Local Search

1) Movement Selection: the best movements were chosen. We choose the movement having the lowest $\Delta_c$ and the highest $\Delta_f$.

V. Experimental Results

In this section we analyse the behaviour of three above-described methods for solving noiseless and noisy FAP instances. This study consists of two parts. At the first one, we summarize the results obtained by SA, PALS and GA for solving each problem instance. Our aim is to offer an analysis from the final result quality and performance points of view. The quality point of view involves an analysis about numbers of contigs and fitness values from best found final solutions. But this analysis also compare the probability of a fragment $i$ preceeds a fragment $j$ at the final permutations for both kinds (noiseless and noisy) of instances. In the second part, we present a detailed study of these probabilistic values during the execution.

We have chosen four sequences from the NCBI web site\(^1\): a human HMC class II region DNA with fibronectin type II repeats HUMMHCFIB, with accession number X60189; a human apolipoprotein HUMAPOBF, with accession number M15421; the complete genome of bacteriophage lambda, with accession number J02459; a sequence of Neurospora crassa BAC, with accession number BX842596 (GI38524243). We used GenFrag [18] to generate the different data sets from these sequences, and we shown them in Table I. GenFrag is a UNIX/C application created to accept a DNA sequence as input and to generate a set of overlapping fragments as output, in order to test any assembly application. Besides we have selected other sequences from the NCBI web site, and they correspond to a human microbion bacterium ATCC 49176 with accession numbers from ACIN02000001 to ACIN02000006. Particularly, we have used the longer sequences from this genome, and we have fragmented them with DNAgen application. These new instances are shown in the second part of the Table I.

This set of instances only represents noiseless data, while our goal is to test how an algorithm can solve the same instance of the problem in two different scenarios: with and without noise. Then, in order to obtain the noisy data, we simulate the noise by changing the score for each instance randomly. In other words, we choose and change some values inside each score matrix (overlapping information) in a uniform way. After this, we obtain a new score for each instance with included noise in it. In summary, we have 16 new instances by using this process, that from now on we will call noisy instances.

A. Analysis of Results

This section studies the performance of each algorithm for solving noisy and non-noisy instances. Hence in Table II, we present the best number of contigs and fitness obtained by SA, PALS and GA algorithms in the 30 runs per each noiseless instance. In tables III and IV, we show the best number of contigs and fitness obtained by those algorithms in 30 executions per each noisy instance. Note that, when SA, PALS and GA algorithms solve noisy instances get noisy solutions. Therefore if the contig number from one of these solutions is one, it does not mean this solution is an optimum neither its respective fitness value is right. So, we need to evaluate the noisy permutation (or solution) using the noiseless score for finding the real values of contig and fitness. The results from this second evaluation appears between parenthesis in tables III and IV, respectively. While in Table V, we report the minimal time that each algorithm requires to find its best solution for each noisy and non-noisy instance. In each table, we remark the best values in foldface. Finally, we explain how the best final solutions are formed by each algorithm.

\(^1\)http://www.ncbi.nlm.nih.gov/
If we interpret the results in Table II we can notice several facts. As we expected, SA and PALS algorithms get an optimal layout (only one contig) in all instances, regardless of their sizes. Moreover, SA fitness values are better than PALS and GA ones. Surprisingly, the Genetic Algorithm cannot find this optimum in the longer instances. Although, it can achieve similar (sometimes better) fitness values than SA and PALS for those instances.

From Table III, we observe again SA and PALS always get optimal layouts when noisy solutions are evaluated using noisy scores. While the GA difficulties to find optimums for longer instances persist. Nevertheless, the noisy fitness obtained by the GA are greater than the found ones by SA and PALS. The second evaluation about noisy solutions, that appears between parenthesis in tables III and IV, give us a different perspective of SA, PALS and GA behaviours. Numerically speaking, only a 32% of optimal noisy solutions found by PALS and SA have really one contig. But for the GA, when the noisy solutions are evaluated with the noiseless score we find two optimal solutions more. Besides, the difference between the real and noisy contig numbers is 0 in the 81% of the instances which are solved by GA. But PALS and SA reach a null difference only in the 32% of the instances. Therefore, we can infer the GA withstands better the noise than SA and PALS. In other words, the GA is the most robust in presence of noise.

Now, if we compare the real and the noisy fitness values from Table IV, we can see that real fitness values are smaller than the noisy fitness found by each algorithm in a 100% of the instances. However, GA finds better fitness values in 12 of 16 noisy instances.

