INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Error Detection in Pedigrees Using Satisfiability-Based Approaches

David Manuel Nunes Martins

Dissertation for the degree of

Master of Science in Information Systems and Computer Engineering

Jury

President: Prof. Ana Maria Severino de Almeida e Paiva
Research Advisor: Prof. Maria Inês Camarate de Campos Lynce de Faria
Committee: Prof. Sara Alexandra Cordeiro Madeira
Prof. Vasco Miguel Gomes Nunes Manquinho

November 2009

Resumo

No campo da biologia computacional, um pedigree é uma estrutura que permite a monitorização de individuos e correspondentes relações familiares. Com pedigrees, torna-se possível seguir o fluxo de informação genética desde os progenitores até aos seus descendentes.

Contudo, com o elevado desempenho das técnicas actuais de genotipagem, a ocorrência de erros em pedigrees torna-se cada vez mais um problema da maior importância e a consistência vai sendo cada vez mais algo com que os geneticistas têm de lidar.

O problema da verificação da consistência de pedigrees consiste na problemática de verificar se um pedigree (uma árvore geneológica com informação de genes associada) é ou não consistente com as leis de herditariedade de Mendel.

Esta verfificação de consistência é um problema NP-completo bastante conhecido e existem dois tipos principais de abordagens que o tentam resolver: abordagens estatísticas e abordagens combinatoriais.

O objectivo deste trabalho consiste na tentativa de desenvolvimento de dois tipos de abordagens combinatoriais baseadas em satisfação (suporte mínimo e codificação k-AC) e tentar medir a sua eficácia com distintos tipos de instâncias.

Palavras Chave: Pedigrees, NP-completo, Satisfação, Suporte Mínimo, Codificação k-AC.

Abstract

In bioinformatics, a pedigree is a structure that allows the monitoring of individuals and their family relations. With pedigrees, it is possible to track the flow of genetic information from parents to offspring.

However, with the high throughput of genotyping techniques, the occurrence of errors in pedigrees is becoming a problem of increasingly greater importance and consistency becomes something that geneticists must deal with.

The pedigree consistency checking problem consists in the problem of verifying if a pedigree (a genealogical tree with associated genotypes) is consistent under the Mendelian laws of inheritance.

This pedigree consistency check is a well-known NP-complete problem and there are two main types of approaches that attempt to tackle it: statistical and combinatorial ones.

The purpose of this work is to try to develop two types of combinatorial approaches based on satisfiability (minimal support encoding and k-AC encoding) and attempt to measure their efficiency against distinct types of pedigree instances.

Key Words: Pedigrees, NP-complete, Satisfiability, Minimal Support Encoding, k-AC Encoding.

Contents

List of Figures

List of Tables

1

Introduction

One of today's most challenging problems in the field of genetics consists in the correct relation between the genes in the human genome and some biological trait an individual may possess. Example of these traits range from blood type and eye color, to those that can predispose an individual for a particular disease.

The method used to tackle this problem is named linkage analysis [1, 2, 3] which is a well established statistical-based method that has already shown significant results: genes causing major diseases (e.g., Parkinson's disease, obesity and anxiety) have already been discovered using this technique.

In simpler terms, linkage analysis works by creating maps that correlate specific genes and/or genetic markers of a determined experimental population. And the more information contained on those maps, that is, the denser these maps become, the higher is the probability that the information may contain errors.

The problem of verifying the correctness of this information (if it contains errors or not) is named the pedigree consistency checking problem.

To describe linkage analysis, consistency checking, and the errors that can be generated in the data in more detail, it is first necessary to introduce some definitions regarding biology and inheritance.

1.1 Biological Background

DNA (deoxyribonucleic acid) is a molecule that contains the genetic instructions used in the development and functioning of all living organisms, with the role of long-term storage of genetic information.

A chromosome is a long chain of DNA and associated proteins that carry portions of the hereditary information in an organism.

Inside a chromosome, a *locus* is a fixed position that carries some specific information (which typically identifies the position of one or more genes). This specific information is named *allele*.

Diploid organisms (e.g. humans) carry chromosomes in pairs, which means that a single locus carries a pair of alleles. This pair of alleles defines the *genotype* of the individual at this locus. If both alleles are the same (that is, if the information described by the two alleles is equal) the individual is said to be *homozygous* at that locus, and if the alleles are different the individual is said to be *heterozygous*.

However, we should note that genotypes are not always completely observable and the indirect observation of a genotype (its expression) is named the *phenotype*.

1.2 Mendelian Laws of Inheritance

The notion of the inheritance of traits between generations was first introduced by Gregor Mendel (1865). And his main contribution, the Mendelian laws of inheritance, are the base principles that rule today's modern genetics [4]. The Mendel's laws, which specify the concept of *Mendelian inheritance* are presented next:

- **Unit factors in pairs**: *Genetic characters are controlled by unit factors that exist in pairs in individual organisms.* The unit factor, is today known as gene. This law implies that the genotype of an individual should always be considered as a pair.
- **Dominance/Recessiveness**: *When two unlike unit factors responsible for a single character are present in a single individual, one is dominant over the other, which is said to be recessive.* Dominance and recessiveness refers to the phenotype, which can be considered an abstraction of the genotype.
- **Segregation**: *During the formation of gametes the paired unit factors separate and segregate randomly so that each gamete receives one or the other with equal likelihood.* This

principle states that any combination of the maternal and paternal alleles have equal likelihood of occurrence in their descendants.

- **Independent Assortment**: *During gamete formation, segregating pairs of unit factors assort independently of each other.* This principle states that each gene is inherited independently of all other genes. This principle is no longer believed to be true, and in fact, one of the main objectives of techniques like linkage analysis is to determine whether there are parts of the genome that are inherited according to a pattern unlikely to occur at random.

1.3 Linkage Analysis and Pedigree Consistency Checking

Having now presented some important concepts in the previous sections, it is now possible to explain in much more detail the concept of linkage analysis and of consistency checking.

Genetic recombination [5, 6] is the process by which a strand of DNA is broken and then joined to a different DNA molecule. This occurs during meiosis, and usually, pairs of alleles that are on the same chromosome, tend to stay together during this process.

It is the task of linkage analysis to determine those genes that are inherited together and establish a correspondence to some trait in the individual (blood type, hair color, diseases).

But as genetic maps become denser, the effect of laboratory typing errors becomes more serious. Chromosome linkage maps are becoming denser as more marker data are collected (where a marker consists on a gene or DNA sequence with a known location on a chromosome that is associated with a particular trait or gene).

These high resolution linkage maps are important for locating disease genes. And building high resolution maps requires large amounts of genotypic data. The typing procedures are complicated and the volume of markers is large enough for errors to occur. As genetic maps become denser, errors can obscure the data by reducing the support for the correct genetic order making genes locations harder to define [7].

A computational problem that is closely related to linkage analysis is that of consistency checking.

A pedigree describes the family relations amongst a set of individuals, and usually comes associated information on their genotypes, that is, on the pair of alleles at a locus in their genome.

Given a pedigree and information on the genotypes (of some) of the individuals in it, the aim of consistency checking is to determine whether these data are consistent with the classic

Mendelian laws of inheritance.

1.4 Solving Pedigree Consistency Checking

There are a number of approaches that tackle the consistency checking problem, and they range from statistical methods to combinatorial ones. Of these two, combinatorial approaches will be the main focus of this work.

Regarding the combinatorial approaches, there are satisfiability (SAT) approaches [8], constraint satisfaction (CSP) approaches [9] and pseudo-boolean (PBO) approaches [10].

One of the objectives of this work, when concerning the combinatorial approaches, is to determine their efficiency against each other, when trying to tackle this problem, and verify which can be used in a particular situation.

For example, we will experiment some SAT encodings like the direct encoding and the support encoding [11, 12] and their performance will be measured using randomly generated pedigree instances as well as real pedigree instances.

1.5 Organization

This thesis starts, in chapter 2, by defining pedigrees, a fundamental structure used to deal with the pedigree consistency checking problem. This chapter also explains in more detail the types of errors occurring in pedigrees and the proof that the pedigree consistency checking problem is NP-complete.

In chapter 3 a list of existing tools used to detect and correct the errors in pedigrees are listed and described in some detail.

Next, in chapter 4 different SAT encodings that can be used to solve the pedigree consistency checking problem are described. And in chapter 5, the encodings described in the previous chapter are used, alongside some additional ones, to formulate the problem and devise new forms for solving the pedigree consistency checking problem. Comparative tests between these encodings are also described in this chapter.

Finally, a chapter with conclusions and details about future work concludes this thesis.

2

Pedigrees

A *pedigree* [4] is a structure used to track the inheritance of genetic traits. Usually, a pedigree has some (even if incomplete) genotype information associated with it.

In short, a pedigree consists in a representation of individuals and their family relations, where each individual has an associated genotype (this genotype consists of two alleles for the gene under consideration).

2.1 Pedigrees in detail

More technically, a pedigree can also be represented as a hypergraph, where the nodes represent the individuals of a family and the arcs represent the relationships between them.

Figure 2.1 represents a simple pedigree which consists of three individuals. Individuals 1 and 2 are the parents of individual 3. Squares represent male individuals and circles represent female ones, and therefore individual 1 is the father of 3 and individual 2 is the mother. All individuals in this pedigree have associated genotype `AB`.

The genotype of each individual in a pedigree is either known (through a genotyping process) or unknown, in which case it corresponds to a set of possible genotypes that can be inferred from the Mendelian laws of inheritance.

For example, also in figure 2.1, since each of the parents has genotype `AB` and the offspring also has the same genotype, there is no way of knowing which of the offspring alleles (either `A`

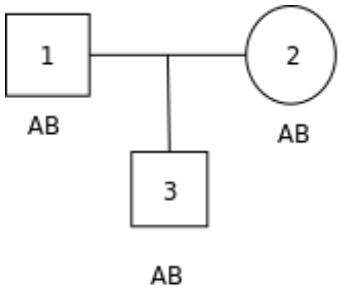

Figure 2.1: Example of a simple pedigree with associated genotype.

or `B`) came from which parent. Therefore writing `AB` or writing `BA` for individual 3 has the same meaning.

A formal definition of *Pedigree* and *Looping Pedigree* follows:

Definition 2.1.1 ***(Pedigree [4])*** *A pedigree consists of a 4-tuple* $\langle V,F,p,m\rangle$ *where:*

- V *is a finite, non-empty set of members of the pedigree,*
- $F \subseteq V$ *is the set of founders,*
- p,m: $V \setminus F \longrightarrow V$ *are the paternal and maternal functions, respectively, where*

  $$p(V\setminus F) \cap m(V\setminus F) = \varnothing$$

  (that is, no individual in the pedigree can be both mother and father), and
- *the transitive closure of the binary relation obtained by the union of p and m is irreflexive (which means that a member of the pedigree is never its own ancestor).*

The set $N = V \setminus F$ *is usually referred as the set of non-founders in a pedigree and is always non-empty.*

Definition 2.1.2 ***(Looping Pedigree [4])*** *Let* $P = \langle V,F,p,m\rangle$ *be a pedigree. Two distinct members u and v of the pedigree are said to mate if they have an offspring in common, that is, if there is a non-founder v' of P such that* $\{p(v'),m(v')\} = \{u,v\}$. *which means that v' is a child of u and v. A loop in P is a non-empty path consisting of distinct edges that start and end at the same node.*

Although a pedigree is acyclic, loops may still exist, which are classified in two types [13]: *inbreeding* and *marriage* loops. When two individuals who share a common ancestor mate and have an offspring, that creates an inbreeding loop; otherwise, it is a marriage loop. Marriage loops occur naturally in real pedigrees, whenever two siblings mate with the same individual. Inbreeding is usually more common in some species than in others. It is very uncommon, but definitely possible, in human pedigrees, while very common in many domestically bred animals.

The occurrence of loops is one of the main difficulties in pedigree analysis: the pedigree consistency checking problem is NP-complete for pedigrees [4] (considering only marriage loops), and some pedigree analysis problems such as the marginal-probability and the maximum-likelihood problem are NP-hard [13].

Figure 2.2 depicts an example of a looping pedigree. In this example, it is possible to witness two loops. One is due to inbreeding, and arises because individuals 4 and 5 mate, and have a common ancestor (individual 2). Another is a marriage loop which stems from individual 6 mating individuals 5 and 7, who are brothers. This figure has no genotypes associated with the individuals.

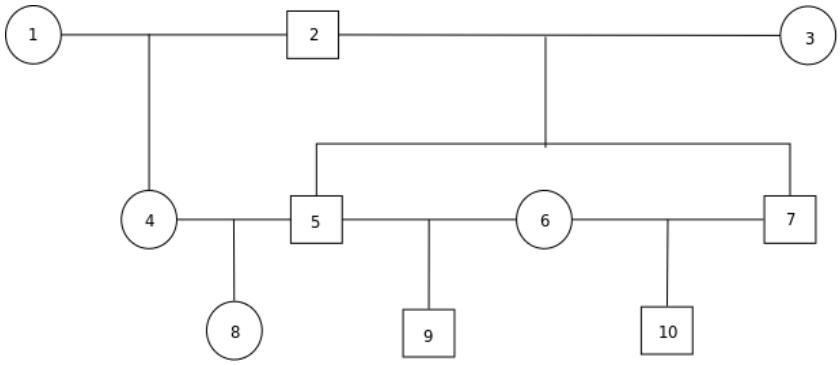

Figure 2.2: Example of a looping pedigree.

A definition for *Genotype Information* and for *Consistent Genotype Information* follows:

Definition 2.1.3 ***(Genotype Information [4])*** *Let $P = \langle V,F,p,m \rangle$ be a pedigree. A genotype information for P is a partial function $G:V \hookrightarrow Two(A)$ (each individual G is assigned two alleles A) that associates a genotype to (some of) the members of the pedigree. The domain, dom(G), of the function is referred to as the set of genotyped members of the pedigree. The genotype information G is complete if dom(G) = V (which means that every member V of the pedigree has an associated genotype).*

Let G and G' be two genotype information for the same pedigree. We say that G' extends G if dom(G) is included in dom(G') (i.e., dom(G)⊆dom(G')), and ∀ g ∈ dom(G): G(g) = G'(g) (which means that G and G' coincide over dom(G)).

Definition 2.1.4 ***(Consistent Genotype Information [4])*** *Let P = ⟨V,F,p,m⟩ be a pedigree.*

1. *A complete genotype information G for P is consistent with P if, for each v ∈ V:*
   (a) *if G(v) = {A,B}, then either A ∈ G(p(v)) and B ∈ G(m(v)), or B ∈ G(p(v)) and A ∈ G(m(v)).*
   (b) *if G(v) = {A,A}, then A ∈ G(p(v)) ∧ A ∈ G(m(v)).*
2. *An incomplete genotype information for P is consistent with P if it can be extended to a complete and consistent genotype information for P.*

In practice, this means that each individual in the pedigree must inherit exactly one allele from each of its parents.

Note that a genotype information needs to be complete in order to show that it is consistent.

Figure 2.3 shows a pedigree with inconsistent and incomplete genotype information. The genotype information is incomplete because individual 4 has no associated genotype. The genotype information for this pedigree is also inconsistent because individual 7 could not have inherited allele C from any of its parents.

