Propagating the nVector Constraint

Haplotype Inference using Constraint Programming

Peter Backeman
Abstract

Propagating the nVector Constraint

Peter Backeman

Genetics research is a wide field and needs computer aid in many different areas. One such problem is the haplotype inference problem by pure parsimony (HIPP). In this thesis the HIPP problem is attacked with a constraint programming (CP) model based on the nVector constraint, for which a new propagator is designed. The results show that the current state-of-the-art model based on SAT-solvers are in general the most efficient, but that the CP approach in some cases finds a better solution when time is limited.
Acknowledgements

This thesis wouldn’t have been completed if it were not for the help of several people. First of all, I wish to thank my supervisor, Pierre Flener, and my reviewer, Justin Pearson, whom introduced me to the problem and supported me all the way and provided invaluable feedback. Serdar Kadioğlu were also of great support, helping me to understand what I was doing and giving suggestions of what to try. Another person always ready to discuss my struggles (as well as more trivial subjects) were Joakim Lindqvist, with whom a part of this thesis were co-written. And lastly, big thanks to Linnéa Anglemark at Språkverkstaden for her help improving my writing.
## Contents

1 *Introduction* 1
   1.1 Problem Statement 1
   1.2 Method 1
   1.3 Delimitations 1
   1.4 Report Structure 1

2 *Background* 3
   2.1 Haplotype Inference 4
   2.2 Pure Parsimony 4
   2.3 Previous Work 5
   2.4 Programming Paradigm 5

3 *Constraint Programming* 7
   3.1 Constraint Satisfaction Problem 7
   3.2 Constraint Optimisation Problem 7
   3.3 Propagation 7
   3.4 Consistency 8
   3.5 Global Constraints 8
   3.6 Search 9
      3.6.1 Branching 9
      3.6.2 Exploration 9
   3.7 Main Algorithm 10

4 *nVector Constraint* 12
   4.1 nValue Constraint 12
   4.2 nVector Constraint 12
   4.3 Reusing nValue 13
   4.4 Consistency 14
   4.5 Compatibility 14
   4.6 Lower Bound 14
      4.6.1 Maximum Clique 15
      4.6.2 Maximum Independent Set 16
      4.6.3 Other Methods 17
   4.7 Upper Bound 18
   4.8 Symmetries 18

5 *nVector Propagator* 19
   5.1 Difference Matrix 19
   5.2 Divide and Conquer 19
   5.3 Constructing Connected Components 21
   5.4 Updates 22
      5.4.1 Updates on $N$ 22
      5.4.2 Updates on $V$ 23
   5.5 Consistency 24
   5.6 Guiding Search 25
6 Modelling HIPP in CP

6.1 The Problem ........................................... 26
6.2 Optimisation ........................................... 26
6.3 Preprocessing ......................................... 26
   6.3.1 Switching 1 and 2 ................................ 26
   6.3.2 Removing Singles ................................ 27
6.4 Model .................................................. 27
   6.4.1 Variables ......................................... 27
   6.4.2 Constraints ....................................... 27
   6.4.3 Symmetry Breaking ............................... 28
6.5 Branching Heuristic .................................. 28
   6.5.1 Variable Selection ................................ 28
   6.5.2 Value Selection .................................... 29

7 Results .................................................. 30

7.1 Implementation ....................................... 30
7.2 Test Cases ............................................. 30
7.3 Lower Bounds ......................................... 30
7.4 Optimisation Technique .............................. 32
7.5 Runtimes .............................................. 33
   7.5.1 Effects of extensions ............................ 33
   7.5.2 Comparison with RPoly .......................... 33
7.6 Discussion and Conclusion ........................... 34
7.7 Future Work ........................................... 35
   7.7.1 Limiting Variables ............................... 35
   7.7.2 Bridges ........................................... 35
   7.7.3 ATLEASTNVECTOR and ATMOSTNVECTOR ....... 35
List of Figures

1  Two pairs of haplotypes (a) and (c) that both conflate into the same genotype (b) ......................... 3
2  Example of a NVeCTOR constraint ................................................. 13
3  Compatibility graph (G) .......................................................... 14
4  A minimum hitting set ............................................................. 15
5  Incompatibility graph (\(\overline{G}\)) with a maximum clique .............. 15
6  Greedy clique construction ....................................................... 16
7  Greedy minimal independent set construction ................................ 18
8  Running example of the NVeCTOR propagator .............................. 20
9  Before split, one compatibility graph ........................................ 20
10 After split, first compatibility graph ......................................... 21
11 After split, second compatibility graph ..................................... 21
12 Compatibility graph of second sub-problem after posting \(V_5 = V_6\) 22
13 After split, first compatibility graph ......................................... 23
14 After split, third compatibility graph ......................................... 24
15 A compatibility graph with lower bound 2 .................................. 24
16 Lower Bound (a) .................................................................. 31
17 Lower Bound (b) .................................................................. 31
18 Runtimes for Lower Bound ...................................................... 32
19 Runtimes ............................................................................. 34
1 Introduction

Advances in genetic technology have opened up countless possibilities for future research and development. One of the major challenges lies in creating a genetic map of different genomes (e.g., the human genome). In this process, a main part consists of taking DNA samples and using these for analysing mutations of different genes.

However, while a more complete picture of the genome is preferred, when DNA samples are acquired, often only partial information about the genetic structure is gained. The haplotype inference problem is concerned with extrapolating this limited data in different ways. These calculations can be very computationally expensive and warrant an efficient solver.

1.1 Problem Statement

There has been research on the haplotype inference problem and there are several different solvers using different technologies (see [4, 21, 10]). Nevertheless, none of the existing solutions can solve all instances in reasonable time, so there is room for improvement. In this project Constraint Programming (CP) is used to try and find an alternative solution method and investigate how it compares to already existing solvers.

The purpose of this thesis is to study whether a CP approach can be used to solve the haplotype inference problem efficiently. The approach is to use the nVector constraint to model the problem. However, currently there is no propagator for the nVector constraint and therefore a large part of this project is devoted to developing one.

Together with such a propagator, a model for the problem will be designed and implemented.

1.2 Method

This project starts out by studying the haplotype inference problem and the research done on this problem together with an analysis of the nVector constraint considering if alternative constraints (e.g., nValue) can be a source of inspiration. Afterwards a propagator will be proposed, implemented, and tested on a model for solving the haplotype inference problem.

1.3 Delimitations

This project will focus on developing a model for the haplotype inference problem using the nVector constraint. Other approaches using CP are indeed possible, for example using a selector array or set variables (see [17]). However, due to time limitations other models will not be considered.

1.4 Report Structure

The structure of the rest of the report is as follows. First there is a complete description of the problem (section 2) followed by a brief description of constraint programming (section 3). Afterwards the nVector constraint is explained

\[\text{See http://www.emn.fr/z-info/sdemasse/gccat/Cnvector.html}\]
(section 4), the propagator developed (section 5) and an explanation of how the HIPPP problem in this project has been modelled in CP using it (section 6). Finally the results and evaluation of this CP model are presented together with suggestions of future research (section 7).
2 Background

Humans have 22 pairs of non-sexual chromosomes, where in each pair one chromosome is inherited from the mother and the other from the father. A single-nucleotide polymorphism (commonly referred to as a SNP) is a genetic difference at a single nucleotide (which can be A, T, G or C), between a pair of chromosomes.

For example, if one of the chromosomes has the genetic sequence AGTGAC and the other one has AATGTC (see Figure 1 at the second and second to last position there are two SNPs. At the second position there are two different alleles: G and A. At the second SNP, the alleles are A and T.

Generally, there are only two different alleles for every SNP among a population, one being called the wild allele, the other one is called the mutant allele. The wild allele refers to the most common genetic code at the SNP, while the mutant allele refers to a mutation, which is rarer.

Figure 1: Two pairs of haplotypes (a) and (c) that both conflate into the same genotype (b)

Between different human beings, about 99.9% of the nucleotides are the same, that is most of the DNA is identical (see [10]). Since the nucleotides that differ are what can define to be the differences between human beings, this warrants a focus on the differences, i.e., the SNPs.

A set of SNPs on a single chromosome that are transmitted together is called a haplotype. This entity contains the most detailed information about the chromosomes and is useful in research. However, when a sample of a DNA is acquired, what is often retrieved is the genotype which consists of the conflation (combination) of the two haplotypes of each pair.

The conflation is constructed in the following way: If (for a given position) two haplotypes share the same allele, then the resulting genotype will show this allele at the respective position. Then, if the two haplotypes have different alleles, the genotype will indicate that the haplotypes do not match at the respective position.

Since haplotype data often can be more useful (see [12]), the task of inferring haplotypes from a given genotype is as a very important challenge. Several different approaches have been tried, with varying results, and this thesis focuses on the Pure Parsimony method, which is based on the coalescence theory. The coalescence theory states that most individuals share a common ancestor and therefore share the same mutations, i.e., every mutation at a specific nucleotide has only happened once, because the likelihood that an identical mutation has happened again in another individual is very small.
Using this assumption, if two particular individuals have the same mutant allele at a specific position, they probably share the same haplotype. Since there usually only exists one mutant allele for each SNP, all individuals sharing the same mutant allele are probably sharing the same haplotype as well. Therefore, for a solution to the haplotype inference problem to conform to the pure parsimony approach, the number of different haplotypes should be minimised.