If we examine the minimal time which is necessary to find the best solution in Table V, we can see the three algorithms take approximately the same execution time for solving noisy and noiseless instances. Besides when SA or GA approaches are used we can notice their execution times grow in a proportionate way to the instance sizes. However, by side, PALS always gets the best fitness values at the end of the execution (after 60 seconds). Because, this algorithmic approach only evaluates the permutation at the end of execution.

Finally, we explain how the behaviour from different
algorithms affects the final permutation composition for both kinds (noiseless and noisy) of instances. For that, we study the probability of a fragment \( i \) precedes a fragment \( j \) for all final permutations which were found by every algorithm in each instance. For example, if one of this precedence relationship (or ordered pair of fragments) has a probability equal to 1, this ordered pair is present in all permutations which were found by a certain algorithm for some instance. Or if the probability is 0.1, the respective pair of fragments is present in a 10% of those permutations.

When the GA solves a noisy or noiseless instance, it preserves between a 5 and a 10% of ordered pairs in the 30 best final permutations (for example for j02459.7 instance is 7%). That means the probability of those pairs is 1 and they are part of all final solutions which are found by GA. Meanwhile for PALS and SA this percentage is lesser than 2 and for noisy instances is zero in almost all cases. This difference is another proof of the GA robustness, although any of these three algorithms solves efficiently the large noisy instances.

In order to summarize the results and get some useful conclusions we present in Table VI two rankings with the algorithms ordered from best to worst. These rankings have been made considering an average of percentages from optimal layouts and best fitness which are found by each algorithm.

On the one hand, in the column Noiseless Cases of Table VI we can see that SA solves accurately all noiseless instances; meanwhile the GA behaviour is the worst. On the other hand, in column entitled Noisy Cases, GA outperforms SA and PALS behaviours.

### Table VI

<table>
<thead>
<tr>
<th>Noiseless Cases</th>
<th>Noisy Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rank</td>
<td>Algorithm</td>
</tr>
<tr>
<td>1</td>
<td>SA</td>
</tr>
<tr>
<td>2</td>
<td>PALS</td>
</tr>
<tr>
<td>3</td>
<td>GA</td>
</tr>
</tbody>
</table>

In figures 1, 2 and 4, we plot the solution composition for noiseless and noisy instances at different stages of PALS, SA and GA searches respectively. In those figures for each probability interval, a bar represents the percentage from ordered pairs of fragments with a certain probability to be part in the solutions during a specific search stage. For example in Figure 1, the white bar in the first interval means that approximately an 18% from ordered pairs of fragments (from non-noisy instance) can hold constant up to 20% of solutions at first iteration. However the white bar in the last interval means a 20% of ordered pairs keeps fixed in almost all solutions from first iteration.

In figures 3 and 5, we draw the fitness evolution for noisy and non-noisy instances during SA and GA searches, respectively. Note that instances plotted in the five figures are representative of the remaining ones.

As we can see in Figure 1, tests where PALS solves noiseless and noisy instances show an important difference between both of them cases. During the noiseless search, almost a 20% of ordered pairs are part up to a 60% of the intermediate solutions.

For noisy cases, the probability to keep constant an ordered pair during the search is low (< 0.2). Therefore, we can see that search is almost continuously changing the precedence relationships. This produces a negative effect in the quality of solutions since PALS cannot find real optimal layouts in a 68% of the noisy instances (see tables III and IV).

As PALS algorithm does not use the fitness calculation during the execution, we cannot study how this value evolves at the runtime.

In figures 1, 2 and 4, we plot the solution composition for noiseless and noisy instances at different stages of PALS, SA and GA searches respectively. In those figures for each probability interval, a bar represents the percentage from ordered pairs of fragments (from non-noisy instance) can hold constant up to 20% of solutions at first iteration. However the white bar in the last interval means a 20% of ordered pairs keeps fixed in almost all solutions from first iteration.

In figures 3 and 5, we draw the fitness evolution for noisy and non-noisy instances during SA and GA searches, respectively. Note that instances plotted in the five figures are representative of the remaining ones.

As we can see in Figure 1, tests where PALS solves noiseless and noisy instances show an important difference between both of them cases. During the noiseless search, almost a 20% of ordered pairs are part up to a 60% of the intermediate solutions.

For noisy cases, the probability to keep constant an ordered pair during the search is low (< 0.2). Therefore, we can see that search is almost continuously changing the precedence relationships. This produces a negative effect in the quality of solutions since PALS cannot find real optimal layouts in a 68% of the noisy instances (see tables III and IV).

