UPTEC X 10 021
Examensarbete 30 hp
November 2010

Clustering of DNA sequence reads from repeat regions using defined nucleotide positions (DNPs)

Lennie Fredriksson

UPPSALA UNIVERSITET

Molecular Biotechnology Programme

Uppsala University School of Engineering

UPTEC X 10 021	**Date of issue 2010-11**

Author

Lennie Fredriksson

Title (English)

Clustering of DNA sequence reads from repeat regions using defined nucleotide positions (DNPs)

Title (Swedish)

Abstract

Sequencing genomes with a high frequency of repeat regions is a difficult task.
The aim of the project was to develop an algorithm to speed up the sequencing process of highly repetitive genome. By using specific differences between the repeats called defined nucleotide positions (DNPs), cluster DNA sequence reads into contigs. The strategy used in the development of the algorithm resulted in a quite complex algorithm. Test runs of the algorithm showed that there is still work to be done to get a desirable result.

Keywords

DNP, repeat regions, SolidClusters, algorithm

Supervisors

Erik Arner
Karolinska Institutet, Center for Genomics and Bioinformatics

Scientific reviewer

Siv Andersson
Uppsala Universitet, Evolutionsbiologiskt Centrum

Project name	Sponsors
Language **English**	Security
ISSN 1401-2138	Classification
Supplementary bibliographical information	Pages **45**

Biology Education Centre Biomedical Center Husargatan 3 Uppsala
Box 592 S-75124 Uppsala Tel +46 (0)18 4710000 Fax +46 (0)18 471 4687

Clustering of DNA sequence reads from repeat regions using defined nucleotide positions (DNPs)

Lennie Fredriksson

Sammanfattning

Detta arbete beskriver skapandet av en algoritm som är tänkt att användas vid sekvensering av parasiten *Trypanosoma cruzi's* genom. Det specifika med *T. cruzi's* genom är att det innehåller en ovanligt hög andel repetitivt DNA. Detta repetitiva DNA är mycket svårt att sekvensera med dagens sekvenserings-algoritmer. Det som försvårar arbetet är likheten som dessa repetitiva sekvenser har. För att lyckas med sekvenseringen måste man hitta skillnader hos de repetitiva delarna och sedan använda dessa för sekvenseringen.

Genom att hitta de små skillnader, sk. naturliga mutationer, som finns mellan dessa repetitiva delar så kan dessa användas för att separera de repetetiva delarna från varandra. Dessa skillnader som man finner kallar man för *Defined Nucleotide Positions* (*DNPs*). Utgående från dessa DNPs skapas objekt som min algoritm tar hand om, och sedan sätter ihop till contig, det vill säga längre DNA bitar som var och en representerar en specifik repetitiv del.

Algoritmen testkördes mot simulerad data, där längd och antalet repetitiva segment kunde varieras.

Examensarbete 30hp
Civilingenjörsprogrammet Molekylär Bioteknik
Hösten 2001

Contents

Abbreviations

BAC	Bacterial Artificial Chromosome
bp	base pair
CC	ClusterContainer
DNP	Defined Nucleotide Position
HS	Hierarchical Sequencing
LLR	Log Likelihood Ratio
MA	Multiple Alignment
PCR	Polymerase Chain Reaction
SC	SolidCluster
SCP	SolidClusterPosition
SW	Smith-Watermann
TRAP	Tandem Repeat Assembly Program
WGS	Whole-Genome Shotgun

1 Introduction

Ever since it became clear that it is the DNA molecule which decides how we look, function and why we get some diseases, scientists around the world have tried to determine it's composition of the four different bases; A, T, G and C. This process is called sequencing, that is, to determine in which order or sequence the bases are.
The sequencing of an organism's genome, is a large, time consuming and costly process, since sequencing is a relatively slow process and the genomes are very large. Today the most common sequencing method for large genomes is the *Shotgun Sequencing method*. This method consists of two parts, first a laboratory step followed by a theoretical analytical step.
Sequencing is a difficult task where many different obstacles have to be overcome. Some of the problems are errors from the first laboratory step, where base separation in the base calling process and sequencing errors are the two major problems. Even the structure of the genome can contribute to major problem for the sequenceing process. Specially if the genome contains a high frequency of repeat regions. These repeats make the shotgun sequncing method together with its assembly programs less favorable.
Repeat regions are very similar stretches of bases that arise many times in the DNA. These repeat regions can be of different length and distribution. If the length of the repeat regions are shorter than the reads produced in the shotgun sequencing method, then they are easier to handle, but if they are longer, i.e. over 500 base pairs (bp), they can be very hard to separate. The distribution can also be different, they can either be placed directly after each other, that is *tandem repeats* or be present at various locations, so called *interspersed repeats*.
Some organisms genomes have an higher frequency of repeat regions than others, and the organism *Trypanosoma cruzi (T. cruzi)* is an example of a parasite having extensive amount of repetitive elements. The genome of the *T.cruzi* is what *the Genome Analysis Group at the Karolinska Institute* is sequencing.
When using shotgun sequencing and alignment software tools of today, these repeat regions are very seldom correctly separated. Instead, these regions will mostly be separated into two different repeat regions placed next to each other.
To correctly separate the repeat regions, two persons in the Genome Analyzis Group at the Karolinska Institute, Martti T. Tammi and Erik Arner, have had as their biggest task to come up with a new sophisticated assembly program that can handle difficult repeat regions. The result became *TRAP, Tandem Repeat Assembly Program*.
The new thing with TRAP is that it tries to find the small differences that actually exists between repeats. The differences are natural mutations. The frequency of these mutations are in the order of 1-2% in a 500 bp fragment. When using these mutation differences the repeats can be separated and assembled in the correct way. To find these differences a *Multiple Alignment (MA)* is setup. The difficulties is to optimize this multiple alignment as much as possible, both when it comes to correctness and speed. The mutation differences in the repeats are defined as *Defined Nucleotide Positions (DNPs)*, and these have to be separated from errors made in the laboratory step, i.e. errors made in the base calling process. The TRAP program handles a huge amount of sequencing data and that slows down the assembly process, so the question is if a clustering process can speed up the run time.
The purpose of this thesis project is to investigate if there exists such a clustering algorithm and what would it look like and will it speed up the assembly process. The algorithm should be fast and correct, that is the two main things. If this algorithm turns out to speed up the assembly process in a desirable way, the clustering module will be put in to the TRAP program. The demand on the program is to handle special indata and to return special outdata. To do this a good communication between all software parts is important.

The TRAP program is written in the programming language C++, so the clustering algorithm will also be in that language.

2 Shotgun Sequencing Strategies

The shotgun sequencing method can be used with many different sequencing strategies. The two most common strategies are *the Hierarchical Sequencing (HS) Strategy* also called *Clone-by-Clone Sequencing* and *the Whole-Genome Shotgun (WGS) Strategy* [1].
The differences between these two strategies is how the "target" DNA being sequenced is organized. In the WGS method the whole genome of the organism is sheared into small fragments and these fragments are then sequenced. In the Clone-by-Clone Strategy the genome is sheared to bigger fragments in size of 100 to 200 kb. The fragments are then inserted into *BACs (Bacterial Artificial Chromosomes)* to make a BAC-library. The BACs are then sheared to smaller fragments and sequenced.
The most powerful methods seems to be a combination of these two strategies.

2.1 Hierarchical Sequencing (Shotgun) Strategy

The initial step when using the HS strategy is to create a map, *Physical Map* [1]. The physical map gives information about how the different BACs are located in relation to the DNA they originate from. The physical map is built from the BACs. The ends of the BAC inserts are sequenced and the sequenced fragments are matched against the origin DNA, see figure 1. The making of the map prevent multiple coverage of the same region and makes sure that the whole genome is covered by the BAC-library. The name HS strategy comes from the way of breaking down the problem into minor parts which is easier to handle e.g. Genome → Chromosomes (chromosome specific libraries) → BAC-libraries → Reads. Reads are the smallest parts, which are sequenced with the MegaBACE^{TM} machine. When the genome has significant amount of repeat regions it may cause problems to make the physical map, because the matching of the BAC ends against the genome will not be consistent.

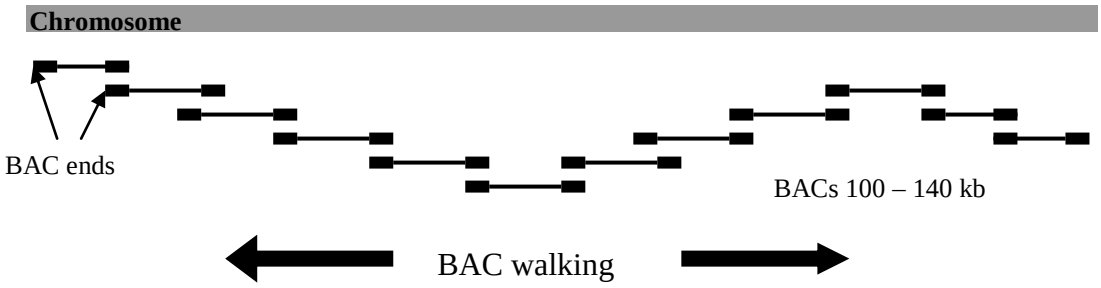

Figure 1: BAC walking. The figure shows how the BACs are building up the Physical Map of the chromosome of interest for sequencing.

2.1.1 Clone-by-Clone Shotgun Sequencing Steps

The strategy used by the Björn Andersson group, the Genome Analysis Group at the Karolinska Institute, is the HS strategy. The part of the genome sequenced is chromosome 3 from the *T. cruzi*. Chromosome 3 is represented as BAC-library, where every BAC clone constitutes a 100-140 kb long part of the chromosome. The following describes the steps in the sequencing process.

2.1.1.1 BAC-library

The BAC-library system is based on *Escherichia coli* (*E. coli*) F factor. Replication of the factor F in *E.coli* is strictly controlled and the F plasmid is maintained in low copy number (one or two copies per cell), thus reducing the potential for recombination between DNA fragments carried by the plasmid. The BAC system is very stable; it is capable of maintaining incorporated human genomic DNA fragments over 300 kb pairs long [2]. The inserts from chromosome 3 are just approx 100 kb long. The size of the mini-F plasmid, pMBO131, from which the pBAC vector is constructed, is 9000 bp, which is very small when compared to the insert with a size of 100 kb.

First step is to pick the BAC clone that will be used for sequencing. This BAC clone is grown to get enough amounts of BACs. After right amount of BACs are achieved, the BAC plasmid is separated from the *E.coli*.

2.1.1.2 Nebulisation and Insert Selection

The nebulisation step will shear the chosen BAC clone plasmids into a broad range of minor linear fragments of different sizes, the fragment size of interest is 2000 bp. After nebulisation the ends of the fragment are made blunt ended, a step called "end repair". To make the fragment ends more specific for the ligation to the sequening plasmid M13 highly efficient, adaptors are put to the ends of the fragments. The fragments are then run on agarosgel to get the specific size 2000 bp.
This will result in fragments 2000 bp long coming from both the insert and from the BAC plasmid, mini-F plasmid [3].

2.1.1.3 M13-library Construction

M13 plasmid is used for the sequence step. The M13 plasmid is split and annealed together with an adaptor. This way the ligation of insert to the M13 vector becomes highly efficient. To attain a large amount of M13 vectors with insert they are transformed into Supercompetent *E.coli* XL-2 blue cells. (The cells are grown on agar for 20 h in 37˚C.)
The cells containing the insert are now visible as placks. These are picked and incubated one more time for 20 h in 37˚C on a shaker. After that the preparation of the M13 vectors according to protocol "High-throughput of M13 DNA" (ThermoMAX prep) [3].

2.1.1.4 PCR – Polymerase Chain Reaction

The purified M13 constructs (vector and insert) are now ready for PCR-sequencing. The sequencing is performed with an automated fluorescence method based on cycle sequencing according to the protocol DYEnamic ET terminator kit (MegaBACE^{TM}) from Amersham-Pharmacia Biotech. In this kit the method of labelling is called Dye Terminator Sequencing. In this case the terminating (ddNTP) base is the one labelled with the fluorescent dye. There exists an alternative labeling method where the primer is labelled instead, Dye Primer Sequencing. During the PCR-sequencing reaction a lot of single stranded fragments of different lengths and labelled with different dyes are created. These fragments are later separated by gel electrophoresis.

2.1.1.5 MegaBACE™

The separation of the PCR-sequencing fragments take place in an automated DNA Analyzis System from Amersham-Pharmacia Biotech called MegaBACE™.

All the steps can also be viewed in figure 2.

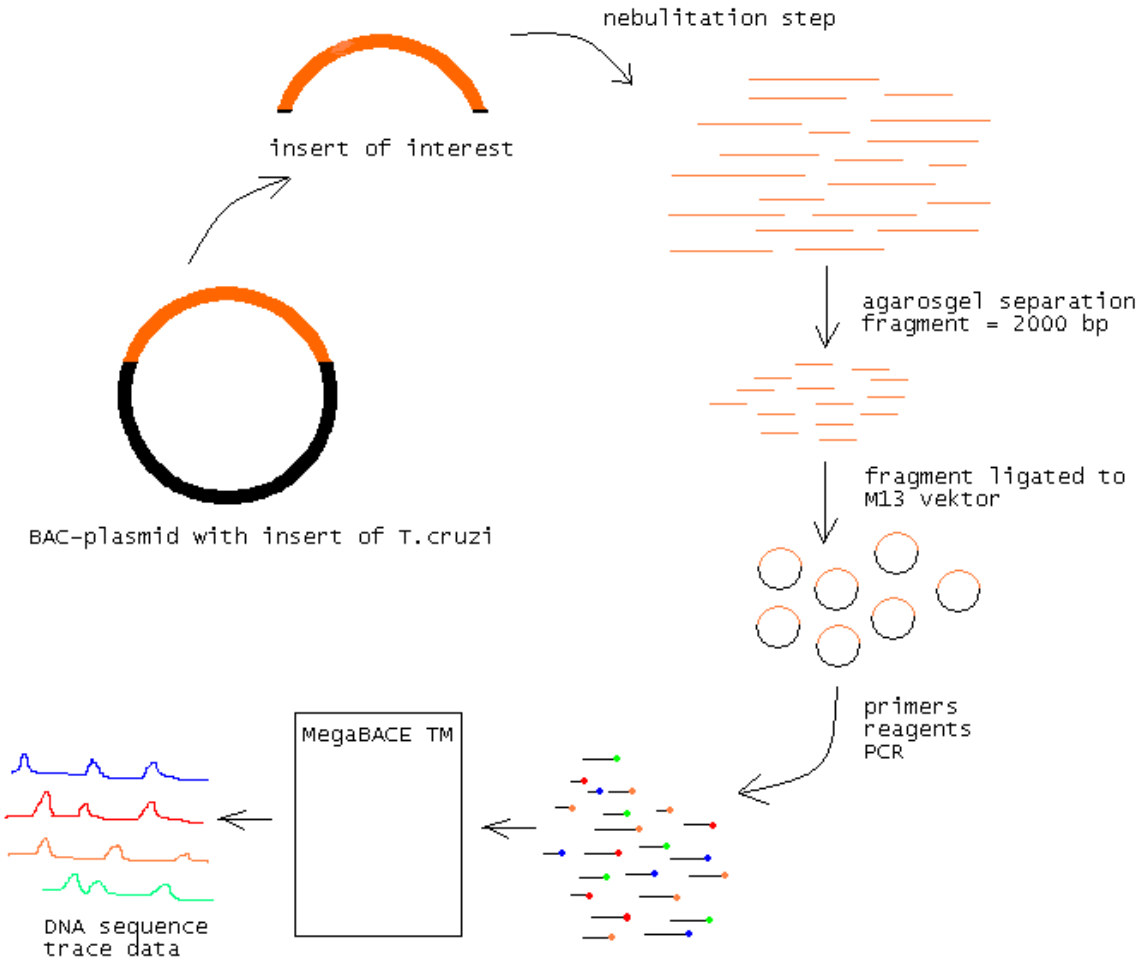

Figure 2: Flowchart of the laboratory step. From the BAC library the T.cruzi fragment is split from the factor F. This 100 kb long fragment is sheared into smaller fragments. To get the length of 2000 bp the mixture is run on agarosgel and this will separate the fragment. The band of 2000 bp cuts out. The fragments are annealed into the sequencing vector M13. To get a large amount of M13 vector with insert the is transformed in to E.coli cells for amplification. The purified M13 with insert are run in the PCR together with primers and PCR-reagents.

