Learning While Searching - A Framework for Solving Constraint Satisfaction Problems

by

Christoph Walters

Thesis
Submitted by Christoph Walters
in partial fulfillment of the Requirements for the Degree of
Bachelor of Computer Science with Honours (1608)

Supervisor: Prof. Mark Wallace

Clayton School of Information Technology
Monash University
November, 2005
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vi</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>viii</td>
</tr>
<tr>
<td>Abstract</td>
<td>1</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Constraint Satisfaction Problems</td>
<td>3</td>
</tr>
<tr>
<td>1.1.1 Description of CSPs</td>
<td>3</td>
</tr>
<tr>
<td>1.1.2 Applications</td>
<td>4</td>
</tr>
<tr>
<td>1.1.3 Solving Techniques</td>
<td>4</td>
</tr>
<tr>
<td>1.1.4 Structuring Search Algorithms</td>
<td>5</td>
</tr>
<tr>
<td>1.2 Project Outline</td>
<td>5</td>
</tr>
<tr>
<td>1.2.1 Aims and Methods</td>
<td>5</td>
</tr>
<tr>
<td>1.2.2 Thesis Structure</td>
<td>6</td>
</tr>
<tr>
<td>2 Research Background</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Definition and limitations</td>
<td>7</td>
</tr>
<tr>
<td>2.1.1 Formal Model</td>
<td>7</td>
</tr>
<tr>
<td>2.1.2 Common assumptions and limitations</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Solving Constraint Satisfaction Problems</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Applicability of the CSP model revisited</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Enhancements of standard backtracking search</td>
<td>11</td>
</tr>
<tr>
<td>2.4.1 Constraint Propagation</td>
<td>11</td>
</tr>
<tr>
<td>2.4.2 Conflict Handling</td>
<td>12</td>
</tr>
<tr>
<td>2.4.3 Learning</td>
<td>13</td>
</tr>
<tr>
<td>2.4.4 Exploration</td>
<td>14</td>
</tr>
<tr>
<td>2.4.5 Summary</td>
<td>14</td>
</tr>
<tr>
<td>2.5 Structuring Search</td>
<td>14</td>
</tr>
<tr>
<td>2.5.1 The PLM model</td>
<td>15</td>
</tr>
<tr>
<td>2.5.2 A Framework for Scheduling</td>
<td>15</td>
</tr>
<tr>
<td>2.5.3 A Language for Search Algorithms</td>
<td>16</td>
</tr>
<tr>
<td>2.6 Summary</td>
<td>16</td>
</tr>
<tr>
<td>3 Framework Architecture</td>
<td>19</td>
</tr>
<tr>
<td>3.1 Structural Overview</td>
<td>19</td>
</tr>
<tr>
<td>3.1.1 Module Hierarchy</td>
<td>19</td>
</tr>
<tr>
<td>3.1.2 Main</td>
<td>20</td>
</tr>
<tr>
<td>3.1.3 Framework</td>
<td>21</td>
</tr>
<tr>
<td>3.1.4 Representation</td>
<td>22</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Overhead test results</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>Job Shop Scheduling Results</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>Comparing Configurations A,B</td>
<td>42</td>
</tr>
<tr>
<td>4.4</td>
<td>Comparing Configurations C</td>
<td>43</td>
</tr>
</tbody>
</table>
List of Figures

3.1 General structure and dependencies of the framework . . . . . . . . . . . . 20
3.2 Possible alternative framework structure . . . . . . . . . . . . . . . . . . . 24
4.1 Configuration Encoding . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
A.1 Problem #1: 5 Variables 20 Values . . . . . . . . . . . . . . . . . . . . . . 54
A.2 Problem #2: 6 Variables 40 Values . . . . . . . . . . . . . . . . . . . . . . 54
A.3 Problem #3: 10 Variables 10 Values . . . . . . . . . . . . . . . . . . . . . . 54
A.4 Problem #4: 15 Variables 12 Values . . . . . . . . . . . . . . . . . . . . . . 55
A.5 Problem #5: 18 Variables 2 Values . . . . . . . . . . . . . . . . . . . . . . 55
A.6 Problem #6: 18 Variables 4 Values . . . . . . . . . . . . . . . . . . . . . . 55
A.7 Problem #7: 18 Variables 8 Values . . . . . . . . . . . . . . . . . . . . . . . 56
A.8 Problem #8: 20 Variables 2 Values . . . . . . . . . . . . . . . . . . . . . . 56
A.9 Problem #9: 22 Variables 5 Values . . . . . . . . . . . . . . . . . . . . . . . 56
A.10 Problem #1: 5 Variables 20 Values #2: 6 Variables 40 Values . . . . . . . 57
A.11 Problem #3: 10 Variables 10 Values #4: 15 Variables 12 Values . . . . . . . 57
A.12 Problem #5: 18 Variables 2 Values #6: 18 Variables 4 Values . . . . . . . 57
A.13 Problem #7: 18 Variables 8 Values #8: 20 Variables 2 Values . . . . . . . 58
A.14 Problem #9: 22 Variables 5 Values . . . . . . . . . . . . . . . . . . . . . . 58
Learning While Searching - A Framework for Solving Constraint Satisfaction Problems

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Christoph Walters
October 31, 2005
Acknowledgments

I would like to thank my supervisor, Mark Wallace, for the substantial and friendly support he gave me during the project, for pointing out interesting aspects of the project to investigate further and most of all for giving me new ideas on how to tackle a particular problem when I could not see a solution.

I also want to thank the other Honours students for their help and support, especially Martin who often provided me with feedback.

Christoph Walters

Monash University
November 2005
Abstract

Constraint Satisfaction Problems (CSPs) are a large class of problems with applications in job-allocation, scheduling, edge-detection, finding euler lines in graphs and many other areas. Since CSPs are NP-complete and therefore difficult to solve, a variety of techniques has been developed to improve algorithms based on searching. The vast amount of possible combinations makes a structured approach necessary so that different configurations can be compared.

In this project, a framework which modularizes searching is developed, allowing users to freely combine a number of techniques and evaluate the performance of the resulting algorithms with minimal effort. In addition, new techniques can be integrated into the open structure thus easing future research on search algorithms. Results show that a large number of modern techniques can be matched to the developed structure. The overhead caused by the framework is shown to be within reasonable bounds for almost all problems and especially low for real world problems.
Chapter 1

Introduction

1.1 Constraint Satisfaction Problems

1.1.1 Description of CSPs

Constraint Labeling Problems (CLPs) are one of many problem classes in Computer Science. A large number of real world problems as well as many tasks encountered within the discipline of Computer Science itself can be represented in this formalism. A representation consists of some variables with associated domains and constraints between these variables and a solution is an instantiation of all variables which satisfies all constraints.

When solving CLPs, one can ask for different kinds of answers. Depending on the question, all solutions, any single solution, an optimal solution given some criteria, the number of solutions or just knowledge about whether the problem has any solution at all may be required. Note that some answers immediately answer others of these questions as well. Hence, the two most interesting problems are Constraint Satisfaction Problems (CSPs) and Constraint Optimisation Problems (COPs). A CSP requires an answer about whether a solution exists whereas in COPs an optimal one or approximately optimal is needed. Efficient approaches to solving problems of these two types are significantly different, with COPs obviously being the harder problems. Since CSPs are combinatorial problems, which implies that it usually takes a long time to solve them, it is important to find efficient algorithms for particular problem instances in order to find solutions within reasonable time limits. The project will concentrate on Constraint Satisfaction Problems.

There are several ways to solve Constraint Satisfaction Problems. Two important strategies are based on classic deterministic search algorithms. The first one is to generate a (random) assignment of values to variables and to repair this assignment if any constraints are violated until a feasible solution which satisfies all constraints has been found. Such an approach is called a repair-based method.

The second approach is to start with an empty assignment and successively assign values to the variables, that is to extend any partial solution to a complete one. This is called a constructive method. The most basic way to solve a CSP in this manner is to use a backtracking algorithm which searches the whole search space in a - for instance - depth-first manner. The resulting algorithm is complete and sound but unfortunately it takes exponential time with the problem size. Therefore, this method is not suitable for other problems than the most basic ones.

Other methods include a number of stochastic algorithms like genetic algorithms or simulated annealing which deviate greatly from the approach taken in this project and are therefore not discussed further. The method examined here is constructive search.
1.1.2 Applications

A typical real world problem that can easily be represented as a CLP is rostering, where personnel needs to be assigned to different tasks. Constraints specify that one person can only do one task at a time and that some workers are qualified to do only some of the tasks. Problems include edge-detection in images or graph-colouring, apart from many other applications. Scheduling problems can be treated as CSPs as well. Variables are - for example - start times of tasks to be done and constraints ensure that no two tasks which require the same resource are executed at the same time. Usually scheduling problems are optimisation problems with the goal to find a feasible schedule using which all tasks are done in a minimum amount of time. By constraining the time of completion of all tasks, however, the problem can be treated as a CSP as well.

