Università degli Studi di Milano Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di laurea in Informatica

Integer Linear Programming approaches on the DNA recombination problem

Relatore: *Bonizzoni Paola*
Co-relatore: *Della Vedova Gianluca*

Relazione della prova finale di:
Antonio Vivace
Matricola 793509

Anno Accademico 2016–2017

Abstract

We introduce the *Computational Biology* field, reporting its multidisciplinary appeal and increasing relevancy. We describe its general method, main application areas and challenges.

We proceed familiarising with *Integer Linear Programming*, defining its inception, uses and approach, and how ILP-based approaches have become a standard optimization technique in bioinformatics, reviewing the solving algorithms and the general mathematical method.

Then, we formalize the "DNA Recombination and Rearrangement" (or "Guided DNA Assembly") problem based on the what is observed in some species of ciliates, followed by an analysis and report of some of existent approaches and their central ideas, limitations and reductions applied.

Finally, a tentative ILP formulation of the DNA Recombination problem is given, describing the implementation tools used and the main encountered difficulties.

Contents

1 — Introduction

1.1 Computational Biology

Computational Biology is defined as the development and application of data-analytical and theoretical methods, mathematical modeling and computational simulation techniques to the study of biological, behavioral, and social systems[1].

The field is now thirty years old and it's covered by many conferences and journals publishing papers. It features graph theory, network flows, combinatorics, integer and linear programming problems, statistical approaches, probabilistic methods, hidden Markov models, neural networks as its tools.

Some of the most important challenges are[2]:

- Protein structure prediction.
- Homology searches.
- Multiple alignment and phylogeny construction.
- Genomic sequence analysis and gene-finding.

In particular, *Computational Molecular Biology* (bioinformatics) focuses on studying existing and emerging approaches, techniques and algorithms for string computation (sequences) providing a significant intersection between computer science and molecular biology [3].

For these reasons, the field is inherently multidisciplinary: it's appealing to the Mathematical Programming and Operations Research community. Today, computational biology papers are written by computer scientists, biologists, statisticians, physicists and mathematicians, pure and applied.

Concretely, the application areas are [4]:

- Evolution. Comparison of whole genomes to highlight evolutionary macro events (inversions, transpositions, translocations). Computation of evolutionary distances between genomes. Computation of common evolutionary subtrees or of evolutionary supertrees.

Figure 1.1: Five types of evolutionary events

- Sequence analysis. Comparison of genomic sequences within individuals of a same species, or intra-species, in order to highlight their differences and similarities. Reconstruction of long sequences by assembly of shorter sequence fragments. Error correction for sequencing machines.

- Hybridization and microarrays. Use of hybridization for sequencing. Use of microarrays for tissue identification, clustering and feature selection discriminating healthy from diseased samples. Design of optimal primers for PCR experiments. Physical mapping (ordering) of probes by hybridization experiments.

- Protein structures. Protein fold prediction from aminoacid sequence (ab-initio), or from sequence + other known structures (threading of sequences to structures). Alignment of RNA sequences depend-

ing on their structure. Protein fold comparison and alignment of protein structures. Study of protein docking and synthesis of molecules of given 3D structure.

- HAPLOTYPING. (DNA mutations) Reconstruction and/or correction of haplotypes from partial haplotype fragments or from genotype data. Analysis of resulting haplotypes and correlation with genetic diseases [5].

Note that the term *bioinformatics* is used also as an umbrella term for the (wider) body of biological studies using computer programming as part of their methodology, as well as a reference to specific analysis "pipelines" that are repeatedly used, particularly in the field of genomics.

2 — Integer Programming

2.1 Definition

Linear programming (ILP) is a technique for the mathematical optimization of a linear objective function, subject to linear equality and linear inequality constraints.

Linear programs are problems that can be expressed in canonical form as:

$$\begin{array}{ll} \text{maximize} & \mathbf{c}^{\mathrm{T}}\mathbf{x} \quad \text{(cost function)} \\ \text{subject to} & A\mathbf{x} \leq \mathbf{b} \\ \text{and} & \mathbf{x} \geq \mathbf{0} \\ & (\mathbf{x} \in \mathbb{Z}^n) \end{array}$$

If the variables are forcibly constrained to be integers, we call the program *Integer* or *Integer Linear* (ILP).

0-1 integer programming or binary integer programming (BIP) is the special case of integer programming where variables are required to be 0 or 1 ($\mathbf{x} \in \{0, 1\}$).

A particular case of integer linear programming is represented by Combinatorial Optimization (CO), that is the class of problems in which the feasible region is a subset of the vertices of the unit hypercube $F \subseteq \mathbf{B}^n = \{0, 1\}^n$, i.e., more simply, problems in which variables can only take value 0 or 1. Linear {0,1} (or binary) programming problems belong to this class [6].

In contrast to linear programming, which can be solved efficiently in the worst case, integer programming problems are in many practical situations (bounded variables) NP-hard and no general algorithm is known. Binary Integer Programming problems are classified as NP-hard too: "0–1 integer programming" is one of the *Karp's 21 NP-complete problems*.

2.2 Algorithms

There are three main categories of algorithms for solving integer linear programming problems [7]:

- EXACT algorithms that guarantee to find an optimal solution, but may take an exponential number of iterations. They include cutting-planes, branch-and-bound, and dynamic programming.
- HEURISTIC algorithms that provide a suboptimal solution, but without a guarantee on its quality. Although the running time is not guaranteed to be polynomial, empirical evidence suggests that some of these algorithms find a good solution fast.
- APPROXIMATION algorithms that provide in polynomial time a suboptimal solution together with a bound on the degree of sub-optimality.

2.3 In Computational Biology

At its inception, the focus of Computational Biology was on the development of efficient algorithms and data structures that were able to deal with the data being introduced in life science applications. Lately, the introduction of high throughput methods for biomedical data analysis and the rise of Systems Biology (the study of systems of biological components) made Statistical Learning approaches a standard [8].

Furthermore, new and accessible sequencing methods caused an exponential growth of the available genomic data.

This element and the fact that biological processes are usually reduced and studied as simulations (because the actual nature of them is still being investigated, as in the case of our problem) lead to the introduction of a lot new optimization problems in the field.

In most cases, these optimization problems are discrete ones: hence the approval of ILP-based approaches as a standard.

Some of the most successful Integer Programming approaches for computational biology problems are described in [9].

