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

Master's Thesis in Computational Logic

Value Orderings based on Solution Counting

by

Jean Christoph Jung

Supervision
Prof. Pedro Barahona
George Katsirelos

Lisbon
September 30, 2008

Declaration

I hereby declare that I wrote this thesis by myself, only with the help of the referenced literature.

Jean Christoph Jung

Acknowledgements

In autumn 2007 I spent three months as a visitor at NICTA, Sydney. In this time my interest for the area of constraint programming was sparked. When I left, we continued the collaboration in two directions: first, we published a paper about compiling DNNF theories to SAT, and second, in the present thesis about solution counting and value ordering heuristics. I want to thank Toby Walsh and George Katsirelos, my supervisors from NICTA, for the possibility to work with them and for the support they gave me during my stay in Sydney and during creation of the thesis.

I owe thanks to Prof. Pedro Barahona who supervised my work here in Lisbon. Special thanks go to Marco Correia whose CSP solver I used and who supported me extraordinarily during my implementation. I want to thank also Valentin Mayer-Eichberger for both moral and scientific assistance.

Abstract

Constraint satisfaction problems are typically solved using backtracking search. One key issue for backtracking is the selection of a good value, but comparably little effort was dedicated to this topic . Another important aspect of constraint programming besides looking for one solution is counting solutions either in the whole problem or in single constraints. Recent works give first ideas how to exploit the advances in solution counting in designing better value ordering heuristics. With this thesis we want to elaborate these in more depth.

We intend to make two main contributions. First, we provide a novel approach for solution counting in individual constraints, which results in a different view on counting in `regular` constraints and in counting algorithms for `extensional` and `grammar` constraints. Second, we consider the problem of a good value selection from different perspectives and derive several value ordering heuristics based on solution counting. We evaluate both the counting and the heuristics by solving rostering problems modelled with `grammar` and `regular` constraints.

Contents

1 Introduction

1.1 Motivation

There are two possible points of view on solution counting: either we aim to determine the number of solutions of the whole proplem or we count solutions in single constraints. Both approaches have applications in belief revision [Dar01] or value ordering heuristics [ZP07, HKBM07, KDG04], but in this thesis we will restrict our attention to counting solutions in individual constraints.

The notion of solution counting in constraints captures both calculating the number of tuples that satisfy the constraint and determining the number of satisfying tuples when one variable is fixed. In general the counting problem is harder than pure constraint propagation, since counting in our sense performs also propagation. Of particular interest are counting algorithms for global constraints, because they contain more information about the structure of the problem. Recent works proposed algorithms for counting, for instance in `regular` and `alldifferent` constraints [ZP07]. In this thesis, we count solutions in `regular`, `extensional`, and `grammar` constraints by translating the constraints to logically equivalent formulas in `sd-DNNF`, a specific class of propositional formulas, and benefit from a linear model counting algorithm in this class.

Solution counting can, in a sense, be considered as a look-ahead technique. Forward checking, for example, instantiates a variable X_i and observes the domains of the variables adjacent to X_i in the constraint graph. Solution counting provides us with information about how many models a constraint will have after a certain assignment. On the one hand, the techniques are incomparable, because solution counting does not give us any concrete information about prunings and FC might detect some prunings, but does not foresee the changes regarding the constraint. On the other hand, both provide information about the future problem; either about the domains or about the constraints. Frost and Dechter [FD95] proposed value orderings that exploit information obtained from forward checking, so it seems obvious that we can define good value orderings based on solution counting.

Our motivation is that, if a value ordering selects more likely the correct value for a variable, we can substantially decrease the search costs for finding a solution. As an example consider a constraint with 100 supports such that $X = 4$ is true in 90 of them. Intuitively, it is likely that $X = 4$ is also true in a solution. Based on this and several other observations we formulate and justify some new heuristics.

The thesis is structured as follows: In the rest of Chapter 1 we introduce the notions and notations that we are using throughout the work. Further we provide some background knowledge of backtrack search and global constraints. In Chapters 2 and 3 we present our main contributions, namely our approach to solution counting and new value ordering heuristics. In Chapter 4 we evaluate the heuristics by solving rostering problems. In the last chapter we discuss our approach, conclude and point to future work.

1.2 Constraint Satisfaction Problems

In this section we provide all the definitions and concepts of constraint satisfaction problems that are necessary in this dissertation. We illustrate the basic definitions on some examples.

A *constraint satisfaction problem (CSP)* is a tuple $\langle V, \mathcal{D}, \mathcal{C} \rangle$ consisting of a set of variables $V = \{X_1, X_2, \ldots X_n\}$, corresponding finite domains $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ and a set of constraints $\mathcal{C}$. The domain D_i of a variable X_i is the set of allowed values for the variable and we denote domain elements with lowercase letters. A *constraint* $C \in \mathcal{C}$ is a relation on the domains of a ordered set of variables $\{X_1, X_2, \ldots, X_k\}$, written as subset of the Cartesian product of the domains $C \subseteq D_1 \times \ldots \times D_k$. The variables a constraint C operates on are called *scope* of C and written *scope*(C). The *arity* of a constraint is $|scope(C)| = k$. Conversely, the number of constraints that have variable X in their scope is called the *degree* of X. A tuple $u = (u_1, \ldots, u_k)$ *satisfies* a constraint C if $u \in C$, otherwise it *violates* the constraint. Every tuple $u \in C$ is called a *support* of C. We define the notion of an *assignment* v to be a set $\{X_1 = d_1, \ldots, X_k = d_k\}$ that assigns values to some variables. We call the assignment *partial* if it does not cover all variables.

A *solution* of a CSP is an assignment of values to all variables that satisfies all constraints. We call a CSP *satisfiable* if it has at least one solution and *unsatisfiable* otherwise. A (partial) assignment v is *inconsistent* if a polynomial time inference mechanism (called *propagator*) can detect that this (partial) assignment can not be part of any solution. For a CSP P we define a hypergraph G_P. Every variable of the CSP corresponds to a vertex in the hypergraph and for every constraint with scope $\{X_1, \ldots, X_k\}$ there exists a hyperedge between the vertices corresponding to $X_1, \ldots X_k$.

Constraints can be specified *extensionally* or *intensionally*. An extensional constraint is just given by the set of allowed or forbidden tuples and often called table constraint (see Example 1.1). In the future we will refer to a table constraint also with `extensional` constraint. Intensional constraints impose a special structure on the variables in their scope, e.g., the `alldifferent` constraint [Rég94] restricts the variables in its scope to take pairwise distinct values. Example 1.2 shows an example of a constraint defined intensionally.

Example 1.1. *Consider variables X_1, X_2, X_3, X_4 with equal domains $D_i = \{1,2,3,4\}$. Then we can specify a 4-ary constraint by explicitly enumerating all allowed tuples. We can represent them as a table:*

X_1	X_2	X_3	X_4
1	1	1	2
2	1	3	4
3	2	2	3
2	3	4	1
2	4	1	1

Table 1.1: Extensional specification of a constraint.

Example 1.2. *Let X_1, X_2, X_3, X_4 be variables with $D_1 = D_2 = D_3 = \{1,2,3\}$ and $D_4 = \{1,2,3,4\}$. The constraint $C_1 =$ `alldifferent`(X_1, X_2, X_3, X_4) constrains the variables to take different values, i.e., the assignment $X_1 = 1, X_2 = 2, X_3 = 3, X_4 = 4$ would satisfy C, whereas the assignment $X_1 = X_4 = 1, X_2 = X_3 = 2$ would not satisfy the constraint.*

The distinction between binary constraints (arity 2) and non-binary constraints (arity greater than 2) is often made in the literature. Every non-binary CSP can be reformulated as constraint satisfaction problem using only binary constraints. This is a well-known result and there exist various techniques for the translation [DP89, BCvBW02]. However, there are constraints with no fixed arity, like `alldifferent`, which are called *global constraints*, because they can impose restrictions on arbitrarily many variables and hence impose a global structure on them. Constraints with fixed arity are called *primitive* constraints. We will see later (Section 1.4) that global constraints play a central role in the area of constraint programming.

1.3 Solving Constraint Satisfaction Problems

The objective of constraint programming is to solve CSPs, i.e., to find an assignment of values to all the variables that satisfies all constraints. In general, it can be easily shown that solving CSPs is NP-complete [GJ79] by reduction from the SAT problem. Evidently, the SAT problem can be written as CSP: Propositional variables map to finite domain variables with domain $\{0,1\}$ and each clause is mapped to a constraint that allows only tuples that would evaluate the clause to *true*. Now, every solution to the CSP is also a solution to the SAT problem, thus CSP is NP-hard. On the other hand, it is also in NP because we can construct a nondeterministic Turing machine that first "guesses" an assignment and then approves that it satisfies all constraints in polynomial time. Having established NP-hardness and containment in NP we conclude that CSP is

NP-complete. So we cannot expect a general algorithm which solves all classes of CSPs in acceptable (polynomial) time, unless P=NP.

On the contrary, different algorithms for solving CSPs have been proposed. We can identify several classes like backtracking, constraint propagation, local search, structure driven search and bucket elimination algorithms. Most local search algorithms are based on flipping variables such that as few as possible constraints are unsatisfied. A prominent example for local search is GSAT [SLM92] which solves SAT problems, but also in pure CSP settings considerable effort was made [FPDN05, vHM05]. Structure driven search algorithms rely on the special, in most of the cases *treelike*, structure of the problem, see for instance [MF85] or, more recently, [CJS08]. Bucket elimination [Dec99] is a general algorithm that can also be used to solve CSPs. However, here we deal only with the notion of backtrack search and propagation.

1.3.1 Backtrack Search

The idea of backtrack search is to explore the search space systematically with the following approach. The procedure starts with an empty variable assignment and assigns values to the variables one by one until all variables got a value. When all variables in the scope of a constraint are instantiated, the algorithm checks whether the assignment satisfies the constraint. If so, it continues with instantiating more variables, otherwise *backtracking* is performed, i.e., the algorithm goes back to the last variable that has a value which was not yet tried, assigns this value and proceeds with the other variables. If all variables are instantiated the assignment is a solution to the CSP. If the algorithm can not instantiate all variables without failure, the CSP is unsatisfiable and the algorithm returns failure. Algorithm 1.1 shows this procedure. This algorithm exhibits exponential time worst case behaviour, because it systematically enumerates all possible assignments in the worst case. On the other hand it requires only linear space.

Several crucial points are left open in Algorithm 1.1. Spending more efforts on them may tremendously speed up the algorithm, although still being worst case exponential. These points are:

- Line 3: the selection of the next variable X_i, the *variable ordering*,
- Line 4: the order obeying to which the values of D_i are enumerated, the *value ordering*,
- Line 6: we can perform even more work before instantiating the next variable, a family of techniques that are collectively referred to as *look-ahead*,
- Line 9: sometimes we can jump back further instead of only one step, or learn from the failures committed. We call these techniques *look-back* strategies.

```
Algorithm: backtrack($P, v$)
input  : A CSP $P$ with variables $X_1, \ldots, X_k$ and a partial assignment $v$
output: A solution to the CSP or failure
1 if all variables have an value assigned then
2     return $v$;
3 $X_i \leftarrow$ unassigned variable from $X_1, \ldots, X_k$;
4 foreach $d$ in $D_i$ do
5     $v \leftarrow v \cup \{X_i \leftarrow d\}$;
6     if the (partial) assignment $v$ is consistent then
7         if backtrack($P, v$) succeeds then return solution;
8     $v \leftarrow v \setminus \{X_i \leftarrow d\}$;
9 return failure;
```

Algorithm 1.1: Backtrack search.

Varying each of these parts yields new variants of backtracking exhibiting different, possibly better behavior. Look-back strategies try to jump back further than one step as in the original algorithm (for instance backjumping and several variations [CvB01]) or try to avoid some consistency checks (e.g. backmarking [Gas77]). For an extensive overview over these techniques see [Fro97]. Look-ahead techniques try to prune values from the domains of variables that are not possible anymore given the partial assignment so far. We do not consider look-back techniques here, but rather give a short introduction into look-ahead and constraint propagation, respectively. More important for our work are variable and value orderings (Sections 1.3.3 and 1.3.4).

1.3.2 Look-ahead Techniques

One problem of the backtracking algorithm proposed so far is that inconsistencies are identified very late. This causes the algorithm to explore more of the search tree than necessary, i.e., if a partial assignment logically implies empty domains for not yet assigned variables, it is redundant to search further, because the current partial assignment cannot be extended to a solution. On the other hand, it might be that assigning a value to a variable implies that some values of other variables can be removed from their domains, hence it is not necessary to test these choices later on in search. If we include these inferences into search, we interleave reasoning with search. The reasoning takes place locally on the level of the constraints (in contrast to global reasoning on the level of the whole CSP). The process of local reasoning is called *(constraint) propagation*, a procedure that performs it *propagator* or *local filter*. Example 1.3 illustrates the idea.

Example 1.3. *Let X_1, X_2, X_3 be variables of a CSP with constraints $C_1 = (X_1 \neq X_2)$, $C_2 = (X_2 \neq X_3)$ and $C_3 = (X_1 \neq X_3)$. Further, assume domains $D_1 = \{1, 2, 3\}$, $D_2 = \{1, 2\}$ and $D_3 = \{1, 2\}$. Suppose we use the backtracking algorithm from Figure 1.1 with a variable ordering choosing X_1, X_2 and X_3 in this order and the value ordering selecting 1 before 2 and 3. In the first step, the algorithm would choose variable X_1 with value 1 and would then proceed with extending the partial assignment with values for X_2 and X_3. However, direct implications of $X_1 = 1$ are $X_2 = 2$ (by C_1) and $X_3 = 2$ (by C_3), so no more search is necessary. Trying to infer more, we can detect even the failure in C_2 because $X_2 = 2 = X_3$.*

The example shows how logical local reasoning can enhance the power of search and prune effectively the search tree. The action of removing inconsistent values from domains is referred to as *(domain) pruning*.

In practice, it has to be considered how much overhead by reasoning is worth doing at every point of search, because it might be expensive. In fact, the extreme case would be that we can solve the CSP without any search but only by logical reasoning, but also this is not possible in polynomial time in general. We concentrate here on two basic notions, namely *forward checking* and *generalized arc-consistency* and choose the latter one in our setting.

Assume the backtracking algorithm chose $X_i = v$ to be the next assignment. Forward checking (FC) [HE80] looks for all variables $X_{k_1}, \ldots, X_{k_m}$ that appear together with X_i in the scope of some constraint. FC computes the consistent values for these variables assuming the extended assignment. In Example 1.3 this process matches to the first implication, namely $X_1 = 1$ implies $X_2 = 2 \wedge X_3 = 2$. However, FC would not detect the failure in that example, because it is not a *direct* implication, but only implied by an intermediate step. In general, variables that are not connected with the currently elected variable are not affected by forward checking. The changes to be made compared to Algorithm 1.1 are minor: The consistency checks in line 6 can be omitted and be anticipated by forward checking the extended partial assignment. Note that for some problems the normal backtracking algorithm can outperform FC, because the checks that are made do not effect any pruning [BG95]. For example, take a run of backtracking which finds the solution in the very first branch of the search tree. In this case FC does not affect the selections backtracking would make and hence would perform unnecessary consistency checks. Bacchus and Grove [BG95] investigated also the relationship between FC and look-back techniques like backmarking and backjumping in depth. Even in the class of forward checking algorithms there are several grades that differ with respect to the extent of look-ahead they perform [BMFL02].

As we have seen, forward checking would infer only the first implication in Example 1.3. To conclude also the second implication we have to use a stronger notion instead of FC. After introducing the notion of generalized arc-consistency in Definition 1.4, we extend Examples 1.1 and 1.2 from the previous section to illustrate the idea.

Definition 1.4 (Generalized Arc-Consistency).
A constraint C with scope $\{X_1, \dots, X_k\}$ is called generalized arc-consistent (GAC) iff for every variable X_i and every value $d \in D_i$ there are values $d_1, \dots, d_{i-1}, d_{i+1}, \dots, d_k$ such that $(d_1, \dots, d_{i-1}, d, d_{i+1}, \dots, d_k)$ satisfies C. If C is binary we call this property arc-consistency. A CSP is called generalized arc-consistent iff every constraint is generalized arc-consistent.

Example 1.5. *Recall the table constraint over variables X_1, X_2, X_3, X_4 from Example 1.1:*

X_1	X_2	X_3	X_4
1	1	1	2
2	1	3	4
3	2	2	3
2	3	4	1
2	4	1	1

Only the values 1, 2 *and* 3 *occur in the column for* X_1. *Having a domain* $D_1 = \{1, 2, 3, 4\}$, *obviously* $X_4 = 4$ *is inconsistent, because this constraint has no support for value* 4. *An algorithm that maintains GAC on this instance would prune* 4 *from* D_1.

Example 1.6. *Assume a constraint* `alldifferent`(X_1, X_2, X_3, X_4) *with* $D_1 = D_2 = D_3 = \{1, 2, 3\}$ *and* $D_4 = \{1, 2, 3, 4\}$. *We observe that the variables* X_1, X_2, X_3 *all have the domain* $\{1, 2, 3\}$. *Since each of these variables has to take a value and all have to be distinct, we conclude that no other variable can take either of the values* 1, 2 *and* 3. *Thus, maintaining generalized arc-consistency would remove* 1, 2 *and* 3 *from* D_4.

We can see that maintaining GAC achieves more domain prunings than forward checking. On the other hand, the algorithms for maintaining GAC involve more computational overhead. For example, achieving GAC on the `alldifferent` constraint is in $O(dn\sqrt{n})$, where n is the number of variables and d the largest domain size [Rég94]. For extensional constraints several algorithms were proposed that make use of sophisticated data structures in order to perform as few checks as possible [BC93, HDT92, CY06]. However, the overhead in comparison to FC seems to pay off, since maintaining arc-consistency algorithms outperform forward checking algorithms on hard problems [BR96].

1.3.3 Variable Orderings

The impact of an appropriate variable ordering has been recognized early. Beginning with the famous article by Haralick and Elliott [HE80] a lot of research has been done to find good orderings. Being "good" is specific to the CSP under consideration, i.e., a good ordering for one problem is not necessarily well suited for all problems. Nevertheless, there are some general principles underlying all the heuristics.