2.2 Whole-Genome Shotgun Sequencing Strategy

A different strategy from the HS strategy, where the sequencing problem is broken down into minor parts, the WGS strategy is to fragment the genome randomly without first making the physical map. With this approach the shotgun-sequencing step have a lot of randomly sequenced fragments (reads) coming from the whole genome. To put all the sequenced pieces together to the resulting genome, will demands for powerful computers and smart assembly programs. This approach will give rise to more fragments to handle at the same time. To finish the sequencing of the genome with WGS strategy, the sequence gaps are filled by using the clone-by-clone based strategy.

3 Base Calling

The resulting PCR product is placed and analyzed in the sequencing machine called MegaBACE^{TM} 1000. The MegaBACE^{TM} 1000 is a product from Amersham Biosciences and is an automated DNA Analysis System with a high-throughput, fluorescence-based DNA system utilizing capillary electrophoresis with up to 96 capillaries operating in parallel. The separation of the fragments takes place in these capillaries. At the end of these capillaries are the fragment laser-beam and the fluorescence are measured and an electrogram is created. See figure 2. The basecalling program task is to interpret this chromatogram and decide the sequence of the bases. A number of different basecalling programs exists like, Phred, LifeTrace, maximum-likelihood base caller (MLB), ABI base caller and the MegaBACE^{TM} own base caller, the Cimarron base caller. The Genome Analysis Group at the Karolinska Institute is using Phred.

3.1 Phred – A Basecalling Program

The MegaBACE^{TM} produces a chromatogram file containing DNA sequence trace data for each of the four (different) bases, A, T, C and G. This data is used in a process called basecalling, were the trace data is converted into corresponding base sequence. The software used for the basecalling process is called Phred.
Phred uses simple Fourier methods to examine the four base traces, and in order to predict an evenly spaced set of peak locations. That is, it determinates where the peaks would be centred if there were no compressions, dropouts or other factors shifting the peaks from their "true" location.
Ideally, all peaks in a chromatogram trace file would be spaced an even distance apart. In reality, sequence composition and the dye primer chemistry can alter the distance between peaks and that way making it difficult to identify bases accurately.
Phred uses information from the region surrounding each peak to determine the probability that a base has been identified correctly. Phred also identifies loop/stem sequence motifs based on dye primer data and splits peaks if there are indications that CC or GG peaks have merged.
Phred also assigns quality values, *Q*, to each base [4]. This Phred quality value is related to the probability of an error in base calling by the formula:

$$Q = -10 * \log_{10}(P)$$

where P is the estimated error probability for that basecall. A quality value ≥ 20 indicates that there is a chance less than a 1 in 100 that the base has been misidentified, that is incorrectly called. Note that high quality values correspond to low error probabilities, and conversely.
The most commonly used method is to count the bases with a quality score of 20 (= Q20) and above, these are considered "high quality bases", or 30 and above, "very high quality bases".

After calling all bases, Phred writes the sequence to a file in either FASTA, PHD or the SCF ("Standard Chromatogram Format") format [4], depending on how you want it. The Quality values for the bases are written to a separate file in either FASTA format or to a PHD file, a text file which contains base calls and quality information, which can be used by the Phrap sequence assembly program in order to increase the accuracy of the assembled sequence.

4 Assembly

After the base calling process a lot of sequence fragments have been created. These fragments or reads are now put together, assembled, into the original DNA sequence. The sequence assembly is the process of constructing the "best guess" contig/clone-sequence from a set of overlapping reads of the clone. The assembly process is complicated by a number of computational problems, and a sequence assembly program must be able to handle these. [5] The most important ones are: *repeated regions* that can be of two kinds – interspersed repeats and tandem repeats, *base-calling errors* or *sequencing errors* – of which there are four types, substitutions, deletions, insertions and ambiguous base calls, *contaminations* – sequence from host organism used to clone shotgun fragments, *unremoved vector sequence* – vector sequences can be present in reads, and if not removed, these may cause false overlaps between reads, *unknown orientation* – it is not known from which strand each fragment originates, *polymorphisms* and *incomplete coverage* – coverage varies in different target sequence locations due to the nature of random sampling.
The tools used for sequence assembly are Crossmatch (used for vector screening, and will not be describe here), Phrap (used for the fragment assembly) and Consed (a graphical tool for sequence finishing).

4.1 Phrap - An Assembly Program

Phrap, Phragment assembly program or Phil's revised assembly program, is for the moment the leading program for assembling shotgun DNA sequence data. The Phrap program is strongly recommended to be used in conjunction with the base calls and base quality values produced by the basecaller, Phred. As input data Phrap uses the two result files produced by Phred, the sequence fasta file and the quality file. It uses the quality information provided by Phred to discriminate repeats and sequencing errors in the assembly process and to construct contig sequences as a mosaic of the highest quality parts of the reads (rather than a consensus). If there is a discrepancy at any position, Phrap uses information from Phred to assign a quality value to the discrepancy. The program is able to handle very large datasets.
Phrap also provides extensive information about each assembly (including quality values for contig sequences) to assist in troubleshooting.

4.1.1 The Phrap Algorithm

To understand how Phrap work in more detail, the different steps are here described.
As input to the Phrap program are the two files containing read sequences and quality data. The read ends are in some extent trimmed off if there are any near-homopolymer runs and read complements are constructed. The next step is to find, look for, all possible overlaps between the reads. This is done by analyzing all reads after words. In Phrap these words are normally 14 bases/letters long, but shorter or longer words can be used. These words are found by using a "Sliding Window" that runs along the read and every word is registered into a structure that keeps track of the word's first base. The structure can be described as a matrix, where the first element is the word's index, described later, and the rest of the elements are pointers to the first base in that word located in different or in the same read. On the same row in the matrix all pointers are pointing at the same 14 bases long word but at different locations among the reads. As an example: if a read is 500 bp long, that read will give rise to 487 (500-13) different words and also 487 new pointers. Every time a word appears a pointer is connected to it and placed in the matrix. The matrix indexes are composed of the words first 10 bases like this. The index is an integer with the base 4 and every word give rise to a

specific integer. Every base is given a number, A = 0, T = 1, G = 2 and C = 3 (base equal to 4). As an example can the sequence, ATGGCTATAC be represented by the following integer:

$4^9 * 0 + 4^8 * 1 + 4^7 * 2 + 4^6 * 2 + 4^5 * 3 + 4^4 * 1 + 4^3 * 0 + 4^2 * 1 + 4^1 * 0 + 4^0 * 3 =$
$262.144*0 + 65.536*1 + 16.384*2 + 4.096*2 + 1.024*3 + 256*1 + 64*0 + 16*1 + 4*0 + 1*3 =$ **109.843**,
the index for the word ATGGCTATAC is 109.843.

When all reads are analyzed for words it is time to find pairs of reads with matching words. All reads with pointers on the same row in the matrix are analyzed at the same time, these reads have the 14 bases long word in common. The reads are put into a list, the list is then ordered in alphabetical order, with the 14 bases and the following bases at count (see figure 3).

Figure 3: Arrangement of reads in unsorted and sorted lists in the Phrap assembly program. All read containing the same word are put into a list, and ordered in alphabetical order. Reads next to each other have a sequence more similar than reads longer away.

After the list has been sorted in alphabetical order, reads next to each other are the ones having most similar base sequence. The reads are then pairwise matched with Smith-Watermann (SW) algorithm [6]. Reads showing high similarity gets a high SW score value and reads with lower similarity get lower SW score. Only read pairs with an SW score over a certain threshold are further used. When every read pair have got an SW score, the next row in the matrix is analyzed, and this is done until all rows are analyzed.
The kept read pairs are analyzed with the Log Likelihood Ratio (LLR), which gives the read pairs a LLR-score [7]. If the LLR-score is ≥ 0, the read pair is statistical possible, and if LLR-score < 0, the read pair is not possible to align. All read pairs are given a LLR-score and put into a list with decreasing LLR-score.
Next step is to construct contig layouts. The first read pair being used is the one with the highest LLR-score, at the top of the list. This read pair is the first part of the contig. Then the read pair with second highest LLR-score is used. If this read pair has common reads with the contig, all different combinations are examined to see that the LLR-score for those

combinations are ≥ 0 (see figure 4). If the LLR-score for some of the different read pairs are less than zero, the two read pairs will not be put together into one contig.

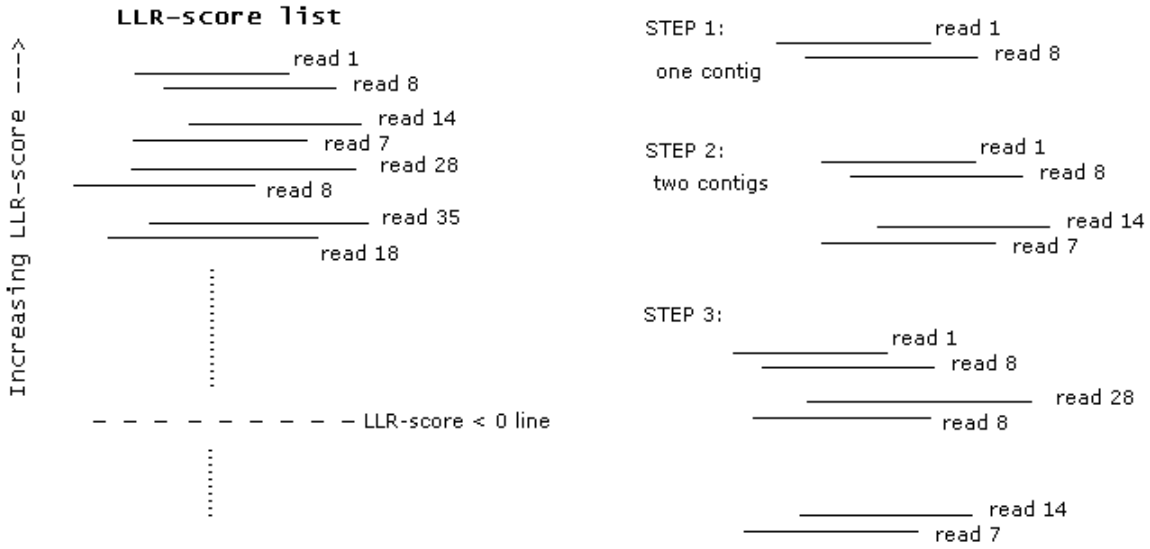

Figure 4: The assembly of reads into contigs in Phrap. In the LLR-score list are the read pairs ordered in increasing LLR-score. The way to pick the read pairs is following the greedy algorithm, the read pair picked is always the one highest in list, with the largest LLR-score. In Step 1, the best read pair constitutes the first contig. When the second best read pair is used, no common reads are found between the two read pairs, so they can not be assembled, instead the result is two contig. In the third step the read pair contains common read with the first read pair. To be able to assemble these read pairs together all possible combinations must be analyzed, in this case, read 1 with read 28. If the read pair with reads 1 and 28 exists and has a LLR-score bigger or equal to zero the pairs can be assembled into one contig.

After all read pairs with LLR-score ≥ 0 have been used, hopefully one big contig is left. But most of the time many contigs have been made. The resulting contigs can be further analyzed. For example if a read is only assembled into one special place in the contig, it can never be assembled into many different regions in the contig. If there exists such reads in the contig, those places are further analyzed to get the best resulting contig. The read pairs in those regions are all realigned with each other and the best result is chosen and put back into the contig.
When all contigs are optimized the consensus string is determined, and this can be made in different ways. Either are those bases with the best quality score put into consensus, or the bases in majority put into consensus. In this way consensus will be made like a mosaic of different parts of the contig.

4.1.2 Consed – A Graphical Tool for Sequence Finishing

After the fragment assembly with Phrap have been done the result can be visualized and edited with Consed. Consed [8] is a program made for editing sequence assemblies created with the Phrap assembly program. It was written specifically for the Phrap, where it takes advantage of quality values assigned by Phred and Phrap and the consensus sequence created by Phrap. In addition to a full set of standard features (view traces, edit reads by inserting a base, deleting a base, substituting a base, etc.), it supports an efficient editing procedure designed for use by Phrap in subsequent reassemblies of the same data set.
With Consed it is able to see the alignment, the consensus, the different reads and the quality of the bases (figure 5).

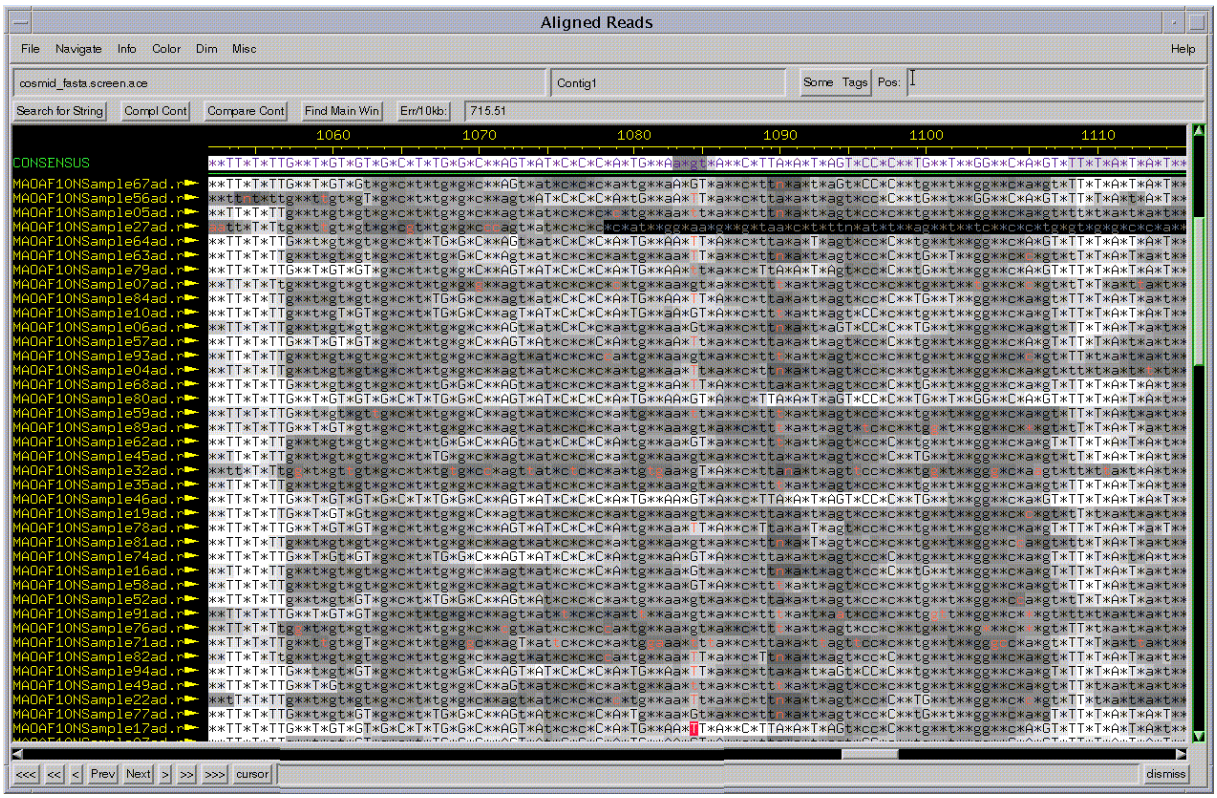

Figure 5: A view of the Consed Program. After assembly, it can be interesting to view the result, this is done with the program Consed. If the assembly is not satisfactory enough it can be edited with Consed.

4.2 TRAP – Tandem Repeat Assembly Program

Even though today's assembly programs are very complex they still have difficulties in assembling different genomes in a correct way. The part that is still difficult to handle is repeat regions in different shapes, dispersed and tandem repeated. Some assembly programs have the possibility to sort out possible repeat regions but they still lack the abilities to separate and assembly them into correct contigs. If the repeat region is shorter than the average read length (500 bp) it is easier to solve, but when the repeat length is longer than the average read length then it turns out to become a large problem [5].
To be able to separate repeat regions the small and unique differences between them have to be found. The differences are very few, can be as low as 1% in 500 bp and are results from mutations in the evolution. Other differences between reads depending on sequencing error is much higher, it can be up to ten times higher.
The real differences between the repeats are called Defined Nucleotide Positions, DNPs.
The program developed in the Björn Anderssons Group by Martti Tammi and Erik Arner is called TRAP (Tandem Repeat Assembly Program) and it is using the DNP method to separate repeats and assemble them into contigs.

4.2.1 DNPs – Defined Nucleotide Positions, and Analysis of the Multiple Alignment

To separate identical repeat regions is impossible, but to separate nearly identical repeats is possible. To do this it is necessary to find unique differences between the repeats and also that the difference in frequency is so high that reads 500 bp long at least contain two unique differences.

