Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Master Thesis

European Master in Computational Logics

Constraint-based modeling of Minimum Set Covering: Application to Species Differentation

David Buezas (32411)

Lisboa
(2010)

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Master Thesis

Constraint-based modeling of Minimum Set Covering: Application to Species Differentation

David Buezas (32411)

Superviser: Dr. Pedro Barahona

Work presented in the context of the European Master in Computational Logics, as partial requisit for the graduation as Master in Computational Logics.

Lisboa
(2010)

To Anna

Acknowledgements

I am very grateful to my supervisor Dr. Pedro Barahona [1] who, in spite of his tight schedule, found time to guide and help me with his agile mind to overcome the obstacles throughout the whole development of the present dissertation. Also I am in debt with João Almeida [2] who first proposed the Species Differentiation Problem and provided clarification about the biological aspects with which this thesis is concerned. My Master studies were supported by the European Master's Program in Computational Logic (EMCL) without which it would have been impossible to study in Europe.

Finally I must thank my girlfriend, family and friends for their continuous support, and my colleague and friend Luciano without whom it would have been hard to keep focus on the work.

[1]CENTRIA-Centro de Inteligência Artificial, Dep. de Informática, Faculdade de Ciências e Tecnologia / Universidade Nova de Lisboa

[2]CREM-Centro de Recursos Microbianos, Fac. de Ciências e Tecnologia / Universidade Nova de Lisboa

Abstract

A large number of species cannot be distinguished via standard non genetic analysis in the lab. In this dissertation I address the problem of finding all minimum sets of restriction enzymes that can be used to unequivocally identify the species of a yeast specimen by analyzing the size of digested DNA fragments in gel electrophoresis experiments. The problem is first mapped into set covering and then solved using various Constraint Programming techniques. One of the models for solving minimum set covering proved to be very efficient in finding the size of a minimum cover but was inapplicable to find all solutions due to the existence of symmetries. Many symmetry breaking algorithms were developed and tested for it. Hoping to get an efficient model suitable for both tasks also the global constraint involved on it was partially implemented in the CaSPER Constraint Solver, together with the most promising symmetry breaking algorithm. Eventually the best solution was obtained by combining two different constraint-based models, one to find the size of a minimum solution and the other to find all minimal solutions.

Keywords: Minimum Set Covering, Enzymes, Species Differentiation, Constraints

Contents

1 Motivation **1**

2 Mapping the Species Differentiation Problem into a Minimum Set Covering problem **3**

3 Alternative approaches and state of the art **6**

3.1 Population based approaches 6

3.1.1 Evolutionary Search Techniques 6

3.1.2 Ant Colony Optimization 9

3.2 Linear Programming approach 11

3.2.1 Linear Programs 11

3.2.2 Integer Linear Programming 12

3.2.2.1 Cutting planes 12

3.2.2.2 Branch and Bound 14

3.2.3 Lagrangian Relaxation 14

3.2.4 Integer Linear Programming formulation of the minimum set covering problem 15

3.3 Local Search strategies 16

3.4 Greedy algorithm 17

3.5 Backtrack algorithm 18

3.6 Conclusions 18

4 Background on CSP **20**

4.1 Constraint Satisfaction Problems 20

4.2 Propagation 21

4.3 Unary, binary and global constraints 22

4.4 The importance of finding the right model for a constraint satisfaction problem . 22

4.5 Minimization 23

5 Minimum set covering models **24**
5.1 Boolean variables model . . . 24
5.2 Count-based Finite Domain model . . . 24
5.3 Benchmarks . . . 25
5.4 Results . . . 27

6 A new minimizing NValue-Based model **29**
6.1 NValue-based Finite Domain model . . . 29
6.2 Results . . . 30
6.3 The NValue constraint . . . 30
6.4 The AtMostNValue constraint . . . 32
6.4.1 Graph theoretic concepts . . . 32
6.4.2 The computational complexity of AtMostNValue . . . 33
6.4.3 A greedy approximate approach for computing the independence number 33
6.4.4 Pruning N . . . 34
6.4.5 Pruning $\bar{X}$ . . . 34
6.4.6 Implementation issues . . . 34
6.4.6.1 Computation of the intersection graph . . . 37
6.4.6.2 Intersection graph reduction . . . 37
6.4.6.3 MD avoidance . . . 38
6.5 The AtLeastNValue constraint . . . 39
6.5.1 Algorithm for the violation cost of the AllDifferent constraint. . . . 39
6.5.2 Filtering algorithm for the AtLeastNValue. . . . 39

7 Finding all solutions of a minimum set covering **41**
7.1 The Boolean model . . . 41
7.2 Symmetry breaking algorithms for the NValue-based model . . . 42
7.2.1 Sequential accumulation of constraints. . . . 42
7.2.2 Relaxed tree accumulation of constraints. . . . 42
7.2.2.1 Proof of completeness of the algorithm. . . . 43
7.2.2.2 Example . . . 44
7.2.3 Combined approach. . . . 44
7.2.4 Breaking the Symmetries during search . . . 44
7.2.4.1 Pruning the domain of $\bar{X}$ . . . 44
7.2.4.2 An efficient data structure to store solutions . . . 45
7.3 Symmetry free Finite Domain model . . . 46
7.4 Analysis of the results . . . 47

8 Conclusions and further work **49**

List of Figures

2.1 Diagram of digestion 4

3.1 Example of a Gomory's cut. 13

4.1 An example of a CSP, the Sudoku game 21

6.1 An intersection graph and its maximum independent set. 33

7.1 Solutions tree 43

List of Tables

5.1 The datasets that will be used for testing purposes. 27
5.2 Comparison between the Boolean and the Count-based models. 28

6.1 Comparison between the Boolean and the NValue-based models. 31
6.2 Execution time of variations of the AtMostNValue filtering algorithm. 38

7.1 Comparison between each model able to find all minimum solutions 48

1

Motivation

The problem of yeast identification was historically addressed through the study of both morphological traits and physiological features [Yar98, BPY00, BM06], but alternative molecular methods have been adopted to obtain the sequence of particular genomic regions and thus identify a given species [KR98, SFFST02].

Although sequencing nucleic acids (DNA) is more accessible than ever, it is still an expensive technique, especially if applied to a high numbers of specimens. In contrast to less expensive techniques like RFLP, RAPD, MSP-PCR (which allow the formation of clusters among the specimens to be identified, with inherent result limitations in scope), ARDRA (Amplified Ribosomal DNA Restriction Analysis) [VRDV+92] was proposed to differentiate between species of a eubacterial family and it represents an approach that goes beyond the mere clustering operation. A fragment of the DNA of the specimen is obtained and copied many times. Enzymes are used to digest each copy, resulting in a set of fragments of DNA whose size can be meassured. Different species show different patterns of sizes for each enzyme, generally enabling the identification of the organism. Variations of ARDRA [SM98, BCFA99, WBLT90] were succesfully applied to distinguish specimens among particular sets of fungal and yeast species.

However, all these approaches are based on the **manual** selection of enzymes such that all species that they consider can be identified.

Recent papers are acknowledging the power of *in silico* contributions in this field. One is limited to the forecast of electrophoretic patterns [PJN07], the other presents a program to assess the utility of a fixed set of endonucleases to distinguish between a given set of sequences [WIR+08]. However, the integration of all the available data in a comprehensive in silico approach, targeting the automatic search of minimum sets of enzymes to identify the species of *any* given specimen among a defined set of possibilities is still to be proposed.

In this disertation, the problem of finding a minimum set of restriction enzymes suitable for the task is referred to as *The Species Differentiation Problem* and tackled by converting it into minimum set cover and then defining a variety of Constraint Programming approaches to devise an efficient way to solve it. Furthermore the diverse techniques to solve the minimum cover are aimed to find not just one minimum set cover, but all of them.

The different techniques are compared using as benchmarks different datasets (one corresponding to real data and the rest randomly generated) to validate the applicability of the methods.

2

Mapping the Species Differentiation Problem into a Minimum Set Covering problem

Among techniques used to identify species, ARDRA has been proposed in [VRDV+92]. This technique identifies one from a set of specimens through analysis of a specific DNA subsequence of its genome. Restriction enzymes (that, as is well known, cut DNA sequences at specific recognition nucleotide sequences, known as *restriction sites*) play a central role in the ARDRA technique that proceeds as follows: First, a "standard" fragment of the test specimen DNA is obtained (in the case of yeasts, the 5.8S-ITS region of their operons), and many copies of it are produced. Secondly, a set of restriction enzymes are separately applied to these copies. The complete digestion of each enzyme yields several smaller nucleotide segments that, subject to gel-electrophoresis, originate bands of different lengths.

Each *yeast - restriction enzyme* pair generates a specific band pattern, but given the similarity of their DNA, several yeasts are likely to present similar patterns when digested by most restriction enzymes. Subject to some experimental error, there is a one-to-one correspondence between fragment sizes and the position of the respective band in the pattern, hence the sizes of the fragments obtained can be approximately calculated from the gel electrophoresis experiments. On the other hand, when its DNA sequence is known, the pattern produced by the digestion of yeast Y (or rather, the 5.8S-ITS region of its operon) by the restriction enzyme R can be computed by running a simulation of a gel electrophoresis experiment. A simple diagram of digestion in this context is shown in Figure 2.1.

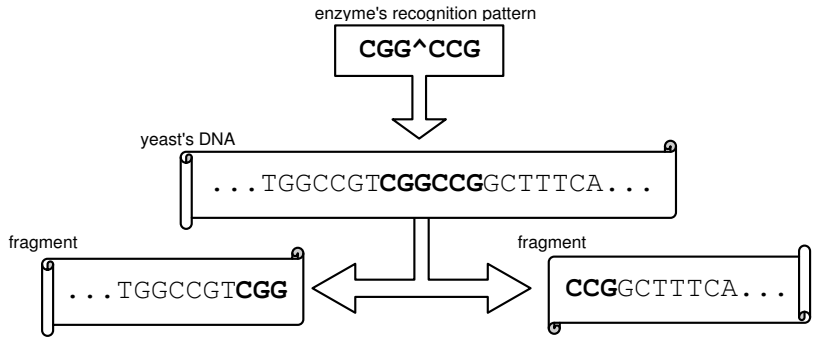

Figure 2.1: Diagram of digestion

A restriction enzyme R differentiates two yeast specimens Y_1 and Y_2 if the patterns it produces from them are distinguishable, i.e. at least one fragment in one of the digested yeasts is of a sufficiently different size from any fragment in the other digested yeast.

It is thus possible to produce a Boolean coverage table D, where rows denote yeast pairs (elements) and columns represent restriction enzymes (sets). In this table the cell in row $Y_i - Y_j$ and column E_k denotes whether that yeast pair is differentiated (covered) by that restriction enzyme. The problem of identifying a yeast among a set of similar yeasts can be formulated as finding a set C of restriction enzymes that differentiate any pair of yeasts, i.e. for all rows there is at least one enzyme in C such that its corresponding column has a true value for that row.

More formally, given a set of yeasts $Y = \{Y_1, ..., Y_{Ny}\}$ and a set of enzymes $E = \{E_1, ..., E_{Ne}\}$, I denote as $P(i,k)$ the induced pattern for Y_i by E_k, i.e. the set of segment lengths produced by the digestion of yeast Y_i by enzyme E_k. Two patterns P and Q are distinct if there is a fragment length in one of them that is sufficiently different (depending on the experimental error and denoted by $\not\approx$) from any fragment of the other pattern i.e.

$$\mathsf{distinct}(P,Q) =_{def} (\exists u \in P)(\forall v \in Q)(u \not\approx v) \vee (\exists u \in Q)(\forall v \in P)(u \not\approx v)$$

Two yeasts Y_i and Y_j are differentiated by a restriction enzyme E_k if the patterns induced in them are distinct:

$$\mathsf{differentiate}(i,j,k) =_{def} \mathsf{distinct}(P(i,k), P(j,k))$$

A discriminating set of enzymes S is a subset of the set E of enzymes that, for any pair of yeasts in the set Y, has an element that differentiates them, i.e.

$$\mathsf{discriminate}(S,Y) =_{def} \forall (i\text{-}j) \in Y \ \exists k \in S : \ \mathsf{differentiate}(i,j,k)$$

A minimal (optimal) discriminating set of enzymes S is a discriminating set with minimal cardinality:

$$\text{min\_disc}(S,Y) =_{def} \text{discriminate}(S,Y) \wedge (\forall R \ \text{discriminate}(R,Y) \rightarrow \#S \leq \#R)$$

Hence, given a set Y of yeast specimens, the Species Differentiation Problem can be regarded as the task of finding, from a set E of available restriction enzymes, a minimal discriminating set C for the set of yeast specimens. In set covering terms, this is translated to: given a universe $\mathscr{U} = \{u_1, ..., u_n\}$ defined as the set of all yeast pairs, and a family $\mathscr{S} = \{S_1, ..., S_n\}$ of subsets of $\mathscr{U}$ representing the covering of each enzyme, find asubfamily $\mathscr{C} \subseteq \mathscr{S}$ of minimum cardinality that covers all elements in $\mathscr{U}$.

Although proven to be NP-Hard [Kar72], the minimum set covering problem is also a main model for several important applications such as in bus [BC96a, WR93], railway [CNS95] and airline crew scheduling, logical analysis of data [BHIK00] and location problems of facilities [MMS01].

Gel electrophoresis apparatus can analyze more than one digested sequences of DNA at the same time, but different enzymes require different environments to cut DNA sequences. Consequently, the smaller the number of enzymes required, the less costly, cumbersome and time consuming the process of specimen identification will be. This fact justifies the search of a *minimum* set cover, instead of just any covering.

Not all enzymes might be available to each biochemistry lab, and the conditions required by each enzyme to function properly are different and require different facilities and apparatus. This means that the set of enzymes that one lab might regard as the best one can be very inappropriate for another for reasons that are very hard to formalize. Also some enzymes are more robust than others in the sense that they do not fail to cut DNA or can lead to the result of the gel electrophoresis experiments to be less prone to human error (J. Almeida, personal communication, January 24, 2010). This fact justifies the search for *all* minimum set coverings, so that the user will have as many alternatives as possible to chose from, depending on the particular conditions of the lab in which the analysis is done and their knowledge about the particular properties of each enzyme.