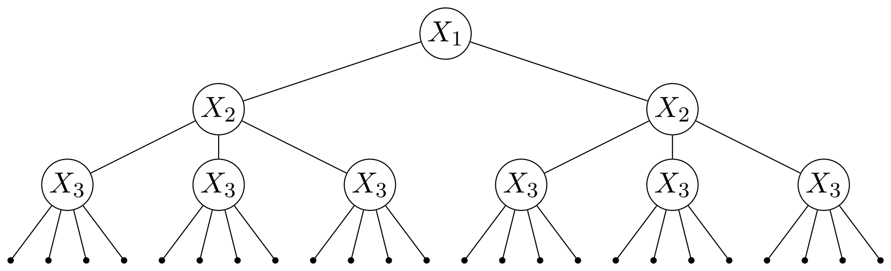

Figure 1.1: Search tree for $X_1 \prec X_2 \prec X_3$.

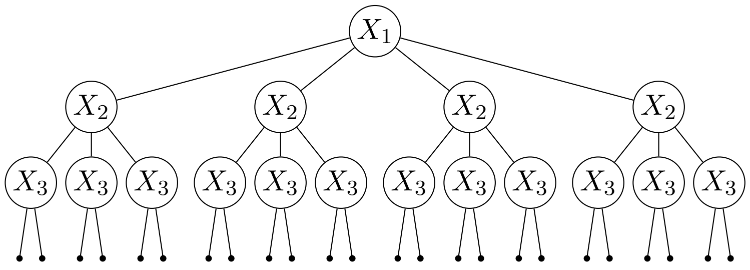
X_1
X_2
X_2
X_2
X_2
X_3 X_3 X_3 X_3 X_3 X_3 X_3 X_3 X_3 X_3 X_3 X_3

Figure 1.2: Search tree for $X_3 \prec X_2 \prec X_1$.

In the following we denote by $X_i \prec X_j$ that the variable ordering chooses X_i before X_j. We consider first a classical example to show the impact of the variable ordering. Assume a CSP with 3 variables X_1, X_2, X_3 with respective domain sizes 2, 3 and 4. Then the search tree explored by backtracking changes its size with changing variable orderings. Figures 1.1 and 1.2 illustrate the different sizes with respect to the orderings $X_1 \prec X_2 \prec X_3$ and $X_3 \prec X_2 \prec X_1$. We observe that the search tree has 9 (inner) nodes for the first ordering and 17 for the second. If look-ahead is desired after assigning a value, we should try to minimize the inner nodes, because look-ahead is performed exactly there. Also note that the outdegree of nodes in high levels (close to the root) has the most impact on the size of the search tree. So it is very important to make good choices in the beginning. To adhere this principle it is reasonable to do some work before the actual search to find better initial choices.

Basically, we can distinguish between *static* and *dynamic* variable orderings. Static means that the ordering is fixed before search, whereas dynamic orderings can change during search. In particular, there may be different variable orderings on different branches of the search tree. Dynamic orderings are usually more flexible than static ones, but it may be that for certain classes of problems a static ordering outperforms all currently known dynamic orderings.

In the following we describe some variable orderings (we also call them variable ordering *heuristics* or just heuristics) and discuss their justifications. Most of them try to pursue the *fail-first* principle [HE80], which can be expressed in various forms, for instance

to succeed fast, try first where you are most likely to fail

or

try to solve first the hardest sub-problems

or, as in the original paper,

try to minimize the expected depth of failure.

The first formulation seems counter-intuitive at first sight, because actually we do not want to fail, but find a solution. Yet the right intuition is that we want to discover failures as early as possible. The fail-first principle was re-investigated twice in [SG98] and [BPW04]. They reject the original fail-first principle by demonstrating that failing early at any price might decrease the average depth for failures on the one hand, but on the other increase the branching factor in search (thus the overall costs). They refuted to follow the principle at any cost; yet the weaker variant – detect failures early in search – is still valid.

Minimum domain ordering

The minimum domain ordering heuristic orders the variables increasingly with respect to their domain size. This heuristic can be justified by the following probabilistic reasoning [SG98]: We assume the probability for an assignment to fail being a constant p. The probability that the assignment for a specific variable X_i fails is the product of this probability over all values in D_i:

$$P(\text{assignment to } X_i \text{ fails}) = \prod_{d \in D_i} p = p^{|D_i|} \tag{1.1}$$

To maximize this (following fail-first) we have to choose the variable with the smallest domain. We refer with dom to this heuristic.

Maximum degree ordering

Another measure of "hardness" of a variable is the number of constraints it is involved in. Intuitively, the more constraints restrict the variable the harder it is to find a value for this variable. Hence, a variable ordering that chooses the variable with the maximum current degree was proposed in [BR96]. We call this heuristic deg.

Domain over degree

Experiments showed that deg works good on sparse problems, whereas it is outperformed by dom on dense problems. We can combine the strengths of the two and construct a heuristic which chooses the variable that minimizes the quotient of domain size and variable degree [BR96]. We call this heuristic dom/deg. The intuition behind dom/deg is to follow both dom (the smaller the domain size the smaller is also the quotient) and deg (the higher the degree of a variable the smaller is the quotient).

Weighted degree heuristics

Another approach to finding hard sub-problems is to learn from past failures that occurred in search. In particular, we can identify constraints that fail more often than others. Variables involved in these should be selected early to detect possible failures as soon as possible. We attach a *weight* to each constraint [BHLS04]. The weight is initialized to 1 for all constraints. Every time a constraint fails its weight is increased by a certain number, 1 in the simplest case. The heuristic Boussemart et al. [BHLS04] propose, called wdeg, selects the variable X_i with the highest *weighted degree* $wdeg(X_i)$.

$$wdeg(X_i) = \sum \{weight(C) \mid X_i \in scope(C) \wedge |Vars_f(C)| > 1\} \tag{1.2}$$

where $Vars_f(C)$ denotes the number of non-ground variables in the scope of C. We can apply the idea of introducing a quotient between domain size and degree also here, yielding dom/wdeg, a heuristic that minimizes the quotient of domain size and weighted degree. Note that the weighted degree coincides with the normal degree if all constraints have weight 1. The heuristic can be improved on by running some probing before the actual search, i.e., performing some (possibly random) search with restarts to update the weights such that they identify the hard parts of the problem [GW07]. After these initial runs a final search run directed by the computed weights is performed.

Ties

In practice, heuristics often end up with ties, e.g., many variables have the same minimum domain size. Instead of choosing one of these variables arbitrarily, we can break ties using another heuristic. In this way we obtain for example the Brélaz heuristic (Bz) that breaks ties of dom by selecting the variable with the highest degree [Bré79].

Other orderings

Two more strategies which attracted recently attention, but are not of particular importance for this dissertation are *impact-based* variable ordering and *neighborhood-based* variable ordering heuristics. The first one, coming from integer programming, is based on computing *impacts* of assignments which measure its search space reduction [Ref04].

It looks for the variable with the highest impact to reduce the search space most. Also this strategy can be improved on by computing an approximation of the impacts before search (for example by doing several random runs and restarts). The second heuristic is not taking into account only one variable but also its neighborhood, i.e., the variables connected to it in the constraint graph [BCS01]. The intuition is that not one variable alone is hard to satisfy, but most of the time a group of connected variables. Considering also the neighborhood tries to exploit this property. This heuristic may in principle be combined with most of the other heuristics described here.

1.3.4 Value Orderings

Whereas considerable effort was made to develop sophisticated look-ahead, look-back and variable ordering strategies, only a few value orderings were proposed. The reason for this is not quite clear, because looking at the proof for containment in NP the influence of value ordering becomes self-evident. Assume the value ordering heuristic was able to make good choices for the values. In the optimal case, i.e., the heuristic chooses always the right value, the algorithm would find a solution in a backtrack free manner, namely in the first branch of search. In relaxation of this we can say that making good choices for the values might help a lot in finding a solution fast.

In general, value orderings do not help in proving unsatisfiabily[1] of a problem faster or to enumerate all solutions, because in either case the search tree as to be explored exhaustively. Only in settings where we are interested in one solution of a satisfiable problem a good value ordering is helpful.

Some work has been done though for value orderings with look-ahead [FD95]. In particular, before choosing a value, forward-checking is performed for every domain element and the values are selected depending on the domain prunings they caused. Two heuristics Frost and Dechter [FD95] proposed are:

- choose the value that causes the least pruning from future variables' domain, and
- to choose the value that causes as few as possible small domains for future variables (along with different methods of tie-breaking).

Other approaches try to estimate the solutions in the sub-space, for example using some kind of relaxation, and choose the value that admits the most solutions [DP87].

They all follow the general principle of value orderings. Conversely to variable orderings, where in general a variable is selected that is *most constrained*, a good value ordering chooses the value that constrains the future problem *least*. This is reasonable, because after all we want to find a solution and this is more likely if variables are instantiated with values that leave most possibilities for future variables.

[1]This is not entirely true. Dechter and Frost [FD95] show that in combination with back-jumping the size of the search tree is affected by the value ordering (Theorem 1).

Recent approaches suggest the usage of solution counting, be it exact or approximated to select a value [ZP07, KDG04, HKBM07]. We will describe these works in more detail in Chapter 3.

1.4 Global Constraints

The success of constraint programming is in part due to the existence of *global constraints*. They are characterized by accepting an arbitrary number of variables (often in form of a sequence) and are used to express global properties of the problem. A big advantage of global constraints is that they can be seen as unit and, thus, can possibly achieve more inference than using a set of primitive constraints with the same logical meaning. A common instance of this proposition is the `alldifferent` constraint. In Example 1.7 we show the enriched propagation owed to the global point of view.

Example 1.7. *Consider again the constraint* `alldifferent`(X_1, X_2, X_3, X_4) *with domains* $D_1 = D_2 = D_3 = \{1, 2, 3\}$ *and* $D_4 = \{1, 2, 3, 4\}$ *from Example 1.2. As shown in Example 1.6, a GAC algorithm would prune 1,2 and 3 from* D_4*. On the other hand* `alldifferent`(X_1, X_2, X_3, X_4) *is logically equivalent to the conjunction of inequalities:*

$$\bigwedge_{1 \leq i < j \leq 4} X_i \neq X_j \tag{1.3}$$

In this (decomposed) problem every single inequality is GAC, so the instance is GAC and no domain pruning is achieved!

Another benefit of global constraints are enhanced modelling possibilities, because we can concisely express global properties. Two global constraints, `regular` and `grammar`, that show these good properties are described in the next sections.

1.4.1 The `regular` Constraint

The constraint `regular`$([X_1, \ldots, X_k], \mathcal{A})$ over a sequence of variables $[X_1, \ldots, X_k]$ states that they have to constitute a word from a regular language given by the deterministic finite automaton (DFA) $\mathcal{A}$. The constraint was first proposed by Pesant [Pes04] who also gave a propagator that achieves general arc-consistency. Later, an incremental algorithm for solution counting and general arc-consistency was added [ZP07]. Regular languages can be used in shift scheduling problems to express that employees are not assigned to the hardest work, for instance a night shift, on two consecutive days. In Example 1.8 we illustrate how to model `regular` constraint.

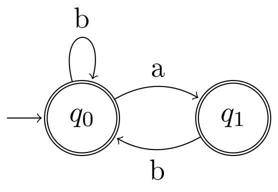

Figure 1.3: A DFA for the regular language L.

Example 1.8. *Let L be the regular language that accepts words over the alphabet $\{a, b\}$ such that there are no consecutive symbols a. This language can be modeled by a regular expression $L = (ab+b)^*(a+\varepsilon)$ or the automaton $\mathcal{A}$ depicted in Figure 1.3. The constraint* `regular`$([X_1, X_2, X_3], \mathcal{A})$ *supports the assignments $(X_1 = b, X_2 = b, X_3 = b)$ and $(X_1 = a, X_2 = b, X_3 = a)$, but not $(X_1 = b, X_2 = a, X_3 = a)$, because the automaton has no path with two consecutive a's.*

Note that we consider only finite sequences of finite domain variables here, so every language can be viewed as finite. Since every finite language is also regular, we can express any property with `regular` constraints. The classical example of a language that is not regular, $L = \{a^n b^n \mid n \geq 0\}$, becomes regular if we add the restriction that every word has to have fixed length. However, this might yield a large number of states such that both modelling the automaton and propagating this constraint might be impractical. Another restriction of this version of regular is its limitation to *deterministic* automata, because there exist languages that can be modelled by non-deterministic automata which are exponentially smaller than the smallest DFA.

1.4.2 The `grammar` Constraint

Once the regular language membership constraint was proposed, it was a logical next step to go up in Chomsky hierarchy, to *context-free grammars (CFG)*, to look for even better modelling tools. So, shortly after `regular`, the `grammar` constraint along with propagators that achieve generalized arc-consistency was published [QW06, Sel06]. Given a sequence of variables $[X_1, \ldots, X_k]$ and a context free grammar G, the constraint `grammar`$([X_1, \ldots, X_k], G)$ supports only evaluations of the X_i's that establish a word in $L(G)$, or more formally

$$\texttt{grammar}([X_1, \ldots, X_k], G) = \{(d_1, \ldots, d_k) \in D_1 \times \ldots \times D_k \mid S \longmapsto^* d_1 \cdots d_k\} \tag{1.4}$$

In Example 1.9 we illustrate the modelling with CFGs. For the algorithms in Section 2 we assume that the grammar is in Chomsky normal form, i.e., all productions are of the form $L \longrightarrow AB$ or $L \longrightarrow a$. Chomsky normal form is only a syntactical restriction, because every grammar can be transformed in linear time into a equivalent one that is in Chomsky normal form. However, in the examples we may use general rules for the sake of convenience.

Example 1.9. *Another non-regular language is* $L = \{a^n b^m \mid n, m \geq 1, n \neq m\}$*, but we can easily model* L *with a context-free grammar. Let* G *be a CFG with productions* $P = \{S \longrightarrow aS_1|S_2b, S_1 \longrightarrow aS_1|aS_1b|\varepsilon, S_2 \longrightarrow S_2b|aS_2b|\varepsilon\}$*. Obviously* $L(G) = L$*, because in the first step (nondeterministically)* S_1 *or* S_2 *is derived. After that* S_1 *(*S_2*) simply makes sure that more a's (b's) are created. Stating the constraint* `grammar`$([X_1, X_2, X_3, X_4], G)$ *allows the tuples* (a, b, b, b) *and* (a, a, a, b) *but not* (a, a, b, b)*.*

In [QW07] an improvement with respect to modelling was introduced, namely *conditional productions.* With their help we can restrict derivations of symbols to have a certain length or to start at a certain position. Consider a production rule $L \longrightarrow AB$. We attach three Boolean functions f_L, f_A, f_B to this rule, one for each nonterminal symbol. From symbol A, for instance, a word of length j can than only be derived starting from position i, iff $f_A(i, j)$ is *true.* For rules $A \longrightarrow a$ we have only one unary Boolean function that conditions A. Conditional productions do not increase the expressivity but facilitate modelling, as shown in Example 1.10.

Example 1.10. *Assume we want to model an employee who works between 6 and 8 hours and sleeps the rest of the day. He also begins his working day no earlier than 9 o'clock. In principle, this is possible with CFGs, but it involves a lot of duplicated states for counting the working hours. We can tackle this problem by introducing conditional productions. We introduce terminal symbols* s *(sleep) and* w *(work), as well as nonterminals* S*, the start symbol,* W *and* N *for work and sleep (night) and an auxiliary symbol* H*. The productions are then* $\{S \longrightarrow NH, H \longrightarrow WN, N \longrightarrow NN|s, W \longrightarrow WW|w\}$*. To add the restriction for the working time we attach the function* f_W *to the production* $H \longrightarrow WN$*.*

$$f_W(i, j) := (i \geq 9) \wedge (6 \leq j \leq 8) \tag{1.5}$$

The condition f_W *makes sure that the position of the* W *(argument* i*) is not before* 9 *o'clock and the length of the derived sequence (argument* j*) is between* 6 *and* 8*. Let* G *be the obtained grammar. The constraint* `grammar`$([X_1, \ldots, X_{24}], G)$ *on the daily routine of our employee makes sure that he exhibits the desired behavior.*

2 Solution Counting Algorithms

2.1 Introduction

Counting the number of solution has interesting applications such as belief revision [Dar01]. Of more interest to us is the application to heuristics in backtrack search [Pes05, ZP07, KDG04]. In general, counting the solutions of a CSP is significantly harder than pure solving it; even counting in only one `alldifferent` constraint is already #P-complete [Val79] while detectingwhether there exists a solution is polynomial [Rég94]. But in many other cases counting becomes tractable if we restrict our attention to single constraints. In Definition 2.1 we define the number of solutions of a constraint; then we give some elementary cases in Example 2.2.

Definition 2.1 (Number of Solutions).
Given a constraint C with scope $\{X_1, \ldots, X_k\}$. Then the number of solutions of constraint C, $sc(C)$ is the number of satisfying tuples:

$$sc(C) = |\{(d_1, \ldots, d_k) \in D_1 \times \ldots \times D_k \mid (d_1, \ldots, d_k) \in C\}| = |C| \tag{2.1}$$

Example 2.2. *Let X_1, X_2, X_3 be variables with domain $D_1 = D_2 = D_3 = \{1, 2, 3\}$. Then the constraint $X_1 < X_2$ has exactly 3 solutions: $X_1 = 1 \wedge X_2 = 2$, $X_1 = 1 \wedge X_2 = 3$ or $X_1 = 2 \wedge X_2 = 3$. Likewise, the weaker constraint $X_1 \leq X_2$ admits 6 solutions: the 3 mentioned and 3 more with $X_1 = X_2$. As another example let C be the constraint $X_1 \neq X_2$. Evidently, we then have $sc(C) = 6$. We can also consider simple arithmetic constraints, like $X_1 + X_2 = X_3$. This constraint has 3 solutions $(1, 1, 2)$, $(1, 2, 3)$ and $(2, 1, 3)$ written as tuples for the sake of simplicity.*

The number of solutions of an extensional constraint specified by its allowed tuples is simply their number.