One example of an application in a more technical background is scene matching. If in a sequence of images, objects are to be mapped on one another so that a system determines which object is the same as an object in a previous frame, the problem can be treated as a CSP. Variables are objects and their values are objects known from previous scenes and some new objects. Constraints could specify that an object cannot disappear without moving out of the image\(^1\) or that an object which has moved little is likely to be the same object as the one at a similar position in the previous frame. A large number of other problems can be expressed as a CSP quite naturally.

1.1.3 Solving Techniques

In the domain of Constraint Satisfaction Problems, a variety of improvements can be made due to the nature of the problem type. These do not change the fact that solving CSPs remains an NP-complete problem but in most cases the application of sophisticated enhancements of backtracking speeds up the search by several orders of magnitude.

Choosing a certain order of variables is one of the aspects of search which can be improved. If “critical” variables are assigned a value early then a possible mistake at this stage will be detected sooner and less work has to be undone by backtracking. Changing the order in which variables are chosen helps the search algorithm explore the “bottlenecks” in the search space first.

Constraint Propagation, which is the process of deducing further constraints from the already explicit ones in order to prune the search space, is another widely used technique. While the ordering of variables is a heuristic, constraint propagation is a deductive method. The additional constraints computed by the propagation algorithm transform the given CSP into an equivalent one which is in general easier to solve by eliminating parts of the search space that cannot contain a solution (pruning).

Once a search algorithm gets to a dead-end, a situation in which any instantiation of one unassigned variable results in a constraint violation, there are several ways how to react. Undoing the last assignment is the simplest method, but often it is useful to undo all assignments up to one variable “responsible” for the conflict.

The algorithm can “learn” something about the problem while searching when encountering dead-ends for example. It is possible to deduce further constraints on the problem similar to what constraint propagation does and these additional constraints can be used to prune the search tree further. The general idea is to record information and use it to prevent an algorithm from rediscovering the same relations over and over again.

\(^1\)This is of course a simplification if one thinks of exploding balloons for instance
1.1.4 Structuring Search Algorithms

Most of the techniques to enhance searching algorithms can be combined since they affect different and independent aspects of searching. In some cases, combinations result in even better search algorithms, sometimes the resulting algorithm is very fast for a limited subclass of CSPs only. Some combinations however, might yield a slower search algorithm, because the overhead of all techniques is greater than the benefits they achieve together.

It becomes evident that for experimental evaluation of different search algorithms, a great deal of work is required to implement all different combinations of techniques. This not only takes very much time but it is also error-prone.

This problem can be overcome in several ways. It is possible to examine search algorithms on a purely theoretical base. This method does not involve any implementation and thus eliminates much effort. However, modelling all techniques and especially their combinations is likely to be extremely challenging and possibly as prone to errors as the straight-forward implementation. In addition, theoretical results are based on assumptions about the distribution of problem properties which might not accurately model problems encountered in practical applications. Results gained by theoretical examination would most probably be either purely qualitative, which is not satisfactory, would include incredibly complicated formulas or would be based on very simplifying assumptions. For all these reasons, this approach is not examined in the current project.

A solution to facilitate experimental evaluation is to divide searching into several mostly independent modules and to fit the enhancement techniques into the schema. A good schema has few interdependencies and simple interfaces between the modules while its structure allows researchers to fit any enhancement technique into one of the modules without impact on other modules or their interfaces.

Some research has already been done on frameworks which partition searching according to the aforementioned goals. However, most of the work done so far remains theoretical; the ideas have not been implemented and tested yet. Other concepts have been implemented but the scope of those frameworks is rather specific. A framework for solving all kinds of CSPs and at the same time providing structure to include all improvement techniques does not seem to exist yet. In particular, the existing frameworks do not take into account the possibilities which arise from using learning techniques.

1.2 Project Outline

1.2.1 Aims and Methods

In the current project, a partition of constructive search is presented and the implementing framework is explained. The chosen structure yields a high degree of flexibility while a number of known techniques to improve the efficiency of searching algorithms is implemented within the framework. The platform for implementation is the ECLiPS system which is a constraint programming system based on the PROLOG-language.

We partition a searching algorithm into a number of phases/modules and subphases. The first phase includes choosing the next variable to assign a value to and choosing a value out of its domain. If the instantiation does not violate any constraints, then the next variable is assigned recursively. If a dead-end is encountered, conflict handling is used to deal with the problem. After conflict handling, at least one assignment is retracted so that the search can continue from a previous consistent state.

Conflict handling can include further subphases: Computing an explanation for the failure might help in determining which variables have to be retracted. The explanations can also be used in another subphase for learning. Results from learning are stored and used to perform additional consistency checks after a variable assignment.
Propagation works in the background mainly. The ECLiPSe system includes an implementation of a few propagation algorithms with different propagation strength. The methods provided automatically detect conflicts after variable assignments and prune - that is reduce - domains of variables otherwise.

Most of the techniques require some background data and create other overhead. A problem which can arise from a high degree of flexibility is that performance can be lost for features which are not even activated in a certain configuration. If neither conflict handling nor learning takes place, for example, then there is no need to compute explanations and thus there is no need to incur any overhead connected to computation of explanations. For this reason, the framework has an initialisation phase in which an appropriate setup depending on the chosen configuration is determined, thus eliminating a major part of the overhead.

To evaluate the performance of the framework, it is tested against a simple backtracking algorithm on a number of problems. These problems are mostly randomly generated using the “Uniform Random Binary CSP Generator” but some job shop scheduling problems are applied as well. Random problems can be easily generated to produce a significant sample set, whereas real world problem like job shop scheduling give a more realistic picture of the performance in practical applications. Of course, for these tests the framework is configured to perform simple backtracking search as well.

1.2.2 Thesis Structure

The following chapter describes previous research and gives more detail on special techniques to improve searching. In the third chapter, the proposed framework is explained in detail, including both the overall system architecture and the interfaces between different modules. To illustrate the concept, the implementation of some techniques is shown. The results of tests comparing the framework against simple backtracking search are reported in the chapter four, followed by an evaluation, discussion and conclusion in chapter five. The appendices contain more detailed listings of results.
Chapter 2

Research Background

This chapter provides an overview of the previous research on Constraint Satisfaction Problems and techniques used to solve them more efficiently. Detailed information on the CSP model itself and its practical application is included. The following sections will present a closer look at strategies and algorithms for solving CSPs. In section one, a definition of the CSP model and its limitations are pointed out. Section 2 concentrates on general methods and a precise description of the search process. A closer look at the role of CSP modelling and its relation to the problems being modelled is presented in section 3. In section 4, a number of techniques that can expedite the search is discussed. The focus of section 5 is on how the search process can be divided into several independent modules and why such a division is desirable. Finally, section 6 will summarize the the overview.

2.1 Definition and limitations

2.1.1 Formal Model

First of all, a formal definition of Constraint Satisfaction Problems is given, which includes definitions of a solution and intermediate steps during the search process. A similar definition can be found in the work by Petit et al. (2003) although definitions in other work usually only differ in terms of the level of detail.

Definition 1 (CSP) A Constraint Satisfaction Problem is given by a set of \( n \) variables \( X = \{x_1, \ldots, x_n\} \) where each variable \( x_i \) has an associated finite domain \( D(x_i) \), and a set of \( m \) constraints \( C = \{C_1, \ldots, C_m\} \) where each \( C_i \in C \) constrains an \( r \)-tuple of variables \( Sc(C_i) = (x_{i_1}, \ldots, x_{i_r}) \) called the scope of \( C_i \). \( C_i \) can then be defined by its scope and the set of possible value combinations that are valid: \( C_i \subset D(x_{i_1}) \times \cdots \times D(x_{i_r}) \). \( r \) is called the arity of \( C_i \) (Nudel, 1983).

This definition includes all information necessary to represent any particular CSP. However, to better exploit the features of certain constraints, often it is not advisable to keep an explicit enumeration of all valid value combinations. The numerical constraint \( X < Y \), for instance, can be easily and comprehensively stored in the way just mentioned. An explicit enumeration of valid value combination of this constraint is extremely cumbersome and inefficient. For that reason, constraints can be regarded and processed in a more general sense by defining a constraint as some predicate over the variables in its scope.

To measure the performance of algorithms on different classes of CSPs, it is important to formally characterize these different classes. A particular problem can be characterized by the exact number of variables and constraints. A class of problems itself is better defined by the tightness of constraints (Nudel, 1983) which refers to the amount of possible value
combinations that a constraint permits compared to the amount of possible combinations in total. Another important factor is the size of the largest domain for any variable (Nudel, 1983). Dechter (2003) describes the characteristics of special CSPs for a number of enhancement techniques and how these can be exploited to achieve better worst-case bounds for the algorithms than for general problems.