As PALS algorithm does not use the fitness calculation during the execution, we cannot study how this value evolves at the runtime.

---

#### Fig. 1. Probability distribution from ordered pairs of fragment for each 38524243.7 case at different PALS stages

For testing how SA behaves on noisy and noiseless instances we observe similar behaviour patterns. In both cases all ordered pairs keep constant up to 20% of solutions throughout execution. Therefore none precedence relationship is present in the solution from the beginning to the end of search process. In Figure 2, we show an example of those patterns.
Fig. 2. Probability distribution from ordered pairs of fragment for each Acin2 case at different SA stages

Fig. 3. Fitness values during SA executions for noisy and noiseless Acin2

If we compare the number of ordered pairs have been used in each case (noisy versus non-noisy), we find the SA uses a 70% fewer pairs in the noisy cases. Thus when SA solves noisy cases, the exploration decreases in a considerable way. This produces a negative effect in the quality of intermediate and final solutions. This effect is easily corroborated in the Figure 3 where we can see that noisy fitness values are much smaller and its growth rate is lower than for noiseless case.

Finally we analyse the GA behaviour for noiseless and noisy instances. As we can see in Figure 4, we cannot set main differences between the two GA behaviours. Unlike PALS and SA, GA uses a small percentages of ordered pairs during most of the search. Therefore GA keeps constant some precedence relationships throughout execution. Note that characteristic is present in the best final solutions found by GA (see Section V-A). But nevertheless, most precedence relationships has a low probability \((<= 0.2)\). Thus those ordered pairs can keep constant up to 20% of intermediate solutions.

Due to the behavior pattern for solving both kind of instances are similar, their respective fitness evolution are similar too (see Fig. 5). At last, if we count the number of ordered pairs used in noiseless instances, we notice that is slightly higher (1.5 or 2%) than in the noisy case.

---

<table>
<thead>
<tr>
<th>Rank</th>
<th>Algorithm</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GA</td>
<td>85</td>
</tr>
<tr>
<td>2</td>
<td>PALS</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>SA</td>
<td>20</td>
</tr>
</tbody>
</table>

TABLE VII

RANKING OF THE ALGORITHMS CONSIDERING ROBUSTNESS IN THE PRESENCE OF NOISE

Fig. 5. Fitness values during GA executions for noisy and noiseless x60189_7

In order to determine what algorithm withstands noise better, we present a ranking in Table VII with the algorithms ordered from best to worst. This ranking has been made to average the percentage of probabilistic intervals where the percentage of ordered pairs is greater than 0.
From Table VII we notice that GA withstands noise better than PALS and SA. Comparing the two last algorithms, we see this ranking favours to PALS but the difference between them is not significant.

VI. CONCLUSIONS

In this paper we present an exhaustive analysis about how noiseless and noisy instances of FAP are dealing by three from the most efficient algorithms (PALS, SA and GA) for solving this problem.

When we compare the algorithms considering optimal layouts and best fitness found for non-noisy instances, we detect SA is the best to solve them. The following in the rank is PALS that always find the optimal layouts but their fitness are smaller. Finally is GA that cannot find optimal layouts for larger instances.

However when this comparison is made on noisy instances, we note GA outperforms SA and PALS. Due to GA finds more real optimal layouts and best fitness than the other two; besides to have a null difference between the real and noisy contig numbers of a 81% against a 32% of SA and PALS. But nevertheless the last three algorithms present difficulties for solving large noisy instances.

When we analyse the algorithmic behaviours throughout search, we study the solution composition and its fitness values at first, intermediate and finals iterations. From this analysis we detect GA is the more robust in presence of noise than SA and PALS. Due to GA keeps small percentages of precedence relationships throughout most search; while SA and PALS cannot keep none. They are continuously changing these relationships. Moreover when SA solves noisy instances its exploration is decreased.

As a future work, it may be interesting to combine the SA and PALS advantages to find good results with the GA characteristic to withstand noise. Henceforth to design a new algorithm for solving efficiently large noisy instances of FAP.

ACKNOWLEDGMENTS

The Argentine author is supported by the Universidad Nacional de La Pampa, and the ANPCYT.

The Spanish author acknowledges funds from the Spanish Ministry of Sciences and Innovation European FEDER under contract TIN2008-06491-C04-01 (M* project, available in URL http://mstar.lcc.uma.es) and CICE, Junta de Andalucía under contract P07-TIC-03044 (DIRICOM project, available in URL http://diricom.lcc.uma.es)

REFERENCES


