UCLouvain

École polytechnique de Louvain

Development of Black-box search and heuristics for OscaR solver for the XCSP3 competition

Author: **Yannick GOULLEVEN**
Supervisor: **Pierre SCHAUS**
Readers: **Pierre SCHAUS, Charles THOMAS, Xavier GILLARD**
Academic year 2018–2019
Master [120] in Computer Science

Abstract

Designing an efficient and robust black-box search is one of the major concerns in Constraint Programming. In the past, various search heuristics and procedures have been proposed in the literature to tackle this problem. However, up to this day there is no unique best search strategy that is able to solve all problems optimally. For this reason, competitions such as the XCSP3 competition are organized to asses the performance of constraint solvers. This thesis compares existing search strategies as well as combinations of them in order to improve the black-box search used by the OscaR solver on constraint optimization problems. Experiments show that the newly adopted search strategy is able to improve the overall performance of the OscaR solver on the fast COP tracks of the 2018 XCSP3 competition.

Contents

Chapter 1

Introduction

Constraint programming is a powerful paradigm for solving constrained combinatorial problems. Real world problems that rely on constraint programming to be solved are generally those that can be naturally formulated in terms of requirements, general properties, and for which domain specific methods lead to very complex formalizations [1]. As an example, fields like molecular biology, electrical engineering, operations research and numerical analysis tend to rely on constraint programming to solve domain specific problems.

Systems that are designed to solve constrained problems are referred to as constraint solvers. In contrast to domain specific methods, constraint solvers use specific search methods that do not incorporate internal knowledge about the problem itself. Efficient search methods are able to greatly reduce the search space that has to be explored, which in return decreasing the time needed to find a solution to the problem. So called black-box solvers go even further, in the sense that they do not require any human intervention to perform well on a large variety of problems [9].

The popularity of constraint programming in many fields is due to its declarative principle under which a user only needs to declare a model of the problem to be solved, but not the means for solving the model. The declarative principle artificially splits the usage of constraint programming into a two step procedure. The first step requires the user to transform the initial problem into a valid model. Once the problem has correctly been modeled, it needs to be expressed in a language that the solver understands. The second step consists in starting the search of the solver in order to find a solution to the previously defined model. For black-box solvers, this second step will always be characterised by the same search procedure regardless of the initial problem.

1.1 XCSP3 Competition

The XCSP3 Competition (`http://xcsp.org/competition`) is an annual competition organized to assess the performance of constraint solvers. In the category of optimization problems, solvers are ranked according to two different point of views. The complete solver point of view ranks the solvers according to the number of times they find the optimal solution. The incomplete solver point of view ranks the solvers according to the number of times they find the best solution.

1.2 Contribution

The aim of this thesis is to improve the search strategy used by the OscaR solver for the upcoming XCSP3 competition. Numerous heuristics and search procedures have already been proposed in the literature. The main focus of this thesis lies in comparing and assessing performance of existing approaches.

1.3 Outline

The rest of this thesis is structured as follows:

Chapter 2: Formal background introduces the theory behind modeling and solving problems in constraint programming. Furthermore, the chapter introduces terminology that is used throughout the thesis.

Chapter 3: Existing heuristics summarizes the existing heuristics for variable and value selection that have been implemented in the OscaR solver.

Chapter 4: Parsing XCSP3 instances describes the XCSP3 format which is used during the competition. In addition, the chapter discusses the problem of extracting decision variables during the parsing process.

Chapter 5: Development of a new search strategy describes the newly developed search strategy that is used by the OscaR solver.

Chapter 6: Conclusion summarises the work of this thesis.

Chapter 2

Formal background

The purpose of this chapter is to introduce the background knowledge needed for this thesis. The first step, which is commonly referred to as the modeling (or modelling) step is described in section 2.1. It consists in transforming a real world problem into a more general yet equivalent representation. Section 2.2 describes by what means the new formalisation of the problem can be exploited to simplify its solving process. Finally, section 2.3 describes how a constraint solver is able to find a solution to the problem through a process called search.

2.1 Modeling

One of the strengths of constraint programming is that it can be used on a vast amount of problems belonging to different domains of application. The aim of the modeling phase is to transform the initial problem into a general representation of it. Constraint satisfaction or constraint optimization problems (CSOPs) are examples of valid representations. Informally, a CSOP consists of a set of variables subject to some constraints. We will first introduce some terminology about the components of a CSOP before giving a more formal definition.

2.1.1 Variables and domains

A variable, denoted by a lowercase letter (e.g. x) follows its mathematical definition. Put simple it is a placeholder, a quantity that can change over time. A variable has to be associated with a domain $D(x) = \{v_1, \dots v_k\}$ which represents the set of possible values the variable can be assigned to. Once the variable is assigned to a value $v \in D(x)$ the remaining values are removed from its domain and the size of the domain $|D(x)|$ reduces to 1. Formally, a variable x is bound to a value if $|D(x)| = 1$ and unbound if $|D(x)| > 1$.

Example 2.1.1. Consider the variable x with domain $D(x) = \{1, 2, 3\}$. At this stage, the variable is unbound and the only possible assignments are: $< x = 1 >$, $< x = 2 >$ and $< x = 3 >$.

2.1.2 Constraints

The purpose of a constraint is to enforce a relation on the domains of a set of variables.

Definition 2.1.1. Consider a finite sequence of variables $X = \{x_1, \ldots, x_n\}$ with respective domains $D(x_1), \ldots, D(x_n)$. A constraint c on X enforces a subset of $D(x_1) \times \cdots \times D(x_n)$.

A constraint c is satisfiable if there is a value in the domain of each variable such that the relation is not violated and unsatisfiable otherwise. From a variable point of view, an assignment $< x = v >$ that satisfies all the constraints is a consistent assignment, while an assignment $< x = v >$ that violates one or more constraints is called inconsistent.

There are many different types of constraints that will restrain the domains of a set of variables in different ways. The following example illustrates the usage of the *allDifferent* constraint [49] on a set of variables. Put simple, the *allDifferent* constraint forbids that more than one variable in the set is assigned to the same value.

Example 2.1.2. Consider the set of variables $X = \{x_1, x_2\, x_3\}$ with domains $D(x_1) = D(x_2) = D(x_3) = \{1, 2, 3\}$. The constraint *allDifferent*(x_1, x_2, x_3) or equivalently *allDifferent*(X) expresses the fact that each variable has to be assigned to a different value. After declaring the constraint, the subset of the Cartesian product of the domains reduces to: [{1, 2, 3}, {1, 3, 2}, {2, 1, 3}, {2, 3, 1}, {3, 2, 1}, {3, 1, 2}]. Equivalently, we can say that there are 6 possible ways to assign the three variables such that the constraint is satisfied.

2.1.3 Constraint Satisfaction Problem

A constraint satisfaction problem (CSP) is a complete representation of a decision problem. Decision problems belong to a family of problems where the goal is to prove satisfiability. In other terms,the information of interest is whether there exists a solution to the problem or not.

Definition 2.1.2. A CSP, $P = (X, C)$ is defined by a set of variables $X = \{x_1, ..., x_n\}$ and a set of constraints $C = \{c_1, \ldots, c_m\}$ involving one or more variables from X. Each variable x_i for i in $1 \ldots n$ has a domain $D(x_i)$ which represents the set of values that can be assigned to x_i. A solution $s = (v_1, \ldots, v_n)$ to a CSP is a set

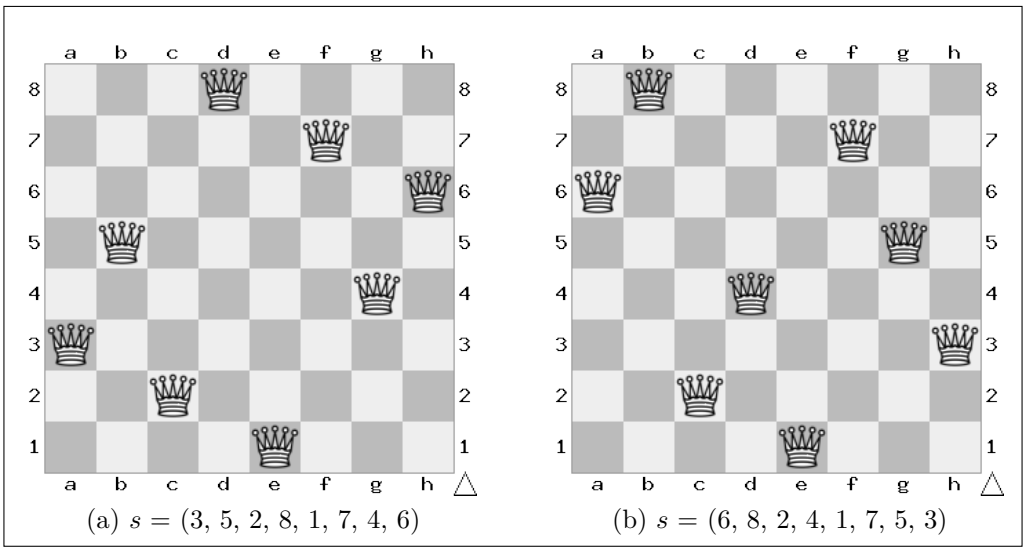
of values, one for each variable, such that the assignments $<x_1 = v_1, \ldots, x_n = v_n>$ satisfies all the constraints.

Finding one solution is a sufficient condition for proving that the CSP is satisfiable. The means for proving that a CSP is unsatisfiable will be discussed in section 2.3. To illustrate how a problem is transformed into a CSP, the *n queens* problem [41] is used as an example

Example 2.1.3. Put simple, the *n queens* problem consists in placing n queens on an n by n chess board, in such a way that they do not attack each other. The corresponding CSP uses a set of n variables: $X = \{x_1, x_2, \ldots, x_n\}$ with domains $D(x_1) = D(x_2) = \cdots = D(x_n) = [1..n]$. Each variable x_i with $i \in [1, n]$ represents the position of the queen placed in the i^{th} column of the chess board. The appropriate constraints are defined by:

- $x_i \neq x_j$
- $x_i - x_j \neq i - j$
- $x_i - x_j \neq j - i$

for $i \in [1, n-1]$ and $j \in [i+1, n]$. The first constraint verifies that no two queens are put in the same row. The two remaining constraints disallow putting two queens in the same diagonal.



(a) $s = (3, 5, 2, 8, 1, 7, 4, 6)$ (b) $s = (6, 8, 2, 4, 1, 7, 5, 3)$

Figure 2.1: Two different solutions to the 8-queens problem

Figure 2.1 illustrates two possible solutions to the *n queens* problem when n is fixed to 8.

2.1.4 Constraint Optimization Problem

A constraint optimization problem (COP) is a complete representation of an optimization problem. In contrast to a decision problem, an optimization problem defines an objective. Solutions are no longer equivalent, but their quality can be measured with respect to the objective function. The ultimate goal in solving a COP is to find the solution that optimizes the objective function.

Definition 2.1.3. A COP, $O = (P, F)$ consists of a CSP P and an objective function F defined over a subset of variables $Z \subseteq X$ which has to be optimized.

Without loss of generality, the terminology that follows assumes that O is a minimization problem.

The objective value $F(s)$ of a solution s is given by $F(< z_1 = s_1, \ldots, z_k = s_k >)$ for $z_1, \ldots, z_k \in Z$. A solution s_1 is better than a solution s_2, if and only if $F(s_1) < F(s_2)$. The set $S = \{s_1, \ldots, s_l\}$ represents the set of all the solutions to the COP.

Definition 2.1.4 An optimal solution s_o to a COP is a solution such that $\forall s \in S : s_o \leq s$.

To illustrate the transformation of an optimization problem into a COP, the *knapsack* problem is used as an example.

Example 2.1.4 Given a set of items of length n, each with a weight and a value, the *knapsack* problem consists in selecting a subset of the items, such that the total weight is less than or equal to a given limit and the total value is as large a possible. In this representation of the *knapsack* problem, a set of n variables: $X = \{x_1, \ldots, x_n\}$ with respective domains $D(x_1) = \cdots = D(x_n) = \{0, 1\}$ is used to represent the decision of selecting an item. Selecting item i with $i \in [1,\ n]$ corresponds to the assignment $< x_i = 1 >$. On the other hand, x_i is assigned to 0 in case we do not select item i. Two more sets of variables, $V = \{v_1, \ldots v_n\}$ and $W = \{w_1, \ldots, w_n\}$ are needed to represent the value and the weight of each item. Notice that each variable $v_i \in V$ and $w_i \in W$ is already bound to a value that is specific to the initial problem. To make sure that the total weight is never greater than a given limit L, the following constraint needs to be added:

$$\sum_{i=1}^{n} w_i \times x_i \leq L$$

Finally, in order to express the fact that the total value has to be maximized, the objective function needs to be defined:

$$\text{maximize}\Big(\sum_{i=1}^{n} v_i \times x_i\Big)$$

2.2 Constraint propagation and filtering

In definition 2.1.2 we defined a valid solution to a CSP as an assignment of each variable to a value such that all the constraints are satisfied. Constraint propagation is a mechanism used in constraint solvers to remove values from the domains of variables that would lead to inconsistencies once assigned. Each constraint has to define its own filtering algorithm responsible for keeping the domains consistent. The filtering algorithm is called each time the domain of a variable that is part of the constraint changes. However, a detailed description of filtering algorithms is out of the scope of this thesis.

Example 2.2.1 Consider three variables x_1, x_2 and x_3 with domains $D(x_1) = \{1, 2, 3\}$, $D(x_2) = \{3, 4\}$ and $D(x_3) = \{1, 7\}$ as well as the constraints:

- $c_1 : x_3 < x_2$
- $c_2 : x_1 \neq x_3$

Without constraint propagation, the assignment $<x_1 = 1>$ does not have an impact on the domains of other variables. When the solver checks for inconsistencies, all the constraints are still satisfied:

- $c_1 : x_3 < x_2 \rightarrow \{1, 7\} < \{3, 4\}$
- $c_2 : x_1 \neq x_3 \rightarrow \{1\} \neq \{1, 7\}$

The same reasoning applies when variable x_2 is assigned to the value 3:

- $c_1 : x_3 < x_2 \rightarrow \{1, 7\} < \{3\}$
- $c_2 : x_1 \neq x_3 \rightarrow \{1\} \neq \{1, 7\}$

Only when trying to assign x_3, the solver realizes that x_3 has no consistent values in its domain. The assignment $<x_3 = 1>$ leads to an inconsistency because of constraint c_2 and the assignment $<x_3 = 7>$ leads to an inconsistency because of constraint c_1.