2.3.1 Advantages

There are a number of additional reasons why ILP should be taken into consideration, even when the problems seems to not require it or the advantage of introducing an ILP formulation isn't initally clear[10]:

- Commercial ILP *solvers* are available (with academic licenses);
- The progress of those solvers has been spectacular: benchmark ILP problems can be solved *200-billion* times faster than twenty-years ago;
- Even for a problem where a worst-case efficient general algorithm might be possible, the time and effort needed to find it, implement it as a computer program, is typically much greater than the time and effort needed to formulate and implement an ILP solution to the problem.
- Some problems can be modeled in a much more efficient way with ILP.
- A new mathematical formulation for classic problems may be studied, allowing the original one to be attacked in new ways. New techniques and relaxations can be applied.

To give a real example, the widely studied Multiple Sequence Alignment problem [11] (or Multiple String Comparison) is one of the most important methodological issues of the field, it shows how many different approaches, versions and formulations can be theorized and exploited: it was reformulated as an optimization problem introducing the concept of *trace* in [12], given branch-and-cut algorithms in [13] and relaxations, such as [14], which proposes a branch-and-bound algorithm with a Lagrangian relaxation.

Among others, [15] reduce the multiple alignment problem to the minimum routing cost tree (MRCT) problem, i.e., finding a spanning tree in a complete weighted graph, which minimizes the sum of the distances between each pair of nodes. They propose a Branch-and-Price algorithm for the MRCT problem and then use it. [16] reduce multiple sequence alignment to a facility location problem. The reduction is then used to provide a Polynomial Time Approximation Scheme for a certain class of multiple alignment problems.

The history of the problem, biological motivations and uses along with many approaches are discussed in *Multiple String Comparison - The Holy Grail*, in [3].

2.4 Design of an ILP formulation

A computational biology problem is generally tackled in this way:

First, a modeling analysis is performed, trying to describe and formalize the underlying biological process into one or more combinatorial objects. The question concerning the biological data is now a mathematical question about the chosen objects. Each object representing a tentative solution has a numerical value associated to it (computed using the *cost* or *objective* function) to measure its quality. We want to find a solution x^* which *maximizes* (or *minimizes*, based on the formulation) $f(x)$ over all the other possible solutions.

2.4.1 Idioms

Here's how many logic expressions can be expressed as linear inequalities without side effects or uncovered cases [10].

Suppose L is an integer linear function of binary variables with M being its upper limit and b a positive integer.

If-Then

$$L \geq b \rightarrow z$$

Linearly:

$$L - (M \times z) \leq b - 1$$

Only-If

$$z = 1 \text{ only if } L \geq b$$

Let s be the smallest value that L can achieve and set $m = s - b$. Linearly:

$$L + m \times z \geq m + b$$

These two idioms can be used as building blocks for many more:

NAND

Let L_1 and L_2 be linear functions whose variables are bounded, and $L_1 \geq b_1$ and $L_2 \geq b_2$. We require that at *most* one of the two linear inequalities is satisfied.

$$z_1 + z_2 \leq 1$$

Where $z_1 = 1$ if $L_1 \geq b_1$ and $z_2 = 1$ if $L_2 \geq b_2$. We use the *If-Then* twice idiom to express these two conditions.

OR

Here we require that at *least* one of the two linear inequalities is satisfied.

$$z_1 + z_2 \geq 1$$

Followed by two *Only-If* idioms to express $z_1 = 1$ *only* if $L_1 \geq b_1$ and $z_2 = 1$ *only* if $L_2 \geq b_2$.

XOR

If we want *exactly* one of the inequalities to be satisfied:

$$z_1 + z_2 = 1$$

Again, followed by two *Only-If* idioms to express $z_1 = 1$ *only* if $L_1 \geq b_1$ and $z_2 = 1$ *only* if $L_2 \geq b_2$ and two *If-Then* (if and only if).

Implied Satisfaction

To express

$$L_1 \geq b_1 \rightarrow L_2 \geq b_2$$

We need an *If-Then* idiom for the first equality, an *Only-If* idiom for the second and

$$z_1 \leq z_2$$

Not-Equal

Suppose Z_1 and Z_2 are linear functions of integer variables whose values are bounded. Then $Z_1 - Z_2$ and $Z_2 - Z_1$ are bounded to integer values, too. Then, we can express

$$Z_1 \neq Z_2$$

as

$$(Z_1 - Z_2 \geq 1) \text{ OR } (Z_2 - Z_1 \geq 1)$$

Using our *OR* idiom previously explained: let s_1 the lower bound for $Z_1 - Z_2$ and s_2 the lower bound for $Z_2 - Z_1$. Set $m_1 = s_1 - 1$ and $m_2 = s_2 - 1$. The final inequalities will be

$$(Z_1 - Z_2) + m_1 \times l_1 \geq m_1 + 1$$
$$(Z_2 - Z_1) + m_2 \times l_2 \geq m_2 + 1$$
$$l_1 + l_2 \geq 1$$

Many of these idioms can be reduced if some or all variables are binary, strictly positive, or bounded.

3 — The DNA Recombination problem in ciliates

3.1 Biological Background

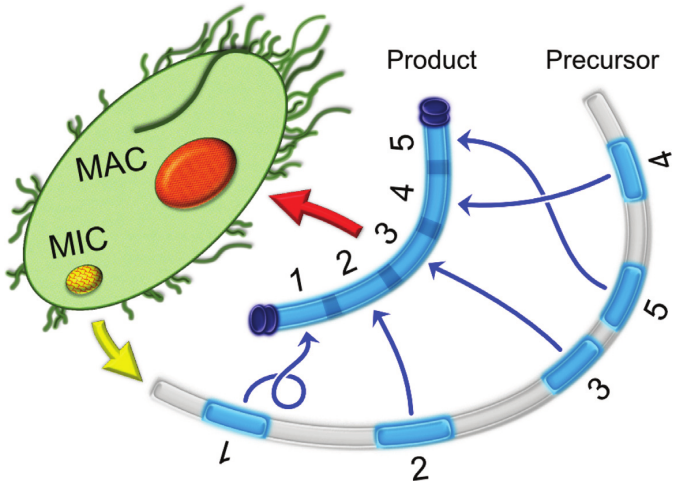

Figure 3.1: In the somatic macronucleus (MAC), chromosomes assemble from precursor MDS building blocks (blue), which may be scrambled in some species. In the germline micronucleus (MIC), the Macronuclear Destined Sequences (MDSs) for all somatic chromosomes are dispersed over the long chromosome, and interrupted by Internally-Eliminated Sequences (IESs) and other noncoding DNA (gray). In some cases, an MDS may appear in a permuted order, or inverted[17].

Ciliated protists (microbial eukaryotes using cilia for locomotion) exhibit nuclear dimorphism through the presence of separate germline and somatic nuclei. The somatic macronucleus (MAC) provides templates for the transcription of all genes required for asexual growth while the germline micronucleus (MIC) is used for the exchange of meiotic products during sexual reproduction [17]. The MAC DNA is the one actively expressed and effectively results in the phenotype of the organism.

