Combinatorial Optimization for Fast Scaffolding

Ivaylo Petrov

Research Master Thesis

Research Master Thesis

Combinatorial Optimization for Fast Scaffolding

Author:
Ivaylo Petrov

Supervisor:
Rumen Andonov
Antonio Mucherino
GenScale

UNIVERSITÉ DE
RENNES 1

Abstract

In recent years the topic of genome assembly has become a focus of a number of research publications. Even though many researchers have focused on the assembly of large genomes such as humans', the assembly of much shorter genomes such as those of the chloroplast and the mitochondria organelles is yet not fully resolved. The available methods do not consider all the available information. They are also unable to provide more than one possible solution for the scaffolding problem. We tackle those unresolved situations by using exact methods that consider all the available information. Our methods are also capable of producing all the optimal solutions. Therefore, we believe our approach will be very beneficial for the biologists.

Contents

1 Introduction

1.1 Context

For a long time researchers have been fascinated with the diversity of living beings on Earth, the study of the differences and the similarities among species. With the advent of technologies, more and more characteristics of species have been documented for many living organisms. It is now known that, inside each living organism, there is a number of building units, called cells. Moreover, in each cell, there is a molecule, called DNA[1]. It consists of a sequence of *nucleotides* and encodes the genetic instructions necessary for the development and the behavior of all living organisms. Therefore, understanding the development and the behavior of living beings, is strictly related to understanding this molecule. The challenge is in the decoding of the information contained in the DNA. Several issues need to be overcome during the decoding process.

One of these issues concerns the sequence of nucleotides of the DNA. That sequence is different for each individual, differing from person to person, exception made for identical twins. Although the DNA is unique for every individual, some parts of it are common for all the people. The same rules apply to all the other species as well. Another critical issue is the following. Once the DNA molecule has been sequenced for a single individual, how to decode and understand the information that it contains?

This is somehow similar to the situation where a compiled version of a software tool in an unknown (let's say alien) programming language is available. In order to analyze such a software tool, any scientist would greatly benefit from having the source code of the program, even if they cannot really read and understand it. They could, however, find similarities in the code that lead to similarities in the software behavior, i.e. in its execution. The situation with the study of the DNA is quite similar. Scientists believe, that if they have the structure of some DNA molecule, it will be much easier for them to try to understand it.

When working with a DNA molecule, one of the main difficulties is the lack of a technology that can read the structure of an entire DNA molecule in one shot. That is why, subsequences of nucleotides are actually read from the DNA. To this end, each subsequence is named *read*.

Many technologies, capable to provide "reads", have emerged. The difference between them is not only given by the size of the reads they provide, but also by the price for each DNA sequencing and the way the reading positions are chosen. For example, for newer technologies, the reading positions are chosen at random. This leads to a need of high coverage[2] in order all the parts of the genome to have good chances of being represented. As for the difference in the read sizes, Sanger sequencing reads have sizes ranging from 500 to 1000 bases. They are quite smaller than the size of the human genome (which is around 3,234.83 mega bases), but still few times bigger than the reads of most of the newer technologies. As stated in El-Metwally et al. [2013] there is also a difference in the number of errors that are inserted during the sequencing. The difference in the price and the throughput, however, makes the newer technologies much more preferable than the old ones in a variety of situations.

Nevertheless, this gives rise to a new issue to overcome: how to gather all those small pieces together. There are two ways to approach this problem. If some reference sequence is available then, it is possible to try to position the pieces onto the reference sequence, while allowing some small changes. Although this approach has the advantage of being simple and fast, it does not

[1] There are some exceptions like a mature red blood cell.

[2] How many times the sum of the lengths of the reads is greater than the size of the genome itself

allow to obtain a genome that is completely different from the template. Furthermore, the choice of the reference genomes, in case there is more than one, affects the result and therefore this approach cannot be very accurate. For these reasons, a second approach was proposed. It consists in assembling the reads on the basis of overlaps between pairs of reads. The approach is called *de novo genome assembly*. An image representing the whole process for producing scaffolds can be seen in Figure 1.

A naive way to approach the problem of whole-genome shotgun (WGS) de novo assembly[3] is by trying each piece with every other and see if they match well together. After that, some heuristics are used to estimate the best matching. Even though this approach was not very efficient, it was used for some time. In Pevzner et al. [2001] was proposed an approach that makes use of the *de Bruijn graph*. For more information about this approach, please refer to the article Pevzner et al. [2001].

It turned out, however, that the short reads are not sufficient to represent more complex parts of the genome, including some partial repetitions. These repetitions are long parts that come at multiple positions in the genome and are longer than the read size. In order to represent these complex parts, some additional information was added to the output of the sequencing of some of the new technologies - **mated read-pairs**[4]. With this new information, the DNA structure can be represented more correctly, allowing the creation of better tools for assembling genomes.

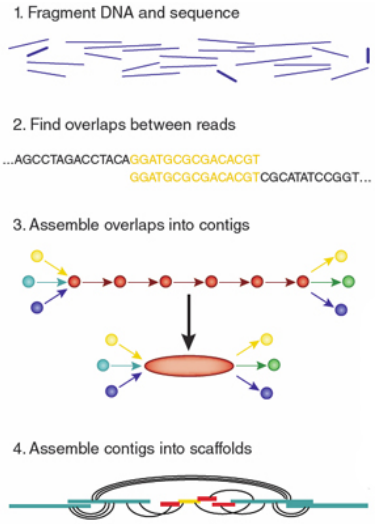

Figure 1: De novo genome assembly

1.2 Assembling strategies

The process of assembling the small reads of DNA into bigger parts can be divided into at least two phases. The first one creates contigs from the read, most commonly using some overlapping criteria. **Contigs** are gapless sequences of nucleotides. The second phase is the **scaffolding**, or the process

[3]This is the so far described process of assembling small reads from different parts of the genome, usually with very high coverage that tries to compensate the randomness of the sequenced places

[4]That is a pair of read for which we have additional information about their relative orientation and distance between them. In the literature they are also referred to as **read pairs**

of producing the best possible scaffolds (according to some criteria and some information that is presented). **Scaffolds** are generally sequences of contigs and gaps, the sizes of which are known. Often, contigs are oriented into scaffolds and ordered linearly in such a way which minimizes the number of unsatisfied scaffolding constraints (as defined in Gritsenko et al. [2012]). Scaffolding constraints might be, for example, relative position and relative orientation of pairs of contigs, inferred from the mated read-pairs.

1.3 Scaffolding

In the scientific literature, there is currently no formal and widely accepted definition for the Scaffolding problem. In an informal way, it is the inferring of the positions and the orientation of contigs along the genome, based on some possibly inconsistent information. Most often that information consists of mated read-pairs, but it can include other sources of information as well. In section 3, we will attempt to provide a formal definition of the problem.

The Scaffolding problem is the last step in the genome assembly process. Kececioglu and Myers [1995] showed that the problem is NP-hard. The main difficulty comes from fact that the available data are inherently noisy. Because of that, Scaffolding algorithms should be conceived for handling such situations.

In this report we will first describe the work that has already been done in section 2. Next we will provide a formal definition of the scaffolding problem in section 3. After that, we will present four approaches for solving it for the chloroplast and the mitochondria organelles in section 4. We will discuss the strengths and the weaknesses of each of them in section 5. Next, we will point some of the possible improvements of the presented methods in section 6. And finally, we will conclude in section 7.

2 Related work

The topic of contig scaffolding is a well studied one. A number of previous attempts have been made to provide a good solution for it. They propose several methods which have a number of positive and negative properties. A short discussion of their strengths and weaknesses will be provided shortly. One of the most common problems is the use of unreliable heuristics that might result in solution very different from the optimal one. Another widespread problem is the use of small portion of the information that is available to the scaffolder. Some of the existing methods aim to solve the problem exactly, but their performance is not always very good in regard of speed or accuracy, according to independent studies (see Hunt et al. [2014]).

- **Velvet** Zerbino et al. [2009] is a very fast algorithm. It uses a simple and fast heuristics to order the contigs in the most probable way, according to the provided information. The accuracy of this approach is generally not as good as that of the others. It also does not use the information about the contig coverage.
- **SOAPdenovo** Li et al. [2010] is another algorithm that tries to use not very complex heuristics in order to scaffold big genomes. The focus of this approach is more on speed than on accuracy. One of the contributions of this approach is its capability to use multiple data sources with multiple insert sizes. On the other hand, the solutions that it provides are not very accurate as it does not aim to fully analyze the provided data.
- **SOPRA** Dayarian et al. [2010] was the first method that would try, at least partially, to solve

the problem exactly. After that article, there have been some other ideas about how to solve the problem exactly. However, according to a number of independent studies (see Hunt et al. [2014]), they are not having the same accuracy as that approach. This is partially due to the fact that the SOPRA algorithm tries to produce correct scaffolds, rather than long ones. It also tends to run slower than a number of its competitors. In the article about SOPRA (see Dayarian et al. [2010]) it is also proposed to use partitioning of the graph in order for every partition to be solved independently. More about the partitioning will be presented in section 4.2. This idea has been employed in other approaches as well.
When the size of the component is not very big, an exact approach, based on Dynamic Programming, is used. Otherwise, a heuristic method, based on Simulated Annealing, is employed.
The SOPRA method also tries to solve the problem of overlapping of contigs exactly, by trying to find the minimum contradicting information that needs to be removed.
To sum up, this approach tries to greatly improve the quality of the scaffolds, that it produces, by better using the provided information. Sometimes the method is very slow, even though it uses some heuristics in order to improve its runtime. It also does not aim to use the contig coverage[5].

- **MIP** Salmela et al. [2011] was inspired by the idea of finding exact solution to the scaffolding problem. The difference between this approach and the previous one is that, here, the authors propose to bundle a number of edges together, so that the structure of the resulting graph is more compact. Moreover, MIP uses heuristics to delete a number of edges, that are considered as not trust-worthy. That way, the resulting graph can be partitioned into small components which can be solved exactly using mixed integer programming techniques.
  Although this technique is quite faster compared to SOPRA and tries to solve the problem exactly, it tends to produce less accurate results. It shares the same attitude to contig coverage as SOPRA - it disregards this information. Due to the time improvement, however, a number of people prefer it to SOPRA Dayarian et al. [2010].
- **Opera** Gao et al. [2011] is another method that tries to solve the problem exactly. The differences between Opera and the previous two methods are in the better grouping of the edges[6] and in the lack of any heuristics. In the paper it is explained that due to the sparse nature of most of the scaffolding graphs, it is possible to achieve an acceptable runtime for most cases even without the use of heuristics. It is also proved that if some of the properties of the graph are bounded, it is possible to solve the problem exactly in polynomial time.
  Despite being exact, this method sometimes produces less accurate results than some of its competitors, according to Hunt et al. [2014]. It also does not use the contig coverage information and it can not provide more than one possible solution.
- In the **GRASS** Gritsenko et al. [2012] paper, the authors formulate the scaffolding problem in terms of a set of scaffolding constraints that might have different weights. That is a significant difference with all the previously described methods, where the only constraints that were used are coming from the mated read-pairs.
  The problem that this approach is trying to solve is to find the most probable scaffold, that would lead to the observation that was made when the data was read. Expectation-

[5] From the text of the article it seems like this is the case. Some other articles also suggest that.

[6] In the previous method, all the edges that suggest the same orientation between a pair of contigs are grouped together. In this method only similar edges are grouped together.

maximization approach is applied for achieving this goal.

Despite providing some good ideas, this approach can not be expected to exactly solve the problem. Another drawback of this approach is that it does not consider contig coverage. It also tries to find just a single optimal solution, which might not be the most useful result for the biologists.

To sum up, the most common problems of the already provided approaches are the use of heuristics, the consideration of a part of the input data and not the whole input data, the attempt to provide only one possible solution. In the current document we will resolve all of the above problems by using exact algorithms that will consider all the available information[7].