2.1 Haplotype Inference

Haplotype Inference can be abstracted into a mathematical problem in the following way. Model the haplotypes as strings over the alphabet \( \{0, 1\} \), where 0 corresponds to the wild type and 1 to the mutant type. Genotypes are modelled over the alphabet \( \{0, 1, 2\} \), where a 0 or 1 represents that the two haplotypes share the same allele, while a 2 represents the situation when the conflated haplotypes have different alleles.

For example, the genotype string A(G/A)TG(T/A)C could be modeled as 021021, given that the first and fourth position (with A respectively G) represents the wild alleles while the third and sixth (with T and C) represents the mutant alleles. The second and fifth position should definitely be 2, since they are heterozygous sites, while the other need more information to be certainly decided.

A pair of haplotypes explains a genotype if it is of the same length and for each position in the string:

- if the genotype contains a 0 then the haplotype also contains a 0.
- if the genotype contains a 1 then the haplotype also contains a 1.
- if the genotype contains a 2 then one of haplotype contains a 0 and the other a 1.

If a pair of haplotypes explains a genotype, this means that the two haplotypes could be the original conflated haplotypes for the given genotype. However, there is almost always more than one possible pair of haplotype strings explaining a genotype string.

The haplotype inference problem is to find a pair of explaining haplotypes for each genotype \( g \in G \). A trivial solution can always be found in the following way. For each genotype, construct two haplotypes, one with a 0 at every place where there is a 2 in the genotype and one with a 1 (and the rest of the string is decided). For example, given the genotype string 2102, an explaining pair would be 0100 and 1101. However, the pure parsimony approach states as a goal to minimise the number of distinct haplotypes.

2.2 Pure Parsimony

The trivial solution would contain \( 2^n \) haplotypes where \( n \) is the number of genotypes. The pure parsimony approach is based on the fact that individuals from the same population often share a common ancestor and that mutation occurs rarely. Thus the number of distinct haplotypes is assumed to be small. This adds the additional objective to the problem that the number of haplotypes should be minimised (which can be well below \( 2^n \)) and thus the problem stops being trivial.
It has been shown in the past that pure parsimony had an accuracy that is comparable with alternative approaches (see [10]).

Formally, the Haplotype Inference by Pure Parsimony (HIPP) problem is defined as an optimisation problem. Given a set of genotypes \( \mathcal{G} \), find a smallest set of haplotypes \( \mathcal{H} \), such that every genotype \( g_i \in \mathcal{G} \) is explained by at least one pair of haplotypes \( h_{i1}, h_{i2} \in \mathcal{H} \).

For a more detailed description of the HIPP problem see [8], [12] and [13].

2.3 Previous Work

There are several earlier attempts to solve the pure parsimony problem. One of the first models is the RTIP model [13] which uses integer linear programming. It enumerates all possible haplotype pairs and associates a variable which states whether the particular pair is used in the solution or not. The main drawback is that it creates an exponential number of variables and constraints with respect to the length of the genotypes.

The PolyIP model, suggested by Brown and Harrower (see [4]), associates a variable with each site of the explaining haplotypes, thus limiting the number of constraints and variables to be bounded by the number of haplotypes multiplied with the length of the genotypes, avoiding an exponential growth.

The RTIP and PolyIP model were combined by Brown and Harrower in the HybridIP model (see [4]) that tries to utilise the strengths of the two previous models. However, no significant improvement could be shown in comparison to the PolyIP model.

Approaches based on branch-and bound (HAPAR, see [21]) as well as using answer set-programming (HAPLO-ASP, see [5]) have also been tried but the most successful model to date is a SAT-based (boolean satisfiability) solution. In its essence, a SAT problem consists of determining whether a boolean formula can be satisfied and finding values for all boolean variables such that the formula is satisfied. The original SAT model for the HIPP problem, SHIPs, was presented by Lynce et al in 2006 (see [15]). It stated the HIPP problem as a satisfaction problem by setting the number of distinct haplotypes to a lower bound, and then successively increasing until a solution was found. This was extended together with the PolyIP model to form a pseudo-boolean optimisation model, RPoly (see [9]). This is currently the state of the art solution.

A previous attempt with constraint programming was made by Pan in 2009 (see [17]), but a faster solution method was not found.

2.4 Programming Paradigm

In this thesis the HIPP problem will be attacked using Constraint Programming (CP) (an introduction of CP can be read in section 3). As noted above, this technique has been tried once before. In this thesis another model will be adopted instead. The earlier attempt in 2009 by Pan (see [17]) utilised a set model as well as selector arrays. However this did not achieve runtimes on par with other solvers.

Since the current state of the art solution for the HIPP problem is based on SAT, which is also a declarative form of programming, it is important to distinguish what kind of advantage a CP approach would have. One of the key differences is that a CP-based model has the possibility of using more global
constraints, that is each constraint has a broader view of the variables compared to the SAT formulas. By utilising this more propagation can be achieved in every step of the search. A propagator that focuses on constraining the number of distinct vectors in an array will be designed and used in this thesis’ model.
3 Constraint Programming

Constraint Programming (CP) is a declarative form of programming for optimisation problems, i.e., instead of coding how things should be done, all that is necessary is to declare what should be calculated and then a general algorithm attempts to solve the problem. Here follows a short introduction, the more interested reader is referred to [1].

3.1 Constraint Satisfaction Problem

Formally, a constraint satisfaction problem (CSP) is defined as a triple \( \langle X, D, C \rangle \), where \( X \) is a set of decision variables, \( D \) is a set of domains and \( C \) is a set of constraints. Each \( d_i \in D \) represents the possible values, the domain, for variable \( x_i \in X \), here denoted by \( D(x_i) \). Every constraint \( c \in C \) is a relation on one or several of the variables in \( X \). \( x_1 < 2 \cdot x_2 \) and \( x_2 \neq 13 \) are two examples of constraints.

The goal for the general algorithm is to find one or all assignments for all variables in \( X \) such that all relations in \( C \) are satisfied. A major part of this process is called propagation.

The set of decision variables together with their current domains is referred to as the store.

3.2 Constraint Optimisation Problem

A constrained optimisation problem (COP) is defined as \( \langle X, D, C, f \rangle \), where \( X \), \( D \) and \( C \) are the same as for a CSP, and \( f \) is a function from the domains of \( D \) to a number. \( f \) is called the optimisation function, whose value is to be maximised (or minimised).

The goal of a COP is to find a solution that satisfies the problem defined by \( \langle X, D, C \rangle \) with the highest (or lowest) value from \( f \). Note that there may be several best solutions with the same value.

3.3 Propagation

At the core of constraint programming is propagation. By analysing the domains of variables with respect to constraints, values that can not partake in any solution can be removed, pruned.

For example, given the constraint \( x_1 < 3 \) and the domain \( D(x_1) = \{0, 1, 2, 3, 4, 5\} \), propagation can remove the values 3, 4 and 5 from the domain since they can not satisfy the constraint.

In this simple example it is easy to see what values can be removed, however more complicated constraints exist. The piece of software that analyses the domains and eliminates impossible values, for a given constraint, is called a propagator.

The task of a propagator is to monitor the variables of the constraint, and whenever a relevant domain change occurs, to remove values that can be deduced not to exist in a solution. When a propagator can not remove any more values from the current store, it is said to be at fixpoint.

\(^2\)This section was written in collaboration with Joakim Lindqvist.
If at any time the propagator modifies the store such that the domain of a variable becomes empty, or if the propagator can deduce that there is no possible assignment for the variables such that the constraint can be satisfied, the propagator will fail. A failed propagator signals that the problem has no solutions from the current store.

On the other hand, if the propagator can deduce that every possible assignment of the variables of the current store would lead to the constraint being satisfied, then the propagator can subsume itself. A subsumed propagator can therefore be ignored in any subsequent propagation of the current store.

3.4 Consistency

Whenever a propagator has reached fixpoint, it has achieved some level of consistency. Consistency refers to how well supported the values in the domains are, i.e., whether there exist, for every value in a domain, values in the other domains such that the constraint can be satisfied.

Perhaps the simplest form of consistency is value consistency, which means that whenever a variable is assigned a value, all other variables (involved in the same constraint) are pruned such that no conflicting assignment exists. The important aspect of this is that no filtering of values occurs until a variable has been assigned.

One of the strongest consistencies is domain consistency, also known as general arc consistency (GAC). If a propagator has achieved domain consistency when fixpoint has been reached, then every value in every domain has support. A value for a variable is said to have domain consistency support if there exists an assignment for the other variables such that the constraint is not violated. Domain consistency can be very hard to achieve and therefore a weaker form of consistency is sometimes used.

An example of a weaker consistency is bounds consistency. Bounds consistency guarantees that each value in each variable’s domain has support by a value between the bounds of the domain for each other variable.

3.5 Global Constraints

A global constraint can be defined as to express the same as a set of local constraints would. It usually comes with a propagation algorithm that propagates more or cheaper than expressing the global constraint using the local constraints. Global constraints are said to encapsulate a set of other constraints [19][20].