**Definition 2 (Solution)** A solution to a CSP is a total function \( s : \mathcal{X} \rightarrow \bigcup_{i \in \{1, \ldots, n\}} D(x_i) \) mapping all variables in \( \mathcal{X} \) to values in their respective domains which at the same time satisfies all constraints. A constraint \( C_i \) is satisfied by a function \( s \) iff \( s(Sc(C_i)) \in C_i \), otherwise it is violated.

Analogously, a constraint - seen from the broader perspective as an arbitrary predicate - is satisfied by a function \( s \) iff its predicate is true under the instantiation of \( s \).

Since the main focus is on constructive search algorithms, an intermediate state of search has to be defined. It is important to note that for each CSP, a number of constraints is known explicitly. However, a number of other constraints that are a logical consequence of the known constraints might not yet be discovered in a state.

**Definition 3 (Partial solution)** A partial solution to a CSP is a partial function \( s \) from \( \mathcal{X} \) to values in the respective domains \( D(x_i) \) so that all explicitly known constraints without any uninstantiated/unassigned variable in their scope are satisfied.

Which variable assignments are partial solutions depends on several factors. First of all, the problem instance is obviously relevant. Next, the propagation algorithm, which can infer additional constraints from the explicit ones, determines when a future failure is detected. If some learning technique is used, a result of previous learning can be that a current assignment is inconsistent and cannot lead to a solution. A partial solution is therefore a partial valuation of the variables for which no inconsistency can be detected by the searching algorithm including all special techniques.

In an implementation of a search algorithm to solve CSPs, the intermediate states usually includes further information such as additional inferred constraints and other data gathered by different features of the algorithm. To give one example, a propagation algorithm maintains a list of the explicitly known constraints at any point during the search. As the state information depends on the special techniques used, this matter is not formalized any further at this point.

An intermediate state in a particular search defined by a partial solution and other data is also called consistent. Consistency of a state does not imply that the corresponding partial solution can be extended to a complete solution. In the simplest form of a backtracking algorithm where constraints are only checked when all variables are assigned, all states including unassigned variables are consistent. On the other hand, the starting state (with an empty assignment) is only inconsistent if no solution to the problem exists and if the algorithm used can determine this without expanding the search tree at all.

### 2.1.2 Common assumptions and limitations

In current research, there are some assumptions that limit the expressiveness of the CSP formalism in order to make the problems at least decidable (Maher, 2003; Wallace, 2000). These are:

- There is a finite number of variables and constraints
- Domains of variables are finite
2.2. SOLVING CONSTRAINT SATISFACTION PROBLEMS

Without these assumptions, the CSP formalism could cover an even greater amount of problems, but in general it would not be possible to decide whether a solution exists to a given problem. In addition to the aforementioned limits, there are some assumptions that do not limit expressiveness and are not used in all formalisations encountered in the literature:

- constraints are binary, that is they only restrict two variables at a time
- constraints are expressed explicitly, e.g. by enumerating all possible value combinations
- for each subset of all variables, there is one constraint at the most

In the current project, the first two limitations are imposed, which means that domains of variables and the number of constraints are finite. The framework is independent of all other simplifying assumptions.

2.2 Solving Constraint Satisfaction Problems

As indicated before, constructive approaches to solving CSPs work by assigning values to variables one by one trying to extend the current partial solution to a complete one. Information whether extending the current partial solution will lead to a solution is of course quite valuable. Complete knowledge would effectively reduce the cost for solving a CSP to constant time, as a solution could be returned immediately. Clearly, gaining this knowledge is quite difficult, and as solving CSPs is NP-complete, it takes exponential time in the same way as simply trying all possible assignments. Depending on the problem type, an appropriate trade-off between cost of knowledge for guiding the search and cost of searching possibly fruitless parts of the search space is required.

Usually in each step, another variable is instantiated. The new state can be used to infer new knowledge about the rest of the search space. After some reasoning process used to infer new constraints or detect 'impossible' values for some variables, that is values that definitely lead to the violation of some constraint later on, another variable assignment can be made based on this new information.

It is useful in some cases to consider a more general case than assigning one value to a variable. Instead, each assignment is regarded as a new constraint of the form \( x_i = v \). The assignment has been formalized this way several times in the literature (Jussien and Lhomme, 2003; Maher, 2003). Beck and Fox (1998) point out that instead of assigning one value to a variable, it might be more useful to split the domain of that variable and invent a new constraint of the form \( x_i \in \{v_1, \ldots, v_l\} \). This is because in large domains - especially when the domain consists of numbers - certain values might not make much difference with regard to the constraints. If a constraint like \( X > v \) with some value \( v \) is violated by the assignment \( X = 1 \), trying \( X = 2 \) is likely to violate the constraint again. A better way could be to impose an additional constraint as \( X > 20 \) tentatively and if this fails, try \( X < 20 \) and leave a definite instantiation for later. Splitting domains of variables can be a good way of reducing the complexity of the remaining problem without restricting the possibilities too much.

An important point to be addressed is how algorithms deal with the situation when they encounter a dead-end, that is an inconsistent state. In this case, some variable assignments need to be retracted and the search must be continued at a different point in the search tree. It is usually an inefficient strategy just to reset the (chronologically) last assigned variable. This is because the reason for the inconsistency might be decisions made way before. Therefore, different strategies can be applied to decide how many and which assignments are to be retracted.
Failure can also be used to learn properties of the problem instance at hand (Junker, 2001). If the reason for failure is recorded, it is possible to avoid the same situation later on. The effectiveness of this method depends heavily on how concise the explanation is. When using a tree search algorithm such as backtracking, it is clear that the very same assignment of all variables will never occur again. However, if the explanation only contains a - preferably small - subset of the assigned variables, then a similar situation might very well reoccur many times in future branches.

### 2.3 Applicability of the CSP model revisited

One of the greatest strengths of the CSP model is that a large number of different problems can be mapped into the CSP formalism while the resulting model remains intuitive. Intuitive models are quite important if communication between the user and the designer of some system is needed - a requirement that applies to most problems that occur in real world applications. In particular, constraints should model the true interrelations of variables as accurately as possible and should be able to be processed efficiently at the same time.

However, the CSP formalism has some flaws which have been criticised recently. Maher (2003) describes complications in modelling constraints that naturally occur in every day problems. The basic problem he sees is that, in the general case, values for variables are treated as uninterpreted constants. In real world problems, however, domains for variables are often numbers or other values that can be interpreted and related to one another. Another problem is that numbers are a common domain for most applications but domains in the CSP model have to be finite. That means that the designer of a system has to impose artificial boundaries on some variables. Common constraints like \textit{alldifferent} or \textit{cumulative} can only be represented with difficulty in the standard model. For these reasons, Maher (2003) proposes an extension to the CSP formalism and explains how different notions of consistency can be formalized in the new model. He argues convincingly that for actual implementations of CSP solving algorithms, the general model has to be extended.

Wallace (2000) criticises the standard formalism from another perspective. Domains can be seen as unary constraints on the variables. However, they are treated in a special way because the algorithms can work more efficiently if the domains are given and do not have to be evaluated together with all other constraints. Still, there are also reasons for underlining the role of different types of binary or k-ary constraints because a special treatment for them would again improve the performance on the implementation level. Linear constraints which model a relation like $X > Y + 2$ are an example for other such constraints. Wallace (2000) persuasively argues that the line drawn between special constraints (the domains) and “other” constraints seems arbitrarily chosen. In addition, the CSP model requires a fixed number of variables for a problem. In some applications however, the actual number of needed variables cannot be known in advance and it even varies when different paths through the search space are explored. The only way to deal with this problem in the general CSP model is to set the number of variables to be the maximum number that is possibly needed. When problems are solved this way, many variables will prove useless in the end and their existence will only encumber the solving algorithm. As evidence, Wallace (2000) gives an example of a CSP where a fixed number of variables “conflicts with the model supporting the best search strategy”.

The arguments given by the two authors indicate that the extended CLP is a more suitable modelling formalism for real world problems than CSP. However, even though the CSP formalism is not suitable for all practical problems, a number of problems remains that can still be adequately represented without extensions. In particular, problems of
2.4 Enhancements of standard backtracking search

In this section, methods for improving efficiency in search will be discussed in greater detail. The most important concept is Constraint Propagation which forms the base most techniques build on. After introducing this concept, the most common strategies and their relation to constraint propagation will be explained.

2.4.1 Constraint Propagation

Constraint Propagation (CP) is a reasoning process which given some constraints infers new ones. A variety of propagation methods to perform this reasoning with different amounts of effort invested is available. One aim is to identify values for variables which cannot be part of a solution, another is to detect more dependencies between several variables. By doing this the search space being explored can be reduced significantly.

Before general algorithms that do CP are described, the idea will be illustrated with an example.