To illustrate the same example with constraint propagation, we assume the existence of two different filtering algorithms, one for each constraint. The pseudo code for both algorithms is given in Algorithm 1 and Algorithm 2 respectively.

Algorithm 1 Filtering for $x_1 < x_2$

```
1: procedure FILTER(x_1, x_2)
2:     for ∀v ∈ D(x_1) do
3:         if v ≥ maximal value ∈ D(x_2) then
4:             remove v from D(x_1)
5:         end if
6:     end for
7: end procedure
```

Algorithm 2 Filtering for $x_1 \neq x_2$

```
1: procedure FILTER(x_1, x_2)
2:     if x_1 is bound to some value v then
3:         remove v from x_2
4:     end if
5:     if x_2 is bound to some value v then
6:         remove v from x_1
7:     end if
8: end procedure
```

Again, the solver starts by assigning variable x_1 to the value 1. This time the filtering algorithm of the *not equal* constraint is called and removes the inconsistent value 1 from the domain of x_3. Since the variable x_3 is also involved in constraint c_1, the filtering algorithm of the *less than* constraint is executed too. The filtering algorithm removes the value 7 from the domain of x_3 which is the last value. Since x_3 can no longer be assigned to any value, the solver can already determine that the assignment $<x_1 = 1>$ is inconsistent.

To conclude, constraint propagation is a mechanism that is executed whenever a value is removed from the domain of a variable. Its main purpose is to further prune the domains of variables involved in the same constraint by removing inconsistent values. As shown in example 2.2.1, constraint propagation helps the solver to identify inconsistent states as soon as possible. The next section will give additional details on how constraint propagation is used during the searching process of the solver.

2.3 Search

There are three main algorithmic techniques for solving CSOPs: backtracking search, local search, and dynamic programming [42]. For the purpose of this theses, a description of backtracking search is sufficient.

The typical way of finding a solution with backtracking search is to perform a depth-first traversal of a search tree. At the beginning of the search, a root node is created which corresponds to the the initial CSOP. At each step of the search, an unbound variable is selected and the node is extended where the branches out of the node represent alternative choices that may have to be examined in order to find a solution. The method for extending a node in the search tree is called a branching strategy. The default choice in this thesis is binary branching: If x is the variable selected at some node, then binary branching will generate two branches:

$<x = v>$ and $<x \neq v>$ for some value v in the domain of x. Roughly speaking, the solver will always expand the current node by creating two new child nodes. In the left child node the variable is assigned to some value, while in the right child node this same value is removed from the domain of the variable. As a result, one or more values are removed from the domain of a variable after each decision. Furthermore, with help of constraint propagation, domains of other variables might also get reduced. In some cases, the solver has to remove all the values from the domain of a variable because none of them would lead to a consistent assignment anymore (see example 2.2.1). Since the variable can no longer be assigned in that part of the tree, the solver needs to backtrack to the previous consistent node and try out the alternative branch.

The solver successfully finds a solution when it reaches a leaf node. In case of a CSP, finding one possible solution is sufficeint to prove consistency and the solver can terminate its search. In some cases however, the user is interested in finding all the solutions meaning that the solver continues its search until the entire search tree is explored. If no solution is found after the search tree has been explored, the solver can prove that no solution exists, i.e. that the CSP is inconsistent.

For COPs, the solver computes the objective value upon finding a solution. Since there might be solutions of better quality, i.e. with a better objective value in other parts of the search tree, the solver needs to continue its search. In order to find strictly better solutions during the search, the classical depth-first traversal of the tree is replaced by a depth-first branch and bound search [2]. The idea behind branch and bound is to compare the objective value of the current best solution with the objective value one can obtain at a given node by relaxing the constraints. If the optimistic approximation of the objective value at the given node becomes worse than the current best solution, the search strategy can prune this part of the search tree knowing that it did not contain any better solutions. Similarly to the inconsistency prove of CSPs, the solver needs to explore the entire search tree to prove that a solution is optimal in a COP.

Solvers that give no guarantee on exploring the entire search tree are called incomplete solvers. They are able to find solutions to both CSPs and COPs but are not designed for proving inconsistency or optimality. On the other hand, complete solvers are designed to explore the entire search tree, meaning that they are able to prove inconsistency and optimality.

Some solvers might be more efficient than others regarding a specific problem. Efficiency can be measured by many aspects, including the running time of the solver, the number of backtracks needed, the amount of memory used, the quality of the last solution, etc. There are many factors that may influence the efficiency of a constraint solver. For the rest of this thesis, we will mainly focus on how the

search strategy impacts the efficiency of the solver.

Definition 2.3.1. The search strategy of a constraint solver determines how the search tree is build during the solving process. At each node, the search strategy needs to make a decision involving a variable, a value and a branching strategy.

The order in which the search strategy selects the variables and values can have a great impact on the efficiency of the solver [20]. Techniques used to establish an order in which variables and values are selected will be discussed next.

2.3.1 Variable ordering

At each node of the search tree, the search strategy needs to decide which variable to assign next. We give a definition of variable ordering adapted from [42].

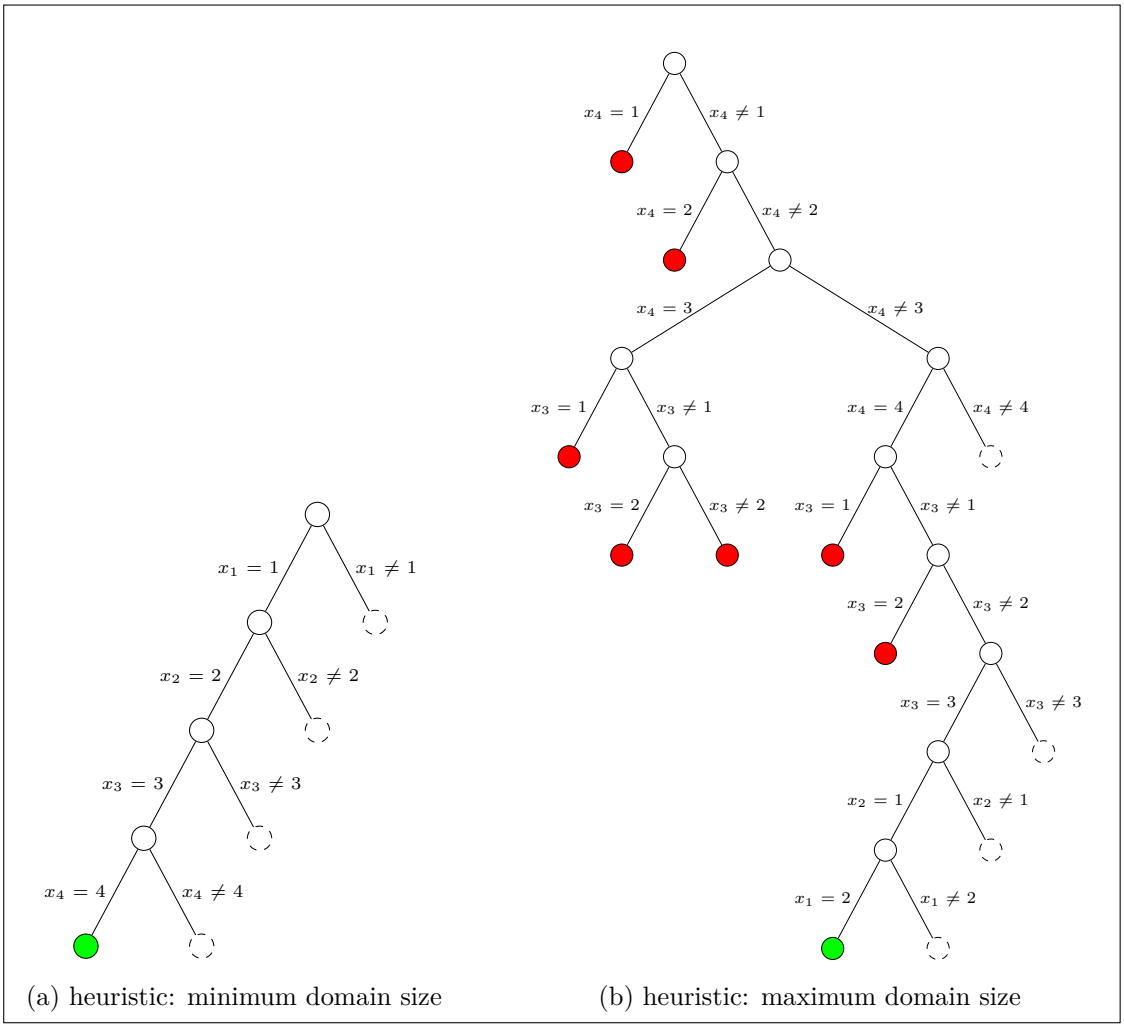
Definition 2.3.2 When solving a CSP using backtracking search, a sequence of decisions must be made as to which variable to branch on or instantiate next. The resulting sequence of decisions is referred to as the variable ordering.

The search strategy relies on a function that computes a score for each variable according to some criterion and selects the variable that optimizes the score to be assigned next. A function with this behavior is called a variable ordering heuristic.

Definition 2.3.3 A variable ordering heuristic $varH(x_i)$ maintains a score for each variable $x_i \in X$. The score of each variable is computed/updated according to some criterion during search.

In the following example, we compare the search trees build by two different variable ordering heuristics. The goal is to highlight the fact that some criteria are better suited for variable ordering heuristics than others.

Example 2.3.1. Consider a CSP defined over the variables x_1, x_2, x_3, x_4 with domains $D(x_1) = D(x_2) = \{1, 2\}$, $D(x_3) = \{1, 2, 3\}$ and $D(x_4) = \{1, 2, 3, 4\}$. There is only one constraint, $c_1 = \textit{allDifferent}(x_1, x_2, x_3, x_4)$. The *allDifferent* constraint in this example uses the filtering algorithm from Algorithm 2 for each pair (x_i, x_j) for $i \in$ [1,3] and $j \in [i + 1,\ 4]$. Figure 2.2 illustrates two search trees that are created by two different variable ordering heuristics. The variable ordering heuristic used to build the tree on the left selects the variable having the smallest domain first. On the other hand, the variable heuristic used to build the tree on the right selects the variable with the largest domain first. Both search strategies perform a binary depth-first traversal of the tree and select the smallest value in the domain of a variable first.



(a) heuristic: minimum domain size

(b) heuristic: maximum domain size

Figure 2.2: Example on how variable ordering heuristics can influence the size of the search tree

In this figure circles represent nodes of the search tree, the root node being at the top. Each edge represents a decision of the search strategy. Nodes colored in green are leaf nodes that contain a solution, while red nodes indicate inconsistencies. When a red node is encountered, the solver needs to backtrack and try out the alternative decision. Finally, dashed circles represent nodes of alternative decisions that are not examined during search.

The variable heuristic used for building search tree (a) relies on the fail-first principle [22], which states that “To succeed, try first where you are most likely to fail". It suggests to assign variables that will most likely lead to inconsistencies early in the search. In our example, variables x_1 and x_2 have only two values in their respective

domains. The fact that they are assigned at the beginning of the search leads to a solution without encountering any inconsistencies.

In comparison, search tree (b) is much bigger and contains a lot of red nodes. During the search, the solver has to try out every possible value in the domain of x_4 to find out that the assignment $< x_4 = 4 >$ is the only assignment that leads to a solution. This is a perfect illustration of the fail-first principle because variables x_2 and x_1 repeatedly lead to inconsistencies and should therefore be assigned first.

2.3.2 Value ordering

After having selected the variable, the search strategy needs to decide which value it will assign to the variable.

Definition 2.3.4. The value ordering corresponds to the sequence of values that are selected for each variable during the backtracking search.

As for variable ordering, value ordering also relies on functions that compute a score for each value in the domain of a variable.

Definition 2.3.5. A value ordering heuristic $valH(x_i, v_k)$ maintains a score for each variable $x_i \in X$ and value $v_k \in D(x_i)$. The score for each variable, value pair is computed/updated according to some criterion during the search.