From now on, the work will be concentrated on the search of all minimum solutions to set covering problems through Constraint Programming, using both randomly generated data sets and real datasets obtained (during the experimental phase of this thesis) from simulations of gel electrophoresis experiments over DNA of yeast digested by enzymes.

Initial efforts to solve this problem were published [BBA10] and it raised the interest of the committee in the Workshop on Constraint Based Methods for Bioinformatics.

3

Alternative approaches and state of the art

In this section many different approaches to finding minimum set coverings are analyzed. The list is far from comprehensive, but each alternative represents one instance of a family of approaches to the problem.

3.1 Population based approaches

These methods involve the existence of many individuals (each of which represent different candidate solutions) which are repeatedly processed in the search for near optimum solutions.f

3.1.1 Evolutionary Search Techniques

This kind of techniques make use of Genetic Algorithms (GAs), which are search heuristics that mimic the process of natural evolution. This search heuristic is used to generate solutions to optimization and search problems.

In a GA, the candidate solutions (called *chromosomes*) are usually encoded as strings of data which represent their *genes*. The process of evolution starts from a random population of candidate solutions from which the best candidates (as judged by a *fitness function*) are selected and modified using diverse techniques to create the next generation of individuals. The algorithm terminates either after a fixed number of generations were produced, or a satisfactory level of fitness has been reached by an individual in the population.

The *fitness function* maps chromosomes into a numerical value, which represents how close the candidate solution is to have the properties that are expected from it.

The *initial population* consist of many individual solutions that are usually generated at random. The size of the initial population is highly problem dependent and it may vary from tens to tens of thousands. Sometimes, it is known that certain kind of individuals are near optimal solutions and in these cases this information can be used to form *seeds* in order to reduce the number of generations needed to achieve optimality.

The *selection stage* of a GA is usually implemented by evaluating the fitness function over each individual in the population and then, by choosing elements of it that show to be better approximations to optimal solutions. To generate the next generation from that set of selected individuals, methods as *crossover* and *mutation* are used. The crossover genetic operator is the analogous of sexual reproduction in the biological counterpart. It mixes two chromosomes by intertwining their genes in some way. Crossover generally comes in two varieties: *scattered crossover* and *n-point crossover*. Scattered crossover consists of the application of a particular mask to determine which genes will be taken from each parent. In n-point crossover, n points are defined so that sections between the points of the chromosomes of each parent are interleaved. *Mutation* consists of the random change of bits of an individual and its purpose is to preserve and introduce diversity in the system, so to avoid local minima by preventing the homogenization of the population.

The following outline summarizes how the GA works:

1. Random creation of the initial population.
2. Creation of a sequence of new populations by:
   (a) Scoring each member of the current population according to its fitness value.
   (b) Scaling the raw fitness scores to normalize the values
   (c) Selecting members for parenting, based on their fitness.
   (d) Production of children from the parents.
      - by randomly mutating a single parent or
      - by combining the chromosomes of a pair of parents using crossover
   (e) Replacing the current population with the children of the selected members.
3. Until the stopping criterion is met.

A very basic GA approach is taken in [DP08]. this technique is used to search for near-optimal solutions to big set covering problems. In their paper, the authors report using MATLAB's Genetic Algorithm Tool and testing the method using the OR-Library [Bea90], a collection of test data sets for a variety of Operations Research (OR) problems. The set covering problems provided there contain thousands of sets and hundreds of thousands elements. The chromosomes are integer vectors $\bar{X} = [X_1, ..., X_n]$ which represent the list of identifiers of the selected sets. The initial population is created giving preference to the apparition of sets that cover the greatest number of elements. The selection procedure used is *tournament selection* which consists of the execution of several games between random individuals, where the winners (according to the fitness function) are selected as parents for the next generation. In the

reproduction phase, both crossover an mutation are used. Their experimental results show that randomly generated masks for scattered crossover worked better than 1 and 2-point crossover. The stopping criteria used was a fixed number of generations which is, obviously, a parameter very dependent on the dataset. Their basic technique fails in finding better covers for the problems in the OR-Library than already known, but is presented here for introductory purposes.

In [BC96b] the chromosomes are represented as a binary vector $\bar{X} = [X_1, ..., X_n]$ where the selection of the $k-th$ set is expressed by $X_k = 1$. The initial population is randomly generated, as it is usually the case for this kind of algorithms. The selection phase involves dividing the population in two pools and tournament selection is used to extract winners from each. Each pair of selected individuals is combined to form a new child solution using the *fusion crossover operator*. This operator (originally proposed by the authors) results in the child solution to inherit more genes from the parent which is the fittest. Notably, their new mixing technique involves creating only one child from each pair of parents, while the classical cross over operators create at least two.

The children solutions generated in this manner are randomly mutated to breed the next generation of the population. This new generation does not take over the previous one, but it steadily replaces individuals which are less fit. As a consequence, the best individuals are always kept so that new comers can mate with old ones. Their approach to avoid local minima without introducing too much variation is to have a variable rate of mutation, which is indirectly proportional to the speed at which the populations genetically converge. This technique has the consequence that the dominating factor governing the direction of search changes with time. At first, when the population is highly diverse, the crossover operator dominates the search but as the evolution process advances it is mutation what stands over.

When new solutions are born, they are first corrected to close feasible solutions using greedy heuristics by the so called *repair* operator. Also their genome is cleaned from redundant genes (which represent sets that can be discarded without loosing the covering properties of the solution) as a way of maintaining solutions as small as possible.

This implementation depends on many parameters that have to be manually set. Some can be deduced from the properties of the dataset in analysis but others can not be known and it is the tester who tries different combinations and try to find the best ones. They report obtaining good results over the datasets in the OR-Library, some times finding solutions of smaller cardinalities than previously known, and other times failing to compete with other approaches.

A different approach is taken in [Ere99], where a non-binary representation is used for candidate solutions. The authors encode chromosomes as a list of set identifiers, one for each element in the universe, which represent the set that will be used to cover each element. With this representation, any individual is a feasible solution to the problem and no repair operator is needed (in contrast with the approach in [BC96b]). However, an additional operator is required to eliminate redundant sets from solutions arising from crossover and mutation. For this local improvement, the authors use a variety of greedy heuristics that find reduced versions of new born solutions.

The initial population is created so that for each gene, the sets that cover it have an equal probability of being chosen. The fitness function is defined in terms of the *cost* of the covering, which is reduced to the number of sets used when unicost set covering is considered. The selection operator in their GA implementation is probabilistic in the sense that it is more likely to choose more fit individuals. For mixing parent individuals they use a special purpose *LP-crossover* based in a relaxation of a Linear Integer Program solved through the Simplex method, what aims to find a good combination of the parent genes. The mutation procedure chooses gens at random and changes them also in a random fashion but giving higher probabilities to set identifiers of smaller cost (which becomes a uniform distribution in unicost set covering).

When they apply their GA to the OR-Library, they obtained very similar results to the pure Linear Programming (LP) approaches that will be presented later, meaning that the coverings found were almost always the same size as those found with the LP approaches. The execution time of their algorithm was some times above and some times under the other LP implementations, what does not allow much generalization about the efficiency of it.

Any evolutionary approach to set covering is bound to find only near-optimum solutions because it does not intend to prove that a better solution is not possible.

3.1.2 Ant Colony Optimization

The ant colony optimization algorithm (ACO) is a probabilistic technique for solving computational problems by reducing them to the search of good paths through graphs. Artificial ants in ACO algorithms can be seen as heuristics that iteratively generate solutions by taking into account search experiences accumulated in the past. These past experiences are modeled through *pheromone trails*

Biologists have observed that ants tend to find the shortest path between their colony and a source of food, what means that imitating their individual behavior might be useful. A way of modeling this individual behavior is:

1. An ant initially wonders randomly around the colony.
2. When it discovers a food source it returns directly to the nest, leaving in its path a trail of pheromone
3. These pheromones are attractive to other ants, which will be inclined to follow the track.
4. While returning to the colony after finding food at the end of that path, these ants will strengthen the pheromone trail.
5. It is usually the case that there are more than one single route to reach the same food source, but after a while the shorter one will be traveled by more ants than the other one.
6. The shortest route will be increasingly enhanced, becoming more attractive on each iteration.
7. Since pheromones are volatile, bad routes will eventually be discarded by the ants.

8. After a while all the ants will be following the same short route, meaning that they will have found a good solution to the problem.

ACO algorithms have been applied to the Set Covering Problem. For example in [SH00] an ant starts with an empty solution and constructs a complete one by adding sets until are elements are covered. Each set identifier j has a associated a pheromone trail τ_j, and a heuristic value η_j. τ_j indicates the learned *desirability* of including the set j in the solution of an ant and η_j indicates a different desirability value obtained by other means, such as the proportion of still uncovered elements that set j will cover if selected. The pheromone trails are updated after a fixed number m_a of solutions were constructed and improved by local search. The ants prefer selecting sets with high associated pheromone trail and/or heuristic value.

The algorithm outline for ACO applied to the Set Covering Problem is shown in Algorithm 1.

Algorithm 1 Ant Colony Optimization for the Set Covering Problem

```
1:  while ¬termination condition do
2:      for all k ∈ {1,...,m_a} do
3:          while solution not complete do
4:              applyConstructionStep(k)
5:          end while
6:          eliminateRedundantColumns(k)
7:          optimizeThroughLocalSearch(k)
8:      end for
9:      updateStatistics
10:     update Pheromones
11: end whilereturn best solution found
```

In the *MAX-MIN* Ant System proposed in [SH00], solutions are constructed by setting the probability of choosing each set for each ant. An ant k chooses set j with probability

$$p_j^k = \begin{cases} \frac{\tau_j(\eta_j)^\beta}{\sum_{g \notin S_k} \tau_h(\eta)^\beta} & \text{, if } j \notin S_k \\ 0 & \text{, otherwise} \end{cases}$$

where the parameter $\beta > 0$ sets the relative influence of heuristic against pheromone information. S_k is the partial solution achieved by the k^{th} ant. After all solutions are computed, the pheromone trails have to be updated by first evaporating them (updating τ_j to $(1-\rho)\tau_j$, $\forall j$) and then adding an amount $\Delta\tau = 1/z$ to the sets contained in the best solution found so far, where z is the cost of the solution used in the pheromone update (which in unicost set covering is the number of sets in the cover). When no new improved solution is found for a given number of iterations, the pheromone trails are re-initialized as a way to reset the search and avoid local minima.

Also, pheromones are initialized to τ_{max}, and the range of values for pheromone trails assigned to each set are limited to the interval $[\tau_{min}, \tau_{max}]$ as a way to avoid premature stagnation

of the search space.

In the paper it is reported that when compared to other algorithms, ACO approaches can reach state-of-the-art performance on various instances, but are not as powerful or robust as those based in Linear Programming. Moreover, ACO approaches are highly dependent in the manual setting of various parameters such as β and m_a.

3.2 Linear Programming approach

Linear programming (LP) constitutes a very important technique for the optimization of a linear objective function, subject to linear equality and inequality constraints. This method is used in many areas, one of it being business and economics due to the fact that problems like maximizing outcome can be straightforwardly stated and efficiently solved.

3.2.1 Linear Programs

In their canonical form, linear programs are expressed as:

$$\begin{aligned} \textbf{maximize/minimize} \quad & c^\top x \\ \textbf{subject to} \quad & Ax \leq b \end{aligned}$$

where x is a vector of variables whose values must be determined, c and b are vectors of known coefficients and A is a matrix of known coefficients. The expression $c^\top x$ is to be maximized/minimized within the limits defined by $Ax \leq b$.

As an example, suppose that a farmer owns a land of A square kilometers and wants to plant a combination of wheat and barley. The farm has a limited amount F of fertilizer and P pesticide, which are resources required by both crops in different amounts per unit of area. Wheat requires F_1 units of fertilizer and P_1 of pesticide, but barley needs F_2 and P_2 per unit of area. Let the selling prices of wheat and barley be S_1 and S_2 respectively. By denoting with x_1 and x_2 the areas planted of each crop, the problem of finding the optimal number of square kilometers to plant each crop can be expressed as the linear programming problem:

$$\begin{aligned} \textbf{maximize} \quad & S_1 x_1 + S_2 x_2 \\ \textbf{subject to} \quad & x_1 + x_2 \leq A \\ & F_1 x_1 + F_2 x_2 \leq F \\ & P_1 x_1 + P_2 x_2 \leq P \\ & x_1 \geq 0 \\ & x_2 \geq 0 \end{aligned}$$

In this simple problem using only two variables, the space search can be represented as a plane. Each of the constraints are actually linear functions that cross the plane delimiting the area of *feasible* solutions from that of *unfeasible* solutions.

Linear programming problems can be solved using different very well known methods such as *Simplex* [NM65], *ellipsoid* [GLS81] and *Interior Point* [Gli99].

Simplex is very efficient in practice, although exponential in the worst case. It takes advantage of the fact that solutions that maximize/minimize the objective function are present in the intersection between constraints and, using mathematical machinery, it visits the corners of the feasible solutions space in an order that guarantees better approximations on each iteration. This way the algorithm can assure that once a local maximum/minimum is found, it is also a global maximum/minimum. Alternative methods such as *Ellipsoid* and variations of Simplex [Kel06] show that Linear Programing is solvable in polynomial time, but these methods are too inefficient in practice to be of much practical use.

Oppositely to Simplex, Interior Point methods reach an optimal solution by traversing the interior of the feasible region. These methods are characterized by the use of continuously parametrized families of approximate solutions that asymptotically converge to the optimum solution. These paths trace smooth trajectories with algebraic properties that can be exploited by the algorithms.

3.2.2 Integer Linear Programming

Consider the manufacture of computers. A linear programming model might give a production plan of 130.6 computers per week. In such a model, it is a fairly reasonable assumption that 130 computers per week would be pretty close to optimality. On the other hand, suppose a company is building roads. Then a model that suggests that 0.8 roads should be built connecting some pair of cities and another 0.4 roads should be finished for other pairs of locations would be of little to no value. Roads come in integer quantities, and that fact should be considered by the models.