Several species of ciliates, such as *Stylonychia* or *Oxytricha*, go through extensive gene rearrangement while differentiating somatic macronuclei from germline micronuclei. This process entails an extensive fragmentation, elimination and sometimes broader rearrangement of the germline DNA, coupled to DNA amplification and telomere addition [18] and form the somatic macronuclei, all under the epigenetic control of novel non-coding RNA pathways [19]. The extent and the nature of these operations varies among ciliate species.

Each gene in the macronucleus may be present in the micronucleus as several nonconsecutive segments (macronuclear destined sequences, **MDSs**) separated by non-coding DNA. During macronuclear differentiation, the non-coding fragments (internal eliminated sequences, **IESs**) that interrupt MDSs in the micronucleus are deleted. Moreover, the order of the MDSs in the micronucleus may not be consecutive, in which case formation of the macronucleus requires unscrambling of the MDS order, as well as IES removal. There exist **pointer**-like sequences that are repeated at the end of the nth MDS and at the beginning of the $(n + 1)$st MDS in the micronucleus. Each pointer sequence is retained as only one copy in the sequence in the macronucleus [20].

The general RNA-guided mechanism that regulate and lead this process of assembly is not known, theoretical investigations can be found in [21] and [22].

3.2 Biological Motivation

The guided genome rearrangement problem has (and it's) been extensively [22] studied, both as biochemical process and mathematical model, as it provides an exaggerated case of a phenomenon observed among different species in different ways [23]. Similar broad scale, somatic rearrangement events occur in many eukaryotic cells and tumors.

Many discrete and topological models, mathematical approaches, biological and biochemical explanations and speculations on the theme can be found in literature (such as [24] [25] [20] and [19]).

3.3 Formalization

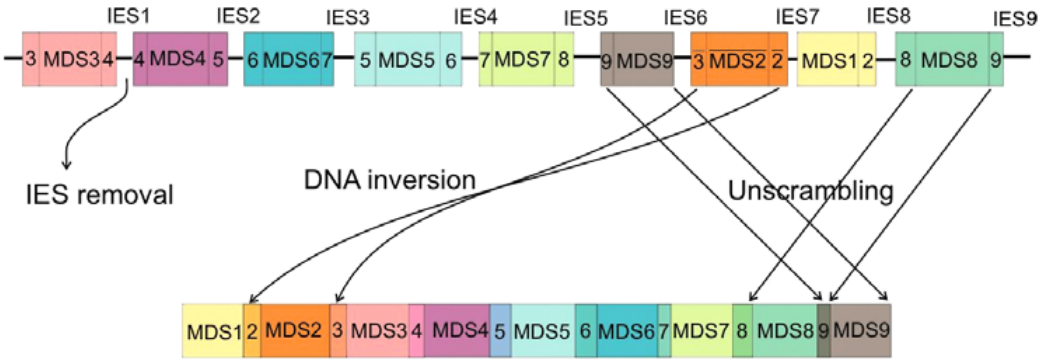

Figure 3.2: Schematic representation of the scrambled Actin I micronuclear germline gene in Oxytricha nova (top) and the correctly assembled macronuclear gene (bottom). Each block represents an MDS, and each line between blocks is an IES. The numbers at the beginning and at the end of each segment represent the pointer sequences. Note that MDS3-MDS8 require permutation and inversion to assemble into the orthodox, linear order MDS1-MDS9 in the macronucleus. The bars above MDS2 and its pointers indicate that this block is inverted relative to the others, i.e., this sequence is the Watson - Crick reverse complement of the version in the macronucleus; from[26].

The recognized events in the rearrangement process are:

- The MAC begins a copy of the MIC DNA. The chromosomes are fragmented and amplified. The result of this process is the *precursor*. ∼90% of the complexity is lost.
  - Fragmentation
  - Amplification
- From the precursor the final MAC DNA is produced through these further operations:
  - Elimination
  - Inversion
  - Gene Scrambling - Unscrambling
  - Telomere Addition

We focus on the second phase, trying to map the following "building blocks":

- **MDSs**, the contiguous sequences copied, inverted or (order) scrambled in the MAC;
- **IESs**, sections not present in the MAC;
- **Pointers**, overlap sections between MDSs in the MAC (maybe inverted), present in multiple copies in the MIC;

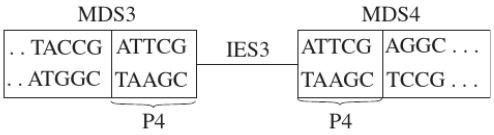

Figure 3.3: An example of a pointer sequence between two MDSs in the MIC [20].

The "inverse", "reverse", "reverse complement" terms refer to the *Watson-Crick reverse complement* of the sequence.

The goal is to produce a *rearrangement map*: a set of disjoint substrings representing the building blocks, eventual operations they will go through the process (scrambling, inversion) and their "destination" on the produced genome.

3.3.1 Real Instance

Ideally, we want to reach an approach capable of treating the entire MAC and MIC sequenced genomes of *Oxytricha trifallax* or *Tetrahymena thermophila*, available publicy online in the MDS_IES_DB ("A database of macronuclear and micronuclear genes in spirotrichous ciliates" [17]).

Oxytricha trifallax has a 487.14 M long MIC sequence, rearranging in a 71.47 M characters long MAC [17].

Formally, given an initial genome(MIC or MAC precursor) and a rearranged one (MAC) the program produces an annotation map of the process the input has been through.

3.4 Existent Approaches

mds-ies-db [17] uses the MDS/IES DNA Annotation Software (MIDAS), [27] to propose an annotation map presuming the MDSs general pattern (using regular expressions) and using the Basic Local Alignment Search Tool (BLAST [28]) to flag matching regions.

The algorithm is the following:

1. Use provided regular expression to mask macronuclear telomeric sequences.
2. Blast MAC sequences against MIC sequences.
3. Process obtained high scoring pairs list resolving overlaps to obtain initial MDS annotation.
4. Blast MAC sequences against MIC sequences again.
5. Use obtained high scoring pairs to fill gaps in the initial MDS annotation and get final MDS annotation for MAC.
6. Output final MDS/IES annotation for MAC and MIC.

An example regular expression used in step **1** is `A0,4(CCCCAAAA)+C0,4`.