Example 2.3.2. Consider the COP with variables x_1, x_2, x_3, x_4 and domains $D(x_1) = D(x_2) = D(x_3) = D(x_4) = \{1, 2, 3, 4\}$. The constraints and the objective are:

- $c_1 : x_3 < x_2$
- $c_2 : x_1 \neq x_2 \neq x_3 \neq x_4$
- objective : maximize$(x_1 + x_4)$

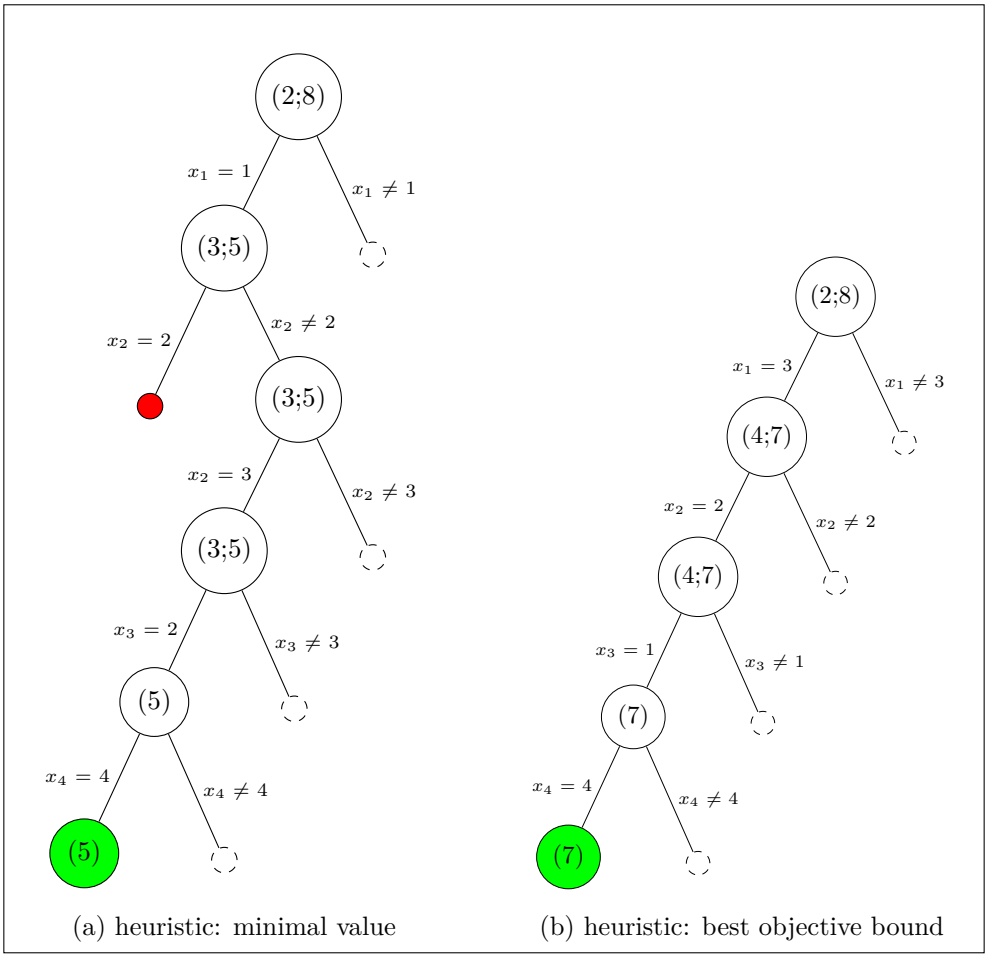
Again, we use the filtering algorithms from section 2.2 The conventions used in Figure 2.3 are the same as for Figure 2.2, except that we additionally include the bounds of the objective function in each node. The search strategy uses binary branching and selects variables in lexicographical order for both search trees. However, this time the value ordering heuristics are different. The score returned by value heuristic in (a) is simply the value itself and the search strategy will minimize this score, i.e. select the lowest value in the domain of a variable. The value heuristic in (b) is known as the bound impact value selector and an adaptation of the pseudocode from [14] is shown in Algorithm 3.

Algorithm 3 BIVS used in Figure 2.2 (b)

```
1: function BIVS(x_i, v_j)
2:     push a temporary node on top of the DFS stack
3:     assign x_i to v_j
4:     propagate the assignment
5:     if the assignment is consistent then
6:         if the objective function has to be minimized then
7:             score ← the lower bound of the objective variable
8:         else
9:             score ← the upper bound of the objective variable
10:        end if
11:    else
12:        if the objective function has to be minimized then
13:            score ← the largest integer value
14:        else
15:            score ← the lowest integer value
16:        end if
17:    end if
18:    remove the temporary node from the DFS stack
19:    return score
20: end function
```

In order to compute the score for a given variable $x_i \in X$ and value $v_j \in D(x_i)$, the heuristic pushes a new temporary node on top of the existing stack and makes the assignment $x_i = v_j$. After the decision is propagated, the heuristic checks if the assignment leads to a consistent state. In a consistent state, the lower bound (resp. upper bound) of the objective variable is returned for minimization (resp. maximization) problems. In case an inconsistency is encountered, the heuristic returns the biggest (resp. smallest) integer number for minimization (res. maximization) problems. Finally, for some fixed variable, the search strategy will select the value minimizing (resp. maximizing) the score returned by the heuristic for minimization (resp. maximization) problems.

If we compare both search trees from Figure 2.3, we observe that the tree on the left side (a) leads to a backtrack and that the solution found in the first leaf node has a worse objective value. The tree on the right side (b) does not lead to backtracks, because inconsistent values are given the worst score and will only be selected if all values in the domain lead to inconsistencies. Furthermore, the values selected are the ones that will optimize the objective function at each node. Therefore, the first solution found is of better quality.



(a) heuristic: minimal value

(b) heuristic: best objective bound

Figure 2.3: Example on how variable ordering heuristics can influence the size of the search tree

Chapter 3

Existing heuristics

Many different heuristics for improving the search strategy in a constraint solver have already been proposed in the literature. The purpose of this chapter is to familiarize the reader with the heuristics that are used in the OscaR solver.

3.1 Variable ordering heuristics

3.1.1 Smallest Domain First (SDF)

Haralick & Elliott [22] were the first to propose the fail-first principle as a variable ordering heuristic. In the literature, the heuristic is generally referred to as the Smallest Domain First (SDF) heuristic (e.g. [4] [5]). The intuition behind this heuristic is simple: when the search strategy has to select a variable, it should select the one with the smallest domain size. The reason why failing early in the search is preferable according to Haralick & Elliott is that it reduces the average path length in the search tree.

In example 2.3.1 we already showed that the SDF variable heuristic leads to a smaller search tree (Figure 2.2 (a)) and that the path length to the solution is minimal.

However, SDF does not always lead to a minimal search effort. Back, Prosser & Wallace concluded their work [5] that the fail-first principle ironically sometimes fails. They showed that a reduction in branch length is sometimes compensated by an increase in the number of branches per node, which may increase search effort. In addition, SDF performs poorly on problems where most variables have the same initial domain size because in its most basic form ties are broken in lexicographical order. For this reason, SDF is generally adapted by including another criterion in the heuristic or by using a more sophisticated tie-breaker.

3.1.2 Domain over weighted degree (dom/wdeg)

The dom/wdeg variable heuristic [7] is an extension of the dom/deg heuristic from [6]. Dom/deg measures the ratio between the domain size of a variable and its degree, i.e. the number of constraints the variable is involved in. At each decision, the search strategy selects the unbound variable with the lowest dom/deg score. Dom/deg improves the tie-breaking capabilities of SDF by including the degree of each variable in its computation: variables having the same domain size do not always have the same degree. Furthermore, it takes into account the constraints of the problem which is an important consideration in constraint programming.

Dom/wdeg additionally associates a weight factor w to each constraint. Whenever the solver needs to backtrack, the weight of the constraint responsible for the inconsistency is increased. Similar as for dom/deg, the search strategy will select the unbound variable minimizing the dom/wdeg score for assignment. Algorithm 4 shows how the score of the dom/wdeg heuristic is computed during the search.

Algorithm 4 Dom/Wdeg variable heuristic

```
1: for x_i in X do
2:     wdeg[x_i] = the constraint degree of x_i
3: end for
4: procedure onFailure        ▷ called whenever an inconsistency encountered
5:     c ← the constraint responsible for failure
6:     for x_i involved in c do
7:         wdeg[x_i] ← wdeg[x_i] + 1
8:     end for
9: end procedure
10: function wdeg(x_i)
11:     return |D(x_i)|/wdeg[x_i]
12: end function
```

3.1.3 Activity Based Search (ABS)

The main idea behind ABS which was introduced in [29] is to measure for each variable $x_i \in X$, how often its domain is reduced during constraint propagation. Each variable is attributed a score called the activity of the variable. At the beginning of the search, the activity is set to 0 for each variable because no propagation took place. After each decision, the activities of the variables will be

updated according to the following two rules:

$$\forall x_i \in X \text{ where } |D(x_i)| > 1 : A(x_i) = A(x_i) \times \gamma$$
$$\forall x_i \in X' : A(x_i) = A(x_i) + 1$$

where X' is the subset of variables that experienced a reduction in their domain after a decision was propagated and $\gamma \in [0, 1]$ the decay parameter used to forget the oldest statistics progressively.

The search strategy selects the unbound variable x_i maximizing the ratio $A(x_i)/|D(x_i)$. If the search strategy only considers the activity of the variable but not its domain size, it would mainly select variables with large domains because they are filtered more often. This is generally a bad idea, because it violates the fail-first principle.

The activities of the variables are only updated once propagation took place during search. Without initializing the activities before actually starting the search, the first few variables are selected blindly. The decisions that are made at the beginning of the search are very important, since it might take a lot of time before the solver can backtrack to those decisions. For this reason, the activities need to be initialized before the search by means of probing.

Definition 3.1.1 Probing: For some number of iterations, assign randomly selected variables to randomly selected values until a dead end is encountered. After each assignment, update the scores of the corresponding variable heuristic.

Pseudo code for initializing ABS activities by probing is given in Algorithm 5.

Algorithm 5 Probing for initializing activities

```
1: procedure PROBING
2:     for <some number of iterations> do
3:         i ← 0
4:         push a temporary node on the DFS stack
5:         while i < |X| ∧ no inconsistencies encountered) do
6:             x ← a random unbound variable
7:             v ← a value in D(x)
8:             Assign v to x
9:             propagate the decision
10:            Update the activities
11:            i ← i + 1
12:        end while
13:        remove the temporary node from the DFS stack
14:    end for
15: end procedure
```

The goal of the probing phase is to provide a realistic estimate of the activity for each variable. Therefore, the number of probing rounds is generally determined by a statistical test. The activities computed during each probing round are stored in a data structure that keeps track of the empirical distribution of the activity of each variable x_i including the mean $\hat{\mu}_A(x_i)$ and the standard deviation $\hat{\sigma}_A(x_i)$. Since the probes are independent and identically distributed, the distribution can be approximated by a normal distribution. The probing phase should be ended when the confidence interval of the student distribution is sufficiently small. [29] suggests a 95% confidence interval:

$$\left[\hat{\mu}_A(x_i) - t_{0.05,n-1} \times \frac{\hat{\sigma}_A(x_i)}{\sqrt{n}}, \hat{\mu}_A(x_i) + t_{0.05,n-1} \times \frac{\hat{\sigma}_A(x_i)}{\sqrt{n}}\right]$$

3.1.4 Objective Bound Selector (OBS)

The variable ordering heuristics described so far do not take into account the objective function of a COP. However, discovering good quality solutions early in the search might drastically reduce the search space due to branch and bound. The OBS variable ordering heuristic proposed in [33] relies on the bound modifications of the objective variable during search. At each node s, the Δ_O function computes the objective bounds modification of the objective variable o as follows:

$$\Delta_O(s) = a \times \underline{\Delta_O} + b \times \overline{\Delta_O}$$

where $\underline{\Delta_O}$ (resp. $\overline{\Delta_O}$) represent the upper (resp. lower) bounds difference between its value before and after propagation. The choice of parameters a and b define the function behavior. For instance in minimization problems, using $a = -1$ penalizes lower bound modifications while $b = 1$ rewards upper bound modifications

The final variable ordering heuristic keeps track of the weighted sum $\widetilde{\Delta_O}(x_i)$ of the Δ_O function values for each variable $x_i \in X$. The weighted sum is updated as follows:

$$\widetilde{\Delta_O(x_i)} = \frac{\widetilde{\Delta_O(x_i)} \times (1-\gamma) + \gamma \times \Delta_O(x_i)}{\gamma}$$

where γ represents the degree of weighting decrease of the exponential moving average i.e. a constant smoothing factor between 0 and 1. A higher value for α discounts older observations faster. The search strategy will select the unbound variable having the highest $\widetilde{\Delta_O}$ score.

Notice that the Δ_O values for each variable are set to 0 at the beginning of the search. As for ABS, it is best to initialize the Δ_O scores before starting the search by probing.

3.1.5 Conflict Ordering Search (COS)

In contrast to the previously defined variable heuristics, COS [18] is not a stand-alone variable ordering heuristic but a generic scheme to guide backtracking search. An alternative variable ordering heuristic is used whenever no unbound variable can be selected by COS. The criterion for selecting variables in COS is similar to Last Conflicts (LC) [25]. In LC, whenever a decision leads to an inconsistency, the responsible variable is given priority over the choice proposed by the underlying variable heuristic. LC "replaces" the underlying variable heuristic as long as the variable that led to a failure is not assigned. Once the variable is successfully assigned, i.e. it did not lead to another inconsistency, the underlying variable heuristic will be used again. Algorithm 6 describes how COS can be used on top of a variable heuristic.

Algorithm 6 Conflict Ordering Search

```
1:  procedure COS(varH)
2:      if an inconsistency is encountered then
3:          if lastVar ≠ null then
4:              timeStamp[lastVar] ← nDesision
5:          end if
6:      end if
7:      conflictSet ← {x_i ∈ X: timeStamps[x_i] > 0 ∧ D(x_i) > 1 }
8:      if conflictSet is empty then
9:          lastVar ← varH()
10:     else
11:         lastVar ← argmax_{x_i ∈ conflictSet}(timeStamps[x_i])
12:     end if
13:     nextVal ← valH(lastVar)
14:     decision ← < lastVar = nextVal, lastVar ≠ nextVal >
15:     nDecision ← nDecision +1
16:     return decision
17: end procedure
```

The main difference between COS and LC is that conflicting variables are not forgotten. Each variable $x_i \in X$ is attributed a timestamp which is initialized to 0 at the beginning of the search. In case a decision leads to an inconsistency, the responsible variable is attributed a timestamp equal to the number of decisions since the start of the search. When the search strategy has to select a new variable for assignment, COS will select the unbound variable with the greatest timestamp, i.e. involved in the latest decision. Only when all unbound variables are associated a timestamp value of 0, the underlying heuristic is used.

Notice that no two variables can be attributed the same timestamp since a decision can only involve one variable. For this reason COS is guaranteed to never break ties between variables. However, the same reasoning is not true for the underlying variable heuristic.

3.2 Value ordering heuristics

3.2.1 Least Constraining Value (LCV)

LCV ordering heuristics [44] consider that a good value ordering heuristic should assign least constraining values. A least constraining value is defined as a value that is expected to participate in many solutions. The assignment of a least constraining value to a variable should maximize the number of remaining values in the domains of the remaining unbound variables after the propagation of the decision.