3 Definition of the scaffolding problem

Let Γ be the input set of contigs and let $l(c)$ and $cov(c)$ be such that for any contig $c, c \in \Gamma$, $l(c)$ gives the length in number of nucleotides of the contig while $cov(c)$ gives the number of times the contig c is expected to occur in the genome. Let $N = \sum_{c \in \Gamma} cov(c)$.

Let $O = \{F, R\}$ represent the possible orientations of each of the contigs, where F stands for forward orientation and R stands for reverse orientation. Let for better notation assume that $\overline{F} = R, \overline{R} = F$ and $\overline{(o_1, o_2)} = (\overline{o_1}, \overline{o_2}), o_1, o_2 \in O$.

Let P be the set of the read pairs that are given. Let $orient : P \to O^2$ gives the relative orientation of the contigs. We should note that in the general case, there might be several read pairs for each couple of contigs $(c_1, c_2), c_1, c_2 \in \Gamma$. Let $dist : P \to \mathbb{N}$ provide the distance between the contigs for each of the read pairs. The distance is measured in number of nucleotide bases. Note that it can be negative, which means that the two contigs have some overlapping. This can be a result of big k-mers' size. Even when there is some overlapping between two contigs, if it is smaller than the size of the k-mers that was used to produce the contigs, then it is not possible to combine the two contigs together. That is when negative distances occur.

Let $contigs : P \to \Gamma^2$ be a function that for each read pair p gives the pair of contigs that it connects. Let $head : P \to \Gamma$ and $tail : P \to \Gamma$ provide the first and the second contig of $contigs(p)$ for any given $p \in P$.

Then the scaffolding problem can be defined as follows - find a triplet of functions

$$(rank, pos, or) : \begin{cases} rank : \{1, \ldots, N\} \to \Gamma \\ pos : \{1, \ldots, N\} \to \mathbb{N} \\ or : \{1, \ldots, N\} \to O \end{cases} \tag{1}$$

Where the function $rank$ gives a mapping between the numbers from 1 to N and contig occurrences. For a given number it provides the contig that is associated with that number. The number does not provide information about the ordering of the contig. That information is provided by the pos function, which is an injective function. That function provides information about the position where each contig occurrence is located on an axis. This can be used to calculate the distances between the contigs. For each given contig occurrence for contig c, that is associated with the number i ($rank(i) = c$), $pos(i)$ gives the position of the center of the contig on an axis, where

[7]Some of the approaches that we will provide lack some of the noted characteristics, but it is possible to use their solutions to help the other approaches.

that contig occurrence should be positioned. The *or* function provides orientation of each contig occurrence.

If $dist\_match : P \to \mathbb{N}$ provides a score for the degree to which the distance of every read pair is satisfied. That is to say that it provides the best score between any contig occurrences of the contigs of the read pair. A possible definition of *dist_match* would be the following:

$$dist\_match(p) : \begin{cases} & \text{if } \exists i,j \in \{1,\ldots,N\}, \text{ such that } rank(i) = head(p) \text{ and } rank(j) = tail(p) \\ 1 & \quad \text{and } \left| |pos(i) - pos(j)| - dist(p) - \frac{l(head(p)) + l(tail(p))}{2} \right| \approx 0 \\ 0 & otherwise \end{cases} \tag{2}$$

And if $or\_match : P \to \{0,1\}$ is a function that for each read pair p provides whether the orientation of any contig occurrences of the contigs of p, according to *or*, is contradicting to the one suggested by p or not. It has a value of 0 if the orientation is contradicting for all of them and 1 otherwise.

$$or\_match(p) : \begin{cases} & \text{if } \exists i,j \in \{1,\ldots,N\}, \text{ such that } rank(i) = head(p) \text{ and } rank(j) = tail(p) \\ 1 & \quad \text{and } (or(i), or(j)) = orient(p) \text{ or } (\overline{or(j)}, \overline{or(i)}) = orient(p) \\ 0 & otherwise \end{cases} \tag{3}$$

Then the following conditions have to be true for the functions *rank*, *pos* and *or*.

- $\forall c \in \Gamma : \left| \left\{ j \in \{1,\ldots,N\} : rank(j) = c \right\} \right| \leq cov(c)$
- Find the *pos* and *or* functions which maximize the following function:

$$\sum_{p \in P} dist\_match(p) \times or\_match(p) \tag{4}$$

The scaffolding problem can be explained then as finding the best positions and orientations of the contigs occurrences, so that a maximal number of mated read-pairs are satisfied. Depending on the choice of *dist_match* function one can have different models for the scaffolding problem that can be solved as optimization problems.

4 Proposed models

In this section we describe four different approaches to the scaffolding problem. They are aiming to provide robust way for scaffolding of the genomes of chloroplast organelles and mitochondria organelles. In this special case of the scaffolding problem, it is realistic to expect that the result will be just one scaffold and not a number of scaffolds, as the coverage that the sequencing machines provide is big enough. Despite that, this problem is not sufficiently well studied. All the approaches that are targeting this part of the scaffolding problem use heuristics. Furthermore, at least most of them do not use the coverage information that is provided by the contig assemblers. Another feature that our approaches aim to improve is the number of possible solutions. In practice it is known that several distinct solutions are possible for the scaffoldings of the genomes of those organelles. All the existing approaches, to the best of our knowledge, aim to provide just one

optimal solution. We, on the other hand, provide all the optimal ones, as that will be useful for the biologies.

For validating the solutions that we provide, we use examples, for which the answers are known. That way we can verify that some of our solutions are the same as the provided one. All the other optimal solutions are equally possible ones, according to our formulation of the problem. In regards of the provided data any of the optimal solution has the same probability to be correct as the provided one.

The examples are created, using data that is provided to GenScale by some of its partners. The data consists of reads, mated read-pairs and the expected final result. Using that raw data, we first filter it[8]. After that, the reads are combined into contigs, using a tool created by GenScale. Next, we map the mated read-pairs information onto the contigs. After that, using the expected final genome, we create the expected solution in terms of contigs positions and orientations. That is the data our approaches are using. It would be relatively easy to add support for **fasta** and **fastq** files to our methods, but this is still not done.

We have provided implementations for all of the approaches that we will describe. They have been designed with the idea of being compared, thus we have being trying to give them the same scoring function. For some of them we use slightly modified scoring function during the construction of the solution, as otherwise we would have to also modify the models. Still, we have implemented ways to score all the methods using exactly the same function, which provides us with some possibility to compare the performance of those different approaches. Those comparisons can be seen in Table 4.

Three of the models that we are going to describe are exact. That means they are trying to find the best possible solution by the criteria that we have set. It is possible to limit the time of search for some of them, which might result in finding suboptimal solution. Some of the approaches have also the possibility to provide all the best solution and not just one of them. We believe that this will be very beneficial for the biologists.

One of the approaches is heuristic, which means that not only we can not guarantee that it always finds the best solution, but we can not find all the solutions that are believed to be optimal, using it. It will be described, nevertheless, as it can also be used together with some of the other approaches in order to improve their performance. This is done as the heuristic results are used to provide information about the expected score of the optimal solution as well as some information about its structure. It is then easy to use that information for the Branch and Prune approach. When the size of the problem is big enough the heuristic approach runs considerably faster than the other approaches. That is why it is rational to use it to try to improve the runtime of the other methods, by providing them with some information about some at least locally best solution.

Before we present the heuristic approach in section 4.5, we will first describe the graph representation that we chose in section 4.1. After that we will give information about possible ways to decrease the size of the problem that we are trying to solve in section 4.2. Then a Linear Programming model will be described in section 4.3. Next a description of a Branch and Prune model will be provided in section 4.4. After that, the heuristic Genetic Algorithm model will be presented in section 4.5. Finally the Distance Based Approach will be described in section 4.6.

[8]This can be done with existing tools like those of Kelley et al. [2010] or Sasson and Michael [2010].

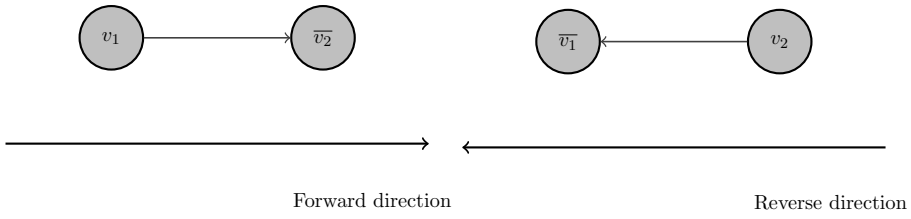

Figure 2: On the left part of the figure is presented the read pair $(v_1, v_2) \rightarrow (F, R)$ with forward direction from left to right. On the right part of the figure the same read pair can be seen, but looked in the opposite direction and as a consequence the orientations of the contigs are reversed. It is equivalent to having a read pair $(v_2, v_1) \rightarrow (F, R)$.

4.1 Graph representation

The graph representation that was chosen is used in almost all the models. We aim to stick with the same representation, so that the descriptions are easier to understand and the comparisons are fare. Here is a description of the representation itself.

- Nodes are built using the contigs and the expected number of times they should be met in the genome. For each time a contig is expected to be presented in the genome, two nodes are created - one for each of the possible orientations of the contig. We assume without loss of generality that one of them represents the reading of the contig in forward direction and the other - in reverse direction. The direction will be important when the read pair information is added to the graph.
- Directed edges are built to connect the nodes that represent contigs that are connected with read pairs. In order two contigs to be connected, their orientations should be consistent with the orientation suggested by the read pair. In other words if we have a read pair that connects $(c_1, \overline{c_2})$, the nodes for c_1 and $\overline{c_2}$ will be connected with a directed edge, but for example c_1 and c_2 will not be connected.
- Directed edges are built for all read pairs when they are looked in reverse direction, i.e. swapping forward and reverse in the orientation of the contigs and swapping the places of the contig numbers - the first becomes the second and vice versa (see Figure 2).

To make things clear, let us consider a simple example. Lets say that we have three contigs $\rightarrow \{c_1, c_2, c_3\}$. For c_1 we have two occurrences in the genome, while for the other contigs we have only one. For those contigs we have the following mated read-pairs: $(c_1, c_2) \rightarrow (F, F), (c_1, c_2) \rightarrow (F, R), (c_1, c_3) \rightarrow (F, F), (c_2, c_3) \rightarrow (R, F)$. Here we have omitted the information about the distance between the contigs in order to simplify the graph. Otherwise it will be presented on the edges. The resulting graph can be see in Figure 3. There each contig is represented as a few vertices - two for each occurrence of the contig. The notation is as follows: the nodes for each contig i are called $v_{i[,k]}$. The second index, if present, means the occurrence of the contig. For example the second occurrence of c_1 is represented as $v_{1,2}$. Throughout this document to signify a reverse orientation of a contig occurrences, we use *overline*. For example $\overline{v_{1,2}}$ denotes the reverse orientation of the contig 1 in its second occurrence. The absence of *overline* signifies forward orientation.