The definition of $sc(C)$ as $|C|$ seems very simple, but in fact not all constraints are specified extensionally, so it might be a significant computational effort to compute the number of satisfying tuples. Being able to count the solutions for a given constraint is already powerful, since we can give better bounds for the number of solutions of a problem [Pes05]. Without any information about the constraints, the upper bound for the number of solutions of a CSP with variables as in Example 2.2 is 27 (the product of the domain sizes). Taking into account the information obtained by counting over the constraint $X_1 + X_2 = X_3$, we can improve this bound to 3.

		X_1			X_2			X_3		
C	$sc(C)$	1	2	3	1	2	3	1	2	3
$X_1 < X_2$	3	2	1	0	0	1	2	-	-	-
$X_1 \leq X_2$	6	3	2	1	1	2	3	-	-	-
$X_1 \neq X_2$	6	2	2	2	2	2	2	-	-	-
$X_1 + X_2 = X_3$	3	2	1	0	2	1	0	0	1	2

Table 2.1: Number of solutions for all possible assignments.

In many cases the number of solutions under the assumption of a certain assignment $X_i = a$ is even more interesting and, given that we can count the number of solutions of the constraint, sometimes easy to compute. We define this notion in Definition 2.3 which is followed directly by Example 2.4.

Definition 2.3 (Number of Solutions under Assignment).
Let C be a constraint with scope $\{X_1, \ldots, X_k\}$, X_i a variable in scope(C) and $a \in D_i$. Then we define the number of solutions of C under the assignment $X_i = a$ as:

$$sc(C, X_i, a) = |\{(d_1, \ldots, d_k) \in C \mid d_i = a\}| \tag{2.2}$$

Example 2.4. *Consider again variables X_1, X_2, X_3 with equal domains $D_i = \{1, 2, 3\}$. The number of solutions of a constraint $X_1 \neq X_2$ with $X_2 = 1$ is 2, since the assignment can be extended only with $X_1 = 2$ or $X_1 = 3$ not with $X_1 = 2 (= X_2)$. We denote this formally as $sc(X_1 \neq X_2, X_2, 1) = 2$. We summarize the solution counts for the constraints in Example 2.2 under all conditions in Table 2.1. The first column gives the constraint, the second column its number of solutions. In the second part of the table you can see the solution counts, each cell indicating the number of solutions for a constraint (row) with one particular assignment (column).*

In the example we can also see two properties that are intuitively clear: the numbers $sc(C, X_i, a)$ are always smaller than $sc(C)$ and the sum of $sc(C, X_i, a)$ over all a in domain D_i is exactly $sc(C)$. We formulate the following observation to capture these two direct consequences of Definition 2.3.

Observation 2.5. *For all constraints C and all variables $X_i \in$ scope(C) it holds:*

1. $\forall a \in D_i . sc(C, X_i, a) \leq sc(C)$
2. $\sum_{a \in D_i} sc(C, X_i, a) = sc(C)$

In this context it is convenient to define also the *solution density* of an assignment in a specific constraint, see Definition 2.6 [ZP07]. Intuitively, it describes the fraction of all solutions that make a particular assignment true.

Definition 2.6 (Solution Density).
The solution density $sd(C, X_i, a)$ of an assignment $X_i = a$ in a constraint C is defined as

$$sd(C, X_i, a) = \frac{sc(C, X_i, a)}{sc(C)} \tag{2.3}$$

As direct consequence of Observation 2.5 and this definition, we find that the solution densities for a constraint C and a variable X_i can be seen to a certain extent as probabilities. We formulize this in Observation 2.7.

Observation 2.7. *For all constraints C and all variables $X_i \in scope(C)$ it holds:*

1. $\forall a \in D_i . 0 \leq sd(C, X_i, a) \leq 1$
2. $\sum_{a \in D_i} sd(C, X_i, a) = 1$

From now on we refer with the term *solution counting* to all the defined notions of solution counting for a constraint, solution counting under an assignment $X_i = a$ and calculating the solution densities. We can give first evidence of how powerful solution counting in fact is by looking again at the first and last row of Table 2.1. We observe that some numbers in these rows are 0. This means that for example the assignment $X_1 = 3$ is not part of any satisfying tuple of the constraint $X_1 < X_2$. Hence we can remove 3 from D_1. In Observation 2.8 we establish this connection between solution counting and GAC of a constraint, namely solution counting captures GAC. The proof follows directly from the definitions and is similar to our argumentation for the example.

Observation 2.8. *For every constraint C, every variable $X_i \in scope(C)$ and every value $a \in D_i$ we have:*

$$sc(C, X_i, a) = 0 \Longleftrightarrow \textit{GAC prunes } a \textit{ from } D_i$$

We will make use of this observation when we apply the model counting procedure also as GAC propagator. We start with describing an algorithm to count models in a specific class of propositional formulas, namely `sd-DNNF`. After that we introduce a very general algorithm to count solutions in constraints by compiling them into `sd-DNNF`, exploiting a linear model counting algorithm for `sd-DNNF`.

2.2 Model Counting in `sd-DNNF`

Results in the knowledge compilation area showed that there are classes of propositional formulas that admit tractable model counting [DM02]. One of these classes is `sd-DNNF`, the class of smooth, deterministic, and decomposable formulas. We provide the necessary definitions, beginning with the classes of formulas `NNF`, `DNNF` and `sd-DNNF`, before we describe the actual counting algorithm.

Definition 2.9 (Negation Normal Form).
Let P be a set of propositional variables. The language of formulas in negation normal form (`NNF`) is defined as the set of rooted, directed acyclic graphs (DAG) where each leaf node is labeled with true, false or a propositional literal (from P) and each inner node is labeled with $\wedge$ or $\vee$ and has arbitrarily many children. Let $vars(F)$ denote the set of variables that occur in formula F.

Intuitively, `NNF` consists of all formulas that have negation only at the level of propositional variables. For example $p \wedge \neg(q \vee \neg s)$ is not in `NNF`, because of the negation in the subformula in $\neg(q \vee \neg s)$. Obviously, this can be rewritten by DeMorgan's laws into $(\neg q \wedge s)$. In this way every propositional formula can be rewritten in linear time and space to an equivalent formula in `NNF`. Now we define some conditions that give rise to the more specific classes `DNNF` and `sd-DNNF` [DM02].

Definition 2.10 (Decomposability, Determinism, and Smoothness).
A formula is called

- *decomposable if for every conjunction $F = F_1 \wedge \ldots \wedge F_k$ it holds*

$$vars(F_i) \cap vars(F_j) = \emptyset \text{ for } i \neq j$$

- *deterministic if for every disjunction $F = F_1 \vee \ldots \vee F_k$ it holds*

$$\{F_i, F_j\} \models \textit{false} \text{ for } i \neq j$$

- *smooth if for every disjunction $F = F_1 \vee \ldots \vee F_k$ it holds*

$$vars(F_i) = vars(F)$$

Decomposability corresponds to dividing the formula into several independent parts, because if subformulas do not share any variables they are completely independent. Determinism in turn is a semantical condition that restricts the disjunctions to be *true* if and only if *exactly* one disjunct is *true*, in particular it is not allowed for two children to be *true* at the same time. Smoothness is a purely syntactical condition and trivial to achieve by adding valid subformulas. More precisely F_i is transformed to $F_i \wedge \bigwedge_{p \in M}(p \vee \neg p)$ where $M = vars(F) \setminus vars(F_i)$ [DM02].

Definition 2.11 (`DNNF` and `sd-DNNF`).
The language `DNNF` is the subset of `NNF` that fulfills the decomposability condition. The language `sd-DNNF` is the subset of `DNNF` that meets additionally the conditions determinism and smoothness.

While smoothness can be added in polynomial time to any formula, this is not the case for decomposability and determinism: in general transforming a formula in `NNF` (`DNNF`) to a formula in `DNNF` (`sd-DNNF`) can result in an exponential growth of the formula, but typically the imposed restrictions provide us with more tractable reasoning like polynomial time consistency checking for both `DNNF` and `sd-DNNF` or model counting for `sd-DNNF`.

The basic idea is simple because of the conditions determinism and decomposability [Dar01]. Intuitively it is clear that the number of models of an and-node that fulfills the decomposability condition is the *product* of the number of models in each conjunct: Since the conjuncts are independent we can combine the models of the conjuncts arbitrarily to obtain models of the conjunction itself. For deterministic or-nodes the number of models is at least the *sum* of the models of the children: Since a model for one child cannot be a model for an other child we do not count models several times by the summation. Smoothness ensures that we can compute the exact number of models by adding *don't care* variables.

For counting the models of a formula F we attach a number $\mathsf{VAL}(N)$ to every node N of the DAG representing F. $\mathsf{VAL}(N)$ corresponds to the number of models of the subformula rooted at node N. It was shown in [Dar01] that computation of these values is sufficient for counting the models of a formula F. The computation of $\mathsf{VAL}(N)$ is recursively defined following our argumentation:

$$\mathsf{VAL}(N) = \begin{cases} 0 & N = l \text{ is a leaf and } l \text{ is known to be } false \\ 1 & N = l \text{ is a leaf and it is not known to be } false \\ \sum_{i=1}^{k} \mathsf{VAL}(N_i) & N = or(N_1, \ldots, N_k) \\ \prod_{i=1}^{k} \mathsf{VAL}(N_i) & N = and(N_1, \ldots, N_k) \end{cases} \tag{2.4}$$

As pointed out in Section 2.1 the aspect of counting the models under the assumption of an assignment is important. The `sd-DNNF` formalism provides us with the ability to compute also the number of models where a certain literal l is *true*. Therefore, we attach values $\mathsf{PD}(N)$ to every node N and provide a recursive rule to calculate them. Intuitively, the value $\mathsf{PD}(N)$ says how many models for the root formula there are, given that the subformula at N evaluates to *true*.

$$\mathsf{PD}(N) = \begin{cases} 1 & \text{if } N \text{ is the root} \\ \sum_{M \text{ parent of } N} \mathsf{CPD}(M, N) & otherwise \end{cases} \tag{2.5}$$

$$\mathsf{CPD}(M, N) = \begin{cases} \mathsf{PD}(M) & M \text{ is an or-node} \\ \mathsf{PD}(M) \cdot \prod_{L \text{ child of } M, L \neq N} \mathsf{VAL}(L) & M \text{ is an and-node} \end{cases} \tag{2.6}$$

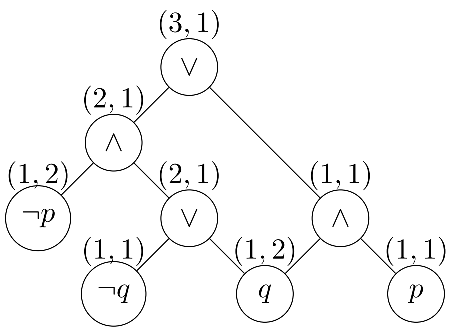

Figure 2.1: `sd-DNNF` for $p \to q$ together with pairs $(\mathsf{VAL}(N), \mathsf{PD}(N))$ for every node N.

The computation rules for the VAL and PD values provide us with a model counting algorithm for `sd-DNNF` that is linear in the size of the formula. In a first (bottom-up) run $\mathsf{VAL}(N)$ is computed for all nodes N. After this, the values $\mathsf{PD}(N)$ are calculated top-down. In the end the value $\mathsf{PD}(l)$ for a leaf with label l gives the number of models if l would be *true*. In particular we have the relations $\mathsf{PD}(p) = sc(F, p, 1)$ and $\mathsf{PD}(\neg p) = sc(F, p, 0)$ when F is the formula we count the models in. We conclude this section with Example 2.12 which illustrates all the defined notions.

Example 2.12. *Let $F = (p \to q)$ the formula we want to perform model counting on. F can be rewritten as $\neg p \vee q$, but this is not yet deterministic. To make it deterministic it is enough to change the formula to the equivalent $F' = \neg p \vee (p \wedge q)$. F' is not yet smooth, but we can easily add the valid subformula $q \vee \neg q$ to the first disjunct, obtaining $F'' = (\neg p \wedge (q \vee \neg q)) \vee (p \wedge q)$. Clearly, F'' is in `sd-DNNF`. Figure 2.1 shows F''. The pair $\mathsf{VAL}(N), \mathsf{PD}(N)$ is attached to every node N. First we perform the bottom-up run to compute $\mathsf{VAL}(N)$ for every N. For every leaf it is 1 since we have no prior knowledge about the leaves. Applying Equation (2.4) we obtain $\mathsf{VAL}(root) = 3$, meaning that $p \to q$ has 3 models which is indeed the case. The top-down run following Equation (2.5) yields the PD values in every node. We see $\mathsf{PD}(p) = 1$, $\mathsf{PD}(q) = 2$, $\mathsf{PD}(\neg p) = 2$ and $\mathsf{PD}(\neg q) = 1$ which conforms with the true numbers.*

Let us now set q to false, i.e., the VAL value of the node labeled with q changes to 0. Updating all the numbers in the DAG in Figure 2.1 gives us the DAG in Figure 2.2. We observe that $\mathsf{PD}(p) = 0$. Recall that the PD values of the leaves correspond to the number of models that evaluate the leaf to true. So p is true in 0 models and can be pruned.

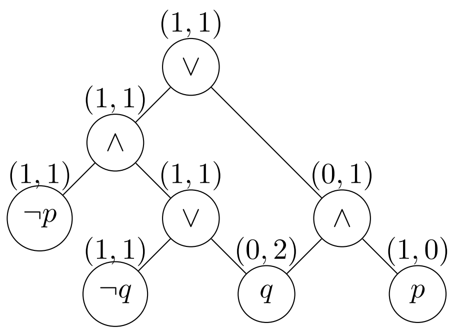

Figure 2.2: sd-DNNF for $p \rightarrow q$ together with pairs $(\mathsf{VAL}(N), \mathsf{PD}(N))$ for every node N, with respect to the additional knowledge $\neg q$.

2.3 Solution Counting using Compilation to sd-DNNF

We can now combine the tractable model counting algorithm for sd-DNNF and Observation 2.8 to get a solution counting algorithm that maintains also GAC by translating constraints into a logically equivalent formula in sd-DNNF. This is a very general approach, because in principle every constraint could be compiled to an equivalent formula in sd-DNNF. Of course this might yield formulas of impractical size for some constraints, for which it is not suitable. One example is the alldifferent constraint. We already mentioned that counting the solutions of alldifferent is $\#P$-complete, whereas the model counting algorithm for sd-DNNF is linear in the size of the formula, hence there cannot be a polynomial sd-DNNF formula. Fortunately, there are some expressive constraints that can be translated into reasonably sized sd-DNNF. Among these are element, regular, extensional and grammar, which we examine in Sections 2.3.1, Section 2.3.2 and Section 2.3.3.

In order to translate constraints to sd-DNNF we have to identify the propositional variables, which may be attached to the leaves. There are several possibilities, for example $(X_i = a)$ with their negations $(X_i \neq a)$ or $(X_i \leq a)$ together with their negations $(X_i > a)$. The difference between them is in $O(|D_i|)$, since we can, on the one hand, express $(X_i = a)$ with $(X_i \leq a)$ and $\neg(X_i \leq a-1)$, and on the other hand, $(X_i \leq a)$ is logically equivalent to $\bigwedge_{b \in dom(X_i), b \leq a}(X_i = b)$. Since we want to perform solution counting for each $X_i = a$ we opt for propositional variables $(X_i = a)$. Now we have $sc(C, X_i, a) = \mathsf{PD}(N)$ when N is the leaf node with label $(X_i = a)$.

2.3.1 Compilation of regular Constraints to sd-DNNF

In this section we show a translation from the constraint regular$([X_1, \ldots, X_k], \mathcal{A})$ to an equivalent formula in sd-DNNF. This is a quite general result as we can express many constraints in terms of regular. Let $\mathcal{A}$ be the DFA $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$, where

- Σ is the alphabet
- Q is a finite set of states
- $q_0 \in Q$ is the initial state
- $\delta : Q \times \Sigma \to Q$ the transition function and
- $F \subseteq Q$ is the set of final states

Without loss of generality we assume δ to be a total function and $D_i \subseteq \Sigma$. If not we construct $\Sigma' = \Sigma \cup \bigcup_{i=1}^{k} D_i$ and introduce an extra state $q_\perp \notin F$ which is the result of all undefined inputs of δ. We define a function f such that $f(1, q_0)$ maps the constraint `regular`$([X_1, \ldots, X_k], \mathcal{A})$ to a formula in `sd-DNNF`:

$$f(i,q) = \begin{cases} false & i = k+1, q \notin F \\ true & i = k+1, q \in F \\ \bigvee_{a \in D_i} G_{X_i = a} \wedge f(i+1, \delta(q,a)) & otherwise \end{cases} \tag{2.7}$$

In Equation (2.7) the subformula $G_{X_i=a}$ abbreviates $X_i = a \wedge \bigwedge_{b \in D_i \setminus \{a\}} X_i \neq b$. This seems redundant because $X_i = a$ implies $X_i \neq b$ for $a \neq b$, but is in fact necessary since on the level of propositional variables this is not true anymore. Intuitively, this function enumerates all paths of length k and assigns *true* (*false*) if the path is (not) a word in the language of $\mathcal{A}$. Afterwards, *true* and *false* can be propagated through the DAG in the obvious way. If the graph collapses to *false* (*true*) the constraint is unsatisfiable (valid). Of course, there might be an exponential number of paths through the automaton, but caching the calls of f prevents us from enumerating them all. In particular, we stay polynomial and the constructed DAG has size $O(k \cdot |Q|)$, because the recursion depth is bounded by $k+1$ and the number of nodes at level i by $|Q|$. At last we check whether the resulting DAG is in `sd-DNNF`. We observe first that all $G_{X_i=a}$ are decomposable and only propositional variables with index i occur. Since we increase i in the recursive call, such variables cannot occur again, hence the subformulas from the third case of Equation (2.7) are decomposable, and so is the resulting formula. Now let F_a and F_b be two arbitrary, but distinct disjuncts in the third case of Equation (2.7) and suppose we are in level i. By definition, $X_i = a$ is a consequence of $G_{X_i=a}$ and $X_i \neq a$ is implied by $G_{X_i=b}$. Evidently the formula $F_a \wedge F_b$ implies then $X_i = a \wedge X_i \neq a$ which is equivalent to *false*, thus the result of the application of f is also deterministic. Smoothness is fulfilled since in every or-node of level i all variables $(X_j = a)$ for all $i < j \leq k$ and all $a \in D_j$ occur. We can prove this easily by induction on $j = k - i$. Example 2.13 illustrates the application of function f.