At first sight, this restriction of integrality may seem innocuous, but in reality it has far reaching effects. With integer variables, a whole new family of problems can be addressed, but the computation of optimum solutions becomes much more costly.

3.2.2.1 Cutting planes

Although solving linear programs with integer coefficients is an NP-hard problem, a relaxation of the integer condition permits a relatively efficient search for integer solutions when combined with *Cutting Plane* techniques.

Among the cutting plane techniques, Gomory's cuts [BCCN96] are probably the most general. A Gomory's cut is a linear inequality constraint that reduces the space search while not excluding any integer solution from it. The combined approach consists in first finding a solution to the relaxed ILP and then, if it is not an integer solution, using it to generate a new linear constraint (the Gomory's cut). This new constraint will effectively reduce the problem by taking the previous non-integer solution out of the search space without removing integer solutions. A new LP is created by adding the new constraint to the previous LP and the process is repeated until a solution is found that uses only integer coefficients. Of course, in the worst

case the problem still takes exponential time to be solved, but in practice the approach works efficiently enough to make it applicable to a broad range of problems.

To exemplify the concept of Gomory's cut consider this tiny Integer Linear Program problem:

$$
\begin{aligned}
\textbf{maximize} \quad & 5x_1 + 8x_2 \\
\textbf{subject to} \quad & x_1 + x_2 \leq 6 \ \ (1) \\
& 5x_1 + 9x_2 \leq 45 \ \ (2) \\
& x_1 \geq 0 \\
& x_2 \geq 0 \\
& x_1, x_2 \text{ integers}
\end{aligned}
$$

The relaxation of the problem can be plotted in a plane. In Figure 3.1 the constraints (1) and (2) are represented by lines and the point that maximizes the objective function is marked with a label. The non integer of the problem is $x_1 = 2.25$ and $x_2 = 3.75$ and it is used to create the Gomory's cut labeled as (cut), which represents the additional constraint $2/3x_1 + x_2 \leq 5$. Note how this cut does not prune any integer solution from the search space (shadowed) but effectively removes the previous solution from it. The process can be repeated iteratively adding new cuts until an integer solution is found.

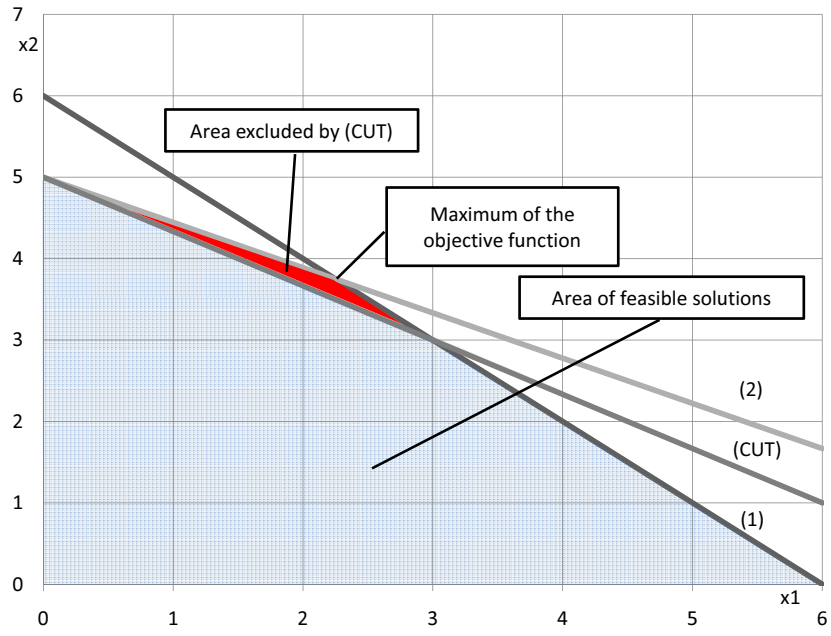

Figure 3.1: Example of a Gomory's cut.

3.2.2.2 Branch and Bound

Branch and Bound techniques work rather differently than cutting planes. They consist in dividing the search space by choosing a variable and *branching* over, creating different linear programs, each of which has a different value fixed for that variable. A *bounding function* estimates the best value of the objective function obtainable in each branch in an optimistic manner, and the Branch and Bound algorithm visits first the nodes predicted to give better solutions. As solutions are found, all branches pending to be solved whose bounding function predicts worst results than the best solution known can be safely discarded. This *pruning* mechanism will not cut any potentially good branch since the bounding function is an optimistic one, meaning that the value predicted by it will be always better or equal than the objective function applied to the best solution in that branch.

The most delicate part in the use of Branch and Bound techniques is finding a tight bounding function. The computation of the bounding function often consist of solving the Lagrangian dual of the original problem. The original formulation of the LP is called the *primal* problem and the dual consist of a transformation which, when solved, provides an upper bound to the optimal value of the primal problem. This transformation has the property that the dual of a dual linear program is the original primal program. By using the dual of a problem in a branch, one can therefore predict a bound on how good solutions in that branch will be and then decide if the branch is to be discarded or not.

3.2.3 Lagrangian Relaxation

The Lagrangian relaxation is even softer than the linear relaxation and it is used to find near-feasible solutions in an efficient manner. It works by moving the constraints into the objective function so as to consider their lack of satisfaction as a penalty. The Lagrangian dual of a Integer Linear Program is often solved through *subgradient optimization*. These problems frequently involve an objective function which is not differentiable in all points of its domain. Assuming the objective function is to be maximized, this means that in those points, the direction of highest gradient (in which the objective function will increase the most) can not be computed by using the derivate of the objective function, since it does not exist. On the other hand, it is possible to use the concept of *subderivative* of the function in a point, which is a slope such that if a line with that slope passes through that point, it is everywhere either touching or below the function but it never crosses through it. The subgradient optimization method takes advantage of this concept and uses it to compute a subgradient of the objective function in the points in which it is not differentiable. This subgradient will point in a direction that will make the objective function increase, and by iteratively following that path, the subgradient optimization method travels though the search space in a direction that maximizes the objective function.

It should be noted that at any non differentiable point of a function, there are an infinite number of subgradients, and it is generally not possible to find the one for which the objective function increases the most.

Some times the Lagrangian relaxation is not used within the context of Branch and Bound,

but just as a heuristic to find near optimum solutions.

3.2.4 Integer Linear Programming formulation of the minimum set covering problem

Given the coverage table D a boolean $m \times n$ matrix, where $M = \{1,...,m\}$, $N = \{1,...,n\}$, a column $j \in N$ is said to *cover* a row $i \in M$ if D_{ij}=1. The minimum set covering problem calls for a minimum subset $S \in N$ of columns such that each row $i \in M$ is covered by at least one column $j \in S$. The Integer Linear Programing definition of the problem is then:

$$\begin{aligned} \textbf{minimize} \quad & \sum_{j \in N} x_j \\ \textbf{subject to} \quad & \sum_{j \in N} D_{ij} x_i \geq 1 \quad i \in M \\ & x_j \in \{0,1\} \quad j \in N \end{aligned}$$

where $x_j = 1$ iff $j \in S$.

As it was said above, the relaxation of the last constraint to $0 \leq x_j \leq 1$, transforms the problem into a non-integer version. This allows much more efficient computation of *near* solutions by allowing instances in which sets are not selected nor discarded. In conjunction with Gomory cuts this technique is able to find *proper* minimum covers in a very efficiently manner. Alternative to the cutting plane techinques, Branch and Bound techniques often use Lagrangian relaxations to solve the dual problem through subgradient optimization methods to provide bounding functions.

The algorithm proposed in [Bea87] uses the Branch and Bound method and the lower bounds are computed using the Lagrangian relaxation together with subgradient optimization. The relaxation of integer problems in the branches are solved to optimality by the dual simplex method. Several dominance procedures to reduce both the number of rows and columns are applied. These dominance procedures basically eliminate columns which cover a subset of rows with respect to other columns, and elements which are covered by a superset of columns with respect to others. Instances of up to 400 rows and 4000 columns are tackled, finding coverings of minimum cardinality.

In [CNS95] it is reported that a Lagrangian-based heuristic solved through subgradient optimization gave impressive results and found very good solutions (although not minimum) to the problem of crew-scheduling at the Italian Railways. This problem generated huge set covering problems with 1000 sets and 10000 elements. The authors found that the dataset was very sparse and had the property that sets that were listed together tended to be similar to each other, facts that where used to compress the internal data structures. The compressed structures were exploited by their optimization methods to increase the overall efficiency of the system. The dataset is reported to be so large that it had to be reduced by eliminating sets using a heuristic strategy. Although fine tuned to its particular application, their technique is said to be robust enough to work also with instances of the OR-Library, but not so efficiently

the other LP implementations.

In [LNF95] a different heuristic based on continuous surrogate relaxations and subgradient optimization is presented. Since their approach does not use either Branch and Bound nor cutting planes, the method is also aimed at huge set covering problems for which near-optimal solutions are deemed acceptable. The surrogate relaxations are created by combining many constraints into one single (less strict) constraint, with the expectation that the simpler system might also mantain some collective property of the constraints. The subgradient optimization method is used to guide the search in the direction of better solutions. Their algorithms applied to the datasets in the OR-Library showed it to be very competitive, finding better solutions than previously known for a couple of instances.

The field of Linear Programming techniques to solve set covering problems is very deep and there exist a wide variety of heuristics, relaxations and transformations to fine tune the search for different kinds of datasets. In this regard, what is important to note is that the main emphasis is put into developing strategies to cope with very large datasets for problems in which only one near-optimal solution is required. This fact makes most of them inappropriate to solve the enzymes problem, where the aim is to find all truly minimum coverings. Techniques like cutting planes and Branch and Bound can be used to find true minimum set coverings and might be applicable to the datasets which are dealt with in this dissertation. However, the addition of non linear constraints (as proposed in the Further Work section of the present thesis) would also make them inappropriate to solve the arising problem.

3.3 Local Search strategies

Local search consists of a metaheuristic used to solve computationally hard optimization problems. It can be applied when the problems are formulated as the search for a solution that maximizes a criterion among a number of candidate solutions.

This kind of algorithms traverse the search space by iteratively moving from candidate solution to candidate solution following a path through the *neighborhood relation*, until a solution deemed good enough is found, or a time bound is elapsed. Usually every candidate has more than one neighbor solution and the choice between them is made with the aid of information about the neighborhood (hence the name *local*) and previous experience.

Typically a set of constraints that an appropriate solution should satisfy is defined and, even though candidate solutions that violate the constraints are permitted, the number of satisfied constraints is used as part of the maximization criterion.

In [Mus06] this technique is applied to solve large set covering problems with some success. The author defined the solutions as a list of identifiers of the sets in the family $\mathscr{S}$. The neighbor relation is defined through the application of three basic *moves*: $ADD\_SET(S)$, $REMOVE\_SET(S)$ and $SWAP\_SETS(S1,S2)$ which generate alternative solutions from the original by adding, removing and replacing sets. Search space pruning is achieved by bounding the size of candidate solutions to be smaller than the size of the last valid solution found. In this context, a solution is called *valid* if it constitutes a cover.

To guide the search, a fitness function is defined to be the sum of the size of the solution and the number of elements that it leaves uncovered. This function is not computed from scratch each time, but it is updated from previous computations of it.

The initial solution is created using a greedy algorithm that starts with an empty solution and iteratively adds the sets which cover most still uncovered elements. Cycles in the search are avoided through a *tabu* list (of a size related to the size of a solution obtained by the greedy algorithm) that temporarily forbids adding/removing sets recently deleted/added.

The stopping criteria used is based on bounding the number of times that the moves can be applied without improving upon the best solution found so far.

This very simple method is cited with introductory purposes, and it is not very competitive with respect to other more involved approaches as the one presented next.

In [YKI03] the solutions are not defined as the list of selected set identifiers but rather as the binary vector $\bar{X} = \{X_1, ..., X_n\}$, where the selection of the $k-th$ set is represented by $X_k = 1$. The neighbor relation is not defined by erasing, adding or replacing sets one at a time, but by switching many elements of $\bar{X}$ at once. The r-flip neighborhood of a solution $\bar{X} = \{X_1, ..., X_n\}$ is the set of solutions obtainable by flipping at most r elements of $\bar{X}$. The size of such a neighborhood is $O(n^r)$ but they propose an implementation based on a Linear Program with Lagrangian relaxations to reduce the number of candidates in the neighborhood without sacrificing solution quality. Their experimental results showed that $r = 3$ gives a very good trade off between the size of the neighborhood and the speed of convergence to near-optimality.

A strategic oscillation mechanism is used to travel through the search space alternating between feasible and infeasible solutions. This technique is used as a way to avoid local minima. Following a path through the frontier is justified by the intuitive fact that changing very little in minimum solutions will result in uncovered elements, what means that minimum solutions must be in the edge of feasibility.

Local search is alternated with variable fixing. This trick also uses a Lagrangian heuristic so that the sets considered as best by it are selected, and those which are considered worse are automatically discarded.

The objective function is defined to be the number of elements not covered (or the sum of their costs in the weighted version) together with a penalty that is used to avoid cycles. Their stop criterion is a fixed number of iterations without finding better solutions to the best known so far.

This sophisticated method is reported to be very robust and efficient, winning also over many Linear Programing approaches in some instances of the OR-Library.

3.4 Greedy algorithm

Since the algorithm is referred to many times in this dissertation, it is considered here. The greedy approach to find a set covering is implemented by accumulating the best sets in $\mathscr{S}$ (i.e. those that cover the most elements in $\mathscr{U}$ still to be covered) until all the universe is covered.

Algorithm 2 Greedy algorithm

Input:
— $\mathscr{U} = \{u_1, ..., u_n\}$, the universe of elements
— $\mathscr{S} = \{S_1, ..., S_m\}$, the family of subsets of $\mathscr{U}$
Output:
— C a cover of low cardinality

1: $C \leftarrow \emptyset$
2: $U \leftarrow \mathscr{U}$
3: **while** $U \neq \emptyset$ **do**
4: $\quad s \leftarrow \underset{k \in \mathscr{S}}{\operatorname{argmax}} |k \cap U|$ ▷ select the most covering set with respect to the still uncovered elements
5: $\quad C \leftarrow C \cup \{s\}$ ▷ the selected set is added to the cover family
6: $\quad U \leftarrow U \setminus s$ ▷ the covered elements are removed
7: **end while**

The pseudo code is shown in Algorithm 2.

