Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

BACHELOR THESIS

Fragment Assembly Problem by Means of Evolutionary Computation

Prague, 2009

Libor Wagner

Supervisor: Ing. Jiří Kubalík, Ph.D.
Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne 3. 7. 2009

[Podpis]

Wagner
Acknowledgement

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. Especially I am deeply indebted to my supervisor Ing. Jiří Kubalík, Ph.D. whose help, suggestions and encouragement helped me in all the time of research for and writing of this thesis.

I want to thank Gabriel Luque for providing us with the instances of DNA Fragment Assembly Problem.

I would also like to thank my family and friends. Especially I would like to than my parents Richard and Lenka and also Šárka whose encouragements helped me to write this thesis.
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Libor Wagner

Study programme: Software Engineering and Management

Specialisation: Intelligent Systems

Title of Bachelor Project: Fragment Assembly Problem by Means of Evolutionary Algorithms

Guidelines:
1. Get acquainted with a problem known as DNA fragment assembly problem, study current approaches for solving this problem and identify issues related to these approaches.
2. Get acquainted with evolutionary algorithms (EAs).
3. Propose and implement an EA for solving the DNA fragment assembly problem.
4. Carry out experiments to assess a performance of the proposed algorithm, statistically evaluate the achieved results and compare the algorithm with other approaches if available.

Bibliography/Sources: Will be provided by the supervisor.

Bachelor Project Supervisor: Ing. Jiří Kubalík, Ph.D.

Valid until: the end of the winter semester of academic year 2009/2010

Prague, December 10, 2008

prof. Ing. Vladimír Mafík, CSc.
Head of Department

doc. Ing. Boris Šimák, CSc.
Head
ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Libor Wagner
Studijní program: Softwarové technologie a management
Obor: Inteligentní systémy
Název tématu: Optimální skládání fragmentů DNA pomocí evolučních algoritmů

Pokyny pro vypracování:
1. Seznamte se s problémem z oblasti bioinformatiky známým jako DNA fragment assembly problem a se standardními postupy používanými pro jeho řešení.
2. Prostudujte metody evolučních algoritmů a jejich možné využití pro tento problém.
3. Navrhněte a naimplementujte evoluční algoritmus pro řešení tohoto problému.
4. Experimentálně ověřte funkčnost navrženého algoritmu, výsledky výhodnotte a pokud možno porovnejte s výsledky dosaženými pomocí jiných známých metod.

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí bakalářské práce: Ing. Jiří Kubalík, Ph.D.

Platnost zadání: do konce zimního semestru 2009/2010

V Praze dne 10.12.2008

prof. Ing. Vladimír Mafík, DrSc.
vedoucí katedry
doc. Ing. Boris Šimák, CSc.
dekan
Abstract

In this thesis the Iterative Prototype Optimization with Evolved Improvement Step algorithm is applied on problem from bioinformatic, which is called DNA Fragment Assembly. This problem is situated on final stage of DNA reading, especially long strands of DNA. Long strands cut many times into large set of fragments. Fragments are then assembled by computer into string corresponding with source DNA. This process is very complicate because the order and direction of fragments is lost in early stage.

The algorithm was implemented and adjusted for needs of DNA Fragment Assembly Problem. Its performance were tested on standard benchmarks and compared to other assembly projects. The result are promising, but efficiency must be improved.

Keywords: DNA, Fragment Assembly, Evolutionary Algorithms
Abstrakt

V této práci je postup Iterativní Optimalizace Prototypů s Evolučním Zlepšovacím Stupněm aplikován na problém z bioinformatiky, zvaný skládání fragmentů DNA. Tento problém je konečnou fází čtení dlouhých řetězců DNA. Dlouhé řetězce DNA jsou několikrát přečteny a tím je vytvořena velká skupina fragmentů. Fragmenty jsou poté složeny počítačem do řetězce který odpovídá čtené DNA. Tento proces je velmi komplikovaný protože jak pořadí tak i orientace se ztratí v předchozím stupni.

Algoritmus byl implementován a upraven pro daný problém. Jeho možnosti byly otestovány na standardních instancích daného problému a porovnány s ostatními projekty. Výsledky ukázali, že tento postup je slibný ale efektivita naší implementace musí být zlepšena.

Klíčová slova: DNA, Skládání Fragmentů, Evoluční Algoritmy
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xi</td>
</tr>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1. DNA Fragment Assembly Problem</td>
<td>3</td>
</tr>
<tr>
<td>1.1 DNA</td>
<td>3</td>
</tr>
<tr>
<td>1.2 The problem</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Repeats and errors</td>
<td>5</td>
</tr>
<tr>
<td>2. Algorithms for solving FAP</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Greedy algorithm</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Ant Colony System</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Sequencing by Hybridization</td>
<td>8</td>
</tr>
<tr>
<td>2.3.1 Error correction</td>
<td>8</td>
</tr>
<tr>
<td>2.4 Local Search</td>
<td>8</td>
</tr>
<tr>
<td>3. POEMS</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Genetic Algorithm</td>
<td>10</td>
</tr>
<tr>
<td>3.1.1 Solution representation</td>
<td>11</td>
</tr>
<tr>
<td>3.1.2 Fitness Function</td>
<td>11</td>
</tr>
<tr>
<td>3.1.3 Selection</td>
<td>11</td>
</tr>
<tr>
<td>3.1.4 Operators, crossover and mutation</td>
<td>12</td>
</tr>
<tr>
<td>3.2 Local Search</td>
<td>12</td>
</tr>
<tr>
<td>3.3 POEMS Implementation</td>
<td>13</td>
</tr>
<tr>
<td>3.3.1 Individuals and Population</td>
<td>13</td>
</tr>
<tr>
<td>3.3.2 Selection</td>
<td>14</td>
</tr>
<tr>
<td>3.3.3 Crossover operator</td>
<td>14</td>
</tr>
<tr>
<td>3.3.4 Mutation</td>
<td>15</td>
</tr>
<tr>
<td>3.3.5 Insertion</td>
<td>15</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3&quot; and 5&quot; end of DNA</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Chaining cloning scheme</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Example of overlapping fragments</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Example of wrongly placed fragment</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>Laboratory errors</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>Example of chimera</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>Crossover example</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>Overlap problem</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>Align example</td>
<td>18</td>
</tr>
<tr>
<td>10</td>
<td>Fitness function experiment progress (overlap)</td>
<td>22</td>
</tr>
<tr>
<td>11</td>
<td>Time consumption</td>
<td>25</td>
</tr>
<tr>
<td>B.12</td>
<td>Progress summary of all experiments (contigs)</td>
<td>vi</td>
</tr>
<tr>
<td>B.13</td>
<td>Progress summary of all experiments (overlap)</td>
<td>vii</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Problem instances</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>Algorithm setup</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>Fitness function experiment results</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>Actions experiment results ($f_1$)</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>Actions experiment results ($f_2$)</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>Random experiment results</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>Comparison with PALS</td>
<td>24</td>
</tr>
<tr>
<td>B.8</td>
<td>Experimental results (fitness)</td>
<td>iii</td>
</tr>
<tr>
<td>B.9</td>
<td>Experimental results (overlap)</td>
<td>iv</td>
</tr>
<tr>
<td>B.10</td>
<td>Experimental results (contigs)</td>
<td>v</td>
</tr>
</tbody>
</table>
List of Algorithms