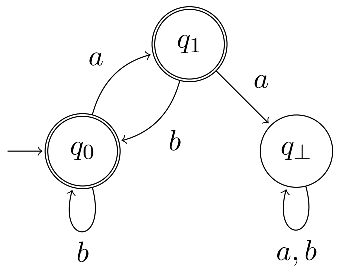

Figure 2.3: A DFA for the language of all words that do not have two consecutive a's.

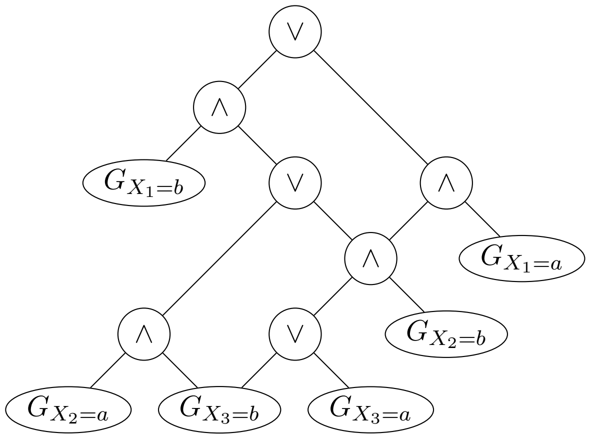

Figure 2.4: The `sd-DNNF` formula equivalent to `regular`$([X_1, \ldots, X_k], \mathcal{A})$.

Example 2.13. *Let $\mathcal{A}$ be the automaton from Example 1.8 whose language consists of all words over $\{a, b\}$ such that no consecutive a's occur. We depict it again with the additional state $q_\perp$ in Figure 2.3. Let the constraint be* `regular`*$([X_1, X_2, X_3], \mathcal{A})$ with $D_i = \{a, b\}$. After applying f and propagating the occurring true and false upwards, we obtain the formula shown in Figure 2.4. We can easily convince ourselves that the result is really decomposable, deterministic and smooth, thus in* `sd-DNNF`*.*

Counting in `regular` constraints is, however, not new. Pesant and Zanarini [ZP07] published a solution counting algorithm that is based on dynamic programming. Their approach uses a layered graph structure that resembles very much the `sd-DNNF` formula we create (viewed as a DAG), and in fact their algorithm does exactly the same computations as our. This is quite interesting for two algorithms emerging from completely different ideas.

The formulas that are obtained from Equation (2.7) do not yet exploit all the succinctness of the `sd-DNNF` formalism. In particular, determinism states only that any two disjuncts have to be contradictory in general, whereas in the result of f the disjuncts contradict already in the first level after the disjunction (recall that they contain $G_{X_i=a}$ and $G_{X_i=b}$, respectively) – they fulfill the *decision* property [DM02]. There are propositional theories that prove the exponential gap between formulas with decision property and formulas without it, i.e., there are theories that have a polynomially sized `sd-DNNF` representation, but if we additionally impose the decision property they have not anymore [Dar01].

2.3.2 Compilation of `extensional` Constraints to `sd-DNNF`

Extensional constraints with scope $\{X_1, \ldots, X_k\}$ can be viewed as relation R on the Cartesian product of the variables' domains $R \subseteq D_1 \times \ldots \times D_k$. The logical reading of such a relation R is shown in Equation (2.8). We can easily observe that this formula is decomposable.

$$R \equiv \bigvee_{(d_1,\ldots,d_k)\in R} \bigwedge_{i=1}^{k} (X_i = d_i) \tag{2.8}$$

If we now replace the assignments $X_i = d_i$ in this equation by the expression $G_{X_i=d_i}$ (recall that this was defined as $(X_i = d_i) \wedge \bigwedge_{d\in D_i\setminus\{d_i\}} (X_i \neq d))$, we obtain Equation (2.9) which is still decomposable, but also deterministic and smooth. In this way, we obtain immediately a model counting algorithm for `extensional` constraints.

$$R \equiv \bigvee_{(d_1,\ldots,d_k)\in R} \bigwedge_{i=1}^{k} G_{X_i=d_i} \tag{2.9}$$

This approach is direct, but potentially there is big room for improvements, because:

- if the relation is very large ($O(|D_1 \times \ldots \times D_k|)$) also the `sd-DNNF` formula becomes very large, since we create an and-node for every tuple,

- the translation is blind, since we do not exploit any structure that might be hidden in the table, and

- the part of the language `sd-DNNF` we use is in fact very restricted because the formulas have only four layers: the first layer is the global or-node, the second layer consists of the and-nodes each representing a tuple, and the two last layers consisting of the formulas $G_{X_i=d_i}$.

One example for structure in the problem is the different domain size of the variables. Based on the idea conveyed by Figures 1.1 and 1.2 from the section about variable orderings we can build a tree-like structure of the table.

X_2	X_3	X_4
1	3	4
3	4	1
4	1	1

Table 2.2: Restriction of C to $X_1 = 2$

.

X_1	X_3	X_4
2	1	4

Table 2.3: Restriction of C to $X_2 = 3$

.

In order to minimize the size of the tree we try to minimize the outdegree of the nodes close to the root. To formalize this, we introduce in Definition 2.14 the notion of *restricting* an (`extensional`) constraint to an assignment $X_i = d$ and illustrate it in Example 2.15.

Definition 2.14 (Restriction of a Constraint).
Let C be an `extensional` constraint with scope $\{X_1, \ldots, X_k\}$, X_i a variable in scope(C) and d a value in the domain of X_i. Then we define a new constraint C' as the restriction $C|_{X_i=d}$ of C to the assignment $X_i = d$ as follows:

1. *scope*$(C') = \{X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_k\}$
2. $C' = \{(d_1, \ldots, d_{i-1}, d_{i+1}, \ldots, d_k) \mid (d_1, \ldots, d_k) \in C \wedge d_i = d\}$

Example 2.15. *We consider the `extensional` constraint C defined by the table:*

X_1	X_2	X_3	X_4
1	*1*	*1*	*2*
2	*1*	*3*	*4*
3	*2*	*2*	*3*
2	*3*	*4*	*1*
2	*4*	*1*	*1*

We depict the restrictions $C|_{X_1=2}$ and $C|_{X_2=3}$ in Table 2.2 and 2.3. Observe that the scope of the new constraints is $\{X_2, X_3, X_4\}$ and $\{X_1, X_3, X_4\}$, respectively. The restriction is possibly empty (and is allowed to be), as we can for example restrict the depicted constraint to $X_1 = 4$.

If we calculate the restrictions to $X_i = d$ for all values $d \in D_i$ and conjoin each of them with the corresponding assignment we remain with the same constraint. We make this formal in Equation (2.10).

$$C \equiv \bigvee_{d \in D_i} (X_i = d) \wedge C|_{X_i=d} \tag{2.10}$$

Using this idea we can refine the logical expression describing the table constraint by defining a recursive function *trans* for translating the table into a logical formula. Therefore, we select in every step a variable and make use of Equation (2.10). We add to the previous equation only two termination cases and apply the function recursively:

$$trans(C) = \begin{cases} \bigvee_{(d) \in C} (X_i = d) & \text{if } scope(C) = X_i \\ \bigwedge_{X_i \in scope(C)} (X_i = d_i) & \text{if } C = \{(d_1, \ldots, d_k)\} \\ \bigvee_{d \in D_i} (X_i = d) \wedge trans(C|_{X_i = d}) & \text{otherwise} \end{cases} \tag{2.11}$$

It is left open how we choose X_i in the recursive case, but in order to minimize the structure we can, for instance, choose the variable with the smallest domain since this results in a node with small outdegree. However, this is only one alternative for selecting a variable. We refer to [JBKW08] where more alternatives are examined.

The formula $trans(C)$ is in `DNNF` but on the level of propositional logic not yet deterministic and smooth. We can make it so by replacing again a leaf labelled with $(X_i = d)$ by $G_{X_i=d}$ with the same arguments as above. As the formulas representing `regular` constraints also the results of this compilation fulfill the decision property.

If we view the arising formula as a DAG we can of course share substructures. In particular we can do this when the function is called twice with the same argument (both scope and possible tuples are equal) during recursion. We can also imagine the function to cache the values for which it already computed the result and return just the stored entry if a cache hit occurs.

We considered some instances of `extensional` constraints to show that we are able to save space by compiling the table using *trans*. On the one hand we looked at tables from the configuration problem Renault [AFM02] and on the other hand at several randomly created tables. We compared only the inner nodes that are created by both of the approaches, since the leaf nodes are the same in either case. In the direct encoding (Equation (2.9)) the number of nodes is the number of tuples of the table plus one, because we create an and-node for every tuple and an or-node as root. In the encoding using the *trans* function we count the number of created nodes. We list also the maximal number of tuples in the table which we determine by looking at the scope of the constraints.

Table 2.4 shows the results for selected tables of the Renault instance. It can be seen that the direct encoding outperforms the encoding using *trans* on very tight constraints, i.e., tables with only few allowed tuples, but is clearly outperformed on constraints with more solutions. To confirm this impression we created some random tables and conducted the same experiments. We distinguish between table constraints on variables with uniform (Table 2.5) and nonuniform domain size (Table 2.6). The domain sizes in the second case are $2, 3, \ldots, k + 1$ in constraints with arity k.

$\|C\|$	$\#_{\max}$	nodes direct encoding	nodes *trans*
48	4050	49	73
130	6480	131	150
336	54000	337	225
342	648000	343	450
1114	40500	1115	130
2025	68040	2026	109
4440	136080	4441	149
6498	136080	6599	194
33437	3402000	33438	448

Table 2.4: Comparison of size for the Renault instance. $|C|$ is the number of tuples in the table, $\#_{\max}$ is the possible number of tuples, i.e., the product of the domain sizes of the variables in $scope(C)$.

d	tightness	nodes direct encoding	nodes *trans*
	0.1	313	556
	0.2	626	751
5	0.3	938	875
	0.4	1251	925
	0.5	1563	963
	0.1	1680	2070
	0.2	3362	2749
7	0.3	3362	3150
	0.4	5042	3328
	0.5	6723	3374

Table 2.5: Random table constraints over 5 variables with uniform domain size d.

As a result of these tables we can see a similar behavior for both uniform and nonuniform domains. This is interesting because for uniform domain sizes in random tables the reason for saving nodes can neither be the structure hidden in variables' domain size nor, being a randomly created, the structure in the table. Hence we suspect that structure sharing in the propositional formula has also big influence in these cases. However, we observe more savings for the tables from the Renault instance, hence we expect the compilation to work well in practice.

arity k	tightness	nodes direct encoding	nodes *trans*
5	0.1	72	149
	0.2	144	200
	0.3	216	223
	0.4	288	258
	0.5	360	242
7	0.1	4032	4603
	0.2	8064	6102
	0.3	12096	6817
	0.4	16128	7081
	0.5	20160	7153

Table 2.6: Random table constraints of arity k with nonuniform domain sizes $2, 3, \ldots, k+1$.

2.3.3 Compilation of `grammar` Constraints to `(sd-)DNNF`

Context-free languages are more expressive than regular languanges and one of the advantages is that we can model nondeterministically with a context-free grammar whereas we can describe every regular language by a *deterministic* finite automaton. But exactly the determinism in the automaton made it easy to translate the `regular` constraint to an equivalent formula in `sd-DNNF`. Since formulas in `sd-DNNF` are deterministic, it is presumably hard to make a translation of nondeterministic notions into `sd-DNNF`. It turns out that nondeterminism itself is not a problem, but *ambiguity* of the grammar is. We show an algorithm that counts the exact number of solutions in nonambiguous grammars and otherwise calculates an upper bound.

Our translation is based on the work of Quimper and Walsh [QW07]. They translate constraints of the form `grammar`$([X_1, \ldots, X_k], G)$ into `CNF`. In an intermediate step they construct an And/Or graph, particularly, a rooted DAG representing all the possible parsing trees for the CFG G given the domains D_i of the variables X_i. The inner nodes of the DAG are labeled with $\wedge$ and $\vee$, whereas the leaves are labeled with assignments $X_i = a$.

Inspired by their work we define a function *graph* that maps a nonterminal symbol L and two numbers l and r to a propositional formula that describes all words that are derivable from L in interval $[l, r]$. Note that our definition is not suitable as a computational basis, but it is convenient to prove the desired properties. Recall that we assume the grammar to be in Chomsky normal form (see Section 1.4.2), i.e., all productions are of the form $L \longrightarrow AB$ or $L \longrightarrow a$.

$$graph(L,l,r) = \begin{cases} \bigvee_{L \longrightarrow a} X_l = a & \text{if } l = r \\ \bigvee_{L \longrightarrow AB} \bigvee_{l \le m < r} graph(A,l,m) \wedge graph(B,m+1,r) & \text{otherwise} \end{cases} \tag{2.12}$$

Intuitively, parsing a symbol L in the interval $[l,l]$, i.e., with length 1, is only possible if there is a corresponding production $L \longrightarrow a$ in G. Otherwise the expression in the first case is *false* (empty disjunction). On the other hand, parsing a nonterminal L in interval $[l,r]$ with $l < r$ is possible if there is a rule $L \longrightarrow AB$ and we can divide the interval into non-empty subintervals $[l,m]$ and $[m+1,r]$ such that we can parse A on $[l,m]$ and B on $[m+1,r]$.

However, in order to make the formula logically equivalent to the constraint, we have to adapt two details. First, it does not yet take the domains of the variables $\{X_1,\ldots,X_k\}$ into consideration. This is easily achieved by checking additionally if a is contained in the domain of X_l in the first case of Equation (2.12). Second, we have to encode the first case as exclusive or, because on constraint level it is not allowed to have $(X_i = a)$ and $(X_i = b)$ for $a \neq b$ at the same time, whereas the formula would admit. We can achieve this by again making use of $G_{X_i=a}$ instead of $X_i = a$. The updated first case is shown in Equation (2.13).

$$\bigvee_{L \longrightarrow a \wedge a \in D_l} G_{X_l=a} \tag{2.13}$$

Applying the function *graph*, we might obtain some leaves that are labelled with *false*, because of possibly occurring empty disjunctions in both cases of Equation (2.12), but they can easily propagated upwards by the usual Boolean rules. In the following, when we write to $graph(L,l,r)$, we refer to the formula without any occurrences of *false* (except for the case when the constraint is unsatisfiable, then the formula itself is *false*) that is obtained by applying the function with updated first case.

We will formalize and prove the logical equivalence of constraint and propositional formula in Proposition 2.17, but before we want to give a small example for a translation.

Example 2.16. *Consider the constraint* `grammar`$([X_1,X_2,X_3],G)$ *where G is a grammar with the rules $S \mapsto aS_1$ and $S_1 \mapsto aS_1 \mid bS_1 \mid a \mid b$, whose language contains all words over $\{a,b\}$ beginning with an a. Assume further domain $\{a,b\}$ for all the variables. The formula resulting from $graph(S,1,3)$ is depicted as DAG in Figure 2.5. Note that the or-nodes are labelled with the current arguments of the function calls and that we can again share subgraphs (the or-node labelled with $(S_1,3,3)$). Obviously the leaf label $X_1 = b$ does not occur because the first symbol cannot be a b. So we can prune b from D_1.*

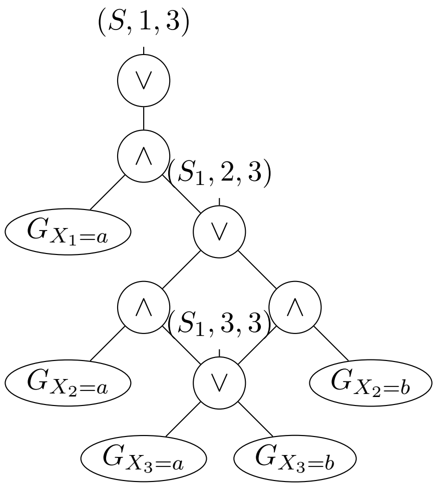

Figure 2.5: The result of *graph*$(S, 1, 3)$ for the grammar in Example 2.16.

Now we come back to the proposition about logical equivalence between the two formalisms:

Proposition 2.17. *Let G be a context-free grammar with start symbol S and $l \leq r$. Then $(d_l, \ldots, d_r)$ is a support of* `grammar`*$([X_l, \ldots, X_r], G)$ iff $\{X_i = d_i \mid l \leq i \leq r\}$ is a model for graph(S, l, r)*

Proof. We prove this statement by induction on $k = r - l$. For the *induction base* let $k = 0$. We have then

$$\begin{aligned} & (d_l) \text{ is support of } \texttt{grammar}([X_l], G) \\ \Longleftrightarrow\ & S \longrightarrow^* d_l \wedge d_l \in D_l \\ \Longleftrightarrow\ & S \longrightarrow d_l \wedge d_l \in D_l \\ \Longleftrightarrow\ & G_{X_l=d_l} \text{ occurs in Equation (2.13)} \\ \Longleftrightarrow\ & \{X_l = d_l\} \text{ is a model of Equation (2.13)} \\ \Longleftrightarrow\ & \{X_l = d_l\} \text{ is a model of } \mathit{graph}(S, l, l) \end{aligned}$$

In the *induction step* we choose $k > 0$. We denote with $G[A]$ a grammar that has the same productions as G but whose starting symbol is A (instead of S by default).