Algorithm 7 Least Constraining Value Selector

```
1:  function LCV(x_i, v_k)
2:      nValuesBefore ← 0
3:      nValuesAfter ← 0
4:      for x_j ∈ X ∧ x_j ≠ x_i do
5:          if |D(x_j)| > 1 then
6:              nValuesBefore ← nValuesBefore + |D(x_j)|
7:          end if
8:      end for
9:      add a temporary node to the DFS stack
10:     assign x_i to v_k
11:     propagate the decision
12:     for x_j ∈ X ∧ x_j ≠ x_i do
13:         if |D(x_j)| > 1 then
14:             nValuesAfter ← nValuesAfer + |D(x_j)|
15:         end if
16:     end for
17:     remove the temporary node from the DFS stack
18:     return nValuesBefore - nValuesAfter
19: end function
```

Algorithm 7 describes the LCV selector. For a given variable $x_i \in X$ and value $v_k \in D(x_i)$, the LCV function will compute the sum of the domain sizes of the remaining unbound variables. It will then assign the variable to the value and propagate the decision. After propagation, LCV will recompute the sum of the

domain sizes and return their difference. The search strategy will select the value minimizing the score returned by the LCV function.

3.2.2 Weighted-max-domain-size (WMD)

The intuition behind WMD [17] is that subtrees where variables have more than one value in their domain are more likely to have a solution. Like LCV it prefers values that lead to larger remaining domains for the remaining variables once assigned. The difference is that the the heuristic uses the minimal domain size in the future variables as a criterion. Since multiple variables may have the same minimal domain size after propagation of a decision, the value returned by the WMD heuristic is the number of those variables. Algorithm 8 gives a description of the WMD function.

Algorithm 8 Weighted-max-domain-size value selector

```
1: function WMD(x_i, v_k)
2:     minDomSize ← maximal integer value
3:     counter ← 0
4:     add a temporary node to the DFS stack
5:     assign x_i to v_k
6:     propagate the decision
7:     for x_j ∈ X ∧ x_j ≠ x_i do
8:         if x_j wat not previously bound then
9:             if |D(x_j)| < minDomSize then
10:                minDomSize ← D(x_j)
11:                counter ← 1
12:            else if D(x_j) == minDomSize then
13:                counter ← counter +1
14:            end if
15:        end if
16:    end for
17:    remove the temporary node from the DFS stack
18:    return minDomSize, counter
19: end function
```

The search strategy will first select the subset of values that maximize the minDomSize score returned by the WMD function. When more than one value is part of the subset, ties are broken by maximizing the counter score of the WMD function.

3.2.3 Bound-Impact Value Selector (BIVS)

The bound-impact value selector from [14] is a value heuristic which selects the value of a variable that leads to the best objective bound after propagation of the decision. We already described BIVS in Algorithm 3 in section **??**. In comparison the the previously described value ordering heuristics, BIVS can only be used in optimization problems, since it requires the existence of an objective variable. Similar to the OBS variable ordering heuristic from section 3.1.4, the goal of BIVS is to find good quality solutions as soon as possible in order to prune the search space afterwards with help of branch and bound.

3.3 Combining heuristics

In section 3.1.2 we have already seen that heuristics like dom/deg combine two heuristics to form a new one. In this section we are going to describe three state of the art techniques that can be used to combine any two heuristics. Notice that in the examples we will combine variable heuristics, but the same reasoning can be extended to value ordering heuristics as well.

3.3.1 Secondary heuristic as tie-breaker

In [10], Brelaz proposed a new method for solving the graph coloring problem. His reasoning can easily be extended to combine any two heuristics on any constraint problem. We denote varH_1 a first variable heuristic and varH_2 a secondary variable heuristic. The search strategy will first ask varH_1 which variable to assign next. If multiple variables have the same score, varH_1 should return all of those variables. If multiple candidate variables are selected by varH_1, the search strategy needs to break ties. Instead of breaking ties in lexicographical order or by simply selecting a random variable, a secondary heuristic varH_2 will select the variable optimizing its criterion out of the candidate set. Algorithm 9 describes a function that uses a secondary variable heuristic as tie breaker. Depending on the problem, multiple variables might optimize the criterion of the secondary variable ordering heuristic too. For this reason, varH_2 needs to implement a tie breaking rule.

Algorithm 9 Secondary heuristic as tie-breaker

```
1: function NextVar
2:     candidates ← varH1(∀ xi ∈ X ∧ |D(xi)| > 1)
3:     if |candidates| > 1 then
4:         next ← varH2(candidates)
5:     else
6:         next ← candidates
7:     end if
8:     return next
9: end function
```

3.3.2 Hybridization through scaling

The main reason why two heuristics should not be combined directly is because of the range differences of their features. If we have feature values for a variable ordering heuristic varH_1 in the range [0, 120] and feature values of a second variable ordering heuristic varH_2 in the range [-8, -2], then by simply adding both feature scores we would end up with a bias towards the first heuristic. Hybridization trough scaling fits the features of both heuristics in the range [0,1] before summing them together. We denote $f_i(x_i)$ the feature value of variable $x_i \in X$ for some heuristic, f_{min} and f_{max} the minimum respectively maximum feature values at the moment of scaling. We are able to scale $f_i(x_i)$ in the range [0,1] with the following formula:

$$f'_i(x_i) = \frac{f_i(x_i) - f_{min}}{f_{max} - f_{min}}$$

Notice that f_{min} and f_{max} may have to be updated several times during the search depending on the heuristics used. Once the feature values of both heuristics are correctly scaled, we can combine them:

$$S(x_i) = \alpha \times varH'_1(x_i) + (1 - \alpha) \times varH'_2(x_i)$$

The parameter $\alpha \in [0, 1]$ represents the importance given to each variable heuristic. By setting α to 0.5, we give equal importance to both heuristics.

3.3.3 Hybridization through Pareto optimality

The intuition behind hybridization of heuristics through Pareto optimality [26] is to find the variable that maximizes the scores of both heuristics simultaneously. In order to define a Pareto optimal variable in this context, we first need to define when a variable is able to dominate another variable according to two variable ordering heuristics heu_1 and heu_2. Li & Li give the following definition:

Definition 3.3.1. given two future variables x_i and x_j, we ay that x_i dominates x_j with respect to heu_1 and heu_2 iff either

$$sc_1[x_i] \geq sc_1[_j] \text{ and } sc_2[x_i] > sc_2[x_j], \text{ or}$$
$$sc_2[x_i] \geq sc_2[_j] \text{ and } sc_1[x_i] > sc_1[x_j]$$

where $sc_1[x_i]$ and $sc_2[x_i]$ are the heuristic score of variables x_i calculated by heu_1 and heu_2 respectively. A Pareto optimal variable x_i is a variable such that no other future variable $x_k \in X$ is able to dominate x_i. Notice that there can be multiple Pareto optimal variables as a Pareto optimum is not guaranteed to be unique. Algorithm 10 describes the procedure for computing the Pareto set i.e. the set of variables that are Pareto optimal.

Algorithm 10 Secondary heuristic as tie-breaker

```
1:  procedure ParetoSet(sc1, sc2)
2:      ParetoSet ← ∅
3:      for x_i ∈ X ∧ |D(x_i)| > 1 do
4:          isPareto ← true
5:          for x_j ∈ ParetoSet do
6:              if x_i dominates x_j then
7:                  ParetoSet ← ParetoSet \{x_j}
8:              else
9:                  if x_j dominates x_i then
10:                     isPareto ← false
11:                 end if
12:                 break
13:             end if
14:         end for
15:         if isPareto then
16:             ParetoSet ← ParetoSet ∪ {x_i}
17:         end if
18:     end for
19: end procedure
```

The paretoSet might contain more than one candidate variable, thus the search strategy needs to use some kind of tie breaking rule. The authors of [26] suggest to select a variable randomly, but other tie breaking rules may be used.

3.3.4 Comparing combination heuristics

We conclude this section by comparing the three previously described combination heuristics. We consider a partially defined COP which consists of a set of variables

	x_1	x_2	x_3	x_4	x_5	x_6
ABS	2	2	1	1	3	4
ABS scaled	0.33	0.33	0	0	0.66	1
OBS	-2	-1	-1	-3	-2	-3
OBS scaled	0.5	1	1	0	0.5	0
OBS scaled + ABS scaled	0.88	1.33	1	0	1.16	1

Table 3.1: ABS and OBS feature values for 6 variables

$X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ each with a domain, subject to some constraints and part of an objective function. The exact domains and constraints as well as the objective function are not of any importance in this example. Instead we assume, that for some complete formulation of the COP, the scores of two variable ordering heuristics (for instance ABS and OBS) are fixed to the values in Table 3.1 at some point during search. The scaled values for both heuristics as well as their sum are also reported in the table.

The order in which each of the combination heuristics would select the 6 variables is given by:

ABS with tie breaking OBS:	$x_6 \rightarrow x_5 \rightarrow x_2 \rightarrow x_1 \rightarrow x_3 \rightarrow x_4$
OBS with tie breaking ABS:	$x_2 \rightarrow x_3 \rightarrow x_5 \rightarrow x_1 \rightarrow x_6 \rightarrow x_4$
Scaled OBS + ABS:	$x_2 \rightarrow x_5 \rightarrow x_3$ or $x_6 \rightarrow \rightarrow x_1 \rightarrow x_4$
Pareto ABS and OBS:	x_2 or x_5 or $x_6 \rightarrow x_1$ or $x_3 \rightarrow x_4$

As we can see, no two combination heuristics of ABS and OBS produce the same variable ordering. However all four heuristics would select x_2 and x_5 in their first three decisions and would assign x_4 at last.

3.4 Enhancing backtracking strategy

One major disadvantage of backtracking search for solving constraint problems is that it suffers from the heavy-tailed phenomenon as shown in [19]. In an experiment in [21], the authors ran a backtracking search solver with randomized search strategy on a CSP for 10.000 runs and reported the number of backtracks for each run. The results perfectly illustrate the heavy-tailed phenomenon. Even tough the median number of backtracks is less than 2.000, about 5% of the runs took over 20.000 backtracks and in about 1% of the runs the solver was not able to provide a solution even after 200.000 backtracks. One of the reasons for this enormous amount of backtracks is thrashing - the fact of repeatedly exploring similar or even

identical subtrees during search. The rest of this section will describe techniques that can be used to limit the effects of heavy-tailed distributions and thrashing during backtracking search.

3.4.1 Randomization

Randomization in the search strategy can be implemented in the tie-breaking rule of the variable and value ordering heuristics. However, for some heuristics tie breaking is a pretty rare occurrence. The authors of [19] propose the introduction of a "heuristic equivalence" parameter H to the heuristic. The parameter represents a percentage such that all variables having a score higher than H times the highest score are considered equally good. The resulting subset of candidate variables will be able to profit from a randomized tie breaking rule.

Randomization mitigates the effect of thrashing by not repeating the same decisions again and again. The search strategy will therefore have better chances to escape a fruitless subtree.

3.4.2 Restarts

Another very popular approach to mitigate the effects of trashing is to restart the search when being stuck in an unpromising subtree. Restarts need to be combined with randomization or nogood learning (see next section) to guide the search towards parts of the search tree that are different from the one previously explored. There are two main criteria that determine when to restart the search: a time limit or a limit on the number of backtracks. When a solver is subject to a time limit which generally occurs in competitions, restarting the search after some duration will give a guarantee on how often the search is restarted. On the other side, the performance of filtering algorithms depends on the complexity of the problem. For very constraint problems, filtering algorithms may need a lot of time to establish consistency and a time limit might restart the search too early.

Restarts should not be used during the optimality proof of the search strategy. During this last part of the search, the search strategy needs to explore the entire search tree. By repeatedly restarting the search, the search strategy won't be able to explore the search tree in its entirety.

3.4.3 Nogood Learning

Nogood Learning or nogood recording is an idea that originated from assumption-based truth maintenance systems (ATMS) [11]. In constraint programming, a nogood (A, c) [45] is an assignment A that violates a constraint c. Nogoods are

recorded each time an inconsistency is encountered and are added as a constraint to the solver. The constraint that is added specifies that the variable can never be assigned to the value that led to a failure. If we reconsider Figure 2.2 we see that the search strategy tried the assignments $x_3 = 1$ and $x_3 = 2$ two times during the search. Unsurprisingly, the assignments also led to inconsistencies the second time they were tried out and the solver had to backtrack. With nogood learning, a constraint would have been added after the first assignments of $x_3 = 1$ and $x_3 = 2$ prohibiting the assignments later in the search. With those two additional constraints, the search strategy would directly assign x_3 to the value 3 which would reduce the overall size of the search tree.

In comparison with randomization, nogood learning gives a guarantee that an inconsistent assignment will never be repeated during search. With randomization, repeating the same mistake just becomes very unlikely.

3.4.4 Phase Saving

Phase saving [34] can be considered as a complementary value ordering heuristic. Whenever a variable is assigned to a value that leads to a consistent state, the value is recorded. The next time the variable has to be assigned during search, phase saving will give priority to the value that was previously recorded. In case no value is recorded for the given variable, or if the recorded value is not part of the variable's domain anymore, the actual value heuristic will select the next value.

3.4.5 Solution-Based Phase Saving

Solution-based Phase Saving [13] is an extension of phase saving where the recorded value is the value a variable was assigned to in the previous solution. Similar to phase saving, it relies on an existing value ordering heuristic that selects values in two cases:

1. No solution has been found yet
2. A solution has been found, but the recorded value has been removed from the variable's domain

To get the most out of solution-based phase saving, the heuristic should be combined with restarts and nogood learning. Once a solution has been found, restarts are used to simulate local search behavior. After a restart, variables will be assigned to their value in the previous solution. Thus, the path from the root node to the solution node is reconstructed step by step. However, the search strategy will have to take another decision before reaching the solution node, because a constraint is added each time a solution is found that the next solution has to be strictly better. At this point, the solver starts a search around the current best solution just

like Large neighborhood search [46]. One major difference between solution-based phasing and large neighborhood search is the creation of the neighborhood. In large neighborhood search a proportion of the variables is fixed in advance. If the neighborhood is small, i.e. the proportion of fixed variables is high, the search might terminate before reaching its limits. Solution-based phase saving on the other hand can dynamically increase its neighborhood thanks to the backtracking nature of the search algorithm.