1. Greedy ................................................................. 7
2. PALS ................................................................. 9
3. Genetic algorithm .................................................. 11
4. Local Search .......................................................... 13
5. POEMS ............................................................... 14
6. Fragment comparison ............................................... 19
Introduction

During the recent 50 years a new research area has been developing, with the aim to find particular function of the information coded in the deoxyribonucleic acid (DNA) which is present in almost every known living organism. The leading project in this area is Human Genome Project\(^1\). Our work is aimed on a problem from computational biology called DNA Fragment Assembly, which is crucial for recognition of DNA function as it is in the early stage of this research and the further research depends on its precision.

Human genome consists of 23 chromosome pairs and it is estimated that it contains up to 3000 genes, coded by just over 3 billions base-pairs\(^1\). The standard method, used for reading DNA by machine, is not accurate on DNA strands longer than 1000 base-pairs, therefore to read complete human genome different methods are used. Long strands of DNA needs to be broken into smaller pieces, called fragments, short enough to be read routinely by standard methods.

The method used to break DNA sequence into fragments is called Shotgun Sequencing. This process is divided into two parts, replication (cloning) when given DNA is cloned many times and these clones are then cut into fragments short enough to be read automatically by machine using standard Chain Termination, method used to read DNA strands. This process does not retain neither order of fragments nor their orientation.

Searching for the order and orientation of fragments in which they come in source DNA is made by computer. This problem is NP-hard and is called DNA Fragment Assembly\(^2\).

Many assembly projects were trying to solve this problem during the past 50 years. There are many different approaches, the most common approach is greedy-based algorithm\(^3\) used in the most popular packages such as a CAP\(^4\), PHARP\(^5\) and Tigr Assembler\(^6\), other projects are using Ant Colony Systems\(^4, 5\), sequencing by hybridization\(^6\), all variety of evolutionary algorithms\(^7, 8, 9, 10\) and local search\(^11\).

In this work an application of Iterative Prototype Optimisation with Evolved Improvement Steps\(^12\) is proposed for solving DNA Fragment Assembly Problem. This approach is combination of local search and genetic algorithm. But has many advantages to the classic genetic algorithm as it is well suited for permutation optimization problems and it is useful for solving large instances of the problem.

---

\(^1\) [www.ornl.gov/sci/techresources/Human_Genome/home.shtml](http://www.ornl.gov/sci/techresources/Human_Genome/home.shtml)
\(^2\) [pbil.univ-lyon1.fr](http://pbil.univ-lyon1.fr)
\(^3\) [www.codoncode.com](http://www.codoncode.com)
\(^4\) [www.jcvi.org](http://www.jcvi.org)
Iterative Prototype Optimisation with Evolved Improvement Steps (POEMS) was implemented, tested on standard benchmarks and compared to other DNA Fragment Assembly projects.

Result shows that our approach is promising and comparable to other DNA Fragment Assembly projects, but more work is needed to improve its efficiency.

The rest of this work is structured as follows. In Section 1 background information is given and DNA Fragment Assembly Problem is described. Section 3 contain description of POEMS algorithm. The Implementation of POEMS algorithm for DNA Fragment Assembly Problem is described in Section 4. Experiments description and their results are given in Section 5. Our conclusion is given in last Section.
1 DNA Fragment Assembly Problem

DNA Fragment Assembly Problem (FAP) is final part of the DNA reading. The DNA reading is part of Human Genome Research, which takes as aim to find information coded in DNA. But ability of reading long strands of DNA is useful in many different research and commercial projects.

1.1 DNA

The deoxyribonucleic acid (DNA) is a double stranded helix which contains genetic information and is present in an almost every known living organism. Each strand is constructed from four types of building blocks called nucleotides. These four nucleotide bases are called Adenine, Cytosine, Guanine and Thymine. Every nucleotide base has 3” hydroxyl group on one side and 5” phosphate group on the opposite side. 3” hydroxyl group and 5” phosphate group stuck together and build the DNA strand. On the both ends of the DNA strand are nucleotide bases which are connected just on the one side and the other side is free, therefore we can say that the DNA strand has 3” (three prime) end and 5” (five prime) end. Every nucleotide base from one strand is connected to an opposite nucleotide from another strand using hydrogen bounds, this couple is called base pair. Bases from opposite strands are not connected by random, but Adenine always connects with Thymine, Cytosine with Guanine and vice versa, hence the second strand is complementary. Complementary strand is reversed, when one strand has its three prime end on left and five prime end on right, the complementary strand has five prime end on left and three prime end on right see Figure 1. Strands are read separately and both are read from three prime to five prime.

![Figure 1: 3” and 5” end of DNA.](image-url)
1.2 The problem

*Chain Termination*, the standard laboratory method used for DNA sequencing (reading) is not accurate on strands longer than 1000 base-pairs. Therefore in order to read longer strands, more complicated approaches are used. The most common method is *Chaingun Sequencing* also called *Chaingun Cloning*, longer DNA strands are firstly cloned many times and than all the cloned DNA strands are cut at random positions into pieces short enough to be read automatically using standard *Chain Termination* see Figure 2.

![Chaingun cloning scheme](image)

Figure 2: Chaingun cloning scheme.