To find these differences between repeats, sequenced reads from these repeat regions are aligned into a multiple-alignment (MA) together with all of their overlaps with other reads [9].
The read that begins the construction of the MA is called "Starting read". All reads that the starting read overlap with are put into the MA. These reads are called 1st-order reads. To get as much information out of the MA as possible, even the 1st-order reads overlap with reads not already in the MA are put into the now total MA. As a clarifying part, the 2nd-order reads are not overlapping with Starting read. To put the 2nd-order reads into the MA is not always done, it is just an option. A graphic description of the MA can be pictured in figure 6. To get the MA as good as possible it is optimized locally by the ReAligner algorithm.

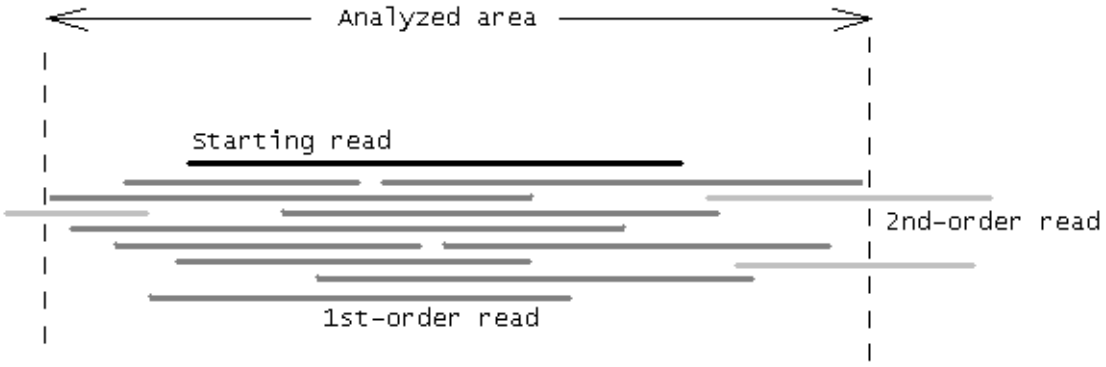

Figure 6: Multiple Alignment structure. A graphical view of the multiple alignment. The starting read can be seen in black, 1st-order reads in dark grey and in light grey 2nd-order reads. The analyzed area is the length from left to right covered by 1st-order reads.

The differences between repeat regions depend of sequencing errors and real differences from mutations. The sequencing errors can be distinguished by the fact that they are distributed randomly, whereas real differences are not. When computing the distributions of true and false differences on one column in the MA, the overlap between them will vary depending on the rate of sequencing error, coverage, the number of repeats, and the number of differences between repeat units. The separation of the distributions is not sufficient for detection of an acceptable rate of true differences (Figure 7a). From the diagram in figure 7a it can be seen that at least six bases in the column must be computed as true differences to be sure that it is not a sequencing error. True differences are also called true positives. If also consideration to the rate of coinciding deviations from column consensus between at least a pair of columns in the MA is taken, it results in an even better separation (Figure 7b). So from this at least three or four coincidences are enough in order to separate error from differences.

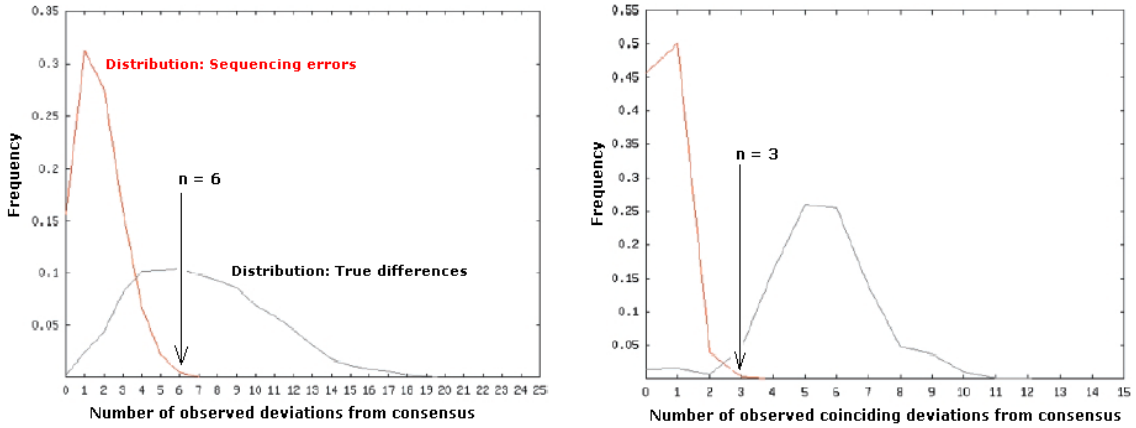

Figure 7: Distribution of differences in a multiple alignment. (a) The distribution of sequencing errors and real differences computed on one column. At least six or seven differences from the consensus must be observed in order to separate true differences from sequencing errors. (b) The distribution of coincidences due to sequencing errors and real differences computed on two columns. Three or more coincidences are enough in order to separate errors from differences.

After the analyzis of the MA, the starting read and the reads with 1st-order are labeled analyzed. The starting read is totally analyzed and the 1st-order reads are partially analyzed. The next step is to pick a new starting read from the set of non-analyzed reads or from the partially-analyzed reads. If the new starting read is taken from latter set, it should be the read less analyzed. This process is repeated until all reads in the data set have been labeled analyzed.

4.2.2 The TRAP Algorithm

The TRAP algorithm can be seen as a chain of steps [10]. The five major steps are (1) preparation, (2) computation of overlaps, (3) analysis of fragments from repeat regions, (4) fragment layout generation, and (5) generation of the consensus sequence. The implementation of the algorithm consists of four separate program modules, TRASH, INITMATCH, BSW and MAM-1.

The first step is the preparation, and here the quality of the reads are analyzed. The sequence reads are partitioned into good quality, the accuracy level q is 89% < q < 98% and very good quality, q > 98%. Reads not in any of these levels are discarded. A good quality of the reads are very important for the accuracy of the following step, which is the computation of overlaps between reads. In *the second step*, where a lot of different overlaps between reads are computed, a great advantage is if all reads are arranged in a sorted way. To do so all reads are stored in a hash table. Next is to compute pair-wise alignments using the SW algorithm, finding overlaps among the subsequences in the hash table.

In these step, also information about the read distribution is given, and since repeat and non-repeat fragment have different distributions, it is possible to divide most of the fragments into two categories, repeats and non-repeats. All different overlaps are scored with a method similar to that of Phrap, and the overlaps are sorted with the highest score first.

The third step of the algorithm is separating nearly identical repeats. To do that single base differences between the sequence fragments must be detected, and determined whether these differences are sequencing errors or real differences. A real difference in a single base are called, Defined Nucleotide Position (DNP), and these positons will be pinpointed in the assembly process. To find possible DNPs a read and all of it's candidate overlap are put into a MA. The MA is optimized and then analyzed column by column. The analyzis is based on the

fact that the sequencing errors and single base differences between repeats can be separated when they are distributed differently. In each column in the MA, the relative frequence of the bases and gaps are calculated and these frequencies are compared with the distribution of sequencing errors and real differences. Bases registered as real differences can be a DNP, but so far they are just DNP candidates. To verify that a DNP candidate is a real DNP, and the fact that reads may have differences at several positions can be used to confirm the DNP. In the process, all reads with the same DNP candidate are compared to see if they have more DNP candidates in common. In the current version of TRAP correlation between at least two DNP candidate positions in a read are required to qualify positions as DNPs.
When all DNPs are verified, it is time for *the fourth step,* fragment layout generation, in which contigs are constructed.
In this part a heuristic algoritm is used, and the starting point is the highest scoring read pairs from the list mentioned above. All pairs matching either of the two reads in the starting pair are identified, and the candidate read with the highest score is chosen for inclusion in the contig. Reads with confirmed DNPs are required to have DNPs and are not allowed to mismatch at a DNP, i.e. DNPs are pinpointed. This process continues until no more reads can be included to the contig. If there exists reads not part of a contig a new contig is started, and this is done until no more reads are left in the database.
In *the fifth and final step* all contigs are assamble into a concensus sequences. How this part is best made is under further investigation.

5 The Thesis Project – Algorithm and Clustering

Even though the TRAP program is very robust it can still be modified. One wishful thing is to speed up the program, to make it faster. One of the ideas is to cluster reads together to make the number of read comparings in the assembly process become fewer. Instead of comparing all reads to each other, base by base, reads with the same DNP candidates are clustered into a container or structure called a SolidCluster (SC) and all these SCs are then compared in the assembly process instead.
The thesis project aim is to develop an algorithm which takes the SCs and arrange them into a structure and then from that structure cluster the SCs in the assembly process into a correct contig.

5.1 Algorithm Development

The plan from the beginning is to concentrate on the pure differences between the repeat regions, the DNPs, instead of putting a lot of time and effort on analyzing overlaps between reads and their scoring like in TRAP. The way of doing this is to find DNP candidates and verify them pairwise and just compare these specific positions in the construction of contigs and consensus sequences. This way you do not have to compare all the base pairs which is a very time consuming part in the overlap investigation. If this strategy is functional, a large time reduction of the process of putting reads together to contigs and consensus sequences can be made.
This algorithm would be a nice transfer from TRAP and its multiple alignment (MA), where all the DNPs are found and later verified. But instead of continue with TRAP's fourth step, each pair of DNP verification is represented by this new object called SC. The SC objects are the objects the algorithm is given to work with.
The optimized MA in TRAP, finds DNP candidates, the DNP candidates are verified in pairs, and each DNP pair forms a SC with the two DNPs and a list with reads containing the DNPs, and finally the SCs are arranged into a structure containing all SCs formed from this multiple

alignment. The approach of this project is to investigate a way of arranging the SCs, and how to cluster these SC into bigger parts, arranging these parts into contigs and finally into consensus sequences.

Even this algorithm, like TRAP, will be developed in several steps, five steps are distinguished, (1) adding SolidClusters, (2) make resolve order, (3) resolve process, (4) chain process, and (5) merge process. Step 3 is the main step, where most of the work is done making contigs.

5.1.0 Finding DNP Candidates

The process of finding DNP candidates is the same as in the TRAP algorithm. The analysis of the optimized MA will return all possible DNP candidates found in the MA. Each DNP candidate found in the MA is given an identification (id) number, specific for that DNP candidate. All reads in the MA containing some of these DNP candidates will have this information added to their DNP list. All DNP candidates found must now be investigated if they really are DNPs, and that is the next part, verifying the DNP candidates.

5.1.1 Verifying DNP Candidates

The verification of DNP candidates is a very important part in the process of developing the new algorithm. The understanding of this part gives the basic knowledge for the development of the algorithm. It is important that each verification is correct and that all possible verifications are done (see figure 8).
The MA has found DNP candidates, and it is time to verify them. Start by getting the first found DNP candidate from the MA, the one with the lowest id number. Since the analysis of the MA is made from left to right, the first found DNP candidate will get the lowest id number. All reads with this DNP candidate id in their list are grouped and will be analyzed. This first DNP candidate is the core of the group of reads, it is called *Start DNP* in the verification process, but called *left_DNP* in a DNP pair if the verification became true. The other DNP candidates on the reads, are called *Target DNPs* in the verification process, but *right_DNPs* in a DNP pair. The verification is made in pairs, where the *Start DNP* candidate verify the *Target DNP* candidate. The verification the other way around, Target DNP to Start DNP is not necessary, since the result will be the same. Due to this the verification is said to work in the direction, left to right.
The number of reads in the group must be at least three, otherwise this DNP candidate, the start DNP, will be classified as false, and be discarded. If that happens the analyzis of the reads in the group will be interrupted, and the verification step continues with the next found DNP candidate id from the MA, and so on. If the group contains three or more, the verification of the group continues. As mentioned before the verification is made in pairs, the start DNP verifies the target DNP. The start DNP is known, so the next part is to find a target DNP, and if this is the first verification in the group, find a DNP candidate with an id number closest above the start DNP id, that's the target DNP. The target DNPs are always positioned to the right on a read, in relation to the start DNP. If at least three reads have this target DNP, the verification is positive, or called true, and a DNP pair is found. The start DNP is called the pair's *left_DNP*, and the target DNP is called the *right_DNP*. This process continues, try to find a second target DNP, verify if it is true or false, and make a second DNP pair. This is repeated until no more target DNPs can be find among the reads in the group. When this is done, look in the MA for the next DNP candidate id, call it the new start DNP, group all reads with that DNP in common, and find target DNPs and verify DNP pairs. Contiune this until no

more DNP candidate ids are found in the MA. Figure 8 will hopefully make the process a bit clearer.

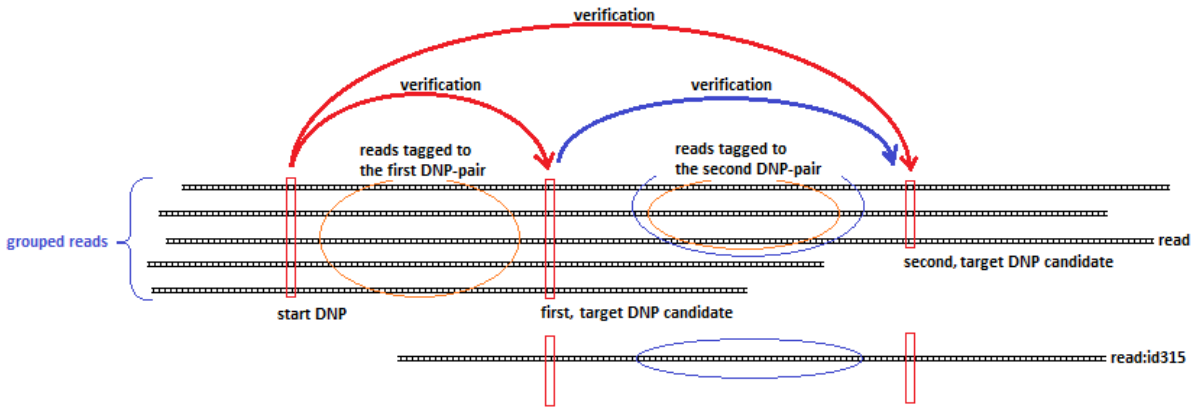

Figure 8: The verification process. Reads grouped with their first DNP, start DNP, in common. The verification of DNP candidates, target DNPs, are in the order left to right. All reads with the two DNP candidates are tagged to be part of that DNP pair. DNPs which have already been verified, first target DNP candidate, can also verify already verified DNPs and new target DNP candidates. This is marked with the blue verification arrow. This way, one more *one-step* verification is made and also the read with id 315 will be tagged. This read was not tagged in the first verification process, because it lacks the first DNP, marked in the figure as start DNP.

Each time a target DNP is verified as true, all reads containing these two DNPs are tagged with that DNP pair. This information is also what is going to be used in the algorithm.
By giving the verification a direction from left to right, a start DNP can only verify target DNP candidates to its right.

Hopefully all reads grouped together will come from the same repeat sequence. If a sequencing error happens giving a read a false DNP candidate, it becomes part of a group by mistake, and is also tagged to a verification and then also being part of a verification by mistake.

An important thing to note (as seen in figure 8) is that a verification between two DNPs next to each other in a read, where the distance between them are relatively short, will have a higher probability of being true, than a verification between two DNPs further apart from each other. For example, in figure 8 a verification between start DNP and first target DNP candidate is more likely than a verification between the start DNP and second target DNP candidate. This is the case since a higher number of reads will cover a shorter distance than a longer distance. If more reads are part of a verification between two DNPs, the verification will be more probable to be true. This feature will be used in the algorithm. Depending on how the verification is done, i.e. if the DNPs are next to each other or if there is one or more DNPs in between, the result are called *one-step* or *two-step*, and so on. If the start DNP is the same, the *one-step* verification should be more or equally probable than the *two-step* verification.
Due to the higher probability of *one-step* verifications the strategy from the outset was changed. The original idea was to let the first DNP on the read, known as start DNP, confirm all the following DNPs on the same read. But that was not the best idea. If a read (in figure 10, read:id315) just overlap with the first target DNP and second target DNP, that one would be excluded, even if it was correct. But if the reads above it are part of the verification, it will be tagged as part of a DNP-pair.

That is why it is so important to let all DNP candidates, in the MA to be the start DNP. After the first DNP with the lowest id number from MA is analyzed, the next DNP with the second lowest id number from the MA is analyzed. This way all DNP candidates in the MA are given a chance to verify other DNPs.
It is also important to understand that verification of DNPs and DNP candidates only can be done in the read and not between reads. That is since a DNP could be verified between reads one have to 100% be sure that the alignment between them is correct, which is not possible.