We have now:

$$\begin{aligned}
& (d_l, \ldots, d_r) \text{ is support of } \texttt{grammar}([X_l, \ldots, X_r], G) \\
\Longleftrightarrow\ & S \longrightarrow^* d_l d_2 \ldots d_r \wedge d_i \in D_i, l \leq i \leq r \\
\Longleftrightarrow\ & d_i \in D_i, l \leq i \leq r \wedge \\
& \exists (S \longrightarrow AB) \exists m. l \leq m < r \wedge A \longrightarrow^* d_l \ldots d_m \wedge B \longrightarrow^* d_{m+1} \ldots d_r \\
\Longleftrightarrow\ & \exists (S \longrightarrow AB) \exists m. l \leq m < r \wedge \\
& (d_l, \ldots, d_m) \text{ is support of } \texttt{grammar}([X_l, \ldots, X_m], G[A]) \text{ and} \\
& (d_{m+1}, \ldots, d_r) \text{ is support of } \texttt{grammar}([X_{m+1}, \ldots, X_r], G[B]) \\
\overset{hyp}{\Longleftrightarrow}\ & \exists (S \longrightarrow AB) \exists m. l \leq m < r \wedge \\
& \{X_j = d_j \mid l \leq j \leq m\} \text{ is model for } \mathit{graph}(A, l, m) \text{ and} \\
& \{X_j = d_j \mid m+1 \leq j \leq r\} \text{ is model for } \mathit{graph}(B, m+1, r) \\
\Longleftrightarrow\ & \{X_j = d_j \mid l \leq m \leq r\} \text{ is model for } \mathit{graph}(S, l, r) \qquad \blacksquare
\end{aligned}$$

Let us now analyze the structure of $\mathit{graph}(S, 1, k)$ in more detail with respect to decomposability, determinism and smoothness. With respect to decomposability and smoothness it is convenient to make the following observation about the occurring variables. The observation follows easily from construction.

Observation 2.18. *Let $X_1, \ldots, X_k$ variables with domains $D_1, \ldots, D_k$ and G a context-free grammar. Then for every non-terminal symbol L in G and every l and r with $1 \leq l < r \leq k$ such that $\mathit{graph}(L, l, r) \not\equiv \mathit{false}$ it holds:*

$$\mathit{vars}(\mathit{graph}(L, l, r)) = \{(X_j = d) \mid l \leq j \leq r, d \in D_j\}$$

Decomposability is a direct consequence of this observation because for every conjunction the intervals corresponding to the conjuncts $[l, m]$ and $[m+1, r]$ are disjoint, hence also the sets of variables that occur inside. For showing smoothness we have to check every or-node. We start with the or-nodes that arise from the first case of Equation (2.12). Obviously, we have $\mathit{vars}(G_{X_i = d_1}) = \mathit{vars}(G_{X_i = d_2})$ for all $d_1, d_2 \in D_i$ so these nodes fulfill the smoothness condition. Now consider an or-node that emerged from the second case with arguments L, l and r and two arbitrary but non-*false* disjuncts F_1, F_2. These disjuncts were produced because there are productions $L \longrightarrow A_1 B_1$ and $L \longrightarrow A_2 B_2$, respectively, and numbers m_1 and m_2 between l and r such that $F_i = \mathit{graph}(A_i, l, m_i) \wedge \mathit{graph}(B_i, m_i + 1, r)$, $i \in \{1, 2\}$. Applying Observation 2.18 yields $\mathit{vars}(F_i) = \{(X_j = d) \mid l \leq j \leq r, d \in D_j\}$, in particular $\mathit{vars}(F_1) = \mathit{vars}(F_2)$.

Now we focus on the determinism criterion. We can see directly that it is fulfilled on the or-nodes generated from first case of Equation (2.12), but for the other or-nodes this does not hold true necessarily. To show this, consider a simple grammar G with the only rule $S \longrightarrow SS \mid s$ and a constraint $\texttt{grammar}([X_1, X_2, X_3], G)$ for variables with

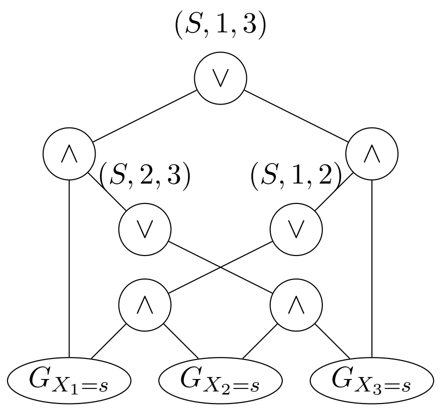

Figure 2.6: Formula obtained from *graph*$(S,1,3)$.

equal domain $D = \{s,t\}$. The application of *graph*$(S,1,3)$ yields the formula depicted in Figure 2.6. The constraint has exactly one support (s,s,s) so the formula has the only model $\{X_1 = s, X_2 = s, X_3 = s\}$ (by Proposition 2.17). Since the root node in Figure 2.6 is a disjunction with two satisfiable children and the formula has only one model this model has to be model for both of the disjuncts. This is in conflict with determinism.

Evidently, neither the grammar in Example 2.16 nor the grammar with the only rule $S \longrightarrow sS \mid s$ (which is in fact equivalent to the previous one) show this behavior – their *graph* formulas are deterministic. Searching for the reason, we find that *ambiguity* of the grammar may cause nondeterminism of the formula. We recall first the definition of an ambiguous grammar and illustrate then with Example 2.20 how ambiguity relates to nondeterminism.

Definition 2.19 (Ambiguous Grammar).
A grammar G with start symbol S is ambiguous if there exists a word $w \in L(G)$ such that there are two derivations with different derivation trees leading to w.

Example 2.20. *x Let G again be the grammar with rules $r_1 : S \longrightarrow SS$ and $r_2 : S \longrightarrow s$ and consider the constraint* `grammar`$([X_1, X_2, X_3], G)$. *Then we can derive the word sss in two different ways (we underline the symbol that is replaced and indicate above the derivation arrow which production rule was used):*

$$\underline{S} \xrightarrow{r_1} \underline{S}S \xrightarrow{r_2} s\underline{S} \xrightarrow{r_1} s\underline{S}S \xrightarrow{r_2} ss\underline{S} \xrightarrow{r_2} sss \qquad (2.14)$$

and

$$\underline{S} \xrightarrow{r_1} \underline{S}S \xrightarrow{r_1} \underline{S}SS \xrightarrow{r_2} s\underline{S}S \xrightarrow{r_2} ss\underline{S} \xrightarrow{r_2} sss \qquad (2.15)$$

There are more derivations that produce sss but they are equal to one of these two. For example

$$\underline{S} \xrightarrow{r_1} S\underline{S} \xrightarrow{r_2} \underline{S}s \xrightarrow{r_1} S\underline{S}s \xrightarrow{r_2} \underline{S}ss \xrightarrow{r_2} sss$$

is equivalent to the second one. However, the two derivations above are really different, because they have different derivation trees, shown in Figure 2.7. We find the two derivations of sss also in Figure 2.6. The first derivation corresponds to the left child of the root and the second to the right child.

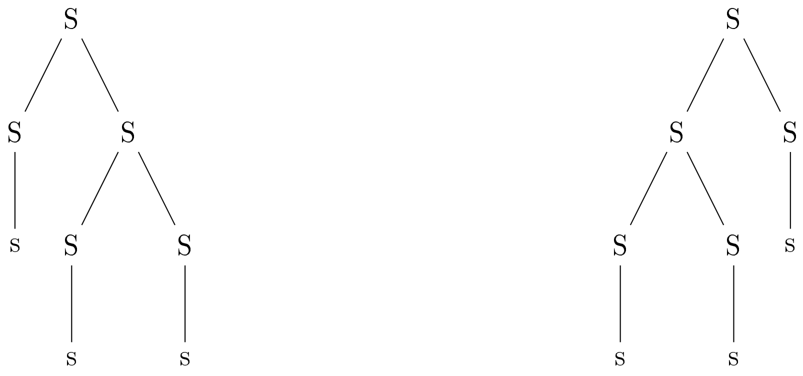

Figure 2.7: The derivation trees of the word *sss*. The left tree corresponds to the derivation in (2.14) and the right one to the derivation in (2.15).

We formulate the exact relation between ambiguity of a grammar and nondeterminism of the corresponding *graph* formula in Proposition 2.21. We do not prove the proposition here, because the proof follows very closely our argumentation above.

Proposition 2.21. *Let C be a constraint* grammar$([X_1, \dots, X_k], G)$*, where G is a grammar with starting symbol S. Then the following are equivalent:*

- *there is a support* $(d_1, \dots, d_k)$ *of C such that* $d_1 \dots d_k$ *can be derived with different derivations from S*
- *the formula* $graph(S, 1, k)$ *is not deterministic.*

We conclude that if the grammar is non-ambiguous then the resulting formula is in sd-DNNF. As a consequence we can then apply the model counting algorithm for sd-DNNF to count the number of supports of the grammar constraint. Unfortunately we cannot determine if a grammar is ambiguous in general, because this is an undecidable problem, and some context-free languages do not even have unambiguous grammars.

However, we can still use the transformation function *graph* for all grammars. We will argue that applying the model counting algorithm to results of the *graph* transformation on ambiguous grammars yields upper bounds for the number of solutions of the constraint and we still have that the upper bound of the solution count of a single assignment is non-zero iff the assignment is GAC, the property described in Observation 2.8.

In Proposition 2.22 we show that the formula obtained from $graph(L, l, r)$ counts the number of different derivations from L in interval $[l, r]$, denoted as $\#_{der}(L, l, r)$. Further we refer with $\mathsf{VAL}(graph(L, l, r))$ to the VAL value of the root node of $graph(L, l, r)$.

Proposition 2.22. *Let G be a grammar which contains the nonterminal symbol L. It holds*

$$\mathsf{VAL}(graph(L, l, r)) = \#_{der}(L, l, r)$$

Proof. The proof is again by induction on $i = r - l$ using Equation (2.12). Consider as *induction base* the singleton interval $[l]$, i.e., $i = 0$. In this case the formula was produced by the (updated) first case of Equation (2.12). Hence it is the disjunction of $G_{X_l=a}$ such that $a \in D_l$ and $L \longrightarrow a$ is a production in G. Obviously, $\mathsf{VAL}(G_{X_l=a}) = 1$ if and only if $a \in D_l$, thus

$$\mathsf{VAL}(graph(L, l, l)) = \{a \in D_l \mid L \longrightarrow a \text{ is production in } G\}|$$

which is also the number of different derivations for L in interval $[l, l]$.

For the *induction step* let $r - l = i > 0$. Now we compute the number of different derivations. Two derivations differ, if they use a different production rule $L \longrightarrow AB$ or the same rule applied to different intervals. Of course they are also different, if they differ in a later derivation step. All this is captured by the following recursive definition:

$$\begin{aligned}
\#_{der}(L, l, r) &= \sum_{L \longrightarrow AB} \sum_{l \le m < r} \#_{der}(A, l, m) \cdot \#_{der}(B, m+1, r) \\
&= \sum_{L \longrightarrow AB} \sum_{l \le m < r} \mathsf{VAL}(graph(A, l, m)) \cdot \mathsf{VAL}(graph(B, m+1, r)) \\
&= \sum_{L \longrightarrow AB} \sum_{l \le m < r} \mathsf{VAL}(graph(A, l, m) \wedge graph(B, m+1, r)) \\
&= \mathsf{VAL}\left(\bigvee_{L \longrightarrow AB} \bigvee_{l \le m < r} graph(A, l, m) \wedge graph(B, m+1, r) \right) \\
&= \mathsf{VAL}(graph(L, l, r))
\end{aligned}$$

■

This is a nice result, but as we will see later the number of derivations can differ greatly from the number of models. First we want to establish a similar bound for the PD values obtained by the model counting procedure of `sd-DNNF`. Our focus is again on the PD values of the leaves, because in the other translations $\mathsf{PD}(X_i = a)$ coincides with the solution counts under the condition of an assignment $sc(C, X_i, a)$. In Proposition 2.23 we relate the PD values again to the number of derivations instead of the number of models. From now on we denote with $\#^*_{der}(L, l, r)$ the overall number of derivations when we fix a subderivation for L in interval $[l, r]$.

Proposition 2.23. *Let t be an or-node in the formula $graph(S, 1, k)$ corresponding to $graph(L, l, r)$. Then it holds:*

$$\mathsf{PD}(t) = \#^*_{der}(L, l, r)$$

Proof. We prove this statement by induction on the depth d of t in the formula $graph(S, 1, k)$. First let $d = 0$, thus t is the root and $\mathsf{PD}(t) = 1$ by definition. On the other hand, $\#^*_{der}(S, 1, k)$ is also 1, because if we fix a subderivation on the root, we fix the complete derivation. For the *induction step* assume t to be the root of $graph(L, l, r)$ and the corresponding or-node is not the root (has depth greater 0). We show the proposition by calculating $\mathsf{PD}(t)$ explicitly. First, by definition we have $\mathsf{PD}(t) = \sum_{N \text{ parent of } t} \mathsf{CPD}(N, t)$. By construction of the *graph* formula all parents N are and-nodes. In the *graph* formula, we can uniquely identify and-nodes by their two children $graph(L, l, m)$ and $graph(R, m+1, r)$, more precisely by a 5-tuple (L, R, l, m, r). Note further, that all and-nodes are binary and that t might be the left or the right child of its parent. So for $\sum_{N \text{ parent of } t} \mathsf{CPD}(N, t)$ we find by definition of the CPD values:

$$\begin{aligned} &\sum_{N \text{ parent of } t} \mathsf{CPD}(N, t) \\ =& \sum_{N=(L,X,l,r,r')} \mathsf{PD}(N) \cdot \mathsf{VAL}(graph(X, r+1, r')) + \\ &\sum_{N=(X,L,l',l-1,r)} \mathsf{PD}(N) \cdot \mathsf{VAL}(graph(X, l', l-1)) \end{aligned} \tag{2.16}$$

By construction, all and-nodes have or-nodes as parents, so the PD value can be calculated as sum of the PD values of the parents. An and-node (L, R, l, m, r) has or-nodes as parents that are the root of $graph(L_i, l, r)$ such that it exists a rule $L_i \longrightarrow LR$ and a number m such that $graph(L, l, m)$ and $graph(R, m+1, r)$ are satisfiable. Thus we can replace $\mathsf{PD}(N)$ and obtain:

$$\sum_{N=(L,X,l,r,r')} \left(\sum_{L_i \longrightarrow LX} \mathsf{PD}(graph(L_i,l,r')) \right) \cdot \mathsf{VAL}(graph(X,r+1,r'))+$$
$$\sum_{N=(X,L,l',l-1,r)} \left(\sum_{L_i \longrightarrow XL} \mathsf{PD}(graph(L_i,l',r)) \right) \cdot \mathsf{VAL}(graph(X,l',l-1)) \tag{2.17}$$

Expanding the inner sum we obtain:

$$\sum_{N=(L,X,l,r,r')} \sum_{L_i \longrightarrow LX} \mathsf{PD}(graph(L_i,l,r')) \cdot \mathsf{VAL}(graph(X,r+1,r'))+$$
$$\sum_{N=(X,L,l',l-1,r)} \sum_{L_i \longrightarrow XL} \mathsf{PD}(graph(L_i,l',r)) \cdot \mathsf{VAL}(graph(X,l',l-1)) \tag{2.18}$$

Now, we can apply the induction hypothesis to the PD term in the first factor, because these or-nodes have lower depth in the formula. Further we can apply Proposition 2.22 to the VAL terms.

$$\sum_{N=(L,X,l,r,r')} \sum_{L_i \longrightarrow LX} \#^*_{der}(L_i,l,r') \cdot \#_{der}(X,r+1,r')+$$
$$\sum_{N=(X,L,l',l-1,r)} \sum_{L_i \longrightarrow XL} \#^*_{der}(L_i,l',r) \cdot \#_{der}(X,l',l-1) \tag{2.19}$$

We look in more detail what we index the sums with. We consider only the first sum, the other case is analogous. Basically every pair $N = (L,X,l,r,r')$ and $L_i \longrightarrow LX$ corresponds to a class of derivations which contain the pattern depicted in Figure 2.8 where we also label the nodes with the intervals that they derive.

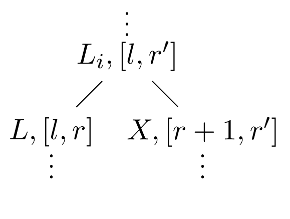

Figure 2.8: Detail of a derivation containing L.

In each of these patterns the number of derivations when a derivation for L is fixed can be computed easily. We know that there are $\#^*_{der}(L_i,l,r')$ possible derivations when

L_i is fixed and to fix L_i we have $\#_{der}(X, r+1, r')$ possibilities (since L is already fixed). Hence, the product of the two numbers is exactly the number of derivations when L is fixed. Since the classes of derivations are disjoint, we have to sum up the products for all classes, arriving at the term in Equation (2.19) as the desired number $\#^*_{der}(L, l, r)$. ■

In a similar way we can argue that the PD value of $G_{X_i=a}$, child of only or-nodes, is exactly the number of derivations where the i-th letter is a. Accordingly, also $\mathsf{PD}(X_i = a)$ is the number of derivations with $X_i = a$. Knowing this, we can conclude that counting the derivations maintains GAC according to Observation 2.8.

$$\begin{aligned} & \mathsf{PD}(X_i = a) = 0 \\ \Longleftrightarrow\ & \text{number of derivations with } X_i = a \text{ is } 0 \\ \Longleftrightarrow\ & a \text{ can be pruned from } D_i \end{aligned}$$

We have now proven that the numbers we calculate are upper bounds that are exact in case of nonambiguous grammars and can still be used to achieve GAC on the `grammar` constraint. Now we want to investigate how much the bounds can deviate from the real value. This problem is equivalent to the question regarding the relation between the number of words of fixed length k and number of derivations for these words. It turns out that the grammar in Example 2.20 exhibits already exponential behavior. Recall that the grammar had the only rule $S \longrightarrow SS \mid s$. The grammar contains exactly one word of length k for $k \in \{1, 2, \ldots\}$, but as we have seen for $k = 3$ there are 2 derivations. Let a_k be the number of different derivations for the word of length k. Clearly, $a_1 = a_2 = 1$ and $a_3 = 2$. For calculating a_k for $k > 3$ we can set up a recurrent relation based on the following observation: Applying the rule $S \longrightarrow SS$ the first time divides the word we want to derive into 2 nonempty and disjoint sequences with length k_1 and k_2 such that $k_1 + k_2 = k$. For length k we have $k - 1$ possibilities for this separation. Since $k_1, k_2 < k$ a_{k_1} and a_{k_2} are known, i.e., the number of different derivations is known for lengths k_1 and k_2. For each possible separation the number of different derivation is the product of a_{k_1} and a_{k_2} because they are independent. The sum of these products over all separations yields then a_k. The situation for $k = 4$ is shown in Figure 2.9. The general recurrency relation for number a_k is given in Equation (2.20).

$$a_k = \sum_{i=1}^{k-1} a_i \cdot a_{k-i} \tag{2.20}$$

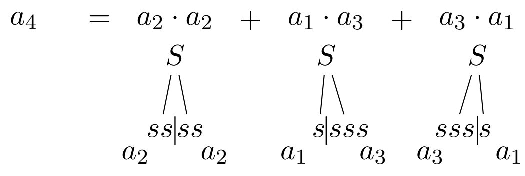

Figure 2.9: Possible separations for a word of length $k = 4$.