This approach produces large set of strings over the alphabet \{A, C, T, G\} with a different length, called fragments. Every letter represents one nucleotide base, A for Adenine, C for Cytosine, T for Thymine and G for Guanine. This set can be characterized by using two main values: the number of fragments in this set and the coverage which is the minimal number in how many fragments is every particular nucleotide base delegated.

During *Chaingun Cloning* the layout of fragments is lost, neither the order in which fragments come from source DNA strand nor the orientation of fragments is preserved. By orientation we mean the strand from which the fragment comes, from forward or reverse complement. All fragments are red in one way, therefore the fragments from reverse complement strands must be reversed and nucleotides converted to their opposites to fit
into forward strand.

Searching for the layout of fragments is a combinatorial optimization task and is done by a computer. Let \( n \) be the number of fragments, then \( 2^n \times n! \) is number of layout combination, where \( 2^n \) represents two possible directions of fragments and \( n! \) represents all the permutation of all fragments, hence to find optimal layout is NP-Hard problem called DNA Fragment Assembly.

The way how to decide whether two fragments come from the same region of a source DNA strand is to measure their overlap score, in other words to find correlation between these two fragments see Figure 3. To find the best overlap score is itself complicated task because of repeats and errors mentioned in Section 1.3.

![Figure 3: Example of overlapping fragments.](image)

Assembled DNA string is called consensus. Its quality can be measured by number of contigs, which is number of detached parts of this DNA string, that parts which are not connected by overlapping fragments.

1.3 Repeats and errors

Due to the small alphabet of nucleotide bases, it is natural that one pattern of nucleotide bases can occur at more than one place of the DNA strand. This fact can mislead the assembly algorithm to place fragment on wrong position see Figure 4, which can lead to false information. Repeats are major bottleneck for all assembly algorithms and no one can solve them absolutely.

![Figure 4: Example of wrongly placed fragment.](image)

In addition to repeats, laboratory work which precedes computer assembly is not absolutely accurate and produces number of errors which must be taken in account during
computation. We recognize three types of errors: substitution, insertion and deletion. When nucleotide base is different from that in the same place in the corresponding source DNA, we call it substitution. When a fragment contains an extra nucleotide base when comparing to the source DNA, then we speak about insertion, sometimes called addition. Deletion is when a nucleotide base is not present in the fragment but is present in the source DNA.

\[ S: \text{accgtagcactgga} \quad S: \text{accgtag-actgga} \quad S: \text{accgtagcactgga} \]
\[ F: \text{cgtag\text{aact}} \quad F: \text{cgtag\text{cact}} \quad F: \text{cgtag-act} \]

(a) Substitution \hspace{1cm} (b) Insertion \hspace{1cm} (c) Deletion

Figure 5: Laboratory errors.

Chimeras are pieces of DNA from different parts of DNA, which connect and are sequenced together and produce false fragment see Figure [6]

\[ S: \text{accgtagactgga} \]
\[ F: \text{cgtagactcgta} \]

Figure 6: Example of chimera.
2  Algorithms for solving FAP

In the recent 30 years many projects has been dealing with DNA Fragment Assembly Problem, but none of them can solve all its obstacles. Many different approaches were invented.

Commonly used approach is divided into three stages: overlap, layout and consensus. In overlap stage all overlapping fragments are found, by comparing all fragments and their reverse complements. Layout stage is the hardest, it consists of finding the order and orientation of all fragments, according to the overlap score. Last stage is to reconstruct layout into DNA sequence called consensus.

2.1  Greedy algorithm

Simplest assembly approach is greedy based algorithm for Shortest Common Superstring problem. In iterations two fragments with best overlap score are selected and replaced by their shortest common superstring. Basic greedy algorithm is shown in Algorithm 1.

Algorithm 1  Greedy algorithm

\textbf{Input: } $S$ fragment set.

\textbf{Output: } $S$ Shortest common superstring.

\begin{algorithmic}
\While{$\text{Size}(S) > 1$}
\State $A, B \leftarrow \text{Select}(S)$ \{Select two strings with best overlap\}
\State $C \leftarrow \text{Combine}(A, B)$ \{Combine two selected strings\}
\State $S \leftarrow \text{Replace}(C)$ \{Replace old strings with new one\}
\EndWhile
\end{algorithmic}

2.2  Ant Colony System

The Ant Colony System is search algorithm with roots in biology studies of insect behavior. Ants can find shortest path between nest and food source without any visual clues. Ants done that by following pheromone track deployed by other ants.

This approach is designed for combinatorial problems especially for Traveling Salesman Problem. As Fragment Assembly Problem is closely related to Traveling Salesman Problem we can apply Ant Colony System on Fragment Assembly Problem and in [4] this method is presented.
Comparing Traveling Salesman Problem and Fragment Assembly Problem, we can use overlap score as reverse analogy to distance of two cities. However, the path from $a$ to $b$ is the same length as path from $b$ to $a$ when speaking about Traveling Salesman Problem but considering fragments the overlap score is dependent on direction and also reverse complement fragments need to be taken into account. Therefore the Smith-Waterman algorithm \cite{13} is used to calculate the overlap score.

2.3 Sequencing by Hybridization

Sequencing by Hybridization which leads to Eulerian Superpath Problem as presented in \cite{6} don not follow classic ”overlap-layout-consensus” approach.

Fragment set $F = \{f_1, \ldots, f_n\}$ is converted to $l$-tuples, when every fragment $f$ donates $m - l + 1$ $l$-tuples where $m$ is length of the fragment. Using fragment set $F = \{f_1, \ldots, f_n\}$ which define de Bruijn graph $G(F_l)$ with vertex set $F_{l-1}$ (the set of all $(l - 1)$-tuples), $(l - 1)$-tuple $v \in F_{l-1}$ is joined by edge with an $(l - 1)$-tuple $w \in F_{l-1}$, if $F_l$ contain an $l$-tuple for which the first $l - 1$ nucleotides coincide with $v$ and the last $l - 1$ nucleotides coincide with $w$. Every $l$-tuple from $F_l$ corresponds to edge in $G$. If we have only one fragment, than this sequence corresponds to a path which visits every edge of de Bjuijn graph, Chinese Postman Problem. Once we introduce multiplicity of edges in de Bjuijn graph, we can transform Chinese Postman Problem to Eulerian Path Problem which is easy to solve.