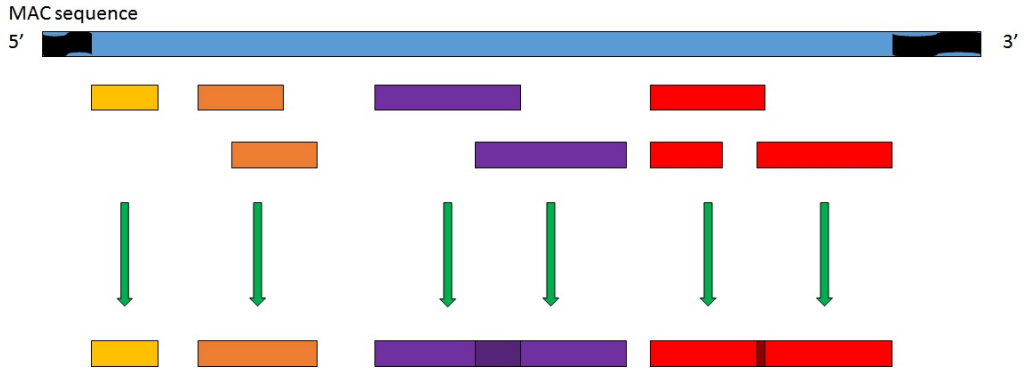

Figure 3.4: Step 3 of MIDAS: Resolve high scoring pairs overlaps by merging HSPs that overlap at least 50%. In this picture, orange HSPs do overlap at least 50% and one of the red HSPs is contained within the other (longer) red HSP. We merge these orange and red HSPs. The purple HSPs and the third red HSP does not satisfy our merging criteria, so we leave them as they are. The remaining overlap between purple blocks and between red blocks defines a pointer sequence. [27]

During step **2**, long (at least 28 nucletides) high scoring pairs (HSPs) are found using BLAST. Next, in step **3** overlapping HSPs are merged (if they overlap at least 50%). Step **4** makes sure that no portion of the MAC sequences remains uncovered, using BLAST with lower requirements (at least 12 nucletides) that can potentially fill the gaps. Finally, MIC sub-sequences that are in between HSPs are considered IESs.

Figure 3.5: Difference of two chromosomes of species A and B. The difference is described with respect to the ordering of the chromosome segments of B [29].

The recent **Sorting by Reversals and the Theory of 4-Regular Graphs** [29] shows an interesting approach in describing the *Reversal* evolutionary event and illustrates its correlation to the DNA recombination problem found in ciliates. The *reversal distance* is defined as the least number of reversals required to accomplish the transformation (an efficient algorithm to compute this distance can be found in [30]). The reversal event is then cast as a special case of double-cut-and-join (DCJ) operation (which can happen within a chromosome or between more of them). Similarly, a formula to compute the DCJ distance is given in [31].

The problem of computing the distance between two permutations is often recast as the problem of computing $d(\pi)$, the reversal distance between a permutation π and the identity permutation $(12...n)$. The reconstruction of one possible sequence of reversals that realizes $d(\pi)$ is called the *sorting by reversals* problem.

The gene assembly in ciliates is finally casted with the theory of sorting by reversals (although ignoring overlap sections).

How 4-regular multigraphs, circle graphs and delta-matroids can be used to study gene assembly in ciliates it's further discussed in [25].

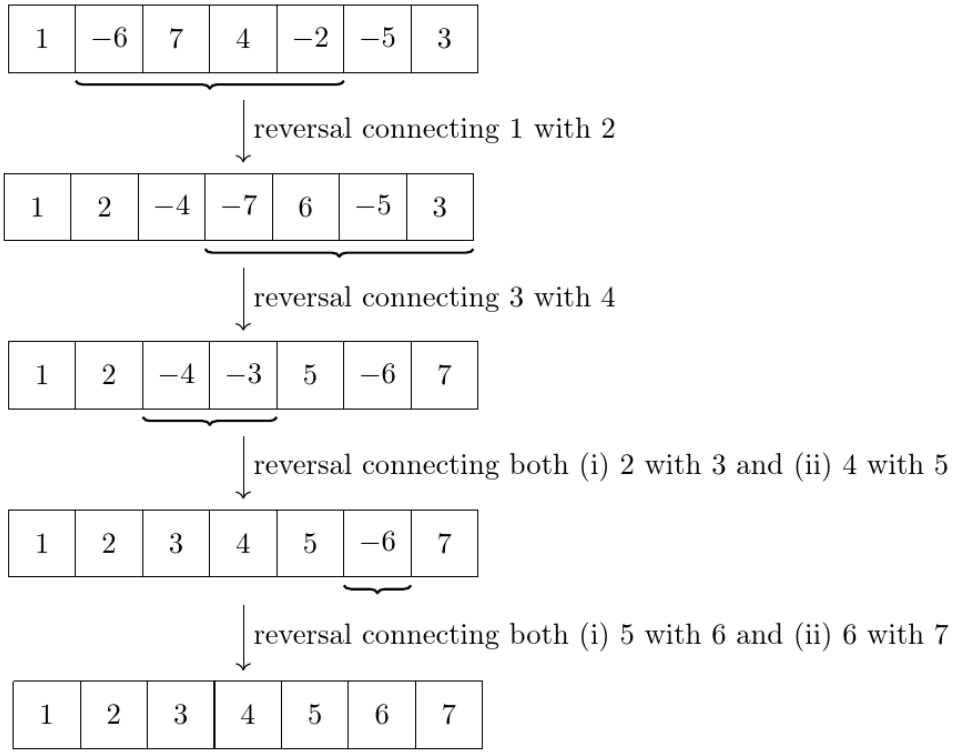

Figure 3.6: Optimal sorting of the signed permutation of Figure 3.7 by reversals [29].

DNA Recombination through Assembly Graph[23] takes a more elaborate combinatorial approach, introducing the notion of *assembly graph* as a finite connected graph where all vertices are rigid vertices of valency 1 or 4. The recombination process is then modeled using polygonal paths, tranverse paths and smoothings.

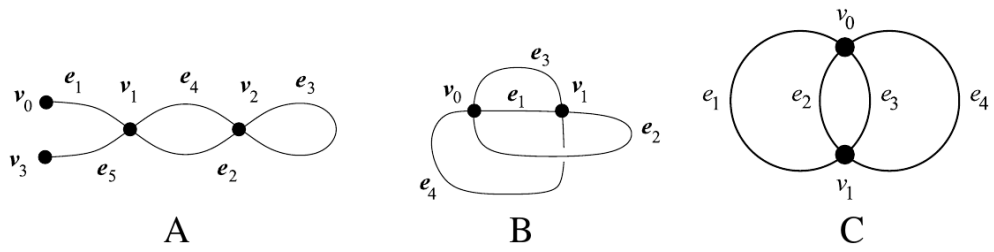

Figure 3.7: Examples of assembly graphs. Simple assembly graphs (A) and (B), and a non-simple assembly graph (C) [23].