One of the most common examples is the ALLDIFFERENT constraint. For three variables $x_1, x_2, x_3$ and the store $\{x_1 \mapsto \{0, 1\}, x_2 \mapsto \{0, 1\}, x_3 \mapsto \{1, 2\}\}$, posting $x_1 \neq x_2, x_1 \neq x_3$ and $x_2 \neq x_3$ would not yield any propagation, even though one can see that $x_3$ cannot be equal to 1 for all of them to hold. This is since, individually, each propagator does not have the information to derive that conclusion.

By instead posting ALLDIFFERENT($x_1, x_2, x_3$), this information resides in one single constraint, and the domain of $x_3$ is constricted to 2. One global constraint might achieve more propagation than several local ones.
3.6 Search

The second major part of constraint solving is search. The main idea is to ensure that complete search is performed, i.e., every possible combination of assignments of values is considered, at least implicitly. By combining complete search with propagation, a large number of assignments can be removed from the search, and this is the strength of constraint programming. If propagation is removed, then what is left is pure brute-search.

3.6.1 Branching

The branching defines a search tree. It is performed if the propagators have finished with their propagation without yielding a solution, but also without failing. The component that executes the branching is called the brancher. The brancher selects one variable that has not been set, and divides its domain into two or more non-empty partitions. Each partition is considered a choice. Every choice is a branch of the current state and constitutes a sub-tree of possible value assignments.

More than one variable can be selected by the brancher for creating a choice. It is then important that at least one of the domains has to be reduced, that all choices together must make up all possible combinations of values of the current state, and that no combination of values may occur in more than one state, for it to be a proper brancher.

The choices, branches, can then be selected to be the new domain of the variables. The reduction of the variables’ domains instigates some more propagation, further reducing the number of possible assignments to be considered in subsequent sub-trees.

Many branchers adhere to the first-fail principle when choosing variables to branch on. It means that the variables that are most likely to fail are selected when partitioning. By failing early, reaching a dead end fast, less of a sub-tree is needed to be explored.

There is also a commonly followed principle for the value selection, the best-first principle. The first choices of the partitioning should then contain those values that are most likely to be in a solution. If a solution exists within the sub-tree of the node, then selecting the best values increases the chance of finding it earlier.

However, if there are no solutions, or if all solutions are to be found, then ordering the choices by values makes no difference as the whole tree, and thus all the choices of each node, will be explored.

The search tree that the brancher defines is finite (decreasing), all assignments of values are considered (complete) and no assignment occurs more than once (non-overlapping). This is due to the properties of the choices.

3.6.2 Exploration

With the search tree defined, several different ways of searching it for a solution exist. Two common ways are presented below, one for the CSP and one for the COP.

Depth-First Exploration

One of the more commonly used methods of exploration for the CSP is depth-
first exploration (DFE). When exploring a new node it makes its first choice, the leftmost branch. If a node fails, i.e., a propagator is unable to satisfy its constraint, then DFE backtracks to the first previous node with untried choices, and explores the next one.

**Branch-and-Bound**

When solving a COP the branch-and-bound method (BB) is often a good choice. With the BB the exploration is done similar to the DFE but an added bound function, here called $bf$, is used.

For every new node visited, the $bf$ computes an upper bound on the maximum value of the optimisation function $f$ (or a lower bound if $f$ is to be minimised), given the current store. This bound is constrained to be greater than the “best” solution found as of yet. If the upper bound is less than the value of the best solution, then exploring the sub-tree of that node will not yield a better solution than the one already found.

### 3.7 Main Algorithm

At the centre of constraint programming is the main algorithm which coordinates the propagation and search. Its basic form is outlined in Algorithm 3.7.

The algorithm starts out (lines 9–16) by calling each propagator that might still propagate, one by one, and instructs them to perform propagation (waking them up). This is repeated until every propagator reports that it can not perform more propagation, i.e., it has reached fixpoint or subsumed itself.

Now (lines 18–25) the store is examined whether it has failed. In that case no more solution can be found from this store and the algorithm backtracks. Otherwise the algorithm examines whether all propagators are subsumed, if so then all constraints are satisfied and the store is a solution.

Next is the branching step (lines 27–31), where the brancher is called and returns a list of choices, in the form of propagators, that can be explored. If the list is empty, no more branching can be performed and the current store is a solution.

At last (lines 32–37) a recursive call is made for each choice returned by the brancher, exploring that sub-tree.

The initial call to the function is $\text{DFE}(S, P, P, b)$, where $S$ are the initial domains, non-pruned; $P$ are all the propagators of the problem and $b$ is the brancher.
1: DFE(Store S, Propagators P, Propagators Q, Brancher b)
2: // S is the current store, P are all non-subsumed propagators,
3: // Q are all propagators that might still propagate and
4: // b is the branching function
5: 6: // execute all propagators p that might still be able to constrict the store,
7: // wake all propagators in P that might be able to propagate more
8: // due to the restricting done by p and remove p from P if it is subsumed
9: while ∃p ∈ Q do
10: Q := Q \ {p}
11: S := p(S)
12: Q := Q ∪ wake(P)
13: if p is subsumed then
14: P := P \ {p}
15: end if
16: end while
17: 18: if ∃x ∈ S : D(x) = ∅ then
19: // S is a failed store, it does not contain a solution assignment
20: return
21: else if P = ∅ then
22: // if all propagators are subsumed then S is a solution
23: output S
24: return
25: end if
26: 27: ⟨C₁, . . . , Cₙ⟩ := b(P, S)
28: // C₁ to Cₙ are the propagator sets enforcing each choice
29: if n = 0 then
30: // if no propagators are returned by b then S is a solution
31: output S
32: else
33: // make a recursive call for each choice and store the solutions
34: for i := 1..n do
35: DFE(S, P, Cᵢ, b)
36: end for
37: end if
38: 39: return

Algorithm 1: Main Algorithm
4 nVector Constraint

Constraints are the core elements of constraint programming. There are a myriad of different constraints, a lot of them can be found in the global constraint catalogue. By utilising the diversity of the available constraints, problems can be modelled efficiently. One strength of constraint programming is that every constraint has its custom made propagator, which can enforce its constraint in an efficient way.

One such constraint is the nVector constraint. It is posted on a set of vectors and limits the number of distinct vectors in the set. It is a generalisation of the nValue constraint, which in turn is a generalisation of the AllDifferent constraint (see [3]). First the nValue constraint will be introduced and afterwards it will be shown how it can be generalised into the nVector constraint.

4.1 nValue Constraint

nValue is posted on an integer decision variable $N$ and an array of $n$ integer decision variables $X = [X_1, X_2, ..., X_n]$. The constraint limits the number of distinct values that the variables in $X$ take to be exactly $N$. Formally:

$$\text{nValue}(N, X) \iff N = |\{X_1, X_2, ..., X_n\}|$$

If $N = n$, the nValue is equivalent to the AllDifferent constraint. For example, if $N$ equals $n$ and $X$ is:

$$X_1 \mapsto \{1, 2, 3\}$$
$$X_2 \mapsto \{2, 3, 4\}$$
$$X_3 \mapsto \{4, 5\}$$
$$N \mapsto \{3\}$$

it can be seen that $X_1, X_2$ and $X_3$ indeed all have to be different from each other. In this case, $X_1 \mapsto 1, X_2 \mapsto 2, X_3 \mapsto 4$ is a solution.

Not only solving the nValue constraint, but even finding a lower bound on $N$ is NP-hard (see [3]). However, it is worth noticing that if the domains of the variables in $X$ are intervals, a lower bound on $N$ can be found in polynomial time (see [2]).

4.2 nVector Constraint

nVector is analogous to nValue, but instead of an array of integer decision variables $X$, it is posted on an array of $n$ vectors $V = [V_1, V_2, ..., V_n]$, where each vector $V_i$ consists of $m$ integer decision variables $V_i = [V^1_i, V^2_i, ..., V^m_i]$. Then the number of distinct vectors is limited to be exactly $N$. Two vectors are equal if and only if the vector elements are pair-wise equal. For example, $V_1 \mapsto [1, 0, 4]$ is equal to $V_2 \mapsto [1, 0, 4]$, while not equal to $V_3 \mapsto [1, 4, 0]$. Formally,

$$\text{nVector}(N, V) \iff N = |\{V_1, V_2, ..., V_m\}|$$

If \( m = 1 \), i.e., the length of the vectors is equal to one, then the \text{nVector} constraint will be identical to the \text{nValue} constraint. For example, given \( V \) (with \( m = 1 \)) and \( N \) as:

\[
\begin{align*}
V_1 &= [V^1_1 \mapsto \{1, 2, 3\}] \\
V_2 &= [V^1_1 \mapsto \{2, 3, 4\}] \\
V_3 &= [V^1_2 \mapsto \{4, 5\}] \\
N &= \{3\}
\end{align*}
\]

It can be seen that this problem is equivalent to \text{nValue}(\( N, [V^1_1, V^1_2, V^1_3] \)). In this way, every \text{nValue} constraint can be modelled as a \text{nVector} constraint, thus the latter is a generalisation of the former.

