Freie Universität Berlin

Bachelor thesis

produced at "Max – Planck – Institut für molekulare Genetik"

—

Improvement of a genome assembly with Hi–C data by simulated annealing

by

Marco Johannes Stöckler
born December 19, 1993

Fachbereich Informatik und Mathematik
Institut für Informatik
Matr.: 4760992

1. Gutachter: Prof. Dr. Martin Vingron
2. Gutachterin: Prof. Dr. Annalisa Marsico
Betreuer: Dr. Robert Schöpflin

Eidestattliche Erklärung

Hiermit versichere ich an Eides statt, dass die vorliegende Arbeit von mir selbstständig und ohne unerlaubte Hilfe Dritter verfasst wurde und ich keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie wörtliche und sinngemäße Zitate als solche kenntlich gemacht habe.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen und wurde bisher nicht veröffentlicht.

Datum, Ort	Unterschrift

Abstract

Although DNA sequencing technologies have enhanced, genome assemblies have still a great ressource consumption. This thesis wants to propose how a genome assembly, using chromatin interaction contacts from the Hi–C method as distances, can be improved. A hierachical cluster analysis and further an optimization technique, called simulated annealing were applied to approximate an optimal genome assembly. Applying this approach on a genome assembly of the *lat.: talpa occidentalis*, the results show that an improvement could be achieved. Still, these methods should be modified and the approximation could be increased further. This approach can be applied to every genome assembly with existing associated Hi–C data to improve genome assemblies.

Contents

I. Introduction

Phentotypic characteristics of the Iberian mole, *lat.: talpa occidentalis*, for example, Morphologies and mutations of extremities or development of testes tissue in female moles, are currently interesting for researchers, because they are not explored on a genetic level. Therefore it is necessary to have the whole genome sequenced. Since this has not been realized yet for the *lat.: talpa occidentalis* it offers an oppurtunity to try out new methods in the procedure of gaining a reference genome.

1. Motivation

To get an idea of the aims of this work, some prior knowledge is required. The following sections will explain the biological background of this bachelor thesis.

1.1. Reference genome

To analyse a specific species on a molecular level, it is common to sequence a reference genome first. A reference genome is a nucleic acid sequence database and a representative of a specific species, but it is not a DNA sequence of a single member of that species. It is often sequenced and assembled from a number of donors and contains mosaics of every donor.

If scientists want to analyse a specific organism, they sequence its DNA and compare it to the reference genome of its species. They can also compare two reference genomes of different species to study the sequence homology of these organisms. Thus, it is possible to gather informations, for example about shared ancestors in the evolution. This can be done by aligning the two reference genomes and detect similarities in the sequences.

Also for most ”next generation sequencing”–methods like ChIP–sequencing (chromatin immunoprecipitation sequencing), which, for example, can be used to observe interactions between DNA and proteins genome-wide or like RNA–sequencing, which is an approach to analyse and profile RNA, reference genomes are necessary.

These examples show that reference genomes are widely used in genetic research. One major problem is that sequencing a reference genome is very expensive and time consuming with current methods.

For example the “Human Genome Project” was a project whose aim was to sequence a reference genome of our species the “Homo Sapiens.” This project started in 1990 and took thirteen years to sequence all three billion base pairs. The resource consumption was immense, it cost round three billion dollars (one dollar per base pair). [1]

Obviously, this is too expensive to do it for every organism and so the number of sequenced genomes is still small and therefore it is important to develop a faster procedure for this part of the genome research.

Since a reference genome can not be sequenced at once, it has to be assembled from fragments, which is then a genome assembly. This thesis wants to propose an Improvement of a genome assembly, hence it is important to understand a genome assembly in detail. The next section will give insight in an assembly.

2. Genome assembly

A genome assembly is an aligning of small fragments of DNA to identify the whole genome. To understand this better, the explanation starts at the beginning.

2.1. Chromosomes and ploidy

Every animal is an eukaryote. Eukaryotes are organisms which consist of cells with a nucleus, surrounded by a membrane. Organisms with cells that do not have a nucleus are called prokaryotes and so the genome is not bound within it. The whole genome in eukaryotic cells is partitioned in chromosomes and consists of DNA and special proteins. This material is also called "chromatin."

Some eukaryotes have multiple copies of chromosomes, this is referred as ploidy level. For instance Mammals are diploid, this means they have two sets of chromosomes in every cell. To go further: The human genome consists of 23 chromosomes and every cell has one copy. These 46 chromosomes in one cell are organized in 44 autosomes (22 unique autosomes) and 2 allosomes (sex chromosome). The 2 allosomes determine the sex of a species.

How is this important for a genome assembly?

2.2. Genome Assembly

The result of a sequencing machine are "reads". Reads are short fragments of the DNA, which cannot be sorted correctly to the chromosomes they belong to by the sequencing machine. It just writes all detected reads separately into a file.

To assemble the small fragments, they need to be aligned to each other. If two reads have a matching part, they can be put together. A collection of matched "reads" is called "contig". These contigs can be linked together to form a "scaffold". A scaffold is a compound of contigs. Between these contigs, there can be gaps, whose length is known. Further, scaffolds can be grouped to chromosomes and ordered within a chromosome. If all these steps are finished, the whole genome is nearly identified. In the process more upcoming reads can fill the remaining gaps by aligning them to the assembly.

So the goal is to assemble the sequences into one big sequence, which is, in the best case, the whole genome, which can then be used as reference genome.

3. Hi–C

“Hi–C” is a method to study the three-dimensional architecture of genomes. This method introduced by, Lieberman–Aiden et al. (2009) [2], can identify chromatin (see section 2.1) interactions genome–wide. It is very important to understand how chromosomes are folded in the nucleus, because it can provide insight into chromatin structure and gene activation.

3.1. The Method

The first step of Hi–C is to cross–link interacting DNA to each other with formaldehyde. Afterwards, these fragments will be cut with a restriction enzyme. To mark the ends, they are being filled with biotin. The next step is a ligation, which results in a number of genome–wide pairs of DNA fragments that were in close proximity to each other in the nucleus. The pairs will be sheared and the junctions that are marked with biotin will be pulled down. These paired–ends can now be sequenced and mapped to a reference genome. [4]

3.2. The Result

The results of this method are genome–wide informations about pairs of fragments that are in close proximity and correspond with each other [3]. These informations are chromatin contacts and can be shown in a symmetric map. The highest contacts are on the diagonal, but there are also spikes visible. Spikes can be identified as chromatin loops and they correlate with gene activation. [4]

4. Association between Hi–C and genome assembly

The Association between a genome assembly and this method to identify close proximity of fragments is not that obvious. A good assembly of complex genomes is connected with great resource consumption and is very hard to achieve. Another problem is today that the most sequenced DNA are very short reads with many gaps. More data needs to be collected to fill these gaps and to assemble this huge amount of genomes in an eukaryotic organism. Unquestionably, constructing a genome assembly has to become more efficient and less expensive than it is today.

Kaplan et al., (2013) [5] used the measured chromatin contacts from Hi–C as a distance between individual contigs to order them accurately and form scaffolds. So, the more chromatin contacts can be mapped to two contigs, the closer these contigs are to each other. The highest number of mapped contacts are of course on the same contig and can not be found on two different contigs. But if a high chromatin contact is found between two different contigs, it is reasonable that they are connected locally.

Also Dudchenko et al., (2017) [6] did a de novo assembly using Hi–C data. They constructed scaffolds of chromosome length. Both authors validated their methods by assembling a "new" human genome. This shows that Hi–C can not only be used to get a deeper knowledge of the chromatin architecture, but can also help to create genome assemblies of species with large genomes in an efficient way.

This association will be explained further by taking the example of the *talpa occidentalis*.

5. Current Status and problems with the assembly "talOcc3"

The assembly of the *talpa occidentalis* currently faces many problems. Only a scattered genome has yet been constructed, consisting of scaffolds that are not ordered to chromosomes. Accordingly, the Hi–C contact matrix does not show a desired result.

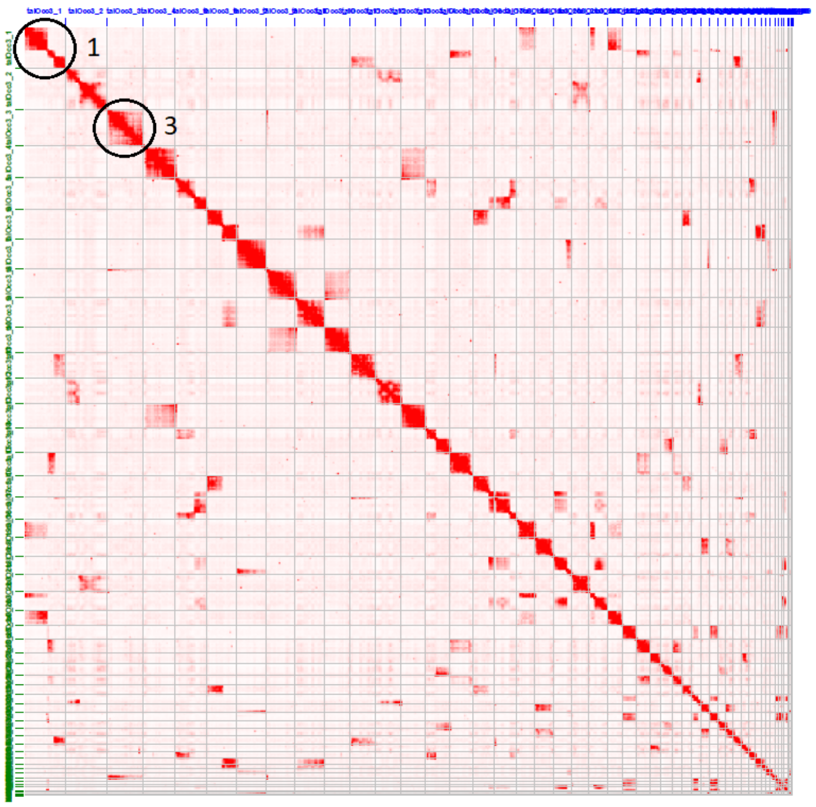


Figure 1: Hi–C map of *Tal. occ.* the longest 49 scaffolds
Visualized by Juicebox [8]

Figure 1 shows the longest 49 scaffolds of the "talOcc3" assembly in a Hi–C contact matrix. This heat map shows the cis–and trans–contacts of all scaffolds to each other. A cis–contact is a matrix entry on the diagonal, which is a contact within one scaffold, and a trans–contact is between two different scaffolds. In this work, like Kaplan et. al (2013) [5] and Dudchenko et al., (2017) [6], the chromatin contacts from Hi–C are interpreted as inversed distances between two scaffolds. Meaning, the higher the distance, the closer is the real distance between two scaffolds.