2.3.1 Error correction

Error correction as introduced in \cite{6} can eliminate up to 98% errors which can significantly reduce the complexity of the problem.

Method used is derived from Sequencing by Hybridization. Given set of $l$-tuples $S = \{s_1, \ldots, s_m\}$, derived from fragments set $F = \{f_1, \ldots, f_n\}$, error in fragment $f$ is distributed into $2l$ $l$-tuples, $l$ into forward $l$-tuples and $l$ into their reverse complement. Than we are looking for change in fragment $f$ that reduce set $S$ by exactly $2l$. However, this approach in special cases can introduce error rather than eliminate.

2.4 Local Search

Local Search algorithm described in \cite{11} called Problem Aware Local Search (PALS). PALS uses one solution which is then optimized, using moves in order. Analogy to
fitness function of Genetic Algorithm, is used to decide which move to make. This function called $\Delta$ is computed only from actually changed fragments and it takes into account also number of contigs.

We chose Problem Aware Local Search algorithm to compare with our Iterative Prototype Optimisation with Evolved Improvement Step implementation, as both approaches are closely related.

**Algorithm 2** Problem Aware Local Search.

1: $s \leftarrow \text{GenerateInitialSolution}$
2: repeat
3:   $L \leftarrow \emptyset$
4:   for $i = 0$ to $N$ do
5:     for $j = 0$ to $N$ do
6:       $\Delta_c, \Delta_f \leftarrow \text{CalculateDelta}(s, i, j)$
7:       if $\Delta_c \geq 0$ then
8:         $L \leftarrow L \cup \langle i, j, \Delta_f, \Delta_c \rangle$
9:     end if
10:   end for
11: end for
12: if $F \neq \emptyset$ then
13:   $\langle i, j, \Delta_f, \Delta_c \rangle \leftarrow \text{Select}(L)$
14:   $s \leftarrow \text{ApplyMovement}(s, i, j)$
15: end if
16: until no changes
3 POEMS

We chose Iterative Prototype Optimization with Evolved Improvement Step (POEMS) algorithm to test its performance on DNA Fragment Assembly Problem. POEMS algorithm is well suited for combinatorial optimization problems and very useful for large instance of problems. The POEMS mechanism is combining two standard approaches: Genetic Algorithm and Local Search.

3.1 Genetic Algorithm

Genetic Algorithms are class of Evolutionary Algorithms which are inspired by evolutionary biology. In the nature strong individuals has better chance to survive and reproduce than weaker ones. During the reproduction offspring are produces as mutated combination of their parents. Over many generations of this natural selection and reproduction, the mean quality of individuals in population is rising. Also completely new features are developed.

Genetic algorithms are search methods used to find ideal or approximate solution to optimization problems by simulating natural evolution. Population of abstract individuals, also called chromosomes or genotype, which represents candidate solution to the given problem, also called phenotype. The population is then evolved towards better individuals. Genetic Algorithm usually starts with randomly generated population of candidate solutions and works in generations. In each generation the every individual is measured by fitness function. At least two individuals are stochastically selected according to their value of fitness function. These selected individuals are than combined and randomly mutated to produce new population. The new population is then used in next generation. Sometimes the best individual from the old population is passed to the new one to prevent algorithm from loosing good solution.

Genetic algorithm can be described in five major steps: evaluation, selection, crossover, mutation and insertion. For every step many different methods are used. Basic Genetic algorithm is shown in Algorithm 3.

Different method from Generational approach mentioned above is to keep just one population and to replace worse individuals by the ones newly generated by crossover and mutation.
Algorithm 3 Simple genetic algorithm.

Input: $f$ fitness function describing given problem.

Output: $bsf$ best solution so far.

1: $P \leftarrow \text{GenerateInitialPopulation}()$
2: repeat
3: $P \leftarrow \text{Evaluate}(f,P)$
4: repeat
5: $I_a, I_b \leftarrow \text{Select}(P)$
6: $O_a, O_b \leftarrow \text{Crossover}(p_a, p_b)$
7: $O_a \leftarrow \text{Mutate}(ch_a)$
8: $O_b \leftarrow \text{Mutate}(ch_b)$
9: $P' \leftarrow \text{Insert}(ch_a, P')$
10: $P' \leftarrow \text{Insert}(ch_b, P')$
11: until $P'$ is full population
12: $P \leftarrow P'$
13: until Termination condition
14: return $\text{bestof}(P)$

3.1.1 Solution representation

Solution is usually represented in two ways genotype and phenotype. Genotype is used for crossover and mutation, whereas phenotype is used for fitness function evaluation. Genotype must be easily converted to phenotype and vice versa. Binary array is commonly used implementation of genotype which is easily converted to numbers as phenotype and back.

3.1.2 Fitness Function

Fitness function is a way how to measure quality of individuals, to decide which individual is better. Ideal fitness function is closely related to problem goal but also the computation time is a factor because fitness function is computed many times during the algorithm.

3.1.3 Selection

There are two commonly used selection strategies: roulette wheel and tournament selection. The first simulates a roulette wheel with different size of compartments. Better
individuals, according to the value of fitness function, have bigger compartments and worse individuals have the smaller ones. It ensures that the better individuals has higher chance to be selected and the worse individuals smaller.

Tournament selection is simpler, at least two individuals are selected randomly and the best from this selection is passed to crossover-mutation.

### 3.1.4 Operators, crossover and mutation

Operators must be defined according to solution representation because wrong combination of operators and representation can produce illegal solutions which must be then penalize by fitness function. Other approach is to implement operators which does not produce illegal solutions.

For binary string representation standard crossover operators are one or two point crossover and uniform crossover. One point crossover needs two parents, both are brake in one random point and one half is swap, that produce two offspring, every with a different part from each parent. Two point crossover is similar but uses two braking points instead of one. Uniform crossover is also bit operator, each bit of offspring has the same probability to come from one or other parent, second offspring is containing bits which are not in the first offspring.

Standard mutation operator for binary string representation is bit switch. Every bit of genotype has equal chance to be switched.

### 3.2 Local Search

Local search is also widely used for optimization problems. Basic idea is to move from one worse solution to better solution by exploration of their neighborhood. Usually an initial prototype solution is generated and by another algorithm its neighborhood is searched for better solution see Algorithm 4.