Another example can be seen in Figure 2. Letting \( V_1 = [V^1_1 \mapsto \{0\}, V^1_2 \mapsto \{0\}, V^1_3 \mapsto \{1\}, V^1_4 \mapsto \{1\}] \) be written as \( V_1 = [0, 0, 1, 1] \) allows describing a satisfying assignment for the constraint in Figure 2 as: \( N = 3, V_1 = [0, 0, 0, 1], V_2 = [0, 0, 0, 1], V_3 = [0, 0, 1, 0], V_4 = [0, 0, 0, 1], V_5 = [0, 1, 1, 0], V_6 = [0, 1, 1, 0]. \)

\[
\begin{align*}
V_1 &= [V^1_1 \mapsto \{0\}, \quad V^2_1 \mapsto \{0\}, \quad V^3_1 \mapsto \{0, 1\}, \quad V^1_4 \mapsto \{0, 1\}] \\
V_2 &= [V^2_1 \mapsto \{0, 1\}, \quad V^2_2 \mapsto \{0\}, \quad V^3_2 \mapsto \{0\}, \quad V^2_4 \mapsto \{1\}] \\
V_3 &= [V^3_1 \mapsto \{0\}, \quad V^3_2 \mapsto \{0\}, \quad V^3_3 \mapsto \{0, 1\}, \quad V^3_4 \mapsto \{0\}] \\
V_4 &= [V^4_1 \mapsto \{0, 1\}, \quad V^4_2 \mapsto \{0, 1\}, \quad V^4_3 \mapsto \{0\}, \quad V^4_4 \mapsto \{0\}] \\
V_5 &= [V^5_1 \mapsto \{0, 1\}, \quad V^5_2 \mapsto \{1\}, \quad V^5_3 \mapsto \{1\}, \quad V^5_4 \mapsto \{0, 1\}] \\
V_6 &= [V^6_1 \mapsto \{0\}, \quad V^6_2 \mapsto \{1\}, \quad V^6_3 \mapsto \{1\}, \quad V^6_4 \mapsto \{0, 1\}] \\
N &= \{1, 2, 3, 4, 5\}
\end{align*}
\]

Figure 2: Example of a \text{nVector} constraint

### 4.3 Reusing \text{nValue}

It is tempting to regard the domains of each vector as an integer variable and then reuse the \text{nValue} constraint. For example, since the domains of all vectors in the constraint in Figure 2 are binary, each vector could be treated as a binary string. The first element corresponds to the first digit, the second element corresponds to the second digit, and so on. \( V_1 = [0, 1, \{0, 1\}, \{0, 1\}] \) would represent the binary strings 0100, 0101, 0110 and 0111, which equals the decimal values of 4, 5, 6 and 7 respectively. In this way, the domain of the vector could be seen as \( V_1 = \{4, 5, 6, 7\} \). If the largest number in any vector’s domain is \( u \), then the vectors could be treated as numbers in base \( u \).

However, this does not work very well. First, the vectors can be very long and the resulting mapping from vector to integers can generate extremely large numbers. Secondly, if the vectors are treated as integers, it will often have a lot of gaps in the domains. The existing algorithms for \text{nValue} work on bounds consistency and treat the variable domains as intervals, and therefore only affect the endpoints of these intervals. This would result in very poor propagation.
4.4 Consistency

Achieving GAC on nValue is an NP-hard problem (see [10]) and thus GAC propagation for this cannot be achieved within reasonable time bounds. Using this knowledge one can conclude that the same is true for the nVector constraint. Every nValue problem can be modelled by a nVector constraint by setting the length of each vector to one (see section 4.2). Because of this complexity, a weaker kind of consistency will be the aim of the developed propagator.

4.5 Compatibility

When trying to enforce the nVector constraint, it is often useful to consider which vectors in \( V \) are compatible with each other. Two vectors \( V_1, V_2 \in V \) are said to be compatible if there are assignments (based on their current domains) such that they are equal. For example, \( V_1 = [0, 0, \{0, 1\}, \{0, 1\}] \) and \( V_2 = [(0, 1), 0, 0, 1] \), are compatible, since they both could be assigned to \( V_1 = V_2 = [0, 0, 0, 1] \).

With the concept of compatibility, one can define the compatibility graph, \( G \). The compatibility graph is a graph where each node represents a vector from \( V \) and there is an edge between two nodes if and only if the corresponding vectors can be equal. The compatibility graph for the vectors in the example of Figure 2 can be seen in Figure 3.

![Figure 3: Compatibility graph (G)](image)

This graph will be utilised in a number of ways and allows properties of the vectors to be calculated using graph-based techniques.

4.6 Lower Bound

Calculating a lower bound on the number of distinct vectors for a given array \( V \) is an important task in the propagation procedure. By retrieving this bound, one can prune values from \( N \). But achieving a lower bound is no easy task. Since nVector is a generalisation of nValue, and finding a lower bound on \( N \) for the latter is NP-hard (see [3]), this is true even for the former.

The true lower bound of \( V \) can be obtained by thinking of each unassigned vector \( v \in V \) as a set of all the vectors it could be. For example, the vector \([0, 0, 0, 1], [0, 0, 1, 0], [0, 0, 1, 1], [0, 1, 1, 1]\) can be viewed as the set \{\([0, 0, 0, 1], [0, 1, 0, 1], [0, 0, 1, 1], [0, 1, 1, 1]\)\}. If each vector is thought of in this way, then the lower bound is equal to the minimum hitting set (MHS) of all the vector sets.

A minimum hitting set is a minimum cardinality set such that there is at least one vector in common for each vector set and the minimum hitting set. An example can be seen in Figure 4 where a minimum hitting set of size two is
$V_1 = \{[0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 0], [0, 0, 1, 1]\}$

$V_2 = \{[0, 0, 0, 1], [1, 0, 0, 1]\}$

$V_3 = \{[0, 0, 0, 0], [0, 0, 1, 0]\}$

$MHS = \{[0, 0, 0, 0], [1, 0, 0, 1]\}$

Figure 4: A minimum hitting set

given. This is a minimal hitting set since there is no vector that is common to all of $V_1$, $V_2$ and $V_3$ and therefore a hitting set of size one can not be constructed.

The MHS is the minimum amount of haplotypes needed. However, finding a minimum hitting set is also an NP-hard problem (see [7]). Therefore, approximation techniques have to be employed. In this project, two different techniques have been tested: searching for a maximum clique and searching for a maximum independent set.

4.6.1 Maximum Clique

One way of finding a lower bound of the number of distinct vectors is by considering the incompatibility graph, $\bar{G}$. This graph is the inversion of the compatibility graph, i.e., there is an edge between each pair of nodes whose corresponding vectors are not compatible.

Figure 5: Incompatibility graph ($\bar{G}$) with a maximum clique

Given this graph a lower bound can be found by identifying the size of the largest clique (a sub-graph where each node in the sub-graph has an edge to each other node in the sub-graph). The cardinality of the maximum clique for a graph $G$ is denoted by $\omega(G)$. For example, in Figure 5 a maximum clique of size three is $\omega(\bar{G}) = |\{V_2, V_3, V_5\}| = 3$, and this is indeed a lower bound for this nVector instance.

However, finding the maximum clique is also NP-complete (see [7]), yet again an approximation technique has to be used.

Greedy: A naïve method (described in [15]) is to start out by selecting a node (from the incompatibility graph) which has the highest-degree (i.e., has the most edges connected to it). Afterwards, extending the clique by the highest degree node which is connected to each node already in the clique. This is repeated
Algorithm 1: Greedy Clique Construction

until no more nodes can be added. This will be a maximal clique (i.e., it is not a sub-graph of another clique) but might not be the maximum clique. The size of this clique can be used as a lower bound.

An example is shown in Figure 6. First $V_6$ is select (a) since it has the highest degree of four. Next, a tie occurs between $V_2$, $V_3$ and $V_4$. Arbitrarily, the lowest numbered node is selected (b). Lastly, another tie occurs between $V_3$ and $V_4$, and $V_3$ is picked and added to the clique (c). Since no more nodes are connected to all of the nodes currently in the clique ($\{V_2, V_3, V_6\}$, a maximal clique of size three has been constructed. Therefore, $N \geq 3$ can be assumed. Note that in this case it is actually a maximum clique that has been constructed, but this is not true in the general case.

The pseudocode can be seen in Algorithm 1. Since finding the highest degree node takes linear time in the number of nodes $n$ and edges $m$, and this is done within the while-loop at most $n$ times, the runtime complexity of CliqueLB is $\theta(n(e + n))$.

4.6.2 Maximum Independent Set

The duality of finding a maximum clique on the incompatibility graph is to find a maximum independent set on the compatibility graph. An independent set is a set of nodes where no two nodes in the set are connected by an edge. A maximum independent set (MIS) is such a set of maximum cardinality for a given graph. The cardinality of the MIS of a graph $G$ will be denoted by $\alpha(G)$.

Given a maximum clique on the incompatibility graph, the same nodes would form a maximum independent set for the graph, i.e., $\omega(G) = \alpha(G)$. Therefore finding a maximum independent set is an equally hard problem as finding a
maximum clique. Two approximations methods will be explored in this project: calculating a lower bound based on the number of edges and nodes (Turán) and greedily constructing a minimal independent set (Greedy).

**Turán and Favaron:** Bessiere et al. suggest using Turán’s proposed lower bound for the maximum independent set (see [3]). A lower bound can be calculated using the following equation:

\[
\alpha \geq \left\lceil \frac{n^2}{2m + n} \right\rceil
\]

where \(n\) is the number of nodes and \(m\) the number of edges. This bound is weaker than the clique bound but can easily be computed in constant time, making it possibly more suitable for repeatable computations, it is referred to as the Turán bound.