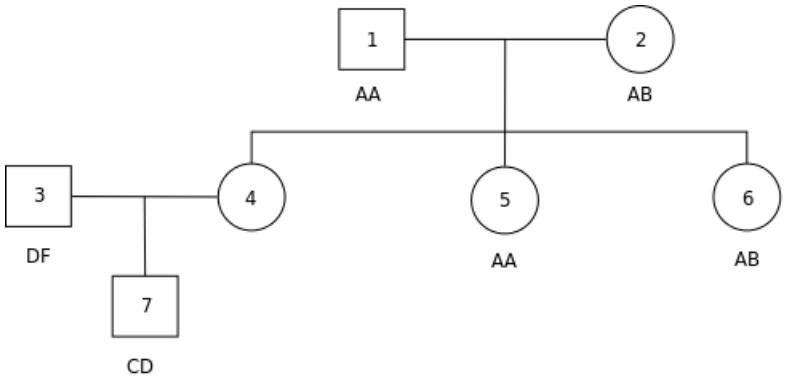

Figure 2.3: Example of a pedigree with associated genotype information.

2.2 Errors occurring in pedigrees

There is not much information available concerning pedigree errors, and the information that is in fact available is not consensual.

Most of the literature available ([7, 14, 15]) refers the existence of two types of errors: *pedigree errors* and *genotyping errors*.

Pedigree errors occur due to the misidentification of individuals and their relationships [7, 15]. Examples of these include non-paternity, unidentified adoption and sample mix-ups.

As for genotyping errors (or *typing errors* according to [7]), these occur mostly due to the misinterpretation of genotypes and data entry error. One can observe a genotyping error when the observed genotype does not correspond to the true underlying genetic information.

Pedigree errors will often be detectable since the incorrect relationships will show genotypes not conforming to Mendelian inheritance, that is, when a family shows an inordinate number of Mendelian inconsistencies, the reason may be often traced to an error in the pedigree information. When these errors do occur, they need only be resolved once [7].

But this is true only when the pedigree is complete [15]. When this is not the case, i.e. when there is no genotype information for some individual(s), a more sophisticated approach is necessary since pedigree errors may not show up through a check for Mendelian inheritance.

Genotyping errors, on the other hand, are not as simple to solve as pedigree errors. One first identifies genotypes that do not conform to Mendel's rules and then determines which individuals are responsible for it. In general, the most parsimonious explanation for the problem is sought for, finding the fewest genotypes which must be removed in order to eliminate the inconsistency [15]. It is important that the genotypes marked as inconsistent with Mendel's laws of inheritance are not simply removed, since in many cases a single error (e.g. in a parent) will lead to a number of genotypes being flagged, and so by identifying the single individual responsible for the problem, one may retain much more of the genotype information.

Figure 2.4 is an example of a complete and consistent pedigree. Nonetheless, should individual 1 be marked as having a wrong pedigree (due to data entry error, for example), the elimination of that genotype information would cascade to all of his descendants and 6 more individuals (the ones that depend from this individual's genotype which are individuals 4, 5, 6, 7, 9, 10) in the pedigree would be marked with possible inconsistent genotype information. Therefore the genotype of individual 1 should not simply be removed in order to retain most of the information regarding this pedigree.

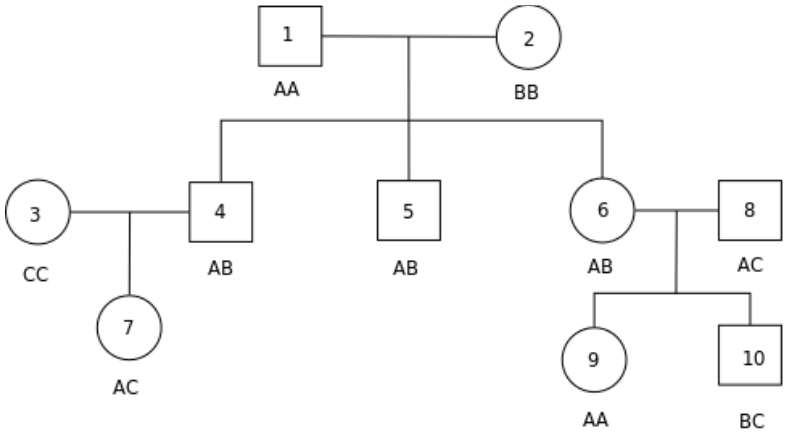

Figure 2.4: Example of a complete and consistent Pedigree

It is also important to remove all pedigree errors prior to the removal of genotyping errors. This is because a proper modification of parental relationships in the pedigree may eliminate the need to remove those genotypes [15].

It is convenient to state that, for the purposes of this work, only genotyping errors will be considered in all tested pedigrees. This means that it will be assumed that the structure of pedigrees is always correct, that is, there are no pedigree errors in the studied data in the following of this work. This approach has also been followed in [16].

There are also some references to other terms referring errors occurring in pedigrees [16]. These terms are *parental errors* and *Mendelian errors*. These types of errors are thought to be the same as pedigree and genotyping errors, respectively.

2.3 Complexity of pedigree consistency checking

Before describing the complexity of the pedigree consistency checking problem, it is first useful to present the concepts of P, NP and NP-completeness.

2.3.1 Classes of problems

The class P consists in the class of problems that are solvable in polynomial time, and the class NP corresponds to the set of problems that are verifiable in polynomial time. This means that, if there was a "certificate" of a solution, then that certificate could be verifiable in time polynomial in the size of the input of the problem.

Any problem belonging to class P also belongs to class NP, because if a problem is in P, then it can be solved in polynomial time without being given a certificate of a solution [8].

Informally, a NP-complete (or NPC for short) problem is in NP and it is as "hard" as any NP problem.

Formally, we say that a problem P is NPC when two criteria are verified for P:

1. P $\in$ NP, and
2. P' $\leqslant_p$ P for every P' $\in$ NP

The first condition states that a solution for P must be verifiable in polynomial time with respect to the size of P.

The second condition states that, for every NP-complete problem P', if it is possible to reduce P' in a polynomial factor to problem a P ($\leqslant_p$), then problem P also belongs to class NPC. A reduction consists in the process of translating a problem into another problem.

It is also important to refer that if a problem satisfies the second condition but not the first one, then that problem is said to be NP-hard.

2.3.2 Consistency checking is NP-complete

Having just described the concept of NPC, it is much simpler to prove that the pedigree consistency checking problem is in fact a NP-complete problem. It is first necessary to prove that it is a NP problem and then prove that it is NP-hard.

Proving this problem to be in NP is not too difficult. Given a pedigree P with genetic information G, it is only necessary to exhibit a certificate that is verifiable in polynomial time. Such a certificate would be a complete and consistent genotype information G^c that extends G as explained by definition 2.1. To make sure G^c is complete and consistent, the conditions in definition 2.1 have to be satisfied for every non-founder in P. This check takes a constant amount of time for each non-founding member of the pedigree, and therefore consistency checking takes a linear amount of time when regarding the total number of individuals in a pedigree [16].

Having proved that the pedigree consistency checking problem is in NP all that is left now is to prove that it is NP-hard. The proof of NP-hardness consists in a reduction from 3SAT as shown in [4] where 3SAT is a special case of the satisfiability problem where all of the clauses have exactly three literals. It is important to say that the pedigree consistency checking problem is only NP-complete if the genetic information is composed of three or more alleles and the

pedigree contains loops. Otherwise, when checking a non-looping pedigree, as shown in [17], can be performed in polynomial time.

3

Pedigree Consistency Checking Tools

The tools used for solving the consistency checking problem employ a variety of algorithms and techniques. A great part of the existing tools use statistic methods for this problem. Tools like Pedcheck [18] and APE [19] are among them. Moreover, there are tools that employ combinatorial approaches for consistency checking. Techniques like constraint satisfaction [10] and satisfiability [8] are used with this purpose.

A more detailed explanation of the tools follows.

3.1 Pedcheck

PedCheck [18] (available from http://watson.hgen.pitt.edu/register/soft_doc.html) is used in the detection of Mendelian inconsistencies. It processes only one locus at a time so as not to be limited by the number of markers it can handle.

PedCheck uses four distinct algorithms. These error-checking algorithms handle the different degrees of difficulty in the identification of an error. Each successive algorithm employs a more powerful approach to the identification of more subtle errors in the analised data.

3.1.1 Nuclear-Family Algorithm

With this algorithm only errors in nuclear families are detected, meaning only errors between parents and offsprings are analised. According to this algorithm an error is flagged if one or more

of the following conditions are true:

- The child is compatible with each parent separately but not when both parents are considered simultaneously;
- There are more than four alleles in a sibship (the set of children born to the same set of parents);
- There are more than three alleles in a sibship with a homozygous child;
- There are more than two alleles in a sibship with two different homzygotes among the siblings;
- An allele is out of bounds of any specified range;
- At an X-linked[1] locus, a male is not coded as "homozygous";
- An individual has only one allele defined (and not the expected two alleles);

The errors identified by this algorithm are named `level 1` errors.

Figure 3.1 depicts three of these level 1 errors (this figure was borrowed from [18]). First, there are more than four distinct alleles among the genotypes of siblings 3, 4 and 5; second, the genotypes of parent 3 and child 7 are incompatible; and third, under the assumption that there are only five alleles specified for this marker, the allele Z for individual 6 is out of range.

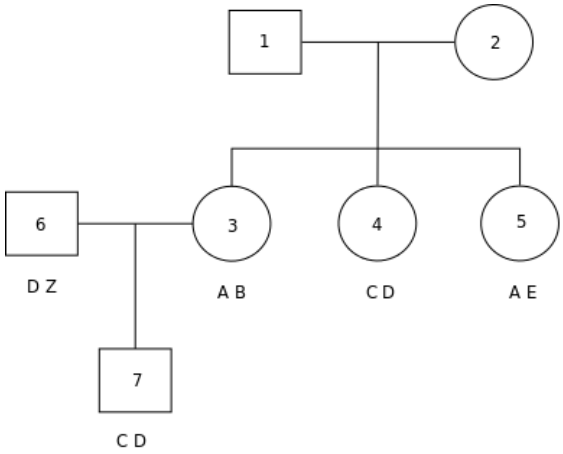

Figure 3.1: Pedigree with `level 1` errors.

[1] A form of inheritance where the gene in question lies on the X chromosome. X-linked genes can be dominant, but are usually recessive.

The nuclear-family algorithm is appropriate as a first check, but even with the correction of these errors, the pedigree may still need to be revised. In fact, this algorithm may be fast (because it involves only nuclear-family information and no genotype elimination), but it does not take into account that there are types of errors other than the ones occurring between members in a sibship.

3.1.2 Genotype-Elimination Algorithm

This algorithm is an extension of the Lange-Goradia algorithm [17], which recursively uses the nuclear-family relationships to eliminate invalid genotypes in the pedigree. The recursion is continued until no more genotypes can be eliminated.

The genotype-elimination algorithm is more powerful than the nuclear-family algorithm because it can discover subtle errors resulting from the elimination of certain genotypes, on the basis of more complex pedigree relations.

For each pedigree, the genotype-elimination algorithm identifies the first nuclear family that is found with an error (and has not already been identified by the nuclear-family algorithm) and outputs the inferred-genotype lists for each member of the nuclear family. Note that only one nuclear family is reported because a genotype-elimination error may propagate to other component nuclear families, and thus, ascertainment of whether the pedigree contains only one such error or multiple distinct ones is difficult.

An error identified by the genotype-elimination algorithm is said to be a `level 2` error.

Figure 3.2 shows an example of a level 2 error (this figure was borrowed from [18]). The nuclear-family algorithm will detect no errors but we can see that an error exists: individual 5 has genotype `BC` and the genotype of individual 3 is unknown; individual 5 could not have possibly inherited the `B` allele from individual 3, because individuals 1 and 2 have no such allele in their genotype.

A genotype-elimination algorithm is said to be complete if it can detect that the set of given genotypes violates the Mendelian laws of inheritance. Thus, if a complete genotype-elimination algorithm finds no errors, then the genotypes in the pedigree are consistent with Mendelian laws of inheritance.

Despite the Lange-Goradia algorithm being complete, it is only so for loopless pedigrees. The extended version of the Lange-Goradia algorithm is complete for all pedigrees. Note that this extended algorithm is not explained in [18], it is explained in detail in [20]).

Although the genotype-elimination algorithm will always find the more subtle errors, the

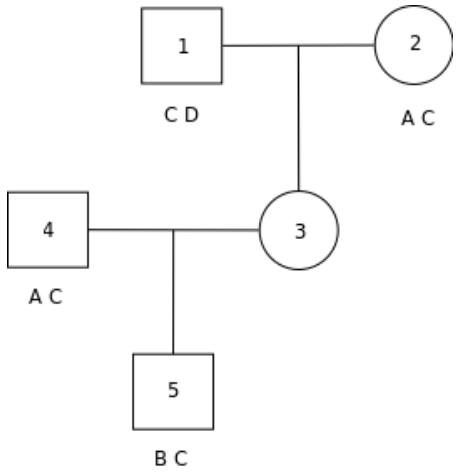

Figure 3.2: Pedigree with a `level 2` error.

identification of the source of these errors is not always so simple since the genotype lists for untyped individuals may be long. And even if there is only one error, the individual involved may be difficult to identify by the examination of the genotype lists only, since either more than one individual may be the source of that error or the error may not be in the particular nuclear family data appearing in the output.

3.1.3 Critical Genotype Algorithm

This algorithm focuses on *critical genotypes*. A critical genotype is one that, when removed from the data, that is, when marked as *unknown*, eliminate the pedigree inconsistency. Despite the removal of a critical genotype, this does not necessarily imply that the genotype is erroneous. For instance, untyping a correctly typed parent can produce a consistent pedigree when the error is caused by a child's genotype.

The concept of critical genotype can be extended to *degree n critical genotypes*, which is defined as the set of genotypes that, when removed simultaneously from the data, eliminate the inconsistency.

The critical genotype algorithm works as follows: iteratively one individual is "untyped" at a time by marking him/her as having an unknown genotype and applying the genotype-elimination algorithm to determine if the inconsistency has been eliminated.

For example, in figure 3.3 there are two critical genotypes: the genotype `BB` of individual 2 and the genotype `AA` of individual 4. Untyping either of these two genotypes will remove the

inconsistency. This picture was also borrowed from [18].

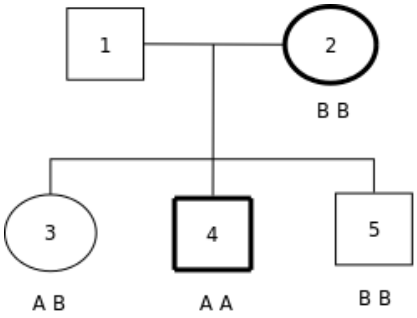

Figure 3.3: Pedigree with two critical genotypes. The frames for those two individuals are marked in bold.

3.1.4 Odds-Ratio Algorithm

When several critical genotypes are identified at a locus, there is no way of deciding *a priori* which critical genotype is most likely to be erroneous. The odds-ratio algorithm is based on single-locus likelihoods of the pedigree and helps distinguishing between alternative critical genotypes.