The model proposed in **RNA-guided DNA assembly** [20] points out how RNA templates can provide a plausible explanation for the homologous recombination. It translates the three molecular operations described in the *Molecular Operation for Gene Assembly* chapter of [22] using the theorized braiding model:

- **Loop recombination** (*ld*) - A pair of repeat sequences (the pointers) flanking a non-scrambled IES between MDSs guides excision of the IES, joining the consecutive MDSs.
- **Hairping recombination** (*hi*) - It is applicable to a portion of a gene containing an inverted MDS, i.e., one copy of the pointer is an inversion of the other.
- **Double-loop recombination** (*dlad*) applies when the segment enclosed between one pair of pointers overlaps the sequence enclosed between another pair of pointers.

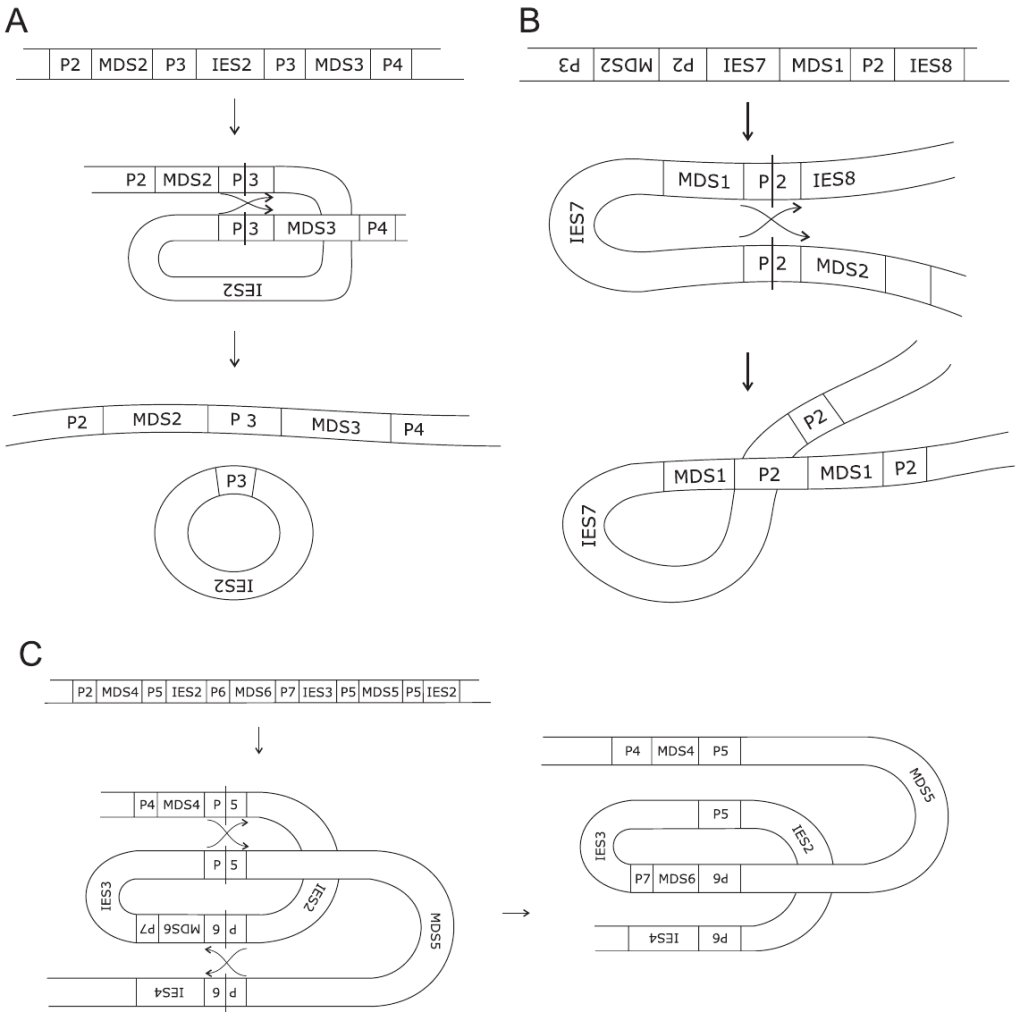

Figure 3.8: Three pointer reduction operations: (A) *ld*, (B) *hi* and (C) *dlad*.

4 — Experimentation

The entire software and documentation produced during the Stage experience and the thesis drafting, including initial and discarded attempts are available in a public Git repository [32] on GitHub.

4.1 Tools

The experimental work of the Stage experience was done on a GNU/Linux Debian `buster/sid` workstation, making large of use of the shell and other tools:

`Git`, a distributed version control system, helped to keep track of every progress in the documents and the software produced.

`Python` [33] and its `IDLE` was used to quickly implement and experiment the algorithms and procedures. It's also the interface to the Gurobi interactive shell. `Ruby` was considered too.

`Gurobi Optimizer` [34] is a commercial optimization solver. It provides a Python interface to formulate linear problems and solve them within Python scripts.

The `TeX` typesetting engine (in the `LaTex` macros environment, with some some extensions like `BiBTex` and the `pdflatex` compiler) were used to produce the documentation, the thesis and the slides.

4.2 List of produced software

The list of produced software follows. A copy is available at [32].

- `gen.py` - generate reduced artificial instances and produce rearrangement maps (4.3).
- `rmapSchema.json` - proposed rearrangement map JSON schema (4.3.1).

- `rmap.py` - example functions to parse and apply the rearrangement map format described in 4.3.1.
- `segment.py` - dynamic programming algorithm to compute every substring of a sequence.
- `ilp.py` - tentative Gurobi implementation of the proposed ILP formulation.

4.3 Reduced artificial instance

Working on the entire genomes sequences would be prohibitive for such approach, and many of the existent solutions make assumptions on the nature of the genomes, as we've seen.

We take into consideration a reduced instance, considering a shorter sequence and only the main events (Scrambling, Inversion, Overlapping, Deletion).