3.5 Objective Landscapes

Objective Landscapes were first introduced in [23] and are actively used in IBM's CP Optimizer [1]. They address the problem that most search strategies do not have a global vision over the problem, especially how decision variables may influence the objective function. To explain how objective landscapes are computed, we consider a COP defined as:

$$\begin{array}{ll} \text{minimize:} & z = f(X) \\ \text{subject to:} & c(X, Y) \end{array}$$

where $X = \{x_1, \ldots, x_n\}$ is the set of decision variables and $Y = \{y_1, \ldots, y_m\}$ denotes the set of auxiliary variables to the problem. For the rest of this section the following conventions are used:

- x_i^L is a lower bound on the value of variable x_i
- x_i^U is an upper bound on the value of variable x_i
- x_i^S is the value of variable x_i in solution S
- Z^L is a lower bound on the objective function f
- Z^U is an upper bound on the objective function f

The computation of objective landscapes requires a function $f_i^* : D(x_i) \rightarrow \mathbb{R}$ such that $f_i^*(v)$ returns the optimal objective value one can obtain when x_i is assigned to v. Mathematically we have that:

$$f_i^*(v) = argmin_{x_i=v} f(X)$$

The function presumes that the constraint solver is able to propagate constraints of the form $f(x) \leq z$ for $z \in [Z_i^L, Z_i^L]$. Additionally, we need to introduce the following terminology:

- Z^L, the smallest value of z such that the propagation of objective cut $f(x) \leq z$ does not fail

[1] `https://www.ibm.com/analytics/cplex-cp-optimizer`

- $X_i^L(z)$ for $z \geq Z^L$ as the lower bound on variable x_i obtained after propagation of $f(x) \leq z$
- $X_i^U(z)$ for $z \geq Z^L$ as the upper bound on variable x_i obtained after propagation of $f(x) \leq z$

Algorithm 11 describes how the values for $\mathrm{X}_i^L(z)$ and $\mathrm{X}_i^U(z)$ are computed.

Algorithm 11 Preliminary computations for objective landscapes

```
1: for j in 1..k do
2:     propagate(f(x) ≤ z_j)
3:     for i in 1..n do
4:         X_i^L(z) ← current lower bound of x_i
5:         X_i^U(z) ← current upper bound of x_i
6:     end for
7: end for
```

Notice that in the first for loop, k values are propagated. The authors of [23] suggest to update the value z_j using:

$$z_j = \frac{Z^U - Z^L}{2^{j-1}} + Z^L$$

In fact only a few tens of values for z_j are needed to approximate the function f_i^* with enough precision.

Finally, we can compute an objective landscape function L_i of a decision variable x_i with:

- For $v \in [X_i^L(Z^L),\ X_i^U(Z^L)]$: $L_i(v) = Z^L$
- For $v < X_i^L(Z^L)$: $L_i(v) = \min\ \{z | X_i^L(z) \leq v\}$
- For $v > X_i^U(Z^L)$: $L_i(v) = \min\ \{z | X_i^U(z) \geq v\}$

The objective landscape function returns for each variable $x_i \in X$ and value $v \in D(x_i)$, the best objective value that is obtained when x_i is assigned to v. To illustrate the usage of objective landscapes during search we conclude this section with an example.

Example 3.5.1. Consider the COP defined over the set of variables $X = \{x_1,\ x_2,\ x_3,\ x_4\}$ with respective domains $D(x_1) = D(x_2) = D(x_3) = D(x_4) = \{1, 2, 3, 4\}$. The variables are subject to the constraint $\mathit{allDifferent}(x_1, x_2, x_3, x_4)$ and the objective is to minimize$(x_1 + x_2)$. At the beginning of the search we have that:

- the lower bound of the objective function $Z^L = 2$,
- the upper bound of the objective function $Z^U = 8$

The first step for defining the objective landscape consists in computing the values for z_j:

- $z_1 = \frac{8-2}{2^0} + 2 = 8$
- $z_2 = \frac{8-2}{2^1} + 2 = 5$
- $z_3 = \frac{8-2}{2^2} + 2 = 3$
- $z_4 = \frac{8-2}{2^3} + 2 = 2$

The next step consists in using Algorithm 11 to compute the bounds of each decision variable:

$X_1^U(8) = 4$	$X_1^U(5) = 4$	$X_1^U(3) = 2$	$X_1^U(2) = 1$
$X_1^L(8) = 1$	$X_1^L(5) = 1$	$X_1^L(3) = 1$	$X_1^L(2) = 1$

$X_2^U(8) = 4$	$X_2^U(5) = 4$	$X_2^U(3) = 2$	$X_2^U(2) = 1$
$X_2^L(8) = 1$	$X_2^L(5) = 1$	$X_2^L(3) = 1$	$X_2^L(2) = 1$

The objective landscape function will then be defined as:

$L_1(1) = 1$	$L_1(2) = 3$	$L_1(3) = 5$	$L_1(4) = 5$
$L_2(1) = 1$	$L_2(2) = 3$	$L_2(3) = 5$	$L_2(4) = 5$

We can exploit the landscape function in different ways during search. When using it as a value ordering heuristic, the value having the lowest objective landscape score is selected first. In this example, the value 1 has the lowest score, since its assignment to one of the variables leads to the best objective value.

The landscape function can also be used to reduce the domains of the decision variables during search when finding a solution. Imagine that we find a solution where $x_1 = 3$ and $x_2 = 1$. As a result, the objective value of the solution is equal to 4. If we loop through the objective landscape function for all values $v \in D(x_1)$ and $D(x_2)$, we can conclude that the values 3 and 4 can be removed from the domains of both variables. The reason is that the best objective value obtainable when one of the variables is assigned to them is 5 which is worse than the current best solution.

Chapter 4

Parsing XCSP3 instances

4.1 The XCSP3 format

XSCP3 is an XML-based intermediate CP format preserving the structure of problem models [8]. The authors of XCSP3 decided to use XML to support the overall architecture of the XCSP3 language in order to make the parsing of instances for constraint solvers fairly straightforward. A typical skeleton of an XCSP3 Problem instance is shown in Figure 4.1.

Syntax 1

```
<instance format="XCSP3" type="frameworkType">
  <variables>
    (   <var.../>
      | <array.../>
    )+
  </variables>
  <constraints>
    (   <constraint.../>
      | <metaConstraint.../>
      | <group.../>
      | <block.../>
    )*
  </constraints>
  [<objectives [combination="combinationType"]>
    (   <minimize.../>
      | <maximize.../>
    )+
   </objectives>]
  [<annotations.../>]
</instance>
```

Figure 4.1: Syntax of a typical XCSP3 problem instance from [8]

A detailed description of all the features XCSP3 is providing is out of the scope of this thesis. We should only mention the concepts that may influence the performance of our search strategy. All variables are declared at the beginning of the file. In XCSP3, it is also possible to declare arrays of variables. An array of variables corresponds to a set of variables that represents the same concept during the modeling phase. Each variable needs to be given an identification name (id) and a domain. Notice that variables that are declared inside an array are all given the same identification prefix. Constraints are declared next in the file. The constraints supported by the XCSP3 format are described in the documentation and a solver should implement the constraints internally. Finally the file may end with two more optional tags. The first tag is used in COPs to define an objective (minimization or maximization) over a subset of variables declared earlier in the file. The last tag may indicate if present which subset of variables are decision variables.

4.2 Extracting decision variables from XCSP3 instances

The identification of decision variables is a crucial step for any constraint solver. Informally, the set of decision variables represents the "real-world" decisions that have to be taken in the problem. The remaining variables, which we will refer to as auxiliary variables are linked to the decision variables via constraints. Auxiliary variables will thus restrict some combinations of assignments for decision variables. An important property of decision variables is that once they are assigned to some value, the remaining auxiliary variables will be assigned automatically as a result of constraint propagation. Modern constraint solvers will therefore only branch on decision variables whenever possible.

In case of XCSP3 files, the problem has already been modeled. If the decision variables are not annotated, there is no guarantee that the solver is able to detect them correctly. To tackle this problem, the current OscaR implementation uses information about how the variables interact in constraints. It considers that some constraints indicate decision variables while others do not. Since there will always be an uncertainty about the correct classification of decision variables, the search is split into two parts. The first part of the search will only branch on the decision variables. If all the decision variables are assigned, two scenarios can occur:

1. The solver finds a solution because the auxiliary variables are bound as a result of constraint propagation
2. The solver does not find a solution because some auxiliary variables are still unbound

In the latter case, the solver did not correctly classify the decision variables and some of them are disguised as auxiliary variables. In order to find a solution, the solver will have to start a second search on the auxiliary variables.

4.2.1 Performance of current strategy

To assess the performance of the current decision variable extraction technique, we ran the OscaR solver on all of the 2018 XCSP3 instances containing variables with more than one id prefix. This accounts for a total of 207 instances from 16 different families of problems. The system used for the experiments consists of an Intel Core i5-6600K CPU with 16GB of RAM running Fedora 30. The solver was run with parameters equivalent to the evaluation framework of the competition.

During the experiment we recorded the following information:

- the id(s) of the variables that are considered as decision variables
- the id(s) of the variables that are considered as auxiliary variables
- The number of times the solver is not able to find a solution (timeout)

Table A.1 contains the detailed results about the searches for each family of instances. Our first observation suggests that the solver considers too many variables as decision variables. *GolomRuler*, *Mario*, *QuadraticAssignment*, *TravelingTournamet* and *TravallingSalesman* are the only families for which the solver is able to make a distinction between decision and auxiliary variables. Notice that for those families the solver only timed out on 1 out of 58 instances. On the other hand, the families of *Fapp*, *NurseRostering* and *SteelMillSlab* are the most problematic. The solver is able to find a solution for only 14 out of 56 instances. We will now introduce a criterion that selects only the most promising subset of variables as decision variables.

4.2.2 Decision variables from constraint degrees

Extracting decision variables from a model is a non trivial task. In the best case the decision variables are already annotated in the file but the solver should also perform well when the annotation is missing. To motivate the choice for our alternative extraction technique, we are going to make an assumption:

Assumption 4.2.1. The subset of decision variables is the most constrained one with respect to its size.

The assumption is based on the fact that assigning only decision variables will already lead to a complete instantiation of the problem. This property of decision variables suggest that they need to be highly constrained in order to reduce the domains of the auxiliary variables. With this assumption in mind, we can develop

a new criterion for selecting decision variables. Algorithm 12 selects the set of variables with the same id prefix having the highest (constraint degree/size of the set) ratio.

Algorithm 12 Selecting decision variables

```
1: degree ← 0
2: nVars ← 0
3: bestDegree ← 0
4: bestSize ← 0
5: bestId ← null
6: for each variable x_i do
7:     if id of x_i ≠ id of x_{i−1} ∨ x_i is the last variable then
8:         score ← degree/nVars
9:         if score > bestScore then
10:            bestScore ← degree
11:            bestSize ← nVars
12:            bestId ← id of variable x_i
13:        else if degree = bestScore ∧ nVars < bestSize then
14:            bestScore ← degree
15:            bestSize ← nVars
16:            bestId ← id of variable x_i
17:        end if
18:        degree ← 0
19:        nVars ← 0
20:    end if
21:    nConstraints ← nConstraints + constraint degree of x_i
22:    nVars ← nVars + 1
23: end for
24: decision variables ← the subset of variables with the same id as bestId
```

The same experiments from section 4.2 were reconducted with the new decision variable extraction method. The results are reported in Table A.2. This time, the solver always selected one single subset of variables as decision variables. The overall number of timeouts went down from 66 to 41. Figure 4.2 illustrates the difference in the number of timeouts for each family between the two methods. The new criterion decreases the number of timeouts on 5 different families. However, the criterion is not optimal since the number of timeouts increases for *TemplateDesign* and *Tal*. The assumption made to establish the criterion does not seem to be valid for all instances of the XCSP3 competition. To conclude, Algorithm 12 will be part of the new OscaR solver since the overall impact seems to improve the search.

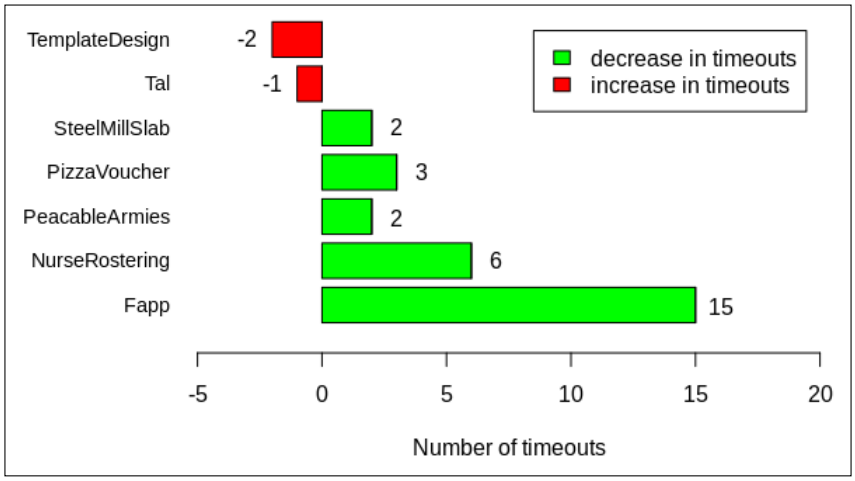
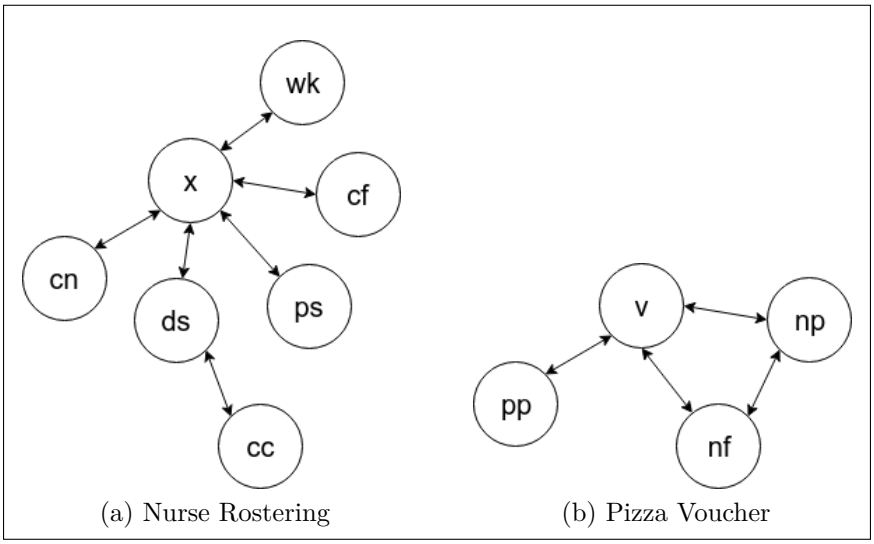


Figure 4.2: Difference in timeouts

4.2.3 Further work

As an alternative we propose the usage of a constraint graph to tackle the decision variable classification problem. A constraint graph illustrates how the variables in the problem are linked through constraints. Constraint graphs for the nurse rostering and pizza voucher problems are shown in Figure 4.3 (a) and (b) respectively.