To sum up, the nodes are $\{v_{1,1}, \overline{v_{1,1}}, v_{1,2}, \overline{v_{1,2}}, v_2, \overline{v_2}, v_3, \overline{v_3}\}$ and the edges are

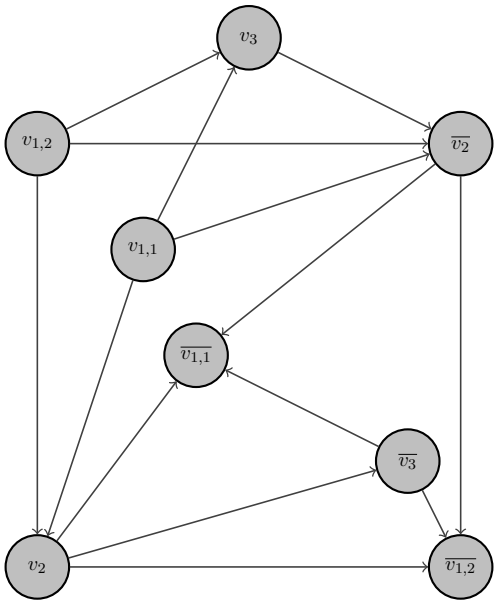

Figure 3: Example graph

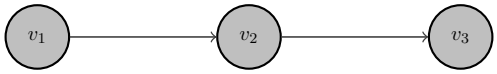

Figure 4: A node, v_2, with exactly one predecessor and exactly one successor.

$$\{(v_{1,1}, v_2), (\overline{v_2}, \overline{v_{1,1}}),$$
$$(v_{1,1}, \overline{v_2}), (v_2, \overline{v_{1,1}}),$$
$$(v_{1,1}, v_3), (\overline{v_3}, \overline{v_{1,1}}),$$
$$(v_{1,2}, v_2), (\overline{v_2}, \overline{v_{1,2}}),$$
$$(v_{1,2}, \overline{v_2}), (v_2, \overline{v_{1,2}}),$$
$$(v_{1,2}, v_3), (\overline{v_3}, \overline{v_{1,2}}),$$
$$(v_2, \overline{v_3}), (v_3, \overline{v_2})\}$$

4.2 Graph reformatting

In this section some techniques to decrease the size of the graph in order to improve the performance of the scaffolder are presented. They do not change the nature of the problem that is being solved as well as they do not change the solutions of the problem. Similar techniques are used in a number of previous works, including Dayarian et al. [2010]; Salmela et al. [2011], but in those cases the covering of the contigs have not been taken into account to the best of our knowledge. The addition of the covering slightly changes the problem.

The first optimization is to detect the cases where one contig has exactly one predecessor and one successor (see Figure 4). In that case if the coverage for the three contigs are $cov(v_1), cov(v_2), cov(v_3)$

Figure 5: A node, v_2, with exactly one predecessor and exactly one successor.

and without loss of generality let us assume that $cov(v_1) >= cov(v_3)$, and if $cov(v_2) = cov(v_3)$, then we can delete the v_2 contig and the v_3 contig as well as its two edges on the figure and substitute them with a new artificial contig and a new edge between v_1 and that artificial contig (see Figure 5).

The rational behind this is that we want to use every contig the number of times it is indicated by the data, if that is possible. By making this substitution we do not change whether it is possible or not to use the contigs the required number of times. To illustrate that let us consider the following scenario: in order to go to the v_2 contig we should always visit v_3 and each time we visit v_3, we should visit v_2, as otherwise we will be unable to visit v_2 the required number of times. Then it is clear that we can merge the two contigs into one artificial contig, for which some of the bases that form it are unknown (this is not allowed for the *normal* contigs).

Another technique, that can be applied, is the **graph partitioning**. It is already studied in a number of other articles (see Dayarian et al. [2010]; Salmela et al. [2011]). The idea that is used in those works is to find articulation points (see Westbrook and Tarjan [1992] for ideas how to do it efficiently) of the graph and partition the graph based on those articulation points. After that for every partition the problem should be solved independently and the results should be recombined. This approach is applicable if the structure of the genome is linear and not circular. Then the orientation and ordering of each of the contigs in one component depends only on the orientation of the articulation vertex. To better illustrate this, we will describe an example.

There are five contigs each of which is met only once. That forms ten vertices:

$$\{v_1, \overline{v_1}, v_2, \overline{v_2}, v_3, \overline{v_3}, v_4, \overline{v_4}, v_5, \overline{v_5}\}$$

The mated read-pairs between them are: $\{(v_1, \overline{v_2}), (v_1, v_3), (\overline{v_2}, v_3), (\overline{v_3}, v_4), (\overline{v_5}, \overline{v_3}), (v_4, \overline{v_5})\}$.

A visualization of this example can be seen in Figure 6A, where instead of writing $v_1, \overline{v_1}$ we have used c_1 in order to simplify the figure. In that figure it can be seen that the graph can be partitioned into two parts with respect to c_3. In Figure 6B the solution for the first partition is provided. In Figure 6C a solution for the second partition is given. It can be seen that although the orientation of contig c_3 for those two solutions is different, they can still be combined to form the global solution as shown in Figure 6D. This is done using the reverse of the original solution for the second part. As it was already described, each read pair can be used as if it is looked from left to right or as if ti is looked from right to left. That is what is used here.

The problem that we are focused on, however, is for ordering contigs of chloroplast genome. It differs from the case on that picture by shape of its genome - it is not linear, but circular. As a consequence it is not true that if we remove any one vertex in the graph, the graph will no longer be connected. As a result there will be no parts that can be solved separately. Furthermore, even if we try to remove two vertices, we will be unable to combine the solutions for the two parts. That is the case as they might contain contradicting information about the orientation of some contig as they have not one, but two vertices in common. For that reason the technique with taking the reverse of every contig occurrence in the subgraph is inapplicable.

Another possible technique to reduce the size of the graph and as a result to improve the performance of the algorithms is presented in Gao et al. [2011]; Salmela et al. [2011]. The idea is

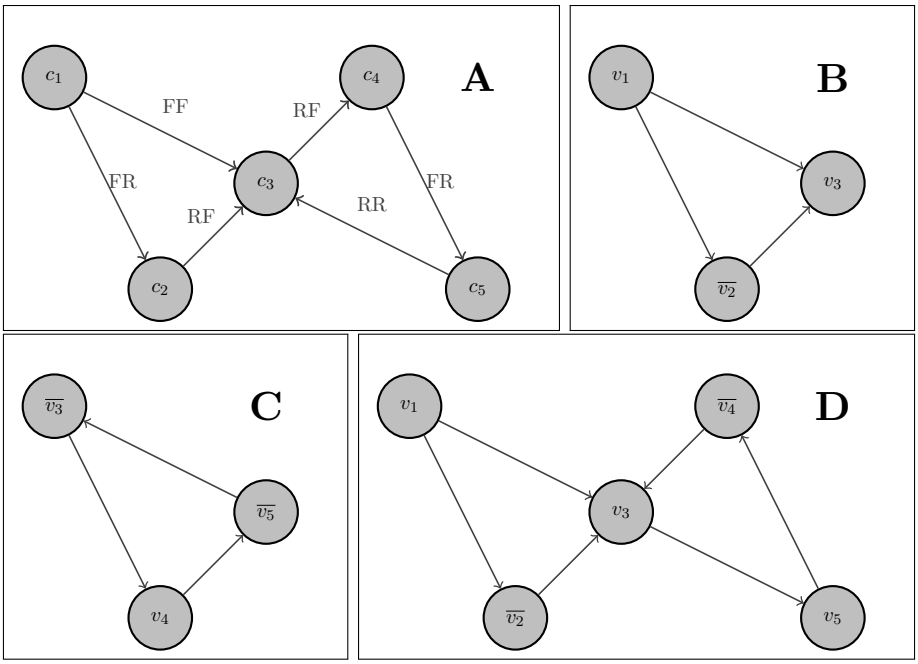

Figure 6: In figure A the structure of the graph is given. After that it is divided at the vertex v_3. The solutions for each of the sub problems are presented in B and C respectively. For those two solutions the contig c_3 has different orientations. This is solved in figure D where the reverse solution of the one in C is used.

to put similar mated read-pairs as a single edge. In Salmela et al. [2011] all the mated read-pairs that suggest the same orientation between a pair of contigs are put together. As a result the size of the graph is significantly reduced. The drawback of this realization of the provided idea is that the mated read-pairs might have significantly different lengths. Then it is not easy to decide how to resolve this. For that reason in Gao et al. [2011] the authors propose only mated read-pairs with similar lengths to be put together.

Another thing to notice about this approach is that the *support*, or the number of times a mated read-pair was presented in the input data, should be preserved. Otherwise incorrect solutions might be produced.

4.3 Maximum Weighted path approach

In this section we are going to present our first method for solving the scaffolding problem. It is somewhat different from the others in that it does not use the distances between the contigs. Despite the fact that it does not use that very important piece of information, this method can produce very nice results, especially in combination with some parts of the other methods.

The objective here is to place linearly the contigs in a way that maximizes the number of read pairs whose orientation corroborates the orientation of the contigs that are associated with them. This is not as strong requirement as when the distances are also used. As a result this approach might produce a number of suboptimal solutions according to the optimization function used for the other approaches. Despite that, it is relatively easy to filter the suboptimal solutions afterwards, at least for the experiments that we have made.

The main motivation of the model is to be simple and to be able to find contradictions in the input data (i.e. to detect that no linear allocation of contigs exist that satisfies the given read pairs' orientations and coverage). When the data is not controversial it tends to find all the optimal solutions very quickly.

Method description

This approach can be seen as a filling of a table M of size $[2 \times N, N]$. To each contig from the input we associate two rows in the table - one row for each orientation of the contig. The column i corresponds to the i^{th} contigs in the solution and can be seen as the i^{th} step in the graph walk. The problem then is to find a path in the graph, such that the below set of constraints are satisfied:

- Each node is visited at most once, meaning that the sum for each row in the table is at most one[9].
- Each of the nodes have just one orientation.
- The orientations of the nodes does not contradict the constraints imposed by the read pairs. This is at least partially enforced by the construction of the graph.
- On each step exactly one contig is visited.

The optimal walk is such that maximizes the read pairs, for which the contigs that they connect, are having the orientation that is suggested by the read pair.

In other words, the sum in each column k ($\sum_{i=1}^{2N} M_{i,k}$) and in each two row that represent the same contig occurrence $j, j+1$ ($\sum_{i=1}^{N} M_{j,i} + M_{j+1,i}$) should be equal to one, meaning that at each step we take, we use exactly one contig occurrence and that this contig occurrence is not used in any of the previous steps.

We have also added the constraints that in order $M_{i,j}$ to be 1, some of the *neighbours* of that node should have a value of 1 on the previous step. This can be implemented as follows: $\forall k, l : M_{l,k} \leq \sum_{w \in neighbours(l)} M_{w,k+1}$. As we know the sum can be either 0 or 1. This statement says that if $M_{l,k}$ was 1 on the previous step, then some of its neighbours should also have value 1 on the next step.

In order to keep track of the mated read-pairs that are satisfied, we introduce additional variables f_i that represent whether some node is used. That information is used after that to verify that the orientations of both contigs of the read pair are the expected ones. If this is the case another variable has value of 1, otherwise its value is 0. The sum of those variables is used as function that we need to maximize.

Mathematically the above constraints can be written as:

$$\text{for each column } k \in \{1, \ldots, N\} : \sum_{i=1}^{2N} M_{i,k} = 1 \tag{5}$$

$$\text{for each row } i \in \{1, \ldots, 2N\} : f_i = \sum_{k=1}^{N} M_{i,k} \leq 1 \tag{6}$$

[9]There might be an exception for the first node in the walk as chloroplast cells often have circular genome.