Verifying rules:

1) A DNP can only verify a DNP after it self (in our case to the left of it). The verifying DNP is called the start DNP and the DNP verified is called target DNP.
2) A DNP can only verify DNPs on the same read, and not between reads. i.e. the start DNP and the target DNP has to be on the same read.

To enable merging of DNP reads into contigs, each read must have at least two DNPs. When a read has more than one DNP candidate, it becomes more probable to separate real repeat differences than differences due to sequencing errors.

5.1.2 Making SolidClusters

Each time a verification is true, a specific object called a SolidCluster (SC) for that verification is made. The SC contains the information about the two DNPs and a list with all reads tagged for that verification. Since the threshold for a verification to be true is at least three reads, the list in the SC will always contain information from at least three reads.
When programming in areas where the amount of data is very large, it is very important that not many copies of the data in use are made. The list in the SC only contains the reads ids, not the actual data object of the reads. This approach will be used to the largest extent throughout the algorithm.
Since the SC object contains a list of read ids, the SC can be seen as a subcluster of these reads. Compared to TRAP, these reads do not have to be compared to each other which speeds up the process. To separate DNPs in SC they are called *left_DNP* and *right_DNP*. A graphical picture of a SC's structure can be seen in figure 9.
Each MA creates a large number of SCs. Reads containing more than two DNPs will be members of more than one SC. The maximum number of true verifications a read with n DNP candidates can be tagged to is $v(n) = n \cdot \left(\frac{n-1}{2}\right)$. That is also true for the maximum number of possible SCs this read can be member of.

5.1.3 Order SolidClusters into ClusterContainer – *add_cluster*.

All different SCs must be stored in a data structure, and the structure choosen is a two dimensional list. The way the SCs are added to this list, should make the following steps go fast and be as correct as possible. The filled structure very much looks like a matrix. The structure is called *left_vec*. The name come from the fact that each row in the two dimensional list contains SCs with their left_DNP in common. This *left_vec* structure is it self part of a larger object called *ClusterContainer (CC)*. The CC also contains the structures; *right_vec*, *work_vec*, *rest_vec* and *resolve_vec* (figure 9). A description for each of the substructures comes along the way.

The SCs are added to *left_vec* based on their *left_DNPs*. The order of the SCs on the row is the smallest step first (to the left), and with increasing step length, to the right. This arrangement will came natural if the MA is analyzed as described before, no sorting of SCs is needed.
At the same time a SC is added to *left_vec*, an object corresponding to that SC called *SolidClusterPosition (SCP)* is made and added to *right_vec*. Each SC contributes to one SCP. The SCP contains two pieces of information, the *left_vec* row number, and what position on that row the SC is placed. And this SCP object is added to *right_vec* on the same row as the SC's *right_DNP*. After all SCs are added into *left_vec* the number of rows are equal to the total number of different *left_DNPs*. The number of SCs on each row are different, depending on how many *right_DNPs* that particular *left_DNP* was able to verify.

This part of the algorithm ended up quite complex. From the start it was just planed to be one structure, the *left_vec*. But after some thinking and after going through some smaller examples of MAs and working out different scenarious of what can happen in the processes of producing SCs, *right_vec* and finally even *work_vec* and *rest_vec* were produced.

During the process of verifing DNP candidates, a relatively high number of SCs are produced and added to *left_vec*. The order the SCs are added to *left_vec* is important for how the continuing of the algorithm turns out. And to be able to use as much information from all the different SCs, *right_vec* is used to keep SC coming from the same verification areas grouped together. SCPs on the same row in *right_vec* are all referring to SC in *left_vec* with the same *right_DNP* ids. The information stored in *right_vec* will be used in future steps in the algorithm.

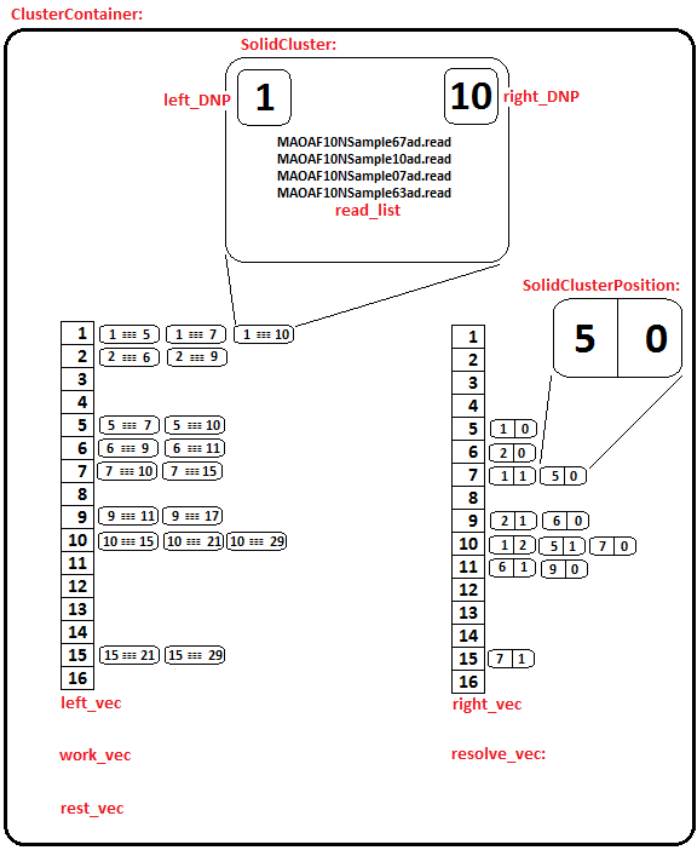

Figure 9: The ClusterContainer structure. The structure of the class ClusterContainer with it is five substructures; *left_vec*, *right_vec*, *resolve_vec*, *work_vec* and *rest_vec*. To the ClusterContainer has some SolidClusters (SCs) been added, and also some related SolidClusterPositions (SCPs). This is how it can look like after the Add_cluster step.

5.1.4 Arranging the rows in *left_vec* after quality – *make_resolve_order*

After adding all SCs to *left_vec* it is time to start resolve them into bigger SCs, which means that all SCs on the same row in *left_vec* will be put into just one big SC. Instead of just representing two different DNPs, *left_DNP* and *right_DNP*, the bigger SC will represent all DNPs on that particular row.

In the beginning, the order the resolving process was done, just from *left_vec's* first row to its last row, from top to bottom. But after some test runs with the resolve process, it became clear that this was not the right way to do it. Since the rows in *left_vec* all have different quality, some rows are better than other, that information should be used. Instead of starting resolve rows with minor quality, it is better to first score the rows after increasing quality and start resolving the rows with highest score. This approach is generally called the "greedy algorithm" approach.

One incident that triggered this part of investigation of row quality, was when a SC turned out to be false, e.g. false positive. That SC messed up the whole resolve process. Hopefully, during quality calculations false positives will have really low quality values, and be resolve really late, or preferably not at all.

The function calculating the quality of the rows in *left_vec* is called *make_resolve_order*. What makes good quality? Since each row contains a number of SCs, a row with high quality, is a row with high quality SCs. High quality SCs are SCs which have a high probability to be true. The probability of a SC to be true is determined by the DNP verification process. The outcome and quality of a DNP verification process is determined by how many reads containing both DNP candidates in the particular verification. A higher number of reads containing the DNPs, gives a higher quality of the verification. But as mentioned before, a true verification needs to contain at least three reads. That is what determine the quality of a row, the number of reads.

A reminder before continuing. A SC with DNPs positioned next to each other in the DNA sequence (i.e. *one-step* SC) , has a higher probability to be true, than a SC with the same *left_DNP* as the former, but with a *right_DNP* more distant than the former's *right_DNP* (big-step SC). That is because a read is more likely to cover a short stretch, then a long stretch of DNA. That is why the first SC on a *left_vec* row should have the highest quality, exception exists, and this is when the first SC is a false positive.

The quality can be calculated either by the mean value of the quality of all SCs on the row, or just the quality value of the first SC on the row. The last option is not so good because the risk of false positive. If this happens the SC is flagged with an error and added to *error_vec*. The approach used is that the best SC on the row represents the quality of the row. That SC is also added to a structure called *resolve_vec*. In the end of the making of resolve order, all rows will be scored and added to *resolve_vec*.

The next step, which is the Resolve Process, resolves the rows in *left_vec* according to the SCs in *resolve_vec*.

One thing to remember is that, the resolve order only tells in what order the rows in *left_vec* should be resolved, not how the SCs on the rows are resolved.

5.1.5 Putting SolidClusters together – *resolve_cluster* – Resolve Process

After all SCs have been added to the *left_vec* and the resolve order have been calculated then comes the next step in the algorithm, the Resolve Process. In this step the goal is to cluster all SCs on the same row into just one big SC. This big SC is also a SC but it differ in that way that it

contains more than two DNPs, it contains all DNPs represented on that row.
The first strategy was to start with the SC at the end of the row, and cluster it with the SC to it is left. This approach was soon discarded. Also here the "greedy algorithm" approach turned out to be the best way to continue. I start with the most probable SC, the *one-step* SC first on the row, and add the second most probable, and so on. This is why the resolve process of a *left_vec* row is given the direction from left to right. Meanwhile this resolving process happens, other SC in *left_vec* can also be clustered into the resulting big SC. These SCs are representing *one-steps*, *two-steps*, and so on, inside the resulting big SC. The way these extra SC are connected to the resolving process is by the *right_vec* and all its SCP objects. Each time a SC is clustered in the resolve process, its corresponding SCP in *right_vec* will be looked up, and all SCP on that row in *right_vec* will be analyzed. An explanation of the different kinds of SCPs on the row in *right_vec*; one of the SCP is the one corresponding to the SC taking part in the resolve process, some of the other SCPs on the row correspond to SCs not taking part of the resolve process. These SCs have a *left_DNP* id less than the resulting big SC, these SC are discarded and because they are not errors they are added to the *rest_vec*. The *rest_vec* will contain SCs which never have been analyzed as errors, but they have also never been used in any resolving process. Finally the SCPs corresponding to SCs taking part of the resolve process, these SC are added to *work_vec*. The *work_vec* will contain all SCs which represent steps inside the resulting big SC from the resolving process. After all SCs on a row in *left_vec* are clustered into the resulting big SC, the process continues to resolve all SCs added to the *work_vec* into the resulting big SC. (See figure 10)

In the resolve process, all DNPs are stored in the resulting big SC. During the clustering of two SC, both their lists of read ids are compared to each other to keep reads which are correct and discard reads which lack some important DNP. To do this, information about where a DNP is positioned and where a read ends is necessary, this check is called *left_check*. This can be compared to the resolve process with the SCs coming from the *work_vec*, where the information about where a DNP is positioned and where a read starts is necessary. This check is called *right_check*.
The goal of *left_check* and *right_check* is to analyze if a read lack a particular DNP. If that is the case it should not stretch over that position, neither to the left nor to the right.

The Resolve Process turned out to be quite complex, especially with the *right_vec* structure. The reason why *right_vec* was made, was to add as much verification information as possible and to cluster as many reads into the resulting big SC. The extra information comes from the verifications inside the resulting big SC. These SCs were not placed on the same row in *left_vec*, just beeing resolved.

During the resolve process, both when finding the resolve order, and the clustring of SCs it is very important that the structure of *left_vec* never changes. That is because the *right_vec* is holding a lot of SCP and each SCP is corresponding to a SC on a particular row and position. That is why a SC can never be added or deleted from *left_vec* in the resolve process. When a SC is used it will be flagged in some way depending on how it has been used. E.g. if a SC is found to be false during the resolve process, the SC is flagged whit an *error flag* and will not be part of any resulting big SC. Also if a SC is added to *work_vec* during the resolve process, it is flagged with the *used flag*. Finally if a SC is put into the *rest_vec* it is flagged with the *rest flag*.

When a row in *left_vec* is going to be resolved, the first SC on the row must be a *one-step* SC. If the first SC already has been used in the resolve process of another row, the rest of the SCs

on that particular row will never be used. Instead these SCs will be added to *rest_vec*. What to do with those SCs have not yet been decided. The question is, should a SC be able to be resolved more than once and thereby be part of many resulting big SCs. But for now a SC can only be resolved once.

When a big SC is completely resolved it will be the first one on that row. The resolve process continues until all SCs in *revolve_vec* are used. (See figure 11.)

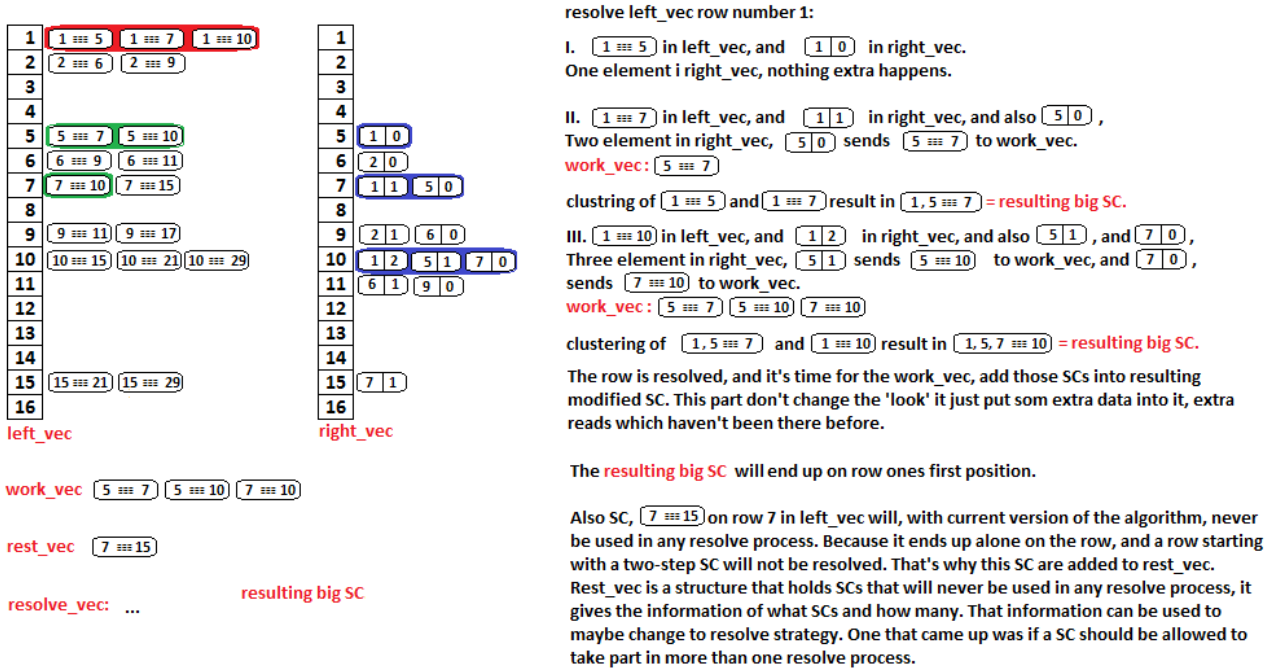

Figure 10: The start of the Resolve Process. How the first row in *left_vec* is resolved. SCs on the row in *left_vec* that are clustered together are marked with red. The positions in *right_vec* that are being used marked with blue. What SCs in *left_vec* those positions (SCP) in *right_vec*, are added to *work_vec* (those SCs are marked with green). And also what kind of SCs that can end up in *rest_vec*.

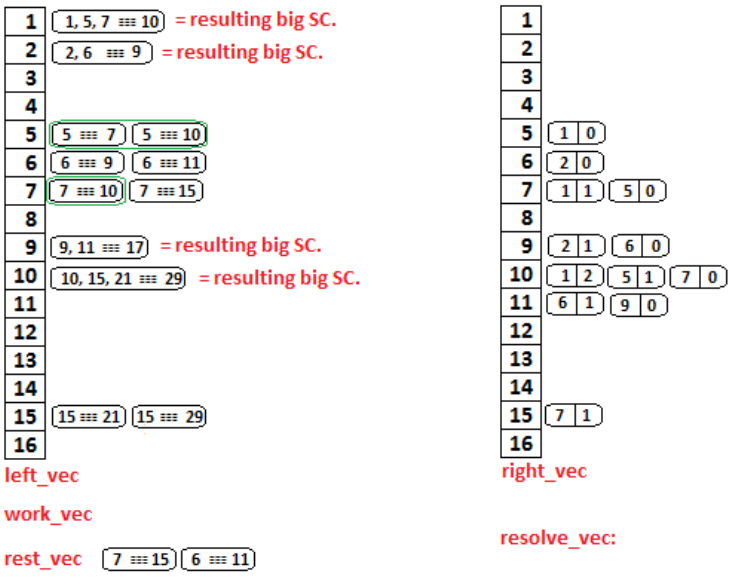

Figure 11: The result of the Resolve Process. *left_vec* after the resolve process. Row 1, 2, 9, and 10 contains resulting big SCs. These resulting big SCs are chained together in the next step in the algorithm, called Chain Process.