(a) Nurse Rostering (b) Pizza Voucher

Figure 4.3: Constraint graphs for the family of nurse rostering (left) and pizza voucher (right) instances

An interesting criterion for extracting decision variables from a constraint graph is to consider the sum of path lengths to reach any other node in the graph. Considering Figure 4.3 this criterion would select the variables with id prefix x for nurse rostering and the variables with id prefix v for pizza voucher as decision variables. However, the implementation and exploitation of constraint graphs in OscaR will be left for future work.

Chapter 5

Development of a new search strategy

The purpose of this chapter is to describe the search strategy OscaR will use to compete in the 2019 XCSP3 competition in the fast COP category. In order to compare the performance of the new search strategy, the following three solvers will be used as a baseline:

OcsaR - Conflict Ordering with restarts 2018-08-17 [32]. The previous search strategy of OscaR uses a conflict ordering search in combination with restarts. The duration of the first search is fixed to 10 seconds and is increased by 30% after each restart to maintain completeness. When the conflict set is empty, the solver uses SDF as a fallback variable ordering heuristic. The value ordering heuristic is a combination of randomized value selection and phase saving. In the previous competition, OscaR was ranked 3^{rd} in the complete solver point of view and 4^{th} in the incomplete solver point of view.

Choco-solver 4.0.7b seq (e747e1e) [35]. Choco-solver obtained the 1^{st} place in the complete solver point of view and the $2nd$ place in the incomplete solver point of view. Details about the search strategy used during the competition are not provided.

Concrete 3.9.2. Concrete solves COPs by using a binary depth-first tree search. The search strategy uses dom/wdeg with random tie-breaking as a default variable ordering heuristic. The value heuristic is a combination of solution-based phase saving and BIVS where ties are broken randomly. The search is restarted periodically with geometric growth.

To assess performance of the new search strategy, we ran different variations and combinations of heuristics on all fast COP instances of the 2018 XCSP competition.

The instances can be downloaded directly from their webiste[1]. The experiments were done on a distant server with two 22c/12t AMD6167 processors and 48GB of RAM running CentOS Linux version 7. To recreate the evaluation environment from the competition, we used a timeout limit of 240 seconds, a memory limit of 1000MB and a CPU core limit of 1. Furthermore, each experiment was repeated three times with a different seed. The reported results correspond to the average of the 3 runs rounded to the best value.

The sections that follow describe the search strategy adopted during the different phases of the search. Section 5.1 proposes a search strategy that is able to find an initial solution for a majority of the instances. In section 5.2 we introduce additional techniques to enhance the search strategy when a solution is known. Finally in section 5.3 the goal of the search strategy is to prove optimality which requires heuristics that work well when no better solution exists.

5.1 Searching a first solution

During the 2018 XCSP3 competition, 346 different instances were given as input to the solvers. In order to have an overview of how many instances were proved to be satisfiable in average, we illustrate the results of two solvers in Figure 5.1. With 84 timeouts, OscaR is one of the solvers that timed out the most without finding a solution. On the other hand, Choco-solver performed best regarding timeouts by leaving only 64 instances unsolved.

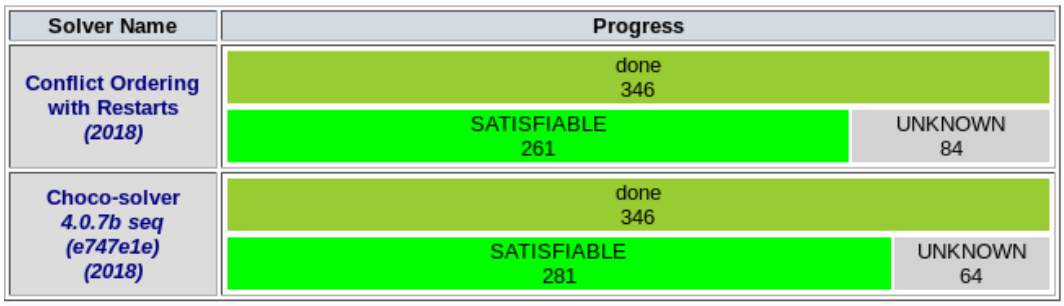


Figure 5.1: Comparison of OscaR and Choco on the number of solutions found

The results from Figure 5.1 are used as a baseline for the first phase of the search strategy. During the first phase, the solver should be able to find a solution to more than 261 instances to be an improvement over the old search strategy. In the best case the solver is able to time out less than Choco-solver.

[1] `http://xcsp.org/series`

5.1.1 Description of the search strategy

The search strategy performs a binary depth-first traversal of the search tree. The variable ordering heuristic consists in a combination of ABS and OBS (ABS_O). In section 3.3 we enumerated different techniques to combine both heuristics. Preliminary experiments on a subset of the instances showed that hybridization through scaling seems the most promising. The activities of ABS are updated according to the rules defined in 3.1.3. However, the rule for updating the weighted sum of Δ_O values in OBS is slightly modified. We replace it with:

$$\widetilde{\Delta_O(x_i)} = \widetilde{\Delta_O(x_i)} \times (1 - \gamma) + \gamma \times \Delta_O(x_i)$$

The difference with the formula defined in section 3.1.4 is the removal of the division by γ. The reason is that in some experiments where γ was set to a value lower than 1, the division by γ led to rather strange updates of the Δ_O value. In case multiple unbound variables have the same heuristic score, one of them is selected randomly.

The feature values of ABS and OBS are initialized by probing before starting the search. During the probing process, only the decision variables are selected for assignment. The duration of the probing process depends on two stopping criteria. The first criterion is a statistical significance test. The probing process is terminated once the confidence interval is within 20% of the empirical mean for all variables and both heuristics. The second criterion is a maximal duration of 8% of the total search time. If no statistical significance is achieved after this duration, then the process is terminated.

The value ordering heuristic that seems most promising is BIVS. It is one of the only existing value heuristics that takes into account the objective function. In OscaR, BIVS follows an implementation similar to Algorithm 3. The only difference is that the implementation in OscaR does not consider all the values in domains that a larger than a given size. For very large domains, BIVS requires a lot of computational time because each assignment has to be propagated. We therefore limit the maximum number of considered values to 25. If a variable with a larger domain is selected for assignment, then 25 random values out of its domain will be considered by BIVS.

Restarts are usually necessary to avoid the long tail phenomenon. Since we do not need to guarantee completeness of the solver during the first phase of the search, a predefined number of backtracks is used as a criterion for restarts. In the current implementation, the number of backtracks after which the search is restarted is set to (number of variables)/2.

In order to avoid thrashing, the search strategy relies on nogood learning in addition to randomized tie-breaking rules. Especially when restarting the search frequently,

remembering previously explored parts of the search tree drastically reduces the search effort.

Another consideration to enhance the robustness of the search strategy is the usage of conflict ordering search. The OBS heuristic does not follow the fail-first principle and selects variables purely on their impact on the objective function. With conflict ordering, highly constraint variables that often lead to failures are assigned earlier in the search, thereby increasing the chance of finding a solution.

5.1.2 Experimental results

In this first experiment we ran four slightly different versions of the previously described search strategy. The search ended once a solution was found, since we are only interested in the quality of the first solution and in the number of timeouts for all instances. The results are summarized in Figure 5.2. The individual results for each instance are stored in a csv file and can be downloaded from a github repository [2].

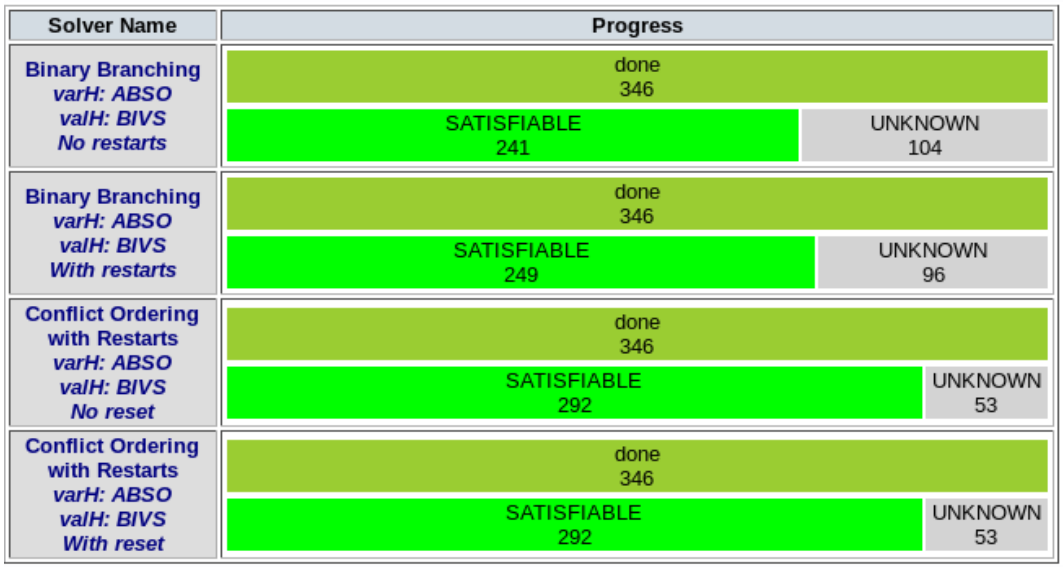


Figure 5.2: Different variations of search strategies using ABS_O and BIVS

The first solver does not use restarts nor conflict ordering during its search. Compared to the solvers in Figure 5.1 we can already conclude that it performs poorly.

The second solver adds restarts to the search strategy and performs slightly better than the previous one. However, the improvement is still not good enough since the number of timeouts is still greater than 84.

[2] `https://github.com/gouya/EPL1819-1431.git`

The third solver uses conflict ordering on top of the scaled ABS_O variable ordering heuristic. This time, the improvement is very convincing. With the search strategy used in this solver, we are able to find a solution to 11 more instances than the Choco-solver.

The fourth solver also uses conflict ordering, but resets the conflict set at each restart. The results in Figure 5.2 are equivalent for both solvers with only 53 timeouts. In order to determine which search strategy to use in the upcoming competition, the quality of the solutions found are compared next.

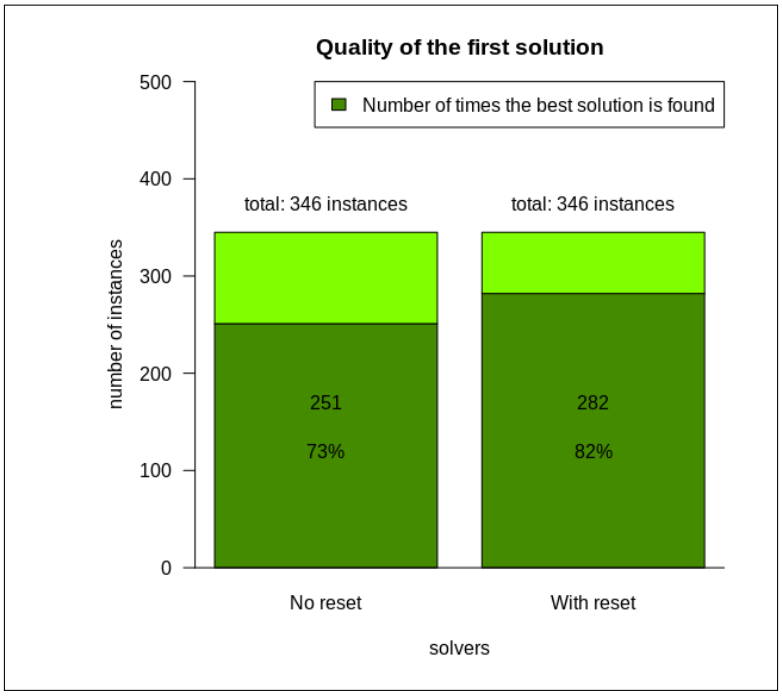


Figure 5.3: Comparison of resetting and not resetting the conflict set

Figure 5.3 illustrates how many times the best solution was found when resetting the conflict set compared to not resetting the conflict set after a restart. The results indicate that resetting the conflict set improves the quality of the first solution.

5.1.3 Conclusion

At this point, we have already improved the robustness of the OscaR solver in two aspects. First, we showed that extracting the right decision variables has a major impact on the search effort. The technique for extracting decision variables proposed in Chapter 4 already improved the robustness of the OscaR solver without changing the search strategy. Second, when combined with the search strategy

proposed in section 5.1.1 OscaR is able to prove satisfiability for more instances than any other solver.

5.2 Improving the solution

The second phase of the search strategy starts with an initial solution to the problem. The goal of this second phase is to improve the quality of the solution as much as possible. In this section we are not worried about proving optimality for a solution. Maintaining completeness of the solver is the topic of section

5.2.1 Description of the search strategy

The search strategy is build around solution-based phase saving since it seems to be a very promising heuristic for improving a solution. When the value of the previous solution is not in the domain of a variable anymore, BIVS is used as a fallback heuristic. We choose BIVS over other value heuristics because prefers the values that have the best impact on the objective bounds after propagation.