It turns out that this sequence is the well-known sequence of CATALAN numbers[2] that occur in various combinatorical counting problems [CG98]. The first numbers of the sequence are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, ... As we can see they grow rapidly, in fact exponentially as

$$a_{k-1} \sim \frac{4^k}{k^{3/2} \cdot \sqrt{\pi}} \tag{2.21}$$

So we cannot expect a nice bound in every case. On the other hand the language of this grammar is trivially regular and can be modelled as well with $S \longrightarrow sS \mid s$ yielding exact model counting. So the hope is that, in practice, these pathological examples can be avoided or minimized with little effort during modelling.

In contrast to the compilation of `regular` and `grammar` constraints, the transformation using the *graph* function makes use of full expressiveness of the `sd-DNNF` formalism (provided that the grammar is non-ambiguous), i.e., the decision property is not fulfilled in general.

We conclude this section with an examination of the size of the formula obtained from $graph(S, 1, k)$ for a grammar G. First we note that every or-node corresponds to parameters L, l, r of the *graph* function, thus there are maximal $|G| \cdot n^2$ or-nodes. Using our earlier observation that an and-node is uniquely identified by a 5-tuple (L, R, l, m, r) we can give an upper bound of the number of and-nodes. Obviously, both l, m, and r are in $O(n)$, and the number of combinations of L and R is bound by the number of productions, $|G|$. So the size of the formula is in $O(|G| \cdot n^3)$, which is also reported in [QW07].

2.4 Conclusions

In this chapter we introduced one of our main contributions, namely our knowledge compilation approach to counting the number of solutions in individual constraints. We provided compilations of expressive constraints such as `extensional`, `regular`, and `grammar` to polynomially sized formulas in `sd-DNNF` and make use of `sd-DNNF`s linear time

[2]The CATALAN numbers are sequence A000108 in the On-line Encyclopedia of Integer Sequences (http://www.research.att.com/~njas/sequences/A000108).

model counting algorithm. However, the approach is not limited to those constraints: on the one hand we easily can give compilations for other constraints such as `element`, and on the other hand other useful constraints such as `among` are covered by them. Moreover, there are even automatic compilers from `CNF` to `sd-DNNF` [Dar04].

3 Counting-Based Heuristics

Having introduced algorithms that count solutions for a set of various constraints, it remains to show how to exploit the extra-information we gain by counting. Except for [ZP07] who propose some heuristics based on solution counting in single constraints, we are not aware of any other works that did so. In [KDG04] and [HKBM07] they try to estimate the probability of an assignment in the whole problem and guide search with these numbers. However, we will concentrate on the first approach.

In this chapter we present some heuristics divided into two groups: heuristics that choose simultaneously the variable and the value (Section 3.1), and "real" value orderings (Section 3.2). For each heuristic we give a justification that tries to explain why the heuristic should work. Considering heuristics it is clear that these justifications are rather hypotheses, because experiments can only support or refute them, but never prove. In Chapter 4 we compare the results of the experiments against the here given hypotheses.

In the following we will make several times the assumption that the constraints are independent. The assumption implies that implications of one constraint never affect other constraints, which is certainly not true in general. However, the dependence relation between constraints is unknown and hard to determine, hence this assumption is often made.

3.1 Constraint-Centered Heuristics

Constraint-centered heuristics were first introduced in [ZP07]. Instead of looking accurately at the properties of a variable (degree or domain size) or a value, they try to take a decision by analyzing the constraints themselves. The intuition is that in presence of global constraints the constraints provide on the one hand good modelling and propagation properties, but on the other hand also more information about the structure of the problem. This information can be made available for example by solution counting. In this section we describe the heuristics MinSC/MaxSD and MaxSD [ZP07]. In addition we propose an alternative to MaxSD that is supported by another mathematical explanation.

We observe that we loose the strict separation between variable and value selection in these heuristics, but we can plug them into the searching framework easily.[3]

[3]In fact it has a larger impact, since we are switching from *n-way branching* to *2-way branching*. We

MaxSD

Intuitively it is preferable to make decisions with high solution densities, because they are more likely to lead to a solution. So the MaxSD heuristic [ZP07] selects the assignment with the highest solution density $sd(C, X_i, a)$ with respect to some constraint C, more formally it selects the assignment $X_i = a$ with:

$$(X_i, a) = \underset{X_i \in V, d \in D_i}{\text{argmax}} \left(\max\{sd(C, X_i, d) \mid C \text{ with } X_i \in scope(C)\}\right) \tag{3.1}$$

This heuristic is very coarse, since it relies only on one specific information, the *maximum* density. To clarify the problem we assume a CSP containing two constraints C_1 and C_2, each with X_1, X_2 in its scope. Assume further solution densities $sd(C_1, X_1, a) = 0.95$, $sd(C_1, X_2, b) = 0.9$, $sd(C_2, X_1, a) = 0.1$, and $sd(C_2, X_2, b) = 0.9$. MaxSD would select $X_1 = a$ since this assignment maximizes the solution density for *one* constraint, whereas a more reasonable choice would be $X_2 = b$. Further MaxSD considers all constraints being equal. In the following we describe several ways to refine MaxSD: we can (1) take the number of solutions of the constraints into account (MinSC/MaxSD), or (2) consider also information of tightness of the constraint (MaxSD$_2$), or (3) use MaxSD only as a value ordering and let a variable heuristic select the variable, see Section 3.2.

MinSC/MaxSD

MinSC/MaxSD [ZP07] is more sophisticated and restricts its attention to the variables that occur in the tightest constraints, i.e., the constraints that have the least number of solutions. Among those variables MinSC/MaxSD chooses the assignment that has the highest solution density. The intuition is that constraints with only a few solutions possibly indicate parts of the CSP that are hard to satisfy. We expect this heuristic to work well in problems with many similar constraints, like for example nonograms, sudoku or latin square problems. In problems consisting of several different constraints, considering only the number of solutions might be an improper measure for tightness of a constraint; particularly, a constraint with currently 20 and initially 1000 solutions might be regarded tighter than a constraint with 10 solutions that had 20 in the beginning.

MaxSD$_2$

We showed in Chapter 2 that solution densities can be considered as probabilities in the sense that $sd(C, X_i, a)$ is the probability that $X_i = a$ given that the constraint is fulfilled:

$$sd(C, X_i, a) = P(X_i = a \mid C) \tag{3.2}$$

will comment on this later.

However, we are more interested in $P(C \mid X_i = a)$, the probability of C being satisfied under the condition that a is assigned to X_i. It turns out that this is proportional to $P(X_i = a \mid C)$ if constraint C and variable X_i are fixed:

$$\begin{aligned} P(C \mid X_i = a) &= \frac{P(C \cap X_i = a)}{P(X_i = a)} \\ &= \frac{P(X_i = a \mid C) \cdot P(C)}{P(X_i = a)} \end{aligned} \tag{3.3}$$

First we apply Bayes' Law twice, then we observe that $P(X_i = a)$ is constant and so is $P(C)$ (since it is the looseness of C). In order to select an assignment $X_i = a$ that maximizes the probability that some constraint is fulfilled we should maximize $P(C \mid X_i = a)$ with respect to C, X_i and a. Assuming $P(X_i = a)$ is a constant k and $\mathit{scope}(C) = \{X_1, \ldots, X_k\}$ we can simplify the expression in Equation (3.3) further:

$$\begin{aligned} \frac{P(X_i = a \mid C) \cdot P(C)}{P(X_i = a)} &= \frac{sd(C, X_i, a) \cdot \frac{sc(C)}{|D_1 \times \ldots \times D_k|}}{k} \\ &= \frac{sc(C, X_i, a)}{k \cdot |D_1 \times \ldots \times D_k|} \end{aligned} \tag{3.4}$$

We denote the heuristic that maximizes the term in Equation (3.4) with MaxSD_2. Although it does not contain the solution densities anymore we stick to the notation, because it is based on a similar idea to MaxSD.

3.2 Counting-Based Value Orderings

The original motivation of our work was to find better value orderings. In this section we consider the value selection from various points of view and introduce some new counting-based orderings. In contrast to the constraint-centered approaches that we described above, we try to combine the solution densities $sd(C, X_i, a)$ for different constraints C to have a more global view. We assume in the following that a variable X_i is already selected by a variable heuristic and occurs in the scope of the constraints $C_1, \ldots, C_k$.

3.2.1 Value Selection as Winner Determination

One point of view is to consider the process of value selection as winner determination or social choice [Arr70] in an *election*. Here the constraints act as *voters* and the values are the *parties*. Each constraint gives his vote in form of the solution density, i.e., a solution density of 0.8 for value a constitutes a high preference for this value whereas a density of 0.01 is a low preference. Typically there are several constraints that give their preferences and the task of determining a good value is interpreted as selecting a value that "matches the preferences best". The outcome of the election is a total order $\prec$ that represents the places (smallest element is winner and so on). The corresponding value ordering is then exactly $\prec$.

To make it more formal assume that variable X_i is selected and $D_i = \{d_1, \dots, d_k\}$, thus X_i has domain size of k. Further let X_i be in the scope of m constraints $C_1, \dots, C_m$. Now, $sd(C_j, X_i, d_l)$ is the preference of constraint C_j, $1 \leq j \leq m$, for value $d_l \in D_i$. We denote this also with p_{jl}. There are several possibilities to determine the outcome of an election of this form, but we restrict our attention to the following idea: We look for values $q_1, \dots, q_k$ such that they are as close as possible to all the constraints preferences, i.e., we try to minimize the distance of $q_1, \dots, q_k$ to all the preferences:

$$f(q_1, \dots, q_k) = \sum_{j=1}^{m} \left(\sum_{l=1}^{k} (q_l - p_{jl})^2 \right) \longrightarrow \min \tag{3.5}$$

For solving this, we differentiate f partially with respect to every q_l and determine the roots of the result:

$$\frac{\partial f}{\partial q_l} = \sum_{j=1}^{m} 2 \cdot (q_l - p_{jl}) = 0 \tag{3.6}$$

Hence, we get for q_l, $1 \leq l \leq k$ the arithmetic mean of the constraints' preferences for value d_l:

$$q_l = \frac{p_{1l} + \ldots + p_{ml}}{m} \tag{3.7}$$

We refer with MaxMean to the heuristic that selects d_l such that q_l is maximal.

A disadvantage of this approach might be that the votes of the constraints are treated equally. In general the constraints differ for example with respect to their tightness and their importance in the problem. Similar to the justification for MinSC/MaxSD, we argue that a tight constraint should have a higher influence on the result of the election. On the other hand we can measure the hardness of a constraint with its weighted degree (see Section 1.3.3), i.e., the more often a constraint fails during search the higher should be its influence for determining a value. We model different influences by attaching a weight w_j to every constraint C_j and modify Equation (3.5) to minimize the *weighted distance* $f_w(q_1, \dots, q_k)$:

$$f_w(q_1, \dots, q_k) = \sum_{j=1}^{m} \left(w_j \cdot \sum_{l=1}^{k} (q_l - p_{jl})^2 \right) \longrightarrow \min \tag{3.8}$$

We solve this with the same approach as Equation (3.5) and obtain the weighted arithmetic mean:

$$q_l = \frac{w_1 p_{1l} + \ldots + w_m p_{ml}}{w_1 + \ldots + w_m} \tag{3.9}$$

The value ordering obtained by this equation selects d_l such that q_l is maximal. We refer to this heuristics with MaxWMean.

Of course, since both in Equation (3.7) and in Equation (3.9) the denominator is equal for every q_l it is enough to maximize the (weighted) sum in the numerator. As an relaxation of this, we can regard only the constraint(s) with the heighest weight(s) and select the value following their solution densities.

In this section we proposed only few orderings based on the view as an election, but in principle we can reuse all the results from social choice theory [Gae06] and try to find approaches there whose properties match the requirements of a good value selection.

3.2.2 Minimal Search Space Reduction

Typically we want to select the value that prunes the search space least, because the probability for a domain wipe-out decreases. Assuming pairwise independence of the constraints X_i occurs in, we can easily calculate the search space reduction that is enforced. When we assign value a to X_i we remove $sc(C_j) - sc(C_j, X_i, a)$ solutions from a constraint C_j. By multiplying this with the product of the domain sizes of the variables not in the scope of C_j we obtain the number of tuples that were removed from the search space. If we assume the constraints to be independent we can sum over all constraints and get an estimation for the search space reduction:

$$\sum_{j=1}^{k} \left((sc(C_j) - sc(C_j, X_i, a)) \cdot \prod_{X_l \notin scope(C_j)} D_l \right) \tag{3.10}$$

The assignment that minimizes this expression is the one that removes least tuples from the search space (given independence). So we want to calculate:

$$\operatorname*{argmin}_{a \in D_i} \left(\sum_{j=1}^{k} (sc(C_j) - sc(C_j, X_i, a)) \cdot \prod_{X_l \notin scope(C_j)} D_l \right) \tag{3.11}$$

We can simplify this expression further. For the sake of simplicity we write $Q(j)$ for the product expression in the end of Equation (3.11). Then we have:

$$\operatorname*{argmin}_{a \in D_i} \left(\sum_{j=1}^{k} (sc(C_j) - sc(C_j, X_i, a)) \cdot Q(j) \right) \tag{3.12}$$

$$= \operatorname*{argmin}_{a \in D_i} \left(\sum_{j=1}^{k} sc(C_j) \cdot Q(j) - \sum_{j=1}^{k} sc(C_j, X_i, a) \cdot Q(j) \right) \tag{3.13}$$

$$= \operatorname*{argmin}_{a \in D_i} \left(- \sum_{j=1}^{k} sc(C_j, X_i, a) \cdot Q(j) \right) \tag{3.14}$$

$$= \operatorname*{argmax}_{a \in D_i} \left(\sum_{j=1}^{k} sc(C_j, X_i, a) \cdot Q(j) \right) \tag{3.15}$$

Suprisingly, on this way we obtain an instance of the MaxWMean heuristic in Equation (3.9), namely when we choose the weights to reflect the size of the search space that is “admitted” by a single constraint C_j, i.e., $w_j = Q(j) \cdot sc(C_j), 1 \leq j \leq k$. Intuitively, this means that constraints C_j with small scope, i.e., big $Q(j)$, have a bigger influence, since pruning some values in them deletes more tuples from the search space.

3.2.3 Value Orderings based on Probabilities

In this section we look at value selection from the angle of probabilities. In general we want to maximize the probability of arriving at a solution with the selected assignment. Given that X_i occurs in the constraints $C_1, \ldots, C_k$, we try to maximize

$$P(C_1 \cap \ldots \cap C_k \mid X_i = a) \tag{3.16}$$

This expression is not calculable, since we do not have the joint probability distribution of $(X_i = a) \cap C_1 \cap \ldots \cap C_k$. One approach to give an estimation is to assume conditional independence of $C_1, \ldots, C_k$ given $X_i = a$ and to consider the probabilities separately, i.e., maximize the following expression:

$$P(C_1 \mid X_i = a) \cdot \ldots \cdot P(C_k \mid X_i = a) \tag{3.17}$$

Since X_i is fixed we have $P(X_i = a) = \frac{1}{|D_i|}$. Reformulating the conditional probabilities according to Equations (3.3) and (3.4) we obtain the following expression to maximize (with respect to a):

$$\prod_{j=1}^{k} \frac{sc(C_j, X_i, a) \cdot |D_i|}{|D_{C_j}|} \tag{3.18}$$

where D_{C_j} is the set of all possible tuples in the scope of C_j. X_i and thus, the constraints $C_1, \ldots, C_k$ being fixed, we end up in selecting a such that

$$a = \underset{d \in D_i}{\operatorname{argmax}} \prod_{j=1}^{k} sc(C_j, X_i, d) \tag{3.19}$$

We denote the heuristic that selects the value in terms of Equation (3.19) with MaxSCProd. Analogously, we can explain a heuristic MaxSDProd that selects a such that

$$a = \underset{d \in D_i}{\operatorname{argmax}} \prod_{j=1}^{k} sd(C_j, X_i, d) \tag{3.20}$$

Finally, we can approximate the expression in Equation (3.20) in two ways: On the one hand a heuristic could try to favor these values that have a very high solution density in one constraint. This is MaxSD deployed only as a value ordering. On the other hand a heuristic could try to avoid very small solution densities, because these indicate low probability of finding a solution. Hence we choose the value a that has the highest minimal density in all constraints, a heuristic that we refer to with MaxMinSD:

$$a = \underset{d \in D_i}{\operatorname{argmax}} \left(\min\{sd(C_j, X_i, d) \mid 1 \leq j \leq k\}\right) \tag{3.21}$$

4 Evaluation

4.1 Setting

4.1.1 Environment

Both the heuristics and the solution counting algorithms are implemented using the constraint solver CaSPER [CBA05] on an Intel Dual Core 1.6 GHz with 1 Gb of RAM using Ubuntu Linux 7.10 and g++ 4.3.1. The advantages of CaSPER are its flexibility, ease of understanding, changing and extending. This is important, because some of the techniques need, as we will see, considerable effort in implementation and are not standard in any constraint solver.