Simplest variant of this algorithm is random local search. Initial solution is randomly generated and then randomly changed. Then both candidate solution, initial and changed, are compared, according to evaluation of fitness function similar to Genetic Algorithms, and worse solution is discarded. Remaining candidate is used as initial for the next iteration.
Algorithm 4 Local search algorithm.

**Input:** $f$ fitness function describing given problem.

**Output:** $bsf$ best solution so far.

- $P \leftarrow \text{GenerateInitialPrototype}()$
- $P \leftarrow \text{Evaluate}(f, P)$

repeat

- $P’ \leftarrow \text{Explore}(P)$
- $P’ \leftarrow \text{Evaluate}(f’, P’)$

if $P’ \geq P$ then

- $P \leftarrow P’$

end if

until Termination condition

3.3 POEMS Implementation

Iterative Prototype Optimization with Evolved Improvement Step (POEMS) algorithm \cite{12} has every character of Local Search. Initial prototype, which represents solution to the given problem, is changed in iterations by number of actions which are called Action Sequence. Evolutionary approach is used to find the best possible Action Sequence. Using this approach the generation of Action Sequence can be separated from the problem, therefor the POEMS mechanism is implemented universally and can be used for different optimization problems.

Basic steps of POEMS algorithm are: generating initial prototype of candidate solution, and then in every iteration run Genetic Algorithm, produce new prototype by applying evolved Action Sequence. Worse prototype is then discarded and better is used in next iteration. These steps are repeated until termination condition is satisfied, usually number of runs, see Algorithm \ref{Algorithm5}.

POEMS mechanism uses Genetic Algorithm to evolve the best possible Action Sequence, this algorithm consists of all steps mentioned in Section \ref{Section3.1}.

3.3.1 Individuals and Population

Individual of POEMS Evolutionary Algorithm is sequence of actions, every action consists of an action id, this id is link between action from GA and action implemented for particular problem, and set of parameters which are passed to action from the problem.
Algorithm 5 Example of POEMS algorithm.

**Input:** $f$ fitness function.

**Output:** $S$ best solution so far.

1. $S \leftarrow \text{GenerateInitialSolution}()$
2. **repeat**
   1. $A \leftarrow \text{RunGA}(f)$
   2. $S' \leftarrow \text{ApplyActions}(S, A)$
   3. **if** $S' \geq S$ **then**
      1. $S \leftarrow S'$
   **end if**
3. **until** Termination condition
4. **return** $S$

Number of parameter is constant for particular problem, but not all of them has to be utilized, hence the number of parameters for actions implemented in the problem side can vary. There is special action with no operation, called NOP action, to assure variable number of actions in sequence.

Individuals in population are stored in shelves called Niche, number of Niches is equal to length of the individual. Niches are numbered form 1 to $n$ where $n$ is length of the individual and each niche can contain individuals with number of active action, actions that are not NOP, same or higher than its number. This improvement described in [14] prevents Evolutionary Algorithm from evolving chromosomes with small number of active actions or even without active actions. This happens when in a late stage of algorithm the prototype is so good that the most of the generated actions worsen the prototype.

### 3.3.2 Selection

For each parent the Tournament Selection is used. Firstly random Niche is selected, then two individuals are selected from that niche and according to their value of fitness function the best is selected for crossover or mutation.

### 3.3.3 Crossover operator

After selection which operator crossover or selection will be used is decided. Crossover operator takes all actions, puts them together and randomly selects that which will be
placed into offspring. This produce the offspring which is random combination of both parents see Figure 7. Actions are not changed in any way.

\[
\text{par1: } \text{NOP A3 A1 A2 NOP} \\
\text{off: } \text{A2 NOP A3 A3 A3} \\
\text{par2: } \text{A3 A2 A1 A3 A2}
\]

Figure 7: Crossover example.

### 3.3.4 Mutation

Every action from individual has probability to be mutated. Mutations is represented by two steps: the first changes the active action to NOP or NOP to random active action, in the second step new set of parameters is randomly generated.

### 3.3.5 Insertion

A new chromosome is inserted into niche according to its number of active actions and its value of fitness function. For example when we are inserting a chromosome with 3 active actions we select niche 1, 2 and 3 and find chromosome with worse value of fitness function from this selection and replace it with a chromosome which we are inserting.
4 Implementation of POEMS for FAP

POEMS algorithm is very universal, and we have to implement also problem side of this algorithm, such as solution representation, actions and fitness function.

4.1 Solution representation

Two different solution representations for DNA Fragment Assembly were presented in [5]: Sorted Order and classic Permutation representation. Sorted order uses array in which fragments are represented by indexes and values represents relative position. Let $S$ be the Sorted Order representation of fragment set $F = \{f_1, f_2, \ldots, f_n\}$ and when $S[i] < S[j]$ then fragments $i$ will be placed before fragment $j$. This representation is well suited for Genetic Algorithms, because when converted into binary array, standard operators such as one or two point crossover and uniform crossover can be used without producing illegal solutions. However this representation is harder to reconstruct to particular fragment layout. As we do not use Genetic Algorithm with prototype solution we have no use for this representation.

Classic Permutation representation uses array in which the fragments are represented as values and indexes represent absolute position of fragments. Let $P$ be the Permutation representation of the same fragment set as above, then $P[k] = i$ means that fragment $i$ will be placed on $k$-th position in layout. We use this representation, as it is easily reconstructed to the fragment layout and as long as we use specially implemented actions, the algorithm will produce only legal solutions.

We also need to represent the orientation of the fragment, therefore our solution consists of two arrays: one Permutation representation of fragment order and the second with true/false values, when value in $j$-th columns is set (equals true) then reverse complement of $j$-th fragment is used instead.

4.2 Actions

We suggest four actions: move, swap, switch and invert. All actions use up to two parameter $a$ and $b$. In the next four sections $P = \{p_1, p_2, \ldots, p_n\}$ is referring to Permutation representation of fragments order and $D = \{d_1, d_2, \ldots, d_n\}$ is referring to vector of orientations.
4.2.1 Move

The move action uses the both arguments $a$ and $b$, it takes every fragment between $a$ and $b$ including $a$ and $b$ and move them one position right if $a < b$ of left if $a > b$, last fragment is placed on the first position in this suborder. In other words, let $a < b$, then $p_{i+1} := p_i$ for $a < i < b$ and $p_a := p_b$ the same for $D$.