By definition, untyping an individual with a critical genotype results in a consistent pedigree. So, after the genotype elimination, the individual must have at least one alternative valid genotype.

The odds-ratio algorithm works by computing and storing the likelihood of the pedigree data for each alternative valid typing at each critical genotype, holding all other critical genotypes at their original value. Specifically, for each alternative genotype the likelihood L is computed. Let $L_{\max}$ be the largest likelihood obtained. Then, for each alternative genotype, the quantity $L_{\max}/L$ is computed, which gives an odds ratio against that alternative genotype versus any genotype with likelihood $L_{\max}$. Any genotype with a value of $L_{\max}$ will have an odds ratio of 1, and, in general, the best supported genotypes will have an odds ratio close to or equal to 1. The algorithm returns each alternative typing together with its odd ratio.

The odds ratio can also be defined in terms of conditional probability. Let P1 and P2 represent two individuals with critical genotypes, D the original pedigree data (with P1 and P2 set to unknown), G1 and G2 the original genotypes of P1 and P2, respectively, and A1 and A2 any two alternative genotypes for P1 and P2 respectively. Then the odds ratio is $P(D|P1 = G1, P2 = A2)/P(D|P1 = A1, P2 = G2)$.

Since computation of the likelihood requires allele frequencies to be specified, the odds ratio algorithm has three variations: (1) use the user-defined allele frequencies; (2) assume that all alleles are equally frequent; and (3) use allele frequencies estimated from all typed individuals in the input file, by counting the number of times the allele appears at a locus, divided by the total number of given alleles.

3.2 Allelic Path Explorer (APE)

Allelic Path Explorer (APE)[2] uses an heuristic algorithm for finding gene transmission patterns on large and complex pedigrees with partially observed genotype data [19]. The method can be used to generate an initial point for a Markov Chain Monte Carlo [21] simulation or to check that the given pedigree and the genotype data are consistent.

The algorithm assumes the existence of a pedigree composed of a number of individuals ($\mathcal{I}$) and mating nodes (which are nodes that explicitly indicate that two given individuals mate), and some (partial) genotype information of the individuals. $\mathcal{A}$ is denoted as the set of labels for the different alleles. The symbol $\emptyset$ is also included in $\mathcal{A}$ to denote the alleles whose labels are still unknown. The set of unordered partial genotypes is given by $\mathcal{G}$ such that $\mathcal{G} = \{\{a, b\} : a, b \in \mathcal{A}\}$. A genotype configuration on the pedigree is a mapping $\tau(\mathcal{I}) \rightarrow \mathcal{G}$.

A *descent path* of an allele is one that tracks the origins of that particular allele up to the founders of a given pedigree. Note that, for each non-founder, there are several possible descent paths since, at each step, a path can follow either the paternal or the maternal line of descent.τ_{data} denotes the genotype configuration that corresponds to the observed data.

The algorithm tries to discover a set of consistent descent paths for the defined alleles in the data. Consistency requires that, for each individual, one allele is inherited from the father and one from the mother and that every individual is a member of at least two descent paths of different kinds of alleles.

The step size of the algorithm is that of a `level`. A level coincides with the generations in the pedigree where generations do not overlap.

The *rank* ρ of an individual i is defined recursively as:

[2] Available at http://www.helsinki.fi/~mpirinen/download.

$$\rho(i) = \begin{cases} 0, & \text{if i is a founder,} \\ \max\{\rho(f_{\mathrm{i}}), \rho(m_{\mathrm{i}})\} + 1, & \text{otherwise.} \end{cases} \tag{3.1}$$

The rank is the largest number of mating nodes encountered on a path from the individual to any of his ancestors in the pedigree. Let $L = \max\{\rho(i)\colon\ i \in \mathcal{I}\}$ (where $\mathcal{I}$ represents the set of individuals in the pedigree). For each $i \in \mathcal{I}$, define the `level` as $l(i)$ =L$-\rho(i)$. Thus, for the individuals at level l, a maximal ancestral path is of length L - $l(i)$.

The search for descent paths proceeds level wise and each level in turn is divided into independent components. Consider the individuals at level l together with their parents, that is, the sets

$$C_{\mathrm{l}} = \{i \in \mathcal{I} : l(i) = \mathrm{l}\} \tag{3.2}$$

and

$$P_{\mathrm{l}} = \{i \in \mathcal{I} : i = f_{\mathrm{j}} \text{ or } i = m_{\mathrm{j}} \text{ for some j } \in\ C_{\mathrm{l}}\} \tag{3.3}$$

The search for a consistent set of descent paths proceeds level by level from 0 up to L. If, at some level $l \leq L$, the current temporary configuration cannot be extended to the set P_l, it has to be changed at some lower level(s).

A configuration for the set P_l corresponds to a mapping $\sigma_l \rightarrow \mathcal{G}$, whereas a temporary configuration $\bar{\sigma} = (\sigma_0, \ldots, \sigma_{l-1})$ at level l is a genotype configuration which is compatible both with the observed data and the configurations $\sigma_k : P_k \rightarrow \mathcal{G}$ for each $k \leq l$ and which leaves the remaining genotypes on the pedigree undefined.

More precisely, the following sets are recursively defined

$$S^l_{(\sigma_0,\sigma_1,\ldots,\sigma_{l-1})} \subseteq \{\sigma : P_l \rightarrow \mathcal{G}\}, \tag{3.4}$$

where $l \leq L$ and each $\sigma_k \in S^k_{(\sigma_0,\sigma_1,\ldots,\sigma_{k-1})}$, for $k \leq l$.

For instance, let S^0 contain those configurations for the set P_0 that have all the observed alleles of the individuals in C_0 transmitted to their parents in P^0 and that are consistent with

the laws of inheritance and the observed data.

The goal of the algorithm is to construct a temporary configuration $(\sigma_0, \ldots, \sigma_{l-1})$, where $\sigma_k \in S^k_{(\sigma_0,\ldots,\sigma_{k-1})}$, for each $k \leq L$. Such a configuration transmits all the defined alleles up to the founders in a consistent way and thus contains enough information for finding a consistent inheritance pattern in the pedigree.

When, at some level $l \leq L$ with the temporary configuration $(\sigma_0, \ldots, \sigma_{l-1})$ and the stored sets $S^k_{(\sigma_0,\ldots,\sigma_{k-1})}$ with $k \leq l$, the next task is to find the set $S^l_{(\sigma_0,\ldots,\sigma_{l-1})}$ and decide how to proceed, depending on whether this set is empty.

If $S^l_{(\sigma_0,\ldots,\sigma_{l-1})}$ is empty, then the temporary configuration $(\sigma_0, \ldots, \sigma_{l-1})$ cannot be extended to the set P^l and must be modified. The largest $k \leq l$ is chosen for which there exists an unchecked configuration in the set $S^k_{(\sigma_0,\ldots,\sigma_{k-1})}$, the first unchecked configuration σ'_k is picked, and the algorithm continues with the temporary configuration $(\sigma_0, \ldots, \sigma_{k-1}, \sigma'_k)$ at level $k+1$. If no such k exists, then the data are inconsistent with the pedigree.

If, on the other hand, $S^l_{(\sigma_0,\ldots,\sigma_{l-1})}$ is not empty, the set is ordered and the first configuration is chosen from it to extend the temporary configuration to level $l+1$. The configurations among the set are ordered according to two methods: the pseudo-conditional genotype probabilities and the amount of genetic variation between relatives. Both of these methods are probabilistic methods.

3.3 Checkfam

In Checkfam[3] [14], two algorithms are implemented that identify errors in genotypic data: a nuclear-family algorithm modified genotype-elimination algorithm.

In the nuclear-family algorithm, the process of finding Mendelian inconsistencies is as follows: (1) Different alleles $(a_{\mathrm{i}}, i = 1, 2, ...)$ are listed from the parents and the children, and a check is done to ensure that the number of alleles is not more than four. (2) If parental genotypes are $(a_{\mathrm{i}}, a_{\mathrm{j}})$ and $(a_{\mathrm{k}}, a_{\mathrm{l}})$, then the genotypes of their children must be $(a_{\mathrm{i}}, a_{\mathrm{k}})$,$(a_{\mathrm{k}}, a_{\mathrm{i}})$, $(a_{\mathrm{i}}, a_{\mathrm{l}})$, $(a_{\mathrm{l}}, a_{\mathrm{i}})$, $(a_{\mathrm{j}}, a_{\mathrm{k}})$, $(a_{\mathrm{k}}, a_{\mathrm{j}})$, $(a_{\mathrm{j}}, a_{\mathrm{l}})$ or $(a_{\mathrm{l}}, a_{\mathrm{j}})$. An error is reported if the data are not in accord with this rule.

The modified genotype-elimination algorithm is based on the extended version of the Lange-Goradia algorithm [17, 20]. The algorithm used in Checkfam is also able to check data from all pedigree structures, including loops.

[3]Available from http://www.genstat.net/checkfam/index.cgi?lang=en.

Before performing the modified genotype-elimination algorithm, each nuclear family in a pedigree is given an integer number named the *ancestor number*. This ancestor number is defined as the number of ancestors described in the pedigree of a child in the nuclear family. Note that the number of ancestors is always the same for all the children of a nuclear family. The nuclear family with a smaller ancestor number is examined later than one with a larger ancestor number. If the examination is performed in this order, all children in a nuclear family should have already been examined for Mendelian inheritance if necessary when that nuclear family is checked.

The algorithm proceeds as follows:

1. The ancestor number is counted for each nuclear family.

2. Mendelian inheritance is checked from one family to another so that the order of the ancestor number is from the largest to the smallest.

3. When the Mendelian inheritance in a nuclear family is checked, all possible genotypes of the parents are listed and saved for examination of the other nuclear families that include them.

4. If there is at least one set of genotypes of all family members that is in accordance with Mendelian inheritance, then the program reports "no errors detected"; otherwise, it reports the errors that have been found.

Figure 3.4 (A) depicts an example of an input file of Checkfam. Figure 3.4 (B) shows the output. The first six columns of the input file represent, respectively: the id of the nuclear family, the id of the individual, the id of the father, the id of the mother, the sex of the individual (1 for male and 2 for female) and the affected status of the disease (1 for unaffected, 2 for affected and 0 for unknown). The following columns correspond to the alleles for each marker.

In figure 3.4 B()(a) two continuous columns corresponding to a marker appear colored for 4 individuals, for which there are technical problems in the genotyping of that marker.

In figure 3.4(B)(b) the entire marker genotype data for the second nuclear-family appears colored, revealing the existence of family misrecordings in that nuclear-family.

In figure 3.4 (B)(c) the second to fifth columns corresponding to the first nuclear family appear colored, highlighting the existence of incompatibilities in sex and family relationships.

From the analysis of figure 3.4 (B) it is possible to conclude that there is a technical problem in the genotyping of the fifth marker. Also it is likely that there is a misrecording in the family

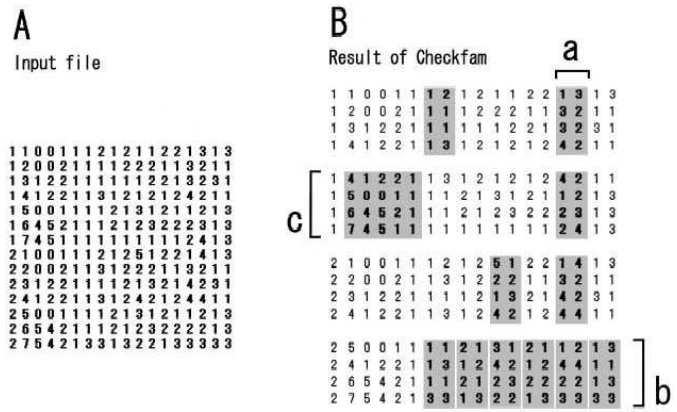

Figure 3.4: Example of input and output data of Checkfam. Example borrowed from [14].

relationships involving individuals 5 to 7 in family 2. The figure also shows a mistake in either the sex or the parent columns in the nuclear family involving individuals 4 to 7 in family 1.

3.4 PCS

PCS[4] [22] is a tool based on Boolean Satisfiability (SAT) [8, 23, 24, 25, 26].

A Boolean formula traditionally represented in *Conjunctive Normal Formula* (CNF), is a conjunction (AND) of disjunctions (OR) of literals. A literal is either x or $\neg x$, for a Boolean variable x. The disjunctions of literals are called *clauses*. The SAT problem consists in finding an assignment to the variables, such that the CNF formula evaluates to `true` (such a formula is said to be *satisfied*), or to prove that no such assignment exists.

PCS first preprocesses the pedigree data in order to search for simple errors. For example, it checks whether every member's father is a male and every member's mother is a female.

The main phase of PCS is the translation of the consistency problem into a SAT problem. The BAT tool [27] is used in order to make this translation. After this translation, the generated CNF file can be given to any standard SAT solver. If a satisfying assignment is found, then the problem is consistent. If the formula is unsatisfiable, then there are incompatibilities in the pedigree. In this case, an *unsatisfiable core* is extracted.

An *unsatisfiable core* is an unsatisfiable subset of the clauses. f this set is minimal, then it becomes satisfiable if any of its clauses are removed. By extracting a minimal *unsatisfiable core*, the set of members that have an inconsistent genomic information can be determined. Instead

[4]This tool is not publicly available.

of reporting one error at a time, PCS iteratively generates unsatisfiable cores and removes the genotype information of the individuals involved until a fixed point is reached. When the fixed point is reached, PCS generates a report outlining all of the inconsistencies found.

3.5 Mendelsoft

Mendelsoft[5], as described in [16], is directly derived from the tool `Toulbar2`, and is a tool for the detection of Mendelian inconsistencies in complex pedigrees. Mendelsoft tackles this problem using weighted constraint satisfaction networks.

A constraint satisfaction problem [16, 9, 28, 24, 11] (X, D, C) is defined by a set of variables $X = \{x_1, \ldots, x_n\}$, a set of matching domains $D = \{d_1, \ldots, d_n\}$ and a set of constraints C. Every variable $x_i \in X$ takes its value from the associated domain d_i. A constraint $c_S \in C$ is defined as a relation over a set of variables $S \subset X$ which defines authorized combinations of values to be assigned to variables in S. The central question in constraint problems is to give a value to each variable in such a way that no constraint is violated (only authorized combinations are used). Such a variable assignment is a solution of the problem.

Often, tuples are not just completely allowed or forbidden but rather authorized or forbidden at some extent or cost. In weighted constraint networks (WCN), constraints map tuples to non negative integers. For every constraint $c_S \in C$, and every variable assignment A, $c_S(A[S]) \in N$ represents the cost of the constraint for the given assignment, where $A[S]$ is the projection of A on the constraint scope S. The objective is then to find an assignment A of all variables such that the sum of all tuple costs $\sum_{c_S \in C} c_S(A[S])$ is minimum. This is called the Weighted Constraint Satisfaction Problem (WCSP) which is NP-hard [16].