Problems of this assembly are directly visible in this heat map. As mentioned, the scaffolds are sorted by length and not according their real locations on the genome, which leads to many spikes across the matrix. Also, some misassembly of scaffolds can be seen in this heat map, as wrong heat flow within a scaffold. For example the scaffold 1 shows such a wrong heat distribution, whereas scaffold 3 looks like it should.

One problem is not directly visible in the map, its a fact that the sequencing machine cannot know if a sequence fragment is read from left to right or the other way round. This can result in a "wrong" orientation of a sequence.

6. Aims of this work

The main aim of this thesis is to improve the genome assembly of *talpa occidentalis* on the basis of the Hi–C data with simulated annealing algorithm. This optimization technique has to solve the two mentioned problems:

1. false ordering of scaffolds
2. false orientation of scaffolds

In the following part, the two main methods, which have been developed, will be explained. The general idea of how to optimize the assembly, the actual work flow and the choice of the programming language will be presented first, followed by the method which calculates a clustering of the scaffolds according to the data from Hi–C. Last but not least, the optimization algorithm and the most important operations will be described.

II. Methods

7. General Idea

As explained, one aim is to cluster scaffolds to chromosomes and rearrange them within. Figure 1 has shown, that there are many problems with the first assembly. So the idea is to change the order of scaffolds in a way that they cluster around the main diagonal and the "heat" decreases, the farther it is from the diagonal. A second aim is to change the orientation of scaffolds, if they are read the wrong way. Also the missassemblies (see 5) of scaffolds can be solved, by breaking them up into two or maybe even more scaffolds, thus making the assembly more precise.

In a perfect solution the heat map has 19 domains that are clearly distinguishable from each other and the "heat" decreases smoothly. These domains will present the 19 chromosomes of the *talpa occidentalis*. The scaffolds within one chromosome are ordered and may have changed orientation.

7.1. Work flow and Tools

The first idea, as mentioned, was to cluster the scaffolds into groups. To achieve this, an appropriate input was needed. The Input was created with HiC–Pro, a pipeline for processing Hi–C data from raw fastq files [9]. It writes the data in a triplet format: (Bin1, Bin2, CountValue), where the "CountValue" is the distance between "Bin1" and "Bin2". So, the file consists of unique pairs of bins with a value. Another file defines the base pair size of a bin and assigns it to a scaffold. An R–Script, that was written for the exact task, processes this input and clusters the scaffolds into new groups. These newly built groups of clusters can be visualized. The Output of the R code is in a format, that can be read by Juicer [7], a tool which processes Hi–C data and provides a *.hic*–file. This file can be an input for the Hi–C visualizing tool Juicebox [8], which builds the Hi–C maps.

The next step was to implement the simulated annealing algorithm as shown in Figure 2. The output of the algorithm is kept very simple and will be explained later in detail. It is then processed by another R–Script, that modifies the files for Juicer according to the results of the algorithm. Then again Juicebox can visualize the outcome.

Figure 2 demonstrates the explained work flow in a simple way. The colouring highlights the external tools (blue), input and generated files (yellow) and self–implemented tools (red). The arrows describe the use and creating of files.

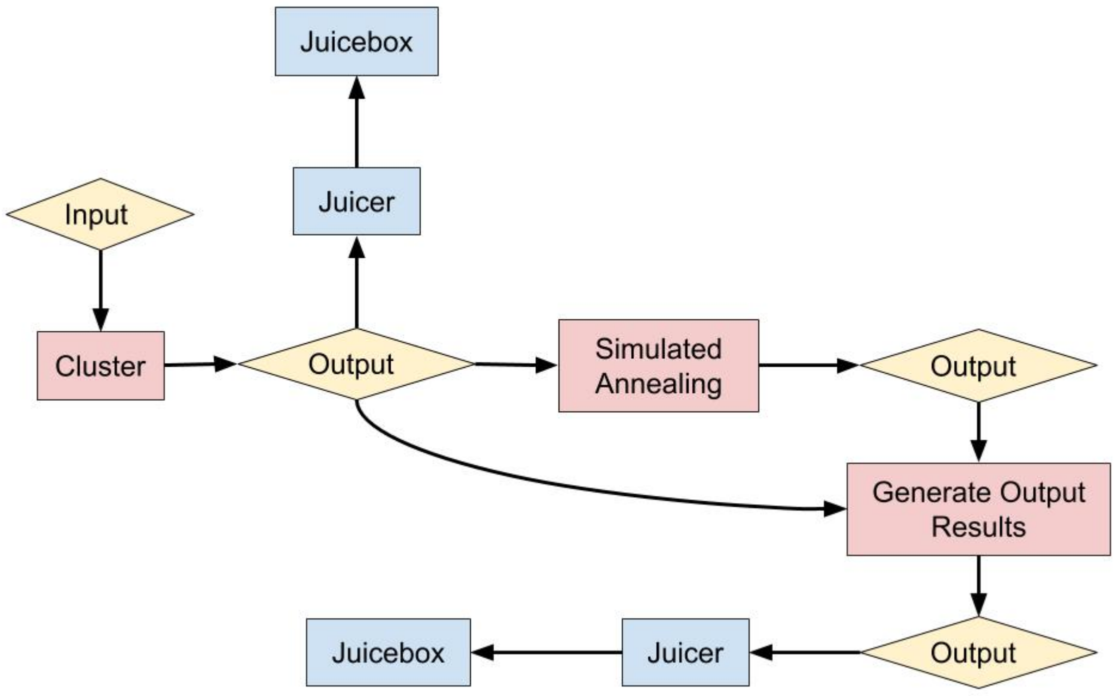


Figure 2: Workflow
Red: Created tools, blue: External tools, yellow: Files

7.2. Programming Language

The choice of a programming language is often a difficult thing. It is very important that the program meets the requirements that are stated, thus the first thing to do is to become aware of these. Next, it is good to know what the specific programming language offers, its advantages and disadvantages and sometimes what resources or preconditions it involves.

So the question is: “What are the requirements?” To answer this, it is helpful to clarify the starting point. In this case the first clustering tool, see figure 2 works with huge input files, which are in table format. Therefore, it is important to read and manipulate the input in an efficient and easy way. The next question is to think about the goal of the program. This script is being used to process the input, cluster the scaffolds and also create many output files, that will be used later. So a next requirement is to write files in a quick way.

As mentioned in section 7.1, the language used for the clustering is R. Although R is not that fast and efficient as other programming languages, it is very easy to use and the programming overhead for reading, writing and manipulating huge files is on a minimal level.

The next tool, which had to be implemented was the simulated annealing algorithm. It was tough to find a suitable data structure for the problem, additional requirements were fast loops and readability of the code. The algorithm was first implemented in Python, due to fast prototyping, readability and great code overview. But it turned out that was not the best choice. Python is an interpreter language and has a dynamic type system. The implementation was really slow, although some runtime optimizations were made. It was not possible to work and improve the code in a short time.

The algorithm was reimplemented in the programming language Java, because of the mentioned problems with Python. The static type system, a faster data structure and the compiled (and not interpreted at runtime) program made the developed algorithm much faster and thereby easier to improve.

The last tool, which generates the results, is also written in R. It is only a very small tool, that does not have to meet many requirements. It was important to read and write into files fast.

For the sake of completeness, it should be noted that the tools are written in Python version 3.5.2, Java 8, and R version 3.3.2.

8. Clustering

To make sure, the data is sufficiently workable for the optimization of the genome assembly, a clustering to the scaffolds had to be applied. The result should show, that the scaffolds can be grouped to distinguishable domains. The clustering should achieve a good and fast grouping of scaffolds, while the optimization algorithm should result in a very good approximated optimal solution for the two problems, mentioned in section 6: false ordering and false orientation of scaffolds. These two methods can be processed independently or successively. Either Pre–cluster the scaffolds first and then find a local optimum of this solution with the algorithm or compute only method.

The following subsections will describe how the clustering is developed, what the major problems were and how they were solved. The method will be explained in theory, no specific code will be analysed.