4.1.2 Search Algorithm

We use the backtrack search algorithm presented in Section 1.3.1 in all our experiments. We pointed out that there are several possibilities to implement look-ahead and look-back techniques, as well as variable and value orderings. Since the experiments we conduct concern only the variable and especially the value orderings we will plug in different orderings there. We do not use any look-back technique whereas we install maintaining GAC as look-ahead strategy. Algorithm 4.1 shows the search algorithm with template functions for variable (Line 3) and value ordering (Line 6).

4.1.3 Implementation of Solution Counting using `sd-DNNF`

We mentioned a few times the difference between using the formula $G_{X_i=a}$ instead of just the Boolean variable $(X_i = a)$ in our compilation into `sd-DNNF`. Basically, it ensured determinism in the sense that the propositional variables $(X_i = a)$ and $(X_i = b)$ do not contradict each other, whereas formulas $G_{X_i=a}$ and $G_{X_i=b}$ do for $a \neq b$. In the presence of a constraint store the situation changes, since the assignments (in the solver not anymore Boolean variables) $X_i = a$ and $X_i = b$ are contradictory. Further we note that we never translate into `sd-DNNF` based on negative literals, but only on positive ones.

These observations lead to the simplification that we can replace subformulas $G_{X_i=a}$ with the simple $(X_i = a)$ (and thus omit negative literals as labels for the leaves) in the implementation. Other problems we have to face are correct updating of VAL and PD values and ability to backtrack. In particular we have to make sure that the values are recalculated whenever and only when necessary and that they are set back to the

```
Algorithm: search($P, v$)
input  : A CSP $P$ with variables $X_1, \ldots, X_k$ and a partial assignment $v$
output: A solution to the CSP or failure
1  if all variables have an value assigned then
2      return $v$;
3  $X_i \leftarrow$ selectVariable($X_1, \ldots, X_k$);
4  $D \leftarrow D_i$;
5  while $D$ not empty do
6      $d \leftarrow$ selectValue($D$);
7      $D \leftarrow D \setminus \{d\}$;
8      $v \leftarrow v \cup \{X_i \leftarrow d\}$;
9      if maintaining GAC does not fail then
10         if search($P, v$) succeeds then return solution;
11     $v \leftarrow v \setminus \{X_i \leftarrow d\}$;
12 return failure;
```

Algorithm 4.1: The search algorithm we use in the experiments.

correct values after occurrence of failure during backtracking. We explore two ways of implementing the propagator, a *decomposition* and a *monolithic* propagator. Another way could be a hybrid between a SAT solver (for propagation) and a special procedure for counting. However, we give only a sketch of the idea, since we did not implement it.

Decomposition

It turns out that using built-in finite domain variables for VAL and PD values together with an apropriate set of propagators to update them meets the requirements with respect to updating and backtrackability. More precisely, we define two propagators that maintain bounds consistency on constraints of the form $X = X_1 \cdot \ldots \cdot X_n$ and $X = X_1 + \ldots + X_n$, respectively. Looking at the rules for computation of VAL and PD values (Equations (2.4), (2.5), and (2.6)) shows that these propagators are sufficient to propagate the numbers within the DAG, because in fact we are only interested in the upper bound of the variables. For example consider an and-node N with children $C_1, \ldots, C_k$, parents $P_1, \ldots, P_l$, and corresponding finite domain variables. We create other variables $\mathsf{CPD}_1, \ldots, \mathsf{CPD}_l$ that represent the CPD values with respect to each of the parents and post the following constraints:

- $\mathsf{VAL}(N) = \mathsf{VAL}(C_1) \cdot \ldots \cdot \mathsf{VAL}(C_k)$ for updating the VAL value of N, and
- $\mathsf{PD}(N) = \mathsf{CPD}(P_1) + \ldots + \mathsf{CPD}(P_l)$ for updating the PD value of N, along with

- analogous constraints for calculating the CPD variables.

Handling of the leaves is also straightforward. For a leaf ($X_i = a$) with corresponding finite domain variables VAL and PD we post:

- $X_i = a \Longleftrightarrow$ VAL, the definition of VAL for leaves,
- PD $= 0 \Longrightarrow X_i \neq a$, the inconsistency definition
- a constraint that calculates the PD value (analogously to above).

The approach is elegant, because we make direct use of the propagation engine of the solver and do not have to care about correct updating or abilitity to backtrack. Note that we *decompose* the actual (structured) constraint into many smaller (uniform) constraints, thus we refer to this technique as decomposition.

Decomposing the constraint, the possibilities for using weighted degree heuristics change. Recall that every constraint had a weight assigned that expressed how often it fails. Decomposing the constraint into many smaller constraints we lose the constraint at first sight and cannot weigh it anymore. The solution we propose is to write a wrapper around the decomposition, which is then weighted and whose degree is increased if any of the constraints inside fails. In fact, we gain flexibility here, since we can even choose the number by which the weight is increased: either we increase it by one or by the number of failing constraints inside. In our experiments, we chose the first option.

The decomposition is a very direct solution, but has several disadvantages. First, since the graphs we create may become very large, we have to create lots of finite domain variables and propagators: first experiments showed that we created around 300.000 both variables and propagators for the smallest instances of the benchmark. The second disadvantage is that we do not have control over the order the propagators are woken up. In first experiments we discovered that some of the propagators are invoked many times during maintaining GAC for an individual constraint. In the optimal case every propagator is woken up exactly once. To attain this aim best, we take the control over the update ourselves and implemented a monolithic version, which has also the advantage of less propagators and finite domain variables.

Monolithic propagator

The monolithic variant of the propagator makes use of CaSPER's *reversible* variables. Basically, reversible variables are linked to the actual solver object and are set back to their correct values on backtracking, very similar to finite domain variables. However, they do not feature a domain, but have only one fixed value. The deployment of these variables ensures the desired property backtrackability. As in the decomposition we assign two (reversible) variables to every node in the And/Or-structure, which represent the VAL and PD values of the node. Additionally we assign a fixed number, the *level*, to every node, which is the maximal distance to the root (note that, since we use shared substructures, in general there is more than one way from the root to the node). Thus the root has level 0, its children level 1, and so on.

We post several (small) filters that wake up on domain changes of the problem's variables and one (big) filter for the solution counting constraint. The small filters provide the big filter with information, which leaf has changed its VAL to 0. The big filter in turn updates level by level, starting with the highest, the VAL values of the nodes. We stop this process at the root or if nothing changes anymore. Using queues for every level in combination with marks efficiently ensures that every node is touched at most once in this run. Also during the first run we mark the nodes where the PD values possibly changed. The graph is traversed starting from lowest level of change, and the PD values in the nodes are updated. Again, if $\mathsf{PD}(N) = 0$ for some leaf N we prune the domain of the attached variable. The filter fails if either the constraint has no more solutions, i.e., $\mathsf{VAL}(root) = 0$, or a domain wipe-out occurs.

Hybrid approach

A disadvantage of both the decomposition and the monolithic propagator is that propagation is only possible by updating all solution countings. Moreover, for achieving GAC on the problem the same propagator might be called several times, because other constraints could infer more information. A hybrid approach could be to export propagation to the unit propagation engine of a SAT solver and perform model counting only when necessary. In particular, the solution counting would have to be performed only once. Generalized arc-consistency on a (`sd-`)`DNNF` formula can be achieved by doing merely unit propagation on suitable clauses [JBKW08].

4.1.4 Evaluation Parameters

We compare our results with respect to two parameters. Of course, time is the most important parameter for assessing an approach, but on the other hand it is clear that counting solutions opposed to pure propagating is computational overhead. Hence it might be, that we do not see positive effects of counting based heuristics looking only at the time needed for solving the problem. We seek information from a second parameter, namely the number of backtracks. This is an accepted parameter for measuring the size of the search tree explored during search as requiring less backtracks indicates a smaller search tree.

4.2 Rostering Benchmark

4.2.1 Problem Description

In order to assess our heuristics in the presence of `grammar` constraints we try to solve rostering problems. The benchmark we describe was originally proposed in [QR07][4] and since then has been considered many times in works about global `grammar` constraints, for example in [QW07]. The problem is to design a work schedule for employees in a multi-activity environment. Parameters of the problem are the number of employees e, the number of different activities k and the demands of the activities at certain times. We denote the activities with $a_1, \dots, a_k$ and the demand of activity a_i at time t with $d(a_i, t)$. The schedule of an employee is subject to the following restrictions:

- an employee can rest (symbol r), have lunch (l), have break (b) or work on an activity (a_i)
- an employee works either full-time (six to eight hours) or part-time (three to six hours)
- full-time workers have to have a break, a lunch and another break; part-time workers only a break
- a normal break is 15 minutes, the lunch break lasts one hour
- once started an activity an employee has to continue it for at least one hour and can only change after a break or lunch
- employees rest in the beginning and at the end of the day

Originally the problem was formulated as an optimization problem. Whenever an employee performs an activity that is not required by the demands or not all demands are fulfilled, additional costs become due. The objective function is to minimize the sum of all costs. However, we consider a relaxation of this problem, where all demands have to be fulfilled and every performed activity has cost 1. We try to minimize the overall costs as in [QW07].

4.2.2 Modelling

We model the day as a sequence of 96 time slots, i.e., every slot corresponds to 15 minutes. We can make this sequence subject to a `grammar` constraint that describes the possible schedules. In Figure 4.1 we give the production rules for this grammar.

[4]Many thanks to Louis-Martin Rousseau and Claude-Guy Quimper who provided us with the original instances of the benchmark.

$$\begin{array}{lll} S \longrightarrow RFR \mid RPR & F \longrightarrow PLP & P \longrightarrow WbW \\ L \longrightarrow lL \mid l & R \longrightarrow rR \mid r & W \longrightarrow A_i \\ A_i \longrightarrow a_iA_i \mid a_i & & \end{array}$$

Figure 4.1: Productions describing an employee's schedule.

The symbols P, F, W, L and R stand for part-time worker, full-time worker, work, lunch and rest, respectively, and A_i denotes the nonterminal for activity a_i. The starting symbol is S as usual. The rules are quite intuitive, for example, the initial rule for S expresses that the employee either works full-time or part-time and rests before and after.

Note that this grammar is not in Chomsky normal form, but can easily be transformed into it. The grammar is (and stays during the transformation to Chomsky normal form) non-ambiguous, because every word can only be derived in exactly one way. So our algorithm provides even exact solution countings.

The rules in Figure 4.1 alone model only the structure of an employee's working day. However, they do not ensure any of the time restrictions, e.g., that a part-time employee works only between three and six hours. To guarantee those we attach conditions to some of the rules. As an example we add the function $f_W(i,j) = (j \geq 4)$ to production $W \longrightarrow A_i$ in order to assure that the same activity is conducted for at least one hour. To production $P \longrightarrow WbW$ we add the Boolean function $f_P(i,j) = (13 \leq j \leq 25)$ expressing that part-time means working between three and six hours (between 12 and 24 slots) plus 15 minutes break (one slot). In this way we can condition the productions such that the grammar describes exactly the requirements on the schedule of an employee.

We model the demands $d(a_i,t)$ with `regular` constraints using an appropriate counting automaton for every time slot t. Its states are associated with tuples that store the number of occurrences of the activities a_i, i.e., a state $(n_1,\ldots,n_k)$ encodes that the automaton read n_i times a_i for $1 \leq i \leq k$. Clearly, the starting state is $(0,\ldots,0)$ and $(d(a_1,t),\ldots,d(a_k,t))$ is the final state. Transitions between two states correspond exactly to the counting. Formally, we have for a fixed time t and k activities the automaton given by Equations (4.1) to (4.6).

$$\Sigma = \{l, b, r, a_1, \ldots, a_k\} \tag{4.1}$$
$$Q = \{(n_1,\ldots,n_k) \mid 1 \leq n_i \leq d(a_i,t), 1 \leq i \leq k\} \tag{4.2}$$
$$q_0 = (0,\ldots,0) \tag{4.3}$$
$$F = \{(d(a_1,t),\ldots,d(a_k,t))\} \tag{4.4}$$
$$\delta((n_1,\ldots,n_i,\ldots,n_k),a_i) = (n_1,\ldots,\min(n_i+1,d(a_i,t)),\ldots,n_k) \tag{4.5}$$
$$\delta((n_1,\ldots,n_k),\{l,b,r\}) = (n_1,\ldots,n_k) \tag{4.6}$$

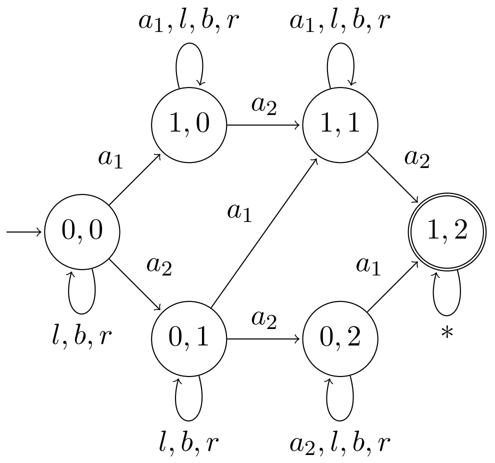

Figure 4.2: Counting automaton for demands $d(a_1, t) = 1$ and $d(a_2, t) = 2$.

In Figure 4.2 we show the counting automaton for two activities with demands 1 and 2, respectively. Obviously, the number of states grows fast with $\prod_{i=1}^{k} d(a_i, t)$, because it is the construction of a product automaton. Nevertheless the approach is applicable, because for this benchmark k is usually small and the sequence of constrained variables (number of employees) short. Note that our model differs from the model in [QW07] in that they model the demands with separated cardinality constraints.

Assuming e employees and k activities we can implement the model in the following way. We create a two-dimensional $e \times 96$ array X of finite domain variables with domains $\{r, l, b, a_1, \ldots, a_k\}$ (or equivalently $[0, k+2]$). Variable X_{ij} corresponds now to the j-th time slot of employee i (with $1 \leq i \leq e$ and $1 \leq j \leq 96$). Then we post a `grammar` constraint describing the schedule on every row and a `regular` constraint with the respective counting automaton on every column. For breaking symmetries we post lexicographical ordering constraints on the rows, i.e., the first row should be smaller or equal to the second, and so on.

4.2.3 Experiments and Results

In this setting, an excellently working static variable and value ordering is known. It instantiates the array of variables from left to right and from top to bottom and enumerates values in the order r, l, b, a_1, …, a_k [QW07]. We denote this heuristic in the following with Static. Sometimes we refer with Static only to the variable ordering from left to right, but it will be clear from context what is meant. We denote the value ordering of Static with Min (since in our implementation the domain is ordered in exactly this way), and the reverse order with Max.

For justification of the Static heuristic consider the following example.

Example 4.1. *Assume an instance with five employees and two activities and a time point t such that $d(a_1, t) = d(a_2, t) = 2$. Assume further the heuristic instantiated all variables X_{ij} with $j < t$ to 0, i.e., the employees rest, and the next variable to instantiate is X_{1t}. Now, Static selects value 0, GAC on the columns implies the other variables to take values a_1 or a_2. After setting X_{2t} and X_{3t} to a_1 (since a_1 is the least value in their domains) GAC infers that $X_{4t} = X_{5t} = 4$. Hence also the lexicographical constraints are consistent. Moreover, because an employee has to work at least one hour on one activity the next three columns are already determined by GAC on the rows.*

Summarizing we can say that the static ordering works well because it tries to assign working activities as late as possible, obeys the lexicographical constraints, and yields good propagation.

We already noted that constraint-centered heuristics and normal value orderings differ with respect to the search algorithm, in particular constraint-centered heuristics perform *2-way branching* whereas we described backtrack search with *n-way branching* (for a deeper comparison see [HM05]). The difference is basically that in 2-way branching at each node we have the decision $(X_i = a) \vee (X_i \neq a)$ whereas in n-way branching we have the decision $\bigvee_{d \in D_i} X_i = d$. Obviously, we can simulate n-way branching with 2-way branching but not necessarily the other way round. Since in principle different search procedures are deployed, we will not compare constraint-centered heuristics directly with the other value heuristics, but compare only among them.

As we want to compare value orderings we fix a variable ordering and combine it with different value orderings. We concentrate on the orderings dom, wdeg, and the Static ordering from left to right.

Our first idea was to solve the problems using dom and different value orderings, but after few instances we noticed that this approach cannot find solutions to harder instances, unless we combined dom with the static value ordering Min. So we switched to the Static variable ordering and combined this with different value heuristics. In all our experiments we set the timeout to 1 hour. We did not expect any approach to prove optimality, because even in [QW07] where they perform propagation without counting the CSP solver is not capable of this.

We list our results for the described experiment in Table 4.1. The first column gives the value ordering, the second column the number of backtracks, and the last two columns the best solution and the time (in seconds) until finding it, respectively. Dashes indicate that no solution was found at all. We include the results for Static ordering as baseline in each table. As we showed in Example 4.1 the value ordering of Static relies heavily on domain knowledge. Hence it is more fair to compare against a heuristic that selects the values randomly, because this corresponds to completely uninformed search. We denote this value ordering with Random.

A single instance of the benchmark is identified by a triple (k, nr, e), where k is the number of activities, e the number of employees and nr the instance number of the benchmark as in [QW07]. The instance is indicated below the respective values in Table 4.1.

Looking at the results (Table 4.1) we observe that the Static ordering is really superior with respect to best solution and, most notably, with respect to time to find it. This fulfills our expectations because Static was especially designed for this problem. We can also see that the counting-based heuristics usually find a better solution than the random approach (except for instances $(2, 5, 4)$ and $(2, 6, 5)$). This can be explained by the better capability of the counting-based heuristics for finding solutions. Therefore a better solution is found earlier with higher probability. A larger part of the search space is explored, since the objective function can thus prune earlier. This observation is also supported by a higher number of backtracks in most of the instances. We have no good explanation for the behavior in instance $(2, 5, 4)$, where Random found a better solution but exhibits less backtracks. We conjecture that choices in the beginning lead the search to a part of the search tree, where failures occur later or where are less solutions. The higher computational overhead for computing the counting-based heuristic is not reflected in the results, so it seems to pay off compared to Random. Among the counting-based heuristics there is hardly a difference; we conjecture them to behave almost equally in a setting when every variable is subject to only two constraints.