A Python script which *procedurally* generates an instance of the problem with given specific characteristics has been developed, it takes the following parameters to shape the desired instance:

- MIC length
- MDS quantity
- Overlap size
- Inverse rate

The generated instance consists in:

- A (randomized) MIC DNA sequence
- A rearrangement map containing positions, inversion flags and annotation for every MDS, Pointer in both MIC and MAC
- The resulting MAC sequence

Running `$ python gen.py`, we get:

```
Generated instance:

MDS     Start   End
0       3       9
1       21      27
2       30      38
3       40      50
4       13      19

P       Start1  End1    Start2  End2
1       7       9       21      23
2       25      27      30      32
3       35      38      40      43
4       46      50      13      17

MIC     ---AAATAT----TGGAGG--ATCGGT---GTAGAATT--ATTTCGTGGA----------
               ^^    ^^^^    ^^  ^^    ^^   ^^^  ^^^   ^^^^

MAC     AAATATCGGTAGAATTTCGTGGAGG
            ^^  ^^   ^^^   ^^^^
```

The parameters used were:

- 60 as MAC length
- A random value in the $2 - 5$ range as MDS quantity
- 0 as Inverse rate
- 30% as Overlap size

The pointer regions are marked in both MIC and MAC. IESs are masked for clarity. The complete annotation map is exposed through four produced objects: `MDS_MAC`, `MDS_Mic`, `Pointer_MIC` and `Pointer_MAC`, easily navigable to fetch any information about the simulated process, e.g., `Pointer_MIC[1]["Start2"]` and `Pointer_MIC[1]["End2"]` gives the position of the second occurrence of the first pointer section in the MIC. A standard JSON object, serializing this data, is produced, too.

4.3.1 (Proposed) Rearrangement map format

Here we show a simple rearrangement map format adopted in this work. A specification gives a coherent and reliable way interpret the events represented by the map. A JSON schema of the format specification is proposed in `rmapSchema.json` and it's further documented on the example library that handles it (`rmap.py`).

The necessity to *apply* them on genomes and produce simulations is described in 4.7.

The following events can be described:

- Deletion. Implicit and trivially computable.
- Scrambling. The array index represents the final ordering in the MAC. Implicit, the scrambled order can be computed using the `start` and `end` positions.
- Overlapping of pointer sections. A pointer section is a sequence common to 2 MDSs. The pointer can appear inversed in one or both occurrences.
- Inversion. An MDS can appear in the resulting genome as the Watson-Creek reverse complement version of the one in the MIC.

Here's how a rearrangement map in this format looks like, generated by *gen.py*.

```
{
  "mac_length": 18,
  "mic_length": 60,
  "mds": [
    {
      "start": 27,
      "end": 32,
      "inverted": 0
    },
    {
      "start": 3,
      "end": 10,
      "inverted": 0
    },
    {
      "start": 14,
```

```
      "end": 24,
      "inverted": 0
    }
  ],
  "pointers": [
    {
      "start1": 30,
      "end1": 32,
      "start2": 3,
      "end2": 5
    },
    {
      "start1": 8,
      "end1": 10,
      "start2": 14,
      "end2": 16
    }
  ]
}
```


This file can now be used by `rmap.py` which reproduces the events described by the map on a given genome.

4.4 ILP formulation

A tentative *pure* integer linear programming formulation follows.

4.4.1 Variables definitions

$$*Eq(i,j,h,l) = \begin{cases} 0 \\ 1, \quad \text{if } \texttt{MIC[i:j]} = \texttt{MAC[h:l]} \end{cases}$$

$$*cwc(i,j,h,l) = \begin{cases} 0 \\ 1, \quad \text{if } \texttt{MIC[i:j]} \text{ is the reverse complement of } \texttt{MAC[h:l]} \end{cases}$$

$$MDS_{MICstart}(i,j) = \begin{cases} 0 \\ 1, \quad \text{if MDS } i \text{ starts at position } j \text{ in the MIC} \end{cases}$$

$$MDS_{MICend}(i,j) = \begin{cases} 0 \\ 1, \quad \text{if MDS } i \text{ ends at position } j \text{ in the MIC} \end{cases}$$

$$MDS_{MACstart}(i,j) = \begin{cases} 0 \\ 1, \quad \text{if MDS } i \text{ starts at position } j \text{ in the MAC} \end{cases}$$

$$MDS_{MACend}(i,j) = \begin{cases} 0 \\ 1, \quad \text{if MDS } i \text{ ends at position } j \text{ in the MAC} \end{cases}$$

$$Inv(i) = \begin{cases} 0 \\ 1, \quad \text{if MDS } i \text{ is inverted in the MAC} \end{cases}$$

$$P_{start}(i,j) = \begin{cases} 0 \\ 1, \quad \text{if } MDS_{MACstart}(i,j) = 1, \text{ Pointer } i \text{ starts at position } j \text{ in the MAC} \end{cases}$$

$$P_{end}(i,j) = \begin{cases} 0 \\ 1, \quad \text{if } MDS_{MACend}(i-1,j) = 1, \text{ Pointer } i \text{ ends at position } j \text{ in the MAC} \end{cases}$$

$$Cov_{MACPOINT}(i,j) = \begin{cases} 0 \\ 1, & \text{if Pointer i covers the position j in the MAC} \end{cases}$$

$$Cov_{MIC}(i,j) = \begin{cases} 0 \\ 1, & \text{if MDS } i \text{ covers the position } j \text{ in the MIC} \end{cases}$$

$$Cov_{MAC}(i,j) = \begin{cases} 0 \\ 1, & \text{if MDS } i \text{ covers the position } j \text{ in the MAC} \end{cases}$$

Variables marked with $*$ will be populated during the preprocessing phase.

Objective Function: $\quad min \sum_{i,j} MDS_{MACstart}(i,j)$

4.4.2 Constraints

MDS integrity and validity

MDSs must correspond to identical or reverse and complemented substrings of MIC and MAC.

(1) $MDS_{MICstart}(i,a)+MDS_{MICend}(i,b)+MDS_{MACstart}(i,c)+MDS_{MACend}(i,d)+Inv(i) \leq 5cwc(a,b,c,d) \qquad \forall i,a,b,c,d$

(2) $MDS_{MICstart}(i,a)+MDS_{MICend}(i,b)+MDS_{MACstart}(i,c)+MDS_{MACend}(i,d) \leq 4Eq(a,b,c,d) \qquad \forall i,a,b,c,d$

Each MDS can start one time, both in the MAC and the MIC.

(3) $\sum_j MDS_{MICstart}(i,j) \leq 1 \qquad \forall i$

(3b) $\sum_j MDS_{MACstart}(i,j) \leq 1 \qquad \forall i$

If an MDS starts, it must end too.