8.1. Idea

The basic idea was mentioned many times before, but never explained in detail. So far it was always called clustering, but actually it is a hierarchical cluster analysis. This method is distance–based and builds a hierarchical structure of specific objects.

There are two different techniques of a hierarchical cluster analysis. One is called the divisive approach, where all observable objects are in one cluster at the beginning and in every recursive step one object is downgraded in the hierarchical rank. This is obviously a "top–down" approach. The other is quite the opposite, the agglomerative approach. This "bottom–up" method considers every object as one cluster and in every recursive step two clusters merge into one. This newly generated cluster moves up in the hierarchical rank.

Further, the hierarchical cluster analysis has many variants how to deal with the distances. There a different metrics, that influence the cluster in different ways, because the measurements of the distances are not the same. For example: Two points in a two dimensional space have different distances, if they are measured by different metrics.

Besides the option to choose between metrics, there is also the option to choose between different linkage criteria. These criteria are functions and determine the linkage of two clusters. For example, "single–linkage" clustering uses the minimum distance of an object in a cluster to calculate the new distances between other clusters, whereas "complete–linkage" clustering uses the maximum distance of an object in a cluster.

To solve the problem, to group scaffolds, it is intuitive to use the agglomerative approach (bottom–up). Starting with one cluster per scaffold, in the end there should be 19 distinguishable clusters for the 19 chromosomes of the iberian mole. Furthermore, the chosen metric is UPGMA, short for "Unweighted Pair Group Method with Arithmetic mean". This metric is often used to reconstruct phylogenetic trees too.

As mentioned, agglomerative clustering is a "bottom–up" approach and at the beginning every object is one cluster. At every step the two clusters with the minimal distance are merged into one. The new distances to the other clusters are calculated as follows:

$$d_{C,K} = \frac{|A| \cdot d_{A,K} + |B| \cdot d_{B,K}}{|A| \cdot |B|} \tag{1}$$

A and B are the two clusters, that are merged into cluster C and with $d_{C,K}$ the new distance can be calculated from cluster C to cluster K. With this equation, all distances, even if the number of objects differs, are calculated *unweighted.* This is necessary to make sure that different numbers of objects do not influence the newly calculated distances.

In the next sections the important steps, that had to be made, and some problems, that had to be solved to achieve a satisfying clustering, will be described.

8.2. Normalizing the data

Before a clustering can be made, the data has to be processed. As mentioned in section 7.1 , the two input files are tables with following columns for the first file (Bin1, Bin2, CountValue) and the second file (Scaffold, StartBase, EndBase, Bin), which defines the size of the bin and assigns a bin to a scaffold. "Start– and EndBase" describe, at which base a bin starts and ends. The input can have different bin sizes. The first step is to assign every bin to a scaffold. So the two files can be merged to a table with seven columns: (Bin1, Bin2, countValue, Scaffold1, Scaffold2, BinCount1, BinCount2), where "Scaffold1" includes "Bin1" and "Scaffold2" includes "Bin2". "BinCount1" refers to the length of "Bin1" and analogous for "BinCount2".

Now, the table can be aggregated. All "CountValues" of each pair (Scaffol1, Scaffold2) can be summed up, so that the new table does not contain bins any more. This table now contains a unique pair of scaffolds with the length of both scaffolds and a value, that describes the distance between these two. The distance is inverse. That means the higher the value, the closer they are to each other.

Since every Scaffold has a different length, the summed up “CountValue” can not be compared correctly. Two Scaffolds of great length have obviously a much higher distance value than two scaffolds of short length. But that does not mean that a higher value automatically implies that the two scaffolds are closer to each other. Of course the long scaffolds have a closer aggregated proximity to each other, but only because more Hi–C contacts can be mapped to a longer scaffold, than to a smaller one. This does not say much of the real proximity. To achieve comparable distances, the values have to be normalized. This normalization is inspired by Kaplan et. al, (2013) [5]. The following equation shows the normalization:

$$\overline{c} = ln\left(\frac{c}{l_i \cdot l_j} + 1\right) \cdot (-1) \text{ for } i \neq j \tag{2}$$

l_i and l_j are the lengths of scaffold i and scaffold j and c describes the distance between these two scaffolds. Every contact is divided by $l_i \cdot l_j$ to set the distance in a comparable relation with the length of each scaffold pair. Then 1 is added, to prevent $ln(0)$. Using the *natural logarithm* sets high numbers in a better relation to smaller ones, since it penalizes high numbers more.

Since the distances are inverse, they can be corrected by multiplying with -1. The distances are negative and the shortest distance is 0. Negative numbers do not affect the clustering. When i equals j, the proximity is obviously 0.

The newly calculated distances can form a symmetric distance matrix with $n \cdot n$ entries, where n is the number of scaffolds. With this distance matrix the clustering can be calculated.

In the next section, some problems, that effect the clustering, will be shown and how the issues are avoided or dealt with.

8.3. Shorten the data

8.3.1. Problems

Many problems occurred during the clustering and only some are solvable. A major challenge was to process the data all at once. The scaffolds vary very much in size, only a few scaffolds have a length of much over a million base pairs, others are smaller than just a hundred thousand base pairs.

Although, the normalization function, as shown in equation (2), tries to set the values in a proper relationship and the clustering method tries to do the same again on the already normalized data, this length variation is to wide to deal with properly. The longest scaffolds were clustered satisfactorily, but the smaller ones had distances, that differ in a minimal range. It is impossible to decide either scaffold i and j with a distance of 0.0001 should cluster or scaffold i and x with a distance of 0.00011. In the end, all scaffolds with very similar distances form one cluster by being attached to each other successively.

Another problem were the misassembled scaffolds, as mentioned in the beginning, see 5. This false assembly had high impact on the clustering, because the wrongly assembled scaffolds could not be allocated correctly. It did not matter how good the normalization of the data was, the result was never satisfying. The Hi–C map of the first clustering explicitly shows the misassembly of a few scaffolds. Some Peaks spread too much and are not clustered around the diagonal.

Longest 49 Scaffolds. In section 5 figure 1 shows the 49 longest scaffolds for a particular reason. All other scaffolds are so small, that they are not visible in the Hi–C map and the clustering did not work with them, due to the small interval of the distances. Because the problem could not be solved in a simple way, only the longest 49 scaffolds were used to show, that a clustering can improve a genome assembly.

Highest 10% values. Even though the normalization function had been used to adapt high values, it was not enough. To uprate small scaffold pairs against longer ones, only the highest 10% Hi–C contacts were aggregated. This workaround does not impact the accuracy of the distances, but it changes the weight of longer scaffolds in favour of shorter scaffolds.

Split misassemblies. The mentioned misassemblies were split up to improve the clustering. This splitting is done manually and is not computed. Therefore the diagonal of the first map (figure 1) was viewed in detail. Every Scaffold, where the "heat" instantly cools down to a minimum can be assumed as a failure of the assembly, because it is impossible that there are suddenly no chromatin interactions among one scaffold. For example the longest scaffold "talOcc3_1" has a misassembly as shown in figure 3. The arrows show the cutting of this scaffold. Thus three new scaffolds are formed and can be clustered individually.

Nine scaffolds were cut and ten new scaffolds were formed that way (the first scaffold was cut twice).

These workarounds and solved problems helped to achieve the goal of this clustering: grouping of chromosomes. As a reminder, this clustering can also be computed separately from the following simulated annealing algorithm.

The following part presents the second method to improve a genome assembly with Hi–C data.

9. Simulated Annealing

Finding an optimal solution for this problem, every possible scaffold ordering has to be tried out. By chance, the first tries are already perfect, but that is nearly impossible, so

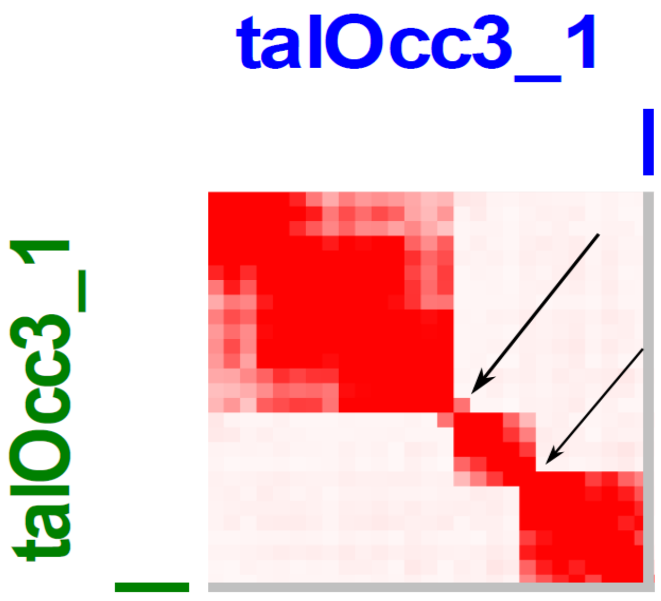


Figure 3: Cut out from figure 1. The first scaffold points out the misassembly
Visualized by Juicebox [8]

that in the worst case every option will have to be tested. An order of length n has $n!$ possibilities. With 59 scaffolds to order, $59! = 1.3868312e^{80}$ possibilities exist to find the optimal solution. Even if a program can check hundred million options in under one second, it will need thousand of years to finish. So the question is how to achieve an approximate solution in a shorter time.

9.1. Concept

Simulated annealing is a probabilistic approach to find an approximated global optimum of a optimization problem. Optimization problems can not be computed directly, due to exponential complexity.