5.1.6 Adding Resulting Modified SCs together – chain_cluster – Chain Process

The Chain Process is a straight forward process. It goes through *left_vec*, from top to bottom and looks for resulting big SCs. When one is found, it marks that row and looks at the resulting big SC's *right_DNP*. It then goes to that position in *left_vec* and if there is a SC there, it picks that out and places that after the SC on the marked row. When this is done a check similar to the one done in the resolve process is made, *left_check* and *right_check*. Next it looks at the latest added SC's *right_DNP*, looks in the *left_vec* on that position, and does this until no more SC can be added. Now this row is done and the SCs on that row is now part of a contig. Unmark the row and continue go through *left_vec* to a row containing a SC not part of a contig and repeat the process. This is done until all rows in *left_vec* is check for SCs. (se figure 12).

The number of contigs in *left_vec* should now be equal to the number of repeats in the multiple alignment, one contig representing one part of each repeat. If there are less number of contigs, maybe a DNP has been verified to belong to a false repeat sequence, or if there are more number of contigs a DNP has not been verified as a true DNP, despite it was. And therefore an important DNP is missing, which are splitting the contig into two pieces.

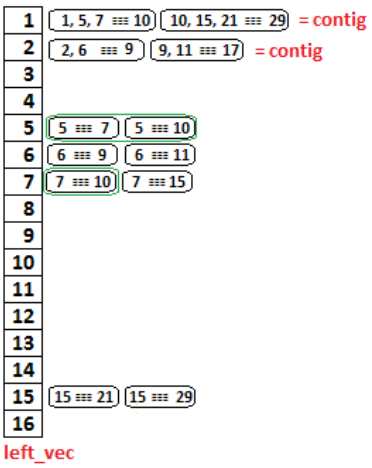

Figure 12: The result of the Chain Process. On row number 1 in *left_vec* two resulting big SCs are placed, and these two res. big SCs are representing a contig from one of the repeat sequences. Also on row number 2 there is a contig representing another repeat sequence.

5.1.7 Making the Chain into one Unit – Merge Process

The fifth and final step in the algorithm is a light-weight step where all SCs on a chained row are merged together in to one large contig SC. This contig SC now contains all DNPs and all reads specific for that particular repeat sequence.

5.2 Programming: Implementation in C++

The actual C++ code of the algorithm is presented in the Appendix A - The implementation of the algorithm made in C++.

5.3 Test Run Results

Some simple test runs of the algorithm have been done on data coming from a program simulating repeat sequences. The program simulates the production of repeat sequences with

different number of repeats, different repeat length, and with different seeds for generation of random numbers in the program.
Because all effort has been on making the algorithm to work correctly, the part of investigating how much faster it is have been limited or been taken away by time constraints.

A total of 5 different test runs were made. The results are presented in table 1.

Table1. Presentation of test run results.

No. repeats	Repeat length	No. contigs	No. spec contigs	No. contigs w. sub	No.of contigs w. errors	No. contigs not comp
10	1000	21	11	7	2	1
10	2000	24	12	8	1	1
15	2000	18	12	6	1	1
20	2000	24	16	7	1	2
5	2000	12	7	4	3	2

The first two columns contain the input data to the program simulating the repeat sequences.
Column three shows how many contigs that were produced in the test run of the algorithm.
Column four is how many specific contigs that were made in the test run.
The difference between column 3 and 4 depends on that some contigs are accually representing the same specific contig from one repeat sequence. (Figure 13.)

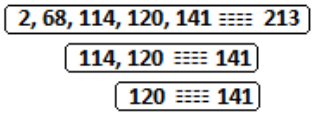

Figure 13: Contig consisting of three subparts. These three (number of (no.) contigs is 3) contigs are all representing one (no. spec contigs is 1) specific contig. This contig will also be one of the contigs with subcontigs, column 5.

Results where many contigs represent the same repeat region happen when the resolve process is done to rows containing SCs with high DNP ids before rows containing SCs with low DNP ids. In figure 13 the SC 120—141 is resolved before 114,120—140, and 2,…141—213 is resolved last.

Column 5 (no. contigs w. sub) represents how many specific contigs that is being represented by more than one subcontig, fig 13.

Column 6 shows how many contigs that contain an error DNP, fig 14.

Figure 14: Contigs containing error DNPs. Shows how two different DNP errors could look like. In the left case, DNP-id 69 is part of two different sequences. The first two coming from the same sequence and the last from another sequence. The SC 69,170 and 208 is resolved first, maybe 74 is an error DNP-id despite it been part of two SCs. In the right case there is also some kind of DNP error. 83 is not part the second SC. The SC 80 and 113 is resolved before the first SC. This can be an indication that 83 is an error DNP. Which of 83 or 113 is the error, or maybe both, can not be determind here.

This situation occurs when SCs contain a DNP which is not common for all SCs in the sequence area. It happens in the verification process. What caused this "error" DNP SC should be investigated more. This shows how important it is that the SCs made must be

correct. If an error occurs it is hard to decide which mismatch that is the real error. But the main strategy should be to take care of all problems as close to the source as possible.

The last column shows how many contigs that have not been chained to its fully length, see fig 15.

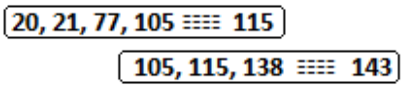

Figure 15: Contig not reaching full length. These two subcontigs have not been chained together to become a contig from 20 to 143. Situations like this are the worst result for the algorithm. These kind of result shows that the resolve process and the chaining process are not done the optimal way.

In addition this result happens in situations like the one separating column 3 and 4, i.e. when rows with higher DNP ids are resolved before those with lower DNP ids. For the strategy *make_resolve_order* to work, then the chain process must be changed.

5.4 Algorithm Summary – Objects, Data Structures and Functions

Objects and Data Structures:

Reads
SolidCluster
SolidClusterPositions
ClusterContainer: left_vec
right_vec
work_vec
rest_vec
resolve_vec

Functions Tree:

add_cluster( )
add_position( )

make_resolve_order( )

resolve_clusters( )
resolve_left_vec_row( )
compare_row( )
resolve_start( )
merge_clusters( )
compare_read_IDs( )
resolve_reads( )
check_right_end( )
add_read_IDs( )
add_work_clusters( )
add_work_clusters( )
compare_work( )
give_dnp_num_end( )
check_left_end( )

chain_clusters( )
check( )

merge_chains( )

5.5 Algorithm Problems and Errors

From the test runs it is clear that there are some things that must be futher investigated. The algorithm test result shows some problems. In one the repeat is represented by more than one contig and another where the repeat is not represented by at least one fully covered contig. These problems are a result of *left_vec* being very sensitive to how it is resolved.
The algorithm is also sensitive to false positive SCs. If the situation of getting false positive SCs is high then a better strategy must be developed to take care of these. If that part should be in algorithm or in the step before, the verification process, can be discussed.

6 Discussion

The way the algorithm works as of today is not satisfying. Some issues are still to be investigated. The question is, is it enough with some smaller changes to the current strategy or must the strategy involve a bigger change. Personally I think something in between can be a working alternative. Some information from this strategy can be used. Instead of making these subclusters on each row, where all kind of SCs are used, *one-step*, *two-step* and so on, just concentrate on using the high quality *one-step* SCs. Instead of trying to take longer steps with a higher risk of containing errors, take smaller steps with less error probability. If the longer steps can give any additional information to the clustring process, which *one-step* can not, it can be in finding DNP errors. By comparing quality of different steps, *one-step*, *two-step*, and so on, with the starting DNP in common, finding a shorter step having less quality then a longer step indicates an error. If this situation is found or even necessary in clustering with just *one-step* SCs remains unclear.

The problem with false positive SCs may be solved by increasing the number of reads to four to make the verification true.

In the strategy of using only *one-step* SCs maybe the quality value has to be more specific than just based on the number of reads, maybe even the number of bases between the DNPs can affect the value, a shorter distance higher value.

The current strategy is also very sensitive to the order of how things are done. There is a conflict between *left_vec's* structural order and its quality order. It turns out that something which is highly dependent on its structural order maybe was not so useful for this problem.

Also if only *one-step* SCs are used maybe the *right_vec* structure can be removed from the algorithm, or any way be used in a less central role, and there by making the algorithm less complex.

To be able to use this DNP strategy to build contigs each read must at least contain two DNPs. It is not only important for the verification process but also for the chaning process. One DNP must connect to the preceding SC and one DNP connect to the following SC. That is also in favour for just using *one-step* SCs to build contigs representing each repeat sequence.
Because this strategy of using DNPs is very dependent on the process of founding DNP candidates and also verifying DNP pairs, an investigation on what kind of difficulties that process can be involved in. Optimizing this part so that the making of false positive SCs are minimized or eliminated.

One thing that must be taken care of in the development of an algorithm is how to handle any possible false positive SCs. How to identify an error and how to eliminate it.

If you look at the big picture how this strategy is done, you can see the DNPs as dots you want to connect with lines. And of course you what to connect them in the right way. Maybe a new strategy in the area of graph theory is possible for solving this problem.

Acknowledgements

Thanks to,

Erik Arner and Martti Tammi, who came up with this project.

My good friend Daniel Oberg, who gave me valuable advice in the writing process.

References

[1] Tammi, MT. Center for Genomics and Bioinformatics, Karolinska Institutet, Stockholm, Sweden, February 2, (2003). The Principles of Shotgun Sequencing and Automated Fragment Assembly. (Special excerpt for lecture)

[2] Garg, K. February 10, (1998), Lecture Notes of Phil Green, *Lecture 11 – Sequence Assembly* (76 – 81)

[3] Boman, P. Master of Thesis (2001). Storskalig DNA-sekvensering av *Trypanosoma cruzi*.

[4] WWW-dokument: Copyright (C) 1993-2002 by Green, P and Ewing, B. http://www.phrap.org/phrap.docs/phred.html. Hämtad 2001

[5] Tammi, M T. Uppsala University (2002), Ph D Thesis: *Software Tools and Algorithms for Shotgun Sequence Assembly*.

[6] WWW-dokument: Copyright (C) 1994-1999 by Green, P. http://www.phrap.org/phrap.docs/phrap.html. Hämtad 2001

[7] Sharma, N. February 12, (1998), Lecture Notes of Phil Green, *Lecture 12 – Sequence Assembly, II – DRAFT* (82 – 87).

[8] Gordon, D., Abajian, C., Green, P. (1998) *Consed*: A Graphical Tool for Sequence Finishing. *Genome Research*, **8**, 195-202.

[9] Tammi MT, Arner E, Britton T, Andersson B. (2002) Separation of nearly identical repeats in shotgun assemblies using defined nucleotide positions, DNPs. *Bioinformatics* Mar;**18**(3):379-88

[10] Tammi, MT, Arner, E and Andersson, B. (2003) TRAP: Tandem Repeat Assembly Program produces improved shotgun assemblies of repetitive sequences. *Computer Methods and Programs in Biomedicine*, Jan:**70**(1):47-59.