4.2.2 Swap

Swap action uses two arguments $a$ and $b$, it takes fragment on $a$-th position a place it on $b$-th position, also its orientation, and vice versa. In other words, $p' := p_b$, $p_b := p_a$ and $p_a := p'$ the same for $D$.

4.2.3 Switch

Switch action uses only one argument $a$, it switches orientation of fragment on $a$-th position. In other words if $d_a = \neg d_a$.

4.2.4 Invert

The invert action uses both arguments $a$ and $b$, it takes fragments between $a$ and $b$ including, and invert their order and orientation. In other words, let $a < b$, then $p'_{a+i} = p_{b-i}$ for $0 \leq i \leq (a - b)$ and $p_i = p'_i$ for $1 \leq i \leq n$ and the same for $D$. Then $d_i = \neg d_i$ for $a \leq i \leq b$.

4.3 Fitness Function

We test two closely related fitness functions. First fitness function mentioned in [8] is sum of overlaps between adjacent fragments. The classic fitness function $F$ is

$$F = \sum_{i=1}^{n-1} w_{(i,i+1)}$$

where $w_{(i,j)}$ is best overlap between fragments on $i$-th and $j$-th position. Genetic Algorithm is maximizing this function. As we need to measure quality of action sequences rather than the the solution, our fitness function $f1$ is computed as difference between value of $F$ fitness function before and after applying action sequence on prototype solution. Fitness function $f1$ is

$$f1 = f_{after} - f_{before}$$
where \( f_{\text{before}} \) is overlap sum before applying action sequence and \( f_{\text{after}} \) is overlap sum after applying action sequence.

Second fitness function we test is similar to one used in PALS project. Not only overlap but also number of contigs is taken in account. The \( f^2 \) fitness function is

\[
f^2 = C + \frac{\max - F}{\text{max}}
\]

where \( C \) is number of contigs, \( \text{max} \) is big number, in our case \( \text{max} \) is number of bases in all fragments of fragment set. Algorithm is minimizing this function.

Due to different lengths of fragments, a short fragment can sink into larger ones. When computing overlap between next fragment in layout and that which sinks into the fragment before, the overlap score is not accurate see Figure 8. Therefore fragments which sink in larger ones are just added to overlap sum, but next fragment is compared to fragments two before.

\[
\begin{align*}
\text{f1}: & \quad \text{accgtagactgga} \\
\text{f2}: & \quad \text{cgtagact} \\
\text{f3}: & \quad \text{gactcctg}
\end{align*}
\]

Figure 8: Overlap problem.

### 4.3.1 Overlap computation

To find best overlap between adjacent fragments, we test all shifts from left-left to right-left align see Figure 9.

In every shift all the overlapping nucleotides are compared. As we use artificially generated benchmarks which does not contain errors mentioned in Section 1.3 we can
discard all shifts where not all overlapping fragments are same as opposite ones. When all apposite nucleotides are equal the overlap score is number of nucleotides which participate in this overlap. Algorithm which compares fragments is shown in Algorithm 6. Highest overlap score of all shifts is final overlap score.

Algorithm 6 Algorithm comparing two fragments.

Input: $a$ and $b$ fragments with lengths $l_a$ and $l_b$, fragment $a$ is that which is moving, and (shift) $s$, number of nucleotides of which the fragment $b$ is moved from left-left align to right.

Output: 0 if fragments are not overlapping and overlap score if they are overlapping.

1: $i \leftarrow 0$
2: if $s < 0$ then
3: $f_a \leftarrow -s$
4: $f_b \leftarrow 0$
5: else
6: $f_a \leftarrow 0$
7: $f_b \leftarrow s$
8: end if
9: while $(f_a + i) < l_a$ and $(f_b + i) < l_b$ do
10: if $a[f_a + i] \neq b[f_b + i]$ then
11: return 0
12: end if
13: $i \leftarrow i + 1$
14: end while
15: return $i$
5 Experiments

Artificially generated benchmarks were made to test and analyze performance of proposed algorithm. These datasets were generated by GenFrag [15]. GenFrag takes complete DNA sequence and generates fragments of given attributes.

Data produced this way does not contain any errors mentioned in Section 1.3, but still contains repeats mentioned in the same Section.

5.1 Problem instances

For the experiments four problem instances were chosen from NCBI[1] database. Human MHC class III region DNA with fibronectin type-III repeats with accession number X60189, which is 3835 bp long. Human apolipoprotein B-100 mRNA, complete cds, with accession number M15421, which is 10089 bp long. Enterobacteria phage lambda, complete genome, with accession number J02459, which is 48502 bp long. Neurospora crassa DNA linkage group II BAC clone B10K17, with accession number BX842596, which is 77292 bp long. Parameters of generated fragment sets are shown in Table 1.

<table>
<thead>
<tr>
<th>Accession No.</th>
<th>X60189</th>
<th>M15421</th>
<th>J02459</th>
<th>BX842596</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length [bp]</td>
<td>3835</td>
<td>10089</td>
<td>48502</td>
<td>77292</td>
</tr>
<tr>
<td>Coverage</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>No. of Fragments</td>
<td>39</td>
<td>66</td>
<td>127</td>
<td>352</td>
</tr>
<tr>
<td>Mean fragment length</td>
<td>395</td>
<td>343</td>
<td>399</td>
<td>406</td>
</tr>
</tbody>
</table>

Table 1: Parameters of problem instances.

5.2 Experimental setup

We perform several experiments on problem instances mentioned above, for all experiments we use algorithm setup which is shown in Table 2 except for the random experiment. Every experiment was tested 20 times on first two instances (X60189 and M15421), 10 times for third (J02459) and only once on last (BX842596) just to demonstrate time consumption. Length of action sequence vary with number of fragments.

In all experiments overlap sum and number of contigs of best fragment layout so far was measured after every iteration of POEMS algorithm. For overlap sum the higher the

Table 2: Algorithm setup for all experiments but random.

5.3 Fitness function

Both fitness functions described in Section 4.3 were tested, using setup shown in Table 2.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>10243.3 / 3.1</td>
<td>15833.2 / 6.4</td>
<td>29208.6 / 19.9</td>
<td>61834.7 / 72.9</td>
</tr>
<tr>
<td>f2</td>
<td>9065.7 / 1.0</td>
<td>12397.2 / 1.0</td>
<td>14394.4 / 1.0</td>
<td>15118.3 / 1.2</td>
</tr>
</tbody>
</table>

Table 3: Results of fitness function experiment. Showing overlap sum and number of contigs.