Of course, this greedy approach does not guarantee that, upon termination, a minimum cover will be found. In fact, notwithstanding the very fast execution time of the algorithm, the solutions found with the benchmark datasets that will be presented latter were never minimum. However, this algorithm can be used in the minimization process to establish a first upper bound for the size of the minimum cover.

3.5 Backtrack algorithm

This kind of algorithms guarantees optimality by searching almost over the whole search space. After first solution is found, it can prune all possibilities that make use of worse solutions. The way in which this approach travels through the search space slightly resembles Constraint Programming approaches, introduced in the next chapter. The pseudo code is shown in Algorithm 3.

This approach involves searching blindly for a complete coverage of the universe. All exact algorithms are bound to be exponential in complexity since minimum set covering is NP-hard, but as this algorithm makes no inference about the consequences of choosing one set over another, it is a very inefficient one.

3.6 Conclusions

In contrast with local, population based and greedy search methods, the backtrack algorithm is able to find all minimum solutions. This method makes binary choices but it does not attempt to infer consequences of the choices before it proceeds further. Integer Programming does such inference but there seems to be no effort to find all minimum solutions in the literature. Constraint programming approaches aims at overcoming the problems of all these approaches through constraint propagation as we will see in the following chapters.

Algorithm 3 Backtrack algorithm

Input:
— $\mathscr{U} = \{u_1, ..., u_n\}$, the universe of elements
— $\mathscr{S} = \{S_1, ..., S_m\}$, the family of subsets of $\mathscr{U}$
Output:
— Solve($\emptyset, \mathscr{S}, \mathscr{S}$) returns a cover of minimum cardinality

Procedure Solve($S, R, Best$) ▷ S: selected sets, R: available sets, $Best$: best covering found
1: **if** $|S| \geq |Best|$ **then**
2: return $Best$
3: **else if** S covers $\mathscr{U}$ **then**
4: return S
5: **else**
6: $R \leftarrow \{s\} \cup R'$ ▷ select s non deterministically
7: $S' \leftarrow S \cup \{s\}$
8: $S_1 =$Solve$(S', R', Best)$ ▷ binary choice
9: $S_2 \leftarrow$Solve(S, R, S_1)
10: return S_2
11: **end if**
End Procedure

Also the set covering instances tackled in the literature are very large and sparse (each set covers very little), which is exactly the opposite case to the datasets that the Species Differentiation Problem require, as it will be shown in Section 5.3. Also there seems to be little to no attempts to solve the problem using Constraint Programming approaches, what leaves some room for experimentation.

4

Background on CSP

In this chapter the main aspects of Constraint Satisfaction Problems are reviewed to provide the framework of the models proposed in following chapters.

4.1 Constraint Satisfaction Problems

Constraint satisfaction problems (CSPs) are mathematical problems whose definition includes a set of objects whose state must satisfy a number of constraints. These constraints are basically limitations that state relations between the objects that must be preserved. Objects are represented as homogeneous variables whose state is the value that they are assigned to.

As a simple and representative example of a CSP let us take the case of the very well known game Sudoku. This game consists of a table of 9×9 cells divided in sections of 3×3 as shown in Figure 4.1. Each cell in the table can be assigned to numbers from 1 to 9 and some of the cells come already labeled. The rules of the game state that there cannot be repeated numbers in any row, column or 3×3 section. In the CSP of Sudoku, there are 9×9 variables whose domains are numbers in the range 1..9 and the constraints simply state the rules of the game, namely that all the variables in a row, a column or a section must be different from each other. Apart from how easy it is to define the problem in this manner, the advantage of using this approach is that there are very efficient algorithms for filtering the allowed values for each variable given a partial labeling of them.

More formally a CSP is a triplet $< \bar{X}, \bar{D}, \mathscr{C} >$, where

- $\bar{X} = \{X_1, ..., X_n\}$ is a set of variables.
- $\bar{D} = \{D(X_1), ..., D(X_n)\}$ is the finite set of possible values for each variable in $\bar{X}$, that is to

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 4.1: An example of a CSP, the Sudoku game

say the *domains* of the variables.

- $\mathscr{C}$ is the finite set of constraints imposed over $\bar{X}$.

A constraint C in the set of constraints $\mathscr{C}$ over the set of variables $var(C) = (X_{i_1}, ..., X_{i_r})$ is a subset $rel(C)$ of the Cartesian product $D(X_{i_1}) \times ... \times D(X_{i_r})$ that specifies the *allowed* combinations of values for the variables $X_{i_1}, ..., X_{i_r}$.

Given a subset $\bar{Y}$ of the variables in $\bar{X}$, an *instantiation* I of $\bar{Y}$ is an assignment of values to variables such that $\forall X \in \bar{Y}$, the value a assigned to X belongs to $D(X)$. An instantiation I *satisfies* a constraint C if and only if the projection of I on $var(C)$ belongs to $rel(C)$. If I does not satisfy the constraint C, then it is said that I *violates* it.

C is *consistent* if and only if there exists a tuple of $rel(C)$ which is valid. A value $a \in D(X)$ is *consistent* with C if and only if $X \notin var(C)$ or there exists a valid tuple of $rel(C)$ in which a is the value assigned to X.

The *Constraint Satisfaction Problem* (CSP) consist of finding a full instantiation I of $\bar{X}$ such that $\forall C \in \mathscr{C}$, I satisfies C.

4.2 Propagation

Constraint solvers typically explore partial instantiations enforcing a local consistency property using specialized and general purpose propagation algorithms. Local consistency requires that all consistent partial instantiations can be extended to another variable in such a way that the resulting assignment is consistent. Local consistency properties can be grouped into various classes depending on how strict they are. The main classes of local consistency are node, arc and path consistency.

Node consistency requires that all unary constraints on a variable are satisfied by each of the values in its domain. This condition can be enforced by simply reducing the domain of each variable to that which satisfies all unary constraints on the variable.

The case of *arc consistency* is slightly more complicated. Given a constraint C on the variables $\bar{Y} \subseteq \bar{X}$, a *support* for $Y_i = v_j$ on C is a partial assignment V of $\bar{Y}$ containing $Y_i = v_j$ that satisfies C. This means that all the other variables in $\bar{Y}$ must have values in their domains that are

consistent with $Y_i = v_j$ with respect to C. If all values in Y_i have support, Y_i is said to be arc consistent. The variable Y_i is *generalized arc consistent* (GAC) on C if and only if every value in $D(Y_i)$ has support on C. A constraint C is said to be GAC if and only if each constrained variable is GAC on C. Enforcing GAC requires eliminating all the values from each variable that do not have support.

Path consistency is similar to arc consistency but it requires pairs of variables to have support. A pair of variables is path consistent with respect to a third variable if any arc consistent assignment of the pair of variables has support on the third one. This kind of consistency is usually not enforced by solvers since its complexity is generally not justified by its pruning power.

Constraint propagation then proceeds by enforcing node and arc consistency over all the variables, effectively reducing the search space without eliminating any solution that satisfies the constraints.

4.3 Unary, binary and global constraints

Constraints are classified depending on the number of variables that they relate. Unary constraints involve only one variable and its domain can be reduced simply by enforcing node consistency over it. Binary constraints state relations between pairs of variables, and their propagation involves enforcing arc consistency. Constraints that involve more than two variables are regarded as *global constraints*. For this class of constraints it might be possible to enforce path consistency, but for efficiency reasons the more relaxed approach of enforcing generalized are consistency is used. Specialized algorithms for enforcing GAC in many global constraints exist, and they work by analyzing underlying properties of the system that have to be verified.

A prominent case of a global constraint is that of AllDifferent [Ré94], in which bipartite graph theory is used to enforce GAC in polynomial time. This constraint is equivalent to stating inequality constraints between each pair of variables, but this simplistic approach misses some properties than can only be exploited when all variables are considered together. The most efficient filtering algorithms for this constraint build a bipartite graph containing the variables in one partition and the set of values in the other, and proceeds by pruning values from variables that do not belong to any maximum matching of the bipartite graph. In following sections the case of the *NValue* global constraint will be analyzed in detail.

4.4 The importance of finding the right model for a constraint satisfaction problem

Most problems can be stated in many different ways that affect deeply the size of the search space and the dynamics of the pruning algorithms. Finding the best way of modeling a problem using constraints involves choosing the right representation of the solutions so that the kind of constraints that can be stated are efficient enough and have sufficient pruning power to solve the problem.

Furthermore sometimes a set of constraints define the problem completely, but the specification of additional *redundant* constraints accelerates the process of finding solutions by allowing the application of more pruning algorithms.

Other times the problems are required to be solved in steps being the set of constraints applied on each step complementary to those on the previous one. First a CSP is solved to find the parameters to build another CSP. This is the case of finding all minimum solutions to set covering problems studied in this dissertation. First the size of a minimum cover is found using a model that is efficient for that task, and then another model (some times similar to the previous one) is used to find all solutions knowing already their cardinality.

4.5 Minimization

Constraint satisfaction problems are also used to find optimal solutions of various optimization problems. In this family of problems constraints are used to specify the properties that a solution must satisfy. The interest is put not in any valid solution but only in those which minimize an objective function defined over the elements of the candidate solutions.

The main method used in this kind of problems is Branch and Bound (BB). The Branch and Bound procedure is an intelligently structured search over the search space. The space of all feasible solutions (that which satisfy the constraints) is partitioned repeatedly into increasingly smaller subsets and an upper bound is calculated for the objective function of the solutions that fall in each subset. After each partitioning, those subsets with a bound that is greater than that of a known solution are excluded from the search. The partitioning proceeds until a solution is found such that the objective function applied to it results in a value which is smaller than the bound for any subset.

In the case of minimum set covering, the objective function gives a lower bound on the size of the candidate solution. Whenever a branch can be predicted to use more sets than the best solution found so far (by means of the objective function), the BB method will automatically discard it.

In Constraint Programming the objective function is actually represented as a variable, called objective variable. A set of constraints are imposed that relate it with the other variables in the problem, so that it reflects the objective function. As the BB method explores a branch, the domain of the objective variable is pruned by the enforcement of the local consistency property, what effectively gives bounds to the objective variable, allowing the BB method to know when a branch can be discarded.

The next chapter will show the application of minimization to solve the set covering problem in constraint programming approaches.

5

Minimum set covering models

Two different methods to solve the problem of finding the size of a minimum cover are presented here. Their implementation was done in Sictus Prolog and shred some light into the subtleties of minimum coverings.

5.1 Boolean variables model

This model is easier to define in terms of a coverage matrix representation. Given D an $m \times n$ matrix, where $M = \{1,...,m\}$, $N = \{1,...,n\}$, a column $j \in N$ is said to *cover* a row $i \in M$ if D_{ij}=1.

A Vector of boolean variables $\bar{X} = \{X_1,...,X_n\}$ is created were the selection of the j^{th} set in the covering is modeled by $X_j = 1$. A Constraint Programming system simply solves the problem of finding an assignment of $\bar{X}$ such that the selection of sets covers all the universe. The covering of the j^{th} element is represented the assertion $\sum_{j=1}^{n} X_j D_{ij} \geq 1$. By minimizing the sum of $\bar{X}$ while labeling it, the minimum solution can be found rather efficiently. The pseudo code is shown in Algorithm 4. This model performs very well in practice since the propagation of constraints drastically prunes the space search of the vector $\bar{X}$.

5.2 Count-based Finite Domain model

Vector $\bar{C}$ is defined having a size fixed on an upper bound of the cardinality of a minimum cover (this upper bound can be computed using the simple greedy algorithm).

Set constraints are imposed over the variables in the vector stating that for each element of $\mathscr{U}$ it is the case that at least one set identified in $\bar{C}$ contains it. This means that after labeling, $\bar{C}$ will consist of a list of set identifiers of $\mathscr{S}$ that represent a cover.

Algorithm 4 Boolean CP model

Input:
— D, an $m \times n$ matrix
Output:
— $\bar{X}$ a vector representing a selection of sets that constitute a cover of minimum cardinality

1: $\bar{X} \leftarrow [X_1, ..., X_n]$ ▷ There is one boolean variable for set in $\mathscr{S}$
2: **for all** $j \in 1..n$ **do**
3: $\quad X_j \in 0..1$
4: **end for**
5: **for all** $i \in 1..m$ **do**
6: $\quad \sum_{j=1}^{n} X_j D_{ij} \geq 1$ ▷ The cover of each element is imposed as a constraint
7: **end for**
8: label($\bar{X}$): minimizing$\left(\sum_{j \in 1..n} X_j \right)$

The search for a minimum solution is done by adding one extra value $\emptyset$ to the domain of the variables of $\bar{C}$ that represents a null selection. By labeling $\bar{C}$ while maximizing the count of $\emptyset$ elements, the number of selected sets is minimized, and the result is a minimum set cover. The variables of $\bar{C}$ are constrained to follow a strict ordering, and an extra limitation is imposed to assure that the non null selections will be present all together at the beginning of the vector; this way the apparition of symmetries is avoided. The pseudo code is shown in Algorithm 5.

Algorithm 5 Finite Domain Count-based model

Input:
— $\mathscr{U} = \{u_1, ..., u_n\}$, the universe of elements
— $\mathscr{S} = \{S_1, ..., S_m\}$, the family of subsets of $\mathscr{U}$
— N, an upper bound on the cardinality of a minimum set covering
Output:
— C a cover of minimum cardinality

1: $\bar{C} = [C_1, ..., C_N]$
2: **for all** $i \in 1..N-1$ **do**
3: $\quad C_i < C_{i+1} \vee C_{i+1} = \emptyset$ ▷ symmetries are broken by imposing a strict ordering
4: $\quad C_i = \emptyset \Rightarrow C_{i+1} = \emptyset$ ▷ and by forbidding sets after the first $\emptyset$
5: **end for**
6: **for all** $u \in \mathscr{U}$ **do**
7: $\quad E \leftarrow \{S : S \in \mathscr{S}, u_i \in S\}$ ▷ E represents all sets that cover the element u_i
8: $\quad \bigvee_{i \in 1..N} C_i \in E$ ▷ a disjunctive constraint is imposed to assure total covering
9: **end for**
10: Count($\bar{C}, \emptyset, M$)
11: label($\bar{C}$): maximizing(M)

5.3 Benchmarks

The datasets of set covering problems freely available for testing purposes are generally incompatible with the kind of instances needed to test the algorithms presented in this dissertation.