In the same paper, it is also mentioned that this bound has been improved by Favaron et al. (see [6]), giving a stricter formula:

\[
\alpha \geq \left\lceil \frac{2n - 2m}{\left\lceil \frac{2m}{n} \right\rceil + 1} \right\rceil
\]

This is referred to as the Favaron bound.

**Greedy:** Another method of finding a lower bound for \(\alpha\) is by greedily constructing an independent set. This is similar to the clique bound, but instead of finding the node with the highest degree, the lowest-degree node is found (as described in [14]). Then it is added to the resulting set, and all of its neighbours are removed as candidates. This is repeated until no more nodes are marked as candidates, and a maximal independent set is acquired (though not necessarily a maximum independent set). The size of this set can be used as a lower bound.

In Figure 7 the construction of a minimal independent set is shown. First node \(V_5\) has been selected (a), thus removing its neighbour \(V_6\). Next in (b) \(V_2\) is picked (since it is the only node with degree 1), removing \(V_1\). And lastly in (c), after \(V_3\) is picked, the resulting set consists of \(\{V_2, V_3, V_5\}\). The cardinality of this set can be used as a lower bound, i.e., \(N \geq 3\).

The pseudocode is shown in Algorithm 2. As finding the lowest degree node, as well as finding all neighbours for a given node both can be made in linear time with respect to the number of nodes \(n\) and edges \(m\), and the while-loop is repeated at most once for each node (since every iteration removes at least one node from the \(\text{Nodes} \) set), the total time complexity is \(\theta(n + m)\).

4.6.3 Other Methods

It can be shown that the size of a minimum clique cover or a minimum vertex cover can also be used as a lower bound. Since these problems are NP-hard and no suitable approximation algorithms were found (the ones found calculates on graphs with special properties), they have not been tested in this thesis.
Figure 7: Greedy minimal independent set construction

\begin{algorithm}
1: \textbf{MISLB}(Nodes } N, \text{ Edges } E) \\
2: \text{MISet} \leftarrow \emptyset \\
3: \textbf{while } N \neq \emptyset \textbf{ do} \\
4: \quad \text{NewNode} \leftarrow \text{FindLowestDegree}(N, E) \\
5: \quad \text{MISet} \leftarrow \text{MISet} \cup \{\text{NewNode}\} \\
6: \quad N \leftarrow \text{Nodes} \setminus (\{\text{NewNode}\} \cup \text{FindNeighbours}(\text{NewNode})) \\
7: \textbf{return } |\text{MISet}|
\end{algorithm}

\textbf{Algorithm 2: Greedy Independent Set Construction}

4.7 Upper Bound

Using various techniques, an upper bound can be estimated as well. However, since this project is focused on minimising the number of distinct vectors, propagation based on the upper bound is not of great interest. Therefore, the trivial upper bound of \( n \), i.e., the number of vectors will be sufficient.

4.8 Symmetries

When analysing constraints it is important to identify symmetries in the solutions. A symmetry is a permutation of the values assigned or of the variables such that solutions are preserved, i.e., by permuting a solution according to the symmetry, the result will also be a solution.

In the \text{nVector} constraint the most important symmetry is between vectors. Swapping the values of any two vectors of an assignment will still be satisfying the constraint. For example, given the following variables and domains:

\[
V_0 = [V_1^1 \mapsto \{0, 1\}, V_2^1 \mapsto \{0, 1\}],
V_1 = [V_1^2 \mapsto \{0, 1\}, V_2^2 \mapsto \{0, 1\}],
V_2 = [V_1^3 \mapsto \{0\}, V_2^3 \mapsto \{1\}],
V_3 = [V_1^4 \mapsto \{1\}, V_2^4 \mapsto \{0\}],
N \mapsto \{1, 2, 3, 4\}
\]

\( V_1 = [0, 1], V_2 = [1, 0], V_3 = [0, 1], V_4 = [1, 0], N = 2 \) satisfy \text{nVector}(V, N).

But swapping the assignments of \( V_1 \) and \( V_2 \), such that \( V_1 = [1, 0] \) and \( V_2 = [0, 1] \) will preserve the solution.

Other symmetries exist in permutation between the elements of the vectors (if all vectors are permuted the same way), see [10].
5 nVector Propagator

A propagator was developed for this project using the techniques described in section 4. The main idea behind the propagator is to create an incremental propagator that also can help guide the searching by maintaining an internal data structure, the compatibility graph.

An incremental propagator uses a state that is saved between invocations of the propagation algorithm for the constraint. For the nVector propagator, this state consists of the compatibility graph (see section 4.5). By analysing the changes that have happened since the last invocation of the propagator, the graph does not need to be rebuilt every iteration, instead only the affected edges can be modified. The propagator designed has 3 arguments, \( \text{nVector}(N,V,D) \). \( N \) is the number of distinct vectors of \( V \) (as described in section 4). An additional argument has been added: \( D \) - the difference matrix, explained next.

5.1 Difference Matrix

The difference matrix is used to enable communication between the propagator and the brancher. It is a \( n \times n \) matrix, where \( n \) is the number of vectors in \( V \). Each position \( d \in D(i,j) \) in the matrix can either be Same, Different or Undecided.

The first meaning, Same, states that the two vectors \( V_i \) and \( V_j \) are assigned to be equal (implying that there are no more unassigned variables in these vectors). If the position has the value Different, it means that the two vectors differs at minimum one position, so they are guaranteed to be different when completely assigned. If the value is Undecided then nothing can yet be concluded.

In this section the example presented in Figure 8 will be used to illustrate the effects of the different parts of the propagator. It will be assumed that the following constraint has been posted:

\[ \text{nVector}(N,V,D) \]

5.2 Divide and Conquer

An effective way of solving large problems is by using divide and conquer. This is the technique of splitting up larger problems into smaller parts and then recombining their solutions. In the nVector propagator, this consists of dividing the \( V \) array into smaller arrays and posting new nVector constraints.

For the divide and conquer approach to be usable, it must be certain that the smaller problems are completely separate and do not interfere with each other. This structure is found in the nVector propagator by observing the compatibility graph. In Figure 8 the compatibility graph for the example given in Figure 8 is shown.

The main idea is to consider the connected components. A connected component is a sub-graph where there is a path between every pair of nodes. For example, in Figure 8 there are two connected components, \( \{V_1,V_4,V_5,V_6\} \) and \( \{V_2,V_3\} \). Note that since there exist no edges between any two nodes (\( V_i,V_j \)) that reside in different components, there is no assignment to the vectors such that \( V_i = V_j \). Therefore, since vectors in different connected components can not interfere with each other, the number of distinct vectors in the complete
$$V_1 = [V_1^1 \mapsto \{0\}, \ V_2^2 \mapsto \{0\}, \ V_3^3 \mapsto \{0, 1\}, \ V_4^4 \mapsto \{0, 1\}]$$

$$V_2 = [V_1^1 \mapsto \{0\}, \ V_2^2 \mapsto \{0\}, \ V_3^3 \mapsto \{0\}, \ V_4^4 \mapsto \{1\}]$$

$$V_3 = [V_1^1 \mapsto \{0\}, \ V_2^2 \mapsto \{0\}, \ V_3^3 \mapsto \{0, 1\}, \ V_4^4 \mapsto \{0\}]$$

$$V_4 = [V_1^1 \mapsto \{0, 1\}, \ V_2^2 \mapsto \{0, 1\}, \ V_3^3 \mapsto \{0\}, \ V_4^4 \mapsto \{0\}]$$

$$V_5 = [V_1^1 \mapsto \{0, 1\}, \ V_2^2 \mapsto \{1\}, \ V_3^3 \mapsto \{1\}, \ V_4^4 \mapsto \{0, 1\}]$$

$$V_6 = [V_1^1 \mapsto \{0\}, \ V_2^2 \mapsto \{1\}, \ V_3^3 \mapsto \{1\}, \ V_4^4 \mapsto \{0, 1\}]$$

$N \mapsto \{1, 2, 3, 4, 5\}$

$D$ is set according to the following table (where $D(i, j)$ equals the cell at row $i$ and column $j$):

<table>
<thead>
<tr>
<th>$D$</th>
<th>$V_1$</th>
<th>$V_2$</th>
<th>$V_3$</th>
<th>$V_4$</th>
<th>$V_5$</th>
<th>$V_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_1$</td>
<td>Same</td>
<td>Undecided</td>
<td>Undecided</td>
<td>Undecided</td>
<td>Different</td>
<td>Different</td>
</tr>
<tr>
<td>$V_2$</td>
<td>Undecided</td>
<td>Same</td>
<td>Different</td>
<td>Different</td>
<td>Different</td>
<td>Different</td>
</tr>
<tr>
<td>$V_3$</td>
<td>Undecided</td>
<td>Different</td>
<td>Same</td>
<td>Undecided</td>
<td>Different</td>
<td>Different</td>
</tr>
<tr>
<td>$V_4$</td>
<td>Undecided</td>
<td>Different</td>
<td>Undecided</td>
<td>Same</td>
<td>Different</td>
<td>Different</td>
</tr>
<tr>
<td>$V_5$</td>
<td>Different</td>
<td>Different</td>
<td>Different</td>
<td>Same</td>
<td>Undecided</td>
<td>Same</td>
</tr>
<tr>
<td>$V_6$</td>
<td>Different</td>
<td>Different</td>
<td>Different</td>
<td>Undecided</td>
<td>Same</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8: Running example of the nVector propagator

Figure 9: Before split, one compatibility graph

graph (actually in the array of the vectors represented by the nodes in the graph) is equal to the sum of the number of distinct vectors in each sub-graph.