Simulated annealing is inspired by annealing in metallurgy. A controlled cooling of material reduces the number of dislocations, because the atoms need time to order and form stable crystals. When the temperature is high, the speed of the atoms is also high and they can dissolve from a crystal. When the temperature lowers, the atoms change to the energetically advantageous crystal. Atoms have less energy, if the temperature decreases and so they can not redissolve from that crystal. The crystal begins to grow. A controlled annealing decreases the thermodynamic free energy, increases the ductility of the material and makes it easier to process the material further. [10]

In this optimization technique the temperature is simulated by a probability. Interim results may change for the worse. This deterioration is necessary to leave a local optimum.

Otherwise the simulation will not find the global optimum, it will remain in the first local optimum and can not leave it again. So the probability to accept a worse solution simulates the behaviour of the atoms during a controlled annealing process.

The probability, that decides if an "atom" is allowed to dissolve from a "crystal", is calculated with the Boltzman distribution:

$$p = e^{(-\frac{\Delta E}{T})} \tag{3}$$

ΔE describes the energy difference and T the temperature. The greater the difference and the smaller the current temperature, the smaller is the chance p to accept a deterioration. The temperature T chills in every step. First, the annealing is very fast and it slows down as the temperature lowers. Equation (4) shows the chilling in every step i.

$$T_i = T_0 \cdot a^i \tag{4}$$

$$a = \left(\frac{T_n}{T_0}\right)^{\frac{1}{n}} \tag{5}$$

T_0 is the start temperature and a describes the cooling factor. With these variables the cooling can be controlled precisely. Equation (3) and (4) simulate the annealing process in metallurgy perfectly and can be applied to different optimization problems. To calculate an adequate cooling factor a, the end and start temperature should be known. Also the number of steps is important to provide, that the desired end temperature can be achieved. Otherwise, the annealing process is either to short or to long and that is in both ways not desirable. Equation (5) shows how an appropriate cooling factor can be accomplished, where n is the maximum count of steps.

9.1.1. Idea

The next task is to apply this method to the optimization problem of the falsely ordered scaffolds. Simulated annealing is the basic structure, the wrapper, but to order and orient scaffolds, some specific operations are being needed. So the idea is to do one operation per step and check, whether it changes for better or for worse. Afterwards, let the algorithm decide, if the change will be accepted or not.

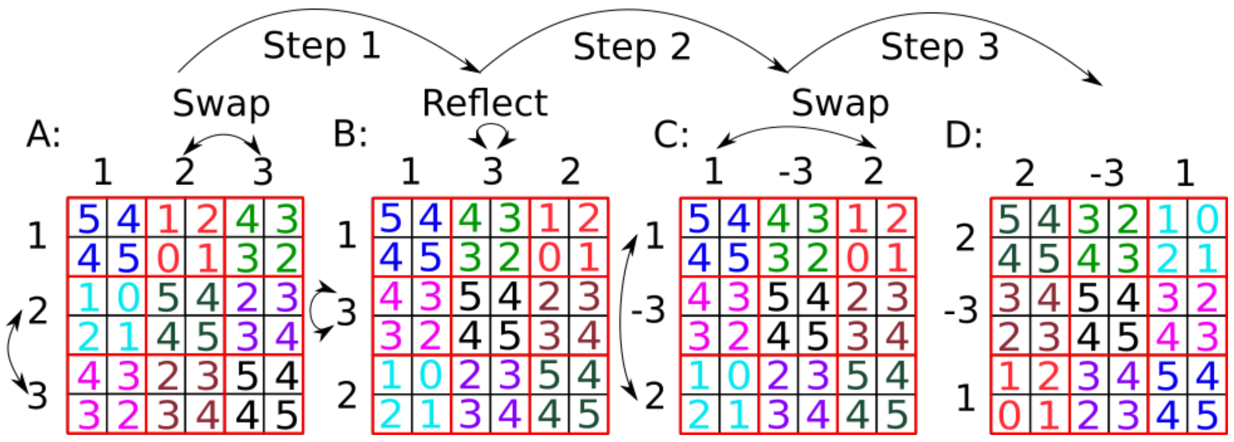
To accomplish this, three main operations are needed. A function has to score the given state, otherwise it will not be possible to distinguish between a deterioration and an improvement. Changing the order of scaffolds has to be maintained by another function. Also a change of the orientation of one scaffold has to be implemented. In every step of the simulated annealing algorithm either two scaffolds will replace each other or one scaffold will be mirrored in its orientation. Then the scoring function compares the previous and the current state of the scaffolds and the simulated annealing algorithm will accept or reject it.

In the following, the concept will be explained further and proven on a detailed example, before the implementation will be fully described.

9.1.2. Proof of concept

The Hi–C map, as shown in section 5, figure 1, is a matrix of matrices. One inner matrix is defined by two scaffolds, where each axis describes one scaffold. The scaffolds are folded into bins. So, every entry is another matrix and these matrices contain the chromatin contacts, respectively the distances between two bins of these scaffolds.

Figure 4 is a reduced example. It can be expanded into any possible size. Also, the inner matrices have different sizes in reality, because the scaffolds have different sizes. The shown matrices have three scaffolds with two bins. One matrix contains nine 2×2 matrices. The outer matrix is a 3×3 matrix.



$$A:\ \begin{array}{c|cc|cc|cc} & 1 & & 2 & & 3 & \\ \hline 1 & 5 & 4 & 1 & 2 & 4 & 3 \\ & 4 & 5 & 0 & 1 & 3 & 2 \\ \hline 2 & 1 & 0 & 5 & 4 & 2 & 3 \\ & 2 & 1 & 4 & 5 & 3 & 4 \\ \hline 3 & 4 & 3 & 2 & 3 & 5 & 4 \\ & 3 & 2 & 3 & 4 & 4 & 5 \end{array} \qquad B:\ \begin{array}{c|cc|cc|cc} & 1 & & 3 & & 2 & \\ \hline 1 & 5 & 4 & 4 & 3 & 1 & 2 \\ & 4 & 5 & 3 & 2 & 0 & 1 \\ \hline 3 & 4 & 3 & 5 & 4 & 2 & 3 \\ & 3 & 2 & 4 & 5 & 3 & 4 \\ \hline 2 & 1 & 0 & 2 & 3 & 5 & 4 \\ & 2 & 1 & 3 & 4 & 4 & 5 \end{array}$$

$$C:\ \begin{array}{c|cc|cc|cc} & 1 & & -3 & & 2 & \\ \hline 1 & 5 & 4 & 4 & 3 & 1 & 2 \\ & 4 & 5 & 3 & 2 & 0 & 1 \\ \hline -3 & 4 & 3 & 5 & 4 & 2 & 3 \\ & 3 & 2 & 4 & 5 & 3 & 4 \\ \hline 2 & 1 & 0 & 2 & 3 & 5 & 4 \\ & 2 & 1 & 3 & 4 & 4 & 5 \end{array} \qquad D:\ \begin{array}{c|cc|cc|cc} & 2 & & -3 & & 1 & \\ \hline 2 & 5 & 4 & 3 & 2 & 1 & 0 \\ & 4 & 5 & 4 & 3 & 2 & 1 \\ \hline -3 & 3 & 4 & 5 & 4 & 3 & 2 \\ & 2 & 3 & 4 & 5 & 4 & 3 \\ \hline 1 & 1 & 2 & 3 & 4 & 5 & 4 \\ & 0 & 1 & 2 & 3 & 4 & 5 \end{array}$$

Figure 4: Concept – Order and orient scaffolds
3×3 matrix contains nine 2×2 matrices.
Each matrix has its own colour. Matrix B, C and D
show interim results after an operation.

The first matrix A of figure 4 shows the start matrix. Here, columns and rows are ordered from left to right, respectively from up to down. The following matrices show interim results after every operation and the columns and rows keep their numeration from matrix A. So, in the first step, scaffold 2 and 3 change places. That means, at first, column 2 and 3 will be swapped, followed by row 2 and 3, to ensure the symmetric characteristic of the matrix.

The next operation mirrors a scaffold with itself, which means, that every matrix in that particular row and column will be reflected, shown in *Step* 2. Matrices in the row of that scaffold will reverse their columns and matrices in the column will reverse their rows, otherwise the new outer matrix would not be symmetric any more. In figure 4, a mirrored scaffold is marked with a "–".

With another swap operation, a perfect result can be achieved, as it can be seen in the last matrix D. The further a number is away from the diagonal, the smaller they become. The smallest numbers, here 0, should be the farthest away from the diagonal. In this example, both zeros are very close to the diagonal in the beginning, but in every next step they slowly drift to the edge.

In figure 5, a heat map of the last matrix, from figure 4 is illustrated. The columns and rows are numbered like the matrix and, as it can be seen, the heat is smoothly decreasing to the left lower and right upper edges. This illustration demonstrates the perfect solution. But in this particular case, there are more ways to reach such a solution: For example, if after the last swap operation, another swap is made, where scaffold 1 and 2 change positions, the heat map and matrix would be the same. But this matter only occurs, where all scaffolds have the same length and must not be considered further in reality. This matter of fact is only mentioned for the sake of completeness, it is not relevant for the concept.