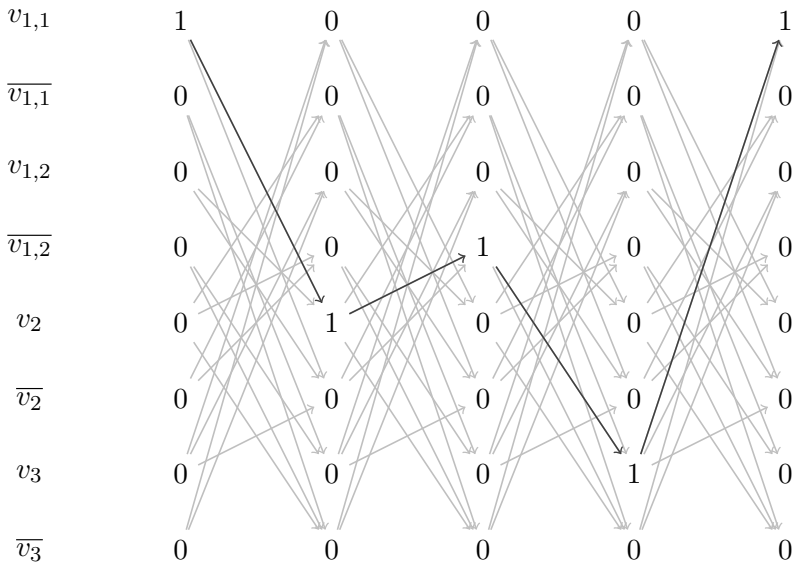

Figure 7: A walk in the example graph as performed by the algorithm

$$\forall i \in \{1, 3 \ldots, 2N-1\} : \; f_i + f_{i+1} = 1 \tag{7}$$

$$\forall k \in \{1, \ldots, N\}, l \in \{1, \ldots, 2N\} : \; M_{l,k} \leq \sum_{w \in neighbours(l)} M_{w,k+1}. \tag{8}$$

$$\text{for each read pair } (i,j) : \; f_i + f_j \leq 1 + y_{i,j} \tag{9}$$

$$\max \sum_{p \in P} y_p \tag{10}$$

In Figure 7 it is possible to see the way the Maximum Weighted path algorithm works. The example that is considered is as follows. The contigs in it are $\{c_{1,1}, c_{1,2}, c_2, c_3\}$. For them the following read pairs are available: $(v_{1,1}, v_2), (\overline{v_{1,1}}, v_3), (v_{1,1,}, \overline{v_2}), (v_{1,2}, v_2), (\overline{v_{1,2}}, v_3), (v_{1,2,}, \overline{v_2}), (v_2, \overline{v_3}), (v_3, v_{1,1}), (v_3, v_{1,2})$.

The black arrows show the actual path that the algorithm found and the light gray ones show the other edges of the graph, that were not used by the algorithm.

As it can be seen, the node v_1 is used three times, even though it has *coverage* of 2. This is done in this way as most of the genomes of the chloroplast are cyclic and as a consequence the first and the last contig in the walk should coincide.

Implementation details

In practice, if some information is missing or just incorrect, it is possible that there will be no path that goes through all the contigs. If that is the case, the approach that we described above will be unable to find any solution as it expects the data to be correct. This expectation allows it to run very quickly. For the biologists, however, this is not always very useful, as the data that they have might contain errors. If the algorithm does not produce any solution the biologists will have no

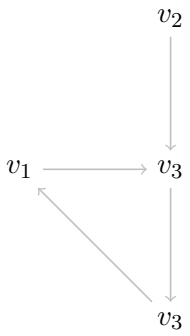

Figure 8: An example where one of the contigs can not be part of the path. In this situation, although it might be possible to position all the contigs, the algorithm detects a possible problem: a missing information about the distance between v_1 and v_2. It will substitute the contig v_2 with the artificial contig.

idea what is the problem. In order to improve that, we introduce an artificial contig. It is special in the following ways. First of all, it is connected with all other contigs. This is to solve the cases similar to the one in Figure 8. In this figure that represents a simplified version of the available information. It is impossible to find a path that goes through all the contigs for this example. For that reason the artificial contig will take the place of contig v_2. As it is connected to all the other contigs, it will be possible to create a path like $v_1, < artificial\ contig >, v_3, v_4, v_1$.

Another thing that is specific for this contig is that it can be used any number of times. In order to restrict the number of uses of this contig, the objective function has a term that penalize the use by some constant times the number of uses of the artificial contig.

$$\max \sum_{p \in P} l_p y_p \tag{11}$$

where $l_p = 1$ if p is one of the provided read pairs and $l_p = -10000$ if p is one of the artificially added pairs.

As a result, this artificial contig is used the minimum possible number of times. When it is used, it replaces a contig occurrence. This provides clues to the biologists as of which part of the genome might contain errors or missing information. It also enables our algorithm to finish successfully.

4.4 Branch and Prune

4.4.1 Overview of Branch and Prune algorithms

In the literature Branch and Prune is a general algorithmic technique for finding optimal solutions for a vast class of optimization problems (see Van Hentenryck et al. [1997]). The idea of the algorithmic technique is to enumerate all the possible solutions of the problem and while doing so to prune some subsets of candidate solutions based on evidence for sub optimality. While the algorithm enumerates the possible solutions, it build a search tree. In that search tree the nodes represent variables and every path from the root to some leaf node represents a possible solution.

The Branch and Prune approach is one of the most natural ones, even though in general it is less efficient than some other approaches. The lack of good efficiency is a result of the way the algorithm

operates. It does not aim to use some specific properties of the problem that would allow it to consider just a small portion of the possible solutions in the general case. That is why in a number of situations there are more efficient algorithms. In a variety of other situations, however, it is not that inefficient. For example, if there are very good pruning conditions that eliminate sufficiently large amount of the solutions, the speed can improve dramatically. In theory, it might be possible for some problems to prune all the suboptimal solutions when you start to build them. This will result in a very efficient algorithm. If there are no strong pruning conditions, this approach tends to be less efficient than some of the other possible ones.

The branch and Prune approach have a similar philosophy to that of the Branch and Bound approach, however, for the Branch and Prune one there is no good way to bound the score of a class of solutions. For that reason it is not always easy to limit the number of explored solutions.

One of the good properties of that approach is that a way to find all the optimal solutions can be easily implemented. It does not require significant additional work both in terms of implementation effort and in terms of computational time. Another one is that it is guaranteed to find the optimal solution for the problem it is exploring.

4.4.2 Our implementation of a Branch and Prune Algorithm

Our implementation of the Branch and Prune algorithmic technique is as follows. We build a path through the graph of consecutive nodes. The idea is to find a path from one node back to itself, that goes to all the other nodes. The last point we made was not entirely correct, as for each contig occurrence we have two nodes in the graph. During the building of the path only one of every such two nodes can and should be visited. See Figure 9 for an example of partial search tree that the Branch and Prune algorithm explores.

As there might be a large number of paths in the graph and some of them will not be able to produce optimal solution, we apply methods to detect as much of those situations, as possible and we prune the search tree accordingly.

For that purpose we use two techniques to detect when a partial solution will not lead to a real optimal solution. The first technique that we use aims to detect if from the current partial solution it will be possible to visit all the contig occurrences that are still unvisited. We also verify that it will be possible to go to the first node in the path from the current configuration as the structure of the genome is supposed to be circular.

The other pruning criterion is the score of the solution. It is possible to find a limit of a score of a partial solution, using the fact that if all the contig occurrences are already positioned (they are part of the path), then it is possible to verify if the read pairs that they participate in are satisfied, given that the other contig in that read pair is also already positioned. As a result the score can only get worse with the addition of new nodes to the path as additional read pairs are verified and they might not be satisfied. In this manner it is possible to decide that some parts of the search tree can not lead to a solution that is at least as good as the current best one. Consequentially, they can safely be pruned.

That process explains what can be done to improve the runtime of the algorithm. It does not, however, provide details about the time of execution of each of those verifications. As they are not very fast ($O(n' + m)$ for the reachability check, where n' is the number of remaining vertices and m is the number of edges). It is then not very hard to notice that it is not practical to run the verifications on each step as that will make the algorithm slower. For that reason we have decided to do the following things. We run the verifications every v step, where v is a parameter

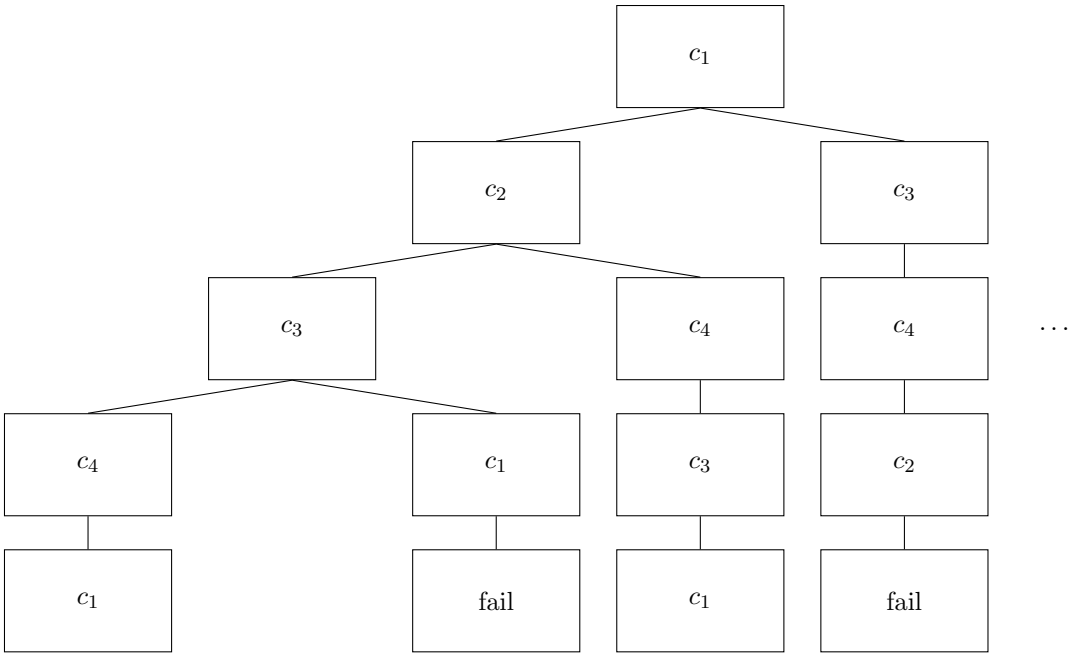

Figure 9: An example of part of the search tree that the Branch and Prune algorithm will explore for a graph with contigs c_1, c_2, c_3, c_4 and read pairs $(c_1, c_2), (c_2, c_3), (c_1, c_3)$, $(c_3, c_4), (c_4, c_1), (c_4, c_2), (c_4, c_3)$. We have skipped the orientation and the distances of the read pairs to simplify the example

of the program. This does not seem very optimal as well, as it might lead to a big number of branches in which we descend and run the verifications, despite the fact that there has been a step that guarantees sub optimality, which has been taken right after the last verification. In order to improve this, we also have dynamic testing. It means that under some circumstances, we decide that it is necessary to run the verifications and we do it. If those circumstances are not present, we do not run the verifications.

The circumstances that were mentioned in the previous paragraph are as follows. If while the algorithm explores a branch, it goes to a configuration where it is not possible to add more nodes to the path or the reachability condition in not satisfied, then the reachability condition is marked as broken and that information is propagated to the previous step of the algorithm. In that case a reachability verification is run for the parent of the current configuration. This is done in order to ensure that all the branches of the search tree, that start from the parent configuration, have chances to produce a solution. In this situation we don't know if the problem with the reachability was introduced with the last "step", or, for example, with the step that lead to it.

Something similar can be applied for the score as well. If at some point the verification or the score of a possible solution suggest that the score of the current path can not be optimal, some of the nodes along the way are verified. The strategy for the verifications is the same as the one described in the previous paragraph.

So far we have described how those checks propagate backwards. In other words how the information that there might be a need of a check is provided to the parent of some partial solution. But sometimes there are situations in which it is clear that such checks are not necessary. For

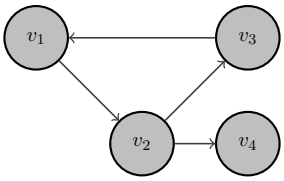

Figure 10: In we are at v_2, we can not know that there is no path through v_4. It is afterwards, when we are at v_3 or v_4 that we can notice that.

example, if some of the branches of the current partial solution leads to a solution, it is clear that it is not necessary to check if it can lead to an optimal score. The same strategy can be used for the reachability tests as well.

Also, if some of the children of the current configuration performed some of the checks and the result was not pruning, then the parent does not need to perform that check again.