(4) $\sum_j MDS_{MICend}(i,j) = \sum_j MDS_{MICstart}(i,j) \qquad \forall i$

(4b) $\sum_j MDS_{MACend}(i,j) = \sum_j MDS_{MACstart}(i,j) \qquad \forall i$

Coverage

$$(5) \qquad \sum_{l \leq j} MDS_{MICstart}(i,l) + \sum_{l > j} MDS_{MICend}(i,l) - 2Cov_{MIC}(i,j) = 0 \qquad \forall i,j$$

$$(6) \qquad \sum_{l \leq j} MDS_{MACstart}(i,l) + \sum_{l > j} MDS_{MACend}(i,l) - 2Cov_{MAC}(i,j) = 0 \qquad \forall i,j$$

Pointer Regions

Each Pointer starts when the correspondent MDS does.

$$(7) \qquad P_{start}(i,j) = MDS_{MACStart}(i,j) \qquad \forall i \neq 1$$

Each Pointer ends when the previous MDS does.

$$(8) \qquad P_{end}(i,j) = MDS_{MACEnd}(i-1,j) \qquad \forall i \neq 1$$

MAC Pointer Coverage

$$(9) \qquad Cov_{MACPOINT}(i,j) = \sum_{b \geq j} P_{start}(i,b) + \sum_{e < j} P_{end}(i,e) \qquad \forall i$$

100% MAC coverage, every part should be covered by (at least) one MDS.

$$(10) \qquad \sum_{i} Cov_{MAC}(i,j) \geq 1 \qquad \forall j$$

Overlap sections are covered by 2 MDS

$$(11) \qquad \sum_{i} Cov_{MAC}(i,j) \leq 2 \qquad \forall j$$

4.5 Preprocessing

This part of the software computes the value for some of the variables, taking the instance as input.

Some of the defined variables are *4-dimensional MIC length* × *MIC length* × *MAC length* × *MAC length* arrays. The necessity of a sparse data structure was immediately clear: *Sparray*, a Python module [35] for sparse n-dimensional arrays using *dictionaries* supporting any number of dimensions and any size was chosen to support these variables.

The Read-Write performance on these objects is satisfying: 15 milion integer values are written in random indexes in a 150M × 150M × 150M × 150M *4d sparray* in less than 20 seconds. The data can then be accessed using a notation similar to the standard array one (`Sparray[Index1, Index2, Index3, Index4]`). *Eq* and *cwc* are populated during this phase, with a naive iterative algorithm.

Here we encounter our first big limitation of a *pure* linear programming approach. Populating *Eq* and *cwc* in our test instance (60 characters long MIC) was trivial but this task becomes so expensive it's infeasible with even genome sequences of more than 1000 characters.

The subsequences matching part must be approached with an high performing alignment tool like *BLAST*.

4.6 Gurobi Implementation

Here is a simple Python example in order to illustrate the use of the Gurobi Python interface. The example builds a model, optimizes it, and outputs the optimal objective value.

Maximize $x + y + 2z$
Subject to $x + 2y + 3z \leq 4$ (c_0) and $x + y \geq 1$ (c_1)
x, y, z binary

On Python:

```
from gurobipy import *

try:
    m = Model("mip1")

    x = m.addVar(vtype=GRB.BINARY, name="x")
    y = m.addVar(vtype=GRB.BINARY, name="y")
    z = m.addVar(vtype=GRB.BINARY, name="z")

    m.setObjective(x + y + 2 * z, GRB.MAXIMIZE)

    m.addConstr(x + 2 * y + 3 * z <= 4, "c0")
    m.addConstr(x + y >= 1, "c1")

    m.optimize()

    for v in m.getVars():
        print(v.varName, v.x)

    print('Obj:', m.objVal)

except GurobiError:
    print('Error reported')
```

Our model starts with variables definition:

```
[...]
MDS_Mic_Start = m.addVars(11, len(mic),
                              vtype=GRB.BINARY,
                              name="MDS_Mic_Start")
MDS_Mic_End = m.addVars(11, len(mic),
                            vtype=GRB.BINARY,
                            name="MDS_Mic_End")
Cov_Mac = m.addVars(11, len(mac), vtype=GRB.BINARY, name="Cov_Mac")
Cov_Mic = m.addVars(11, len(mic), vtype=GRB.BINARY, name="Cov_Mic")
[...]
```

To define constraints we make large use of the `addConstrs` method:
Add multiple constraints to a model using a Python generator expression. Returns a Gurobi *tupledict* that contains the newly created constraints, indexed by the values generated by the generator expression [34].

Here is how some constraints described in 4.4 are translated in a Gurobi model:

$$(3) \qquad \sum_j MDS_{MICstart}(i,j) \leq 1 \qquad \forall i$$

$$(3b) \qquad \sum_j MDS_{MACstart}(i,j) \leq 1 \qquad \forall i$$

```
m.addConstrs((sum(MDS_Mic_Start[i,a] for a in range(len(mic))) <= 1
                    for i in range(11)), name="3")

m.addConstrs((sum(MDS_Mac_Start[i,a] for a in range(len(mac))) <= 1
                    for i in range(11)), name="3b")
```

$$(10) \qquad \sum_i Cov_{MAC}(i,j) \geq 1 \qquad \forall j$$

```
m.addConstrs((sum(Cov_Mac[i,j] for i in range(11)
                  for j in range(macl)) == macl), name="c10")
```

The objective function:

$$min \sum_{i,j} MDS_{MACstart}(i,j)$$

```
m.setObjective((sum(MDS_Mic_Start[i,a] for a in range(0,len(mic))
                                       for i in range(0,11))),
                                       GRB.MAXIMIZE)
```

11 is an upper limit of MDS quantity.

4.7 Correctness

Lemma. Suppose

I to be an instance of the problem,

P to be the ILP formulation associated to I,

S to be an assignment to every variable of P satisfying the constraints.

Then it's possible to build a solution for I with cost equal to the objective function in S.

Proof. The idea is checking if S produced by Gurobi is compatible with the known rearrangement map of the instance, i.e., if the solution proposed by Gurobi represents our DNA recombination events. "Applying" the computed map to the instance MIC should build an exact copy of the MAC.

The complete test workflow follows.

1. Produce an *artificial* instance I of the problem with the software described in the *Reduced artificial instance* section. By definition, this instance exhibits the rearrangement events we want to study (Scrambling, Overlapping, Deletion, Inversion) and it's a compliant instance of the formalised problem. A *known* rearrangement map R_1 is produced too.

2. Run the preprocess script on the generated instance and populate some of the variables of the ILP formulation associated to I

3. Run the Gurobi implementation of P passing the populated variables. An assignment S of every variable of the formulation will be computed.

4. Build a rearrangement map R_2 using S.

5. If S is a solution then it R_2 maps the rearrangement events. It is possible to simulate those events on the instance MIC and exactly obtain the MAC. R_2 is comparable with R_1.

Step 1 is described in 4.3 while steps 2-4 are handled in the `ilp.py` script. Step 5 is covered in `rmap.py` where we provide a function that computes a final rearranged sequence given an initial sequence and a (compliant) rearrangement map 4.3.1.

This entire procedure can be pipelined and automatized to allow running a variety of tests.