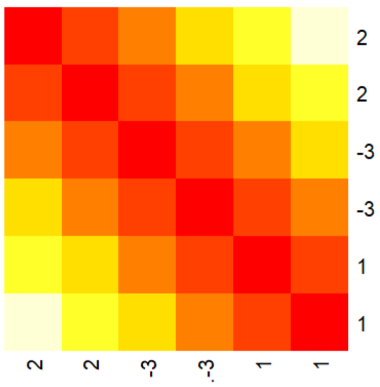


Figure 5: Heatmap of the last matrix from figure 4
Illustrates the optimal heat flow. The Hi–C map should look like this in an optimal solution.

Of course, this example is staged and the simulated annealing algorithm has to do the operations randomly, but this example should help to understand, what an optimal solution of this problem could look like. However, this example does not show the decision, in which the newly formed matrix is accepted, because that requires a function, that can compare two matrices. This function should penalize high heat more, the further it is away from the diagonal, and less on, or almost on the diagonal. A quadratic function dependent on the distance to the diagonal is capable of doing so, because the higher the range, the higher the score.

A major benefit using such a function is that different possibilities can be tested easily by changing only one parameter. The gradient of that function determines, how high the penalty score will be and how fast the penalty will rise in relation to the distance from the diagonal. In equation (6) the parameter a defines exactly this gradient, because if, for example, $a = 1$ the penalty is only determined by the size of the value v, but if $a < 1$ the penalty would be eased. The same applies to $a > 1$ the other way round, the penalty would be enforced.

$$s_{i,j} = a \cdot (|i - j|)^2 \cdot v_{i,j} \tag{6}$$

Equation (6) calculates the score of point (i, j) in the matrix, where s is the calculated score, a determines the increasing or decreasing, $(|i - j|)$ describes the distance to the diagonal and v is the value at point (i, j). All scores of all entries sum up to the score of the whole matrix. That sum makes matrices comparable and makes it possible to distinguish between a better or worse state.

Now, all operations and functions are theoretically explained and the implementation of those can described. This section presented the considered concept for the optimization problem and showed an example with an optimal solution. Also, the scoring function, (6), was introduced and described.

The implementation of this concept design will now be described in detail. Every Operation will be examined carefully and a detailed runtime analysis will be made. But first, the further process of the data will be explained, because, as mentioned in section 7.2, a suitable data structure needed to be found and the data had to be in the right format to apply it to this structure.

9.2. (Pre–) process the data

The first step of the process is obvious: The algorithm needs a matrix to work with, because the tables, as described in 7.1, are not functional for this task. The x and y axis of this matrix are the bins and an entry is the distance between these axis. The distances were inverse, so the highest value equalled the smallest distance. This matrix is the first input of the program, but it is not yet formed into scaffolds, just bins.

More information is required to split this matrix into scaffolds and form a matrix of matrices, so another file, that is able to tell, when the last bin of a scaffold is reached, is necessary. With these two files, the data structure, a matrix of matrices as a 4 dimensional array, can be built with the right entries.

Another file can contain an alternative initial sorting of scaffolds, for example after the clustering. If the algorithm should not accept worse states, this could be a good initialization. It is not necessary if the simulated annealing algorithm should accept deteriorations, because the sorting will be burst instantly.

The output is a lot more different, because, for example, the result matrix is not needed. In every step, the algorithm tracks, whether a swap or a reflect is done and stores the change in an associated list. The order–list tracks an accepted swap by swapping the list entries of the same indices like in the real swap. The reflected–list is initialized with zeroes and on every reflect a XOR–operations with 1 is made. If a scaffolds experience a re–reflect, the related list entry is set to 0 again. Of course, the reflect-list has to track not only reflects, but also swaps, otherwise the list will be useless in the end. These two lists are written into files, where they can be processed further. As shown in the flow chart of section 7.1 figure 2, another R–Script helps to visualize the output.

In conclusion, starting with one matrix, scaffold sizes and an initial sorting of scaffolds as input files, the output are only two files with tracked swaps and reflections. In addition, all scores and temperatures can be written to files, too.

9.3. Implemented Functions

In the following, the main algorithm, respectively the simulated annealing algorithm will be demonstrated. Also the major operations, like the scoring, the swap and the reflect will be analysed form a practical point of view. The runtime will be analysed further and in the end, possible improvements of the implementation will be introduced.

9.3.1. Main Algorithm

Listing 1 shows the algorithm in pseudo code with inspired syntax from the programming language Python. This listing is kept simple and functions like *accept()* or *reject()* are not real, they just represent the task in a short way.

The first input parameter n, besides the matrix, determines the size of the for–loop. Parameter a defines the slope of the scoring function and equals the a from equation (6). The *chance* is the probability of a swap, respectively of a reflection and with *end_annealing* the annealing function, based on n, can be computed.

Before the program arrives at the loop, a few pre-calculations have to be made. At first the scoring of the initial matrix is needed to compare it to upcoming changes. The next step is to calculate the curve of the annealing process. As mentioned, the last parameter defines, at which temperature the annealing process should stop. With parameter n, the exact annealing curve can be calculated, because the temperature always starts at hundred degrees.

Now the loop can begin: At first, two random integers in an interval between 0 and the number of scaffolds will be stated. These numbers will be needed later, because they determine the scaffolds, that should switch places or which scaffold should be mirrored. Next, the decision, if either a swap or a reflect will be processed, has to be made. Therefore, another random number between 0 and 1 is chosen and compared to the input variable *chance*. The higher this variable, the more often a swap, instead of a reflect, will be computed. After a matrix changing operation, the new score of this matrix has to be calculated. If it has improved, the old matrix will be discarded and the loop begins at the top. If it is a deterioration, the probability of accepting this matrix has to be computed or it will be rejected and the new matrix will be discarded. The probability to accept a deterioration is greater, as mentioned in equation (3), when the difference between the new and the old score is small. So, a matrix with a much worse score will be hardly be accepted and the lower the temperature, the more unlikely it is to accept a worse state. That means, that the probability, that the algorithm already found the global optimum, is getting higher, as the temperature, respectively the number of loops left, decreases.

The last step is to decrease the current temperature about a small percentage according to the calculated cooling factor *annealing*, as was shown in equation (5). Once the loop reaches its end, the last thing to do is to return both lists that tracked all final changes during the annealing process. These tracks are, for the sake of convenience, not included in listing 1.

```
def simulated_annealing(matrix, n, a, chance, end_annealing):
    score = scoring(a)  # compute the first scoring
    old_score = score   # save the score
    # start temperature is always 100 degrees
    start_temp, temp = 100, 100
    # calculate annealing function
    annealing = (end_annealing/start_temp)**(1/n)
    for i in range(0, n):
        n1, n2 = random(0, scaffold_num)
        # calculate two random numbers
        # between 0 and numbers of scaffolds
        if probability(chance):
            swap(matrix, n1, n2) # do the swap
        else:
            reflect(matrix, n1) # do the reflect
        # compute the new score
        score = scoring(matrix, a)
        # accept if better or a deterioration is accepted
        if score < old_score orzufällige zahl gezogen
            probability(-(score - old_score) / temp)):
            accept(matrix)
        else: # reject otherwise
            reject(matrix)
        # cool down!
        temp = start_temp * annealing**i
return reflected, order
```

Listing 1: Simulated Annealing Algorithm in pseudo code

After the main algorithm was presented, the next important function is the calculation of the score. In equation (6) the scoring is described in theory, but there are a few mentionable details in the implementation.

9.3.2. Scoring

To score a matrix in the mentioned manner, the scoring equation (6) has to be applied on every entry. Since the matrix is symmetric, it is only necessary to calculate either the upper or the lower matrix and multiply the result with two, because lower and upper are equal. Listing 2 shows the implementation of the scoring function and as shown, the second for–loop restricts the calculation only to the upper matrix, which is more efficient than to calculate the whole matrix. The implementation contains a matrix of matrices, which is a four dimensional array. So, if the loop is currently on a matrix that lays on the diagonal of the whole matrix, only the upper matrix needs to be added to the score. The actual diagonal is always set to 0 and it is not necessary to score these, because the values on the diagonal will always be on the diagonal.

If the current looped matrix is not on the diagonal, the length of all matrices between this and the diagonal matrix of the current row needs to be added to the distance. In the pseudo code below, Listing 2, the most inner for–loop with index h computes this length. Afterwards, the length of the diagonal matrix and the row of the current value $matrix[i][j][x][y]$ needs to be added too. Then the score of this value can be computed according to the mentioned quadratic function (6). The final step is to add it to the whole score.

```
def scoring(matrix, a):
  score = 0
  for i in range(0, len(matrix)):
    for j in range(i, len(matrix)): # only upper matrix
      for x in range(0, len(matrix[i][j])):
        for y in range(0, len(matrix[i][j][x])):
          distance = 0
          if i == j: # on diagonal
            if y > x:
            # upper matrix of diagonal matrix
              distance = abs(x - y)
          else:
          # length between diagonal
          # matrix and current matrix
            for h in range(i + 1, j)):
              distance += len(matrix[i][h][x])
            # length of diagonal matrix
            diagonal = len(matrix[i][j][x] - (x + 1))
            row = y + 1 # current row
            # this is the complete distance to the diagonal
            distance += row + diagonal
          score += a * (dist**2) * matrix[i][j][x][y]
  return score * 2 # multiply with 2 to score the whole matrix
```

Listing 2: Scoring function in pseudo code

The input parameter a defines the gradient of the scoring function and is the same as in equation (6). The actual implementation differs from the therotical equation, because the data structure contains not 2 indices, but 4 and the distance is not so easily computable.