Despite the fact that the pruning logic in our implementation can eliminate the need to consider a vast number of candidate solutions, it seems that there is more that can be done. This is particularly true when we consider if all the unvisited vertices in some partial solution can be visited in the path. Currently we only verify that each of those vertices is reachable from the current one. This, however, do not guarantee that there will be a path that will be able to visit all of them. As it can be seen in Figure 10, it is impossible to find a path through v_4 that goes to all the other nodes as well. Our verification can not detect that. It will claim that it is possible to construct a solution. This leads us to the question whether it is possible to do something better. The graph can be simplified so that it contains only the vertices that are not visited and the last visited vertex. The task now is to find a path in the graph that goes exactly once through the vertices of the graph, staring from a particular node in the graph and ending in another particular one. This is a special case of Hamilton Path Problem. This problem is well studied and it is proven to be NP-complete not only in the general case, but also for some more specific cases like directed planar graphs with indegree and outdegree at most two (see Garey et al. [1974] and Plesnik [1979]).

The next question that comes to mind is whether we can use some heuristics. There are some very good heuristics that run in $O(n^3)$ time. They tend to find a Hamilton path in the graph most of the time when there is one. There are, however, cases when there exists such Hamilton Path in the graph, but the heuristics can not find it. In order to be usable for our algorithm, the heuristic should only make errors of type 1 - false positive. In other words it can say there is a Hamilton Path when there is none, but it can not say that there is no Hamilton Path when there is one. Unfortunately such heuristic can not exist, because in order to say that there is a Hamilton Path, it should find one such path. Then it would be possible in polynomial time to verify whether the provided path has the required properties. If this is not the case, the result of the invocation of the algorithm will be corrected. As a result all the errors that this hypothetical algorithm will make can be corrected in polynomial time. Its runtime is also polynomial, which means that this algorithm can be modified to find a Hamilton Path in polynomial time. This is impossible, and consequentially such algorithm does not exist. As a result there is no obvious way to improve this part of the pruning in our implementation of the Branch And Prune Algorithm.

In order to minimize the calculation during each of the verifications, a number of caches are used, so that no calculation is performed multiple times. According to our experiments this as well as the use of a custom linked list implementation greatly improves the runtime.

Another technique that we use in order to improve the runtime of that algorithm is to run a number of threads. The number of threads that is to be used is left to the user to choose. The best results are expected to be achieved when using number of threads close to two times the number of processing cores on the CPU (for CPUs with hyper-threading).

The work is distributed to the threads in the following way. First all the threads receive a part of the search tree to explore. After that each thread can receive additional subtrees to explore, if it finishes exploring the one it is assigned. That way the effectiveness is around 350% for 4 threads, which we consider a good improvement.

As it was already noted, the Branch and Prune method has a number of good properties. It, however, do not work for all possible graph configurations. It is possible that some information about distance between pairs of contigs is missing. If the missing information is between contigs that have to be put one next to the other in the final solution, this method will fail. For that reason we add artificial contigs. They have a length of zero and their only purpose is to provide connection between real contigs. As it is not known in the beginning where those connections will be needed, the artificial contigs are connected to every other contig. Apart from those differences from the real contigs, the artificial contigs are almost the same as the others. They can be used only once, etc.

As we don't want them to influence the score of the solutions, the links between them and the other contigs do not have a fixed length. The length is determined when the length of the read pairs to the neighboring contigs are considered. It is the most probable one. In other words, the one that is most often met, if there is more than one possible. We use that value to calculate the number of violated read pairs afterwards.

In the Maximum Weighted path approach there are again artificial contigs, but they are not exactly the same as those presented here. The difference is that there they replace the use of a read contig, whereas here this is not the case. Another difference is that here the number of artificial contigs should be explicitly given, for example by the user, whereas in the Maximum Weighted path approach this is not necessary.

4.5 Genetic algorithm

4.5.1 Overview of genetic algorithms

The idea of trying to simulate evolution of *individuals* that describe some problem in order to find the best solution dates from the 1950s, but it gained more popularity during the late 1960s and early 1970s as an optimization technique. One of the important materials on this topic was Fogel et al. [1966]. That was the first book that tried to study in a great depth the evolutionary approach to artificial intelligence.

In the 1970s the work of John Holland greatly contributed toward the popularization of the genetic algorithms (see Holland [1975]) as reliable optimization techniques.

As a consequence of the research, a number of different variations of genetic algorithms are now present. They all have a number of things in common. First of all, the possible solutions of the problem are presented as individuals of some population. The problem then is to find the *fittest* of those individuals, where the *fitness* of an individual is determined by a function.

Quite often such approaches are used for optimization problems and in those cases the fitness is related to the optimization problem. Let us take an example to illustrate that. In our example we aim to minimize the expenses in a company, where a set of possible strategies can be applied

to achieve that. Any number of strategies can be simultaneously used, but sometimes some pairs of strategies influence one another and if they are used together they perform worse compared to the case when any single one of them is used. In that example each individuals represent one of all the possible subsets of the set of possible strategies. As the number of individuals is limited, it is possible that not all subsets will be presented. It is also acceptable to have individuals that represent the same subset, although it is not always advisable, as it is not expected to improve the final result.

In the example above it is possible to encode each individual using words from a binary alphabet, where each position of the word encodes whether some strategy is in the subset of strategies that this individual represents or not. For example, if the word is 01101, this will mean that the first strategy is not used by this individual, the second and the third ones are used. The forth is not used and the fifth is used.

It is often very important to choose a good representation of the individuals, so that it is easy to perform operations on them afterwards. The selection of the best individual representation is often one of the hardest problems when constructing a genetic algorithm.

The first step in any genetic algorithm is to **generate a random initial population**. If the representation of the individuals allows it, it is possible to just put random letters from the alphabet that is used to encode the individuals, at each position of the new individuals. Sometimes this is impossible as individuals need to have some special properties in their representation and in those cases a more complex random generation of individuals is needed.

After the initial population is generated, the algorithm starts to iterate. On each iteration the following steps are performed. First we copy the best few individuals from the current generation into the next one. This step might be omitted, but it is considered useful in order not to loose the best individuals from the current population. The name of this part of the genetic algorithm is **elitism**.

After that step a step named **crossover** is performed. This is a process by which from two individuals from the current generation, one or two individuals are produced for the next generation. The way the individuals from the current generation are chosen also varies, but it is always connected to the fitness of the individuals. The better individual, the higher chances it has to reproduce some of its *genes* in the next population. The way the *genes* of the *parents* are combined to produce the next generation can also vary. In some implementations the individuals are words with a fixed length in a finite alphabet and the crossover is performed in the following way. The genes of one of the parent are taken from the beginning of the word up until some random position. After that the genes of the second parent are taken. That way the offspring has some of the genes of both its parents. For example if one of the parents is **11000011** and the other is *00001111* and the algorithm just takes the first half of the genome of the first parent and the second half of the genome of the second parent and puts them together, the result will be **1100***1111*.

Sometimes one additional step is performed. It is called **mutation**. Its purpose is to make sure that there is some chance a local optimum, which is not a global one, to be avoided. That way the global optimum will eventually be reached. In essence, it chooses a random set of individuals and changes some of their genes in a random fashion. For example if the alphabet is binary, the mutation of an individual will select one or a number of random locations in its genome and it will flip the value at the selected position or positions. In other words, it will change any 0 to 1 and the other way around. It is also possible that instead of just flipping the value at that position, a random value is assigned. Both are equivalent if the probabilities for mutation of an individual

and having 0 or 1 are set accordingly.

The process of creating new generations is repeated either until some optimal value is reached, or a fixed number of iterations. Some people would also like to start a number of runs of the whole algorithm - initialization and repeating crossover and mutation, in order to ensure that the there will be some good initial population. The rational is that if all the initial individuals have a low score, it will be difficult to have a nice solution in the end.

Although this approach might seem quite random, it in essence does something that is not that random. It assumes that the optimal solution is not isolated and that around it the other solutions are also good ones. As a result the process of giving higher chances of crossover of good solutions seems very natural. As other individuals also have chances of participating in the crossover, this approach might avoid some local optimums. Another thing that helps to avoid local optimums is the mutation which, being completely random, allows this approach to move away from any local optimum and go to the global optimum. Thus, solutions that are more likely to contain important features for the fitness function are more likely to recombine and to give those nice properties that they have to the next generation.

This technique might be seen as searching with a hyper-plane in the space of the possible solutions. It is expected that in that space around the best solution there is a number of other solutions that also have a nice score.

Common problems for genetic algorithms are the representation of the solutions as individuals, efficient fitness, crossover and mutation functions. Another common problem is the situation where all the individuals become very similar, but their fitness function is far from being optimal. Although the mutation tries to overcome this last problem, sometimes this might greatly affect the effectiveness of the algorithm.

Despite the fact that this technique often produces very nice results, if the search space does not have good properties, it is possible to produce results that are very far from the globally optimal ones. For that reason a number of people prefer to use it in combination with other approaches that mitigate that problem.

4.5.2 Our implementation of a genetic algorithm

As it has been noted in the previous section, one of the most important and nontrivial parts when designing a genetic algorithm is to choose a good representation of the individuals. We considered a number of possible representations, the most important two of which are:

- Associate consecutive natural numbers to each of the contig occurrences and then encode each individual as a permutation of those numbers. That gives you the order in which the contigs should be met in the final ordering and thus a possible solution.
- A set of edges of the graph that when used position all the contig occurrences unambiguously.

Although the first representation have some benefits, for example, the operations for initialization, mutation and mating are almost trivial, it also has some disadvantages. One of them is that it expects the graph to be fully connected. Another one is that the solutions are very hard to evaluate when this representation is used. The reason for that is the ambiguity of the used edges. The end vertices do not strictly identify the edges that need to be used. In order to evaluate the solutions, however, the specific set of edges that connect the vertices has to be known. As a result all the possible sets of edges should be constructed and a score should be calculated for all of them. After that the best score is to be selected. We expect this to take a considerable amount of time. That is one of the reason to prefer the other approach.

When we are using the other representation, it is easier to evaluate the individuals and there is no problem if the graph is not complete. It, however, needs an nontrivial initialization, mutation and mating procedures. Due to the problems with the first representation, which we considered as very hard to solve, the second one was selected.

In order the set of edges for each individual to be a valid solution, i.e. to give information how to order all the contigs, it should form a spanning tree of the vertices in the graph[10]. We can verify the correctness of this by noting that if the edges form a spanning tree, they can be relatively positioned to one another unambiguously. If the edges, however, do not form a spanning tree, in the general case it will be impossible to order them unambiguously. Either there will be chance for contradiction if there is a loop in the set of edges, or the relative distance between some pairs of contig occurrences will be unknown, if the edges do to form a single connected component in the graph.

Note: Individuals can be viewed as a binary vectors that represent which edges are included and which are not. The representation with explicitly referring to the edges seems more natural, however, and that is why in the explanations we will use it.

There have been a number of studies as of whether it is better to have a complex alphabet to represent the individuals of a genetic algorithm, or a binary alphabet is the best choice. So far the results are inconclusive (see Whitley [1994]).

Initialization During the initialization extra care should be taken so that the initial population contain sets of edges that form spanning trees for the vertices. This is done using a technique that selects random edges and tries to add them to the existing configuration. If the edge can not be added as for example it will form a cycle in the configuration, it is skipped. In such a manner a random set of edges, that is a subset of all the edges, is selected and it is used for the initialization of one individual.

In order to construct the new individuals quickly, we use a technique similar to that of Kruskal algorithm (see Eisner [1997]). The idea is to shuffle the edges randomly and then to start to try to use each edge in turn, in order to combine subtrees of vertices. Some attention should be taken so that no vertices that represent the same contig occurrence are added. It might be necessary, however, to reverse an entire subtree, so that it can be added to another subtree. If, for example, the only way to connect two subtrees is through some edge, but the orientations of the end nodes of the edge are such that it is not possible to directly use that edge, one of the end nodes of the edge should be oriented in the reverse direction. In order to do that, the whole subtree that it belongs to, should be reversed.

