Variable Neighborhood Search as Genetic Algorithm Operator for DNA Fragment Assembling Problem

Gabriela Minetti
Laboratorio de Investigación en Sistemas Inteligentes
Universidad Nacional de La Pampa
Argentina
minettig@ing.unlpam.edu.ar

Gabriel Luque - Enrique Alba
Dpto. de Lenguajes y Ciencias de La Computación
Universidad de Málaga
Spain
(gabriel,eat)@lcc.uma.es

Abstract

Many specific algorithms and metaheuristics have been proposed for solving the DNA fragment assembly problem, but new algorithms with more capacity for solving this problem are necessary. The fragment assembly problem consists in building the DNA sequence from several hundreds (or even, thousands) of fragments obtained by biologists in the laboratory. This is an important task in any genome project since the rest of the phases depend on the accuracy of the results of this stage. In order to achieve this objective we propose a hybrid algorithm that achieves very accurate results in comparison with other metaheuristics.

1. Introduction

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). These methods are better than many algorithms which have been specially developed for this problem, but more efficacy and efficiency are necessary since smaller DNA sequences are more than 3,000 bases long. To achieve these properties, efficacy and efficiency, we propose a hybrid GA which has given very good results in this kind of problem [6, 8, 10], using a VNS descendent method, which has resolved this problem in a very efficient way [2], as an additional genetic operator.

Genetic Algorithm is a very popular evolutionary algorithm. GAs are characterized by using a population of multiple encoding tentative solutions to search in different regions of problem space at the same time. Variable Neighborhood Search is an explorative local search method. This search method swaps between neighborhood structures from a given set. We show a behavior analysis of GA with VNS as an additional genetic operator taking into account the quality of the results and the consumed CPU time. Finally, we compare our approach with well-known assemblers in the literature.

The rest of this article is organized as follows. The next section introduces the DNA fragment assembly problem. Section 3 explains how the GA is hybridized with VNS to solve this problem. Section 4 shows the experiments performed and discusses the results. Finally, the last section concludes and provides hints on further research.

2. The DNA Fragment Assembly Problem

We consider the shotgun sequencing method which consists of: (i) several copies of the DNA are produced and each copy is broken into millions of random fragments, (ii) those fragments are read by a DNA sequencing machine, and (iii) an assembler pieces together the many overlapping reads and reconstructs the original sequence [11].

The assembling of DNA fragments is divided into three different phases: the overlap phase (finding the overlapping among fragments), the layout phase (finding the order of fragments based on computed similarity scores), and the consensus phase (deriving the DNA sequence from the layout). In the assembly process, 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 fi-
nal 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:

$$\text{Coverage} = \frac{\sum_{i=0}^{n} \text{length of the fragment } i}{\text{target sequence length}}$$

where \( n \) is the number of fragments.

Specifically, the assembly of DNA fragments into a consensus sequence corresponding to the parent sequence constitutes the “fragment assembly problem”. It is a permutation problem.

3. A Hybrid Genetic Algorithm for solving the DNA FAP

Genetic Algorithms (GAs) [5], a special class of Evolutionary Algorithms (EAs), are computer-based solving systems, which use evolutionary computational models as a key element in their design. They have a conceptual base of simulating the evolution of individual structures via the Darwinian natural selection process. As it is shown in Algorithm 1, a GA maintains a population of multiple tentative solutions (individuals) which evolves throughout generations by reproduction of the fittest ones. 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).

**Algorithm 1 Genetic Algorithm**

1. \( t = 0 \); \{ it is the generation number \}
2. initialize \( P(t) \); \{it is the population at generation \( t \) \}
3. evaluate individuals in \( P(t) \);
4. while not condition do
5. \( t = t + 1 \);
6. select \( C(t) \) from \( P(t - 1) \);
7. apply variation operators (recombine and/or mutate) to individuals in \( C(t) \);
8. build \( C'(t) \);
9. evaluate individuals in \( C'(t) \);
10. replace some individuals in \( P(t - 1) \) with \( C'(t) \) to build \( P(t) \);
11. end while

We use the permutation representation 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. If all fragments are present in the chromosome and they are not duplicate in the chromosome, such permutation is a feasible solution. The permutation representation requires special operators to make sure that we always obtain legal (feasible) solutions. In order to maintain a feasible solution, two conditions must be satisfied: all fragments must be present in the ordering, and no duplicate fragments are allowed in the ordering.