The authors of solution-based phase saving suggest to use ABS as a variable ordering heuristic to avoid defining highly restrictive neighbourhoods. Alternatively, [36] suggests to define neighborhoods from variables that are involved in conflicts. During the experiments, we compare both approaches. Furthermore, we also investigate the possibility of adding OBS to both heuristics in order to take into account to objective function.

In order to explore new neighborhoods, the search needs to be restarted. The duration of the search implicitly defines the size of the neighborhood. We use a luby sequence [28] where a random duration between 2 and 8 seconds is selected after each restart. Restarting the search at different frequencies creates neighborhoods of different sizes. Each time a search ends without finding a better solution, the probability for selecting a longer duration increases. In this way, we avoid repeatedly exploring neighborhoods that are too small.

During the search nogoods are recorded to avoid reexploring the same parts of the search tree after each restart. Both the variable and the value heuristic use the same randomized tie-breaking rules to guarantee a minimum of diversity during search.

An objective landscape function is used to remove values from the domains of decision variables that cannot contribute to a better solution (see example from section 3.5). Before each restart, the search strategy verifies that the values remaining in the domains of the decision variables can be part of a better solution by exploiting the landscape.

5.2.2 Experimental results

In this second experiment, we compare the usage of different variable ordering heuristics in combination with solution-based phase saving. The results are depicted in Figure 5.4.

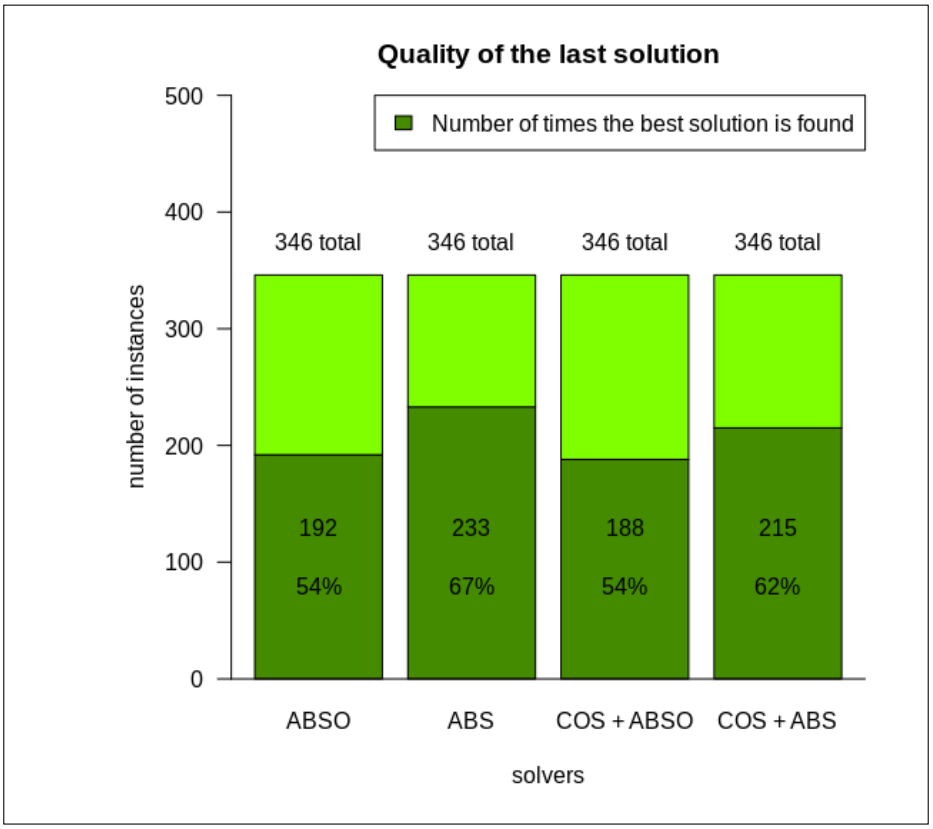


Figure 5.4: Comparison of different variable ordering heuristics in combination with solution-based phase saving

From the suggestions made about variable ordering heuristics in combination with solution-based phase saving in [13], we decided to experimentally compare ABS with ABS_O as well as the impact of conflict ordering search.

The results suggest that ABS is better suited in combination with solution-based phase saving than ABS_O. A possible explanation is that ABS updates the activities of several variables after the propagation of a decision. In a certain sense, similar activity scores indicate that variables might be related. On the other hand, OBS updates the score of a variable only when it has been assigned. For this reason, the score of OBS does not reflect how variables interact with each other.

Conflict ordering does not seem to have a positive impact either. The experiments confirm the conclusion from [13] that *ABS* is indeed best suited for defining neighborhoods during search.

In order to compare the results with solvers that participated in the 2018 XCSP3 competition, we denote OscaR 2019 the solver that uses solution-based phase saving with *ABS* as a variable heuristic. We start by comparing OscaR 2019 with the implementation of OscaR from last year (OscaR 2018). Figure 5.5 reports the number of times each solver is able to find the best solution. Notice that if both solvers find the same solution, then both solvers are given credit.

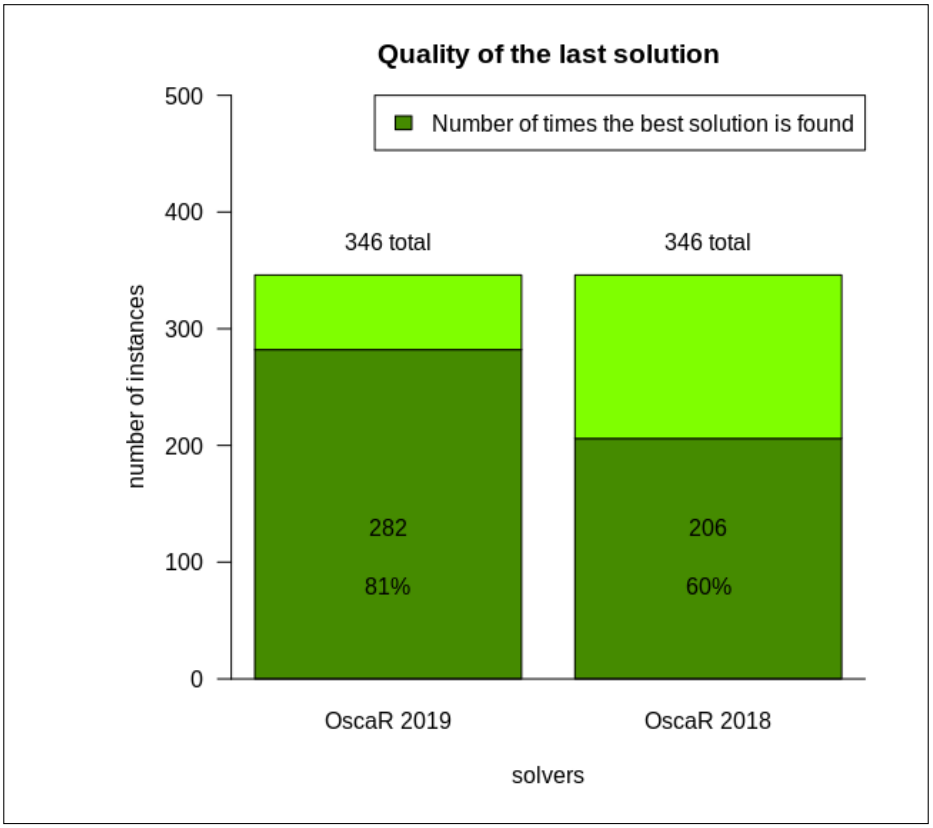


Figure 5.5: Comparison between the new and the old search strategy for OscaR

The new solver is able to outperform OscaR 2018 on 76 instances. The two families of problems where the new solver is not able to provide better solutions consistently are: *StillLife-wastage* and *Tal*. To give an explanation on why the new solver is having such difficulties on those instances we need to reconsider Table A.1 and A.2. OscaR 2018 considered all variables to be decision variables for both problems. As a consequence it branches on all the variables at the same time. The new solver relies on the decision variables extraction technique from 4.2.2 and considers only

one subset of variables to be decision variables. From the results obtained in this experiment, it seems that for the two families of problems in question, the solver did not select the right decision variables.

Next, we compare the performance of the new solver with Choco-solver. From an incomplete solver point of view, Choco placed 2^{nd} during the 2018 competition. The results are summarized in Figure 5.6.

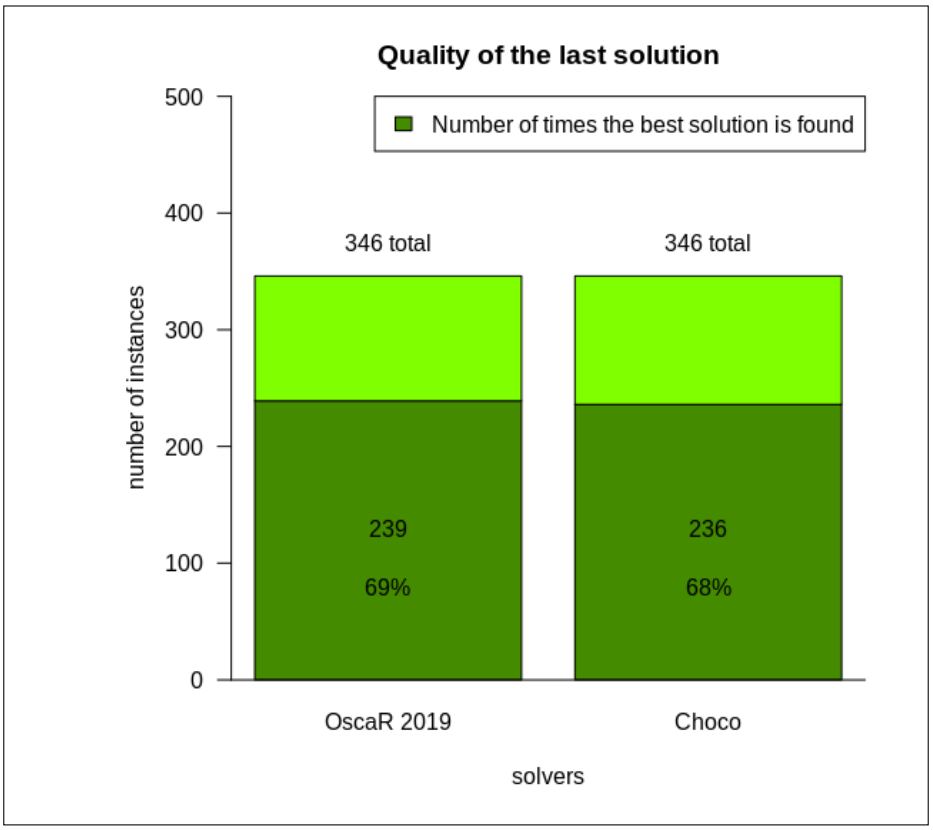


Figure 5.6: Comparing the new search strategy to Choco-solver

OscaR 2019 performs slightly better then Choco-solver, but the difference is minimal. The solver performs especially good on instances from the *Auction-cnt*/*Auction-sum*, *Knapsack*, *PeacableArmies* and *LowAutocorrelation* families. On the other hand, it does not perform well on *StillLife*, *GraphColoring*, *SumColroing* and *TSP* problems. The problems in *GraphColoring* and *SumColoring* are characterized by a large set of variables with large domains. The current implementation of BIVS only considers 25 values in a domain of a variable to reduce computational effort when selecting the value. Performing badly on those problems probably means that BIVS does not consider the right subset of values.

To conclude this section, we compare the performance of the new solver with Concrete. Concrete was the winner of the 2018 XCSP competition considering the incomplete solver point of view. The results are summarized in Figure 5.7.

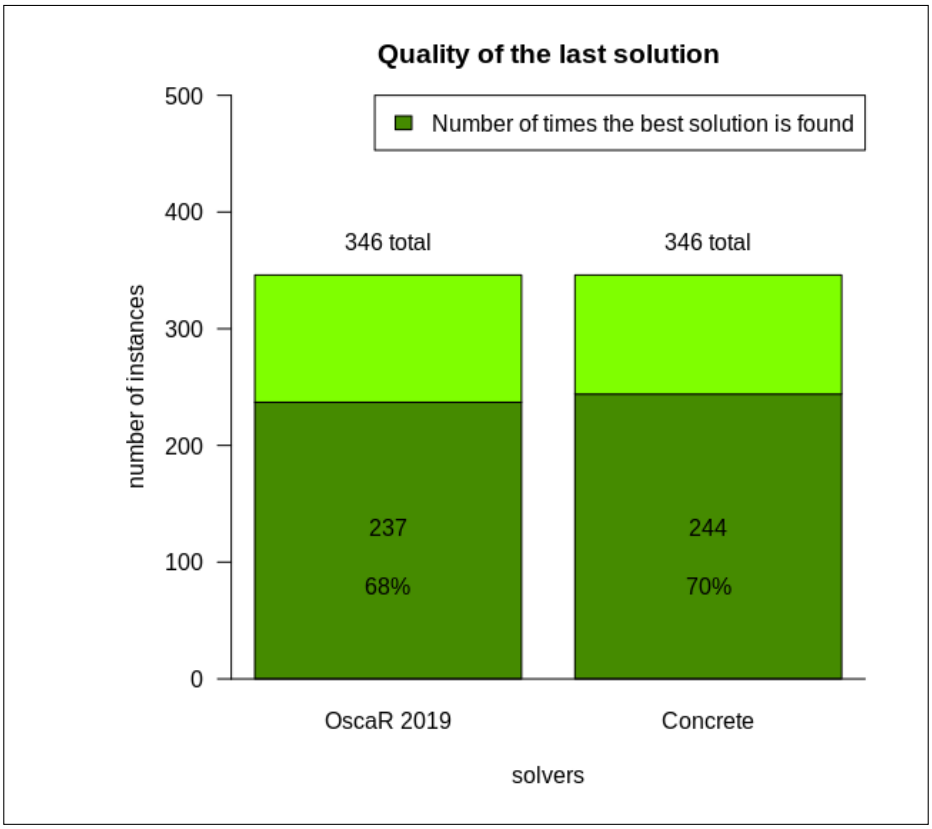


Figure 5.7: Comparing the new search strategy to Concrete

This time, OscaR 2019 is not able to perform better than the other solver. We should point out that both solvers use a similar search strategy according to the description file of Concrete [50]. Concrete also relies on solution-based phase saving and BIVS to improve the solution. The two main differences are the variable ordering heuristic and the frequency of the restarts. Concrete uses dom/wdeg as a variable ordering heuristic instead of ABS. To determine when to restart the search, Concrete replaces the luby sequence by a sequence of geometric growth. The reason for this is that the same strategy for improving the solution is also used to prove optimality. Another difference between the two solvers is how decision variables are extracted. Unfortunately, the description file contains no information on this subject.