Appendix A – The implementation of the algorithm made in C++


```
/*-------------------------------------------------------------------------
---------------*/
// Location: lennief/Xjobb/cluster
// Filename: ClusterPosition.h
// Auther: LENNIE FREDRIKSSON
// Project: Master of Thesis at Björn Andersson's group, KI.
// Project name:
// Supervisors: BJORN ANDERSSON & Erik Arner, Martti T. Tammi
// Start date: 2001 09 25
// Latest change date: 2002 04 03
// Program/Class/File information:
//
/*-------------------------------------------------------------------------
---------------*/

#ifndef CLUSTERCONTAINER_H
#define CLUSTERCONTAINER_H
#include "solidc.h"
#include "SolidClusterPosition.h"
#include <vector.h>

/*-------------------------------------------------------------------------
---------------*/

class ClusterContainer {
  // Methods.
 public:

  /*------------------------------------ construktors ------------------------
---------------*/

  /* ClusterContainer(); */

  /*----------------------------------- add_clusters() ----------------------
---------------*/

  void add_cluster(SolidCluster &s, int left); /* Adds a SolidCluster to the con
tainer. */
  void add_position(SolidClusterPosition &scp, int right); /* Adds s SolidCluste
rPosition to the right_vec. */
  void make_resolve_order(); /*  */

  /*---------------------------------- resolve_clusters() --------------------
---------------*/

  void resolve_clusters();
  void resolve_left_vec_row(int row);
  void compare_row(int row);
  int resolve_start(int row);
  void compare_read_IDs(SolidCluster &sca, SolidCluster &scb);
  void resolve_reads(SolidCluster &sca, SolidCluster &scb, int &tal);
  void compare_work(int row);
  void add_work_cluster(int row,int start_index, int index);
  void merge_clusters(int row, int &start_index, int index, int &control_number)
;
  void check_right_end(SolidCluster &temp, SolidCluster &sca, int right_DNP_numb
er);
  void add_read_IDs(SolidCluster &temp, SolidCluster &sca);
  void check_left_end(SolidCluster &s, int DNP_pos, int row);
  int give_dnp_num_end(SolidCluster &s, int dnp_number);

  /*---------------------------------- chain_clusters() ----------------------
```

```
---------------*/

  void chain_clusters();
  void merge_chains();
  int get_size();
  int get_num_clusters(int index);
  //    const SolidCluster& get_cluster(int row, int index);
  SolidCluster get_cluster(int row, int index);
  void check(SolidCluster &first_sc, SolidCluster &second_sc);
  void add(SolidCluster &s, int row, int column);
  SolidCluster left_check(SolidCluster &s, int right_DNP_pos);

  /*----------------------------------- Xtra functions() ----------------------
---------------*/

  void print( ostream & os );
  void print_reads( ostream & os );
  void print();
  void print_reads();
  void erase();
  void print_SolidCluster(SolidCluster &s);
  void get_reads_in_chain(vector<int> &v);
  void print_read_vector(vector<int> &v) const;
  void set_left_and_rightmost_DNP_nums(int row, int index, int read_index, int l
eft, int right);

  /*----------------------------------------------------------------------------
---------------*/

  // Members.
 private:

  vector< vector<SolidCluster> > left_vec; // left_vec[0] points to SolidCluster
s with left_DNP_number=0 etc.

  vector< vector<SolidClusterPosition> > right_vec;

  vector<SolidCluster> work_vec;

  vector<SolidCluster> rest_vec; // in this structur are SolidClusters put, left
overs from the resolving step.
  vector<SolidCluster> error_vec;

  vector< vector<SolidCluster> > resolve_vec; // this structur will keep the res
olve order.

  vector<int> error_readID_vec; // this structur will keep track of the readID f
rom the error reads.

  /*----------------------------------------------------------------------------
---------------*/
};

#endif

/*----------------------------------------------------------------------------
---------------*/
```

```
/*-------------------------------------------------------------------------------------------*/
// Location: lennief/Xjobb/erikcluster/
// Filename: ClusterContainer.cpp
// Auther: Lennie Fredriksson
// Project: X-arbete at Björn Andersson's group.
// Project name:
// Handledare: Erik Arner & Martti T. Tammi
// Start date: 2001 11 29
// Latest change date: 2002 04 16
/*==================================== File Information ======================================*/

/* Program/Class/File information:
   This information was written 2001 12 something. Allot have happened afterwards.
   The main information about the Program can be found i file Information_ClusterContainer.cpp.
   This PART have been tested for different kinds of data:
   ¤  IMPORTANT!!! One important thing for the user to think of is to add the SolidClusters to
   the ClusterContainer in increasing order of left_DNP_numbers. Just to get the SolidCluster-
   Positions in right_vec in right order, so the order in work_vec later on get right.
   ¤  It has been tested for data coming from just one repeat region and the result came out OK.
   The main-program used was "ProjectTest_one_repeat_region.cpp".
   DNPs from repeat region 1: (8) - 10 - 13 - 15 - 18 - 21 - 23 - 26 - 29 - 32.
   ¤  And it has been tested for data coming from different repeart regions
   (four different regions) and the result came out OK.
   The main-program used was "ProjectTest_four_repeat_region.cpp".
   DNPs from repeat region 1: (8)- 10 - 13 - 15 - 18 - 21 - 23 - 26 - 29 - 32.
   repeat region 2:  9 - 11 - 14 - 17 - 19 - 22 - 28 - 33.
   repeat region 3: 12 - 16 - 20 - 25 - 30.
   repeat region 4: 24 - 27 - 31 -(34).
   ¤  And it has been tested for data coming from different repeart regions (four different
   regions) and where one repeat region have had an NOT OK DNP position (repeat region 1 got
   DNP 14 to it), and the result came out quite OK. The NOT OK DNP was located in the inner
   part of the alignment, which makes it easier to resolve than if the NOT OK DNP had come from
   the first part of the alignment (then you have nothing to compare with).The main-program used
   was "ProjectTest.cpp".
   ¤  THE USE OF FLAGS in the function, add_work_cluster().
   Flags:
   10000: This flag indicates that the SolidCluster have been used in the resolving process.
   The SolidCluster have put in the work_vec for later use.
   20000: This flag indicates that the SolidCluster have been registered as a not okej
   SolidCluster.
   30000: This flag indicates that the SolidCluster have been added to the rest_vec.
   ¤  And it HAVE been tested for data, when the SolidCluster in the start have alot of reads
   containing another DNP-position or an unknown DNP-position. And the result turned out god.
   But more testing have to be done.  */

/*-------------------------------------------------------------------------------------------*/
#include "ClusterContainer.h"
#include <iostream.h>
#include <assert.h>
#include <stdlib.h>
/*==================================== constructors ==========================================*/
/* ClusterContainer::ClusterContainer(){} */
/*==================================== add_clusters() ========================================*/

/* This function will add SolidClusters to left_vec.
   At the same time this happens related SolidClusterPosition objects will be created and added
   to right_vec. Each SolidCluster added to the left_vec will be "creating" a related
   SolidClusterPosition added to the right_vec.
   When new SolidClusters are added to left_vec, two different cases can occur:
   1) left_vec have free space and
   2) no space are available so more space have to created. */

void ClusterContainer::add_cluster(SolidCluster &s, int left){
  for(int i = left_vec.size(); i < left+1; i++){
    vector<SolidCluster> temp;
    left_vec.push_back(temp);
  }
  left_vec[left].push_back(s);
  SolidClusterPosition temp_scp(left, left_vec[left].size()-1); // creats a SCP.
  add_position(temp_scp, s.get_right_DNP_num()); // put the SCP in right_vec.
}
```

```
/* This function will add SolidClusterPosition to right_vec.
   This function, add_position(...), looks like add_cluster(...)
   When new SolidClusterPosition are added to right_vec,
   even here two different cases can occur:
   1) right_vec have free space and
   2) no space are available so more space have to be created. */

void ClusterContainer::add_position(SolidClusterPosition &scp, int right){
  for(int i = right_vec.size(); i < right+1; i++){
    vector<SolidClusterPosition> temp;
    right_vec.push_back(temp);
  }
  right_vec[right].push_back(scp);
}

// This make_resolve_order() is build from left_vec.

void ClusterContainer::make_resolve_order(){
  cerr<<"make_resolve_order()"<<endl;
  for(int l = 0; l < left_vec.size(); l++){
    if(left_vec[l].size() != 0){

      // This for-loop, if the first SolidCluster desides!
      // for(int m = resolve_vec.size(); m < left_vec[l][0].get_num_reads()+1; m++){

      // This for-loop, if the best SolidCluster on the row desides!
      int best_element_on_row = resolve_start(l);
      for(int m = resolve_vec.size(); m<left_vec[l][best_element_on_row].get_num_reads()+1; m++){
        vector<SolidCluster> temp;
        resolve_vec.push_back(temp);
      }
      // resolve_vec[ left_vec[l][0].get_num_reads()].push_back(left_vec[l][0]);
      resolve_vec[left_vec[l][best_element_on_row].get_num_reads()].push_back(left_vec[l][0]);
    }
  }
}

/*================================== resolve_clusters() ====================================*/
/* This function resolves left_vec. The way this is done have changed during development. */

void ClusterContainer::resolve_clusters(){
  cout<<"\n============================== RESOLVE_CLUSTERS() ==============================\n";
  if(resolve_vec.size() > 0){
    for(int i = resolve_vec.size()-1; i > -1; i--){
      cerr<<"========= RAD "<<i<<" I RESOLVE_VEC ========= "<<endl;
      if(resolve_vec[i].size() != 0){
        for(int j = 0; j < resolve_vec[i].size(); j++){
          cerr<<"-------- Resolve rad "<<resolve_vec[i][j].get_left_DNP_num()<<" --------"<<endl;
          resolve_left_vec_row(resolve_vec[i][j].get_left_DNP_num());
        }
      }
    }
  }
  //-------------------------------------------------------------------------------------------
  // This part erase the empty rows in right_vec after the resolvning process.
  for(int i = 0; i < right_vec.size(); i++){
    if(right_vec[i].size() == 0){
      right_vec.erase(right_vec.begin() + i,right_vec.begin() + i + 1);
      i--;
    }
  }
}

// This function resolves a row i left_vec.

void ClusterContainer::resolve_left_vec_row(int row){
  cerr<<"~~~~~~~~~ COMPARE_ROW("<<row<<") ~~~~~~~~~"<<endl;
  compare_row(row); // resolves the row "row" in left_vec.
  cerr<<"~~~~~~~~~ COMPARE_WORK("<<row<<") ~~~~~~~~~"<<endl;
  compare_work(row); // resolves SolidClusters coming from the right_vec and put in the work_vec.
}
```

```
/* ************************************ compare_row() ************************************ */

/* When a row is compared TWO different cases can occur:

   CASE I.  Rows starting with a TAKEN SolidCluster (flag 10000) WILL NOT BE resolved.
   CASE II. Rows starting with a ERROR SolidCluster (flag 20000) WILL BE resolved.

   This function resolves SolidClusters in one row (the same row). At the same time are related
   SolidClusters (coming from the SolidClusterPositions in right_vec) added to the work_vec,
   rest_vec or just flagged if they are error-SolidCluster. The SolidClusters in work_vec are
   resolve later in compare_work(row).
   The way the resolve process is done is like follow: The first correct SolidCluster becomes
   the start_position SolidCluster. When two SolidClusters are resolved, the result ends up in
   the start_position SolidCluster. When the hole row is resolved, the start_position
   SolidCluster is added to the first position on that row. After that are SolidClusters put
   in work_vec resolved to the resolved row. */

void ClusterContainer::compare_row(int row){
  assert(row < left_vec.size());
  /* This integer, first_merge_control,
     contain the number of merge_cluster taking place on that row. */
  int first_merge_control = 0;
  int start_position = resolve_start(row);

  /* This if-sentence is a check that the start SolidCluster has an OK left_DNP_num (<10000) and
     that there are SPC for that SolidCluster in right_vec,
     otherwise this left_vec row i NOT CORRECT!!! */
  if(left_vec[row][start_position].get_left_DNP_num() < 10000 &&
     right_vec[left_vec[row][start_position].get_right_DNP_num()].size() != 0){

    if(left_vec[row].size() > start_position + 1){ // the following element on the row.
      for(int i = start_position + 1; i < left_vec[row].size(); i++){
        if(left_vec[row][i].get_left_DNP_num() < 10000)
          /* This function resolves the first element (element [0]) in row with
             element [i] on the same row. */
          merge_clusters(row, start_position, i, first_merge_control);
      }
    }
    if(first_merge_control == 0) // This means that this is the first merge on this row.
      add_work_cluster(row, start_position, start_position);
    // the resolved SolidCluster should always be in position zero on the row.
    left_vec[row][0] = left_vec[row][start_position];
  }

  /* The row is intepreted as NOT OK! */
  else if(left_vec[row][start_position].get_left_DNP_num() < 10000 &&
          right_vec[left_vec[row][start_position].get_right_DNP_num()].size() == 0){
    for(int j = 0; j < left_vec[row].size(); j++){
      error_vec.push_back(left_vec[row][j]);
      cerr<<"error_vec next!!! ---------- compare_row "<<endl;
      left_vec[row][j].set_left_DNP_num(20000);
    }
  }

  /* All SolidClusters are already used. */
  else
    cerr<<"raden är redan använd till sista SolidClustret !!"<<endl;
}

/* This function return the position i left_vec[row] for the SolidCluster which have most
   number of reads. */

int ClusterContainer::resolve_start(int row){
  cerr<<"resolve_start()"<<endl;
  int resolve_start = 0;
  while(left_vec[row][resolve_start].get_left_DNP_num() >= 10000 &&
        left_vec[row].size() > resolve_start+1){
    resolve_start++;
    cerr<<"row = "<<row<<" och resolve_start = "<<resolve_start<<endl;
  }
  if(left_vec[row].size() > resolve_start){
    for(int l=resolve_start+1; l < left_vec[row].size(); l++){
```

```
      if(left_vec[row][l].get_left_DNP_num() < 10000 &&
         left_vec[row][resolve_start].get_num_reads() < left_vec[row][l].get_num_reads())
        resolve_start = l;
    }
  }
  cerr<<"Row = "<<row<<" och Resolve_Start = "<<resolve_start<<endl;
  cerr<<"Antal reads i Start_SolidCluster = "<<left_vec[row][resolve_start].get_num_reads()<<endl
;
  for(int l=0; l < left_vec[row].size(); l++)
    cerr<<"Antal reads för SolidCluster("<<l<<") = "<<left_vec[row][l].get_num_reads()<<" och lef
t_DNP_number = "<<left_vec[row][l].get_left_DNP_num()<<endl;;
  return resolve_start;
}

/* This function should for each SolidCluster[row (row i left_vec), index (position/coloum in
   row)] look in right_vec on the row, related to the SolidCluster's right_DNP_number.
   The SolidClusters related there will be added to work_vec, rest_vec or not be used at all,
   that's depending on it's left_DNP_number. If the left_DNP_number is smaller than the
   left_vec's row the SolidCluster should be added to rest_vec otherwise it will be added to
   work_vec or not. SolidClusters not added to work_vec are/can be NOT OK SolidClusters.
   The SolidClusters in work_vec are later used in compare_work(row) and the SolidClusters in
   rest_vec are for now not used at all.

   The SolidClusters added to work_vec and rest_vec are copies of the SolidClusters in left_vec.
   The reason why copies are made is not to change the positions of the SolidClusters in
   left_vec. It is because each SolidCluster's position is "registrered"/mapped in some way in
   right_vec. So no changes in left_vec are allowed.
   The way of marking that a SolidCluster is "taken" is to set it's left_DNP_number to 10000.
   And if the SolidCluster is added to rest_vec it's left_DNP_number is set to 30000.
   And if the SolidCluster is an error, it's left_DNP_number is set to 20000. */

void ClusterContainer::add_work_cluster(int row, int start_index, int index){
  cerr<<"left_vec["<<row<<"].size() = "<<left_vec[row].size()<<" och Index = "<<index<<endl;
  assert(row < left_vec.size());
  assert(index < left_vec[row].size());
  int right_vec_row = left_vec[row][index].get_right_DNP_num();

  /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
  /* Go through the right_vec on row = right_vec_row. Start from the end and go to position 0. */
  for(int i = (right_vec[right_vec_row].size() - 1); i > -1; i--){
    /*_______________________________ add to rest_vec ______________________________________*/
    if(right_vec[right_vec_row][i].get_left_DNP_number() < row){
      /* Must control that the SolidClusters added to work_vec have bigger left_DNP_number than
         the row(number) they are related from. If they have smaller left_DNP_number they will
         be added to rest_vec. And if they have the same number it will be ignored, because
         that's the row being resolved. After that the SolidClusterPosition in right_vec is
         deleted.
         This part is not valid NOW! */
    }
    /*_______________________________ add to work_vec ______________________________________*/
    if(right_vec[right_vec_row][i].get_left_DNP_number() > row){
      /* check_number is an int-flag.
         When a SolidCluster has an OK left_DNP_number, check_number will be set to 99999. */
      int check_number = 0;

      /*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        +++++++++++++++++++++++ KOLLAR UPP ATT KORREKTA DNPs FINNS ++++++++++++++++++++++++++
        ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        In this part the left_DNP_number of the SolidCluster related from right_vec most be
        check that it is part of the resolved SolidCluster's DNPs being formed. If it's OK
        (the left_DNP_number is a part of the resolved SolidClusters DNPs) it will be added
        to work_vec, and if it's NOT OK it will just be flaged, 20000, and not be used. */

      if(start_index != index){
        for(int j = 0; j < left_vec[row][start_index].get_num_DNP_nums(); j++){
          if(left_vec[row][start_index].get_DNP_num(j) ==
             right_vec[right_vec_row][i].get_left_DNP_number()){
            work_vec.push_back(left_vec[right_vec[right_vec_row][i].get_left_DNP_number()][right_
vec[right_vec_row][i].get_left_vec_position()]);

            /* If the SolidCluster is used, added to work_vec, it will be flagged with a
               left_DNP_number equal to 10000. */
```

```
            left_vec[right_vec[right_vec_row][i].get_left_DNP_number()][right_vec[right_vec_row][
i].get_left_vec_position()].set_left_DNP_num(10000);

            check_number = 99999;
            break;
          }
        }
      }
      /*---------------------------------- add to error_vec ---------------------------------*/
      /* If the check_number is equal to 99999, the SolidCluster is part of the DNPs and
         have been added to work_vec, otherwise not and will just be flaged, 20000. */
      if( check_number != 99999){
        error_vec.push_back(left_vec[right_vec[right_vec_row][i].get_left_DNP_number()][right_vec
[right_vec_row][i].get_left_vec_position()]);

        // If the SolidCluster is wrong it will be flagged with a left_DNP_number equal to 20000.
        left_vec[right_vec[right_vec_row][i].get_left_DNP_number()][right_vec[right_vec_row][i].g
et_left_vec_position()].set_left_DNP_num(20000);
      }
    }
    /*_____________________________________ NO move of SC _____________________________________*/
    if(right_vec[right_vec_row][i].get_left_DNP_number() == row){}

    right_vec[right_vec_row].erase(right_vec[right_vec_row].begin() + i, right_vec[right_vec_row]
.begin() + i + 1);  // This part erases the SolidClusterPosition.
  }
}

/* This function resolves two SolidClusters. The SolidClusters are element [0] on row and
   element [index] on the same row and the result is then put in SolidCluster [0].
   In this function it will be investigated if the next SolidCluster on the row contains more,
   less or equal number of reads. The next SolidCluster should contain less or maybe equal
   number of reads. If it contains more number of reads something wrong maybe in the air.
   And if the number/procentage of equal reads is low, then an error is more probable to have
   happened. In this function there are TWO cases!! */

void ClusterContainer::merge_clusters(int row,int &start_index, int index, int &control_number){
  assert(row < left_vec.size());
  assert(index < left_vec[row].size());
  cerr<<"merge_cluster - start_index["<<row<<"]["<<start_index<<"] readnumber = "<<left_vec[row][
start_index].get_num_reads()<<" och index["<<row<<"]["<<index<<"] readnumber = "<<left_vec[row][i
ndex].get_num_reads()<<endl;

  /* This part investigate if the next SolidCluster contains less or equal number of reads than
     the former, which it should. Otherwise it is removed and replaced with the following/next
     SolidCluster. */
  if(left_vec[row][start_index].get_num_reads() < left_vec[row][index].get_num_reads()){
    cerr<<"---------- !!!!! OBSEVERA !!!!! ----------"<<endl;
    int common_reads = 0;
    for(int i = 0; i < left_vec[row][start_index].get_num_reads(); i++){
      for(int j = 0; j < left_vec[row][index].get_num_reads(); j++){
        if(left_vec[row][start_index].get_read_ID(i) == left_vec[row][index].get_read_ID(j)){
          common_reads++;
          break;
        }
      }
    }
    cerr<<"common reads = "<<common_reads<<endl;
    cerr<<"common read% = "<<(common_reads*1.0)/(left_vec[row][index].get_num_reads()*1.0)<<endl;
    cerr<<"control_number = "<<control_number<<endl;
    cerr<<"kvoten ( < 2.4) = "<<double(left_vec[row][start_index].get_num_reads())/double(left_ve
c[row][index].get_num_reads())<<endl;
    /* This part controls if common reads is the bigger part of the next SolidCluster
       (in this case 60%).
       If not, the FIRST SolidCluster is taken to be an error-SolidCluster. */

    if(double((common_reads*1.0)/(left_vec[row][index].get_num_reads()*1.0)) <= 0.6 || double(lef
t_vec[row][start_index].get_num_reads())/double(left_vec[row][index].get_num_reads()) > 2.4 ){
      error_vec.push_back(left_vec[row][index]);
      cerr<<"error_vec next!!! "<<endl;
      left_vec[row][index].set_left_DNP_num(20000);
    }
```

```
    else{
      cerr<<"===== CASE I: ====="<<endl;
      cerr<<"antal reads innan = "<<left_vec[row][start_index].get_num_reads()<<endl;
      compare_read_IDs(left_vec[row][start_index], left_vec[row][index]);
      control_number++;
      cerr<<"control_number CASE1 = "<<control_number<<endl;
      if(control_number == 1){
        add_work_cluster(row, start_index, start_index);
        add_work_cluster(row, start_index, index);
        left_vec[row][start_index].set_right_DNP_num(left_vec[row][index].get_right_DNP_num());
      }
      if(control_number > 1){
        add_work_cluster(row, start_index, index);
        left_vec[row][start_index].set_right_DNP_num(left_vec[row][index].get_right_DNP_num());
      }
      cerr<<"antal reads efter = "<<left_vec[row][start_index].get_num_reads()<<endl;
    }
  }
  else{
    cerr<<"===== CASE II: ====="<<endl;
    cerr<<"kvoten ( < 2.6 ) = "<<double(left_vec[row][start_index].get_num_reads())/double(left_v
ec[row][index].get_num_reads())<<endl;
    if(double(left_vec[row][start_index].get_num_reads())/double(left_vec[row][index].get_num_rea
ds()) < 2.6 ){
      cerr<<"kvoten ( < 2.6 ) = "<<double(left_vec[row][start_index].get_num_reads())/double(left
_vec[row][index].get_num_reads())<<endl;
      int common_reads = 0;
      for(int i = 0; i < left_vec[row][start_index].get_num_reads(); i++){
        for(int j = 0; j < left_vec[row][index].get_num_reads(); j++){
          if(left_vec[row][start_index].get_read_ID(i) == left_vec[row][index].get_read_ID(j)){
            common_reads++;
            break;
          }
        }
      }
      cerr<<"common reads = "<<common_reads<<endl;
      cerr<<"antal reads innan = "<<left_vec[row][start_index].get_num_reads()<<endl;
      compare_read_IDs(left_vec[row][start_index], left_vec[row][index]);
      control_number++;
      cerr<<"control_number CASE2 = "<<control_number<<endl;
      // the first resolve-process on the row differs a little from the second resolve-process.
      if(control_number == 1){
        /* adds related SolidClusters coming from the first SolidCluster in left_vec with a
           left_DNP_number other then 10000,20000 or 30000 (element j) in row to work_vec. */
        //cerr<<"row = "<<row<<", start_index = "<<start_index<<" och index = "<<index<<endl;
        add_work_cluster(row, start_index, start_index);
        add_work_cluster(row, start_index, index);
        left_vec[row][start_index].set_right_DNP_num(left_vec[row][index].get_right_DNP_num());
      }
      if(control_number > 1){
        // add work-cluster to work_vec coming from "right_vec"(row, i)
        add_work_cluster(row, start_index, index);
        left_vec[row][start_index].set_right_DNP_num(left_vec[row][index].get_right_DNP_num());
      }
      cerr<<"antal reads efter = "<<left_vec[row][start_index].get_num_reads()<<endl;
    }
  }
}

/*---------------------------------- compare_read_IDs() ----------------------------------*/

/* This function compares read_IDs between two SolidClusters. These two SolidClusters are also
   the parameters of the function. THE RESULT WILL BE ADDED TO THE SOLIDCLUSTER SCA.
   The right_DNP_number_control is a number which keep control if the SolidClusters compared
   have common reads. Just to be shore if the resulting SolidCluster should have sca's
   right_DNP_number (right_DNP_number_control == 0) or scb's right_DNP_number
   (right_DNP_number_control > 0). */

void ClusterContainer::compare_read_IDs(SolidCluster &sca, SolidCluster &scb){
  int right_DNP_number_control = 0;
  resolve_reads(sca, scb, right_DNP_number_control);
}
```

```
/* This function take two SolidClusters and compare there read_IDs (sca -> scb). Save common
   reads and investigate if not common reads should be saved or not.
   THE RESULT WILL BE ADDED TO SCA. */

void ClusterContainer::resolve_reads(SolidCluster &sca, SolidCluster &scb, int &control_number){
  SolidCluster temp_solidcluster(sca.get_left_DNP_num(), sca.get_right_DNP_num());
  assert(scb.get_num_reads() > 0);
  for(int i=0; i < scb.get_num_reads(); i++){ // the outer for-loop. scb's reads.
    assert(sca.get_num_reads() >= 0);
    for(int j=0; j < sca.get_num_reads(); j++){ // the inner for-loop. sca's reads.
      if(scb.get_read_ID(i) == sca.get_read_ID(j)){
        /* scb.read_IDs[i] = int and means: that's the number of the read which is placed in
           read_IDs on position i. If the read is common, it will be added to the temporary
           SolidCluster temp_solidcluster. */
        temp_solidcluster.add_read(sca.get_read_ID(j),sca.get_leftmost_DNP_num(j),sca.get_rightmo
st_DNP_num(j));
        /* The read is added to the temp_solidcluster.
           This event is registered by the right_DNP_number_control. */
        control_number ++;
        sca.delete_read(j);
        /* If the read is common it will be added to temp_solidcluster and then deleted from
           sca, just so it will be easier to see what reads in sca who's not common.
           These reads will be further investigated. */
        break; // breaks the inner for-loop.
      }
    }
  }
  if(sca.get_num_reads() != 0) // Futher investigation of not common reads.
    check_right_end(temp_solidcluster, sca, scb.get_right_DNP_num());
  add_read_IDs(temp_solidcluster, sca); // The result is put in sca (element [0] in row).
}

/* This function investigates not common reads.
   If they end before or after next DNP position, and if they should be saved or not. */

void ClusterContainer::check_right_end(SolidCluster &temp, SolidCluster &sca, int right_DNP_numbe
r){
  while(sca.get_num_reads() > 0) {
    if(sca.get_rightmost_DNP_num(0) < right_DNP_number){
      /* Check if read[i] stops before next DNP-position. If it does, the read is correct, and
         it will be added to the temporary SolidCluster. */
      temp.add_read(sca.get_read_ID(0),sca.get_leftmost_DNP_num(0),sca.get_rightmost_DNP_num(0));
      // sca.get_read_ID(i) gives the read number at position [i] in sca's vector<int> read_IDs.
    }
    else{
      // If the read is NOT correct, it's readID will be added to error_readID_vec.
      cerr<<"check_right_end() - L461 sca.get_read_ID(0) = "<<sca.get_read_ID(0)<<endl;
      error_readID_vec.push_back(sca.get_read_ID(0));
    }
    sca.delete_read(0); // the read is then deleted from SolidCluster sca.
  }
}

/* This function will add the resulting reads in SolidCluster temp to a SolidCluster sca */

void ClusterContainer::add_read_IDs(SolidCluster &temp, SolidCluster &sca){
  assert(sca.get_num_reads() == 0); // check that sca has no reads, that's important.
  for(int i = 0; i < temp.get_num_reads(); i++)
    sca.add_read(temp.get_read_ID(i),temp.get_leftmost_DNP_num(i),temp.get_rightmost_DNP_num(i));
}

/*-------------------------------------------------------------------------------------------*/
/* ************************************ compare_work() ************************************ */

/* This function will compare SolidClusters placed in work_vec with the resolved SolidClusters
   in left_vec. And the control is made with: check_left_end().
   This work_vec will work as a queue, first-in-first-out. So when a element is done it's
   deleted and. That's why always the element [0] is taken. */

void ClusterContainer::compare_work(int row){
  cerr<<"work_vec.size() = "<<work_vec.size()<<endl;
```

```
  if(work_vec.size() == 0){
    cerr<<"0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000"<<endl;

    /* If a resolved row not have related SolidCluster in work_vec, you can suspect that
       something is wrong or less good. */

  }
  while(work_vec.size() > 0){
    int dnp_num_end = give_dnp_num_end(left_vec[row][0], work_vec[0].get_left_DNP_num());
    /* at this time the left_vec[row][0]-element will contain all the different DNP's for the
       resolved row SolidCluster. The DNP-numbers must be placed in DNP_nums in increased order.
       First all common reads are deleted from the SolidCluster from work_vec. */
    // The outer for-loop, temp_wsc reads.
    for(int j=0; j < work_vec[0].get_num_reads(); j++){
      // The inner for-loop, left_vec[row]'s last elements reads.
      for(int k=0; k < left_vec[row][0].get_num_reads(); k++){
        // This part delete reads which are common.
        if(work_vec[0].get_read_ID(j) == left_vec[row][0].get_read_ID(k)){
          work_vec[0].delete_read(j);
          j--;
          break;
        }
      }
    }
    /* Second, this part take care of the not common reads. */
    if(work_vec[0].get_num_reads() != 0){
      /* if the read in temp_wsc (the outer for-loop) is not common it should be investigated
         that it ends before the former DNP number. */
      check_left_end(work_vec[0], dnp_num_end, row);
    }
    // erase the first SolidCluster i work_vec.
    work_vec.erase(work_vec.begin(), work_vec.begin() + 1);
  }
  cerr<<"RESULTAT efter WORK_VEC - antal reads kvar = "<<left_vec[row][0].get_num_reads()<<endl;
}

/* This function will return the DNP position from the vector<int> DNP_nums which is before the
   DNP position given as a parameter. The vector DNP_nums always comes from the resolved row. */

int ClusterContainer::give_dnp_num_end(SolidCluster &s, int dnp_number){
  assert(s.get_num_DNP_nums() >= 2);
  for(int i=1; i < s.get_num_DNP_nums(); i++)
    if(dnp_number == s.get_DNP_num(i)){
      assert(s.get_DNP_num(i-1) >= 0);
      return s.get_DNP_num(i-1);
    }
}

/* This function will investigate not common reads, if they end or not end before the former DNP
   number (DNP_num). If they end before, they will be added to left_vec[row][0].read_IDs. */

void ClusterContainer::check_left_end(SolidCluster &s, int DNP_num, int row){
  for(int i = 0; i < s.get_num_reads(); i++){
    if(s.get_leftmost_DNP_num(i) > DNP_num){
      left_vec[row][0].add_read(s.get_read_ID(i),s.get_leftmost_DNP_num(i),s.get_rightmost_DNP_nu
m(i));
    }
    else{
      // this part collect error readIDs, just for a check !!!!!!!
      cerr<<"check_left_end() - L537 s.get_read_ID(i) = "<<s.get_read_ID(i)<<endl;
      error_readID_vec.push_back(s.get_read_ID(i));
    }
  }
}

/*****************************************************************************************/
/*==================================== chain_clusters() ====================================*/

/* This function will put together the resolved SolidCluster formed. This will be done as follow:
   First find a row which is resolved in left_vec, start from the top of left_vec.
   The SolidCluster on that row will give information to which row to go to next.
   That information comes from it's right_DNP_number. The SolidCluster on the next row will be
```

```
   added to the first row after the first SolidCluster. This will be done until a row is empty.
   Each time this is done (that a SolidCluster is added to the first row), it must be CHECKED
   that no reads in the next SolidCluster reach through the former SolidCluster's
   right_DNP_position. This control will be a form of left_check().
   This function will also delete used SolidClusters. */

void ClusterContainer::chain_clusters(){
  cout<<"\n================================ CHAIN_CLUSTERS() ================================\n";
  for(int i = 0; i < left_vec.size(); i++){ // i = the row in left_vec which is being chained.
    int j = 0;
    if(left_vec[i].size() != 0 &&
       (left_vec[i][0].get_left_DNP_num() == 10000 ||
        left_vec[i][0].get_left_DNP_num() == 20000 ||
        left_vec[i][0].get_left_DNP_num() == 30000)){

      /* This part erase all SolidClusters which have been used,
         that have a left_DNP_number equal to 10000 or bigger. */
      left_vec[i].erase(left_vec[i].begin(), left_vec[i].end());
    }

    /* This should investigate that the row [i] in left_vec is not empty and that the first
       element is not already used. */
    if(left_vec[i].size() != 0 && left_vec[i][0].get_left_DNP_num() < 10000){
      left_vec[i].erase(left_vec[i].begin() + 1, left_vec[i].end());

      /* This part erase all SolidClusters except the first one in the row [i]. */
      int right_dnp_num = left_vec[i][0].get_right_DNP_num();

      /* This should investigate that the row [right_dnp_num] in left_vec is not empty and that
         the next row in left_vec exists. */
      while(right_dnp_num < left_vec.size() &&
            left_vec[right_dnp_num].size() != 0 &&
            left_vec[right_dnp_num][0].get_left_DNP_num() < 10000){

        /* THE FIRST ELEMENT LEFT_VEC[I][0] AND
           THE SECOND ELEMENT LEFT_VEC[RIGHT_DNP_NUM][0].

           the check_function should remove reads from the first element (=SolidCluster) which
           reach through the next DNP-number in the next resolved SolidCluster AND remove reads
           from the second element which reach through the former DNP-number in the former
           resolved SolidCluster. This should be done like follow:

           FIRST RIGHT_CHECK (First SolidCluster -> Second SolidCluster):
           Now the reads from the first SolidCluster are controlled if they have a
           rightmost_DNP_num bigger than the next SolidCluster's left_DNP_num, and if that's