In a previous work [?], we have realized that if a GA uses a seeding strategy to generate individuals in the initial population, it can help exploit promising regions of the search space. In consequence we use a constructive heuristic to generate an initial solution 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. Particularly we perform the following steps:

1. It starts generating a partial solution with a single randomly selected fragment.
2. The next fragment, which will be inserted, must satisfy the following conditions: (i) it must not have been inserted yet, (ii) it must form a contig with the last fragment inserted and, (iii) it will have the highest overlap score.
3. If we can not insert any fragment according to the previously mentioned conditions, we randomly select one from the remaining fragments.
4. Steps 2 or 3 are repeated until all fragments are inserted into the chromosome.

We work with a specially designed recombination operator for permutation representation to avoid infeasible solutions. The operator considered in this study is Order crossover (OX) [3]. We chose the OX because we have obtained high quality results in earlier job [?]. The order-based crossover operator first copies the fragment IDs between two random positions of the first parent into the offspring’s corresponding positions. Then it copies the rest of the fragments from the second parent into the offspring in the relative order presented in the second parent. If the fragment ID is already present in the offspring, then it is skipped.

In this work, we use the swap mutation operator. This operator randomly selects two positions from a permutation and then swaps the two resulting fragments.

The fitness function, \( F(l) \), 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.
We use two different selection methods. The binary tournament selection is used to choose the individuals to be modified by the variation operators. This technique randomly selects two individuals and finally chooses the best one. However, for building the population for the next generation we select the best \( \mu \) individuals from the \( \mu \) parents and \( \lambda \) offsprings.

### 3.1 VNS algorithm as a new operator in the GA

The Variable Neighborhood Search (VNS) is a meta-heuristic presented by Mladenovic and Hansen [9]. VNS solves optimization problems by doing systematic changes of neighborhood within a local search. VNS is a descendant method which does not follow a single trajectory since explores different predefined neighborhoods of the current solution using a local search (LS). The current solution is changed by a new one if and only if an improvement have been made. The basic idea is to change the neighborhood structure when the local search is trapped on a local optimum. A neighborhood structure in a solution space \( S \) is a mapping \( N : S \rightarrow 2^S, x \rightarrow N(x) \), where \( N(x) \) constitutes the neighborhood of \( x \). The steps of a basic VNS are shown in the Algorithm 2.

#### Algorithm 2 Basic VNS Algorithm

Initialization:
Select the set of neighborhood structures \( N_k, k = 1, \ldots, k_{\max} \);
Find an initial solution \( x \);
Choose an end condition;
while end condition is not met do
  \( k = 1 \);
  \( k := k_{\max} \) do
    Shaking:
    Generate randomly \( x' \in N_k(x) \);
    Local Search:
    Obtain the local optimum \( x'' \) by applying some local search to \( x' \);
    Move or not:
    if \( x'' \) is better than \( x \) then
      \( x = x'' \);
      \( k = 1 \);
    else
      \( k = k + 1 \);
  end if
end while

 Particularly, we use an operator based on VNS version according to the permutation representation and the fragment assembly problem. Thus, VNS is probabilistically applied by the GA in the same way that classical operators (recombination and mutation ones). In a previous work [8], we have implemented the VNS version used here. In this version, the neighbors in each neighborhood are generated by swapping between two positions from an initial solution. Meanwhile the number of neighborhoods, \( k_{\max} \), varies according with the instance. This value is proportional to number of fragments from each instance (see Eq. 3). This feature allows to adapt the computational effort and the efficacy of our VNS to the problem complexity, intensifying the search when this complexity is increased.

\[
k_{\max} = (\text{numberOfFragments}/n) + 1
\]

where \( n \) is an integer between 1 and 10.