In fact, the number of distinct vectors in the complete graph will be equal to the number of connected components in the final graph (when all vectors have been assigned).

By using this knowledge, whenever the compatibility graph is divided into more than one component, the constraint can be replaced by a series of new, smaller constraints - one for each connected component. For example, the constraint represented by Figure 8 can be replaced by two new nVector constraints together with a constraint on $N$ to ensure the sum of results equals the original $N$.

$$nVector(N_1, [V_1, V_2, V_3, V_4], D_1)$$

$$nVector(N_2, [V_5, V_6], D_2)$$

$$N_1 + N_2 = N$$
Algorithm 3: Constructing Connected Components

\begin{algorithm}
\begin{algorithmic}
1: \textbf{CC}(Nodes N, Edges E)
2: \textbf{CurrentComponent} ← 0
3: \textbf{while} \( N \neq \emptyset \) \textbf{do}
4: \hspace{1em} \textbf{NewNode} ← \textbf{RandomNode}(N)
5: \hspace{1em} \textbf{MarkNode}(\textbf{NewNode}, \textbf{CurrentComponent})
6: \hspace{1em} \textbf{Reachable} ← \textbf{FindReachable}(\textbf{NewNode}, N, E)
7: \hspace{1em} \textbf{MarkNodes}(\textbf{Reachable}, \textbf{CurrentComponent})
8: \hspace{1em} \textbf{N} ← \textbf{Nodes} \setminus (\{\textbf{NewNode}\} \cup \textbf{Reachable})
9: \hspace{1em} \textbf{CurrentComponent} ← \textbf{CurrentComponent} + 1
\end{algorithmic}
\end{algorithm}

\begin{itemize}
\item \( N_1 \) and \( N_2 \) are two new integer decision variables introduced to keep track of the numbers of distinct vectors of the new constraints. The sum of these has to be equal to the original \( N \), and thus guarantees that a solution for the new constraints is indeed a solution for the original \texttt{NVECT} constraint. \( D_1 \) and \( D_2 \) are slices of the original \( D \) matrix corresponding to the correct rows and columns. The two new compatibility graphs can be seen in Figures 10 and 11.
\end{itemize}

\begin{figure}[h]
\centering
\includegraphics[width=0.2\textwidth]{first_graph.png}
\caption{After split, first compatibility graph}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.2\textwidth]{second_graph.png}
\caption{After split, second compatibility graph}
\end{figure}

5.3 Constructing Connected Components

To identify the actual connected components of a certain state, Algorithm 3 is used. It starts out by picking an arbitrary node, doing a DFS (depth-first search) to identify every node reachable from the picked node. During the DFS every node is marked as belonging to the same connected component. When the search is over, another unmarked node is picked and another DFS is started. This is repeated until every node is marked.

The time complexity of this procedure is calculated by analysing the searches. Every edge will be followed at most twice (once from each direction), and each node will be marked once. Thus the running time will be proportional to the number of edges, \( m \), and nodes, \( n \), i.e., \( \theta(n + m) \).
5.4 Updates

The most important part of a propagator is to define how and when propagation should be done. Assuming the propagator is at fixpoint, this section will discuss what kind of update could open up a possibility for propagation to be done.

The propagator is invoked any time the domain of either the \( N \) variable or any vector \( v \in V \) is changed. Note that the only change that can occur is that values are removed from the domain. In the propagator, updates on \( N \) and updates of a \( v \in V \) are treated as two separate cases.

5.4.1 Updates on \( N \)

When the domain of \( N \) has changed, it is analysed to see what kind of change has happened. In this propagator, changes on \( N \) lead to propagation only if \( N \) is assigned a value. The other possible update would be a removal of one (or more) values from \( N \)’s domain, reducing it to a size greater than one. No relevant propagation possibilities have been found using these kinds of reductions of the domain.

When \( N \) is fixed to a single value, the assigned value is checked and propagation is performed according to the following rules:

- \( N = 1 \) If \( N \) is set to one, that means all vectors have to be equal. Therefore the following can be enforced:

  \[
  V_1 = V_2 = V_3 = \ldots = V_m
  \]

  For example, if the state shown in Figure 11 would have \( N = 1 \), then \( V_5 = V_6 \), i.e., \( V_5^5 = V_6^6, V_5^6 = V_2^6, V_3^5 = V_3^6 \) and \( V_4^5 = V_4^6 \) can be posted. This would yield the instant propagation of \( V_5^5 = 0 \).

- \( N = 2 \) If the number of different vectors should be exactly two, it means the vectors in \( V \) can be divided into two groups, where each vector in a group is equal. This can be utilised by trying to form these two groups.

  Given a pair of unequal vectors (\( V_i, V_j \)), the remaining vectors have to be equal to one of them (or both). Each vector \( v \in V, v \notin \{V_i, V_j\} \) is checked. If \( v \neq V_i \), then \( v \) has to be equal \( V_j \) and thus \( v = V_j \) is enforced. If \( v \neq V_j \), then \( v = V_i \) is enforced. If \( v \) could be equal to both \( V_i \) and \( V_j \) then no constraint is posted.

Figure 12: Compatibility graph of second sub-problem after posting \( V_5 = V_6 \)

After this has been posted, the propagator can subsume itself since it is satisfied if and only if the posted constraint is satisfied.

Since the only checking required is to see whether \( N \) equals 1 or not, enforcing this takes worst-time equal to posting the \( m \) equality constraints, i.e., \( (\theta(m)) \).
In this propagator no special care is taken in picking $V_i$ and $V_j$ but the first two unequal vectors are chosen. This could perhaps be improved by selecting the vectors in a more clever fashion.

Take as an example the first sub-problem (seen in Figure 10) and assume $N = 2$. $V_2$ and $V_3$ are two nodes that can not be the same. These two nodes are picked, and each other node has to be equal to at least one of them, otherwise $N > 2$. Therefore, $V_4$ has to be equal to $V_3$ since it is different from $V_2$, while $V_1$ can not be constrained since it is compatible with both $V_2$ and $V_3$. The result can be seen in Figure 13.

![Figure 13: After split, first compatibility graph](image)

Since each edge might have to be checked to find an incompatible pair of nodes (actually, the first non-existing edge is searched for) as well as each node has to be assigned to a group, this will take linear time, with respect to the number of nodes, $n$, and the number of edges $m$, to compute. Therefore, enforcing this rule takes $\theta(n + m)$ time in the worst case.

### 5.4.2 Updates on $V$

If an update occurs on a vector $v \in V$, the compatibility graph is updated with respect to the node corresponding to $v$. That is, every edge connected to $v$ is checked if it is still there. Note that removing values from the domain of $v$ could only lead to edges being removed from the graph. New edges can never be added.

If the update on $v$ did not lead to an update of the graph, no propagation is made and nothing happens. However, if one or more edges are removed, the following actions take place:

**Rebuild Connected Components** Before the domain change of $V$, the whole graph used to be a single connected component, but now it might have to be split into several components. The changes are analysed such that if it is the case that the graph now contains $x > 1$ components, then $x$ new nVector constraints are posted and the current propagator is subsumed.

In the first sub-problem from Section 5.2 (see Figure 10), if $V_4^1 = 1$, then the new compatibility graph will be as seen in Figure 14. Here the new connected components will be $\{V_3, V_4\}$ and $\{V_1, V_2\}$ and therefore two new constraints can be posted (replacing the current one).

```
NVector(N_1, [V_3, V_4], D_1)
NVector(N_2, [V_1, V_2], D_2)
```
\[ N_1 + N_2 = N \]

Which is same to the way divide and conquer was described above. Reconstructing the connected components takes time linearly proportional to the number of edges and nodes (see Section 5.2). Thus enforcing this rule takes \( \theta(n + e) \) time.

![Figure 14: After split, third compatibility graph](image)

**Lower Bound** Removing an edge could lead to that being a new lower bound on the number of distinct vectors. A new estimation is calculated and posted as a constraint on \( N \). For example, the graph seen in Figure 15 has a lower bound of 2 (using the MIS method). If the edge between \( V_2 \) and \( V_3 \), as well as the edge between \( V_2 \) and \( V_4 \), are removed, the same graph as seen in the previous section (see Figure 9) is reached. Calculating on this graph gives the lower bound of 3 (see Figure 7).

![Figure 15: A compatibility graph with lower bound 2](image)

Several methods has been presented for calculating a lower bound (see Section 4.6), and the methods have different time complexities. The Turán and Favaron methods both have a constant time complexity, while the greedy maximum independent set and clique bound methods have a linear complexity (see Section 4.6). Thus enforcing this rule has a worst case runtime of \( \theta(n + m) \) in the number of nodes \( n \) and edges \( m \).

### 5.5 Consistency

With the propagator described in this section, no defined consistency is reached after propagation. Even achieving bound consistency is NP-hard. This can be seen by considering the bounds on \( N \).