9.3.3. Swap

Although the implementation of the swap function is as trivial as in theory, it is worth to showing. On the one hand for the better understanding of the runtime analysis, which will be performed later and on the other hand to present all main operations that are called in the simulated annealing algorithm. Listing 3 has the two input parameters, that are the randomly computed values in the main algorithm, listing 1.

At first, the rows will be switched. Then, the columns of both scaffolds need to be computed additionally, afterwards, every matrix of the columns will be swapped individually.

Switching the rows and the columns is necessary to keep the symmetric characteristic of the matrix. Every matrix on the diagonal must always be on the diagonal.

```
def swap(n1, n2):
  # swap rows
  matrix[m1], matrix[m2] = matrix[m2], matrix[m1]

  # get column n1 and n2
  column1, column2 = get_column(n1), get_column(n2)
  for i in range(0, len(col1)): # col1 and col2 have equal length
    # swap columns
    matrix[i][n1] = col2[i]
    matrix[i][n2] = col1[i]
  return matrix
```

Listing 3: Swap function in pseudo code

get_column returns the column $n1$, respectively $n2$ and is not shown here. The for–loop loops through all rows of every matrix, though it does not matter if the restriction is $col1$ or $col2$, because every column has the same length and contains matrices of every scaffold.

The last function is the reflection of one scaffold, the implementation of this is not as trivial as the swap operation and will be presented next.

9.3.4. Reflect

Besides the swap operation, the reflect operation is another possibility to improve the score of the matrix. At first sight, it is actually the same: Reflecting rows and columns. But the reflection between column and row differs. A matrix in the column needs to be reflected by row and a matrix in the row needs to be mirrored by its columns. With these two reflections, the matrix on the diagonal will be reflected once by row and once by column and will be the same as before, so that the symmetry is guaranteed.

Listing 4 shows the implementation with help functions *get_column_array*, *get_column* and *reverse*. The first function needs an ordinary matrix as input and returns this matrix with reflected columns, respectively every column array of that matrix will be reversed. Secondly, the *get_column* function returns all matrices of a specific column from the whole matrix. Last but not least, *reverse*, as its name implies, reverses the given array.

```
def reflect(matrix, n1):
  # reflect rows
  for i in range(0, len(matrix)):
    matrix[n1][i] = get_column_array(matrix[n1][i])
    # this function returns the a matrix reflected by columns

  # reflect columns
  column = get_column(n1)
  for i in range(0, len(matrix)):
    for j in range(0, len(matrix[i][n1])):
      matrix[i][n1][j] = reverse(column[i][j])#
      # this reflects a matrix by row
  return matrix
```

Listing 4: Reflect function in pseudo code

Now that all major operations have been presented and explained in detail, the runtime analysis can begin. In the following the runtime of every operation will be analyzed.

9.4. Runtime analysis

Analysing an algorithm includes mostly the analysis of runtime and storage, but in this case the input is huge and storage was not a limiting factor. Only the runtime of the algorithm will be inspected further. A runtime analysis examines the algorithm by the amount of time required to execute, depending on the size of the input. This is also called time complexity of an algorithm. If the algorithm behaves differently depending on the input, the worst case input will be assumed and the complexity is described as an upper bound of the algorithm. Further, an algorithm is called efficient, if a runtime can be achieved in polynomial time, that means the upper bound complexity is not greater as a polynomial function[11]. This analysis will examine all presented operations individually, followed by the whole algorithm. A favourable result is, of course, a polynomial runtime. Firstly, the scoring function will be analysed, followed by the swap and reflect functions.

The scoring function loops through a $m \times m$ matrix, but because of the symmetry of the outer matrix, it only has to loop through the upper matrix. Because the inner matrices have different sizes and the function loops through all matrices on the diagonal completely, the runtime depends on the varying factor v, where v are all values of the lower matrix that are contained in a diagonal matrix. The runtime is $\frac{1}{2} \cdot m^2 + v$, where the factor v has to be added to all values of the upper matrix. So, in big O notation the runtime is in $\mathcal{O}(m^2)$.

Analyzing the swap operation will be much easier, because it does not loop through single values, but through matrices. On the one hand, the function consists of a swap of rows, which can be computed, due to data structure benefits, in constant time. On the other hand, it consists of a swap of columns, which more difficult. The data structure does not provide to select all matrices of one column, so every matrix of one column has to be selected by itself. In a swap, two columns have to be swapped, which means, that both columns need to be looped. That takes s steps for every column, where s is the number of scaffolds, respectively the size of the outer matrix. In conclusion, the runtime is $3s$, which is in $\mathcal{O}(s)$.

Last but not least, the reflect operation needs to be analysed, which is the most complex function. First, the row of the scaffold will be reflected, so that all columns in a matrix of that row have to be reversed. When the vertical axis of this row has y entries and the horizontal axis has a total length of m (length of the outer matrix), the runtime to reverse all columns is then $\frac{1}{2} \cdot y \cdot m$. To reverse an array, only half of it has to be looped. Reflecting a column is very similar, because all rows of the column have to be reversed. The runtime depends on the entries of the horizontal axis x. This equals $\frac{1}{2} \cdot x \cdot m$. The total runtime would be $\frac{1}{2}m \cdot (x + y)$. Because $x < m$ and $y < m$ is given, the runtime is in $\mathcal{O}(m)$. It is important to note, that this runtime is only valid in theory and the implemented runtime is slightly higher, due to data structure reasons. This will be explained further in section 11.2.

Concluding, to calculate the total, the runtime of all operations has to be added. But since either a reflect or a swap operation will be called in one step, the runtime can only be approximated. Assuming, that a reflect and a swap will have the same probability to be called, the runtime would be $n \cdot \frac{1}{2} \cdot (m + s^2)$, where n is the number of steps, this simulated annealing algorithm should take. In big O notation, this runtime is in $\mathcal{O}(n \cdot (m + s^2))$.

In part II all applied methods to improve a genome assembly were explained. First the hierachical clustering analysis, and second the simulated annealing algorithm. The main operations were described in detail with an eye on the runtime of each operation. The results will be presented in the following part.

III. Results

The results will be shown chronically, starting with interim results of the clustering and final results. Afterwards, the results of the simulated annealing will be presented. In Section 11.1, the temperature and scoring behaviour during an execution of the simulated annealing will also be explained.

10. Clustering Results

The first clustering analysis was computed, before the misassemblies were split up. Figure 6 shows the Hi–C map, where the scaffolds are sorted according to the order of this clustering. It shows that many scaffolds are grouped into distinguishable domains with a smoothly decreasing "heat". Still, the Hi–C map illustrates, that some scaffolds could not be clustered as desired. The misassemblies, as shown in section 8.3.1, figure 3, are now clearly visible, since the "heat" changes alternately in these domains.

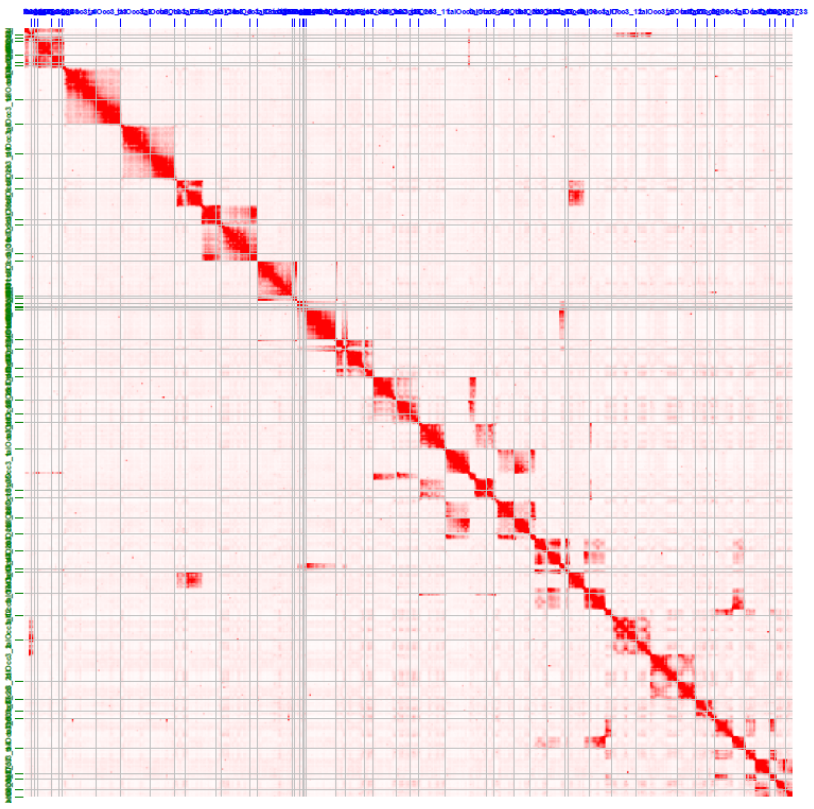


Figure 6: Hi–C map of the first clustering of the longest 49 scaffolds
Visualized by Juicebox [8]

With these falsely assembled scaffolds, a continued work would not give favourable results, as figure 6 shows. After the misassemblies were split in more scaffolds, the same cluster analysis provided a far better result because the splitted scaffolds could be clustered to different domains. The Hi–C map presents, as shown in figure 7, that all scaffolds can be clustered into distinguishable groups. Anyway, it still shows some spikes far from the diagonal and though the heat flow is smoother, it is still not at the optimum.