Imagine that the following CSP is given:

\[ X = (X, Y, Z), \]
\[ D(X) = D(Y) = D(Z) = \{1, 2, 3, 4, 5\}, \]
\[ C = \{C_1, C_2\} \text{ and } Sc(C_1) = \{X, Z\}, Sc(C_2) = \{Y, Z\} \] with
\[ C_1 = X < Z \text{ and } C_2 = Z < Y. \]

A constructive search algorithm could assign \( X := 1 \) and \( Y := 1 \) as a first step but when trying to find a value for \( Z \) afterwards, the algorithm fails since there is no value for \( Z \) which is smaller than \( Y \) and in the domain of \( Z \). Using simple backtracking, the next variable for \( Y \) could be 2 which will lead to a dead-end again.

On the other hand, looking at both constraints, one can easily see that the constraint \( C_3 := \{(X, Y) | X < Y\} \) is a logical consequence of the first two ones. Using this new constraint, the first dead-end would have been avoided from the start. In larger problems, a significant amount of searching can be saved by considering constraints that can be inferred. Furthermore, after assigning a value \( v \) to \( X \), even more constraints can be inferred using the new knowledge. Remember that each assignment can be regarded as a new constraint. \( X = v \land C_1 \) implies that all values less or equal \( v \) are now invalid values for \( Z \). This can be treated as a domain reduction for \( Z \). For instance, \( X = 3 \land C_1 \) implies that \( D(Z) \) can be reduced to \( \{4, 5\} \).

There is a variety of algorithms which perform constraint propagation. Dechter (2003) provides a very good overview of these. They can be roughly classified by the amount of propagation they achieve. Arc-consistency is the probably best known level of propagation and several algorithms (AC1, AC2, AC3, ...) exist to guarantee it. Arc-consistency enforcing algorithms only remove invalid values from the domains of variables, they do not return any other new constraints. AC3 in for instance the binary case checks all constraints \( C_i \) between any two variables \( X, Y \) and checks whether any value \( x \in D(X) \) does not have support. Not having support means that the constraint currently examined is violated for all values \( y \in D(Y) \), or in other words: \( \forall y \in D(Y) : (x, y) \notin C_i \). If any
value from $D(X)$ is removed by this, all constraints $C_j$ concerning $X$, that is $X \in Sc(C_j)$ will be checked again, until either a domain of any variable is empty or no constraints are left for checking any more. Of course, an empty domain indicates an inconsistent state.

The process of finding and eliminating inconsistent values from variable domains is called **Look-Ahead**. The simplest way to do so is called **Forward Checking (FC)**, which is yet another and very basic form of Constraint Propagation. A FC algorithm checks whether for each uninstantiated variable $X$ and for each value in its domain $x \in D(X)$ all constraints permit this value in combination with some other variables. If some value is not allowed by all constraints concerning the according variable, then this value is removed from its domain. FC does not check whether the possible values for other variables are consistent among each other. Therefore, FC is strictly weaker than arc-consistency enforcing algorithms. In the same way, any other propagation method can be used to eliminate inconsistent values for future variables and to detect inconsistencies caused by earlier decisions faster.

Other levels of consistency/propagation are **path-consistency** or, even more generally, **k-consistency** (Dechter, 2003) and they achieve more propagation but have a higher complexity. The reasoning methods discussed (FC, arc-consistency, path consistency and k-consistency) are mostly based on finding possible value combinations out of the domains of the variables. A large number of different reasoning techniques can be applied as well, using the full range of logic. The more propagation is done, the earlier any inconsistency in the current partial solution can be detected. Dechter (2003) states that the amount of effort put into propagation that still pays off in the end increases with the overall problem difficulty. She proposes that typical problems for testing CSP solving algorithms are usually too simple to make path-consistency or even higher levels of propagation worthwhile. This is because the testing problems are usually rather simple ones compared to problems one can find in real world applications. She claims that such - more difficult - problems could benefit from these types of propagation.

### 2.4.2 Conflict Handling

**Look-Back** is another method to enhance standard backtracking search. Look-Back methods are applied when an algorithm encounters a dead end. Remember that a dead end occurs in a situation when some domain becomes empty or, in other words, when every instantiation of a variable results in an inconsistent state. Simple backtracking retracts the chronologically last assignment for variable $X$ and tries a different value for the variable before ($Y$). However, it is possible that the inconsistency is not at all influenced by the value of $Y$ and changing its value only leads to the very same dead end. Often it is worthwhile to find the actual reason or an **explanation** for the inconsistency and act on it. **Backjumping** retracts all assignments up to the first (that is chronologically last assigned) variable $Z$ which is in the scope of any constraint that contains $X$ as well (Ginsberg, 1993). However, a constraint with these two variables in its scope does not necessarily have to be one that is violated and therefore this method can be extended to **conflict-directed backjumping (CBJ)**. For CBJ, explanations for the inconsistency are computed. Explanations are sets of constraints that by themselves are inconsistent. An explanation can contain constraints given by the problem definition as well as decisions made during the search so far. For example, an explanation can be formally written as:

$$\neg (C_1 \land C_4 \land C_5 \land X_2 = b_2 \land X_3 = a_3)$$

meaning that the instantiations of $X_2$ and $X_3$ are in conflict with the constraints $C_1, C_4, C_5$ given by the problem definition. For solving CSPs, the problem constraints can be seen as the **background** for the explanation since these constraints are static and can not be changed or retracted to solve the problem. There are applications where it is useful to inform a user about the reason for a complete failure - if the CSP has no solution at
2.4. ENHANCEMENTS OF STANDARD BACKTRACKING SEARCH

all - and this could be used to debug a system or relax a problem accordingly so that a solution can be found for the new problem. This however, is not the scope of the current research. Thus, the variable instantiations are considered with the background of the problem constraints to compute explanations.

It is easy to see that the set of all decisions and all constraints in the problem together must be an explanation for the dead end (Jiang et al., 1996). However, this explanation is not very helpful because it does not yield any information on which variable assignments caused the conflict. Most probably a subset of this full explanation is an explanation as well. The less elements an explanation contains the more information can be gathered from it. For this reason, we usually want to find minimal explanations, that is explanations where any subset is not an explanation any more. Junker (2001) presents an algorithm which can compute explanations faster than previous ones by recursively splitting the set of constraints and skipping subsets that do not contain a conflict. Though this definitely is an improvement, the overall effect of this has to be evaluated. Propagation methods are necessary to find concise explanations because they uncover the interdependencies between several constraints (Junker, 2001).

It has to be decided whether in a dead end situation all minimal explanations or just one is computed. Knowing all minimal explanations might enable us to decide on which one to act so that the savings are maximized. de la Banda et al. (2003) propose algorithms to compute all minimal subsets efficiently. The authors motivate the idea of using multiple explanations, illustrate the problem with examples and propose improvements to their basic method step by step.

2.4.3 Learning

A particular benefit of minimal explanations is that they often do not contain all decisions made so far. As a result, even after changing values for some variables, a similar situation can occur again. If the explanation has been recorded - for example as a new constraint forbidding this partial assignment - a future dead end can be detected at once. Such recorded constraints are called no-goods (Jiang et al., 1996) and can be used globally in the search process. This extends the role of explanations for backjumping fundamentally. After an explanation for a conflict has been learnt, it is not necessary anymore to explore the search tree until the conflict caused by the same partial assignment actually occurs again. Instead, the algorithm can stop searching and initiate backtracking as soon as it becomes apparent that the same partial assignment will be tried again. Thus, explanations serve two quite different purposes in searching algorithms. The downside of recording explanations is that storing all of them leads to exponential space requirement (Douence and Jussien, 2002).

One way to reduce the space consumption is to do a subsumption check (Ginsberg, 1993). If for instance a no-good 1: \( \neg(X_2 = b \land X_3 = b) \) and another no-good 2: \( \neg(X_1 = d \land X_2 = b \land X_3 = b) \) are recorded the second no-good can be discarded without loss of information since the first no-good subsumes it. In every instance when no-good 2 is violated, no-good 1 is violated as well. Using such a subsumption check, the space complexity can be reduced, though probably it remains exponential.

Forgetting explanations is essential if the space complexity is to remain polynomial. Finding a good forgetting algorithm is fairly difficult because in principle, all no-goods are relevant. Relevant means that a situation in which a no-good is violated could occur again during the search. As a result, a careful balance has to be found between keeping possibly still relevant no-goods and deleting others which are unlikely to yield great benefits in future.
2.4.4 Exploration

Exploration for solving CSPs consists of two parts: Firstly, choosing the next variable which will be instantiated or at least constrained and secondly, choosing a value for this variable from its domain or choosing the subset of its domain that the variable will be constrained to. In standard backtracking, an arbitrary unbound variable and an arbitrary value out of its domain is chosen. It is not possible to find a strategy for these choices which is better for every single problem instance, but some guidelines for educated guesses can be devised which speed up the search in some of the cases. These techniques are fundamentally different from the aforementioned ones since they are based on heuristics and not on deduction. Heuristics are not guaranteed to lead the searching algorithm down the right path all the time, but on the other hand, time can be saved in many cases when the heuristics proposed the right choice while deductive methods are unable to provide any information.