We shortly recapitulate the heuristics we compare in the experiments.

MaxMinSD Selects the value that has the highest minimal solution density over all constraints.

MaxSCProd Selects the value such that the product of the solution *densities* is maximal.

MaxSDProd Selects the value such that the product of the solution *counts* is maximal.

MaxMean Selects the value such that the sum of the solution densities is maximal.

MaxSD Selects the value with the highest solution density in some constraint.

MaxWMean Selects the value that maximizes the weighted sum of its solution densities; the weights are according to the constraints' weights.

MaxSD (2-way) Selects the assignment with maximal solution density in the problem.

MinSC/MaxSD (2-way) Selects the assignment with highest solution density in the constraints with least solutions.

MaxSD$_2$ (2-way) Selects the assignment that maximizes a different notion of solution density.

ordering	bt	best	t	bt	best	t
Static	1155285	26.75	1.1	266521	21.0	1.2
Static, Random	306823	29.50	13.3	301405	22.0	12.4
Static, MaxMinSD	528470	28.25	1.7	281577	21.0	1823
Static, MaxSCProd	594700	28.50	5.9	392131	21.0	1321
Static, MaxSDProd	536292	28.25	225	397968	21.0	1290
Static, MaxMean	521373	28.25	225	397940	21.0	1295
Static, MaxSD	397249	28.25	317	285308	21.0	1790
	$(1,2,4)$			$(1,8,3)$		
Static	321072	38.0	181	299257	37.00	0.9
Static, Random	90631	–	–	315912	43.25	489
Static, MaxMinSD	114781	–	–	560304	42.75	262
Static, MaxSCProd	112134	–	–	568406	42.75	251
Static, MaxSDProd	113389	–	–	560111	42.75	278
Static, MaxMean	112607	–	–	550784	42.75	286
Static, MaxSD	82728	–	–	380725	42.75	403
	$(1,4,6)$			$(1,3,6)$		
Static	909666	24.0	0.8	466053	19.00	1817
Static, Random	76778	–	–	187994	20.25	623
Static, MaxMinSD	383759	24.0	2278	204567	20.00	3417
Static, MaxSCProd	360913	24.0	1426	169018	19.00	1016
Static, MaxSDProd	381706	24.0	2293	174521	19.00	1020
Static, MaxMean	387594	24.0	2261	174730	19.00	1013
Static, MaxSD	292682	24.0	3063	173673	19.00	1021
	$(1,5,5)$			$(2,9,3)$		
Static	496431	26.50	1.3	188528	25.0	8.9
Static, Random	179918	25.50	312	167165	25.0	1612
Static, MaxMinSD	364047	26.50	178	191501	25.0	105
Static, MaxSCProd	371330	26.25	100	165198	25.0	30.4
Static, MaxSDProd	368792	26.25	145	189357	25.0	34.7
Static, MaxMean	364335	26.50	150	189357	25.0	35.9
Static, MaxSD	354681	26.00	105	188814	25.0	18.0
	$(2,5,4)$			$(2,1,5)$		
Static	276137	31.50	117	84175	26.75	2.2
Static, Random	236897	37.00	201	137422	24.75	3202
Static, MaxMinSD	320439	35.25	3586	128108	27.50	487
Static, MaxSCProd	310029	35.50	562	118862	27.50	16.5
Static, MaxSDProd	336247	35.25	1210	126192	27.50	16.0
Static, MaxMean	323337	35.00	1290	90605	27.50	8.6
Static, MaxSD	328144	34.75	1230	126267	27.50	6.1
	$(2,8,5)$			$(2,6,5)$		

Table 4.1: Results for Static variable ordering and varying value ordering.

ordering	bt	best	t	bt	best	t
MaxSD	305189	26.25	437	222876	20.50	370
MaxSD$_2$	118290	26.75	3381	215947	21.00	16.3
MinSC/MaxSD	232539	28.00	739	360876	21.25	1101
		$(1, 2, 4)$			$(1, 8, 3)$	
MaxSD	103477	–	–	444200	43.75	1595
MaxSD$_2$	117678	–	–	374930	39.25	1.4
MinSC/MaxSD	132147	38.5	3023	354046	41.50	2566
		$(1, 4, 6)$			$(1, 3, 6)$	
MaxSD	702040	27.25	279	262322	19.00	50.0
MaxSD$_2$	276632	25.00	1.0	190685	21.00	2392
MinSC/MaxSD	262804	24.50	1788	149900	19.25	3090
		$(1, 5, 5)$			$(2, 9, 3)$	
MaxSD	306472	26.75	696	178332	25.0	151
MaxSD$_2$	225141	25.50	1.4	218896	25.0	1321
MinSC/MaxSD	310494	26.50	1361	253316	25.0	816
		$(2, 5, 4)$			$(2, 1, 5)$	
MaxSD	333116	34.50	406	124390	27.50	1949
MaxSD$_2$	293309	36.25	1957	69817	–	–
MinSC/MaxSD	327350	36.00	989	230883	26.75	1252
		$(2, 8, 5)$			$(2, 6, 5)$	

Table 4.2: Results for solving the benchmark with constraint-centered heuristics

We conducted experiments under the same conditions for the constraint-centered heuristics MinSC/MaxSD, MaxSD, and MaxSD$_2$. The results for these experiments are shown in Table 4.2. Again the results are not indicating that one heuristic is superior to the others. We observe that the number of backtracks during the hour is usually smaller for the MaxSD$_2$ heuristic. This can be explained by the higher overhead of computing it compared to MinSC/MaxSD and MaxSD (we implemented it using logarithms since otherwise the quotients $sc(C, X_i, a)/|D_1| \times \ldots \times |D_k|$ become too small).

A principal problem of the benchmark is that as soon as we switch from decision problem to optimization problem, we tend to explore the complete search space, therefore the value heuristic loses on influence. Moreover, since we do optimization and do not perform counting on the objective function, the search for an optimal solution is more or less blind. One way to overcome this problem would be a counting scheme on the objective function. In fact, the goal here is to minimize the number of working hours, so we can also use a simple pseudo constraint that gives preference to the values r, l, and b.

heuristic	(1,2,4)	(1,3,6)	(1,5,5)	(1,8,3)	(1,4,6)	(2,9,3)	(2,5,4)	(2,1,5)	(2,6,5)	(2,8,5)
Static	1.03	0.9	0.78	0.7	181	0.9	1	1.1	2.2	1.7
Static, Random	1.4	0.7	–	0.7	–	1.3	1.7	1.8	2	2.2
Static, MaxMinSD	1.6	0.7	0.9	1.3	–	7	1.1	1.2	3.4	1.8
Static, MaxSCProd	5.9	0.8	1	0.7	–	1	1.1	1.1	1.6	2.5
Static, MaxSDProd	1.6	0.8	1	0.7	–	1	1.1	0.9	1.6	2.6
Static, MaxMean	1.6	0.8	1	0.7	–	1	1.1	1.1	2.2	2.5
Static, MaxSD	2.2	1.09	1.3	0.8	–	1.4	1.1	1.1	1.6	2.5
MaxSD	1.34	0.9	1.1	0.6	–	0.9	1.2	1.6	2.1	1.9
MaxSD$_2$	16	0.7	0.8	1.3	–	5	1.3	13	–	13
MinSC/MaxSD	1.4	0.7	0.8	0.5	674	1.1	1	3.3	2	1.6
wdeg, Min	0.9	0.9	0.6	0.3	15	1.1	0.6	0.4	0.9	1.9
wdeg, Random	0.75	1.28	0.38	0.35	18.7	0.68	0.53	0.43	3.2	1.4
wdeg, MaxWMean	0.7	2.5	0.5	0.5	80	1.1	0.7	0.7	0.8	1.5

Table 4.3: Comparison of time to find first solution.

Based on the observation regarding optimization, we decided to conduct experiments in order to compare the time for finding the *first* solution, hence to solve the decision problem. We included also the weighted degree heuristics for selecting a variable combined with Random, Min, and MaxWMean in these experiments. For the heuristics that rely on weighted constraints we performed probing with restarts. The parameters for this were 100 restarts with a cutoff of 40 fails in completely random search, i.e., with random variable and value selection. Already during probing we discovered that the instances are easily satisfiable, since even random search often succeeded in finding solutions. As indicated by the observation during probing we found that none of the approaches clearly outperformed the others. Moreover, there is almost no backtracking necessary to find solutions in either case. We present our results in Table 4.3. Note that the results for Random value selection are averaged over 10 runs and that we additionally omitted the symmetry breaking constraints for the weighted degree heuristics, since we suspect the symmetry breaking constraints to impede the search to go into the direction the weighted degree heuristic in combination with the value ordering suggest.

5 Conclusion and Future Work

5.1 Conclusion

Our main contribution is the presentation of a general couting scheme based on compilation to the knowledge representation language `sd-DNNF` [DM02]. Our approach exploits a linear time counting algorithm in this class [Dar01]. Our approach captures a known solution counting algorithm for `regular` [ZP07], and provides us with the possibility to count exactly solutions in `extensional` constraints and approximately in `grammar` constraints, which are new results. This collection of constraints is rather expressive, because several other constraints, such as `among` and `element`, can be formulated in terms of them. Moreover, since `BDD`s are a subclass of `sd-DNNF`, we can also use our algorithm to count solutions for theories or constraints that are represented as `BDD` formulas. However, limitations of this approach are constraints, for which the counting problem is #P-hard, for example `alldifferent`.

Our motivation was to make use of the information gained by solution counting in value ordering heuristics. We considered the problem of a good value selection from different points of view and proposed, based on this, some new heuristics. We then evaluated them by solving rostering problems. The results indicate a slightly better performance compared to random value selection in terms of finding better solutions in the optimization problems. The investigation of the decision problems did not lead to conclusions because the problems are too easy to satisfy.

5.2 Future Work

Of course, this thesis presents only an early attempt to examine counting-based value heuristics. For a more complete analysis of the idea, we have to continue work in several directions.

Although the constraints we have considered here are quite expressive, more work has to be done in the area of counting, since there are still constraints where no solution counting algorithm is known. Global cardinality and sequence constraints belong to this group, and there might exist constraints were only approximation algorithms are practicable, like for `alldifferent` [ZP07]. New approaches to counting solutions can vary from expressing the constraint in terms of constraints an algorithm is known for, over compiling constraints directly to `sd-DNNF` (be it automatically or constructive), to completely new counting schemes.

The space for other heuristics based on value orderings is not yet exhausted. We imagine that other heuristics combine the constraint-centered information better, for example by applying belief propagation techniques [HKBM07]. Also a better comparison between 2-way branching and n-way branching in presence of sophisticated variable/value ordering heuristics is necessary to understand the role of the value ordering in practice better.

There are only few works that actually evaluate counting-based value selections. On the one hand we can investigate the existing benchmarks in more depth. For example, in our setting we can take the objective function into account or make the problems artificially more difficult by adding some random constraints. We conjecture that this would break the dominance of the static variable and value ordering and possibly leads to clearer picture in comparison to random orderings. On the other hand, provided new counting algorithms, we can run benchmarks involving more types of constraints.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AFM02] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic csps application to configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.

[Arr70] Kenneth Arrow. *Social Choice and Individual Values*. New Haven: Yale University Press, December 1970.

[BC93] Christian Bessière and Marie-Odile Cordier. Arc-consistency and arc-consistency again. In *Proceedings of AAAI 1993*, pages 108–113, 1993.

[BCS01] Christian Bessière, Assef Chmeiss, and Lakhdar Sais. Neighborhood-based variable ordering heuristics for the constraint satisfaction problem. In Toby Walsh, editor, *CP*, volume 2239 of *Lecture Notes in Computer Science*, pages 565–569. Springer, 2001.

[BCvBW02] Fahiem Bacchus, Xinguang Chen, Peter van Beek, and Toby Walsh. Binary vs. non-binary constraints. *Artificial Intelligence*, 140(1/2):1–37, 2002.

[Ben06] Frédéric Benhamou, editor. *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, volume 4204 of *Lecture Notes in Computer Science*. Springer, 2006.

[Bes07] Christian Bessiere, editor. *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*. Springer, 2007.

[BG95] Fahiem Bacchus and Adam J. Grove. On the forward checking algorithm. In Ugo Montanari and Francesca Rossi, editors, *CP*, volume 976 of *Lecture Notes in Computer Science*, pages 292–308. Springer, 1995.

[BHLS04] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In de Mántaras and Saitta [dMS04], pages 146–150.

[BMFL02] Christian Bessière, Pedro Meseguer, Eugene C. Freuder, and Javier Larrosa. On forward checking for non-binary constraint satisfaction. *Artificial Intelligence*, 141(1/2):205–224, 2002.

[BPW04] J. Christopher Beck, Patrick Prosser, and Richard J. Wallace. Trying again to fail-first. In Boi Faltings, Adrian Petcu, François Fages, and Francesca Rossi, editors, *CSCLP*, volume 3419 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2004.

[BR96] Christian Bessière and Jean-Charles Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In Eugene C. Freuder, editor, *CP*, volume 1118 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 1996.

[Bré79] Daniel Brélaz. New methods to color vertices of a graph. *Commununications of the ACM*, 22(4):251–256, 1979.

[CBA05] Marco Correia, Pedro Barahona, and Francisco Azevedo. CaSPER: A programming environment for development and integration of constraint solvers. In *Proceedings of the First International Workshop on Constraint Programming Beyond Finite Integer Domains (BeyondFD'05)*, pages 59–73, 2005.

[CG98] John Horton Conway and Richard K. Guy. *The Book of Numbers*. Springer, February 1998.

[CJS08] Martin C. Cooper, Peter G. Jeavons, and András Z. Salamon. Hybrid tractable CSPs which generalize tree structure. In Malik Ghallab, Nikos Fakotakis, and Nikos Avouris, editors, *ECAI*. IOS Press, 2008.

[CvB01] Xinguang Chen and Peter van Beek. Conflict-directed backjumping revisited. *Journal of Artificial Intelligence Research (JAIR)*, 14:53–81, 2001.

[CY06] Kenil C. K. Cheng and Roland H. C. Yap. Maintaining generalized arc consistency on ad-hoc n-ary Boolean constraints. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI*, pages 78–82. IOS Press, 2006.

[Dar01] Adnan Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics*, 11(1-2):11–34, 2001.

[Dar04] Adnan Darwiche. New advances in compiling CNF to decomposable negational normal form. In de Mántaras and Saitta [dMS04].

[Dec99] Rina Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.

[DM02] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[dMS04] Ramon López de Mántaras and Lorenza Saitta, editors. *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*. IOS Press, 2004.

[DP87] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1987.

[DP89] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366, 1989.

[FD95] Daniel Frost and Rina Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'95*, pages 572–578, 1995.

[FPDN05] Alan M. Frisch, Timothy J. Peugniez, Anthony J. Doggett, and Peter Nightingale. Solving non-boolean satisfiability problems with stochastic local search: A comparison of encodings. *Journal of Automated Reasoning*, 35(1-3):143–179, 2005.

[Fro97] Daniel Frost. *Algorithms and Heuristics for Constraint Satisfaction Problems*. PhD thesis, University of California, Irvine, 1997.

[Gae06] Wulf Gaertner. *A Primer in Social Choice Theory*. OUP: Oxford, June 2006.

[Gas77] John Gaschnig. A general backtrack algorithm that eliminates most redundant tests. In *IJCAI*, page 457, 1977.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, January 1979.

[GW07] Diarmuid Grimes and Richard J. Wallace. Sampling strategies and variable selection in weighted degree heuristics. In Bessiere [Bes07], pages 831–838.

[HDT92] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2-3):291–321, 1992.

[HE80] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.

[HKBM07] Eric I. Hsu, Matthew Kitching, Fahiem Bacchus, and Sheila A. McIlraith. Using expectation maximization to find likely assignments for solving CSP's. In *AAAI*, pages 224–230. AAAI Press, 2007.

[HM05] Joey Hwang and David G. Mitchell. 2-way vs. d-way branching for csp. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 2005.

[JBKW08] Jean Christoph Jung, Pedro Barahona, George Katsirelos, and Toby Walsh. Two encodings of DNNF theories. In *ECAI'08 Workshop on Inference methods based on Graphical Structures of Knowledge*, 2008.

[KDG04] Kalev Kask, Rina Dechter, and Vibhav Gogate. Counting-based look-ahead schemes for constraint satisfaction. In Wallace [Wal04], pages 317–331.

[MF85] Alan K. Mackworth and Eugene C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1):65–74, 1985.

[Pes04] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In Wallace [Wal04], pages 482–495.

[Pes05] Gilles Pesant. Counting solutions of CSPs: A structural approach. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI*, pages 260–265. Professional Book Center, 2005.

[QR07] Claude-Guy Quimper and Louis-Martin Rousseau. A large neighborhood search approach to the multi-activity shift-scheduling problem. Technical report, CIRRELT, 2007.

[QW06] Claude-Guy Quimper and Toby Walsh. Global grammar constraints. In Benhamou [Ben06], pages 751–755.

[QW07] Claude-Guy Quimper and Toby Walsh. Decomposing global grammar constraints. In Bessiere [Bes07], pages 590–604.

[Ref04] Philippe Refalo. Impact-based search strategies for constraint programming. In Wallace [Wal04], pages 557–571.

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

[Sel06] Meinolf Sellmann. The theory of grammar constraints. In Benhamou [Ben06], pages 530–544.

[SG98] Barbara M. Smith and Stuart A. Grant. Trying harder to fail first. In *Proceedings of European Conference on Artificial Intelligence 1998*, pages 249–253, 1998.

[SLM92] Bart Selman, Hector J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.

[Val79] Leslie G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8:189–201, 1979.

[vHM05] Pascal van Hentenryck and Laurent Michel. *Constraint-Based Local Search.* MIT Press, August 2005.

[Wal04] Mark Wallace, editor. *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*. Springer, 2004.

[ZP07] Alessandro Zanarini and Gilles Pesant. Solution counting algorithms for constraint-centered search heuristics. In Bessiere [Bes07], pages 743–758.