Consider a pedigree defined by a set of individuals I. For a given individual $i \in I$, $pa(i)$ is noted as the set of parents of i. At the considered locus, the set of possible alleles is $\{1, \ldots, m\}$. The set of all possible genotypes is denoted by G and has cardinality $\frac{m(m+1)}{2}$. For a given genotype $g \in G$, the two corresponding alleles are denoted by g^l and g^r and the genotype is also denoted as $g^l|g^r$. The experimental data is made of phenotypes, and for each individual in the set of observed individuals $I' \subset I$, its observed phenotype restricts the set of possible genotypes to those which are compatible with the observed genotype. This restricted set of genotypes is denoted by $G(i)$.

The corresponding constraint network encoding this information uses one variable per indi-

[5] Available at www.inra.fr/mia/T/MendelSoft/

vidual ($X = I$). The domain of every variable $i \in X$ is defined as the set of all possible genotypes G. If an individual i has an observed phenotype, an unary constraint that involves the variable i and authorizes the genotypes in $G(i)$ is added to the network. Finally, to encode the Mendelian laws for every individual i that is a non-founder $i \in X$ is added a single ternary constraint involving i and the two sets of parents of i, $pa(i) = \{j, k\}$ This constraint only authorizes one allele in i from each parent, that is

$$(g_i^l \in g_j \wedge g_i^r \in g_k) \vee (g_i^l \in g_k \wedge g_i^r \in g_j) \tag{3.5}$$

To model the possibility of genotyping errors, genotypes in G which are incompatible with the observed phenotype $G(i)$ should not be completely forbidden. Instead, a soft constraint (or cost function) forbids them with a cost of 1, thereby obtaining a weighted constraint network with the same variables as before, the same hard ternary constraints for Mendelian laws and soft unary constraints for modeling genotyping information.

If an assignment of all variables is considered, as indicating the real genotype of all individuals, the sum of all costs determined by all unary constraints on this assignment precisely gives the smallest number of errors made during typing. Finding an assignment with a minimum number of errors follows the parsimony principle and is consistency with a low probability of independent errors.

Figure 3.5 illustrates a CSP representing a CSP. There are 12 individuals and 3 distinct alleles. The possible genotypes are $G = \{1|1, 1|2, 1|3, 2|2, 2|3, 3|3\}$. There are 7 individuals with an observed phenotype. The corresponding CSP has 12 variables, with maximum domain size of 6, and 8 ternary constraints representing the parental relationships.

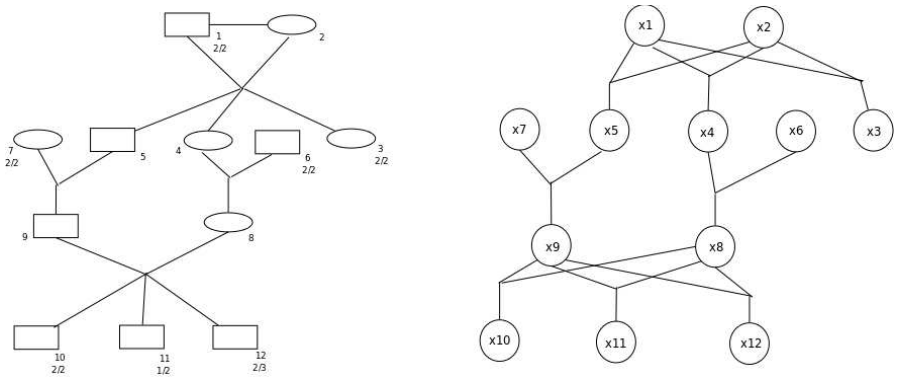

Figure 3.5: A pedigree and its corresponding CSP (on the right). This example was borrowed from [16].

Regarding error correction, the simple parsimony criterion is usually not sufficient to distinguish among alternative values. Since errors and Mendelian inheritance are typical stochastic processes, Mendelsoft applies a probabilistic model. Specifically, a Bayesian network [29] is used, which is a network of variables related by conditional probability tables (CPT) forming a directed acyclic graph (DAG). To model errors, a usual approach is to distinguish the observed phenotype O from the truth genotype T. A CPT $P(O|T)$ relates the two variables and models the probability of errors.

Following these definitions, the error correction model is presented.

Conditional table $P_i^{error}(O_i|T_i)$ represents the probability of the observed phenotype of individual i, O_i, when facing its true (unkown) genotype T_i. It is assumed a constant probability of error α. The probability of observing the true genotype is $1-\alpha$ and the remaining probability mass is distributed equally among remaining values.

Conditional probability table $P_i^{mendel}(T_i|pa(i))$, representing Mendelian inheritance, connects T_i and its corresponding parent variables. Given that each parent has two alleles, there are four possible combination for the children and all combinations are equiprobable (probability $\frac{1}{4}$). But since parental alleles are not allways different, the genotype of a child has probability $\frac{1}{4}$ times the number of combinations of parental alleles that produce this genotype.

Finally, prior probabilities $P^{founder}(i)$ for each genotype must be given for every founder i. These probabilities are obtained by estimating the frequency of every allele in the genotyped population. For a genotype, its probability is obtained by multiplying each allele frequency by the number of its possible combinations.

The probability of a complete assignment $P(O|T)$ (all the true and observed values) is then defined as the product of the three sets of probabilities ($P^{error}, P^{mendel} and P^{founder}$).

3.6 Performance Comparison between Mendelsoft and Pedcheck

As previously mentioned, Pedcheck and Mendelsoft use two different approaches for Pedigree Consistency Checking. Pedcheck uses a mainly statistical approach and Mendelsoft uses a combinatorial one [28].

For testing these tools, both random and real instances were used. The ultimate goal on using these instances was to compare their running times on both Pedcheck and Mendelsoft, and

verify which of the approaches was the most efficient.

3.6.1 Synthetic Pedigree Instances

The synthetic instances were generated using a pedigree generator designed by geneticists at INRA [30]. The instances generator is controlled by three main parameters: number of founder individuals, number of male founder individuals, and number of simulated generations. There is a total of four different collections (A, B, C and D), each of them containing several different subsets of instances divided by an increasing number of individuals (except for collection D where instances are divided by the number of founding males). Each subset contains about 40 to 60 different pedigree instances.

Table 3.1 describes the four different classes of randomly generated pedigrees in more detail.

Table 3.1: Four different random classes of pedigree instances

	Founders	Generations	Males	Individuals	Number of subsets
Collection A	4...44	3	2	[10,170]	21
Collection B	8...68	5	4	[28,388]	16
Collection C	20...200	5	20	[140,1100]	9
Collection D	200	6	4...100	[1000,1376]	9

Figures 3.6, 3.7, 3.8 and 3.9 compares the CPU times required by each of the collections running in both Mendelsoft and Pedcheck. These figures are `pairplots` and they describe, for the same instance, the running times on both tools (Mendelsoft vertically and Pedcheck horizontally).

Each dot in a figure represents the running time of a collections' subset for both encodings. And the running time of a subset corresponds to the average of running times of the instances in it. For the same dot, two running times can be observed, one for each encoding on each of the `pairplots` axis.

By analyzing figure 3.6 we can see that the running times are practically the same on the first three or four subsets of instances. But this difference keeps increasing as instances also increase (in terms of number of individuals). The running times in Pedcheck increase rapidly as the running times in Mendelsoft increase in a more subtle manner.

Considering figure 3.7 and the fact that these instances in collection B are somewhat larger in the number of individuals when comparing them to the instances in class A, the difference of

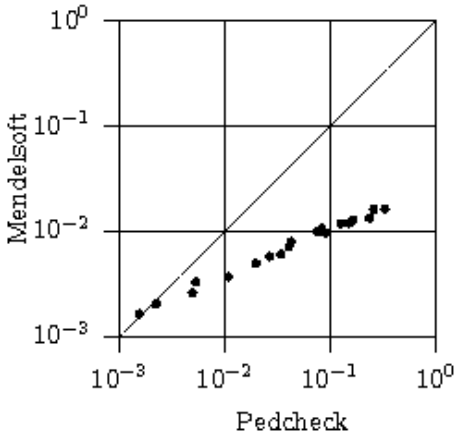

Figure 3.6: CPU times for Mendelsoft vs. Pedcheck in collection A of instances.

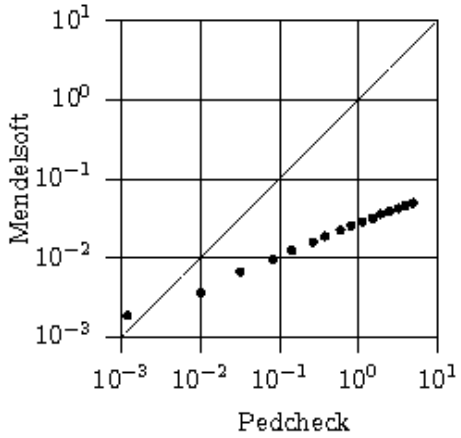

Figure 3.7: CPU times for Mendelsoft vs. Pedcheck in collection B of instances.

running times between Pedcheck and Mendelsoft is much more significant. When comparing the obtained results, we may also conclude that as instances grow in size, so does the time increases when using Pedcheck. CPU times also increase when running the instances on Mendelsoft, but not on such a larger scale as Pedcheck. By comparing figures 3.6 and 3.7 we can also observe a similar behavior with the difference that running times in figure 3.7 are much higher (due to a greater complexity of these instances).

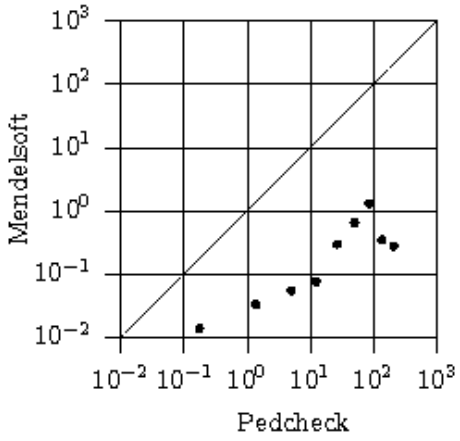

Figure 3.8: CPU times for Mendelsoft vs. Pedcheck in collection C of instances.

The instances described in figure 3.8 are considerably larger than the previous ones. The number of individuals in this class of instances is far greater and the complexity of the pedigrees is also much higher. Nonetheless, Mendelsoft can solve all of these instances in less than a second. Pedcheck, on the other hand, takes orders of magnitude more to solve the same instances. With more than 500 individuals in the pedigree, the running times in Pedcheck start increasing rapidly.

The D class proved to be the most difficult to solve in terms of CPU time. More than half of the instances in Pedcheck took over 300 seconds to solve while a substantial portion of them also took over 30 seconds to solve (some took over 100 seconds). It is interesting to observe that the main parameter, that is, the main factor responsible for the variation of the instances in this class, is not the number of individuals in the pedigree, but the number of founding males in the pedigree.

Curiously, Pedcheck takes the most time in solving instances when the number of founding males is quite low and the number of individuals is quite high.

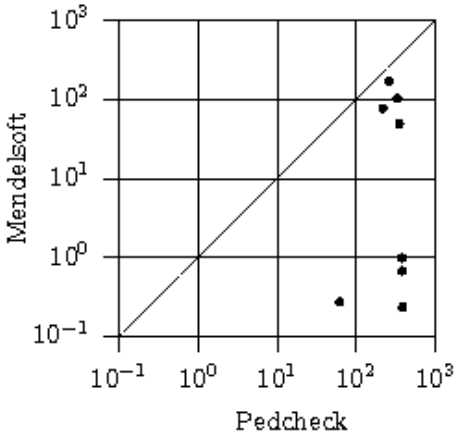

Figure 3.9: CPU times for Mendelsoft vs. Pedcheck in collection D of instances.

3.6.2 Real Pedigree Instances

A total of 11 real pedigree instances were run with Pedigree and Mendelsoft. The first three instances are human genotyped pedigrees (genetic studies of eye, cancer, and Parkinson diseases) as reported in [18, 20]. The following group (*berrichon*) consists in a group of real pedigree instances coming from sheep animals as reported in [20]. Finally, the group *langlade* are pedigree instances coming from sheep animals provided by the CTIG (*Centre de Traitement de l'Information Génétique*) in France. The files correspond to all the genotypings done for the enhancement program of genetic resistance to *scrapie* disease [31].

Table 3.2 describes the parameters of the real pedigree instances as well as the CPU time it took to run them in both Mendelsoft and Pedcheck. They are: the name of instance, the number of individuals, the number of genotyped individuals, number of alleles, number of founders, number of generations, running time in Mendelsoft and running time in Pedcheck.

From table 3.2 it is possible to compare the behavior of Pedcheck and Mendelsoft on real pedigree instances. Most of these instances are considered to be very difficult either by having many individuals or by having a large number of alleles (for example, instance *cancer* has a total of 8 alleles).

Observing the results we can conclude that instances with the higher number of individuals

Table 3.2: CPU times for Mendelsoft vs. Pedcheck in real pedigree instances.

name	*inds*	*genotyped*	*alleles*	*founders*	*ngen*	*Pedcheck Time(secs)*	*Mendelsoft Time(secs)*
eye	36	28	6	11	4	0.013	7.492
cancer	49	37	8	18	5	0.103	9.383
parkinson	37	13	4	7	7	0.002	0.001
berrichon1nc	129516	2448	4	8821	17	1208.53	28.353
berrichon1	129516	2483	4	8786	17	1215.73	26.225
berrichon2nc	27255	10215	4	4719	19	13.291	0.032
berrichon2	27255	10215	4	2381	19	11.159	0.005
langlade1	1355	711	9	298	13	-	15.868
langlade2	1355	715	7	298	13	-	338.155
langlade3	1355	787	5	298	13	-	35.968
langlade4	1355	672	8	298	13	-	260.142

are the ones that end up taking the greater amount of time to be run by both tools. But as Mendelsoft solves them in about 30 seconds, Pedcheck takes over 1000 seconds. Another interesting observation regards the number of alleles in Mendelsoft: instances with a relatively high number of alleles take a considerable amount of time to be solved.

As an overall comment on the test results, the main conclusion that can be withdrawn is that the effectiveness of Mendelsoft (when regarding running times) is orders of magnitude faster than the Pedcheck tool for the same set of instances. This can occur due to a number of reasons. First, because the used instances were taken from a repository maintained by the creators of Mendelsoft, it may be possible that those instances are somewhat better prepared to run under Mendelsoft. The other reason may be due to the underlined principles and concepts used to employ both tools. Mendelsoft uses the parsimony principle, meaning it will try to find the minimum number errors that, when removed, can fully explain the data. On the other hand, Pedcheck uses statistical methods to find all errors that a pedigree may contain. The conclusion is that, when instances are complex enough in the number of individuals and the number of errors, the work that needs to be done by Pedcheck far exceeds the work done by Mendelsoft. Hence the large gap in running times.

4

Maximum Satisfiability and Related Encodings

In this chapter, the SAT encodings used to tackle the pedigree consistency checking problem are presented. Most of the described encodings in this section are of the utmost importance for the basis of this work. These encodings actually form the foundations of the developed work regarding pedigree consistency checking.