Two heuristics for choosing a variable are first-fail and most-constrained Brélaz (1979). In first-fail, a variable with the smallest domain is chosen. The reason behind this is that a variable with a small domain yields less flexibility than one with a large domain. Of course this is not necessarily true, depending on the problem instance, but still it improves the efficiency for certain problem types.

Choosing the most constrained variable is another heuristic which is based on the following reasoning: If a highly constrained variable is assigned very late in the searching process and if every instantiation fails due to one of the many constraints on this variable, then it is likely that a large part of the search done so far needs to be undone. If that variable is instantiated early, however, then a mistake could be detected earlier with less wasted effort. In addition, the instantiation of the highly constrained variable will reduce the size of the sub-search tree which is beneficial for the efficiency again.

2.4.5 Summary

In this section, we have described constraint propagation methods and their applications for look-ahead and look-back techniques as well as for finding no-goods. Heuristics have also been covered briefly. Propagation is the core of any look-ahead algorithm and can also be utilised for look-back during the process of finding explanations.

There are many other methods to improve the efficiency in solving CSPs including stochastic local search, relational consistency or tree decomposition methods which are described in great detail in (Dechter, 2003) and partly concern specific classes of CSPs. Exploring all these techniques in appropriate detail however would exceed the scope of this project.

2.5 Structuring Search

Facing the wide variety of different methods to enhance searching at different stages, it is natural to ask whether several of these techniques - and which ones in particular - can be combined and whether their benefits will accumulate (Jussien and Lhomme, 2003). Furthermore, research has shown that for different classes of problems different configurations of the techniques result in the most efficient search algorithm. To create a model which makes it possible to determine which methods can be combined while preserving completeness and soundness of the overall algorithm, it is necessary to develop an abstract model of searching in CSP. Partitioning the search algorithm into several phases makes it possible - in best case - to assign every technique to one of the phases.

The ultimate goal is to have a search framework which specifies several stages of the search such that every existing technique for enhancing CSP solving algorithms can be
assigned to one of the stages and that all of these stages only need to communicate through a small interface while the overhead is kept minimal. Jussien and Lhomme (2003) and Beck and Fox (1998) have already proposed frameworks that come close to this goal. For some special techniques, it has been emphasized that they can be applied independently of other parts of the search algorithm (Junker, 2001; Jiang et al., 1996; Ginsberg, 1993).

2.5.1 The PLM model

Jussien and Lhomme (2003) distinguish three modules:

The propagation component that “filters”, that is deletes, parts of the search space that cannot contain a solution and “checks” whether the current state of search is still consistent.

The learning component that “records” and “forgets” information so that dead ends can be avoided in future after they were encountered in some context.

The moving component that “extends” the current partial solution by further exploring the search space and can “repair” a state if some inconsistency is detected.

This partition seems quite sensible and useful and successful testing of the framework is reported in the paper as well. However, combining the “extension” and “repairing” operators into one component suggests a stronger connection between these two processes than there is in fact. Also, the chosen partition is slightly too coarse. Recording and forgetting no-goods is not necessarily closely connected to the process of computing the explanations. As argued above, the moving component which does the conflict handling needs the explanations as well. A more detailed partition is likely to give more freedom for optimisation to a user while incurring almost no extra effort or overhead.

2.5.2 A Framework for Scheduling

Beck and Fox (1998) propose a model that is on first glance quite similar to the one we develop. The main components of the framework are “propagators”, “heuristic-commitment techniques” and “retraction techniques”. A heuristic-commitment component adds (decision) constraints to the existing constraints, that is, it instantiates a variable or at least tentatively reduces its domain. Propagators then calculate some of the logical consequences of these decisions in connection with the other explicit constraints. The degree to which these consequences are computed can vary depending on which propagation algorithm is implemented. If dead-ends are encountered, the retraction component is responsible for repairing the state by retracting some decisions made before. Like the other components, the exact way this component has to work is not specified either.

This framework exhibits a partition of the tasks in a searching algorithm that appears more natural and intuitive. However, there are several differences between this framework and the one we develop. This one first of all is not implemented but rather a concept and the authors note themselves that in an implementation, greater interdependencies between the components might occur and as result the overall structure might have to be revised. Beck and Fox (1998) claim that their work supports the understanding of constraint-directed search and accepting this as the main objective, this framework presents a valuable contribution to the field. The project we plan to undertake aims at actually implementing a system that includes and extends the work in question.

Furthermore, the framework by Beck and Fox (1998) has been devised for the particular domain of scheduling problems and - apart from this - does not contain a significant amount of learning and recording of no-goods. Beck and Fox (1998) state that the representation of
constraints is of great importance for efficient processing. While a significant interrelation between representation and solving algorithm cannot be denied, our approach will try to remain independent of any representation both to allow arbitrary problem representations to be used as inputs with little effort and to remain general.

2.5.3 A Language for Search Algorithms

Laburthe and Caseau (1998) propose a different way to consider search algorithms in a formal way. The authors developed the language “SALSA” which is capable of describing a variety of different approaches to searching in a formal and concise way. The scope of their language mainly lies on optimisation techniques although search algorithms for any solution are supported as well. The scope is not limited to constructive search either, but different hybrid approaches can be expressed in SALSA. The language formalises what happens at choice points in a search process. The options range from using a specific technique followed by another - each an arbitrary number of times - to trying whether one technique is applicable and using an alternative if that is not the case. In their paper Laburthe and Caseau (1998) show that several problems (n-queens, GSAT, travelling salesman problem, . . .) can be expressed within a few lines using the SALSA language.

The authors concentrate on expressing techniques for ordering variables and choosing values for them while chronological backtracking seems to be assumed as a basic feature. SALSA allows users to specify hybrid approaches which combine constructive search followed by local optimisation or a limited breadth-first search instead of the standard depth-first search. Direct support of advanced deductive methods such as look-back techniques or learning are not evident, however. On such matters, it might be possible to express the techniques, but the language is too general to support this directly. Such a level of detail, however, is probably not the intention of the authors.

The language itself is not a framework for searching but only provides and encourages a formal view on the search process. For that reason, SALSA is a valuable contribution to the field of structured approaches to searching. The language could be used to describe which search algorithms can be implemented using a given framework and consequently, different frameworks could be compared more easily.

de Givry and Jeannin (2003) presents a system which was partly inspired by the SALSA language and tries to address some of the issues. The authors divide searching into three components: A definition of the complete search tree, a set of conditions for partial exploration and finally a module to combine a number of partial solutions with the goal to find a complete solution. Again, the focus is mainly on different exploration techniques and variable ordering heuristics. Domain splitting for instance is directly supported. The library proposed by de Givry and Jeannin (2003) can be used to implement hybrid search algorithms.

In this section, the need for frameworks for solving constraint satisfaction problems has been explained and two existing ones have been presented. A modelling language and an implementation of a comparable concept have been described briefly.

2.6 Summary

Constraint Satisfaction Problems have a wide range of applications but are in general very hard to solve since they are NP-complete. For this reason, a variety of improvements to standard searching have been developed which reduce the average time complexity and make a larger part of all CSPs tractable. These methods reduce the search space by pruning parts of it that cannot contain solutions or offer heuristics to explore the most promising parts first. Constraint Propagation as a key concept and explanations
for conflicts are the predominant opportunities to achieve the former while a variety of guidelines of different complexity has been developed for the latter.

In order to combine these methods, measure their performance and determine their interrelations, a general approach to structure search is required. While several such frameworks have already been developed, none of them provide control over all of exploration, propagation, learning, conflict handling and explanation, which we believe to be important for full flexibility. Some techniques are not only not customisable in several of the existing frameworks, but are not even supported. For these reasons, a new framework is developed.
Chapter 3

Framework Architecture

The aim of the current research is to design a framework for constructive search in solving Constraint Satisfaction Problems. The architecture of the system is described in this chapter. Section one presents an overview of the modules and how they relate to one another. The interfaces between all modules are discussed in greater detail in section two and section three covers the methods used to evaluate the framework.

3.1 Structural Overview

The framework consists of a number of modules which correspond to categories of improvement techniques and their prerequisites. Examples for these are the conflict handling or propagation modules. These modules can be extended by users by adding a new technique in the respective area. The interfaces of these modules are as simple as possible while giving users a high degree of flexibility for different approaches. These modules will be referred to as user modules in future:

- Exploration
- Propagation
- Conflict Handling
- Explanation
- Learning

Several other modules exist which build the connection between the modules or provide auxiliary functions which are used frequently. The list of these modules is

- Main
- Framework
- Representation
- Problems
- Util

3.1.1 Module Hierarchy

The main module is used to initiate queries in the framework module, load problems from files using the problems module and control a few other options. The framework module performs the setup of a problem, controls the search, manages information generated...
Figure 3.1: The figure shows the modules of the framework and their dependencies. User modules are highlighted by an additional dashed box. An arrow indicates that one module uses another module. Most user modules actively pass information back to the framework. The module “util” is not included in the graph since all modules use it and do so only to look up commonly used functions.

during this process and performs the function calls in other modules. The setup includes an initialisation phase in which the optimal setup for the current configuration is determined automatically, a configuration being the set of predicates chosen for the user modules. Using the initialisation phase, features which are not required in the current configuration can be turned off in order to reduce the overhead. The problem to be solved is read and interpreted using the representation module which translates an external format into the internal one.