Bounds consistency states that each extremal value in the domain of \( N \) should have support within the bounds of the variables in \( V \). However, to guarantee that the minimum value in the domain of \( N \) has support, it has to be equal to the lower bound of the number of distinct vectors, which has been
shown to be NP-hard (see section 4.6). Since just the problem of calculating this value is NP-hard, achieving bounds consistency becomes intractable.

5.6 Guiding Search

One part of constraint programming is to guide the search in a good direction, i.e., making good guesses on what value assignment of a variable that will most likely lead to a solution. In this project, search has been guided by the NVECTOR propagator by sharing the difference matrix. This will be explained in detail in the next section.
6 Modelling HIPP in CP

The most important, and arguably hardest, part of constraint programming is the modelling of the problem. By choosing a different model, a previously, in practical time, unsolvable problem can be solved in reasonable time. All through this project, the plan has been to create a model of the HIPP problem using the nVector constraint (see Section 4). Given an efficient propagator for this constraint, the problem can be elegantly modelled.

6.1 The Problem

In this model the following is used to denote the different parts of the problem. Given is a set of genotypes $G$ which is to be explained by a set of haplotypes $H$. $G_i[k]$ denotes the value at the $k^{th}$ position of the $i^{th}$ genotype. $H_{i1}$ and $H_{i2}$ are the two haplotype identifiers explaining $G_i$.

6.2 Optimisation

The HIPP problem is an optimisation problem since the goal is to minimise $N$. There are several different methods of modelling optimisation problems using constraint programming. In this project two different techniques will be used. First ordinary branch-and-bound (see Section 3.6.2) will be tried. Secondly, by setting $N$ to its lowest possible value, the task is converted into a satisfaction problem (similar to what is done in [10]). If it would fail, $N$ is increased by one and the process is repeated. In this way, the problem can be modelled as a satisfaction problem and the minimisation aspect can be disregarded.

6.3 Preprocessing

The data given for the HIPP problem is formatted as strings of zeros, ones and twos. Remember that the zeros and ones are representing the homozygous wild and mutant types while the twos represents heterozygous SNPs. By preprocessing these strings, the size of the problem instances can be reduced, sometimes significantly, and also making the modelling of the problem easier. Here follows a description of a few preprocessing techniques used for this project.

6.3.1 Switching 1 and 2

A simple change that is made with the input data is to reverse the roles of the positions with value 1 and the positions with value 2. Originally the homozygous (where both explaining haplotypes shared the same value) positions were denoted by 0 and 1, while the heterozygous positions (where one of the explaining haplotypes had a 0 and the other a 1) were denoted by 2.

But by changing such that the homozygous position is denoted by 0 and 2, the constraint for the heterozygous positions can be more elegantly calculated. If these numbers are used instead, two explaining haplotypes will add up to the genotype. For example, given the genotype $g = [0, 2, 2, 1]$, and an explaining pair of haplotypes $h_1 = [0, 0, 1, 1]$ and $h_2 = [0, 1, 0, 1]$. Following this transformation, the genotype would be $[0, 1, 1, 2]$. Adding up the haplotypes element-wise yields $h_1 + h_2 = [0 + 0, 0 + 1, 1 + 0, 1 + 1] = [0, 1, 1, 2]$, which is exactly the genotype after transformation.
The rest of this thesis is written using the original notation, i.e., the data is not preprocessed using this technique before being presented.

6.3.2 Removing Singles

Some of the SNPs can be removed before starting to solve the problem if they all share the same value. If every genotype \( g \in G \) shares the same homozygous value at a certain position, then this position can be removed. After a solution is found a complete solution can be constructed.

For example given genotypes \([0, 0, 1], [2, 2, 1], [1, 0, 1]\), seeing that the last element always is one, it can be removed and afterwards restored to every solution, thus giving the easier problem \([0, 0], [2, 2], [1, 0]\).

6.4 Model

To create a CP model for a given instance of the HIPP problem, a pair of decision variable arrays are created for each genotype. Each element in each array of the pair is given the values 0, 1 as possible assignments if the genotype is homozygous in that position, otherwise it is given either 0 or 1 as the value, depending on whether it is corresponding to the wild or mutant allele. Call the collection of all the haplotype arrays \( H \).

The haplotype inference problem of finding \( 2n \) haplotypes explaining \( n \) genotypes of length \( m \) is modelled by creating an array, \( V \), of \( 2n \) vectors of size equal to the length of the genotypes. Afterwards, to each genotype \((i)\) a pair of vectors \((V_{2i}, V_{2i+1})\) are assigned, these vectors are then required to explain this particular genotype.

A decision variable \( N \) is also required, which represents the number of distinct haplotypes in the vector \( V \). Its initial domain is set as follows. The lower bound is set to the value calculated using one of the techniques presented in Section 4.6 (different version of the model uses different techniques). The upper bound is set to the trivial upper bound of \( 2n \) (it was shown that a solution can always be found using \( 2n \) vectors, see Section 2.1). The goal is to minimise this number.

6.4.1 Variables

- \( N = [1..2n] \)
- \( V = [V_1, V_2, ..., V_{2n}], \) where each \( V_i = [V^1_i, V^2_i, ..., V^m_i] \)

6.4.2 Constraints

- For each \( G_i \), at each position \( k \): \( V^1_i[k] + V^2_i[k] = G_i[k] \)
  This ensures that each pair of haplotypes explaining the same heterozygous genotype are different. For homozygous genotypes (containing no heterozygous sites) will be identical.
- \( \text{nVector}(N, V) \)
  The main constraint that ensures the number of distinct vectors is equal to \( N \).
6.4.3 Symmetry Breaking

As noted in the previous section, a symmetry in the `nVector` constraint is that the vectors can be permuted while preserving (non-)solutions. However, in general it can be quite hard to ensure that only one possible permutation can be obtained. However, the pairs of haplotypes explaining the same genotype are special. At the beginning of execution, each haplotype is identical to the other one in the same pair. Therefore it is easy to break the symmetry in this pair by assigning the first heterozygous site of the pair such that the first, if any, haplotype has a zero in that position and the second haplotype has a one, similar to what Lynce et al. have done (see [15]).

For example, if $V_1 = [1, \{0, 1\}, 0, \{0, 1\}]$ and $V_2 = [1, \{0, 1\}, 0, \{0, 1\}]$ both explain the same genotype (i.e., $G = [1, 2, 0, 2]$). Both $V_1 = [1, 0, 0, 0], V_2 = [1, 1, 0, 1], V_1 = [1, 1, 0, 1], V_2 = [1, 0, 0, 0]$ are valid assignments. However, this symmetry can be broken by requiring the first heterozygous element of $V_1$ to be smaller than the corresponding element of $V_2$. In this case it posts the constraint that the second element of $V_1$ equals 0 and the second element of $V_2$ equals 1. Thus the new domains are: $V_1 = [1, 0, 0, \{0, 1\}]$ and $V_2 = [1, 1, 0, \{0, 1\}]$.

6.5 Branching Heuristic

One of the techniques this project set out to try was to share a data structure between a propagator and a brancher to enable efficient branching. In this model it is the compatibility graph (implemented as the difference matrix) described in Section 4.5).

When picking a branching heuristic, two different choices has to be considered. The variable selection as well as the value selection. The first one refers to which variable should be chosen, and the latter refers to what value should be tried.

6.5.1 Variable Selection

First-fail refers to the variable selection heuristic which advises that the variable which will most likely lead to failure should be chosen first. Commonly, this means that the variable with the fewest choices (i.e., smallest domain) is picked.

For this brancher, the first-fail principle is adhered to by finding a vector which is currently compatible to the highest number of other vectors, this is equal to finding a node in the compatibility graph which has the most incident edges. The idea behind this is that when assigning values to the variables of a highly connected node (having a lot of incident edges), it is highly unlikely that all edges would remain. Since the original high count of edges causes underestimations on the lower bound, sorting these nodes out first will find quicker if the lower bound is achievable. This method is chosen since in this project what is sought for is specifically the minimal solution.

The actual method used is to find a node with highest degree, then an element of the respecting vector has to be chosen. Each element of the vector is checked in turn and for each choice (in this case either setting it to 0 or 1) is analysed to see how many edges would be removed if the value would be chosen.

For example, analysing the state seen in Figure 8 it can be seen in the compatibility graph (Figure 9) that the node $V_1$ has three incident edges which
is more than for any other node. Then by analysing the elements, it is seen that $V_{3}^1$ and $V_{4}^1$ are the only elements which are not assigned. And setting the the third element to a 0 (1) removes 0 (2) edges, while setting the fourth to a 0 (1) removes 1 (2) edges. Therefore either the third element or the fourth element is chosen.

### 6.5.2 Value Selection

A common method when deciding on the value ordering (what value to try next in every step of the search) is to assign the value which has the highest probability to be a part a solution (see [18]), this is called \textit{best-first}. By analysing the state as described above, the brancher can calculate which variable ($V_{j}^i$) should be assigned what value ($val$) to remove the most number of edges. Since removing edges makes the problem more likely to fail, this assignment is avoided and the opposite choice is made, i.e., $V_{j}^i \neq val$ is posted.
7 Results

The purpose of the project was to design and develop a propagator for the \textit{nVector} constraint and use it to model the HIPP problem. To measure the efficiency of the result, the model is compared with the currently best solver for the HIPP problem, the RPoly model (see \cite{10}).