4.1 SAT, Max-SAT and Partial Max-SAT

In propositional logic, a variable x_i can take only one of two values: *true* (value 1) or *false* (value 0). A literal l_i is either a variable x_i or its negation $\neg x_i$. A clause is a disjunction of literals, and a CNF formula is a multiset of clauses. An assignment of truth values to the variables satisfies the corresponding positive literal if the variable is assigned value 1 and satisfies the corresponding negative literal if it is assigned value 0. An assignment satisfies a clause if it satisfies at least one literal of the clause, and satisfies a CNF formula if it satisfies all the clauses in that formula. An empty clause contains no literals and cannot be satisfied.

The Max-SAT problem is the problem of finding an assignment to the variables in a CNF formula that maximizes the number of satisfied clauses. In Partial Max-SAT, some clauses are called *non-relaxable* or *hard* clauses and the others are called *relaxable* or *soft* clauses. Solving a Partial Max-SAT instance consists of finding an assignment that satisfies all of the *hard* clauses and the maximum number of *soft* clauses.

There are two approaches to solve the Max-SAT problem [32]: heuristic/approximation algorithms, and exact algorithms. The first ones find near-optimal solutions and the latter ones compute exact solutions.

4.1.1 Inference Rules

Inference rules [33, 34, 32], mainly used in exact algorithms, are a way of efficiently pruning the search space when solving a given SAT or Max-SAT instance. These inference rules (also called transformation rules) allow to transform an instance ϕ into another equivalent instance ϕ', but much simpler. In the best case, inference rules will produce empty clauses which do not have to be recomputed at every cycle of a given SAT or Max-SAT solver [32].

A Max-SAT inference rule is said to be *sound* if it transform an instance ϕ into another equivalent instance ϕ'.

The most efficient and simpler inference rule that most SAT solvers first try to enforce is *unit propagation* [35, 36]. This rule is based on unit clauses (which are composed of a single literal). If a set of clauses contains a unit clause l, the other clauses are simplified by the application of the following rules:

1. Every clause that contains l is removed.
2. In every clause that contains $\bar{l}$, this literal is deleted.

It becomes simple to observe that this is indeed a very efficient rule in the sense that, as negated literals are removed from clauses, they may originate other unit clauses and propagation can start all over again.

Unfortunately, unit propagation is unsound for Max-SAT because the simplification of a formula may not find its optimal value, and because when the goal is to maximize the number of satisfied clauses, the simple proof of unsatisfiability is simply not sufficient and even pointless.

Some other useful inference rules are:

- The pure literal rule: If a variable only appears as either a positive or a negative literal in a Max-SAT instance, all of the clauses containing that literal are removed;
- The resolution rule: If, in a SAT instance, there exist the clauses $x \vee D$ and $\overline{x} \vee D'$, where D and D' are disjunctions of literals and $D \neq D'$, then both clauses are replaced by $D \vee D'$;

- The almost common clause rule: If a Max-SAT instance contains a clause $x \vee D$ and a clause $\overline{x} \vee D$, where D is a disjunction of literals, then both clauses can be replaced by the clause D. In practice, for efficiency reasons, this clause is applied when D contains at most one literal [32];

The resolution rule, despite preserving satisfiability, does not preserve equivalence. Therefore, this rule cannot be applied to Max-SAT, except in particular cases like the almost common clause rule.

It is important to state that, despite rules such as the unit propagation rule and the resolution rule cannot be applied to Max-SAT, they can be applied, however, to the hard part of Partial Max-SAT instances, since all hard clauses must be satisfied.

4.2 Max-CSP

From section 3.5 we have that a constraint satisfaction problem [16, 9, 28, 24, 11] (X, D, C) is defined by a set of variables $X = \{x_1, \ldots, x_{\mathrm{n}}\}$, a set of matching domains $D = \{d_1, \ldots, d_{\mathrm{n}}\}$ and a set of constraints C. Every variable $x_{\mathrm{i}} \in X$ takes its value in the associated domain d_{i}. A constraint $c_{\mathrm{S}} \in C$ is defined as a relation on a set of variables $S \subset X$ which defines authorized combination of values for variables in S. The main goal is to find a variable assignment such that no constraint is violated (only allowed combinations are used). Such a variable assignment is called a solution of the problem.

Often, tuples are not just completely allowed or forbidden but rather allowed or forbidden at some extent or cost. In weighted constraint networks (WCN), constraints map tuples to non negative integers. For every constraint $c_{\mathrm{S}} \in C$, and every variable assignment A, $c_{\mathrm{S}}(A[S]) \in N$ represents the cost of the constraint for the given assignment, where $A[S]$ is the projection of A on the constraint scope S. The objective is then to find an assignment A to all variables such that the sum of all tuple costs $\sum_{c_{\mathrm{S}} \in C} c_{\mathrm{S}}(A[S])$ is minimum. This is called the Weighted Constraint Satisfaction Problem (WCSP) which is NP-hard [16].

A CSP is node consistent if, for every variable X_i, every value of its domain is allowed for the unary constraints on X_i. And a CSP is arc consistent if, besides being node consistent, for every constraint on two variables X_i and Y_j, for all $a \in d(X_i)$, there exists $b \in d(Y_j)$, such that (a, b) is in the constraint [28, 12].

Node consistency and arc consistency are techniques that potentially rule out many inconsistent assignments at an early stage of the search, and can effectively cut short the search for a

consistent assignment [28].

4.3 Translating CSP into SAT

The work developed in this thesis is partially based on previous work which concerns the mapping and encoding of CSP problems into SAT problems [11, 24, 37, 38, 39, 12, 40].

There has been a significant amount of research in the past years in the development of algorithms for both SAT and CSP problems, and many improvements were developed for both approaches. Therefore, the translation of CSP instances into SAT (and vice-versa) can prove itself beneficial. For CSPs, enforcing arc consistency is often the best tradeoff between the amount of pruning that needs to be done and the cost it actually takes [41].

The main encodings from CSP to SAT (and the ones that were more extensively explored in this work) are the *direct encodings* [12, 11] and the *support encodings* [42]. There is another mapping named *log-encoding* [38] which as not been considered.

Another important encoding is the *minimal support encoding* which also has great importance in this work [12].

Next, we describe the following encodings:

4.3.1 Direct Encoding

In the direct encoding of CSP into SAT, a Boolean variable is added for each value a CSP variable can be assigned to. Formally, for each value j of variable X_i, a Boolean variable x_{ij} is added to the SAT instance. Assuming a CSP variable has a domain of size m, the direct encoding contains clauses that ensures that each variable is given a value: $\exists_i x_{i1} \vee \cdots \vee xim$ (called *at-least-one* clauses); and contains clauses that ensure that ensure that each variable takes no more than one value (called *at-most-one* clauses). Also, there exist clauses that eliminate the possibility of combinations of forbidden value assignments (called *conflict-clauses*). For example considering that, in a given CSP problem, $X_1 = 2$ and $X_3 = 1$ is not allowed, then the clause $\overline{x_{12}} \vee \overline{x_{31}}$ is added to the SAT mapping. The following example[1] depicts a direct support encoding of a CSP problem into SAT.

Example 1 *The direct encoding of the CSP* $\langle X, D, C \rangle = \langle \{X, Y\}, \{d(X) = \{1, 2, 3\}, d(Y) = \{1, 2, 3\} \}, \{X \leq Y\} \rangle$ *contains the following clauses:*

[1] This example was borrowed from [12]

at-least-one	$x_1 \vee x_2 \vee x_3$	$y_1 \vee y_2 \vee y_3$	
at-most-one	$\overline{x}_1 \vee \overline{x}_2$	$\overline{x}_1 \vee \overline{x}_3$	$\overline{x}_2 \vee \overline{x}_3$
	$\overline{y}_1 \vee \overline{y}_2$	$\overline{y}_1 \vee \overline{y}_3$	$\overline{y}_2 \vee \overline{y}_3$
conflict	$\overline{x}_2 \vee \overline{y}_1$	$\overline{x}_3 \vee \overline{y}_1$	$\overline{x}_3 \vee \overline{y}_2$

4.3.2 Support Encoding

The support encoding (also called *AC-encoding* [41]) encodes into clauses the support for a value instead of encoding the conflicts [42, 12, 37]. The support of an assignment $i = v$ (meaning variable i is assigned value v) across a constraint is the set of assignments of the other variables which allow $i = v$. Formally, if $v_1, v_2, \ldots, v_k$ are the supporting values for $j = w$ in variable i, then the supporting clause will be $x_{i,v_1} \vee x_{i,v_2} \vee \ldots \vee x_{i,v_k} \vee \overline{x}_{j,w}$. This clause is equivalent to $x_{j,w} \rightarrow x_{i,v_1} \vee x_{i,v_2} \vee \ldots \vee x_{i,v_k}$ which means that as long as $x_{j,w}$ holds (that is, $x_{j,w} \neq FALSE$), then at least one of the supports must hold.

There is one support clause for each pair of variables i, j involved in a constraint, and for each value in the domain of j. Unlike conflict clauses, a similar clause in each *direction* is necessary, one for the pair i, j and one for j, i. As an example of a support encoding, lets take into consideration example 4.3.1. The support encoding will have the same at-least-one and at-most-one clauses whereas the conflict clauses will be replaced by support clauses[2]:

Example 2 *Example of a support encoding.*

at-least-ones	$x_1 \vee x_2 \vee x_3$	$y_1 \vee y_2 \vee y_3$	
at-most-one	$\overline{x}_1 \vee \overline{x}_2$	$\overline{x}_1 \vee \overline{x}_3$	$\overline{x}_2 \vee \overline{x}_3$
	$\overline{y}_1 \vee \overline{y}_2$	$\overline{y}_1 \vee \overline{y}_3$	$\overline{y}_2 \vee \overline{y}_3$
support	$\overline{x}_2 \vee y_2 \vee y_3$	$\overline{y}_1 \vee x_1$	
	$\overline{x}_3 \vee y_3$	$\overline{y}_2 \vee x_1 \vee x_2$	

In this example, the support clause x_1 is missing because it is subsumed by $y_1 \vee y_2 \vee y_3$ and the support clause for y_3 is missing because it is subsumed by $x_1 \vee x_2 \vee x_3$.

4.3.3 Minimal Support Encoding

The minimal support encoding [12] is considered a variant of the support encoding. The basis of this encoding starts with the observation that the support encoding contains unnecessary

[2]This example was also borrowed from [12]

clauses. For a binary constraint C_k with scope $\{X, Y\}$, it is enough to add the support for either the values of X or the values of Y; it is not necessary to add a clause in each *direction*.

Example 3 *Example of a minimal support encoding*[3].

at-least-ones	$x_1 \vee x_2 \vee x_3$	$y_1 \vee y_2 \vee y_3$	
at-most-one	$\overline{x}_1 \vee \overline{x}_2$	$\overline{x}_1 \vee \overline{x}_3$	$\overline{x}_2 \vee \overline{x}_3$
	$\overline{y}_1 \vee \overline{y}_2$	$\overline{y}_1 \vee \overline{y}_3$	$\overline{y}_2 \vee \overline{y}_3$
support	$\overline{x}_2 \vee y_2 \vee y_3$	$\overline{x}_3 \vee y_3$	

4.4 Encoding Max-CSP into Partial Max-SAT

In this section, we will extend the previous encodings to encode Max-CSP instances into partial Max-SAT.

4.4.1 Direct Encoding for Partial Max-SAT

The direct encoding for Max-CSP adapts the direct encoding from CSP into SAT [12].

The direct encoding of a Max-CSP instance $\langle X, D, C\rangle$ is the partial max-SAT instance that contains the at-least-one and the at-most-one clauses as hard clauses for every CSP variable in X, and contains a soft clause for every *nogood* of every constraint of C. The hard clauses are represented in square brackets and the soft clauses are represented in curly brackets and the soft clauses are represented in parentheses.

Example 4 *The partial max-SAT encoding for the max-CSP problem of the CSP instance in Example 4.3.1. Hard clauses are represented in square brackets and soft clauses in round brackets.*

at-least-ones	$[x_1 \vee x_2 \vee x_3]$	$[y_1 \vee y_2 \vee y_3]$	
at-most-one	$[\overline{x}_1 \vee \overline{x}_2]$	$[\overline{x}_1 \vee \overline{x}_3]$	$[\overline{x}_2 \vee \overline{x}_3]$
	$[\overline{y}_1 \vee \overline{y}_2]$	$[\overline{y}_1 \vee \overline{y}_3]$	$[\overline{y}_2 \vee \overline{y}_3]$
conflict	$(\overline{x}_2 \vee \overline{y}_1)$	$(\overline{x}_3 \vee \overline{y}_1)$	$(\overline{x}_3 \vee \overline{y}_2)$

4.4.2 Minimal Support Encoding for Partial Max-SAT

The minimal support encoding for Max-CSP is, in similarly to the direct encoding for Max-CSP, also based on the minimal support encoding from CSP into SAT [12].

[3] Another example from [12].

The minimal support encoding of a Max-CSP instance $\langle X, D, C\rangle$ is the partial max-SAT instance that contains as hard clauses the corresponding at-least-one and at-most-one clauses for every CSP variable in X, and contains as soft clauses the minimal support clauses.

Example 5 *The partial max-SAT minimal support encoding of the Max-CSP problem of the CSP instance in example 4.3.1.*

at-least-one	$[x_1 \vee x_2 \vee x_3]$	$[y_1 \vee y_2 \vee y_3]$	
at-most-one	$[\overline{x}_1 \vee \overline{x}_2]$	$[\overline{x}_1 \vee \overline{x}_3]$	$[\overline{x}_2 \vee \overline{x}_3]$
	$[\overline{y}_1 \vee \overline{y}_2]$	$[\overline{y}_1 \vee \overline{y}_3]$	$[\overline{y}_2 \vee \overline{y}_3]$
support	$(\overline{x}_2 \vee y_2 \vee y_3)$	$(\overline{x}_3 \vee y_3)$	

4.4.3 Support Encoding

The support encoding of a Max-CSP instance $\langle X, D, C\rangle$ is similar to the direct and minimal support encodings, an extension the support encoding from CSP into SAT. It contains, as hard clauses, the at-least-one and at-most-one clauses for every CSP variable in X, and contains, for every constraint $C_k \in C$ with scope $\{X, Y\}$, a soft clause of the form $S_{X=j} \vee c_k$ for every support clause $S_{X=k}$ encoding the support for the value j of the CSP variable X, where c_k is an auxiliary variable, and a soft clause of the form $S_{Y=m} \vee \overline{c}_k$ for every support clause $S_{Y=m}$ encoding the support for value m of the CSP variable Y.

Example 6 *The partial max-SAT support encoding of the Max-CSP problem of the CSP instance in example 4.3.1.*

at-least-ones	$[x_1 \vee x_2 \vee x_3]$	$[y_1 \vee y_2 \vee y_3]$	
at-most-one	$[\overline{x}_1 \vee \overline{x}_2]$	$[\overline{x}_1 \vee \overline{x}_3]$	$[\overline{x}_2 \vee \overline{x}_3]$
	$[\overline{y}_1 \vee \overline{y}_2]$	$[\overline{y}_1 \vee \overline{y}_3]$	$[\overline{y}_2 \vee \overline{y}_3]$
support	$(\overline{x}_2 \vee y_2 \vee y_3 \vee c_1)$	$(\overline{y}_1 \vee x_1 \vee \overline{c}_1)$	
	$(\overline{x}_3 \vee y_3 \vee c_1)$	$(\overline{y}_2 \vee x_1 \vee x_2 \vee \overline{c}_2)$	

When introducing auxiliary variables, it is guarantied that the number of violated CSP constraints is the same as the number of violated partial max-SAT clauses [12].