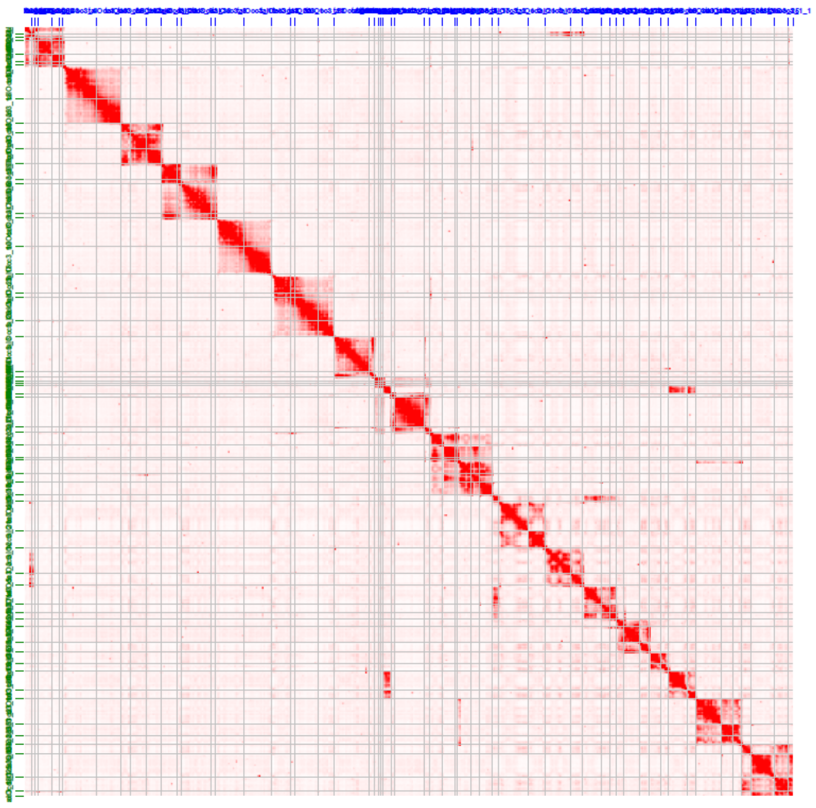


Figure 7: Hi–C map of the second clustering after the split up
Visualized by Juicebox [8]

Since only the longest 49 scaffolds of the "talOCc3" assembly could be computed, there are not 19 visible domains, but 17. If all scaffolds had the similar length, all scaffolds could be computed and the domains could probably be interpreted as chromosomes.

In the following section the results of the simulated annealing algorithm will be presented.

11. Simulated Annealing Results

The result of the simulated annealing algorithm shows that the scaffolds are grouped into domains and contacts align near the diagonal. Of course, the more steps the simulated annealing algorithm computes, the better the domains are distuingishable. To compute the best approximation, the number of steps increases exponentially, because the probability, to do an operation, which is an improvement, decreases significantly with every step.

An important fact is, that a few domains that are emerged from the simulated annealing algorithm, are equal to the domains from the clustering as shown in figure 7.

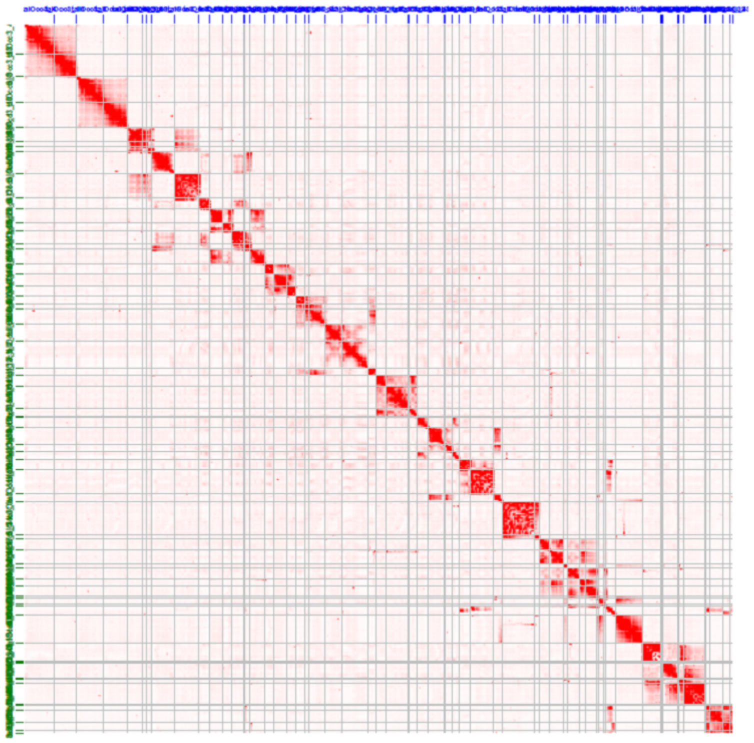


Figure 8: Hi–C map after 50.000 steps
Visualized by Juicebox [8]

Figure 8 shows the result after 50.000 steps. The “heat” clearly arranges around the diagonal, but it can be seen that the “atoms”, did not find the energetically most favorable “crystall” yet, respectively the contacts are not yet ordered to the definite domain. If the “atoms” have more time to cool down, the domains will be much more distuingishable, respectively the simulated annealing has to compute more steps.

The Hi–C map in figure 8 shows the result after a normal execution of the simulated annealing algorithm. As mentioned before, the simulated annealing can also be computed after the clustering. So, the initial ordering of the matrix equals to the ordering from the clustering and no deteriorations will be accepted anymore. Otherwise, the calculated ordering of the cluster will burst immediately. With not accepting worse states, the initial ordering can now be improved by the simulated annealing algorithm. Figure 9 shows a heat map, where no deteriorations were made. This is the result after 500.000 steps and it can be seen that this map has a worse state then figure 8, even though the beginning matrix was ordered like the clustering, figure 7 and 10 times more steps were computed.

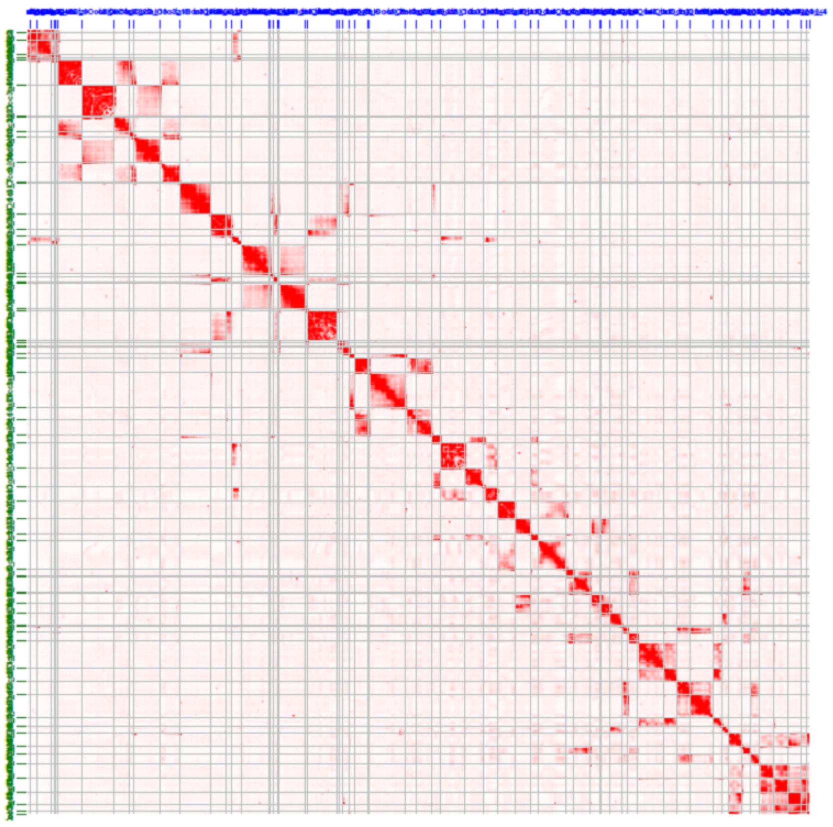


Figure 9: Hi–C map after simulated annealing without deteriorations
The initial ordering is from the clustering
Visualized by Juicebox [8]

11.1. Temperature & scoring behaviour

The matrix, during the simulated annealing algorithm, changes a lot. These changes can be visualized as a plot of the scoring function. Figure 10 shows a plot of all scores in 50.000 steps. In the beginning, the score decreases very quickly, then an improvement will only have small impact on the score, because the contacts are already near the diagonal. Also, it can be seen that a much worse state will not be accepted anymore, only these, where the score is slightly higher than before (see equation (3)).

Figure 11 shows the behaviour of the temperature. Due to the presented temperature function in section 9.1 equation 4, the cooling curve is always the same, regardless of the number of steps the simulated annealing algorithm computes. On the one hand, this is a great benefit, because it is not possible for the cooling to be too fast or too slow. On the other hand, it is a disadvantage, because if the simulated annealing algorithm should be computed with an initial ordering, this calaculation could benefit from a low beginning temperature. A low temperature will allow only slightly worse states and the results could be better than prohibiting any deteriorations.

In the next part IV the methods and results will be discussed.