The intended area of application of these methods is that of relatively small set covering problems with particular properties arising from the patterns obtained from gel electrophoresis experiments applied to digested fragments of DNA. As it is stated in the motivation section, this is useful to enable low cost identification of species in a family of physically very similar organisms.

One dataset was constructed using real data about the Species Differentiation Problem and it is labeled as *Real_data*. The enzymes were applied to the DNA of each yeast species and then gel electrophoresis simulations were computed to find the patterns that each combination of enzyme - yeast would produce. This patterns are compared to find the yeast pairs that each enzyme is capable of differentiate.

However, since the information necessary to build more datasets for benchmarking purposes is not easily available, the only alternative is to study the properties of the original problem and use that information to artificially build new different coverage tables.

It only takes one digested DNA fragment to be of a different size in two species to make them distinguishable through the enzyme with which it was digested, so in spite of the fact that physically similar organisms typically share recent common ancestors (what makes their DNA to be similar), enzymes have an outstanding capacity for differentiating them. The consequence of this is that each enzyme will cover a considerable percentage of the total numbers of yeast pairs. In the context of set covering (and using the real dataset as reference), this means that when generating random datasets the probability p that a set will cover any given element should be high. In turn, this fact results in very dense coverage tables. Here (with the exception of datasets Gen_25 and Gen_26) values for p between 45% and 65% are considered:

$$0.45 \leq p \leq 0.65$$

Moreover, the typical number of species in this kind of analysis is not large [VRDV+92], and here between 15 and 25 species will be assumed. The number of different species pairs that can be constructed form n different species is $n*(n-1)/2$, resulting in a universe of a size between 100 and 300:

$$100 \leq |Universe| \leq 300$$

Finally the total number of commercially available enzymes is 350 but for testing purposes, the total number of sets in the family will be considered to be between 250 and 450.

$$250 \leq Sets \leq 450$$

These parameters are somewhat arbitrary, but are based upon the analysis of the problem which the generated datasets intend to represent, and the assumption that the information gathered from real enzymes and yeast species is representative.

The parameters created to compare the efficiency of the various minimum set covering solvers are presented in Table 5.1. The first line corresponds to the dataset generated using the results of gel electrophoresis experiments over the DNA of 23 yeast species digested by 350

Name of the dataset	Number of Sets	|Universe|	p
Real_data	350	253	0.5
Gen_1	250	100	0.45
Gen_2	250	100	0.55
Gen_3	250	100	0.65
Gen_4	250	200	0.45
Gen_5	250	200	0.55
Gen_6	250	200	0.65
Gen_7	250	300	0.45
Gen_8	250	300	0.55
Gen_9	250	300	0.65
Gen_10	350	100	0.45
Gen_11	350	100	0.55
Gen_12	350	100	0.65
Gen_13	350	200	0.45
Gen_14	350	200	0.55
Gen_15	350	200	0.65
Gen_16	350	300	0.45
Gen_17	350	300	0.55
Gen_18	350	300	0.65
Gen_19	450	100	0.45
Gen_20	450	100	0.55
Gen_21	450	100	0.65
Gen_22	450	200	0.45
Gen_23	450	200	0.55
Gen_24	450	200	0.65
Gen_25	450	300	0.25
Gen_26	450	300	0.35
Gen_27	450	300	0.45

Table 5.1: The datasets that will be used for testing purposes.

enzymes and the other 27 were randomly generated using the parameters stated in each line.

5.4 Results

The processor time in seconds taken by each of the minimizing algorithms applied over each of the datasets is shown on Table 5.2. In the table, the cells marked with (-) represent a time longer than 3000 seconds. For the datasets Gen_25 and Gen_26 the size of the minimum cover was not found by any of the models within the time limits imposed. The geometric mean of the speedup factor of the Boolean model for the rest of the datasets is 2.76, showing that the Boolean model is generally the fastest of the two. This fact motivated effort into finding another Finite Domain model that might improve over the Boolean one.

Dataset	Size of cover	Boolean	Count-based	Speed (Boolean)
Real_data	2	10	20	2
Gen_1	5	3	25	8.33
Gen_2	3	0.5	20	40
Gen_3	3	5	45	9
Gen_4	6	45	50	1.11
Gen_5	4	12	23	1.91
Gen_6	4	4	15	3.75
Gen_7	5	15	27	1.8
Gen_8	3	20	19	0.95
Gen_9	3	7	10	1.42
Gen_10	6	65	101	1.55
Gen_11	6	70	95	1.35
Gen_12	2	33	110	3.33
Gen_13	3	12	60	5
Gen_14	3	13	23	1.77
Gen_15	3	10	21	2.1
Gen_16	6	150	312	2.08
Gen_17	6	44	430	9.77
Gen_18	2	10	6	.6
Gen_19	3	8	30	3.75
Gen_20	4	125	-	-
Gen_21	3	6	30	5
Gen_22	5	105	433	4.12
Gen_23	3	115	735	6.4
Gen_24	3	235	103	0.43
Gen_25	-	-	-	-
Gen_26	-	-	-	-
Gen_27	6	2552	-	-
Geometric mean				2.76

Table 5.2: Comparison between the Boolean and the Count-based models.

6

A new minimizing NValue-Based model

The failure of the minimizing Count-Based model presented in the previous section motivated further analysis of the problem. This resulted in a very efficient model based in the NValue global constraint which is capable of finding the size of a minimum cover in record time.

6.1 NValue-based Finite Domain model

This model is somewhat the dual of the boolean model. Instead of finding a solution through the labeling of a vector that has as many elements as the cardinality of $\mathscr{S}$, this approach involves a vector with one variable for each element in $\mathscr{U}$

Now each variable X_i in the $\bar{X}$ vector is associated to the element u_i in the universe, and its domain is the set of identifiers of subsets in the family $\mathscr{S}$ that cover such element. By labeling $\bar{X}$ while minimizing the number of different values that it makes use of, minimum set coverings are found. The pseudo code is shown in Algorithm 6.

To be effective, this model requires minimization of the number of distinct values in list X (or equivalently, the size of set C). In CP systems this can be achieved using the $Nvalue(N, L)$ global constraint (proposed in [BHH+06]) that maps into the finite domain variable N, the number of distinct values in list L.

Algorithm 6 NValue-based Finite Domain model

Input:
— $\mathscr{U} = \{u_1, ..., u_n\}$, the universe of elements
— $\mathscr{S} = \{S_1, ..., S_m\}$, the family of subsets of $\mathscr{U}$
Output:
— C a cover of minimum cardinality

1: $\bar{X} = [X_1, ..., X_{|\mathscr{U}|}]$ ▷ one Finite Domain variable for each element in $\mathscr{U}$
2: list_to_set($\bar{X}, C$)
3: **for all** $i \in 1..|\mathscr{U}|$ **do**
4: $X_i \in \{k \in 1..|\mathscr{F}| : u_k \in S_i\}$ ▷ the domain of X_i is the set of set identifiers in $\mathscr{F}$ that cover the i^{th} element of $\mathscr{U}$
5: **end for**
6: label($\bar{X}$): minimizing($|C|$)

6.2 Results

Table 6.2 shows a comparison between the processor time (in seconds) required by the Boolean and the NValue-based models. Both models were implemented in Sicstus Prolog and (in general terms) NValue proved to be much more efficient than the Boolean one, with a geometric average speedup of 2.06. This is due to the fact that the search space of this model only contains feasible solutions, and there are very efficient filtering algorithms of the NValue constraint.

6.3 The NValue constraint

In order to understand the subtleties behind this model, the NValue global constraint will be now analyzed in detail. This constraint was also partially implemented in CaSPER to try to find particular variations of it that may be useful in optimizing it to this particular use. CaSPER is a C++ library for generic constraint solving.

The NValue constraint is a generalization of both AtleastNValue and AtMostNValue, that will be presented later. It constraints the vector $\bar{X}$ of variables so that it uses exactly N different values. It prunes both the finite domain variable N and the vector $\bar{X}$. It must be noticed that since N is a finite domain variable, this constraint can be used to state upper and lower bounds for the cardinality of $\bar{X}$. This is achieved by propagating the following constraint:

$$AtMostNValue(\bar{X}, N) \wedge AtLeastNValue(\bar{X}, N)$$

At first it appears that NValue is fully expressed by the above constraint, but there is a case in which further pruning can be achieved. When $|D(N)| = 2$ but $min(N) + 1 \neq max(N)$ (that is to say that the only two values in the domain of N are not contiguous) then there are cardinalities of $\bar{X}$ which are forbidden by NValue but allowed by both AtMostNValue($\bar{X}, max(N)$) and AtLeastNValue($\bar{X}, min(N)$). In those cases, the extra pruning is provided by propagating a disjunctive constraint involving both AtMostNValue and AtLeastNValue over $\bar{X}$ but with different values for N. Since $|D(N)| = 2$, there are only two possible values that $\bar{X}$ can use, and

Dataset	Size of cover	Boolean	NValue-Based	Speedup (NValue)
Real_data	2	10	1	10
Gen_1	5	3	0.5	6
Gen_2	3	0.5	1	0.5
Gen_3	3	5	2	2.5
Gen_4	6	45	110	0.4
Gen_5	4	12	12	1
Gen_6	4	4	7	0.57
Gen_7	5	15	6	2.5
Gen_8	3	20	2	10
Gen_9	3	7	3	2.33
Gen_10	6	65	5	13
Gen_11	6	70	61	1.14
Gen_12	2	33	20	1.65
Gen_13	3	12	4	3
Gen_14	3	13	2	6.5
Gen_15	3	10	6	1.66
Gen_16	6	150	43	3.48
Gen_17	6	44	32	1.37
Gen_18	2	10	2	5
Gen_19	3	8	6	1.33
Gen_20	4	25	32	0.78
Gen_21	3	6	7	0.85
Gen_22	5	105	38	2.76
Gen_23	3	115	152	0.75
Gen_24	3	235	-	-
Gen_25	10	-	1330	-
Gen_26	7	-	2200	-
Gen_27	6	2552	-	-
Geometric mean				2.06

Table 6.1: Comparison between the Boolean and the NValue-based models.

those are necessarily $min(N)$ and $max(N)$, so it is enough to prohibit any value *between* $min(N)$ and $max(N)$.

Putting it all together, the filtering algorithm for the NValue constraint is schematized as:

$$NValue(\bar{X},N) \equiv \begin{cases} AtLeastNValue(\bar{X},N) \wedge AtMostNValue(\bar{X},N) \\ \\ |D(N)| = 2 \wedge min(N)+1 < max(N) \implies \begin{cases} AtLeastNValue(\bar{X},max(N)) \\ \vee \\ AtMostNValue(\bar{X},min(N)) \end{cases} \end{cases}$$

6.4 The AtMostNValue constraint

Given a vector of variables $\bar{X}$ and a finite domain variable N, the AtMostNValue constraint states that at most N different values will be used by $\bar{X}$. This constraint does not only filter the domains of the variables in $\bar{X}$ but also the minimum value of N. There are many filtering algorithms for this constraint and three have been analyzed in [BHH+06]. The algorithm that proved to be the most efficient in practice makes use of the concept of the *independence number* of a graph related to the vector $\bar{X}$. This number is directly linked to the lower bound of the cardinality of $\bar{X}$, referred to as $card\downarrow(\bar{X})$. Although obtaining this number is an intractable problem, an approximation of it is used to prune the domain of N and of the variables in $\bar{X}$.

6.4.1 Graph theoretic concepts

Given a family of sets $\mathscr{F} = \{S_1,...,S_n\}$ and a graph $G = (V,E)$ with the set of vertices $V = \{v_1,...,v_n\}$ and a set of edges E, G is the *intersection graph* of $\mathscr{F}$ iff

$$\forall i,j < v_i,v_j > \in E \iff S_i \cap S_j \neq \emptyset$$

that is to say that the intersection graph of $\mathscr{F}$ is a graph that has arcs between each pair of sets in $\mathscr{F}$ that share elements. Given a vector of variables $\bar{X}$, the intersection graph of it is denoted by $G_{\bar{X}} = (V,E)$ where $V = \{v_1,...,v_n\}$ and $\forall i,j < v_i,v_j > \in E \iff D(X_i) \cap D(X_j) \neq \emptyset$.

The *neighborhood* of a node v is the set of nodes in the graph that are connected to v through edges. The *degree* of a node v is the size of its neighborhood.

The *independent set* is a set of vertices with no edge in common. Finally the concept that is needed for this constraint is the one of *independence number* of a graph G, denoted $\alpha(G)$. $\alpha(G)$ is the number of vertices in G that belong to an independent set of maximum cardinality. An independent set of the induced graph $G_{\bar{X}}$ corresponds to a set of variables in $\bar{X}$ whose domains do not intersect, meaning that no pair of variables in the set can be instantiated with the same value. From that it is self evident that any full instantiation of $\bar{X}$ will require at least as many different values as $\alpha(G_{\bar{X}})$. Therefore $\alpha(G_{\bar{X}}) \leq card\downarrow(\bar{X})$.