5.3 Proving optimality

The last phase of the search strategy consists in proving optimality for a given solution. Knowing when to switch from improving the solution to proving optimality is a non-trivial task. In this thesis we suggest to prove optimality after n consecutive timeouts without finding a better solution during the improvement phase. In the experiments conducted for this section, n was set to 8.

When proving optimality trying to fail early in the search is reasonable, since it reduces the overall size of the search tree. Proving that no better solution exists in a COP is very similar to proving unsatisfiability for a CSP. For this reasons, previous heuristics such as BIVS and OBS become less effective.

5.3.1 Description of the search strategy

The two variable ordering heuristics that we take under consideration are ABS and dom/wdeg. We will always combine them with conflict ordering, because we want to give priority to variables that lead to failures.

The value heuristics that we considered for the optimality prove are LCV, and simply selecting the smallest value in the domain of the variable (minValue). In [14] the author recommended not to use BIVS when no solution exists. Its criterion relies on comparing objective bounds before and after an assignment. However, if no better solution exist, the criterion of BIVS fails.

Some solvers rely on restarts with geometric growth to maintain completeness of their search strategy. Since we do not know in advance when the solver is going to enter phase 3 of its search, we do not restart the search anymore to maintain completeness.

5.3.2 Experimental results

Table 5.1 illustrates how many times the solver was able to prove that it found the optimal solution to a problem. When ABS_O is used as a variable heuristic to guide the search before conflict ordering takes place, the number of optimal solutions is minimal. Activity-based search on its own performs a little bit better, but the results are still mediocre. The results are best when dom/wdeg is used on top of conflict ordering search.

Considering value heuristics, the simplistic minValue heuristic performs better than LCV on the instances of the 2018 XCSP3 competition.

	Number of optimal solutions
ABS & LCV	63
ABS & minValue	68
ABSO & LCV	56
ABSO & minValue	59
dom/wdeg & LCV	73
dom/wdeg & minValue	85

Table 5.1: Number of solved instances

5.4 Final results

To illustrate how the new OscaR solver would have performed in the last competition, we recreated the results with the new solver. The final search strategy is the result of the three phases that we just described.

5.4.1 Complete solver point of view

The results from a complete solver point of view are presented in table 5.2.

Rank	Solver	Number of solved instances
1.	Choco-solver	86
2.	Concrete	85
2.	Oscar 2019	85
4.	Oscar - Conflict ordering with restarts	83
5.	Concrete - SuperNG	83
6.	OscaR - Hybrid	59
7.	Cosoco	51
8.	Sat4j - CSP	41

Table 5.2: Results of the 2018 XCSP3 competition including the new solver: complete solver point of view

The OscaR 2019 solver shares the second place with Concrete. Compared to the OscaR solver from last year, it is able to prove optimality on two more instances.

5.4.2 Incomplete solver point of view

From an incomplete solver point of view the results are summarized in Table 5.3 Notice that in contrast to the results from the official website, ties are not broken by the cumulative T1 time.

Rank	Solver	Number of best solutions found
1.	Concrete	160
2.	Choco	151
3.	Oscar 2019	149
4.	OscaR - Hybrid	147
5.	Concrete - SuperNG	142
6.	OscaR - Conflict ordering with restarts	137
7.	Cosoco	114
8.	Sat4j - CSP	78

Table 5.3: Results of the 2018 XCSP3 competition including the new solver: incomplete solver point of view

This time Oscar 2019 places 3^{rd} in the overall ranking. According to the results from both point of views, the new search strategy is an improvement over both Oscar - Hybrid and Oscar - Conflict ordering with restarts. On the other hand, the results also show that there is still some room for improvements since we are unable to beat the best solver.

Chapter 6

Conclusion

Designing efficient black-box search strategies is one of the major concerns in constraint programming. Throughout this thesis we showed how variable and value orderings influence the efficiency of a constraint solver.

A new technique for extraction decision variables based on their constraint degree has been introduced in Chapter 4. We showed that the new extraction method is efficient for reducing the number of timeouts on the fast COP instances of the 2018 XCSP3 competition.

In Chapter 5 we proposed a new search strategy for the OscaR solver to compete in the upcoming XCSP3 competition in the fast COP category. The search strategy is split into three distinct phases where each phase corresponds to a different objective during the solving phase. In the first phase, the search strategy tries to find an initial solution of good quality for a majority of instances. Conflict ordering search in combination with ABS_O as variable and BIVS as value heuristic has proven to be best for this task. The second phase of the search strategy aims at improving the solution. It relies on solution-based phase saving to simulate local search behaviour during backtracking search. Experiments show that solution-based phase saving is best combined with *ABS* and BIVS. Additionally, an objective landscape is exploited in between restarts to remove values than can no longer contribute to a better solution. The last phase of the search strategy tries to prove optimality for the last solution found. Conflict ordering in combination with dom/wdeg has shown to be a reasonable choice.

In experimental results, the new search strategy has shown to be an improvement over OscaR - Hybrid and OscaR - Conflict ordering with restarts. However, there are still some improvements to make to be able to outperform the best solvers of the competition consistently.

Appendix A

Decision variable extraction

A.1 Extracting decision variables based on constraints

Instances (number)	decision variables	auxiliary variables	Timeouts
Bacp (4)	prd, nco, ncr, cp		0
CrossWordDesign (13)	r, c, pr, pc, br, bc, x		9
Fapp (18)	f, p, k, v1, v2, d		15
GolombRuler (3)	y	x	0
LowAutocorrelation (14)	x, y, c, s		0
Mario (10)	s	f, g	0
NurseRostering (21)	x, ps, ds, wk, cn, cf, cc		15
PeacableArmies (14)	b, w		5
PizzaVoucher (10)	v, np, nf, pp		4
QuadraticAssignment (19)	x	d	0
SteelMillSlab (17)	slab, load, loss		12
StillLife (13)	x, w, ws, z		0
Tal (10)	c, l, a, i, cnt		0
TemplateDesign (15)	d, npt, u, nptv		5
TravelingTournament (14)	o	h, a, t	1
TravellingSalesman (12)	c	d	0

Table A.1: Influence of decision variables on the number of timeouts

A.2 Extracting decision variables based on constraint degrees

Instances (number)	decision variables	auxiliary variables	Timeouts
Bacp (4)	prd	nco, ncr, cp	0
CrossWordDesign (13)	x	r, c, pr, pc, br, bc	9
Fapp (18)	f	p, k, v1, v2, d	0
GolombRuler (3)	y	x	0
LowAutocorrelation (14)	x	y, c, s	0
Mario (10)	s	f, g	0
NurseRostering (21)	x	ps, ds, wk, cn, cf, cc	9
PeacableArmies (14)	b	w	3
PizzaVoucher (10)	v	np, nf, pp	1
QuadraticAssignment (19)	x	d	0
SteelMillSlab (17)	slab	load, loss	10
StillLife (13)	x	w, ws, z	0
Tal (10)	a	c, l, i, cnt	1
TemplateDesign (15)	npt	d, u, nptv	7
TravelingTournament (14)	o	h, a, t	1
TravellingSalesman (12)	c	d	0

Table A.2: Influence of decision variables on the number of timeouts

Bibliography

[1] Krzysztof Apt. *Principles of constraint programming*. Cambridge university press, 2003.

[2] Egon Balas and Paolo Toth. Branch and bound methods for the traveling salesman problem. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA MANAGEMENT SCIENCES RESEARCH GROUP, 1983.

[3] Roman Bartak. Theory and practice of constraint propagation. In *Proceedings of the 3rd Workshop on Constraint Programming in Decision and Control*, volume 50, 2001.

[4] J Christopher Beck, Patrick Prosser, and Richard J Wallace. Toward understanding variable ordering heuristics for constraint satisfaction problems. In *Proceedings of the Fourteenth Irish Artificial Intelligence and Cognitive Science Conference (AICS03)*. Citeseer, 2003.

[5] J Christopher Beck, Patrick Prosser, and Richard J Wallace. Trying again to fail-first. In *International Workshop on Constraint Solving and Constraint Logic Programming*, pages 41–55. Springer, 2004.

[6] 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, *Principles and Practice of Constraint Programming — CP96*, pages 61–75, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[7] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence*, ECAI'04, pages 146–150, Amsterdam, The Netherlands, The Netherlands, 2004. IOS Press.

[8] Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. Xcsp3: An integrated format for benchmarking combinatorial constrained problems. *arXiv preprint arXiv:1611.03398*, 2016.

[9] Frédéric Boussemart, Christophe Lecoutre, Arnaud Malapert, and Cédric Piette. About benchmarking and competitions of solvers in constraint programming. *The International Planning Competition (WIPC-15)*, page 1, 2015.

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

[11] Johan De Kleer. A comparison of atms and csp techniques. In *IJCAI*, volume 89, pages 290–296. Citeseer, 1989.

[12] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. In *Search in artificial intelligence*, pages 370–425. Springer, 1988.

[13] Emir Demirović, Geoffrey Chu, and Peter J. Stuckey. Solution-based phase saving for cp: A value-selection heuristic to simulate local search behavior in complete solvers. In John Hooker, editor, *Principles and Practice of Constraint Programming*, pages 99–108, Cham, 2018. Springer International Publishing.

[14] J. Fages and C. Prud'Homme. Making the first solution good! In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1073–1077, Nov 2017.

[15] Daniel Frost and Rina Dechter. In search of the best constraint satisfaction search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1)*, AAAI '94, pages 301–306, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[16] Daniel Frost and Rina Dechter. In search of the best constraint satisfaction search. In *AAAI*, volume 94, pages 301–306, 1994.

[17] Daniel Frost, Rina Dechter, et al. Look-ahead value ordering for constraint satisfaction problems. In *IJCAI (1)*, pages 572–578. Citeseer, 1995.

[18] Steven Gay, Renaud Hartert, Christophe Lecoutre, and Pierre Schaus. Conflict ordering search for scheduling problems. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming*, pages 140–148, Cham, 2015. Springer International Publishing.

[19] Carla P Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of automated reasoning*, 24(1-2):67–100, 2000.

[20] Carla P Gomes, Bart Selman, Henry Kautz, et al. Boosting combinatorial search through randomization. *AAAI/IAAI*, 98:431–437, 1998.

[21] Carla P Gomes, Bart Selman, Ken McAloon, and Carol Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *AIPS*, pages 208–213, 1998.

[22] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In *IJCAI*, 1979.

[23] Philippe Laborie. Objective landscapes for constraint programming. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 387–402. Springer, 2018.

[24] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Recording and minimizing nogoods from restarts. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:147–167, 2007.

[25] Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. Reasoning from last conflict (s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.

[26] Hongbo Li and Zhanshan Li. A novel strategy of combining variable ordering heuristics for constraint satisfaction problems. *IEEE Access*, 6:42750–42756, 2018.

[27] John DC Little, Katta G Murty, Dura W Sweeney, and Caroline Karel. An algorithm for the traveling salesman problem. *Operations research*, 11(6):972–989, 1963.

[28] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

[29] L. Michel and P. Van Hentenryck. Activity-Based Search for Black-Box Contraint-Programming Solvers. *arXiv e-prints*, page arXiv:1105.6314, May 2011.

[30] Steven Minton. Integrating heuristics for constraint satisfaction problems: A case study. In *AAAI*, pages 120–126. Citeseer, 1993.

[31] Bernard Nudel. Consistent-labeling problems and their algorithms: Expected-complexities and theory-based heuristics. *Artificial Intelligence*, 21(1-2):135–178, 1983.

[32] OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[33] Anthony Palmieri and Guillaume Perez. Objective as a feature for robust search strategies. In John Hooker, editor, *Principles and Practice of Constraint Programming*, pages 328–344, Cham, 2018. Springer International Publishing.

[34] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *International conference on theory and applications of satisfiability testing*, pages 294–299. Springer, 2007.

[35] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017.

[36] Charles Prud'homme, Xavier Lorca, and Narendra Jussien. Explanation-based large neighborhood search. *Constraints*, 19(4):339–379, 2014.

[37] Antonio Ramos, Peter Van Der Tak, and Marijn JH Heule. Between restarts and backjumps. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 216–229. Springer, 2011.

[38] Philippe Refalo. Impact-based search strategies for constraint programming. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 557–571, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[39] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI*, volume 94, pages 362–367, 1994.

[40] Jean-Charles Régin. *Global Constraints and Filtering Algorithms*, pages 89–135. Springer US, Boston, MA, 2004.

[41] Igor Rivin, Ilan Vardi, and Paul Zimmermann. The n-queens problem. *The American Mathematical Monthly*, 101(7):629–639, 1994.

[42] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[43] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[44] Norman Sadeh and Mark S Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial intelligence*, 86(1):1–41, 1996.

[45] Thomas Schiex and Gérard Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal on Artificial Intelligence Tools*, 3(02):187–207, 1994.

[46] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming — CP98*, pages 417–431, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[47] Barbara M Smith. Modelling for constraint programming. *Lecture notes for the first international summer school on constraint programming*, 2005.

[48] Barbara M Smith and Stuart A Grant. Trying harder to fail first. *Research Report Series-University of Leeds School of Computer Studies Lu Scs Rr*, 1997.

[49] Willem-Jan Van Hoeve. The alldifferent constraint: A survey. *arXiv preprint cs/0105015*, 2001.

[50] Julien Vion. Concrete 3.9. 2: A csp solving software & api.

[51] Toby Walsh et al. Search in a small world. In *Ijcai*, volume 99, pages 1172–1177. Citeseer, 1999.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**