The central search procedure contains the important references to the user modules. The framework asks the exploration module to choose a variable and a value for it. After an instantiation, the propagation algorithm is engaged automatically using the features of the ECL/PS\textsuperscript*e} language. If no conflict is detected in the propagation stage, the results of the learning component - which are stored inside the framework data - are queried. If no inconsistency is detected on that level either, the framework calls the search procedure recursively. If an inconsistency is detected at some stage or if the recursive call of the search predicate fails, conflict handling is engaged by the framework.

In the current implementation, the conflict handling can try to find an explanation for the failure using the explanation module. After the explanation is retrieved, the learning module is activated by the conflict handling algorithm to process the explanation and update the information stored in the framework module. Finally, the conflict handling algorithm determines a set of variables to be retracted by the framework, which handles this task and continues the search at the specified point in the search space. All user modules have access to data managed by the framework module.

The overall structure is shown in figure 3.1. A more detailed discussion the modules follows while special emphasis is on the user modules.

### 3.1.2 Main

The main module provides the interface for a user of the framework when an actual problem or a number of problems is to be solved. Predicates to retrieve a set of problems matching a name pattern, run the framework on a list of problems with one configuration or with a list of configurations or to perform tests with predefined configurations are available.
Importing functions from the problems and utility module, problems can be read from a file and the level of debug output can be set.

3.1.3 Framework

The primary tasks of the framework module are setting up a problem to be solved, initialising the configuration for the search, providing shared data to the user modules and controlling the search itself. Due to the controlling position of the framework module, several details of the implementation are mentioned briefly in this section already, even if the framework module is concerned with those details in a secondary role. The first thing to happen during setup is the initialisation phase which could be a module of its own. However, in the current implementation, the initialisation phase is part of the framework module and for that reason it is explained in detail at this point:

In a loop, the algorithm chosen for each of the user modules can request certain features from other algorithms or ask them about those features. Another algorithm which understands a request or a question can reply. Finally, the first algorithm can use the information for further initialisation. An example illustrating this process is as follows:

A conflict handling technique which requires explanations will post a request to the explanation algorithm to compute them. The explanation algorithm accepts this request, now knowing that it actually will become active eventually. “SimpleExplain” - one of the explanation algorithms - then asks the framework to monitor conflicts, the core method used by this explanation algorithm, which the framework will accept as well. Because conflict monitoring is incompatible with non-trivial propagation, simpleExplain also asks the propagation algorithm whether it does only consistency checking or whether any propagation is done. If the latter is the case, simpleExplain will cause the initialisation phase to fail because the combination of techniques is not feasible. The overall effect is that no overhead is incurred if the conflict handling algorithm does not need explanations and that an invalid combination of techniques is not permitted.

After the initialisation phase in which the configuration of the framework has been determined, the domains and constraints for the particular problem are set up which means that they are made known to the system. Setting up the domains is a rather straightforward task using the information on variables and domains given in the problem. If a “parallel variable list” - which basically creates a copy of the problem for techniques to work on - was requested, the domains are set up for these “parallel variables” as well. Setting up the constraints is quite similar, with the only difference that conflict monitoring can be turned on or off depending on the outcome of the initialisation phase. In general, setting up a constraint means that a constraint is passed to the propagation algorithm which will take care of that matter in the background for the whole search. However, problems can include numerical constraints like \( \geq, < \) etc. The ECLiPSe system provides built-in propagation for these operators which is used instead of the propagation algorithm the user specified in the configuration. Since the user has no control over the propagation in such cases, this is a violation of the overall concept of the framework which was to give the user a maximum flexibility. However, even though it would have been possible to implement propagation methods identical to the standard propagation methods for these special operators, it is highly likely that any choice deviating from the specialised propagation methods provided by ECLiPSe would have yielded less efficient propagation behaviour. Therefore, the limited control for special operators seems to be as good a trade off as possible to allow users to use numerical constraints, which yield high performance propagation combined with extremely small space requirement, while still allowing them to use predefined propagation methods for generic constraints.

\(^1\) The explanation algorithm QuickXPlain requests this
Before the search itself starts, a second initialisation phase starts in which each algorithm has a chance to take into account the problem specific details which include the number of variables, their domains and domain sizes, the number of constraints on each variable and so on. The learning algorithm makes use of this feature to initialise its internal data structure to record no-goods.

The search process includes a loop of choosing a variable, a value for the variable and handling conflicts if any are detected. This is done using the user modules only (exploration and conflict handling) which are called by the framework module. However, the framework also assists the learning algorithm in the current implementation by checking which no-goods have been discovered in deeper levels of the search tree before the latest backtracks occurred. These new no-goods have to be (re-)activated before the search can continue. The framework also keeps track on which values for the current variable were rejected by the learning algorithm, that is which inconsistencies can be explained by a no-good. This information is used by the conflict handling trying to find the explanation for failure.

3.1.4 Representation

This module is used to translate an external format for problems into the internal one. Methods for choosing any variable out of a set of variables or removing a constraint from a set of constraints are available. This module can be extended by users to provide compatibility with more external formats.

3.1.5 Problems

The problems module provides methods to read problems from files to facilitate handling and storing of large numbers of problems. In addition to this, a function to unload a problem\(^2\) and a number of sample problems exist so that no external problem data files are required to test the framework on a few simple problems.

A problem definition consists of a name for the problem, a list of variables, one (finite) domain for each of these variables and a list of constraints. As mentioned in 3.1.3, these constraints are either standard ones which are handled by the propagation algorithm or constraints built into the ECL\(^{i}\)PS\(^e\) system which are handled by ECL\(^{i}\)PS\(^e\) itself. The basic form of a constraint is a pair \((\text{Scope}, \text{Set Of Valid Value Combinations})\). The scope specifies which variables are constrained and the set of valid value combinations defines which values these variables may be assigned to at the same time. A more general form of standard constraints is an arbitrary goal which contains some variables. Depending on whether such a predicate includes operators for which ECL\(^{i}\)PS\(^e\) provides specific propagation methods or not, this constraint can be handled by the propagation algorithm chosen by the user.

3.1.6 Util

A large number of auxiliary predicates - mainly for list handling - are available in the util module. The output of the framework is managed here as well. The function to change the stream to send output to can be set and the level of detail of output can be adjusted. Finally, a predicate evaluating sets of data and finding their mean value and the variance is available.

3.1.7 Exploration

The exploration module contains two sorts of algorithms and one of each has to be chosen for a configuration. The first type are algorithms that choose variables which are to be

\(^2\)for example to free memory if many large problems are loaded
assigned a value in the next step. Information about the set of uninstantiated variables can be used for this decision. So far, three different algorithms have been implemented for this task: The first algorithm chooses an arbitrary variable thus acting like simple backtracking search. The second one chooses a variable with a minimal domain size and implements a first fail heuristic. The last algorithm implements the most constrained heuristic by counting and comparing the number of constraints on each variable.

The second task fulfilled by the exploration module is to provide algorithms which choose a value for the variable chosen before. This is not restricted to assigning a specific value to the variable but also the domain of a variable could be restricted (for example domain splitting). This however, has not been tested yet and more research has to be done to find out how significant the extensions are that the framework would require to fully support this technique. The implications for conflict handling and learning algorithms, in particular how explanations are to be stored, have to be examined. The only algorithm implemented for this task chooses an arbitrary value for the variable which corresponds to simple backtracking.

3.1.8 Propagation

Propagation methods are supposed to perform look-ahead and reason about the current state which mainly consists of the current partial solution in this case. Propagation is one of the core features of the ECLiPS system and for this reason, the built-in features have been used widely. Four levels of propagation strength are available in the current version of the framework. The first one only checks consistency if all variables of a constraint are instantiated which corresponds to simple backtracking. The other propagation methods use increasingly strong implementations of propagation using built in predicates of ECLiPS. Propagation algorithms simply fail if an inconsistency is detected and the framework reacts to this. If no inconsistency is detected, the domains of some variables might be reduced and nothing else happens.