As an example, lets consider the variable vector $\bar{X} =< X_1, ..., X_6 >$ such that:

$$X_1 \in \{2,3\} \; X_2 \in \{3,4\} \; X_3 \in \{1,4,5\}$$
$$X_4 \in \{5,6\} \; X_5 \in \{6,7\} \; X_6 \in \{2,3,7\}$$

Its induced intersection graph ($G_{\bar{X}}$) will have 6 nodes $v_1, ..., v_6$ (one for each variable in $\bar{X}$) and one edge between each node pair whose related variables have domains that intersect. The independence set of maximum cardinality will consist of nodes $\{v_1, v_3, v_6\}$, being $\alpha(G_{\bar{X}})$=3. This example is represented in Figure 6.1

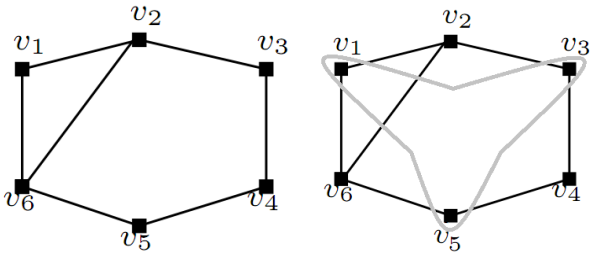

Figure 6.1: An intersection graph and its maximum independent set.

6.4.2 The computational complexity of AtMostNValue

It has been proven [BHH+06] that testing a value for support is NP-complete by reducing 3-SAT to it, and also that enforcing GAC is NP-hard. Pruning $\bar{X}$ is also NP-hard and even computing a tight lower bound on N is no easier. That means that the only alternative is to find approximate solutions to the problem, that is to say that we should be satisfied with a non tight lower bound of $card\downarrow(\bar{X})$. This way some pruning can be achieved without diving into intractability.

6.4.3 A greedy approximate approach for computing the independence number

Here a simple heuristic algorithm for computing a lower bound for $\alpha(G_{\bar{X}})$ is introduced. Called MD for *minimum degree*, the algorithm is schematized in Algorithm 7. This routine systematically removes the vertex v of minimum degree as well as its neighborhood. By proceeding in this fashion, this algorithm finds the size of a large independent set in $G_{\bar{X}}$. It chooses nodes of minimum degree so that eliminating their neighborhood will have the least impact in the size of the resulting graph. This way the process can be repeated many times, resulting in an independence set of large cardinality, whose size is a lower bound for $\alpha(G_{\bar{X}})$. We should keep in mind that $\alpha(G_{\bar{X}})$ constitutes, in turn, a lower bound for $card\downarrow(\bar{X})$, what makes MD useful for pruning both N and $\bar{X}$.

Algorithm 7 MD

```
1: if ( then V = ∅
2:     return 0
3: end if
4: min_deg ← ∞
5: for all v ∈ V do
6:     if degree(v) < min_deg then
7:         min_deg = d(v)
8:         v_min = v
9:     end if
10: end for
11: V ← V \ {neighborhood(v_min) ∪ {v_min}}
12: return 1 + MD(V, E)
```

6.4.4 Pruning N

MD provides us with a lower bound for $\alpha(G_{\bar{X}})$ which is also a lower bound for $card \downarrow (\bar{X})$. For that reason, whenever MD tells that the independence number is greater than $min(N)$, all the values below the result of MD can pruned from N.

6.4.5 Pruning $\bar{X}$

In order to prune $\bar{X}$, observations made in [Bel00] must be considered since they are relevant when using MD to compute $min(N)$. First, let A be a set of variables that form an independent set of the intersection graph, and let us consider a variable X_i from $\bar{X}$ which does not belong to A. If X_i were assigned to a value v outside $\bigcup_{a_j \in A} D(a_j)$, then the minimum number of values required would be greater than $\alpha(G_{\bar{X}})$. Therefore, if MD gives an approximation of $\alpha(G_{\bar{X}})$ which is equal to $max(N)$ we can prune v from X_i [1]. Of course, if the approximation of $\alpha(G_{\bar{X}})$ comes to be strictly greater than $max(N)$, it is the case that the constraint is violated and the solver must backtrack immediately.

6.4.6 Implementation issues

The filtering algorithm for the AtMostNValue was fully implemented in CaSPER, the whole pruning routine is schematized in Algorithm 8.

In [BHH+06] it is stated that an incremental approach to update the intersection graph between executions of the filter was an overhead and it is therefore not considered in this dissertation. In their work, they suggest an implementation in which the intersection graph is never actually created but, instead, intersection checks are computed each time it is needed to know if an edge links two nodes. They base the suggestion on the assumption that the domains of the variables are internally stored as bit vectors by the constraint solver, resulting in very efficient

[1] In [BHH+06] this is achieved through the propagation of the constraint $\forall X_i \in \bar{X}, \exists X_j \in A$ s.t. $X_i = X_j$. Later this is used in a generalization that uses all independent sets. In practice however, this approach is way too costly and it constitutes more of burden than an optimization for the solver. Therefore, taking clarity and efficiency into account, I decided to present this simpler technique instead.

(close to constant time) set intersection operations between domains. However in the case of CaSPER the domains are stored as ordered lists of integers, making the approach impossible. On the other hand, experimental results (Table 6.2) show that an intermediate approach greatly reduces the execution time of the filtering algorithm in the CaSPER system.

Algorithm 8 Filtering algorithm for the AtMostNValue constraint

1: $g_{val} \leftarrow \emptyset$
2: $g_{var} \leftarrow \emptyset$
3: **for all** $X \in \bar{X}$ **do**
4: **if** $|D(X)| = 1$ **then**
5: $g_{val} \leftarrow g_{val} \cup D(X)$
6: $g_{var} \leftarrow g_{var} \cup \{X\}$
7: **end if**
8: **end for**
9: $min(N) \leftarrow max(min(N), |g_{val}|)$ ▷ if $g_{val} > max(N)$ the domain of N will be wiped out and the solver will immediately backtrack
10: $K \leftarrow max(N) - |g_{val}|$
11: $d_{var} \leftarrow \bar{X} \setminus g_{var}$
12: **for all** $X \in \bar{X} \setminus g_{var}$ **do**
13: **if** $D(X) \cap g_{val} \neq \emptyset$ **then**
14: $d_{var} \leftarrow d_{var} \setminus \{X\}$
15: **end if**
16: **end for**
17: **if** $K = 0$ **then**
18: **for all** $X \in \bar{X} \setminus g_{var}$ **do**
19: $D(X) \leftarrow D(X) \cap g_{val}$
20: **end for**
21: **else if** $K = 1$ **then**
22: **for all** $X \in \bar{X} \setminus g_{var}$ **do**
23: $D(X) \leftarrow D(X) \cap (g_{val} \cup \bigcap_{y \in d_{var}} D(y))$
24: **end for**
25: **else**
26: $A \leftarrow \emptyset$
27: **while** $|d_{var}| > 0$ **do**
28: $n \leftarrow |d_{var}|$
29: $min \leftarrow \infty$
30: **for all** $X \in d_{var}$ **do**
31: $degree_X \leftarrow |\{Y | X \neq Y \in d_{var} \wedge D(X) \cap D(Y) \neq \emptyset\}|$
32: **if** $min > degree_X$ **then**
33: $min \leftarrow degree_X$
34: $Y \leftarrow X$
35: **end if**
36: **end for**
37: **end while**
38: $min(N) \leftarrow max(min(N), |A| + |g_{val}|)$
39: **if** $K == |A|$ **then**
40: **for all** $X \in \bar{X}$ **do**
41: $D(X) \leftarrow D(X) \cap (g_{val} \cup \bigcup_{y \in A} D(y))$
42: **end for**
43: **else**
44: Sleep until $K - |A|$ new values are assigned to variables
45: **end if**
46: **end if**

6.4.6.1 Computation of the intersection graph

In Algorithm 7 the node of minimum degree is found many times, each with respect to a graph smaller than the one before. Calculating the degree of a node requires checking if the domain of the variable associated to that node intersects with the domains of the other variables. That means that intersection checks will have to be computed many times over the same sets, which is an unnecessary overhead. Instead, the approach taken here is first to compute the whole intersection graph and then to store it as an *Adjacency list*. This way the graph can be incrementally updated during the computation of the whole independence set.

If there are n variables, computing the intersection graph requires $n(n+1)/2$ test of intersection. Each of those may require at most d equality checks, where d is the size of the domains in $\bar{X}$, amounting to a total complexity in $O(dn^2)$. The first advantage of this approach is that looking for the vertex of minimum degree takes linear time, it just involves checking the size of the adjacency list of each node[2]. Furthermore, the graph can be easily updated for eliminations: when a node is deleted from the graph, only the adjacency list of the nodes which are neighbors to that node have to be updated. By using a specialized set data structure for the list, eliminations are completed in logarithmic time.

6.4.6.2 Intersection graph reduction

To further increase the efficiency of MD, the following technique to reduce the number of nodes in the intersection graph is used. First the set of ground variables (g_{var}) and the corresponding set g_{val} of values assigned to variables in g_{var} need to be computed. Already all values smaller than $|g_{val}|$ can be pruned from N. There are three cases in which MD does not even need to be computed:

1. If $|g_{val}| > max(N)$ there are already too many different values assigned to $\bar{X}$, and immediate backtracking is required.

2. If $|g_{val}| = max(N)$ then $\bar{X}$ is using as many different values as it can, so it is safe to prune all values outside g_{val} from the domain of all non instantiated variables in $\bar{X}$.

3. When $|g_{val}| = max(N) - 1$ it is obvious that only one new value can be used by $\bar{X}$. To find the values that cannot, in this case, be used by $\bar{X}$, d_{var} is defined to be the set of variables whose domain has no intersection with g_{val}, that is $d_{var} = \{X \in \bar{X} | D(X) \cap g_{val} = \emptyset\}$. The domain of any variable $X \in \bar{X}$ can be pruned to $D(X) \cap (g_{val} \cup \bigcap_{y \in d_{var}} D(y))$.

If it is the case that $|g_{val}| < max(N) - 1$ then the lower bound on N needs to be computed through MD and if $MD(\bar{X})$ gives a lower bound of $card \downarrow (\bar{X})$ which is equal to $max(N)$ then some pruning can be achieved as proposed in section 6.4.5.

[2]This can be further improved if the list of nodes were to be maintained ordered by size of the neighborhood, but would add an overhead to the process of node deletion.

Sets	|Universe|	avg |Set|	Original	IG	IG + MD avoidance
200	70	30	3.0	1.3	0.5 secs.
170	70	30	4.4	1.7	0.6 secs.
150	70	30	2.6	1.3	0.5 secs.
200	50	30	3.7	2.1	0.5 secs.
170	50	30	2.0	1.0	0.4 secs.
150	50	30	5.0	2.6	1.0 secs.
Average			3.4 secs.	1.7 secs.	0.6 secs.

Table 6.2: Execution time of variations of the AtMostNValue filtering algorithm.

6.4.6.3 MD avoidance

Computing MD is still somewhat expensive and it should be avoided when possible. It was explained above that computing MD is unnecessary when $|g_{val}| \geq max(N) - 1$, but there is another case in which MD can be avoided without reducing the pruning power of the filter. This technique was developed in the experimental phase of the present work.

Let us suppose that MD over a partial instantiation $I(\bar{X})$ is such that $MD(I(\bar{X})) < max(N)$, meaning that at this point no values from $I(\bar{X})$ can be pruned. Moreover let us consider another instantiation $I'(\bar{X})$ which is equal to $I(\bar{X})$ with the exception that a variable in $I'(\bar{X})$ was assigned to a value outside the set of values to which variables are assigned to in $I(\bar{X})$. This means that in this case $card\downarrow(I(\bar{X}))+1 \leq card\downarrow(I(\bar{X}))$ and since $MD(\bar{X})$ is a good approximation of $card\downarrow(\bar{X})$, it will usually also be the case that $MD(I(\bar{X}))+1 \leq MD(I(\bar{X}))$.

That leads to the conclusion that if $MD(I(\bar{X})) < max(N)$ then MD can be delayed until applied to an instantiation $I''(\bar{X})$ which makes use of $max(N) - MD(I(\bar{X}))$ new values with respect to $I(\bar{X})$. As $MD(\bar{X})$ is not necessarily equal to $card\downarrow(\bar{X})$, it might be the case that its computation is avoided for too long, resulting in a loss of early pruning; however experimental results also shown in Table 6.2 suggest that execution time is greatly reduced by this technique, and that it is generally a considerable improvement.

In the Table 6.2, *Sets* is the number of sets considered, $|Universe|$ is the total number of elements in the universe, $avg\ |Set|$ is the average size the sets. Each group is the average execution time of ten random instances. *Original*, *IG* and *IG+MD Avoidance* stand for the variations of the filter, respectively: the original, the one computing the independence graph and finally one computing independence graph and avoiding the computation of MD. It must be noted that the *Original* version does not work as it is intended by its authors since they assume that the domains of variables are stored as bit vectors, which is not the case for the CaSPER system.

Further experimental analysis of the behavior of this avoidance procedure showed that even when $MD(I(\bar{X})) \neq card\downarrow(I(\bar{X}))$, the difference between $MD(I(\bar{X}))$ and $card\downarrow(I(\bar{X}))$ was usually the same as between $MD(I''(\bar{X}))$ and $card\downarrow(I''(\bar{X}))$ meaning that delaying the computation of *MD* does not cause much loss in the pruning power of the algorithm.

6.5 The AtLeastNValue constraint

This constraint links the variable vector $\bar{X}$ with the finite domain variable N so that $\bar{X}$ must use at least N different values. Enforcing GAC on this constraint prunes not only the domain of the variables in $\bar{X}$ but also that of N, by finding upper bounds of $card(\bar{X})$ in polynomial time.

From [Bel00], it is known that $card\uparrow(\bar{X})$ is the cardinality of the maximal matching of the bipartite graph with one class of vertices representing the variables and the other the values, and where there is an edge between each variable and each one of the values in its domain. It is the same principle behind the most common algorithm for enforcing GAC on the AllDifferent constraint [Rég94]. The propagation procedure for AtLeastNValue studied in this dissertation is derived from the variable-based violation cost for the SoftAllDiff constraint as described in [PRB01]. This violation cost counts the number of variables that need to be reassigned to satisfy the AllDifferent constraint and is thus equal to the difference between the number of variables and $card\uparrow(\bar{X})$.