The situation where the two end nodes of an edge are oriented in such a way that the edge can not be used is as follows. Let us suppose that the edge is between contig occurrences c_i and c_j. The read pair is $(c_i, \overline{c_j}) \rightarrow (F, R)$. In that situation if in the subtrees both c_i and c_j have *Forward* orientation, then it is not possible to use that edge. If we rotate the entire subtree of either c_i or c_j, then that will be possible. We use this technique to ensure that from the random subtrees, that we produce, it will always be possible to produce a spanning tree for the graph.

All the individuals in the initial population are created using the procedure that was described above.

[10]This is not entirely true. Not all the vertices should be covered by the spanning tree. Just those that represent different contig occurrences.

Scoring Each individual is assigned a score that is based on the ordering and orientation of the contigs. One of the components of the score is the number of read pairs that are compatible with the layout of the contigs. For a read pair to be compatible, it should suggest a compatible orientation of the contigs and the distance between the contigs should be the one suggested by the read pair. Naturally, a small difference is acceptable for the distance, as the data might not be absolutely precise.

Another criterion for the score of an individual is the presence of contig overlappings. If overlappings are presented, the score is not very good as it is not expected that any contig should share a considerable amount of its space with any other contig in the final solution.

The last criterion verifies if the structure of the scaffold is circular or not. If it is not, the score is adjusted to punish this.

Elitism As it was noted earlier, sometimes it is very useful to save some portion of the best individuals, so that they are not lost as a result of the mutation and the crossover. This process is named *Elitism*. This feature of out genetic algorithm can be controlled by a command line parameter that determines the percentage of the best individuals in the population that should be copied into the next generation without modifications. If this parameter is set to 0, then this feature is disabled.

On the other hand, if the value of this parameter is very high, this will prevent the algorithm from quickly converging to the optimal solution.

Crossover The basic idea of the crossover was already presented during the overview of the genetic algorithms. It is to produce new individual, i.e. new possible solution, that hopefully share some of the good properties of both its parents. The selection of the parents is biased in favor of the ones that have better fitness score.

The way that we have implemented this part of the genetic algorithm is as follows. First, the score of each of the individuals is calculated. After that, the scores are scaled so that the values are better distributed. Next, the mating pairs are selected. Each mating pair produces one new individual for the next generation.

The process of mating of individual was particularly complicated in this representation of the individuals. The problem is that every individual should represent a valid spanning tree of the contig graph. For that reason the mating procedure actually tries to substitute some of the edges of one individual, with edges from the other. To be more specific, first the edges that form the spanning tree of one of the individuals are copied into the new individual. After that, an edge from the second one is selected, such that it will form a cycle if added. There might be no such edge, but there might be a number of such edges as well. If there is more than one such edge, we randomly select one, we add it to the spanning tree and we remove a randomly chosen another one, so that there is no longer a cycle in the resulting graph. This process is repeated a random number of times.

Mutation The mutation procedure is somewhat similar to the crossover one, however, after the individuals for the mutation are selected, for each of them only one edge is substituted with an edge from all the possible ones. That possibly leads to reintroducing edges that have been missing in the population.

There is a number of options that can be given to the genetic algorithm. The user is presented with the possibility to set the path to the example that needs to be solved, the number of independent runs of the algorithm, the number of repetitions for each of the runs, the percentage of individual that should be transfered without change to the next generation and the percentage of individuals that need to be obtained through mutation.

4.6 Distance based optimization approach

In this section we are going to present yet another approach. The idea of this approach is the closest to the formulation of the problem that we provided, compared to the other approaches that we have already presented. It tries, using linear programming, to find the best positions of the contigs on an axis that goes from 0 to ∞.

There are real-valued variables that represent the two endpoints of each contig. We add a number of constraints and we want the solver to find the best values for those variables, so that the total error for each read-pairs is minimized.

The first set of constraints we have are for the distance between the two endpoints s of every contig c. It should be exactly equal to the length of that contig.

The other constraints that we use are essential for the correct calculation of the objective function. We have multiple variables for each read pair that connects multi-occurrence contigs. Per each combination of occurrences we have one variable. Those variables are used to measure the degree to which each read pair is satisfied. For example, if contig c_1 has three occurrences and contig c_2 has two occurrences and there is a read pair between them, we have $12 = cov(c_1) \times cov(c_2) \times 2$ variables. The $\times 2$ term comes from the fact that each read pair can be looked from the reverse direction as it was already noted.

After that we use boolean variables to filter all except one of those places where the read pair can be positioned. The objective function then is to minimize the sum of the variables that were left after the filtering.

Below we are going to present a more formal and detailed information about the implementation of this approach.

Implementation details

Based on the input data described in Section 3, we can consider that implicitly we construct an oriented graph $G(V, E)$ as follows. To every contig c we associate $cov(c)$ vertices in V and we introduce the set $rep(c) = \{1, \ldots, cov(c)\}$. All these vertices get the same list of neighbors. The list of neighbors is determined by the read pairs in P that the contig c participates in. Therefore, any read pair p in which the end contigs are (c', c'') generates $cov(c') \times cov(c'') \times 2$ edges in E. Note that the orientation that is suggested by the read pair is not enforced in that graph. As a result extra care needs to be taken so that the contigs are oriented in accordance with the read pair.

Let us now present some notations that will enable us to better mathematically express the problem. Let for every $e \in E$ the binary functions $oHead(e) \in \{0, 1\}$ (respectively $oTail(e) \in \{0, 1\}$) provide the orientation of the first (respectively the second) contig of the edge. Without loss of generality we can assume that *Forward* orientation is represented by 1 and *Reverse* orientation is represented by 0.

In addition, let $head : E \to \Gamma$ and $tail : E \to \Gamma$ provide the contig for the first vertex in the edge and the contig for the second one respectively. Let also $l\_e : E \to N$ gives the length of each

edge. The length of an edge is the same as the length of the read pair that it represents.

For any $e \in E$ we associate four real variables denoted by $startHead(e), endHead(e), startTail(e)$, $endTail(e)$. These variables correspond to the positions where the *start* and *end* parts of the contigs associated with the edge are placed.

Let $E(p)$ be such set that every edge in the graph that represents the read pair p be in it.

Let us note that for each read pair, there might be a number of edges. This is a result of putting $cov(c)$ vertices in the graph for each contig. The optimization function, however, needs to optimize the number of satisfied read pairs, not the number of satisfied edges. For that purpose we use the α_e variable, where $e \in E$. The idea is that just for one of all the edges that represent any given read pair, we should optimize the difference between the expected distance between the contig occurrences[11] and the actual one in the solution. We will add the following constraint for the values of α, in order to ensures that each read pair will be "placed" at least at one position. In other words, it ensures that at least for one pair of vertices that represent the contigs of the read pair, we will minimize the difference between the distance between those vertices and the distance suggested by the read pair.

$$\forall p \in P \quad \sum_{e \in E(p)} \alpha_e = 1 \tag{12}$$

Let the function $\underline{gap(e)}$ represents the difference between the expected distance between a pair of contigs according to the edge e and the actual one. Let also $\overline{gap(e)}$ represent the difference between the distance between the furthest ends of a pair of contigs and the one expected by the edge e[12].

Formally, this can be written as:

$$\begin{aligned}\underline{gap(e)} = {} & oHead(e) \times oTail(e) \times (startTail(e) - end.head(e)) \\ & + oHead(e) \times (1 - oTail(e)) \times (endTail(e) - endHead(e)) \\ & + (1 - oHead(e)) \times oTail(e) \times (startTail(e) - startHead(e)) \\ & + (1 - oHead(e)) \times (1 - oTail(e)) \times (endTail(e) - startHead(e)) \\ & - l\_e(e)\end{aligned} \tag{13}$$

$$\begin{aligned}\overline{gap(e)} = {} & oHead(e) \times oTail(e) \times (startTail(e) - endHead(e)) \\ & + oHead(e)) \times (1 - oTail(e))) \times (endTail(e) - endHead(e)) \\ & + (1 - oHead(e))) \times oTail(e)) \times (starttail(e) - startHead(e)) \\ & + (1 - oHead(e))) \times (1 - oTail(e))) \times (endTail(e) - startHead(e)) \\ & - l\_e(e) - l(head(e)) - l(tail(e))\end{aligned} \tag{14}$$

$$\forall e \in E \; y_e \geq \max(\overline{gap(e)}, -\overline{gap(e)}) \tag{15}$$

$$\forall e \in E \; y_e \geq \max(\underline{gap(e)}, -\underline{gap(e)}) \tag{16}$$

[11] It is provided by the read pair.

[12] To get this distance one have to sum the length of the edge with the lengths of the two contigs.

$$\forall e \in E \; y_e \geq 0 \tag{17}$$

The optimization problem can be written as

$$\min \sum_{e \in E} \frac{\alpha_e y_e}{|l\_e(e)|} \tag{18}$$

which is subject to constraints (13), ..., (17).

When all those equations are simplified and are given to a solver, it is possible to find the best positions for each of the contig occurrences of the contigs in Γ so that the read pairs are best satisfied.

We divide by the length of the edge in (18), in order to ensure that the longer read pairs are not more important than the short ones. Difference of 20 between the expected and the actual distance between two contigs might be negligible, if the length of the edge is 2000. It might be very significant, if the length of the edge is 40. As $l\_e(e)$ might take negative values, we divide by the absolute value of the length of the edge. If the length of the edge is zero, we divide by 1 instead.

5 Results

All the approaches that we described so far were run against a set of examples that have been provided by the GenScale team and its partners. For those examples one possible solution is also given. For some of them there might be small errors in the proposed solutions. For others, there might be a number of equivalently good solutions, according to the scoring functions of our algorithms. For that reason we have been using manual approach for comparing the solutions that our algorithms would find with those that were provided.

We will first describe the result of each of the approaches in turns and after that we will provide some information about their differences and how each of them compares to the others.

5.1 Maximum Weighted path approach

The simplicity of this approach lets it run very quickly for most of the cases. After the solutions are found, it is possible to calculate their scores according to the scoring function used for the Branch and Prune approach. After that we can filter all the suboptimal ones. That way very quickly we can obtain solutions that use distances and are very likely to be optimal[13].

Despite the fact that this approach is supposed to be very fast as it simplifies the task by not considering some of the information, for some of the big instances it still takes a considerable amount of time to finish. This mainly happens when there are a number of errors in the input data. The algorithm can be run in such a way that it will provide information about possible errors. The information consists of numbers of contigs, for which the algorithm believes that the coverage is incorrect. This is done with the use of artificial contigs and limiting the time for which the algorithm tries to solve the problem. The idea is that after that it will be possible to request verifications for those contigs, or at least regeneration of the contigs, but with different parameters. This will result in the creation of different contigs, that hopefully will not contain that much errors.

[13] Later we will provide information for the specific cases when the results will not be optimal

Another thing that can be done is to detect that for some time the algorithm is unable to improve the score of the solution it has found. Then it is possible to stop the algorithm and to provide the best solution that is found so far. It should become clear that this might not be the optimal solution. The downside of using such approach for improving of the runtime is that there is no guarantee for the optimality of the provided solution and that there can be only one such pseudo-optimal solution. It is impossible to quickly generate other equivalent solutions. The good thing is that the user will not have to wait too long for some solution. It is possible that the algorithm will be able to produce the best solution relatively quickly, but after that it will need a considerable amount of time to actually prove optimality. For that reason the use of a timeout seems like a reasonable approach.

To sum up, this approach is very useful for solving examples that have relatively small amounts of errors. It can be used also to detect if there are some errors in the data, so that if any of the other approaches is employed, it will be possible to tune some parameters that will allow it to better handle those errors.