During the constraint setup, constraints given by the problem definition are analysed and depending on the result, they are passed on to the chosen propagation algorithm or passed to the ECLiPS system for handling. Propagation methods usually delay a constraint goal until some relevant variables are instantiated. When that happens, the goal is woken up by the suspension mechanism provided by ECLiPS and the reasoning done by the propagation algorithm is done. As mentioned before, the goal fails if an inconsistency is detected, otherwise, it succeeds while possibly suspending itself on the remaining unbound variables again.

3.1.9 Conflict Handling

A conflict handling algorithm should provide instructions to the framework concerning which variables have to be retracted to solve a conflict. The simplest way to do so is to report nothing, such as the simple backtracking version does. If no information is returned by the conflict handling, the framework simply retracts the chronologically last variable. For more sophisticated methods, the conflict handling algorithm can query the explanation algorithm to find a reason for the inconsistency. An explanation is a subset of the instantiated variables which - bound to their current values - conflict with one or more constraints of the problem. A typical method for conflict handling which also has been implemented, is conflict directed backjumping. This method instructs the framework to retract all variables up to the first one which is in an explanation.

Conflict handling is engaged if a single value is an inconsistent instantiation for the current value and in another form if all instantiations result in a conflict. No use was made of conflict handling for a single value inconsistency, but to leave this door open
Figure 3.2: To overcome the dependencies between conflict handling, learning and explanation algorithms, the structure of the framework could be changed according to this diagram. The framework module gathers explanations and forwards them to the conflict handling and learning modules.

for future development, the possibility still exists. Conflict handling after failure of all possible instantiations works by collecting and combining the explanations which can be found for each value in the domain of the current variable.

When explanations for each possible value are collected, a “best” one has to be found, that is an explanation which yields the largest jump backwards while it is guaranteed that no solutions are skipped. A predicate which iterates through all values of a given variable and automatically returns the best possible explanation out of all is provided so that users implementing more algorithms do not need to concern themselves with such details.

In the current version of the framework, conflict handling algorithms call the learning algorithms since the knowledge of the reason for each failure is important for learning but only the information relevant to the retraction of variables is returned to the framework. A way to overcome the unnecessary dependency between conflict handling and learning would be to let the framework call the explanation algorithm and forward the explanations to the conflict handling and/or learning algorithm. The initialisation phase could be used to predetermine whether any explanations have to be computed at all. The revised structure as it was outlined here is shown in figure 3.2.

3.1.10 Explanation

As mentioned in 3.1.9, an explanation is a subset of the instantiated variables which is in its current valuation in conflict with the constraints of the problem. These subsets have to be found by explanation algorithms. If no conflict could be found by an explanation algorithm, it is also valid to return an empty explanation which means signals this meaning to the other modules.

Two algorithms have been implemented to find explanations. The first one, simple explain, uses the built in feature of conflict monitoring. If turned on, conflict monitoring can be used to return the set of constraints which are violated by an assignment. Simple explain works by unifying the scopes of all violated constraints which suffices to implement backjumping (Ginsberg, 1993). This union contains all variables that might be responsible for an inconsistency. However, the explanation returned that way might be more general than necessary; it is not guaranteed that the explanation is minimal in the sense that a subset of it is not an explanation any more. Another problem is that conflict monitoring is incompatible with non-trivial propagation methods. Sometimes when
combined with a propagation algorithm that does more than consistency checking on fully instantiated variables, conflict monitoring returns constraints which contain not instantiated variables. In that case, constraints including instantiated variables which caused some domain pruning on the variables in the conflict constraint are omitted and therefore the required information cannot be extracted without major effort. In order to make conflict monitoring compatible with propagation methods, it would be necessary to record all constraints involved in domain reductions during search. Such intrusive behaviour would firstly be opposed to the idea of leaving propagation, conflict handling and finding explanations independent of one another and secondly be cumbersome to implement. For these reasons, the incompatibility has not been repaired.

The second algorithm that was implemented, QuickXPlain, overcomes this issue and always returns minimal explanations. The algorithm was devised by Junker (2001). QuickXPlain is a non-intrusive algorithm which means that the propagation method is not influenced by it at all and conflict monitoring is not required. However, this benefit is countered by the need to (re-)compute the effect of the propagation algorithm on different subsets of all instantiated variables. For this reason, the quickXPlain algorithm takes more time when computing an explanation but takes no time at all during normal propagation. The problem instance determines which explanation algorithm performs more efficiently.

In greater detail, QuickXPlain works on two sets of constraints, the background constraints and “unexplored” constraints. Remember that variable instantiations can be regarded as decision constraints. Initially, all instantiations are unexplored constraints on the problem. QuickXPlain then checks different subsets of the unexplored constraints to determine which of them are required to cause a minimal conflict. Thus, QuickXPlain successively adds unexplored constraints to the background, that is activates them, until the propagation machine detects a conflict. When that happens, the set of recently added unexplored constraints is divided by half into two new subsets. Then quickXPlain is called recursively with the old background unified with the first half of unexplored constraints plus the last decision constraint that was tried when the conflict occurred forming the new background. The unexplored constraints for this first recursive call are ones in the second half of unexplored constraints. Alltogether, they must lead to a conflict.

If the new background contains a conflict set already, an empty explanation is returned. Otherwise the method does the same as stated above and eventually returns an explanation. The first call of QuickXPlain will use this explanation returned after using the first half of unexplored constraints as additional background, add this explanation to the old background and call itself recursively for a second time using the first half of unexplored constraints as new unexplored constraints. The result of QuickXPlain is the unification of the explanations returned by the two recursive calls and the last decision constraint that lead to a conflict.

To find a much more detailed description of the algorithm, the paper by Junker (2001) should be referred to. In principle the algorithm works by recursively partitioning the set of unexplored constraints and finds a minimal conflict set faster than previous methods. In the implementation of the framework, QuickXPlain needs a copy of the problem to operate on. The reason for this is that ECL/PS² as a logic programming language does not allow users to arbitrarily retract assignments of variables at any point. Thus, to simulate the activation of different constraints at different times, these computations have to be carried out on a ‘shadow problem’ which is given by a list of ’parallel’ variables. Each variable in the original problem has its own parallel variable which has a copy of its domain and a copy of its constraints. This parallel problem is created by the framework module during domain setup and constraint setup and has to be requested by the explanation algorithm using the initialisation phase 3.1.3.
3.1.11 Learning

In this framework, learning algorithms have to determine and record sets of instantiated variables which are in conflict with some initial or deduced constraint of the problem. These sets are similar to explanations, but since the information learnt is supposed to be used at all points in the search space, the values for the variables in the explanation are stored as well. An explanation together with the (ordered) set of corresponding values is a no-good. These no-goods are managed and activated by the framework. A learning algorithm has to store all no-goods that it finds in a list which is checked by the framework module. No-goods are then activated by the framework by creating suspended goals which are quite similar to constraints, only that they are not handled by the propagation algorithm and that they are added during the search, not before. It would have been possible to pass these constraints to the propagation machine as well. However, this would have complicated the framework and created a closer relationship between learning and propagation. The data structure used for storing the no-goods has to be initialised by the learning algorithm. Since the structure depends on the number of variables in the current problem instance, this initialisation cannot be done during initialisation phase one. Therefore, the second initialisation phase exists which passes relevant information such as the variable list to the algorithms.

In order to find no-goods, the learning algorithm is called by the conflict handling in the current version of the framework and is given the explanations found during the conflict handling. Naturally, each explanation together with the corresponding values is a no-good, since the definition of an explanation implies that an instantiation of the included variables to their respective values is in conflict with some constraint.

Storing all these no-goods has two disadvantages, though. Firstly, storing all no-goods takes exponential space in worst case. This is obvious since the search tree includes a number of leaves which grows exponentially with the problem size. Secondly, storing only the no-goods found by conflict handling - let us refer to these as primitive no-goods - does not result in great savings. A no-good is checked by the framework as soon as all its variables become instantiated and for a primitive no-good this is at the same time as when the propagation algorithm detects the inconsistency as well.

To make the effect of learning more powerful and reduce the space requirements, some reasoning on the no-goods learnt so far is needed. One way to do so, which is inspired by Ginsberg (1993) and only reduces the space needed, is to perform a so-called subsumption check which checks whether any no-good covers all cases in which other no-goods apply. The first algorithm for subsumption checking follows the ideas outlined in Ginsberg (1993)’s paper. However, in order to create more powerful learning techniques, two extensions have been developed in the course of the project. These algorithms actually make learning more powerful by investing more computational effort into moving on from primitive no-goods to more general ones. Thus they can prune the search space more efficiently. To illustrate the process of subsumption checking for all three algorithms, let for example the following no-goods be given:

- $n_1 := ((x_1, x_2, x_3), (a, b, c))$
- $n_2 := ((x_2, x_4), (b, a))$
- $n_3 := ((x_1, x_3), (a, c))$

$n_1$ means that if any valuation contains $x_1 = a \land x_2 = b \land x_3 = c^3$ then this valuation is either already in conflict or it is a partial solution which can never be extended to a

\footnote{The index 1 for $x_1$ does not mean that $x_1$ is the first variable which is assigned to a value. Several other variables can be instantiated before, between and after the variables in the explanations.}
3.1. STRUCTURAL OVERVIEW

more reasoning is needed. Imagine that the no-goods are neither a subset nor a superset of the variables in \( n_3 \), therefore the final set of no-goods is \( \{ n_2, n_3 \} \).