A proof for the previous statement is given in [12] and it is done by using some defined MAX-SAT rules of inference such as the complementary unit clause [32].

5

Pedigrees and Maximum Satisfiability

Remembering chapter 2 and the concepts defined in the previous chapter, it is now possible to present the mappings of pedigree structures into max-SAT structures in order to search for possibly faster and more efficient ways to solve the pedigree consistency checking problem.

The main objective of this work was to start from the encodings of pedigrees into Max-CSP defined in [16] and described in section 3.5, and then extend those encodings and translate them into Partial Max-SAT. This was to be done by means of the *minimal support encoding* and the *support encoding* [12, 37, 41].

Because the pedigree consistency checking problem involves ternary constraints (parental relations combine father, mother and a descendant), the simple support encoding was found to be unusefull for solving this problem. Since the support encoding uses support clauses to encode the supports of a *single* variable into another *single* variable [41], it can only be used for binary networks, and the problem at hand requires solving ternary networks.

An alternative was to employ the *k-AC encoding* [41] which is a generalization of the *AC encoding* [42]. These encoding will be explained further in this section.

5.1 Mapping pedigree structures

When mapping pedigree structures it is first necessary to relate the characteristics found on them to the characteristics of partial Max-SAT instances.

Information on parental relations are mapped in the structure of a pedigree. Therefore that information needs to be mapped into partial Max-SAT. Considering that parental information is assumed to always be correct, there will exist a number of *hard clauses* in partial Max-SAT that describe each of the parental relations. Since those relations involve three individuals, the clauses will be ternary.

Also, a pedigree may come with genotype information on some (maybe all) individuals. For each individual that has a genotype associated to the pedigree, an unary clause is introduced in the partial Max-SAT instance. Since the consistency checking problem is based on the correct assignment of genetic information to each individual present in the pedigree, these clauses will be *soft*: this information may possibly be wrong.

Still, before mapping parental relations into ternary clauses and genotype information into unary clauses, it is first necessary to define the structure of the partial Max-SAT instance, that is, it is first necessary to define the partial Max-SAT variables that will compose the instance. Since a pedigree instance posesses n individuals and each individual may be assigned, at first, to every combination m of alleles in the pedigree, there will be a total of $m \times n$ variables in the partial Max-SAT instance.

Considering figure 5.2, which has 3 individuals and two different alleles (that can be combined in three different ways), there will be a total of 9 partial Max-SAT variables. They are the following:

$$\begin{array}{ccc} X_{1_{AA}} & X_{1_{AB}} & X_{1_{BB}} \\ X_{2_{AA}} & X_{2_{AB}} & X_{2_{BB}} \\ X_{3_{AA}} & X_{3_{AB}} & X_{3_{BB}} \end{array}$$

As observed in figure 5.1 there are five parental relationships (between individuals 1, 2 and 5, individuals 1, 2 and 6, individuals 3, 4 and 7, individuals 3, 4 and 8 and individuals 6, 7 and 9), and all individuals have an associated genotype. For all of the parental relationships there will be a set of ternary clauses that describes them (those sets of clauses represent the CSP constraints of all genotype values that can be assigned to the individuals present according to Mendelian laws of inheritance). Finally, for all associated genotypes there will be an unary clause that states that a certain individual x possesses, for example, information AB.

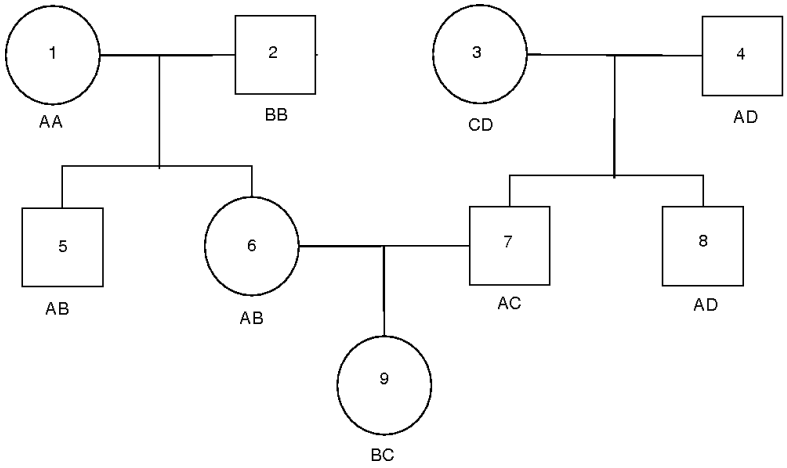

Figure 5.1: Example of a pedigree with 3 parental relationships with an assigned genotype for every individual.

5.1.1 Mapping into Minimal Support Encoding

The mapping of pedigree problems described in CSP instances into the minimal support encoding is described as follows.

First, and like all of the detailed encodings in the previous chapter, the *at-least-one* and the *at-most-one* clauses are created in order to assign one genotype to one individual. These clauses will be *hard* clauses as they cannot be unsatisfied.

Second, the ternary constraints are mapped (representing parental relationships). Since in partial Max-SAT (and also in SAT and Max-SAT) a CSP variable will produce m variables (being m the arity of the CSP domain for that variable), all of the combinations between an individual and the respective genotype that conform to the Mendelian laws of inheritance must be described into ternary clauses. These clauses will also be *hard* ones since, assuming that parental information in pedigrees is always correct, by violating one of these clauses the problem itself would be unsatisfiable.

Finally, the unary constraints are mapped. This is done simply by indicating the SAT variable that takes the allowed domain value for that specific CSP value (individual). The result is a *soft* unary clause for each indicated variable.

It is important to state that the only *soft* clauses in the partial Max-SAT instance will be the unary ones due to the fact that these clauses are the ones that represent an assignment of a genotype to an individual. The pedigree consistency checking problem corresponds to finding

the minimum number of wrongly assigned genotypes in a pedigree (or the maximum number of correctly assigned ones). A simple example follows:

Example 1 *This example begins first by using a small pedigree instance and only then are the clauses specified.*

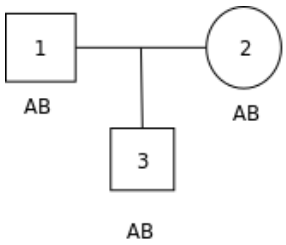

Figure 5.2: A simple pedigree with associated genotype. The pedigree has three individuals, each with genotype **AB**. A family relationship is also observable between the individuals (individuals 1 and 2 are parents of 3).

at-least-one clauses

$[x_{1_{AA}} \vee x_{1_{AB}} \vee x_{1_{BB}}]$ $[x_{2_{AA}} \vee x_{2_{AB}} \vee x_{2_{BB}}]$ $[x_{3_{AA}} \vee x_{3_{AB}} \vee x_{3_{BB}}]$

at-most-one clauses

$[\overline{x}_{1_{AA}} \vee \overline{x}_{1_{AB}}]$ $[\overline{x}_{1_{AA}} \vee \overline{x}_{1_{BB}}]$ $[\overline{x}_{1_{AB}} \vee \overline{x}_{1_{BB}}]$
$[\overline{x}_{2_{AA}} \vee \overline{x}_{2_{AB}}]$ $[\overline{x}_{2_{AA}} \vee \overline{x}_{2_{BB}}]$ $[\overline{x}_{2_{AB}} \vee \overline{x}_{2_{AB}}]$
$[\overline{x}_{3_{AA}} \vee \overline{x}_{3_{AB}}]$ $[\overline{x}_{3_{AA}} \vee \overline{x}_{3_{BB}}]$ $[\overline{x}_{3_{AB}} \vee \overline{x}_{3_{BB}}]$

minimal support clauses

$[x_{1_{AA}} \wedge x_{2_{AA}} \rightarrow x_{3_{AA}}]$ $[x_{1_{AA}} \wedge x_{2_{AB}} \rightarrow x_{3_{AA}} \vee x_{3_{AB}}]$ $[x_{1_{AA}} \wedge x_{2_{BB}} \rightarrow x_{3_{AB}}]$
$[x_{1_{AB}} \wedge x_{2_{AA}} \rightarrow x_{3_{AA}} \vee x_{3_{AB}}]$ $[x_{1_{AB}} \wedge x_{2_{AB}} \rightarrow x_{3_{AA}} \vee x_{3_{AB}} \vee x_{3_{BB}}]$ $[x_{1_{AB}} \wedge x_{2_{BB}} \rightarrow x_{3_{AA}} \vee x_{3_{BB}}]$
$[x_{1_{BB}} \wedge x_{2_{AA}} \rightarrow x_{3_{AB}}]$ $[x_{1_{BB}} \wedge x_{2_{AB}} \rightarrow x_{3_{AB}} \vee x_{3_{BB}}]$ $[x_{1_{BB}} \wedge x_{2_{BB}} \rightarrow x_{3_{BB}}]$

unary clauses

$(x_{1_{AB}})$ $(x_{2_{AB}})$ $(x_{3_{AB}})$

For each individual a where $a = \{1, 2, 3\}$ there exists a set of variables x_{aij} where ij is a combination of alleles A and B (for this specific pedigree these combinations will be $ij = \{AA, AB, BB\}$). Regarding the ternary clauses, each clause of the type, for instance $x_{1ij} \wedge x_{2ij} \rightarrow x_{3ij}$ is equivalent to $\overline{x}_{1ij} \vee \overline{x}_{2ij} \vee x_{3ij}$ which means that, as long as $x_{1ij} \wedge x_{2ij}$ hold (that is $x_{1ij} \wedge x_{2ij} \neq False$) then x_{3ij} must hold.

One of the drawbacks of the minimal support encoding is that it does not maintain *arc consistency* through unit propagation [12]. On the other hand, the *k-AC encoding* does maintain it.

5.1.2 Mapping into k-AC Encoding

The *k-AC encoding* is considered a generalization of the *support encoding* [41]. Contrary to the support encoding, it can be extended to any constraint arity other than binary.

The main difference between the support encoding and the *k-AC encoding* is that the support clauses in the support encoding are replaced by *k-AC* clauses. *k-AC* represents the encoding of supports on subsets S of variables of any size, for an instantiation of another subset T of any size.

Because a literal stands for an assignment, an instantiation or a support of several variables corresponds to a conjunction of positive literals.

Let $[v_1, \cdots, v_p]$ be a support on $X_1, \cdots, X_p$ of a given instantiation on other variables. The conjunction that encodes this support is the clause $(X_1v_1 \wedge \cdots \wedge X_pv_p)$. And to maintain the encoding in CNF, an extra variable s is necessary for this support, and the following equivalence $s \leftrightarrow (X_1v_1 \wedge \cdots \wedge X_pv_p)$. This clause, in turn, is equivalent to the following clauses: $(\overline{s} \vee X_1v_1), \cdots, (\overline{s} \vee X_pv_p)$ and $(\overline{X}_{1v_1} \vee \cdots \vee \overline{X}_{pv_p} \vee s)$.

S is called the *support variable*, and if the support is unary (a single $Y = v$ for instance), there is actually no need for an extra variable, and the support variable is the corresponding boolean variable (Y_v).

Formally, let C_S be a constraint, $T = \{X_1, \cdots, X_k\} \subset S$ be a set of k variables, $I = [v_1 \in D(X_1), \cdots, v_k \in D(X_k)]$ an instantiation of T and $\{s_1, \cdots, s_m\}$ the supports of I on $S - T$. Then the following *k-AC* clause is added: $\overline{X}_{1v_1} \vee \cdots \vee \overline{X}_{kv1} \vee s_1 \vee s_2 \cdots \vee s_m$.

This previous clause is equivalent to $I \rightarrow (s_1 \vee s_2 \vee \cdots s_m)$, which means that one of the supports s_m must hold as long as I holds.

When all supports of instantiation I are falsified, I will be itself falsified and the *k-AC* clause is reduced to the conflict clause of length k forbidding instantiation I.

Figure 5.3 shows four possible *k-AC* encodings for a given ternary constraint.

As observed in the tables, for a ternary constraint, there are four possible ways to create its corresponding *k-AC* encoding. These different forms of encoding will depend on the arity of the instantiation (and, of course, the arity of the support).

A 0-AC encoding signifies that the instantiation corresponds to 0 variables, and all that is left is the support conjunction. Hence, the use of auxiliary variables.

The 1-AC encoding corresponds to the instantiation of one variable, when the other two variables in the constraint will be its support.

X	Y	Z
a	a	b
a	b	b
b	a	a
b	a	b

0-AC encoding	3-AC encoding
$True \rightarrow (S_1 \vee S_2 \vee S_3 \vee S_4) \wedge$ $(X_a \wedge Y_a \wedge Z_b) \leftrightarrow S_1 \wedge$ $(X_a \wedge Y_b \wedge Z_b) \leftrightarrow S_2 \wedge$ $(X_b \wedge Y_a \wedge Z_a) \leftrightarrow S_3 \wedge$ $(X_b \wedge Y_a \wedge Z_b) \leftrightarrow S_4$	$((X_a \wedge Y_a \wedge Z_a \rightarrow False) \wedge$ $((X_a \wedge Y_b \wedge Z_a \rightarrow False) \wedge$ $((X_b \wedge Y_b \wedge Z_a \rightarrow False) \wedge$ $((X_b \wedge Y_b \wedge Z_b \rightarrow False)$

2-AC encoding	1-AC encoding
$((X_a \wedge Y_a \rightarrow Z_b) \wedge$ $((X_a \wedge Y_b) \rightarrow Z_b \wedge$ $((X_b \wedge Y_a) \rightarrow (Z_b \vee Z_a)) \wedge$ $((X_b \wedge Y_b) \rightarrow False) \wedge$ $((X_a \wedge Z_a) \rightarrow False)$ $((X_a \wedge Z_a) \rightarrow (Y_a \vee Y_b)) \wedge$ $((X_b \wedge Z_a) \rightarrow Y_a) \wedge$ $((X_b \wedge Z_b) \rightarrow Y_a) \wedge$ $((Y_a \wedge Z_a) \rightarrow X_b) \wedge$ $((Y_a \wedge Z_b) \rightarrow (X_a \vee X_b) \wedge$ $((Y_b \wedge Z_a) \rightarrow False) \wedge$ $((Y_b \wedge Z_b) \rightarrow X_a)$	$(X_a \rightarrow (S_1 \vee S_2) wedge$ $(X_b \rightarrow (S_3 \vee S_1) \wedge$ $(Y_a \rightarrow S_4 \vee S_5 \vee S_6)) \wedge$ $(Y_b \rightarrow S_4) \wedge$ $(Z_a \rightarrow S_7) \wedge$ $(Z_b \rightarrow (S_7 \vee S_8 \vee S_9)) \wedge$ $((Y_a \wedge Z_b) \leftrightarrow S_1) \wedge$ $((Y_b \wedge Z_b) \leftrightarrow S_2) \wedge$ $((Y_a \wedge Z_a) \leftrightarrow S_3) \wedge$ $((X_a \wedge Z_b) \leftrightarrow S_4) \wedge$ $((X_b \wedge Z_a) \leftrightarrow S_5) \wedge$ $((X_b \wedge Z_b) \leftrightarrow S_6) \wedge$ $((X_b \wedge Y_a) \leftrightarrow S_7) \wedge$ $((X_a \wedge Y_a) \leftrightarrow S_8) \wedge$ $((X_a \wedge Y_b) \leftrightarrow S_9)$

Figure 5.3: The first table describes a ternary constraint involving variables X, Y, Z, the allowed tuples are provided. The second table describes the four possible k-AC encodings of the constraint. This example was borrowed from [41].