The shaking process consists in generating a solution from \( N_t \) two positions, of an initial solution copy, are swapped. The first one represents the neighborhood structure and the second one is randomly selected. We use a non consumed-time and simple process (swap) to generate a solution since we only need a solution belonging to a particular neighborhood, which can be modified and improved by the 2-opt heuristic as a local search.

Finally we have established two issues in the VNS algorithm where two different stop criterions are necessary. The first issue is related to the quantity of times the whole algorithm iterates (first while sentence in Algorithm 2); and the second one is related to the stop criterion used in the second while sentence in Algorithm 2. We have experimented with two different settings of the first stop condition: one experiment has been executed during a single complete iteration and the another has been carried out using Eq. 5 to calculate the iteration total number. For the next stop criterion in the second while sentence, we need to limit the number of times that the neighborhoods are explored given that the combination between the neighborhood number (proportional to permutation size) and some instance sizes can produce an extremely long loop. For that, we have established a maximal iteration number which is calculated in Eq. 4.

\[
\text{iter}_{\text{stop}} = \left( k_{\max}/t \right) + 1
\]

\[
\text{iter}_{\text{max}} = \left( (\text{iter}_{\text{stop}}/t) + 1 \right)
\]

where \( k_{\max} \) is the neighborhood number and \( t \) is an integer between 1 and 10.

### 4 Experimental Results

In this section we analyze the behavior of our proposed methods. We have chosen two sequences from the NCBI web site\(^1\): a human MHC class II region DNA with fibronectin type II repeats HUMMHCFIB, with accession number X60189, which is 3.835 bases long and a human apolipoprotein HUMAPOBF, with accession number M15421, which is 10.089 bases long. We used GenFrag [4] to generate the different data sets shown in Table 1. 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. For each algorithm, we have performed 30 independent runs per instance. We have used a Pentium IV at 2.4 GHz and 1

\(^1\)http://www.ncbi.nlm.nih.gov/
GB RAM. The operating system used is SuSE Linux with 2.4.19-4GB kernel version.

We have established different configurations for GA and VNS operator. In Table 2, configurations for GA are shown: all the three GAs use the same setting but with different stopping criteria. For VNS operator we use a 33% of chromosome size as $k_{max}$ and the $t$ value is 10. We also analyze two different configurations of this operator: the first one, VNS$_1$, only performs one iteration, and the second one, VNS$_2$, iterates as maximum $\text{iter}_{max}$ times. From these GA and VNS configurations arise the following hybrid algorithmic proposals: $GA_1 + VNS_1$, $GA_2 + VNS_1$, $GA_3 + VNS_1$, $GA_1 + VNS_2$ and $GA_3 + VNS_2$. Note all these parametric values have been “optimized” through a previous hand-tuning comparing other different values.

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

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Instance</th>
<th>X60189</th>
<th>M15421</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coverage</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Fragment Length</td>
<td>395</td>
<td>386</td>
<td>423</td>
</tr>
<tr>
<td>Number of Fragment</td>
<td>39</td>
<td>48</td>
<td>66</td>
</tr>
</tbody>
</table>

### Table 2. GA parametric configurations

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Configurations</th>
<th>$\text{GA}_1$</th>
<th>$\text{GA}_2$</th>
<th>$\text{GA}_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stop Criterion</td>
<td>1000 generations</td>
<td>2000</td>
<td>5000</td>
<td></td>
</tr>
<tr>
<td>$\mu$</td>
<td>$512$ (Population size)</td>
<td>$512$ (Offsprings size)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recombination op. and Probability</td>
<td>$0.7$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutation op. and Probability</td>
<td>Swap mutation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Third op. and Probability</td>
<td>Variable Neighborhood Search</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent Selection</td>
<td>Binary tournament</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replacement</td>
<td>The best $\mu$ solutions from $(\mu + \lambda)$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

First of all, we compare the GA performance without hybridization and the VNS performance as a metaheuristic. From Table 3, we can see the GA generally obtains better numerical performance than VNS but the time-consumed by VNS is significatively smaller. For this reason we think it is quite interesting to apply VNS as a operator of GA. Our goal is to use VNS to escape from local optimum and thus it improves the results of the GA.