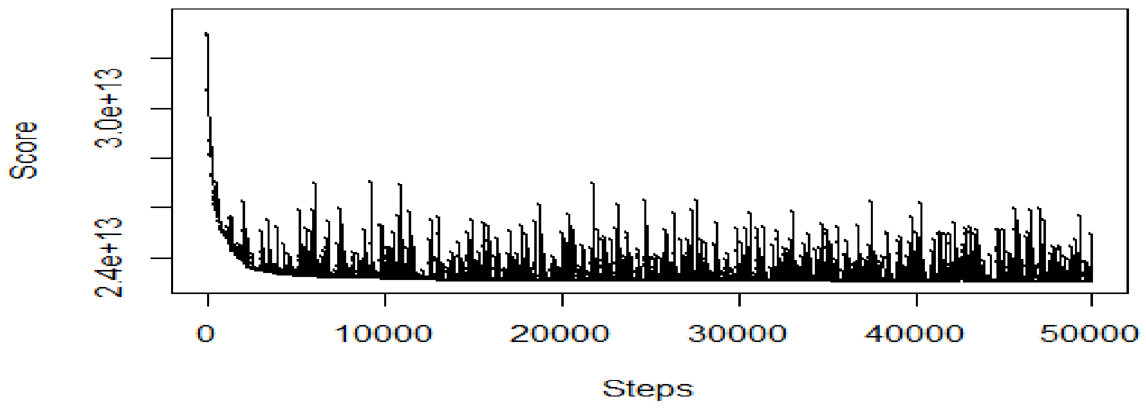


Figure 10: Scoring behaviour in 50.000 steps
Plot made with R

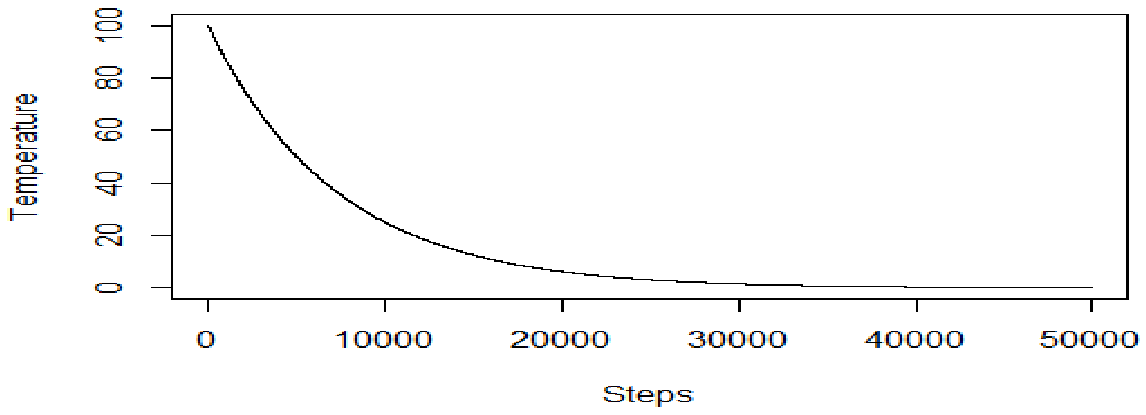


Figure 11: Temperature behaviour in 50.000 steps
Plot made with R

IV. Discussion & Outlook

11.2. Discussion

This section is basically split into three parts: First, the clustering will be discussed, followed by possible improvements of operations or data structure of the simulated annealing algorithm and finally, the results of both methods will be reviewed.

As mentioned in section 8.3.1 only the 49 longest scaffolds have been taken into account. Of course, instead of just cut them and discard the smallest, more effort in the normalization of the data could be worthwhile.

Considering only the 49 longest scaffolds, much data got lost and the genome assembly is not complete. Nevertheless, the effective use to improve an assembly with a cluster analysis on Hi–C data could be proven. It has yet to be shown, how this approach would behave, using a different data set with more equal scaffold length.

Misassemblies, which could be observed, see section 8.3.1, had to be splitted. Since this was done manually, the splitting may not be exact. The clustering results show that the splitting was a necessary step, because these splitted scaffolds could be aligned to different domains. Anyway, it should be done more exactly, using, for example, computational methods to achieve better results.

In the simulated annealing algorithm also some possible improvements can be made to achieve better and especially faster results. To improve the code, every operation will be viewed separately. Of course, not every operation can be improved, since some improvements, that make the implementation more efficient may have impact on the result. Further, some possibilities will be discussed.

A possibility may be to improve the scoring function, by looping only through the upper matrix without the diagonal matrices. The runtime would be exactly $\frac{1}{2} \cdot m^2$. This is only possible because the values on a diagonal matrix do not change and can be set to zero. However, the score would be smaller possibly without influencing the result.

Furthermore, the scoring function could penalize in a different way. A threshold, for example, could determine a maximal penalty, which may leads to a smaller scope of the Boltzman distribution, see equation (3), thus might leading to a more exact result. This may also provide better results for the simulation without accepting worse states. The spikes on the edges in figure 7 have suche a huge impact on the score, that the score is higher than in figure 9, althoug figure 7 looks better.

In the reflect operation there are the most possibilities for improvement. On the one hand, all changes can be calculated in–place, that would be more efficient in terms of storage saving. On the other hand, a different data structure, that can select rows and columns independently and in constant time, may be an option. This could improve the runtime significantly. The currently used data structure only selects columns by rows, which is inefficient when all entries of a column are needed.

For the sake of completeness, it is necessary to say that an implementation of the simulated annealing algorithm in the programming language C could be faster and could result in a decreasing of absolute runtime. However, this goes hand in hand with an increased development time, due to memory allocation etc.

In conclusion, the figures in part III show that an approximated solution, which improves the genome assembly, could be achieved. This approximation can be further improved by, for example, starting the simulated annealing algorithm again using the result order of a previous simulation. However, the resulting Hi–C maps, prove that this thesis has accomplished its aims. The false ordering and orientation of scaffolds could be improved with a simulated annealing algorithm. Though a good approximation has been reached, further improvements would provide even better results.

Section 11.3 gives an outlook of this work.

11.3. Outlook

This thesis is an approach to improve a genome assembly based on Hi–C data and proves that such an improvement can be approximated with an optimization technique called simulated annealing. This approach is, of course, not limited to the Iberian mole, and can be applied to any other species of which a genome assembly and associated Hi–C data exist. In section 11.2 many possibilities for improvement concerning the single steps of the process in this work were presented.

A more general idea might be an improvement of a genome assembly step by step: First, clustering all scaffolds and then only compute single domains with the simulated annealing algorithm. This is especially usefull, when the domain contains many small scaffolds. The ordering within such a domain can be improved further with the simulated annealing algorithm. After all domains are computed, they can be put together as the whole genome assembly, where each domain can be identified as a chromosome.

Anyway, this thesis will hopefully inspire future work on improving a genome assembly with Hi–C data.

References

[1] http://www.ngfn.de/de/verstehen_der_menschlichen_erbsubstanz.html [23.08.2017, 10 : 16]

[2] Lieberman-Aiden E, van Berkum NL, Williams L, et al. Comprehensive mapping of long range interactions reveals folding principles of the human genome. Science (New York, NY). 2009

[3] Van Berkum NL, Lieberman-Aiden E, Williams L, et al. Hi-C: A Method to Study the Three-dimensional Architecture of Genomes. Journal of Visualized Experiments : JoVE. 2010

[4] Suhas S.P. Rao, Miriam H. Huntley, Neva C. Durand, Elena K. Stamenova, Ivan D. Bochkov, James T. Robinson, Adrian L. Sanborn, Ido Machol, Arina D. Omer, Eric S. Lander, Erez Lieberman Aiden. A 3D Map of the Human Genome at Kilobase Resolution Reveals Principles of Chromatin Looping. Cell 159 2014

[5] Kaplan N, Dekker J. High-throughput genome scaffolding from in-vivo DNA interaction frequency. Nature biotechnology. 2013

[6] Dudchenko, O., Batra, S.S., Omer, A.D., Nyquist, S.K., Hoeger, M., Durand, N.C., Shamim, M.S., Machol, I., Lander, E.S., Aiden, A.P., et al. (2017). De novo assembly of the Aedes aegypti genome using Hi-C yields chromosome-length scaffolds. Science eaal3327. Published online ahead of print on March 23, 2017.

[7] Neva C. Durand, Muhammad S. Shamim, Ido Machol, Suhas S. P. Rao, Miriam H. Huntley, Eric S. Lander, and Erez Lieberman Aiden. "Juicer provides a one-click system for analyzing loop-resolution Hi-C experiments." Cell Systems 3(1), 2016.

[8] Neva C. Durand, James T. Robinson, Muhammad S. Shamim, Ido Machol, Jill P. Mesirov, Eric S. Lander, and Erez Lieberman Aiden. "Juicebox provides a visualization system for Hi-C contact maps with unlimited zoom." Cell Systems 3(1), 2016.

[9] Servant N., Varoquaux N., Lajoie BR., Viara E., Chen CJ., Vert JP., Dekker J., Heard E., Barillot E. HiC-Pro: An optimized and flexible pipeline for Hi-C processing. Genome Biology 2015, 16:259 doi:10.1186/s13059-015-0831-x http://www.genomebiology.com/2015/16/1/259

[10] Rob A. Rutenbar Simulated Anneling Algorithms: An Overview. IEEE Circuits and Devices Magazines January 1989

[11] Prof. Dr. Uwe Schöning. Theoretische Informatik – kurz gefasst. Kapitel 3, 5. Auflage 2008 Spektrum Akademischer Verlag Heidelberg.

List of Figures

Listings