5.2 Branch and prune

In spite of the naive way in which the Branch and Prune approach searches for optimal solutions, it does very well on most of the small and medium-size examples that we have. As its complexity is exponential on the average, when the size of the input reaches some critical level, every additional contig or read pair increases considerably the runtime. As a result for all the small and medium size examples, we were able to find the result in a matter of seconds. For some of the big ones, however, it would range from 5 hours on a 4 core CPU to no final result in two weeks. During that time the algorithm was able to go to very good results, but it was unable to finish exploring all the possible solutions. Thus, it was unable to prove that this is the optimal score for that example.

The parameters that can be provided can dramatically affect the runtime of the algorithm. For example, if we add a single artificial contig, this makes this approach run N times slower, where $N = \sum_{c \in \Gamma} cov(c)$. Another option that can significantly reduce the runtime is the best solution option. If we know that the best solution has score close to 0, the pruning based on the score will be able to eliminate large sections of the search tree. If, on the other hand, the value is too big, it will be unable to eliminate any solution at all until an actual solution is found. In order to select good values for those parameters, the genetic algorithm can be used.

5.3 Genetic algorithm

As it was stated earlier, the Genetic algorithm is not guaranteed to find the best solution for any given problem. Its strength is in that both in theory and in practice the runtime is approximately linearly dependent on the size of the input for any fixed parameters. If the input is fixed and the parameters are not, then the runtime is again approximately linearly dependent of the values of the parameters. That makes it good for solving small as well as big instances (see Table 3).

Although it gives very good results for almost all the examples, it fails to do that for one of them[14]. For that example the output it provides is very far from the optimal one that we were able to produce with the other approaches. Our explanation is that the optimal solutions have some very specific structure, that is very hard to produce using the genetic algorithm. For all the

[14]It is called "cucumis". It is the second biggest one. The structure of the solutions does not seem very complex

Table 1: Genetic Algorithm times/score with different arguments for the different examples. The arguments are encoded as a triple (number of runs, population size, repetitions count for each run). The values for each pair of example and arguments is in the format time/score.

	(1, 50, 50)	(1, 200, 50)	(1, 200, 200)	(5, 200, 200)	(1, 500, 600)	(15, 500, 600)
Euclyptus	0.0s / 0	0.0s / 0	0.1s / 0	0.5s / 0	0.1s / 0	0.4s / 0
Acorus	0.6s / 1	0.7s / 1	1.1s / 1	3.5s / 1	4.0s / 1	41.8s / 1
Agrostis	0.2s / 0	0.6s / 0	0.8s / 0	1.1s / 0	0.5s / 0	1.3s / 0
Atropa	0.0s / 0	0.2s / 0	0.3s / 0	2.4s / 0	4.4s / 1	10.4s / 0
Lecontella	0.7s / 104	1.2s / 2	2.3s / 2	5.6s / 2	6.8s / 2	86.1s / 2
Oenothera	0.8s / 113	1.2s / 14	2.0s / 212	6.9s / 113	14.2s / 12	118.0s / 11
Pinus	1.0s / 720	1.4s / 219	2.5s / 119	9.5s / 123	13.1s / 119	181.2s / 15
Cucumis	0.9s / 1607	1.5s / 519	3.7s / 1403	12.6s / 814	18.1s / 914	253.8s / 409
Euglena	0.9s / 227	1.6s / 325	4.2s / 114	16.5s / 10	24.0s / 7	317.1s / 6

other examples it gives results that are as good as those of the Branch and Prune approach or even better. Later we will provide information how this is possible.

The runtime for the big instances is a very good one, but that for the small ones is not as good as that of the other approaches. The reason is that in order to ensure that the best solution will be found, we are forced to run a number of iterations of the algorithm. If that approach is unable to find a solution with score 0, it is impossible to be sure that there will not be a better solution. That in turn makes it run a number of turns even after the optimal solution is found.

Using a good set of parameters, however, makes the genetic algorithm run approximately as quickly as the other approaches for most of the small examples. As a result, it is generally safe to always use it to pre-solve the example. If it is unable to find a solution with score of 0, or if we want to have all the optimal solutions, we need to use some of the other techniques as well.

Due to the fact that the genetic algorithm and the Branch and Prune one share some code, it is very easy to make them work together. First the genetic algorithm tries to figure out the number of artificial contigs[15] and the best score that the Branch and Prune should be able to produce. After that the Branch and Prune algorithm uses that information to configure itself in such a way that it will be most efficient. It will either prove that the genetic algorithm has found the best solution and it will find all the equivalent ones, or it will find a better solution.

The performance and the reliability of the results that the genetic algorithm provides, depend on the parameters that it receives. As it can be seen from Table 1, if improper values are used for the parameters, the provided solutions might be very different from the best one. For example, Lecontella has a solution that is very far from the optimal one, when small parameters are used (see the table). If the example is not very small and the results will be used directly for the Branch and Prune algorithm, it is better to spend some additional time to find a good heuristic solution, than to start with a very bad initial solution.

[15]This is not currently implemented, but we have planned it for a next version.

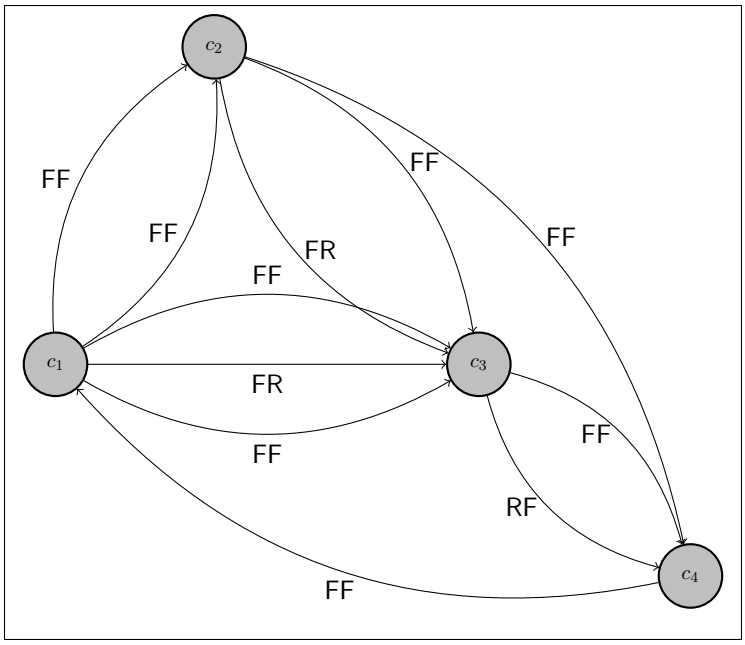

Figure 11: An example that produces different results with Maximum Weighted path approach and Branch and Prune approach

5.4 Distance based approach

Despite the really elegant and clean idea that this approach has, so far there are still a number of errors that it makes. Sometimes it produces solutions that are not connected, sometimes it positions contigs one over another. For that reason, we can not consider it as finished. It tends to run fast, especially when there are not too many errors. It also sometimes places contig occurrences of the same contig at the same position, when the coverage is incorrect in the data. So far we have no proof that this is a consistent behavior. It seems somewhat natural if we consider that only a limited number of read pairs are available and the algorithm tries to use them to position the contigs. In the case where the coverage of some contig is incorrect, it is very likely that there will be insufficient number of read pairs for that contig. As a result, it is more likely to place the contigs occurrences at the same positions.

5.5 Comparison

Despite our efforts, the models that we created are not always equivalent. For example, the Maximum Weighted path approach might produce results that can easily be converted to the optimal results of the Branch and Prune approach. For some special examples, however, it is possible that such conversion is not possible at all. Consider the following problem. We have four contigs c_1, c_2, c_3, c_4 and each of them has length of 10. For those contigs we have the following read pairs: $\{(c_1, c_2, 50), (c_1, c_2, 50), (c_1, c_3, 20), (c_1, c_3, 20), (c_1, \overline{c_3}, 20), (c_2, c_3, 20), (c_2, \overline{c_3}, 20), (c_2, c_4, 20), (c_3, c_4, 50), (\overline{c_3}, c_4, 50), (c_4, c_1, 20)\}$. For a visual representation of the graph, please see Figure 11.

There are three possible paths that the Maximum Weighted path algorithm will consider. The first is c_1, c_2, c_3, c_4, c_1 and the number of violated read pairs for it is 3. The second one is

$c_1, c_2, \overline{c_3}, c_4, c_1$ and the number of violated read pairs for it is 4. The third one is $c_1, \overline{c_3}, c_2, c_4, c_1$ with number of violated read pairs - 4. It will then choose the one with score three. However, if we consider the lengths of the read pairs as the Branch an Prune approach would do, the scores will change. The path c_1, c_2, c_3, c_4, c_1 will have 6 violated read pairs, whereas the path $c_1, \overline{c_3}, c_2, c_4, c_1$ will have 4 violated read pairs. This proves that the two models do not always provide the same result. That is not due to an error in either of the two approaches. The difference is in the subject of optimization. In a number of practical cases the two different optimization functions produce the same result, however, there are also cases when this is not the case.

With the use of other examples, it is possible to show that none of the approaches are actually equivalent. It is even possible that the Genetic algorithm will produce a better solution than that of Branch And Prune without artificial contigs. For this to happen, the artificial contigs should be required in the optimal solution, but not in some of the suboptimal ones. That way the Branch and Prune algorithm will produce an optimal solution and it will appear that it is the best one. But if the right number of artificial contigs is added, then it will be possible for the Branch and Prune approach to produce even better results. That, however, will make it run slower.

The Maximum Weighted path approach and the Branch and Prune approach share the same idea for the structure of the optimal solution: it should be a path in the graph. This effects the optimal solutions that those approaches can find. For that reason artificial contigs have to be added to the explored structure so that any solution can be produced, at least in theory. The price that needs to be paid is the runtime.

Those approaches, however, have some very valuable properties as well. They are the only ones that can find all the optimal solutions for some problem. They can guarantee that under some circumstances, there are no better solutions than those that they provide. They can also provide information about the quality of the raw data. Under perfect circumstances there will be no need for artificial contigs as all the contigs that need to be adjacent in the final solution, will have read pairs that will give information about their relative position and orientation. The places where we are forced to use artificial contigs are places where the raw data was not detailed enough. We can provide this information to the scientists, giving them the solution that we were able to find as well as some information that if provided, will enable us to produce better results. The team's understanding is that this will be very useful for the biologists.

On the other hand, the Distance based approach and the Genetic algorithm aim to satisfy the maximum possible number of read pairs. It does not matter what structure those edges will create in the graph, as long as all the contig occurrences have some positions and as long as there are not too many overlappings. That allows them to avoid the use of additional data that would slow them down. The Distance based approach can even guarantee that it has found an optimal solution, but that might require a considerable amount of time. Also the fact that in that approach a single violated read pair can greatly change the value of the objective function might not be the best decision. It assumes that all the read pairs are correct and in this regard this is the best way to handle the difference between the expected distance between the contigs and the actual one. If there might be read pairs that are absolutely wrong, however, the application of that strategy can change the solution so that they are less violated, which is not what most people would expect.

The sizes of the examples, that were run, can be found in Table 2. There is information about the count of contigs, the count of read pairs, the total number of contig occurrences, the biggest coverage for any contig in the example and the correctness of the provided solution. The correctness of the provided solutions is checked manually. *Small error* indicates that the provided solution had

Table 2: Examples sizes

Name	#contigs	#read pars	total contig occurrences	Biggest coveraga	Correctness of provided solution
Euclyptus	4	5	5	2	Small error
Acorus	5	10	7	2	Correct
Agrostis	7	15	11	3	Correct
Atropa	8	14	11	2	Small error
Lecontella	7	17	11	3	Correct
Oenothera	19	43	25	2	Small error
Pinus	23	67	34	3	Incorrect
Cucumis	29	113	45	2	Correct
Euglena	24	101	54	4	Correct

small error in the end. If the last one-two contigs from such example solutions are ignored, they will be correct ones. When *Incorrect* is indicated, the provided solution had significantly different coverage for the contigs. That makes it impossible for our algorithms to find the same solution as the one that was provided.