Since the maximal matching in the bipartite graph will assign as many different values to $\bar{X}$ as it is possible, the maximal matching is indeed the value of $card\uparrow(\bar{X})$. It must be noticed that [PRB01] deals with over-constrained problems and therefore the constraint presented there aims to minimize the violation cost of AllDifferent and will prune all values that do not belong to a maximum matching when possible, which is not the case for AtLeastNValue. Given a variable X and a value v in its domain, v should be pruned only when the assignment of v to X causes the variable violation cost of AllDifferent to be greater than the difference between the number of variables in the vector and the minimum value of N.

6.5.1 Algorithm for the violation cost of the AllDifferent constraint.

Let $\bar{X}$ be a vector of variables, the *value graph* $VG(\bar{X}) = (\bar{X}, D(\bar{X}), E)$ is the bipartite graph such that $(X,v) \in E$ iff $v \in D(\bar{X})$. Let us note by $\mu(G)$ the cardinality of a maximum matching of a graph G. Let $\bar{X}$ be a vector of variables over which the AllDifferent constraint has been imposed. $|\bar{X}| - \mu(VG(\bar{X}))$ is a lower bound of the violation cost of the constraint. The rationale of this is that $\mu(VG(\bar{X}))$ gives the maximum number of different values that can be used by $\bar{X}$, and if subtracted from the number of variables in $\bar{X}$, results in the minimum number of repeated values in $\bar{X}$, which is also the number of assigned values that should change in order to make AllDifferent satisfied (i.e its variable based violation cost). Anyway for AtLeastNValue, it is not the violation cost of AllDifferent what matters, but the minimum number of different values that must be used by $\bar{X}$. This value ends up being exactly $\mu(VG(\bar{X}))$.

6.5.2 Filtering algorithm for the AtLeastNValue.

The maximum size of a matching in a bipartite graph can be computed in polynomial time [AMO93] and the violation cost can be used to prune the domains of the variables in $\bar{X}$. However, as it was stated above, one can avoid using the violation cost and use directly the value $\mu(VG(\bar{X}))$, which tells the maximum number of values that can be used by $\bar{X}$.

Let us note by $\bar{X}_{X=v}$ a vector equal to $\bar{X}$ except that the variable X was assigned to the value v in its domain. If $\mu(VG(\bar{X})) < min(N)$, the constraint can not be satisfied and the solver must immediately backtrack. If $\mu(VG(\bar{X})) \geq min(N)$ one can prune from N all the values above $\mu(VG(\bar{X}))$ and also try for every combination of variables X_i and values v in its domain if $\bar{X}_{iX=v}$ violates the constraint. That is if $\mu(VG(\bar{X}_{X=a})) < min(N)$ then a can be safely pruned from the domain of X_i. After completing that procedure all values in all variables have support in $\bar{X}$ and therefore are is GAC. This algorithm is schematized in Algorithm 9

Algorithm 9 Filtering algorithm for the AtLeastNValue constraint

```
1: max_card ← μ(VG(X̄))
2: max(N) ← min(max(N), max_card)
3: if max_card ≤ max(N) then
4:     for all X ∈ X̄ do
5:         for all v ∈ D(X) do
6:             if μ(VG(X̄_{X=v})) < min(N) then
7:                 D(X) = D(X) \ {v}
8:             end if
9:         end for
10:    end for
11: end if
```

However, this can be implemented much more efficiently as shown in [PRB01] using the same techniques as in the AllDifferent constraint but allowing $|\bar{X}| - N$ variable violations. This optimization was not implemented since it is immaterial in the problem of minimizing the cardinality of $\bar{X}$.

7

Finding all solutions of a minimum set covering

In this chapter, the problem of finding all minimum set coverings is tackled. This task proved to be computationally very demanding and involves finding symmetry free models or symmetry breaking algorithms for those models which present them. On the other hand, the set covering problems related to species differentiation have minimum solutions of very small cardinality, which means that the search space of all minimum solutions is relatively small. Incidentally, this fact permits efficient computation of all minimum solutions. All the algorithms in this chapter are provided with the size of a minimum cover obtained by the application of any of the minimizing models presented earlier. This chapter will conclude with an experimental analysis of the efficiency of each method.

7.1 The Boolean model

The Boolean Model presented in Section 5.1 can be applied to this task. Since the Boolean vector $\bar{X}$ has one element for each set in $\mathscr{S}$ (stating if it is selected or not), there is only one way of representing each covering, what makes this model completely symmetry free. This fact makes it appropriate to find all solutions. The only modifications it requires are the removal of the minimizing parameter in the labeling and the introduction of the extra constraint:

$$\sum_{k \in 1..|\mathscr{S}|} X_k = N$$

being N is a fixed value representing the size of a minimum covering.

7.2 Symmetry breaking algorithms for the NValue-based model

The NValue-based model proved to be the fastest in finding the size of a minimum cover, and because of that it was suspected that it might be also the fastest in finding all minimum solutions. Unfortunately, the model cannot be directly used for that purpose since many repetitions are obtained. For example, let us assume that $\mathscr{U}$ is composed of the three elements $\{u_1, u_2, u_3\}$ and that $\mathscr{S} = \{S_1, S_2, S_3\}$ is such that:

$$S_1 = \{u_2, u_3\} \;\; S_2 = \{u_1, u_3\} \;\; S_3 = \{u_1, u_2\}$$

The tree elements in $\mathscr{U}$ would be represented as the vector $\bar{X} = [X_1, X_2, X_3]$ where $X_1 \in \{2,3\}$, $X_2 \in \{1,3\}$ and $X_3 \in \{1,2\}$. $\bar{C}$ is defined as the set of values used by $\bar{X}$. This configuration allows six different labellings for $\bar{X}$ which use the least number of values and therefore minimize the cardinality of C, namely:

$$\bar{X}_1 = [2,1,1] \;\; \bar{X}_2 = [3,1,1] \;\; \bar{X}_3 = [3,3,2]$$
$$\bar{X}_4 = [2,1,2] \;\; \bar{X}_5 = [3,3,1] \;\; \bar{X}_6 = [2,3,2]$$

but since C is a set, each pair of labellings in the same column represent the same minimum cover. Here there are only two repetitions per solution, but when real datasets are used the number of repetitions is so large that it prevents the enumeration of all solutions.

7.2.1 Sequential accumulation of constraints.

The set of values used in an instantiation of $\bar{X}$ represent the solution $\bar{C} = \{c_1, ..., c_N\}$ and a new $\bar{X}'$ vector is created with another constraint imposed over it to avoid the rediscovery of C as a solution. It is not possible to take any of the c_i in $\bar{C}$ out from the domain of the elements of $\bar{X}'$ since that would prevent the labeler to find a new solution that makes use of some c_i from $\bar{C}$ but not others; the new constraint imposed over $\bar{X}'$ has to assure that the elements of $\bar{C}$ are not used all *together.* Internally, reification is heavily used and the result is the disjunction of a conjunction:

$$\bigvee_{c_i \in \bar{C}} \left(\bigwedge_{X' \in \bar{X}'} X' \neq c_i \right)$$

As more and more new solutions are found, the new vectors have to be constrained to avoid them reappearing, what makes the constraint system more complex in each iteration. The increasing complexity of the system proved to make this method inapplicable to find all solutions in set covering problems, even though it can be successfully used to find a fraction of them.

7.2.2 Relaxed tree accumulation of constraints.

A different approach is to relax the conditions and allow *some* repetitions. After finding the first solution it is possible to enforce the use of at least one different element, what would assure

new solutions. The first solution found, here noted as the *seed*, is used to create a new group of solutions each of which will, in turn, bred the next generation, resulting in a tree hierarchy of solutions.

For instance, let us assume that the *seed* is $\{a,b,c\}$. That permits three descendants, each of which will be the result of forbidding one different subset of $\mathscr{S}$. Since stripping the domain of variables does not increase the complexity of the constraint system in this problem, this approach will result in much simpler CSPs than the previous one. However it does not guarantee that the whole tree will be free of repetitions.

By iterating the process with the new solutions, the whole tree can be built and all the solutions found. It is important to notice that in order to assure algorithm termination, forbidden subsets have to accumulate, that is, if a subset is not available in a node, it can neither be available in any of its descendants.

7.2.2.1 Proof of completeness of the algorithm.

To prove that the algorithm is complete it suffices to show that for any solution $\bar{C}$, there is a path from the root of the tree to $\bar{C}$. This is achieved by Algorithm 10

Algorithm 10 Path to $\bar{C}$

1: $Node \leftarrow Root$
2: **while** $Node \neq \bar{C}$ **do**
3: $\quad Node \leftarrow descendant(Node, Label)$ s.t. $Label \notin \bar{C}$
4: **end while**

Note that the step 3 always succeeds since $\bar{C}$ is a solution and none of its elements have been made unavailable through the path followed. Also note that the same step may involve choosing an alternative at random, what means that there may be many paths leading to $\bar{C}$ and therefore many repetitions in the tree.

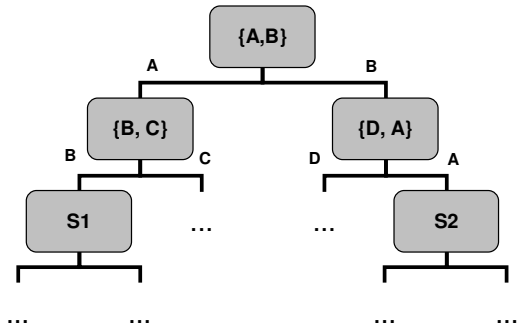

Figure 7.1: Solutions tree

7.2.2.2 Example

. Let us use the solution $\{a,b\}$ as a seed. This leads to two possible new solutions, one taking a from the available subsets in $\mathscr{S}$ and the other taking b. These two new solutions lead, in turn, to other solutions as shown in Figure 7.1, in which for convenience the arcs of the tree are labeled with the identifier of the subset made unavailable.

Under the node $S1$ lye all the solutions that do not use subsets a and b , which is also the case for the node $S2$, so tree $S2$ should be punned as it adds no new solutions. That is due to the fact that the subtree labeled $\{b,c\}$ has all the solutions which do not use subset a and that includes the solutions that make no use of *a and b* present in the subtree $S2$. This observation shows that all branches labeled with the same set as a left sibling of any ancestor can be safely pruned without losing solutions.

This symmetry breaking technique involves less complex constraint systems, but even using tree pruning the number of repetitions proved to be high enough to make it unpractical.

7.2.3 Combined approach.

The sequential approach involves very complex constraint systems and the tree approach is not fully effective in avoiding repetitions, but a combination of the two gives better results. In each node of the tree, it is not necessary to forbid all previously found solutions (as in the sequential approach). By traveling upwards in the tree, it is possible to know all the subsets which where taken out of the domain of the $\bar{C}$ vector. All already found solutions in which one of those subsets appear can not be found again and therefore, constraints to forbid those solutions do not have to be posted in that node. This increases the efficiency but not enough to make it competitive.

7.2.4 Breaking the Symmetries during search

Let us recapitulate on the problem. The NValue constraint allows us to find a minimum set covering but fails to find all of them due to a huge amount of symmetries. In this approach there is one variable for each element of $\mathscr{U}$, and its domain is composed of the identifiers of all sets in the family $\mathscr{S}$ that cover that element. A labeling of $\bar{X}$ that uses the minimum number of elements is therefore a minimum covering. The problem is that there are many encodings of $\bar{X}$ that represent the same covering, therefore there are many labellings representing the same solution. The approach taken here is to accumulate solutions in an external data structure and then use them to disallow the reappearing of the same set of elements in $\bar{X}$ by pruning its domain.

7.2.4.1 Pruning the domain of $\bar{X}$

Let g_{vals} be the set of values used by a partially instantiated $\bar{X}$ and N the size of a minimum cover. The basic idea behind the algorithm (schematized in Algorithm 11) is to wait until $|g_{vals}| = N-1$ and, at that point, to look for all previously found solutions of size N which

are supersets of g_{vals} and prune from $\bar{X}$ the extra value in them. The pruning power of this algorithm is much dependent in the difference between $|\bar{X}|$ and the size of a minimum cover N. If N is near $|\bar{X}|$ pruning will be possible only when almost all variables in $\bar{X}$ are already labeled, giving a very poor performance. In the other extreme case, when N is small (which is the case in the species differentiation problem), it is likely that early pruning will be possible since it should not take long until $N-1$ values are used by partial instantiations of $\bar{X}$. In the worst case that would imply waiting until all but one variable in $\bar{X}$ are labeled, but that case is highly unlikely. Of course that when N is too large there is little hope in finding all solutions to the minimum set covering problem since that would typically mean that there are too many of them.

Algorithm 11 Breaking Symmetries during search in O(N*m)

```
1: g_val ← ∅
2: d_var ← ∅
3: for all X ∈ X̄ do
4:     if |D(X)| = 1 then
5:         g_val ← g_val ∪ D(X)
6:     else
7:         d_var ← d_var ∪ X
8:     end if
9: end for
10: if |g_val| = N − 1 then
11:     for all Sol ∈ Solutions do
12:         if g_val ⊂ Sol then
13:             prune_val ← Sol \ g_val
14:             for all X ∈ d_var do
15:                 D(X) ← D(X) \ prune_val
16:             end for
17:         end if
18:     end for
19: end if
```

To compute set inclusion between the values used in the partially instantiated $\bar{X}$ and each previously found solution, N equality checks must be done. If m solutions were found before, that implies a complexity for the algorithm in $O(N*m)$. To improve this, let us consider other data structures to store the previously found solutions so that the algorithm can be made more efficient.

7.2.4.2 An efficient data structure to store solutions

It is clear that the set inclusion only needs to be computed when the size of g_{val} reaches $N-1$ and also that all previously found solutions are of size N, which is the size of a minimum cover. That means that each solution Sol, has exactly N subsets of it of size $N-1$. By keeping those N subsets of Sol, it is possible to avoid the set inclusion check with g_{val} and only check for equality. Furthermore, a map from each subset to its missing element can be kept so that

whenever a match between g_{val} and a subset of a solution is found, the element that has to be pruned from X can be found in constant time. The problem is that set equality checks may also take linear time and now there are $m * N$ set equality checks to be done. To overcome this, the solutions subsets are kept in a hash table that points to the missing element in each, avoiding in this way the comparison between g_{vals} and each subset of each previously found solution. If subsets of more than one solution are equal, then there will be an entry in the hash table pointing to more than one value.