           the case, the reads have to be checked if they also have a rightmost_DNP_num even
           bigger or equal to the DNP-number to the right of that left_DNP_number. If both cases
           occur, then the reads should be members of the next SolidCluster. So if a read have
           both cases and is member of two SolidClusters, it's deleted from the next
           SolidCluster, and if it has both cases and is not a member of the next SolidCluster
           it is deleted from the first SolidCluster.

           SECOND LEFT_CHECK (Second Solidcluster -> First SolidCluster):
           Now the reads from the second SolidCluster are controlled if they have a
           leftmost_DNP_num less than the former SolidCluster's right_DNP_num, and if that's
           the case, the reads have to be checked if they also have a leftmost_DNP_num even
           smaller or equal to the DNP-number to the left of that right_DNP_number. If both cases
           occur, then the reads should be members of the next SolidCluster. So if a read have
           both cases and is member of two SolidClusters, it's deleted from the former
           SolidCluster, and if it has both cases and is not a member of the former SolidCluster
           it is deleted from the second SolidCluster. */

        if(left_vec[right_dnp_num][0].get_left_DNP_num() < 10000){
          left_vec[right_dnp_num].erase(left_vec[right_dnp_num].begin() + 1,left_vec[right_dnp_nu
m].end());
          check(left_vec[i][j], left_vec[right_dnp_num][0]);
          j++;
          left_vec[i].push_back(left_vec[right_dnp_num][0]);
          // left_vec[right_dnp_num].erase(left_vec[right_dnp_num].begin(), left_vec[right_dnp_nu
m].begin() + 1);
          assert(left_vec[right_dnp_num][0].get_right_DNP_num() != 0);
          left_vec[right_dnp_num][0].set_left_DNP_num(10000);
```

```
          right_dnp_num = left_vec[i][j].get_right_DNP_num();
        }
      }
    }
  }
}

void ClusterContainer::check(SolidCluster &first_sc, SolidCluster &second_sc){
  /*========== FIRST, RIGHT_CHECK: The read in first_sc which is not OK is deleted. ==========*/
  for(int i=0; i < first_sc.get_num_reads(); i++){ // check read [i] in first SolidCluster.
    /* this part checks if the read reaches through the next SolidClusters left_DNP_num, if so
       it must be futher controlled. */
    if(first_sc.get_rightmost_DNP_num(i) > second_sc.get_left_DNP_num()){
      int k=0;
      /* this part checks if the read reaches through even the second next DNP_num,
         if so it must be a member of the next SolidCluster. */
      if(first_sc.get_rightmost_DNP_num(i) >= second_sc.get_DNP_num(1)){
        /* check if read is a member in second SolidCluster. */
        for(int j=0; j < second_sc.get_num_reads(); j++){
          /* if this is true, read [i] in SolidCluster first_sc is a member of SolidCluster
             second_sc, and this advent is registered with the int k.. And just continue with
             next read in first_sc.  */
          if(first_sc.get_read_ID(i) == second_sc.get_read_ID(j)){
            k++;
            second_sc.delete_read(j);
            break;
          }
        }
        /* if k=0, then the read is not common for the two SolidCluster and it will be deleted.
           That's because it's missing one importanted DNP-number */
        if( k == 0 ){
          cerr<<"check(right) L651 first_sc.get_read_ID(i) = "<<first_sc.get_read_ID(i)<<endl;
          error_readID_vec.push_back(first_sc.get_read_ID(i));
          first_sc.delete_read(i);
          i--;
        }
      }
    }
  }
  /*========== SECOND, LEFT_CHECK: The read in second_sc which is not OK is deleted. ==========*/
  for(int i=0; i < second_sc.get_num_reads(); i++){ // check read [i] in second SolidCluster.
    /* this part checks if the read reaches through the former SolidClusters right_DNP_num,
       if so it must be futher controlled. */
    if(second_sc.get_leftmost_DNP_num(i) < first_sc.get_right_DNP_num()){
      int k=0;
      /* this part checks if the read reaches through even the second former DNP_num,
         if so it must be a member of the former/first SolidCluster. */
      if(second_sc.get_leftmost_DNP_num(i) <= first_sc.get_DNP_num(first_sc.get_num_DNP_nums() -
2)){
        // check if read is a member in first SolidCluster.
        for(int j=0; j < first_sc.get_num_reads(); j++){
          /* if this is true, read [i] in SolidCluster second_sc is a member of SolidCluster
             first_sc. And just continue with next read in second_sc.  */
          if(second_sc.get_read_ID(i) == first_sc.get_read_ID(j)){
            k++;
            second_sc.delete_read(i);
            break;
          }
        }
        // if k=0, then the read is not common for the two SolidCluster and it will be deleted.
        if( k == 0 ){
          cerr<<"check(left) L680 second_sc.get_read_ID(i) = "<<second_sc.get_read_ID(i)<<endl;
          error_readID_vec.push_back(second_sc.get_read_ID(i));
          second_sc.delete_read(i);
          i--;
        }
      }
    }
  }
}
/*-------------------------------------------------------------------------------------------*/
/*==================================== merge_chains() ======================================*/
```

```
void ClusterContainer::merge_chains(){
  cout<<"\n================================ MERGE_CHAINS() ================================\n";
  for(int i = 0; i < left_vec.size(); i++){ // i = the row in left_vec which is being chained.
    if(left_vec[i].size() >= 2){
      for(int j = 1; j < left_vec[i].size(); j++){
        for(int k = 1; k < left_vec[i][j].get_num_DNP_nums(); k++)
          left_vec[i][0].add_DNP_num(left_vec[i][j].get_DNP_num(k));
        for(int l = 0; l < left_vec[i][j].get_num_reads(); l++)
          left_vec[i][0].add_read(left_vec[i][j].get_read_ID(l), left_vec[i][j].get_leftmost_DNP_
num(l), left_vec[i][j].get_rightmost_DNP_num(l));
      }
      left_vec[i][0].set_right_DNP_num(left_vec[i][0].get_DNP_num(left_vec[i][0].get_num_DNP_nums
()-1));
      left_vec[i][0].remove_DNP_num(left_vec[i][0].get_DNP_num(left_vec[i][0].get_num_DNP_nums()-
1));
      left_vec[i].erase(left_vec[i].begin()+1, left_vec[i].end());
    }
  }
}

/*================================== Xtra functions() ==================================*/

/* Here are some Xtra functions written for the testrun.
   This function will print the structur of the SolidCluster-element in left_vec, right_vec,
   resolve_vec, work_vec and rest_vec. Every SolidCluster-element will be represented by a "*"
   and each SolidClusterPosition in right_vec will be represented by a "+". */

void ClusterContainer::print(){
  // printing left_vec structur.
  cout <<"LEFT_VEC: \n";
  for(int i = 0; i < left_vec.size(); i++){
    if ( left_vec[i].size() < 1 ) continue;
    cout <<"rad " << i <<": ";
    for(int j = 0; j < left_vec[i].size(); j++)
      cout <<"SC: "<<left_vec[i][j].get_left_DNP_num()<<" "<<left_vec[i][j].get_right_DNP_num()<<
" ";
    cout <<"\n";

  }
  cout <<"\n";
  // printing right_vec structur.
  cout <<"RIGHT_VEC: \n";
  for(int i = 0; i < right_vec.size(); i++){
    if ( right_vec[i].size() < 1 ) continue;
    cout <<"rad " << i <<": ";
    for(int j = 0; j < right_vec[i].size(); j++)
      cout <<"SCP: "<<right_vec[i][j].get_left_DNP_number()<<" "<<right_vec[i][j].get_left_vec_po
sition()<<" ";
    cout<<"\n";
  }
  cout <<"\n";
  // printing resolve_vec struktur.
  cout <<"RESOLVE_VEC: \n";
  for(int i = 0; i < resolve_vec.size(); i++){
    if ( resolve_vec[i].size() < 1 ) continue;
    cout <<"rad " << i <<": ";
    for(int j = 0; j < resolve_vec[i].size(); j++)
      cout <<"SC: "<<resolve_vec[i][j].get_left_DNP_num()<<" "<<resolve_vec[i][j].get_right_DNP_n
um()<<" ";
    cout<<"\n";
  }
  cout <<"\n";
  // printing work_vec struktur.
  cout <<"WORK_VEC: \n";
  for(int i = 0; i < work_vec.size(); i++)
    cout <<"*\n";
  cout <<"\n";
  // printing rest_vec structur.
  cout <<"REST_VEC: \n";
  for(int i = 0; i < rest_vec.size(); i++){
    cout <<"SC: "<<rest_vec[i].get_left_DNP_num()<<" "<<rest_vec[i].get_right_DNP_num();
```

```
    cout<<"\n";
  }
  cout <<"\n";
  // printing error_vec structur.
  cout <<"ERROR_VEC: \n";
  for(int i = 0; i < error_vec.size(); i++){
    cout <<"SC: "<<error_vec[i].get_left_DNP_num()<<" "<<error_vec[i].get_right_DNP_num();
    cout<<"\n";
  }
  cout <<"\n";
  // printing readID_vec structur.
  cout <<"ERROR_READID_VEC: \n";
  for(int i = 0; i < error_readID_vec.size(); i++){
    cout <<"readID: "<<error_readID_vec[i];
    cout<<"\n";
  }
  cout <<"\n";
}

void ClusterContainer::print( ostream & os ){
  // printing left_vec structur.
  os <<"LEFT_VEC: \n";
  for(int i = 0; i < left_vec.size(); i++){
    if ( left_vec[i].size() < 1 ) continue;
    os <<"rad " << i <<": ";
    for(int j = 0; j < left_vec[i].size(); j++)
      os <<"SC: "<<left_vec[i][j].get_left_DNP_num()<<" "<<left_vec[i][j].get_right_DNP_num()<<"
";
    os <<"\n";
  }
  os <<"\n";
  // printing right_vec structur.
  os <<"RIGHT_VEC: \n";
  for(int i = 0; i < right_vec.size(); i++){
    if ( right_vec[i].size() < 1 ) continue;
    os <<"rad " << i <<": ";
    for(int j = 0; j < right_vec[i].size(); j++)
      os <<"SCP: "<<right_vec[i][j].get_left_DNP_number()<<" "<<right_vec[i][j].get_left_vec_posi
tion()<<" ";
    os <<"\n";
  }
  os <<"\n";
  // printing work_vec struktur.
  os <<"WORK_VEC: \n";
  for(int i = 0; i < work_vec.size(); i++)
    os <<"*\n";
  os <<"\n";
  // printing rest_vec structur.
  os <<"REST_VEC: \n";
  for(int i = 0; i < rest_vec.size(); i++){
    os <<"SC: "<<rest_vec[i].get_left_DNP_num()<<" "<<rest_vec[i].get_right_DNP_num();
    os <<"\n";
  }
  os <<"\n";
  // printing error_vec structur.
  os <<"ERROR_VEC: \n";
  for(int i = 0; i < error_vec.size(); i++){
    os <<"SC: "<<error_vec[i].get_left_DNP_num()<<" "<<error_vec[i].get_right_DNP_num();
    os <<"\n";
  }
  os <<"\n";
  // printing readID_vec structur.
  os <<"ERROR_READID_VEC: \n";
  for(int i = 0; i < error_readID_vec.size(); i++){
    os <<"readID: "<<error_readID_vec[i];
    os <<"\n";
  }
  os <<"\n";
}

// This function prints the reads coming from the resulted CONTIG.
```

```
void ClusterContainer::print_reads(){
  for(int i=0; i < left_vec.size(); i++){
    if(left_vec[i].size() != 0){
      cout <<"Reads in SolidCluster on row "<<i<<"\n";
      cout <<"The row have "<<left_vec[i].size()<<" SolidCluster \n";
      for(int j=0; j < left_vec[i].size(); j++){
        cout <<"The read in SolidCluster "<<j<<" are follow: \n";
        cout <<"DNP number's are ";
        for(int l=0; l < left_vec[i][j].get_num_DNP_nums(); l++)
          cout <<left_vec[i][j].get_DNP_num(l)<<" ";
        cout <<"and the\nread numbers are: ";
        for(int k=0; k < left_vec[i][j].get_num_reads(); k++)
          cout <<left_vec[i][j].get_read_ID(k)<<",";
        cout<<"\n\n";
        cout<<"leftmost:"<<endl;
        for( int k = 0; k < left_vec[i][j].get_num_reads(); k++ )
          cout<<left_vec[i][j].get_leftmost_DNP_num( k )<<"\t";
        cout<<endl;
        cout<<"rightmost:"<<endl;
        for( int k = 0; k < left_vec[i][j].get_num_reads(); k++ )
          cout<<left_vec[i][j].get_rightmost_DNP_num( k )<<"\t";
        cout<<endl;
      }
    }
  }
  cout <<"\n";
}

void ClusterContainer::print_reads( ostream & os ){
  for(int i=0; i < left_vec.size(); i++){
    if(left_vec[i].size() != 0){
      os <<"Reads in SolidCluster on row "<<i<<"\n";
      os <<"The row have "<<left_vec[i].size()<<" SolidCluster \n";
      for(int j=0; j < left_vec[i].size(); j++){
        os <<"The read in SolidCluster "<<j<<" are follow: \n";
        os <<"DNP number's are ";
        for(int l=0; l < left_vec[i][j].get_num_DNP_nums(); l++)
          os <<left_vec[i][j].get_DNP_num(l)<<" ";
        os <<"and the\nread numbers are: ";
        for(int k=0; k < left_vec[i][j].get_num_reads(); k++)
          os <<left_vec[i][j].get_read_ID(k)<<",";
        os<<"\n\n";
        os<<"leftmost:"<<endl;
        for( int k = 0; k < left_vec[i][j].get_num_reads(); k++ ) {
          os<<left_vec[i][j].get_leftmost_DNP_num( k )<<"\t";
        }
        os<<endl;
        os<<"rightmost:"<<endl;
        for( int k = 0; k < left_vec[i][j].get_num_reads(); k++ ) {
          os<<left_vec[i][j].get_rightmost_DNP_num( k )<<"\t";
        }
        os<<endl;
      }
    }
  }
  os <<"\n";
}

// This function prints the reads coming from the resulted SolidCluster.

void ClusterContainer::print_SolidCluster(SolidCluster &s){
  for(int i=0; i < s.get_num_DNP_nums(); i++){
    cerr<<"SolidCluster resolved : ";
    for(int j=0; j < s.get_num_reads(); j++)
      cerr<<s.get_read_ID(j)<<",";
    cerr<<"\n";
  }
}

/* This function erase the ClusterContainer's inner structurs, the left_vec, the right_vec,
   the resolve_vec, the work_vec and the rest_vec. */
```

```
void ClusterContainer::erase(){
  cout<<"=================================== ERASE() ===================================\n";
  // ------------ erase left_vec --------------------
  while(left_vec.size() > 0){
    while(left_vec[0].size() > 0)
      left_vec[0].erase(left_vec[0].begin(), left_vec[0].begin() + 1);
    left_vec.erase(left_vec.begin(), left_vec.begin() + 1);
  }
  // ------------ erase right_vec -------------------
  while(right_vec.size() > 0){
    while(right_vec[0].size() > 0)
      right_vec[0].erase(right_vec[0].begin(), right_vec[0].begin() + 1);
    right_vec.erase(right_vec.begin(), right_vec.begin() + 1);
  }
  // ------------ erase resolve_vec -----------------
  for(int i = 0; i < resolve_vec.size(); i++)
    resolve_vec[i].erase(resolve_vec[i].begin(), resolve_vec[i].end());
  resolve_vec.erase(resolve_vec.begin(), resolve_vec.end());
  // ------------ erase work_vec --------------------
  while(work_vec.size() > 0)
    work_vec.erase(work_vec.begin(), work_vec.begin() + 1);
  // ------------ erase rest_vec --------------------
  rest_vec.erase(rest_vec.begin(), rest_vec.end());
  // ------------ erase error_vec -------------------
  error_vec.erase(error_vec.begin(), error_vec.end());
  //------------ erase readID_vec -------------------
  error_readID_vec.erase(error_readID_vec.begin(), error_readID_vec.end());
  cout<<"================================= DONE ERASING ================================\n";
}

/* This function will give a list/vector with all reads part of the CONTIG/chain on the row. */

void ClusterContainer::get_reads_in_chain(vector<int> &v){
  for(int i=0; i < left_vec.size(); i++)
    if(left_vec[i].size() != 0)
      for(int j=0; j < left_vec[i].size(); j++)
        for(int k=0; k < left_vec[i][j].get_num_reads(); k++)
          v.push_back(left_vec[i][j].get_read_ID(k));
}

void ClusterContainer::print_read_vector(vector<int> &v) const {
  cout<<"Read i CONTIG : ";
  for(int i=0; i < v.size(); i++)
    cout<<v[i]<<",";
  cout<<"\n\n";
}

/*-------------------------------------------------------------------------------------------*/

int ClusterContainer::get_size(){
  return left_vec.size();
}

int ClusterContainer::get_num_clusters(int index){
  assert( index < (int)left_vec.size() );
  return left_vec[index].size();
}

//const SolidCluster& ClusterContainer::get_cluster(int row, int index){
//assert( row < (int)left_vec.size() );
//assert( index < (int)left_vec[row].size() );
//return left_vec[row][index];
//}

SolidCluster ClusterContainer::get_cluster(int row, int index){
  assert( row < (int)left_vec.size() );
  assert( index < (int)left_vec[row].size() );
  return left_vec[row][index];
}
void ClusterContainer::set_left_and_rightmost_DNP_nums(int row, int index, int read_index, int le
ft, int right)
{
```

```
  assert( row < (int)left_vec.size() );
  assert( index < (int)left_vec[row].size() );
  assert( read_index < left_vec[row][index].get_num_reads() );

  left_vec[row][index].set_leftmost_DNP_num( read_index, left );
  left_vec[row][index].set_rightmost_DNP_num( read_index, right );
}
```

```
/*-------------------------------------------------------------*/
// Location: lennief/Xjobb/
// Filename: SolidClusterPosition.h
// Auther: Lennie Fredriksson
// Project: X-arbete at Björn Andersson's group.
// Project name:
// Handledare: Erik Arner & Martti T. Tammi
// Start date: 2001 10 13
// Latest change date: 2001 11 15
// Program/Class/File information:
//
/*-------------------------------------------------------------*/

#ifndef SOLIDCLUSTERPOSITION_H
#define SOLIDCLUSTERPOSITION_H
#include <vector.h>

/*-------------------------------------------------------------*/

class SolidClusterPosition {
 public:
  SolidClusterPosition(int left_num, int left_pos);
  int get_left_DNP_number() const;
  int get_left_vec_position() const;

 private:
  int left_DNP_number, left_vec_position;

};
#endif

/*-------------------------------------------------------------*/
```

```
/*-------------------------------------------------------------*/
// Location: lennief/Xjobb/
// Filename: SolidClusterPosition.cpp
// Auther: Lennie Fredriksson
// Project: X-arbete at Björn Andersson's group.
// Project name:
// Handledare: Erik Arner & Martti T. Tammi
// Start date: 2001 10 15
// Latest change date: 2001 11 15
// Program/Class/File information:
//
/*-------------------------------------------------------------*/

#include "SolidClusterPosition.h"
#include <iostream.h>
#include <assert.h>

/*-------------------------------------------------------------*/

// kontruktor.
SolidClusterPosition::SolidClusterPosition(int left_num, int
left_pos) {
  left_DNP_number = left_num;
  left_vec_position = left_pos;
}

// returnerar SolidClusterPosition elementeta left_vec
position/rad.
int SolidClusterPosition::get_left_DNP_number() const {
  return left_DNP_number;
}

// returnerar SolidClusterPosition elementets left_vec
position/kolumn.
int SolidClusterPosition::get_left_vec_position() const {
  return left_vec_position;
}

/*-------------------------------------------------------------*/
```