In Table 3 information about the runs of every of the approaches with every of the examples can be found. Note that the number of solutions for the Genetic algorithm and the Distance based approach is always one as they can not produce more than one solution. When there is no good information about the way some approach behaves for some example, we have put NA at that row. As it can be seen, for most of the examples the approaches are able to provide exactly the same or equivalent solution to the one, provided with the data. There are some examples, however, where the provided solution was not produced by any of the approaches. We believe that for them the solutions that we provided are good enough, but it was not possible to manually verify that.

The score for the Maximum Weighted path approach are provided in terms of satisfied read pairs, whereas those for Branch and Prune approach and Genetic Algorithm approach are provided in terms of violated read pairs.

In Table 4 it can be seen how the Branch and Prune approach compares to the Maximum Weighted Path approach when the solutions of the latter are converted and scored as solutions of the first. As it can be seen, for all the examples that we ran the comparison, the results are identical. This makes us believe that for some situations the use of the generally faster Maximum Weighted Path approach is better. As it was already stated, the two approaches might not always find the same results. In those situations the Branch and Prune approach will be able to produce better results than its opponent.

6 Future work

It is possible to improve each of the approaches that were described so far. For most of them it would be good to improve the performance. For example, the Maximum Weighted path approach and the Distance optimization approach rely on linear programming solvers. They are not optimized for the types of tasks that are required for those two approaches. As a result, we believe that it

is possible to improve the performance by using "dedicated solvers". In other words, solvers that know more about the tasks that they are trying to solve. As a result they can use a number of characteristics of the expected optimal solution in order to improve the runtime and the memory usage.

The Genetic algorithm can also be run in multi-threaded mode. The idea would be to either run each of the separate executions on separate threads, or to divide the creation of the next generation into independent parts and each thread will take care for a separate part. For example, each thread will calculate the scores and will make some distinct part of the next population. After that the different parts of the next population will be put together. We believe that this can significantly improve the runtime of the Genetic algorithm, without using much more memory.

For the Branch and Prune approach, it is possible to find a upper bound of the score of a partial solution. For example, it is possible to assume that all the remaining read pairs will not be satisfied. It is possible to subtract the number of read pairs that will be used for finishing the current solution from that number. This will provide an upper found of the score of the current solution. It can be used to alter the Branch and Prune approach and to make it Branch and Bound. Then it is possible that a much greater number of suboptimal solutions will be filtered. That can lead to a big improvement in the runtime. It is also possible to try to find some other criteria that will allow some suboptimal solutions to be pruned.

Another possible improvement would be the tuning of the algorithms in order to provide better results in the case of a greater number of errors. That would be required so that bacteria genome can be scaffolded using the provided approaches. For it the number of errors is a few times greater than that of chloroplast and mitochondria organelles. Also the size of the genome of a bacteria is a few times greater than that of the examples that we consider. As a result the runtime of the provided approaches should be improved in order to enable them to run in some acceptable amount of time.

It would also be very interesting to setup some of the already available approaches for scaffolders and run them for the examples that we have. That way it would be possible to really understand the quality of the scaffolding approaches that we have provided.

Another thing that can be improved in the current implementation of the Distance based approach is the addition of some penalty for contig overlapping. It seems also useful to allow that approach to use just some of the contig occurrences from the input if that will avoid overlappings. It will have to pay some penalty on the optimization function for that, however.

7 Conclusion

The contig scaffolding problem for second-generation data is a famous research topic. From a computational view this is a very difficult (NP-hard) combinatorial optimization problem and efficient tools for its solution are crucially missing. All existing approaches either use heuristics with no performance guarantee in terms of quality of solutions, or exact techniques that do not profit from all available input data information. Moreover, they target to find only one solution and hence do not satisfy the requirement to provide all optimal solutions.

The goal of this study is to explore various combinatorial optimization techniques for solving the scaffolding problem and overcome the above weaknesses. Our first contribution consists in proposing a formal definition and formulating the underlying optimization problem.

Furthermore, we focus on a particular case of scaffolding problem related to the assembly of

the circular genomes of the chloroplast and the mitochondria organelles which are not yet fully resolved. For this particular case we develop four different approaches that are designed to provide several (possibly all) optimal solutions. One of them is a heuristic based on a genetic algorithm, while the other three are exact algorithms that use three different combinatorial optimization techniques. In the first exact approach we propose a Mixed Integer Linear program for solving the scaffolding problem. This program uses the powerful Gurobi solver. This approach considers a simplified version of the problem without using all available information (namely the distances between the contigs). However, even with such a partial information this approach is able to detect inconsistencies in the input data. When it exists, the solution of this simple task can be used to create a solution for the original problem.

Our second exact approach aims to explore all possible solutions based on Branch and Prune search strategy. The underlying algorithm has been implemented in the framework of this internship. It uses a number of optimizations, so that the runtime is acceptable for medium size instances. When combined with our heuristic approach it should be able to find all the optimal solutions even for larger instances.

The last approach we proposed here uses all available information and using the AMPL programming language it generates a Quadratic Mixed Integer Program. The optimization problem is then solved using the KNITRO solver for nonlinear optimization.

All approaches we have created have been tested on a benchmark set provided by GenScale partners. In a reasonable amount of time, they find solutions that are either the same as the provided solutions for our test examples, or at least equivalent to them. This makes us very confident in the correctness of our approaches and we strongly believe that they will be useful for the community of the domain.

References

Adel Dayarian, Todd P Michael, and Anirvan M Sengupta. Sopra: Scaffolding algorithm for paired reads via statistical optimization. *BMC bioinformatics*, 11(1):345, 2010.

Jason Eisner. State-of-the-art algorithms for minimum spanning trees - a tutorial discussion, 1997.

Sara El-Metwally, Taher Hamza, Magdi Zakaria, and Mohamed Helmy. Next-generation sequence assembly: Four stages of data processing and computational challenges. *PLoS computational biology*, 9(12):e1003345, 2013.

L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial Intelligence Through Simulated Evolution*. John Wiley & Sons, 1966. URL `http://books.google.com/books?id=75RQAAAAMAAJ`.

Song Gao, Wing-Kin Sung, and Niranjan Nagarajan. Opera: reconstructing optimal genomic scaffolds with high-throughput paired-end sequences. *Journal of Computational Biology*, 18(11): 1681–1691, 2011.

Michael R Garey, David S Johnson, and Larry Stockmeyer. Some simplified np-complete problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 47–63. ACM, 1974.

Alexey A Gritsenko, Jurgen F Nijkamp, Marcel JT Reinders, and Dick de Ridder. Grass: a generic algorithm for scaffolding next-generation sequencing assemblies. *Bioinformatics*, 28(11):1429–1437, 2012.

John H Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

Martin Hunt, Chris Newbold, Matthew Berriman, and Thomas D Otto. A comprehensive evaluation of assembly scaffolding tools. *Genome biology*, 15(3):R42, 2014.

John D Kececioglu and Eugene W Myers. Combinatorial algorithms for dna sequence assembly. *Algorithmica*, 13(1-2):7–51, 1995.

David R Kelley, Michael C Schatz, Steven L Salzberg, et al. Quake: quality-aware detection and correction of sequencing errors. *Genome Biol*, 11(11):R116, 2010.

Ruiqiang Li, Hongmei Zhu, Jue Ruan, Wubin Qian, Xiaodong Fang, Zhongbin Shi, Yingrui Li, Shengting Li, Gao Shan, Karsten Kristiansen, et al. De novo assembly of human genomes with massively parallel short read sequencing. *Genome research*, 20(2):265–272, 2010.

Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

Jaan Plesnik. The np-completeness of the hamiltonian cycle problem in planar digraphs with degree bound two. *Inf. Process. Lett.*, 8(4):199–201, 1979.

Leena Salmela, Veli Mäkinen, Niko Välimäki, Johannes Ylinen, and Esko Ukkonen. Fast scaffolding with small independent mixed integer programs. *Bioinformatics*, 27(23):3259–3265, 2011.

Ariella Sasson and Todd P Michael. Filtering error from solid output. *Bioinformatics*, 26(6): 849–850, 2010.

Pascal Van Hentenryck, David McAllester, and Deepak Kapur. Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis*, 34(2):797–827, 1997.

Jeffery Westbrook and Robert E Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7(1-6):433–464, 1992.

Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

Daniel R Zerbino, Gayle K McEwen, Elliott H Margulies, and Ewan Birney. Pebble and rock band: heuristic resolution of repeats and scaffolding in the velvet short-read de novo assembler. *PloS one*, 4(12):e8407, 2009.

Table 3: Models results

Approach, example name	Best score	Time	#solution	correct	notes
Maximum Weighted path approach					
• Euclyptus	4	0.2s	2	Almost	See Table 2
• Acorus	9	0.5s	2	Yes	
• Agrostis	13	0.5s	4	Yes	
• Atropa	11	1s	2	Almost	See Table 2
• Lecontella	14	0.6s	4	Yes	
• Oenothera	40	163s	4	Almost	See Table 2
• Pinus	-19939	1h	0	NA	Stopped by timeout, See Table 2
• Cucumis	99	35s	32	Yes	
• Euglena	-29897	1h	0	NA	Stopped by timeout
Branch and Prune approach[16]					
• Euclyptus	0	0.1s	2	Yes	See Table 2
• Acorus	1	0.1s	8	Yes	
• Agrostis	0	0.1s	2	Almost	Equivalent
• Atropa	0	0.1s	2	Almost	See Table 2
• Lecontella	2	0.2	4	Yes	
• Oenothera	12	1.1s	36	Yes	
• Pinus	Unknown	2d	0	NA	See Table 2
• Cucumis	6	12h	16384	Yes	
• Euglena	≤ 19	10h	0	NA	The run did not finish
Genetic algorithm approach					
• Euclyptus	0	0.0s	1	Almost	See Table 2
• Acorus	1	0.6s	1	Equivalent	
• Agrostis	0	0.2s	1	Equivalent	
• Atropa	0	0.2s	1	Almost	See Table 2
• Lecontella	2	86.1s	1	Equivalent	
• Oenothera	11	118.0s	1	Unknown	Differs significantly from the provided one, but seems correct
• Pinus	15	181.2s	1	Unknown	See Table 2
• Cucumis	409	253.8s	1	No	Differs significantly
• Euglena	6	317.1s	1	Unknown	Seems correct
Distance optimization approach					
• Euclyptus	0.0	0.3s	1	Yes	
• Acorus	0.08	1s	1	Equivalent	
• Agrostis	0.0	1s	1	Yes	
• Atropa	0.0	1s	1	No	Many overlappings
• Lecontella	86.11	7.5s	1	No	Not cyclic
• Oenothera	567.35	120s	1	No	Not cyclic, has overlappins
• Pinus	1699.93	1h	0	No	Stopped by timeout, see Table 2
• Cucumis	0.36	150s	1	No	Overlappings
• Euglena	39.27	1h	0	NA	Stopped by timeout

Table 4: Comparison between Branch and Prune approach and Maximum Weighted Path approach when the solution are evaluated by the Branch and Prune approach scoring function.

Name	#solutions Branch and Prune	Best score Branch And Prune	Time Branch and Prune	#solutions Weighted Path	Best score Weighted Path	Time Weighted Path	Comparison
Euclyptus	2	0	0.1s	2	0	2s	Correct
Acorus	8	1	0.1s	8	1	2s	Correct
Agrostis	2	0	0.1s	2	0	2s	Correct
Atropa	2	0	0.1s	2	0	2s	Correct
Lecontella	4	2	0.2s	4	2	2s	Correct
Cucumis	16384	6	12h	16384	6	140s	Correct