The 2-AC encoding represents the instantiation of two variables with only one variable remaining as the support.

The 3-AC encoding is the case that corresponds to the instantiation of all three variables. As observed in the second table of figure 5.3, the clauses that represent this encoding are all conflict clauses. This is because, if the clauses were to correspond to allowed tuples, they would be of the type, for instance, $X_a \wedge Y_a \wedge z_b \rightarrow True$, which is useless. Instead, the nogoods are mapped, and will originate an encoding of conflict clauses.

This implies that the k-AC encoding is a generalization of not only the support encoding but also of the direct encoding and, therefore, direct and support encodings are particular cases of the k-AC encoding.

Out of these four possible encodings, the one that can be used for encoding the consistency checking problem will be the 2-AC encoding. This is because, when modelling parental relationships, it is necessary to verify which values of a variable support the values of two other variables (in this case, for instance, which values of a child support the values of their two parents).

The following is an example of a pedigree mapped into the 2-AC encoding. The basis of the problem is the same as the one used for describing the example in the previous section regarding

the minimal support encoding.

Example 2 *Considering figure 5.2, the clauses under the k-AC encoding will be:*
at-least-ones clauses

$[x_{1_{AA}} \vee x_{1_{AB}} \vee x_{1_{BB}}]$ $[x_{2_{AA}} \vee x_{2_{AB}} \vee x_{2_{BB}}]$ $[x_{3_{AA}} \vee x_{3_{AB}} \vee x_{3_{BB}}]$

at-most-ones clauses

$[\overline{x}_{1_{AA}} \vee \overline{x}_{1_{AB}}]$ $[\overline{x}_{1_{AA}} \vee \overline{x}_{1_{BB}}]$ $[\overline{x}_{1_{AB}} \vee \overline{x}_{1_{BB}}]$
$[\overline{x}_{2_{AA}} \vee \overline{x}_{2_{AB}}]$ $[\overline{x}_{2_{AA}} \vee \overline{x}_{2_{BB}}]$ $[\overline{x}_{2_{AB}} \vee \overline{x}_{2_{AB}}]$
$[\overline{x}_{3_{AA}} \vee \overline{x}_{3_{AB}}]$ $[\overline{x}_{3_{AA}} \vee \overline{x}_{3_{BB}}]$ $[\overline{x}_{3_{AB}} \vee \overline{x}_{3_{BB}}]$

k-AC clauses

$[x_{1_{AA}} \wedge x_{2_{AA}} \rightarrow x_{3_{AA}}]$ $[x_{1_{AA}} \wedge x_{2_{AB}} \rightarrow x_{3_{AA}} \vee x_{3_{AB}}]$ $[x_{1_{AA}} \wedge x_{2_{BB}} \rightarrow x_{3_{AB}}]$
$[x_{1_{AB}} \wedge x_{2_{AA}} \rightarrow x_{3_{AA}} \vee x_{3_{AB}}]$ $[x_{1_{AB}} \wedge x_{2_{AB}} \rightarrow x_{3_{AA}} \vee x_{3_{AB}} \vee x_{3_{BB}}]$ $[x_{1_{AB}} \wedge x_{2_{BB}} \rightarrow x_{3_{AB}} \vee x_{3_{BB}}]$
$[x_{1_{BB}} \wedge x_{2_{AA}} \rightarrow x_{3_{AB}}]$ $[x_{1_{BB}} \wedge x_{2_{AB}} \rightarrow x_{3_{AB}} \vee x_{3_{BB}}]$ $[x_{1_{BB}} \wedge x_{2_{BB}} \rightarrow x_{3_{BB}}]$
$[x_{1_{AA}} \wedge x_{3_{AA}} \rightarrow x_{2_{AA}} \vee x_{2_{AB}}]$ $[x_{1_{AA}} \wedge x_{3_{AB}} \rightarrow x_{2_{AB}} \vee x_{2_{BB}}]$ $[x_{1_{AA}} \wedge x_{3_{BB}} \rightarrow False]$
$[x_{1_{AB}} \wedge x_{3_{AA}} \rightarrow x_{2_{AA}} \vee x_{2_{AB}}]$ $[x_{1_{AB}} \wedge x_{3_{AB}} \rightarrow x_{2_{AA}} \vee x_{2_{AB}} \vee x_{2_{BB}}]$ $[x_{1_{AB}} \wedge x_{3_{BB}} \rightarrow x_{2_{AB}} \vee x_{2_{BB}}]$
$[x_{1_{BB}} \wedge x_{3_{AA}} \rightarrow False]$ $[x_{1_{BB}} \wedge x_{3_{AB}} \rightarrow x_{2_{AA}} \vee x_{2_{AB}}]$ $[x_{1_{BB}} \wedge x_{3_{BB}} \rightarrow x_{2_{AB}} \vee x_{2_{BB}}]$
$[x_{2_{AA}} \wedge x_{3_{AA}} \rightarrow x_{1_{AA}} \vee x_{1_{AB}}]$ $[x_{2_{AA}} \wedge x_{3_{AB}} \rightarrow x_{1_{AB}} \vee x_{1_{BB}}]$ $[x_{2_{AA}} \wedge x_{3_{BB}} \rightarrow False]$
$[x_{2_{AB}} \wedge x_{3_{AA}} \rightarrow x_{1_{AA}} \vee x_{1_{AB}}]$ $[x_{2_{AB}} \wedge x_{3_{AB}} \rightarrow x_{1_{AA}} \vee x_{1_{AB}} \vee x_{1_{BB}}]$ $[x_{2_{AB}} \wedge x_{3_{BB}} \rightarrow x_{1_{AB}} \vee x_{1_{BB}}]$
$[x_{2_{BB}} \wedge x_{3_{AA}} \rightarrow False]$ $[x_{2_{BB}} \wedge x_{3_{AB}} \rightarrow x_{1_{AA}} \vee x_{1_{AB}}]$ $[x_{2_{BB}} \wedge x_{3_{BB}} \rightarrow x_{1_{AB}} \vee x_{1_{BB}}]$

unary clauses

$(x_{1_{AB}})$ $(x_{2_{AB}})$ $(x_{3_{AB}})$

Analyzing the listing of clauses necessary to describe the instance with the *k-AC* encoding (2-AC to be more precise), the first noticeable feature is that there exist three times more *k-AC* clauses than support clauses in the corresponding minimum support encoding. This is because, since the objective is to model a parental relationship, not only the clauses of the type $father \wedge mother \rightarrow child$ are present, but also the clauses of the type $father \wedge son \rightarrow mother$ and $mother \wedge son \rightarrow father$ are present. As defined by the 2-AC encoding, all of the variables need to be combined two by two in an instantiation with the remaining variable acting as the support. In the pedigree context, the father, mother and child variables need to be combined two by two as a conjunction to form an instantiation. This, in turn, and regarding the ternary constraint, will generate three times more clauses than the corresponding minimal support encoding.

Also in this encoding is the presence of conflict clauses alongside support clauses. For instance, clauses of the type $x_{1_{BB}} \wedge x_{3_{AA}} \rightarrow False$ exist alongside support clauses. Remember that $x_{1_{BB}} \wedge x_{3_{AA}} \rightarrow False$ is equivalent to $\overline{x}_{1_{BB}} \vee \overline{x}_{3_{AA}}$. Extrapolating these clauses upward into the pedigree consistency checking problem, all that they are doing is enforcing the Mendelian laws of inheritance saying that one individual x_1 with genotype AA can never originate individual x_3 with genotype BB (for that to happen individual x_1 must have at least one allele B, being that

the other B allele must come from individual x_2).

5.2 Comparative tests between encodings

Having described new ways to encode the pedigree consistency checking problem, namely, in the minimal support encoding and the 2-AC encoding, what follows is a description of the tests which were run to measure the efficiency of both encodings.

These tests can be divided into two different categories: random pedigree instances tests and real pedigree instances tests. Despite the different types of pedigrees used, the goal remained the same: to verify the efficiency of the direct, minimal support and k-AC encoding, and to identify which of the encodings is the most suitable to encode pedigrees in order to solve the pedigree consistency checking problem.

All of the instances were tested under the same conditions. The *MSUnCore* tool for max-SAT [43] was used. *MSUnCore* tool works by detecting minimal unsatisfiable cores [44, 45] and corrects them, one at a time, until the instance is satisfiable. A minimal unsatisfiable core is the minimal set of unsatisfiable clauses that makes the complete formula unsatisfiable. In case one of those clauses were to be removed the core would be satisfiable.

All of the experiments were performed on a 3 GHz Intel Xeon with 4 GB of RAM.

5.2.1 Random pedigree instances

The random instances used for these tests were the same as the ones used in section 3.6 when comparing Pedcheck and Mendelsoft. They consist of four different collections of instances (A, B, C and D), each one divided in subsets grouped by the number of founders in the pedigree (except for class D where the subsets are grouped by number of founding males in the pedigree). The details of the random pedigree collections are described in table 3.1.

Figures 5.4, 5.5, 5.6, and 5.7 detail the results of running the random pedigree instances with the minimal support encoding against the direct encoding. Figures 5.8, 5.9, 5.10, and 5.11 detail the results for the same instances, but this time comparing the CPU times between the k-AC encoding against the minimal support encoding.

Each figure is a `pairplot` that displays dots along a two dimensional graphic. The two dimensions correspond to the running times of two encodings that are the subject of the current test (one encoding on the horizontal axis and the other one on the vertical axis).

Each dot in a figure represents the running time of a collections' subset for both encodings. And the running time of a subset corresponds to the average of running times of the instances in it. For the same dot, two running times can be observed, one for each encoding on each of the `pairplots` axis.

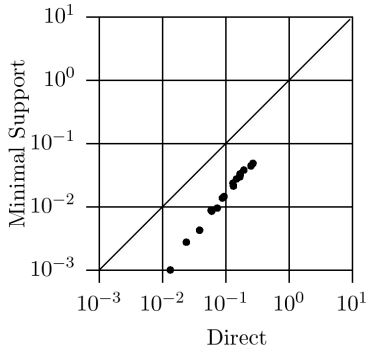

Figure 5.4: CPU times for minimal support encoding vs direct encoding in collection A of random instances.

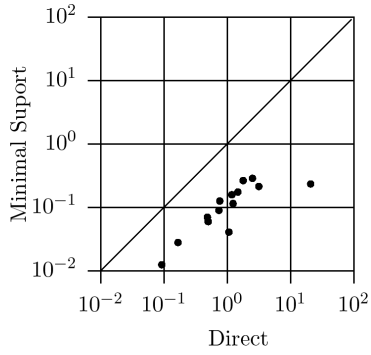

Figure 5.5: CPU times for minimal support encoding vs direct encoding in collection B of random instances.

By comparing the results between direct encoding and minimal support encoding, several observations can be drawn. First, in every test, the increase in the running times increases in a somewhat linear fashion for both encodings. This is due to the increase in the complexity of the instances (resulting from the number of individuals and in the number of the founding males).

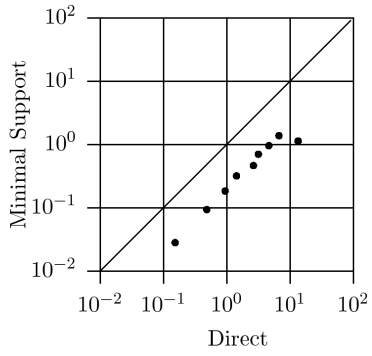

Figure 5.6: CPU times for minimal support encoding vs direct encoding in collection C of random instances.

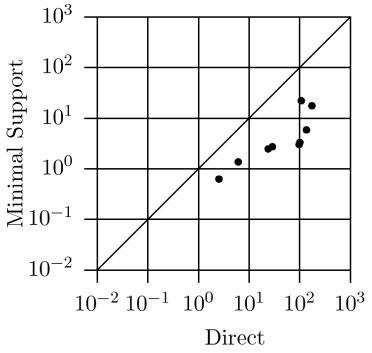

Figure 5.7: CPU times for minimal support encoding vs direct encoding in collection D of random instances.

The difference is in the actual running times that the instances took in the two encodings.

In all tests between the direct and the minimal support encoding, the same instance takes more time with the direct encoding when compared to the same instance with the minimal support. This is observable in all four tests (figures 5.4, 5.5, 5.6, and 5.7). This gap in running times becomes clearer as the complexity of the collections increases: in figures 5.5, 5.6 and also 5.7, this is much more evident: instances that take less than one second to be solved in the minimal support encoding actually take more than ten seconds to run in the direct encoding.

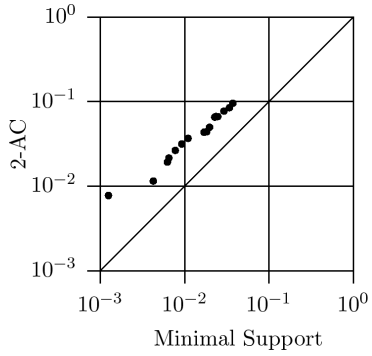

Figure 5.8: CPU times for minimal support encoding vs 2-AC encoding in collection A of random instances.

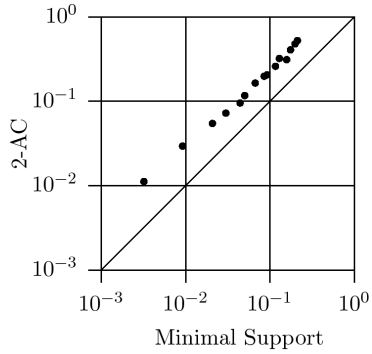

Figure 5.9: CPU times for minimal support encoding vs 2-AC encoding in collection B of random instances.

The tests between the minimal support encoding and the 2-AC encoding displays an almost

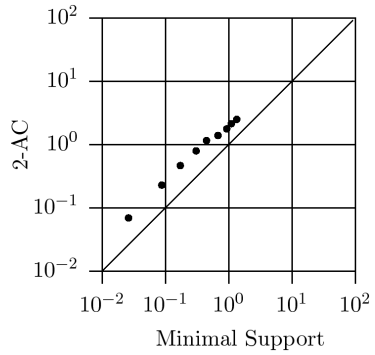

Figure 5.10: CPU times for minimal support encoding vs 2-AC encoding in collection C of random instances.

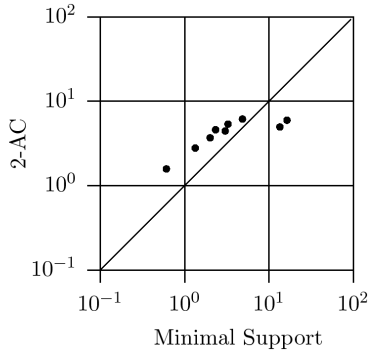

Figure 5.11: CPU times for minimal support encoding vs 2-AC encoding in collection D of random instances.

similar behavior as the tests between the minimal support encoding and the direct encoding.