As expected using the second fitness function the algorithm produces only one contig for almost all benchmarks, but the overlap sum is way worse than the first fitness function. The reason for poor overlap score of the second fitness function is that the value of overlap sum is not constantly increasing but can decrease every time when the number of contigs decreases.

In Figure [10] you can see the progress of overlap sum over all generations for all tested benchmark. This graphs shows that $f1$ fitness function converge very quickly to its local optimum and then stagnate there. Second fitness function converge slower than first especially on harder benchmarks.
5.4 Actions

We test which action has bigger influence on quality of final solutions. We run eight experiments for every benchmark in two sets, each set uses different fitness function \((f_1, f_2)\), first test in set uses only move and switch action, second test uses only swap and switch action, the next test uses only invert and switch action and the last test uses all actions. All the tests uses switch action because swap and move actions do not change the directions of fragments.

Results of actions experiment using \(f_1\) fitness function shown in Table 4 shows that the move action is better for maximizing the overlap sum, however the invert function is better for minimizing number of contigs. The swap action does not show any outstanding results.

Results of the second part of the action experiment, which use \(f_2\) fitness function, is shown in Table 5. These results only strengthen the conclusion of the first part of this experiment.
### Table 4: Results of actions experiment with $f_1$ fitness function. Showing overlap sum and number of contigs.

<table>
<thead>
<tr>
<th>Exp.</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>10243.3 / 3.1</td>
</tr>
<tr>
<td>move</td>
<td>10279.8 / 4.5</td>
</tr>
<tr>
<td>swap</td>
<td>8562.6 / 7.0</td>
</tr>
<tr>
<td>invert</td>
<td>9374.3 / 3.4</td>
</tr>
</tbody>
</table>

### Table 5: Results of actions experiment with $f_2$ fitness function. Showing overlap sum and number of contigs.

<table>
<thead>
<tr>
<th>Exp.</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>9065.7 / 1.0</td>
</tr>
<tr>
<td>move</td>
<td>7832.8 / 1.0</td>
</tr>
<tr>
<td>swap</td>
<td>5781.4 / 1.0</td>
</tr>
<tr>
<td>invert</td>
<td>6199.9 / 1.0</td>
</tr>
</tbody>
</table>

### 5.5 Random

Random experiment should prove that POEMS evolutionary step can effectively improve action sequence, therefore during this experiment no evolutionary computation is applied. Population is randomly generated, evaluated and best action sequence is selected. We set up this experiment to have the same number of evaluations as experiments which uses evolutionary computation, therefore random experiment uses niche size of 220 individuals. Result of this experiment should be significantly worse than other experiments if so the genetic algorithm has capacity to improve population. Results of random experiment are shown in Table 6.

The results of random experiment has proved that Genetic Algorithms has better chance to find optimal action sequence than random generation.
5.6 Discussion

The major disadvantage of our algorithm is its time consumption. But the result are promising and comparable to other assembly projects.

5.6.1 Comparison to PALS

We chose Problem Aware Local Search see Section 2.4 as it is similar to our approach. We can include only the number of contigs for our comparison, because we do not know how the final fitness value, which is provided in [11], is computed.

<table>
<thead>
<tr>
<th>Exp.</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>10243.3 / 3.1</td>
</tr>
<tr>
<td>f1 random</td>
<td>9097.0 / 6.4</td>
</tr>
<tr>
<td>f2</td>
<td>9065.7 / 1.0</td>
</tr>
<tr>
<td>f1 random</td>
<td>6199.9 / 1.0</td>
</tr>
</tbody>
</table>

Table 6: Results of random experiment. Showing overlap sum and number of contigs.

Results which you can see in Table 7 shows that our approach, using $f2$ fitness function, is better than PALS in minimizing number of contigs. However we know from fitness function experiment that $f2$ is significantly worse than $f1$ fitness function concerning the overlap sum.

5.6.2 Time consumption

Algorithm was implemented in Java 1.6.0 and experiments were performed on Linux machine with Intel Pentium M Processor 2.0 GHz with 1.2 GB of memory.
Time consumption is major problem of our implementation of POEMS approach. In Figure 11 is shown that in worst run on BX842596 benchmark, with 773 fragments, computation takes up to thirteen hours. This is partly caused by additional computation for experimental usage such as the computation of overlap sum and number of contigs after each iteration. However main cause of this high time consumption is inefficiency of fitness function evaluation.

The main difference of computation times is between experiments using the first ($f_1$) and the second ($f_2$) fitness function. This is mainly because the first fitness function is computed twice for every evaluation when the second fitness function is computed only once.

![Figure 11: Time consumption of all experiments.](image-url)

High time consumption of our algorithm is mainly created by inefficient implementation of fitness function computation. We suggest number of improvements.

First fitness function is computed from two overlap sums one before and second after applying action sequence, both are computed for every evaluation. But because the prototype is same for all evaluation in one iteration, first overlap sum computation can
be computed only once for each iteration. This improvement should clear the difference
between first and second fitness function.

Next improvement involve both fitness functions, every action sequence affects only small number of fragments and the rest is left unattached but the both fitness functions are computed from all fragments. Computing fitness function only from affected fragments should rapidly improve time consumption. Similar approach was described in [11].

5.6.3 Other algorithm improvements

We use two arrays for solution, one for order and second for orientation, but we come up with second representation using only one array. In one array representation the order is stored as in two arrays but direction is stored in sign, positive values marks forward fragments and negative backward. This representation should simplify application of actions.
Conclusion

The DNA Fragment Assembly is complicated combinatorial optimization problem in bioinformatic. It is NP-hard problem, therefore finding the solution by exploring every possibility is impossible, with exception of small instances. That's why other methods such a heuristic are used.

We have implemented Iterative Prototype Optimization with Evolved Improvement Step a local search, which is using Genetic Algorithm to evolve best possible action sequence, which are then applied on prototype solution. We have studied its performance on Fragment Assembly Problem.

We perform several experiments on artificially generated problem instances, to estimate optimal setup. Our results shows that this approach is promising and comparable to other assembly projects, but very inefficient and time consuming. We have suggested improvements which should improve this inefficiency.