7.1 Implementation

The HIPP model was implemented using C++ and the constraint programming library Gecode\cite{4}. A custom propagator for the \textit{nVector} constraint was implemented as well as a custom brancher.

The propagator developed consists of a naive base which doesn’t do any propagation but merely checks when all of the variables have been assigned if it fulfills the condition of being the right number of distinct vectors. This propagator was extended with three parts: lower bound propagation on $N$, posting according to patterns based on updates on $N$, and splitting into sub-problems based on updates on $V$. These three extensions are referred to as LOWER\_BOUND, PATTERNS and SPLIT.

7.2 Test Cases

The data used was a combination of real-life instances as well as generated ones. The actual data used are the ABCD, ACE, IBD and NONUNIF (excluded from the lower bound measurements) data sets from \cite{10}. In total it consisted of 300 (260 without NONUNIF) different problem instances.

The platform used was an GNU/Linux kernel 2.6.37 on a PC with 3.0GHz CPU (4 cores), 8096 KB of L2 cache, and 3 GB of RAM memory. The Gecode version is 3.7.3. The RPoly program can be downloaded from Ana Graça’s homepage\cite{5}.

7.3 Lower Bounds

An investigation was made to see the efficiency of different ways of estimating the lower bound. Only the ABCD, ACE and IBD data sets were analysed this way.

In Figure 16 and 17 the resulting lower bounds can be seen. The greedy maximum independent set (MIS) and greedy clique partition (CLIQUE) always calculate the same bound. This hints at the greedy methods chosen might be a poor choice, they are perhaps too similar working on the graphs which are inversions of each other and where one picks the highest degree node (CLIQUE) the other one looks for the lowest degree node (MIS). The Turán (TURAN) and Favaron (FAVARON) bounds does also calculate the same lower bound for each instance. The CONS bound refers to a maximum independent set calculated using a constraint solver that creates an actual maximum independent set.

CONS was always at least as good as the greedy approach, which should be the case, and there were only a few instances (30) were it was strictly better. The greedy approaches were always at least as good as the constant time bounds, and

\footnote{http://www.gecode.org/}

\footnote{http://sat.inesc-id.pt/~assg/rpoly/}
they were only equal in very few instances (15). This suggests that disregarding
time issues, the constraint bound is to be preferred, followed by the greedy
graph approaches.

Figure 16: Lower Bound (a)

Figure 17: Lower Bound (b)
The runtimes for the different methods can be seen in Figure 18. Here only the MIS and FAVARON runtimes are presented for clarity. The MIS method was always faster than the CLIQUE method (and they both achieved the same bound), and FAVARON and TURAN were more or less equally fast. When inspecting the runtime of the different approximations, it can be seen that the constraint bound takes in some cases a very long time. Sometimes as much as $10^8$ seconds. This is clearly not time worth investing in calculating the bound as in many cases the bound is the same as a cheaper approach or just increases it by one.

More interestingly is comparing the MIS and TURAN methods. The former takes on average along the lines of 100 milliseconds which is hardly much more than the roughly 10 milliseconds spent by the latter. This is a very small investment for the vast improvement (an average of 4) of the lower bound. Due to this fact, for the rest of the experiments, the CLIQUE method was chosen to calculate the lower bound.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure18}
\caption{Runtimes for Lower Bound}
\end{figure}

### 7.4 Optimisation Technique

As discussed in section 6.2, two different techniques were tried for implementing the optimisation part of the model: Branch-and-bound and updating the lower bound on $N$ incrementally until a solution were found. Several test runs were made with both techniques and the ordinary branch-and-bound outperformed the incremental approach by 5% on average. However, in two cases the incremental approach were much faster, finishing in 10% and 50% of the runtime of the branch-and-bound version. Since the branch-and-bound version were on average faster, it was chosen as the technique of the final model.
7.5 Runtimes

The 300 problem instances were run with a timeout set to 60 seconds. First the naïve model was tried, then with each extension added separately (to see the improvement by each part). Also the full constraint programming model (as presented in section 6) was compared with the RPoly solver (see [10]).

<table>
<thead>
<tr>
<th>Model</th>
<th>Solved</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve</td>
<td>7</td>
<td>293</td>
</tr>
<tr>
<td>LOWER_BOUND</td>
<td>10</td>
<td>290</td>
</tr>
<tr>
<td>PATTERNS</td>
<td>7</td>
<td>293</td>
</tr>
<tr>
<td>SPLIT</td>
<td>56</td>
<td>244</td>
</tr>
<tr>
<td>ALL</td>
<td>74</td>
<td>223</td>
</tr>
</tbody>
</table>

Table 1: Effects of Extensions

7.5.1 Effects of extensions

In Table 1 it can be seen that the extension with greatest effects is to split the problem into sub-problems. It makes almost 50 more instances (16% of total instances) solvable compared to the naïve model. The propagation of the lower bound on $N$ only makes three more instances (1%) finish before the timeout, while the patterns extensions has no impact.

Interestingly, the PATTERNS and LOWER_BOUND extensions have greater effect when combined with SPLIT. When all extensions are combined a total of 74 instances are solved, which is more than the sum of the improvements. This shows that the less efficient extensions still are relevant in the complete model.

The reason could be that when breaking down the problem into sub-problems, more time is spent on solving small problems, where the PATTERNS and LOWER_BOUND extensions actually have a chance on incurring propagation. For example PATTERNS is only activated when $N$ is equal to 1 or 2, which seldom happens in big instances.

<table>
<thead>
<tr>
<th></th>
<th>CP</th>
<th>Rpoly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solved instances</td>
<td>74</td>
<td>223</td>
</tr>
<tr>
<td>Timeout</td>
<td>226</td>
<td>77</td>
</tr>
</tbody>
</table>

Table 2: Solved vs Timeouts

7.5.2 Comparison with RPoly

Table 2 shows that the RPoly software is more successful than the CP-based approach (see Table 2). In the following statistics, the instances where both solvers timed out have been excluded. In Figure 19 the runtimes of the two programs are compared. The RPoly solver is in every instance at least as fast as the CP solver presented in this thesis (disregarding differences less than 100 ms).

However, when studying the instances where both tools do time out (see Table 3) it can be seen that in all instances the CP solver has found a better
solution than the RPoly solver. It is not certain whether it is optimal or not, i.e., has the minimum amount of haplotypes, but it is a solution in the sense that each genotype is explained. This means that the constraint programming approach could be advantageous in certain instances.

<table>
<thead>
<tr>
<th>Program</th>
<th>Best solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP</td>
<td>76</td>
</tr>
<tr>
<td>RPoly</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3: Best solution at timeout

These particular instances, where none of the tools found an optimal solution before timeout, were ran once again, but with a timeout of 1000 seconds. The result was that five more instances could be solved by the RPoly solver, while the CP solver could not find any more optimal solution. And in the remaining 71 cases, the current solutions of the CP solver at timeout were all better (smaller) than the ones of the RPoly.

![Figure 19: Runtimes](image)

7.6 Discussion and Conclusion

The constraint programming model implemented in this project can be seen to be much slower than the RPoly model. However, in all instances where both solvers timeout the CP approach manages to find a better solution than the RPoly method in the same amount of time. Often the CP approach presented here can actually find an optimal solution quickly, but proving it optimal takes a lot of time.
Solving the HIPP problem using constraint programming was tried with hope that the global knowledge of the \text{nVector} constraint could be exploited in a way that the existing SAT-solver (RPoly) can not. However, it seems that this advantage was not great enough to compensate for the fact that the HIPP problem, since the haplotypes consists of ones and zeros, is very suitable to be solved with a boolean approach.

7.7 Future Work

There are a few ideas that could be tried to improve the efficiency of the constraint model. These were not implemented in this project due to time constraints.

7.7.1 Limiting Variables

One idea is to avoid creating decision variables. Each pair of haplotypes that is explaining the same genotype will be linked. At the homozygous sites of the genotype, the respective haplotype sites will be defined from start. At the heterozygous sites, the haplotypes will be different. This means that one of the haplotypes could be assigned to use the same variables as the first one, with the exception that on the heterozygous sites, it should view the inverted value of the variable, i.e., a one should be a zero and vice versa. This would reduce the number of decision variables that are undecided of the model by half.

7.7.2 Bridges

Some time is spent in constructing the connected components, and every time an edge is removed, the algorithm is restarted from the beginning, because it is non-incremental. There is a concept called bridges (see [11]) which identifies the edges that cannot be removed without splitting a connected component. By keeping track of the bridges of the sub-problems, the reconstruction algorithm could maybe be avoided in many instances.

7.7.3 AtLeastNVector and AtMostNVector

In this project, a general propagator for \text{nVector} was designed and implemented. However, when studying the design of a \text{nValue} propagator, it is common to split it up into two smaller problems consisting of \text{AtMostNValues} and \text{AtLeastNValues} to reduce the complexity with loss of global propagation. Perhaps the same can be done to for \text{nVector}, i.e., splitting it into \text{AtLeastNVector} and \text{AtMostNVector} and be able to use some properties of the sub-problems.
References