There exists an almost linear increase of running times in all tests (figures 5.8, 5.9, 5.10, and 5.11), and the running times with the 2-AC encoding are somewhat larger than the running times with the minimal support encoding. This is true for almost all instances except for a particularly small number of subsets in collection D of instances (figure 5.11). In fact there are two subsets of instances in collection D that took over ten seconds with the minimal support encoding and took over less than ten seconds with the 2-AC encoding. This may indicate that, for instances that have a significantly higher degree of complexity, the 2-AC encoding can prove itself to be more effective than the minimal support encoding.

5.2.2 Real pedigree instances

For testing real pedigree problems, not as many real instances could be used as the ones used for the random instance tests. This was because real instances are somewhat more difficult to find than randomly generated ones and when you do find them, a great part of them is unusable. Either because the pedigree is incomplete (when parents are missing from individuals) or erroneous (when female individuals are marked as a male parents and vice versa). This makes the pedigree inconsistent and therefore, cannot be used for testing.

Altogether, five instances were used. Three instances (*connel*, *parkinson*, *sobel*) were taken from [20], and are human genotyped pedigrees. The next two instances are also human genotyped pedigrees describing genetic studies on cancer and eye disease (described in [18]).

Table 5.1: Description of real pedigree instances and cpu running times between direct and minimal support encoding.

	Individuals	Alleles	Direct	Minimal Support
eye	36	6	0.63	0.04
cancer	49	8	10.28	0.23
connel	12	3	0.005	0.001
parkinson	37	5	0.04	0.04
sobel	7	3	0.002	0.00

Analyzing this previous table, it is possible to observe that the running times are larger for the direct encoding than for the minimal support encoding. As in the randomly generated instances, the minimal support encoding is much faster in solving instances when compared to the direct encoding.

Table 5.2: Description of real pedigree instances and cpu running times between 2-AC and minimal support encoding.

	Individuals	Alleles	Minimal Support	2-AC
eye	36	6	0.03	0.08
cancer	49	8	0.17	0.44
connel	12	3	0.001	0.002
parkinson	37	5	0.02	0.01
sobel	7	3	0.002	0.00

The comparison between the minimal support encoding and the 2-AC encoding also bares striking similarities with the corresponding tests on randomly generated instances. The minimal support encoding continues to be faster, even against the 2-AC encoding, but the difference in running times is not as accentuated as when compared to the direct encoding. There is a case where, despite not being very large, the running time in 2-AC is actually faster than the running time in minimal support.

6

Conclusions and Future Work

Pedigree consistency checking is a NP-complete problem and therefore, no efficient solution to solve the problem in polimonial time has been found.

Two types of approaches are identified when tackling this problem: statistical approaches and combinatorial approaches. As the first type of solutions employ statistical methods and tools to solve the problem at hand, the running time takes greater importance than the accuressness of the solution. On the other hand, with combinatorial approaches, the exact opposite occurs: the accuressness of the solution's result takes more importance than the time the actual result takes to be found.

In theory, this difference holds true. However, as shown in section 3.6, the exact opposite occurs. The fact is that a combinatorial tool (Mendelsoft) is orders of magnitude faster than a statistical tool (Pedcheck) for the same set of instances. One of the reasons for this large gap in running times is due to a number of reasons. Out of these reasons, the main one may be due to the underlining principles and concepts used in the tools. As Mendelsoft follows the parsimony principle, it will only try to find the minimum number of errors that, when removed from the data, can fully explain it. Pedcheck, on the other hand, applies statistical algorithms to discover all existing pedigree errors. When instances become large enough in the number of individuals and in the number of errors, the amount of work done in Pedcheck is considerably greater than the amount of work done in Mendelsoft for the same instance.

The main focus of this work was to attempt to use certain SAT encodings and model the

pedigree consistency checking problem into a SAT one, more specifically, into a partial Max-SAT problem. In partial Max-SAT, there are hard clauses and soft clauses, and the objective is to satisfy all the hard clauses and the maximum number of soft ones. Since in the used pedigrees all of the paternal relations were considered correct and the associated genotype information could be considered incorrect, partial Max-SAT instances could be a good approach for this problem by transforming paternal relations into hard clauses and transforming genetic information into soft clauses.

Both minimal support and k-AC encodings were generated not from pedigrees directly but from weighted constraint satisfaction (WCSP) instances. The objective in doing this translation is to transform the WCSP constraints into partial Max-SAT clauses. It proved to be simpler to take the instances in WCSP format and use well known algorithms to transform them into Max-SAT instances, rather than transforming the instances directly from a pedigree format.

One of the difficulties of modelling partial Max-SAT instances was the fact that, despite existing considerable work describing the modelling of WCSP into Max-SAT, most of this work only considers modelling constraints no higher than binary (meaning only two variables in a constraint). Since pedigrees imply parental relationships (which are mapped into WCSP constraints with three variables correlating father, mother and child), it was necessary to model a set of ternary clauses in partial Max-SAT that could take a direct correlation from each ternary constraint defined in a WCSP instance.

With the minimal support encoding, it proved itself simple to map ternary constraints into ternary clauses: it was only necessary to include the extra variable in WCSP as a set of SAT variables comprised of all the possible variables in that WCSP variable domain, and extend the binary clauses into ternary ones.

Regarding the k-AC encoding, this encoding was not originally considered in the proposed work. Instead, only the minimal support encoding and the support encoding were the only initial encodings that were considered for this work. However, after discovering that the support encoding could only be applied for binary constraint networks [41], the k-AC encoding was considered for this work. The difference between the k-AC encoding and the minimal support encoding was in the number of clauses that describe ternary constraints. In the minimal support encoding only clauses in "one direction" are considered, for instance only clauses of the type $father \wedge mother \rightarrow child$. In the k-AC encoding clauses of the type $father \wedge child \rightarrow mother$ and $mother \wedge child \rightarrow father$ were also considered in addition to the previous ones.

It is the inclusion of these extra clauses for every parental relationship that allows for a

particular case of arc consistency (k-AC consistency) to be maintained through unit propagation. This is the difference between the minimal support encoding and the k-AC encoding.

In theory, the possibility of using unit propagation when running Max-SAT instances in a SAT solver should significantly decrease running times of tested instances. However this was not the case: instances encoded by means of the minimal support encoding took less time to run when compared to the k-AC encoding. These running times however, are not as distant between them as when comparing the running times to another encoding (the direct encoding), which could signify that the cost of using k-AC encoding is not that big when compared to the minimal support encoding.

6.1 Future Work

On a final note, other tests with more real pedigree instances (with a higher level of complexity both in the number of individuals and the number of errors) are considered necessary in other to substantiate the concluded work.

Regarding future work, the use of other SAT encodings, namely regular encodings and sequential encodings [39, 40] are suggested in order to condense pedigree instances by reducing the number of clauses.

Another recommended approach would be the use of pseudo-Boolean algorithms and encodings, thus comparing the generated instances against the partial Max-SAT instances.

Bibliography

[1] Göring, H.H.H., Terwillige, J.D.: Linkage Analysis in the Presence of Errors I: Complex-Valued Recombination Fractions and Complex Phenotypes. American Journal of Human Genetics **66**(4) (2000) 1095–1106

[2] Göring, H.H.H., Terwilliger, J.D.: Linkage Analysis in the Presence of Errors II: Marker-locus genotyping errors modeled with Hypercomplex Recombination Fractions. American Journal of Human Genetics **66**(4) (2000) 1107–1118

[3] Ott, J.: Computer-simulation Methods in Human Linkage Analysis. Proceedings of the National Academy of Sciences **86**(11) (1989) 4175–4178

[4] Aceto, L., Hansen, J.A., Ingólfsdóttir, A., Johnsen, J., Knudsen, J.: The Complexity of Checking Consistency of Pedigree Information and Related Problems. Journal of Computer Science and Technology **19**(1) (2004) 42–59

[5] Kowalczykowski, S.C.: Initiation of Genetic Recombination and Recombination-Dependent Replication. Trends in Biochemical Sciences **25**(4) (2000) 156–165

[6] Meselson, M.S., Radding, C.M.: A General Model for Genetic Recombination. Proceedings of the National Academy of Sciences **72**(1) (1975) 358–361

[7] Ehm, M.G., Kimmel, M., Cottingham, R.W.: Error Detection for Genetic Data Using Likelihood Methods. American Journal of Human Genetics **58**(1) (1996) 225–234

[8] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Second Edition. McGraw-Hill Book Company (2001)

[9] Russel, S., Norvig, P.: Artificial Intelligence A Modern Approach, Second Edition. Prentice Hall (2003)

[10] Boros, E., Hammer, P.L.: Pseudo-Boolean Optimization. Discrete Applied Mathematics **123** (2002) 155–255

[11] Walsh, T.: SAT v CSP. In: CP '02: Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming, London, UK, Springer-Verlag (2000) 441–456

[12] Argelich, J., Cabiscol, A., Lynce, I., Manyà, F.: Encoding Max-CSP into Partial Max-SAT. In: ISMVL '08: Proceedings of the 38th International Symposium on Multiple Valued Logic (ismvl 2008), Washington, DC, USA, IEEE Computer Society (2008) 106–111

[13] Piccolboni, A., Gusfield, D.: On the Complexity of Fundamental Computational Problems in Pedigree Analysis. Journal of Computational Biology **10**(5) (2003) 763–773

[14] Saito, M., Saito, A., Kamatani, N.: Web-based detection of Genotype Errors in pedigree data. Journal of Human Genetics **47**(7) (2002) 377–370

[15] Broman, K.: Cleaning Genotype Data. Genetic Epidemiology **17 Suppl 1** (1999) S79–S83

[16] Sanchez, M., Givry, S., Schiex, T.: Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques. Constraints Journal **13**(1-2) (2008) 130–154

[17] Lange, K., Goradia, T.: An Algorithm for Automatic Genotype Elimination. American Journal of Human Genetics **40**(3) (1987) 250–256

[18] O'Connell, J.R., Weeks, D.E.: Pedcheck: A Program for Identification of Genotype Incompatibilities in Linkage Analysis. American Journal of Human Genetics **63**(1) (1998) 259–266

[19] Pirinen, M., Gasbarra, D.: Finding Consistent Gene Transmission Patterns on Large and Complex Pedigrees. IEEE/ACM Transactions on Computational Biology and Bioinformatics **3**(3) (2006) 252–262

[20] O'Connell, J.R., Weeks, D.E.: An Optimal Algorithm for Automatic Genotype Elimination. American Journal of Human Genetics **65** (1999) 1733–1740

[21] Luo, Y., Lin, S.: Finding Starting Points for Markov Chain Monte Carlo Analysis of Genetic Data from Large and Complex Pedigrees. Genetic Epidemiology **25**(1) (2003) 14–24

[22] Manolios, P., Oms, M.G., Valls, S.O.: Checking Pedigree Consistency with PCS. In: Tools and Algorithms for Construction and Analysis of Systems. (2007) 339–342

[23] Manquinho, V.M., Marques-Silva, J.: On Solving Boolean Optimization with Satisfiability-Based Algorithms. In: Sixth International Symposium on Artificial Intelligence and Mathematics. (2000)

[24] Givry, S.D., Larrosa, J., Meseguer, P., Schiex, T.: Solving Max-Sat as Weighted CSP. In: Principles and Practice of Constraint Programming (CP), Springer Verlag (2003) 363–376

[25] Silva, J.P.M., Sakallah, K.A.: GRASP - a New Search Algorithm for Satisfiability. International Conference on Computer-Aided Design (1996) 220–227

[26] En, N., Srensson, N.: Translating Pseudo-Boolean Constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation **2** (2006) 1–26

[27] Manolios, P., Srinivasan, S.K., Vroon, D.: BAT: The Bit-Level Analysis Tool. In: Computer Aided Verification. (2007) 303–306

[28] Barták, R.: Constraint Propagation and Backtracking-Based search: A Brief Introduction to Mainstream Techniques of Constraint Satisfaction. (2005) Available at: http://www.math.unipd.it/∼frossi/cp-school/.

[29] Heckerman, D., Geiger, D., Chickering, D.M.: Learning Bayesian Networks: The combination of Knowledge and Statistical Data. Machine Journal **20**(3) (September 1995) 197–243

[30] Vitezica, Z., Mongeau, M., Manfredi, E., Elsen, J.M.: Selecting Loop Breakers in General Pedigrees. Journal of Human Heredity **57**(1) (2004) 1–9

[31] Vitezica, Z., Elsen, J.M., Ruop, R., Diaz, C.: Using Genotype Probabilities in Survival Analysis: a Scrapie Case. Genetics Selection Evolution **37**(5) (2005) 403–415

[32] Li, C.M., Manyà, F.: MaxSat, Hard and Soft Constraints. In Biere, A., Heule, Marijn J. H.;van Maaren, H.T., eds.: Handbook of Satisfiability. Volume 185., IOS Press (2009) 613–631

[33] Ansótegui, C., Bonet, M.L., Levy, J., Manyà, F.: Inference Rules for High-Order Consistency in Weighted CSP. Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (2007) 167–172

[34] Larrosa, J., Heras, F., de Givry, S.: A Logical Approach to Efficient Max-SAT Solving. Artificial Intelligence **172**(2-3) (2008) 204–233

[35] Bonet, M.L., Levy, J., Manyà, F.: A Complete Calculus for Max-SAT. In: International Conference on Theory and Applications of Safisfiability Testing. Volume 4121 of Lecture Notes in Computer Science., Springer-Verlag, Springer-Verlag (2006) 240–251

[36] Zhang, H., Stickel, M.E.: An Efficient Algorithm for Unit Propagation. In: Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96). (1996) 166–169

[37] Gent, I.P.: Arc Consistency in SAT. In: 15th European Conference on Artificial Intelligence. (2002)

[38] Gavanelli, M.: The Log-Support Encoding of CSP into SAT. In: Constraint Programming. (2007) 815–822

[39] Argelich, J., Cobiscol, A., Lynce, I., Manyà, F.: Sequential encodings from Max-CSP into partial Max-SAT. In: Sat'09. (2009)

[40] Argelich, J., Cobiscol, A., Lynce, I., Manya, F.: Regular Encodings from Max-CSP into Partial Max-SAT. In: International Symposium on Multyple-Value Logic. (2009)

[41] Bessiere, C., Hebrard, E., Walsh, T.: Local Consistencies in SAT. In: Proceedings SAT-2003, Springer (2003) 299–314

[42] Kasif, S.: On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks. Artificial Intelligence **45**(3) (1990) 275–286

[43] Marques-Silva, J.: The MSUNCORE MAXSAT Solver

[44] Marques-Silva, J., Planes, J.: On Using Unsatisfiability for Solving Maximum Satisfiability. coRR **abs/0712.1097** (2007)

[45] Marques-Silva, J., Planes, J.: Algorithms for Maximum Satisfiability using Unsatisfiable Cores. In: DATE '08: Proceedings of the conference on Design, Automation and Test in Europe, New York, NY, USA, ACM (2008) 408–413