In a second place, we statistically analysis the behavior of the five different configurations from three point of view: best found solution, time to find the best solution, and the total time consumed by the algorithm. For that, we have made analysis of variance for non-parametric classification. Since the assumption on the normality and the homogeneity of the variance was impossible to verify, we use the Kruskal-Wallis test. The significance level ($\alpha$) is equal to 0.05 for all tests performed and the probability value ($p$-value) of statistical test is the probability of wrongly rejecting the null hypothesis if it is in fact true. The $p$-value is compared with the significance level and, if it is smaller, the result indicates that compared methods are significatively different. The obtained results are:

- Considering the best found individual we deduce that for all instances, the behavior produced by each hybrid algorithm setting does not differ significatively since their respective $p$-value is greater than $\alpha$. This similarity can be observed in Table 4.

- Taking into account the consumed time to find the best solution we can infer that for all instances with accession number X60189 the different settings have a similar behavior since they take in average less than 600 seconds and their $p$-value is greater than $\alpha$. While for all instances with accession number M15421 there are two well marked groups: the first one iterates 1000 generations as maximum ($GA_1 + VNS_1$ and $GA_1 + VNS_2$) and the second one performs until 5000 generations ($GA_2 + VNS_1$, $GA_3 + VNS_1$ and $GA_3 + VNS_2$). That is shown in Table 5, the first group takes less than 31 seconds to find the best individual; while the second group takes more than 200 seconds.

- Now, we analyze the total time consumed by each hybrid method. Again, we can observe two different groups. The first group is formed by $GA_1 + VNS_1$ and $GA_1 + VNS_2$, and the other group is constituted by the remaining approaches. The reason is that the GAs of first group iterates until 1000 generations (less than 40 seconds for X60189 instances and 865 seconds for M15421 instances); while $GA_2 + VNS_1$, $GA_3 + VNS_1$ and $GA_3 + VNS_2$ performs between 2000 and 5000 generations (between 15 and 195 seconds for X60189 instances and between 627 and 4100 seconds for the rest ones). Note that the difference between these two groups is given by the number of generations but not by the iteration number of VNS operator (see Table 6). That means the time consumed by each iteration of VNS operator is not significative with respect to total execution time of the GA.

Now, we are going to study the population evolution during the execution of each algorithmic proposal. For that we will use a representative instance, M15421_7. Looking at Figure 1, we can observe that all the variants have a similar behavior: before the first 1000 generations, the fitness grows quickly, and then the improvement of the solutions is very slow. In fact, that improvement is not statistically significant as we said above and it can be seen in Table 4.
In general we can infer several conclusions. The VNS operator is necessary to avoid a premature convergence and if we consider its different iteration number, we can not find differences in the results. For that reason, the settings with a single VNS iteration seems to be the most appropriate since we obtain better results when this number is grown but the consumed-time is increased too.

Finally, we compare our hybrid algorithm with assemblers found in the literature: Parson’s GA [10], a pattern matching algorithm (PMA) [7], a heuristic technique (PALS) [1], and commercially available packages: CAP3 or Phrap. We compare them in terms of the final number of contigs assembled. Figure 2 gives a summary of the results.

From Figure 2, we can observe that the use of a Genetic Algorithm, which inits its population with an intelligent seeding and uses a metaheuristic as a third genetic operator, obtains better results than a standard GA. Also, those results are competitive with respect to other approaches presented in the literature. In fact, this hybrid algorithm obtains a single contig (optimum) for 6 out of 7 FAP instances. In the instance where they do not find the optimal solution, their results are similar to very well-known commercial packages (CAP3 or Phrap).

5. Conclusions

In this work, we present a Genetic Algorithm which has been hybridized in two different points: the first one is related to a seeding strategy to generate the initial population and the second is related to a descendent method used as an additional genetic operator. We have proposed a greedy method as a seeding strategy, which has been specially designed for this problem. Furthermore, we use a metaheuris-