Possible future work should be aimed to improve efficiency by applying our suggestions. The next step is to take in account laboratory errors ad apply our approach on real problem instances.
References


A Attached CD

CD attached to this document contain:

- Electronic version of this document.
- Source code of all experiments.
- Problem instances in FASTA\footnote{www.ncbi.nlm.nih.gov/blast/fasta.shtml} file format.
B Tables and figures
<table>
<thead>
<tr>
<th>exp.</th>
<th>min</th>
<th>mean</th>
<th>max</th>
<th>min</th>
<th>mean</th>
<th>max</th>
<th>min</th>
<th>mean</th>
<th>max</th>
<th>min</th>
<th>mean</th>
<th>max</th>
<th>min</th>
<th>mean</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>9583</td>
<td>10243.3</td>
<td>10915</td>
<td>15004</td>
<td>15833.2</td>
<td>17227</td>
<td>26775</td>
<td>29208.6</td>
<td>30172</td>
<td>59935.0</td>
<td>61884.7</td>
<td>63760</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f1-move</td>
<td>9773</td>
<td>10279.8</td>
<td>11074</td>
<td>14740</td>
<td>15429.6</td>
<td>16420</td>
<td>28942</td>
<td>30474.5</td>
<td>32476</td>
<td>66220</td>
<td>69340.7</td>
<td>72789</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f1-swap</td>
<td>7534</td>
<td>8562.6</td>
<td>9706</td>
<td>12033</td>
<td>12791.3</td>
<td>13614</td>
<td>24225</td>
<td>26152.9</td>
<td>28025</td>
<td>60033</td>
<td>61371.1</td>
<td>63383</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f1-invert</td>
<td>8659</td>
<td>9374.3</td>
<td>9972</td>
<td>12223</td>
<td>13251.1</td>
<td>15066</td>
<td>19304</td>
<td>22023.0</td>
<td>25269</td>
<td>34444</td>
<td>36624.6</td>
<td>39490</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f1-random</td>
<td>8051</td>
<td>9097.0</td>
<td>9731</td>
<td>12082</td>
<td>12849.5</td>
<td>13530</td>
<td>18646</td>
<td>20511.9</td>
<td>22374</td>
<td>40751</td>
<td>42300.4</td>
<td>43971</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td>1.3187</td>
<td>1.4117</td>
<td>1.4693</td>
<td>1.3837</td>
<td>1.4527</td>
<td>1.5218</td>
<td>1.6618</td>
<td>1.7156</td>
<td>1.7688</td>
<td>1.8701</td>
<td>2.0941</td>
<td>2.8948</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f2-move</td>
<td>1.3963</td>
<td>1.4917</td>
<td>1.5941</td>
<td>1.4172</td>
<td>1.5305</td>
<td>1.6073</td>
<td>1.6641</td>
<td>1.7243</td>
<td>1.7618</td>
<td>1.8775</td>
<td>2.1686</td>
<td>3.8990</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f2-swap</td>
<td>1.5677</td>
<td>1.6249</td>
<td>1.6877</td>
<td>1.6091</td>
<td>1.6631</td>
<td>1.7156</td>
<td>1.7653</td>
<td>1.7955</td>
<td>1.8360</td>
<td>1.8984</td>
<td>2.1058</td>
<td>2.9053</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f2-invert</td>
<td>1.3738</td>
<td>1.4787</td>
<td>1.6507</td>
<td>1.4675</td>
<td>1.5550</td>
<td>1.6166</td>
<td>1.6964</td>
<td>1.7619</td>
<td>1.8513</td>
<td>1.8920</td>
<td>2.6086</td>
<td>4.9368</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table B.8: Final fitness function values for all experiments.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>mean</td>
<td>max</td>
<td>min</td>
</tr>
<tr>
<td>f1</td>
<td>9583</td>
<td>10243.3</td>
<td>10915</td>
<td>26775</td>
</tr>
<tr>
<td>f1-move</td>
<td>9773</td>
<td>10279.8</td>
<td>11074</td>
<td>28942</td>
</tr>
<tr>
<td>f1-swap</td>
<td>7534</td>
<td>8562.6</td>
<td>9706</td>
<td>14740</td>
</tr>
<tr>
<td></td>
<td>12033</td>
<td>12791.3</td>
<td>13614</td>
<td>24225</td>
</tr>
<tr>
<td>f1-invert</td>
<td>8659</td>
<td>9374.3</td>
<td>9972</td>
<td>19304</td>
</tr>
<tr>
<td>f1-random</td>
<td>8051</td>
<td>9097.0</td>
<td>9731</td>
<td>18646</td>
</tr>
<tr>
<td>f2</td>
<td>8179</td>
<td>9065.7</td>
<td>10500</td>
<td>11703</td>
</tr>
<tr>
<td>f2-move</td>
<td>6256</td>
<td>7832.8</td>
<td>9304</td>
<td>12054</td>
</tr>
<tr>
<td>f2-swap</td>
<td>4813</td>
<td>5781.4</td>
<td>6662</td>
<td>8302</td>
</tr>
<tr>
<td>f2-invert</td>
<td>5383</td>
<td>8034.0</td>
<td>9651</td>
<td>7528</td>
</tr>
<tr>
<td>f2-random</td>
<td>4743</td>
<td>6199.9</td>
<td>7220</td>
<td>5500</td>
</tr>
</tbody>
</table>

Table B.9: Final values of overlap sum for all experiments.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>mean</td>
<td>max</td>
<td>min</td>
</tr>
<tr>
<td>f1</td>
<td>1.0</td>
<td>3.1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>f1-move</td>
<td>2</td>
<td>4.5</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>f1-swap</td>
<td>4</td>
<td>7.0</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>f1-invert</td>
<td>1</td>
<td>3.4</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>f1-random</td>
<td>3</td>
<td>6.4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>f2</td>
<td>1</td>
<td>1.0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>f2-move</td>
<td>1</td>
<td>1.0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>f2-swap</td>
<td>1</td>
<td>1.0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>f2-invert</td>
<td>1</td>
<td>1.0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>f2-random</td>
<td>1</td>
<td>1.0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table B.10: Final number of contigs for all experiments
Figure B.12: Summary of contigs progress of all experiments.
Figure B.13: Summary of overlap sum progress of all experiments.