The computation of the hash of g_{vals} is linear on its size but that hash will point directly to the elements that can be pruned from the domain of X in near constant time. This improved algorithm for symmetry breaking during search is schematized in Algorithm 12

As an example, consider the following set of previously found solutions:

$$Solutions = \{\{1,2,3\},\{2,3,4\},\{1,3,4\}\}$$

which results in the following set of maps between subsets and missing elements:

$$\{1,2\} \to 3 \;\; \{1,3\} \to 2 \;\; \{2,3\} \to 1$$
$$\{2,3\} \to 4 \;\; \{2,4\} \to 3 \;\; \{3,4\} \to 2$$
$$\{1,3\} \to 4 \;\; \{1,4\} \to 3 \;\; \{3,4\} \to 1$$

Keeping in mind that the objective is to find which subsets match with g_{val} to then prune the missing elements from the domain of X, all the entries with the same key can be joined together and mapped to the set of values than can be pruned from g_{val}.

$$\{1,2\} \to \{3\}$$
$$\{1,3\} \to \{2,4\}$$
$$\{1,4\} \to \{3\}$$
$$\{2,3\} \to \{1,4\}$$
$$\{3,4\} \to \{1,2\}$$

This algorithm was implemented in CaSPER and used together with the partial implementation of the NValue constraint.

7.3 Symmetry free Finite Domain model

The previous model required extensive work to break the symmetries. Here a variation of the Count-based from Section 5.2 is presented. The vector $\bar{C}$ that must be labeled will now have a size fixed by the cardinality of a minimum cover and the constraints imposed over the variables

Algorithm 12 Breaking Symmetries during search in O(N)

```
1: g_val ← ∅
2: d_var ← ∅
3: for all X ∈ X̄ do
4:     if |D(X)| = 1 then
5:         g_val ← g_val ∪ D(X)
6:     else
7:         d_var ← d_var ∪ X
8:     end if
9: end for
10: if |g_val| = N − 1 then
11:     prune_vals ← solutions_hash.get(g_val)
12:     for all X ∈ d_var do
13:         D(X) ← D(X) ∩ prune_vals
14:     end for
15: end if
```

in the vector state that for each element of $\mathscr{U}$ it is the case that at least one set identified in $\bar{C}$ contains it. This means that after its labeling, $\bar{C}$ will consist of a set of subset identifiers of $\mathscr{S}$ that represent a minimum cover. Furthermore, symmetries are broken by imposing a strict ordering over the elements of $\bar{C}$, allowing the algorithm to find all solutions. This algorithm is a simplified version of Algorithm 5 The pseudo code is shown in Algorithm 13.

Algorithm 13 Symmetry Free Finite Domain Model

Input:
— $\mathscr{U} = \{u_1, ..., u_n\}$, the universe of elements
— $\mathscr{S} = \{S_1, ..., S_m\}$, the family of subsets of $\mathscr{U}$
— N, the cardinality of a minimum set covering
Output:
— C a cover of minimum cardinality

```
1: C̄ = [C_1, ..., C_N]
2: C_1 < ... < C_N                          ▷ symmetries are broken by imposing a strict ordering
3: for all u ∈ 𝒰 do
4:     E ← {S : S ∈ 𝒮, u_i ∈ S}              ▷ E represents all sets that cover the element u_i
5:     ⋁_{i∈1..N} C_i ∈ E                   ▷ a disjunctive constraint is imposed to assure total covering
6: end for
7: label(C̄)
```

7.4 Analysis of the results

All methods were provided with the size of the minimum covering. Since finding all solutions is much more time consuming, only one third of the datasets were tested. The NValue-Based model coupled with the symmetry breaking filter was run in CaSPER and the rest in Sicstus Prolog. Although this comparison is somewhat unfair, CaSPER is generally much faster than

Dataset	Solutions found	Boolean based	NValue-based	Symmetry Free FD
Real_data	330	905	650	50
Gen_3	217	230	128	25
Gen_6	134	238	503	705
Gen_9	>400	347	>1000	128
Gen_12	12	7	3	20
Gen_15	295	266	335	60
Gen_18	25	7	4	15
Gen_21	291	209	>1000	215
Gen_24	>400	565	>1000	630
Gen_27	>400	>1000	>1000	>1000

Table 7.1: Comparison between each model able to find all minimum solutions

Prolog, what only proves the case that the NValue model could not be adapted to find all solutions efficiently, regardless of the effort put in making it so.

In Table 7.1 a comparison is shown of the time in seconds each of the algorithms took to find *all minimum* covers with respect to each one of the data sets. For some datasets the number of solutions can be too big to compute in reasonable time so the algorithms were stopped after the 400^{th} solution. The maximum time allowed was 1000 seconds.

The NValue-based model seems to work rather well when the number of solutions is very small, but this observation misses the subtleties of the way in which it works. The way in which symmetry breaking was implemented in this model requires $N-1$ values being used by partial instantiations of $\bar{X}$ in order to prune anything. This means that after a solution is found and backtracking rolls $\bar{X}$ back to having no instantiated variables, it can be the case that another path is followed from there that can only end up in the same solution. The algorithm can not realize this until too many variables are instantiated, resulting in a big search space.

The Symmetry Free Finite Domain Model proved to be the most efficient to find all solutions, winning over the Boolean and the NValue-based models. This suggest that the most efficient comprehensive approach to find all minimum solutions of a set covering problem involves using two different models in a complementary manner. First the NValue-based model can be used to find the size of a minimum cover and then the Symmetry Free Finite Domain Model can be fed with that and used to find all minimum covers.

8

Conclusions and further work

By simulating the results of gel electrophoresis experiments over the digested DNA of a set of similar species, the bioninformatics problem of species differentiation was mapped to set covering. In order to provide the maximum number of alternatives to the labs, all minimum set coverings are required to be found. This prevents the use of many standard set covering techniques since most of them aim to near-optimum solutions. For that reason, many constraint programing approaches were developed to tackle both the search for the size of a minimum solution as the set of all minimum solutions.

The best model for minimization made use of the NValue constraint, which was partially implemented in the CaSPER C++ constraint solver. Also much effort was put into developing symmetry breaking algorithms to enable this model to find all solutions, but a symmetry free finite domain model proved to be the most efficient for that task. In turn, this calls for a hybrid approach to find all minimum solutions. The first step is to find the size of a minimum solution using the NValue based minimizing method and the second is to provide its result to the symmetry free finite domain model to find all minimum solutions.

The set covering problem considered here falls inside the class of unicost-set coverings, meaning that the cost of selection of each set is equal. If the actual market price of each enzyme were to be considered together with a valuation of the *inconveniences* that using it generate for a particular lab, a lab dependent cost would be attachable to each enzyme. With that information at hand, the problem could be mapped to multicost set covering, and the optimization problem that would arise would be that of minimizing cost, not cardinality of the covering.

Further work can include comparisons with Integer Programming approaches to find the size of minimum coverings, and a more varied set of benchmarks for testing purposes. A

deeper analysis of the structure of the set covering problems that arise from species differentiation could be useful to provide randomly generated datasets that follow more closely the properties of those generated using real data. Also more databases of DNA of other families of species could be considered when the information becomes available.

As the approach has risen interest in Constraints in Bioinformatics [BBA10], comments received suggest several directions for further work, as it is the case of the *robustness* of a cover. When enzymes digest DNA, the patterns that arise from gel electrophoresis experiments are used to identify the species. However, some coverings may fail to contain enzymes that generate sufficiently distinct patterns for every pair of species. That may give rise to the possibility of human error when reading the results. The probability of human error can be approximated using a measure on how different the patterns are when each enzyme is used over each specimen pair. Since coverings contain more than one enzyme, which are applied separately to the DNA sequences, the total probability of human error for each species pair should be measured as the multiplication of the probabilities of error to which each enzyme in the cover gives rise when applied to that species pair. A constraint that forbids coverings with a probability of error above certain threshold for each species pair would not be linear (making linear programming methods inappropriate), but can easily be modeled through constraint programming and can also be studied in further work.

Bibliography

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, February 1993.

[BBA10] David Buezas, Pedro Barahona, and João Almeida. Minimizing set of enzymes to differentiate species. In *Proceedings of WCB 2010 Workshop on Constraint Based Methods for Bioinformatics*, pages 2 – 8, July 2010.

[BC96a] E. Balas and M.C. Carrera. A dynamic subgradient-based branch and bound procedure for set covering. *Operations Research*, 44:875 – 890, 1996.

[BC96b] J. E. Beasley and P. C. Chu. A genetic algorithm for the set covering problem. *European Journal of Operational Research*, 94(2):392 – 404, 1996.

[BCCN96] E. Balas, S. Ceria, G. Cornuéjols, and N. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19(1):1–9, July 1996.

[BCFA99] Esteve-Zarzoso B, Belloch C, Uruburu F, and Querol A. Identification of yeasts by RFLP analysis of the 5.8s rRNA gene and the two ribosomal internal transcribed spacers. *International Journal of Systematic Bacteriology*, 49:329–337, 1999.

[Bea87] J. E. Beasley1. An algorithm for set covering problem. *European Journal of Operational Research*, 31(1):85 – 93, 1987.

[Bea90] J. E. Beasley. Or-library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

[Bel00] Nicolas Beldiceanu. Pruning for the minimum constraint family and for the number of distinct values constraint family. In *In Proceedings CP 01*, pages 211–224. Springer, 2000.

[BHH+06] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Filtering algorithms for the NValue constraint. *Constraints*, 11(4):271–293, 2006.

[BHIK00] Endre Boros, Peter L. Hammer, Toshihide Ibaraki, and Alexander Kogan. Logical analysis of numerical data. *Mathematical Programming*, 79:163–190, 2000.

[BM06] K. Boundy-Mills. The yeast handbook, biodiversity and ecophysiology of yeasts. pages 67–100, 2006.

[BPY00] J. Barnett, R. Payne, and D. Yarrow. *Yeasts, characteristics and identification*. Cambridge University Press, 3 edition, 2000.

[CNS95] Sebastian Ceria, Paolo Nobili, and Antonio Sassano. A lagrangian-based heuristic for large-scale set covering problems. *Mathematical Programming*, 81:215–228, 1995.

[DP08] Gouwanda Darwin and S. G. Ponnambalaim. Evolutionary search techniques to solve set covering problems. *World Academy of Science, Engineering and Technology*, 39, 2008.

[Ere99] Anton V. Eremeev. A genetic algorithm with a non-binary representation for the set covering problem. In *Proceedings of Symposium on Operations Research*, pages 175–181. Springer-Verlag, 1999.

[Gli99] Francois Glineur. Interior-point methods for linear programming: A guided tour, 1999.

[GLS81] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169–197, 1981. 10.1007/BF02579273.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[Kel06] Jonathan A. Kelner. A randomized polynomial-time simplex algorithm for linear programming. In *In STOC*, pages 51–60, 2006.

[KR98] C.P. Kurtzman and C.J. Robnett. Identification and phylogeny of ascomycetous yeasts from analysis of nuclear large subunit (26s) ribosomal DNA partial sequences. *Antonie van Leeuwenhoek*, 73:331–371, 1998.

[LNF95] Antonio L., Lorena N., and Lopes F.B. A surrogate heuristic for set covering problems. *Location Science*, 3:62–62(1), May 1995.

[MMS01] Mohammad Mahdian, Evangelos Markakis, and Amin Saberi. Greedy facility location algorithms analyzed using dual fitting with factor-revealing lp. *Journal of the ACM*, 50:127–137, 2001.

[Mus06] Nysret Musliu. Local search algorithm for unicost set covering problem. pages 302–311. 2006.

[NM65] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4):308–313, 1965.

[PJN07] Raspor P, Zupan J, and Cadez N. Validation of yeast identification by in silico RFLP. *Journal of Rapid Methods and Automation in Microbiology*, 15:267–281, 2007.

[PRB01] Thierry Petit, Jean-Charles Regin, and Christian Bessiere. Specific filtering algorithms for over-constrained problems, 2001.

[R94] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 362–367, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[Rég94] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI*, pages 362–367, 1994.

[SFFST02] G. Scorzetti, J.W. Fell, A. Fonseca, and A. Statzell-Tallman. Systematics of basidiomycetous yeasts: a comparison of large subunit d1/d2 and internal transcribed spacer rDNA regions. *FEMS yeast research*, 2:495–517, 2002.

[SH00] Thomas Stützle and Holger H. Hoos. Max-min ant system. *Future Generation Comp. Syst.*, 16(8):889–914, 2000.

[SM98] O. Schmidt and U. Moreth. Genetic studies on house rot fungi and a rapid diagnosis. *European Journal of Wood and Wood Products*, 56(6):421–425, 1998.

[VRDV+92] M. Vaneechoutte, R. Rossau, P. De Vos, M. Gillis, D. Janssens, et al. Rapid identification of bacteria of the Comamonadaceae with amplified ribosomal DNA restriction analysis (ARDRA). *FEMS Microbiol Lett*, 72:227–233, 1992.

[WBLT90] T. White, T. Bruns, S. Lee, and J. Taylor. Amplification and direct sequencing of fungal ribosomal RNA genes for phylogenetics. pages 315–322, 1990.

[WIR+08] Wei W, Lee IM, Davis RE, Suo X, and Zhao Y. Automated RFLP pattern comparison and similarity coefficient calculation for rapid delineation of new and distinct phytoplasma 16sr subgroup lineages. *International Journal of Systematic and Evolutionary Microbiology*, 58:2368–2377, 2008.

[WR93] Anthony Wren and Jean-Marc Rousseau. Bus driver scheduling - an overview, 1993.

[Yar98] D. Yarrow. The yeasts, a taxonomic study. methods for the isolation, maintenance and identification of yeasts. pages 148–152, 1998.

[YKI03] Mutsunori Yagiura, Masahiro KISHIDA, and Toshihide Ibaraki. A 3-flip neighborhood local search for the set covering problem, 2003.