4.8 Conclusions

Among other problems, inequalities like 1 and 2 described in 4.4 could not be used in Gurobi, being too memory aggressive. They had to be replaced with constraints built using native Gurobi methods, allowing more than bare linear algebra.

A *pure* integer linear programming approach in this terms is clearly not successful, given the magnitude of the data to process and our choices.

A mixed approach should give better results: preprocessing the common substrings (BLAST) and producing possible instances consisting of compatible subsets of matching substrings speeds up the initial part and won't bloat the ILP formulation with huge matrices of data.

A scoring function must be designed to measure the quality of the possible maps: an ILP formulation could then optimize the problem.

Note that we (and other approaches like MIDAS [27]) use a greedy criteria for MDSs annotation: a solution with the largest possible MDSs annotation is considered the best, requiring a 100% MAC coverage.

As far as we know the process could actually behave differently: new speculations and the availability of transitional genomes, showing the process during intermediate phases, could change this view.

Bibliography

[1] NIH Biomedical Information Science and Technology Initiative Consortium. NIH working definition of bioinformatics and computational biology, 2000.

[2] David B. Searls. "Chapter 1 - Grand challenges in computational biology ". In David B. Searls Steven L. Salzberg and Simon Kasif, editors, *Computational Methods in Molecular Biology*, volume 32 of *New Comprehensive Biochemistry*, pages 3 – 10. Elsevier, 1998.

[3] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, New York, NY, USA, 1997.

[4] Giuseppe Lancia. Mathematical programming in computational biology: an annotated bibliography. *Algorithms*, 1(2):100–129, 2008.

[5] Paola Bonizzoni, Gianluca Della Vedova, Riccardo Dondi, and Jing Li. The haplotyping problem: An overview of computational models and solutions. *Journal of Computer Science and Technology*, 18(6):675–688, Nov 2003.

[6] Federico Malucelli. "Introduction to Operation Research - Integer Linear Programming".

[7] Laura Galli. Algorithms for Integer Programming. 2014.

[8] Ernst Althaus, Gunnar W. Klau, Oliver Kohlbacher, Hans-Peter Lenhof, Knut Reinert. Integer Linear Programming in Computational Biology. In: Lecture Notes in CS 5760.

[9] Giuseppe Lancia. Integer programming models for computational biology problems. *Journal of Computer Science and Technology*, 19(1), 2004.

[10] Dan Gusfield. Integer linear programming in computational biology tutorial. In *Integer Linear Programming in Computational Biology: An entry-level course for biologists (and other friends)*. Cambridge Press, 2018.

[11] Humberto Carrillo and David Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics*, 48(5):1073–1082, 1988.

[12] John Kececioglu. *The maximum weight trace problem in multiple sequence alignment*, pages 106–119. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[13] John D. Kececioglu, Hans-Peter Lenhof, Kurt Mehlhorn, Petra Mutzel, Knut Reinert, and Martin Vingron. A polyhedral approach to sequence alignment problems. *Discrete Applied Mathematics*, 104(1):143 – 186, 2000.

[14] Ernst Althaus and Stefan Canzar. *A Lagrangian Relaxation Approach for the Multiple Sequence Alignment Problem*, pages 267–278. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[15] Matteo Fischetti, Giuseppe Lancia, and Paolo Serafini. Exact algorithms for minimum routing cost trees. *Networks*, 39(3):161–173, 2002.

[16] Winfried Just and Gianluca Della Vedova. Multiple sequence alignment as a facility location problem. In *Proceedings of the Prague Stringology Club Workshop 2000, Prague, Czech Republic, September 2, 2000*, pages 60–70. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, 2000.

[17] Jonathan Burns, Denys Kukushkin, Kelsi Lindblad, Xiao Chen, Nataša Jonoska, and Laura F. Landweber. mds ies db: a database of ciliate genome rearrangements. *Nucleic Acids Research*, 44(D1):D703–D709, 2016.

[18] D.M. Prescott. *The DNA of Ciliated Protozoa. Microbiol.* 1994.

[19] Talya Yerlici and Laura F Landweber. Programmed genome rearrangements in the ciliate oxytricha. 2, 12 2014.

[20] Angela Angeleska, Nataša Jonoska, Masahico Saito, and Laura F. Landweber. "RNA-guided DNA assembly". *Journal of Theoretical Biology*, 248(4):706 – 720, 2007.

[21] Robert Brijder, Hendrik Jan Hoogeboom, and Grzegorz Rozenberg. *From Micro to Macro: How the Overlap Graph Determines the Reduction Graph in Ciliates*, pages 149–160. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[22] Andrzej Ehrenfeucht, Tero Harju, and Ion Petre. *Computation in Living Cells: Gene Assembly in Ciliates (Natural Computing Series)*. SpringerVerlag, 2004.

[23] Angela Angeleska, Nataša Jonoska, and Masahico Saito. Dna recombination through assembly graphs. *Discrete Applied Mathematics*, 157(14):3020 – 3037, 2009.

[24] David M. Prescott. Genome gymnastics: Unique modes of dna evolution and processing in ciliates. 1:191–8, 01 2001.

[25] Robert Brijder and Hendrik Jan Hoogeboom. *The Algebra of Gene Assembly in Ciliates*, pages 289–307. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[26] D.M. Prescott, A.F. Greslin. Scrambled actin I gene in the micronucleus of Oxytricha nova. *Developmental Genetics*, (13):66–74, 1992.

[27] USF Math-Bio Research Lab. MDS/IES DNA Annotation Software.

[28] Grzegorz M Boratyn, Christiam Camacho, Peter S Cooper, George Coulouris, Amelia Fong, Ning Ma, Thomas L Madden, Wayne T Matten, Scott D McGinnis, Yuri Merezhuk, et al. Blast: a more efficient report with usability improvements. *Nucleic acids research*, 41(W1):W29–W33, 2013.

[29] Robert Brijder. Sorting by reversals and the theory of 4-regular graphs. *CoRR*, abs/1701.07463, 2017.

[30] Sridhar Hannenhalli and Pavel A. Pevzner. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. *J. ACM*, 46(1):1–27, January 1999.

[31] Anne Bergeron, Julia Mixtacki, and Jens Stoye. *A Unifying View of Genome Rearrangements*, pages 163–173. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[32] Antonio Vivace. Integer Linear Programming approaches on the DNA Recombination problem in ciliates - Software and Documentation repository. `https://github.com/avivace/dna-recombination`.

[33] Guido Rossum. Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

[34] Gurobi Optimization Inc. Gurobi Optimizer Reference Manual, 2014.

[35] Jan Erik Solem. Sparray: Sparse n-dimensional arrays in Python.