As mentioned before, this method was proposed by Ginsberg (1993). It reduces the space requirement but to increase the pruning power of the learning component, more reasoning is needed. Imagine that the no-goods \( n_4 := ((x_1, x_2), (a, a)) \) and \( n_5 := ((x_1, x_2), (a, b)) \) have been learnt. If the domain of \( x_2 \) is \( \{ a, b \} \), one can observe that \( n_4 \) and \( n_5 \) include the same set of variables and cover the whole domain of \( x_2 \). Therefore, whenever \( x_1 = a \) holds, either \( n_4 \) or \( n_5 \) will hold which implies that \( n_4 \) and \( n_5 \) can be replaced by \( n_6 := ((x_1), (a)) \). With this reasoning step, the required space has been reduced and - which is the most useful effect - a future problem can be detected earlier. The new no-good will prevent \( x_1 \) from taking the value \( a \) immediately whereas normally the conflict would only be detected when trying to assign a value to \( x_2 \) later in the search tree. Depending on the number of other variables which are assigned before \( x_2 \) is tried, this reduce the search space significantly.

The first two methods of subsumption check together result in some pruning and reduction of space requirements. Like with propagation algorithms, a method as complete as possible is desirable. If more no-goods can be merged, inconsistencies can be detected earlier while less space is required to store all no-goods. To improve the reasoning power of the subsumption check and merge operation, two other versions of the merge process have been developed and implemented.

The second version of the algorithm to merge no-goods has been inspired by the following problem. If the aforementioned no-good \( n_4 \) and the no-good \( n_7 := ((x_2), (b)) \) are stored, then the domain of \( x_2 \) is completely covered as well, but the sets of variables in the no-goods are not exactly equal any more. However, one is the subset of another. Including all no-goods concerning a subset of the variables of the set of variables in some currently observed no-good increases the complexity of the reasoning algorithm but the merging power increases similarly. In addition, the new no-good does not always subsume all no-goods used to deduce it. Therefore it cannot be guaranteed that this method reduces the space requirement in general. Often, the increased pruning power outweighs the space requirements though which makes this method worthwhile.

The third and most powerful version to merge no-goods relaxes the condition on the relation between the sets of variables even more. Given the no-goods \( n_8 := ((x_1, x_2), (a, b)) \) and \( n_9 := ((x_2, x_3), (a, e)) \), these can be combined into another no-good: \( n_{10} := ((x_1, x_3), (a, e)) \). If \( x_1 = a \) and \( x_3 = e \) then any value for \( x_2 \) will match one of the no-goods. The method used here is to consider all no-goods that contain a variable being observed currently. Then, all sets of no-goods that cover the domain of this variable completely are merged into a new no-good if that is possible. The reason why this might fail is that two no-goods might include the same variable (different from the currently observed one) but forbid a different value for it. Again, it is important to note that often the new no-good does not subsume the no-goods used to deduce it. In general, this “merging” method does not even deduce no-goods of smaller size than the one used in the deduction. It is easy to understand that the number of no-goods in the database increases dramatically using this method. Not surprisingly, a few experiments, which are not summarized in this document, have shown that the method is not feasible if used just as it is. The computational effort for the reasoning process is so dominating that the pruning possible with the results can never outbalance that cost. The solution to this problem is the application of a forgetting algorithm.

Forgetting algorithms are not a module of their own in the current version of the framework, but it would be easy to change this. Forgetting plays an important role for
learning algorithms Ginsberg (1993); Jiang et al. (1996). Ginsberg (1993) proposed to forget, that is erase, all no-goods which deviate from the current partial solution. That means that if a no-good concerns a variable which already has been assigned another value than the one forbidden by this no-good, then this no-good is irrelevant for the current branch in the search tree and can be forgotten. As described in the previous chapter 2.4.3, finding a good trade off between deleting many possible useful no-goods and sustaining a large amount of data stored is difficult. In theory, all no-goods are relevant at all times. The proposed forgetting algorithm attempts to draw a sensible line between the theoretically and immediately/practically relevant no-goods.

Forgetting no-goods as it is supported by the framework means that no-goods are completely deleted, not only put aside until they become directly relevant again. Keeping the no-goods would have no effect on the space complexity, the reduction of which is one of the main aims in using a forgetting algorithm in first place.

Applying this forgetting algorithm makes the space complexity polynomial. This algorithm has been implemented and tested in connection to all merging algorithms discussed before. While the effect on the first two merging algorithms was insignificant or even detrimental, the third and strongest version could run not only in acceptable time with this addition, but it could even outperform the other two methods. Since this part of the research is merely an addition to the actual aim of the project, the relevant test results are not described or explained in this document.

3.2 Module Interfaces

While in the previous section 3.1 the objectives of the modules were described mostly individually, the focus of this section is on the communication between the user modules and the framework. If more algorithms are to be implemented for either of the user modules, it is essential to know which information is available and what conditions the result has to obey to. The predominant aim in the design of the interfaces is to maximise the flexibility which is achieved by providing as much useful information as possible while imposing as few restrictions as possible on the output. Each user module will be discussed separately. For each module, the list of input parameters and the expected output parameters is listed. To formalize the requirements, preconditions which can be expected to be true and postconditions which have to be fulfilled by an algorithm are given. All algorithms can use a ‘robust’ data storage called a shelf provided by the framework which allows the algorithms to store data which is not erased by backtracking.

3.2.1 Exploration - Choosing a Variable

**Input:** list of variables, list of indices of not instantiated variables  
**Output:** index of the chosen variable  
**Preconditions:** Both lists are not empty and for each index in the second list there is one variable in the first list.  
**Postconditions:** The index of the variable to assign must be in the list of indices of not instantiated variables.

The most important input parameter is the list of indices of variables which are not yet instantiated. Obviously, one of the variables referred to by these indices has to be chosen. The list of variables can be used to retrieve information about individual variables like their domain and the constraints on them. The index of the chosen variable has to be returned. This interface leaves room for a wide range of techniques, including stochastic algorithms.
3.2.2 Exploration - Choosing a Value

**Input:** the variable to be assigned, the index of this variable, indexlist of not instantiated variables, indexlist of instantiated variables, list of initial constraints  

**Output:** updated indexlist of not instantiated variables, updated indexlist of instantiated variables  

**Preconditions:** The variable to assign is an uninstantiated problem variable and the index of this variable refers to it. The index is an element of the indexlist of not instantiated variables and it is not an element of the indexlist of instantiated variables.  

**Postconditions:** If the algorithm tries to instantiate the variable, then the index of the assigned variable is moved to the indexlist of instantiated variables. The variable is instantiated or an instantiation results in a conflict and the variable is tentatively instantiated. If the variable domain is to be reduced only, the indexlists remain unchanged.

An algorithm to choose a value for a variable can work in two ways. The first one is it assign one particular value to the variable and instantiated it. The current version of the framework has been tested with such an algorithm. The domain of the variable can be retrieved using functions provided by ECLiPSe. The variable has to be assigned one value out of this domain, which can either work or result in the detection of some conflict. If a conflict occurs, the exploration algorithm has to tentatively instantiate the variable so that the conflict can be analysed.

Another option is to constrain the variable further, for example by splitting its domain in half. This method has not been tried in the implementation. Further investigation on this topic includes the implications for explanation finding, learning and the communication between the two exploration techniques. The algorithm choosing the next variable should be instructed to choose another variable in the next step because otherwise the same variable would be constrained further and the end result would be the same as an instantiation after a number of steps.

3.2.3 Propagation

**Input:** scope of a constraint, a constraint goal, the indexlist of the scope  

**Output:** none  

**Preconditions:** The constraint goal contains exactly the problem variables that form the scope and the indexlist corresponds to these variables.  

**Postconditions:** The propagation predicate fails if the constraint goal is violated.

Propagation algorithms are different from algorithms for other user modules insofar as they are called once during the constraint setup and are in general delayed until sufficiently many (usually all) variables are instantiated. Depending on the propagation strength, uninstantiated variables are bound by the propagation if their value can be inferred using the constraint. Most of the work necessary for propagation is provided by ECLiPSe. Other propagation techniques, for example the consistency check which is implemented already, can make use of the suspension library provided by ECLiPSe which allows delaying goals until some condition is satisfied.

3.2.4 Conflict Handling - Single Conflict

**Input:** the variable to assign, its index, a value, the list of all variables, the list of “parallel” variables, indexlist of instantiated and not instantiated variables, a reference to the explanation algorithm and to the learning algorithm, the initial list of constraints  

**Output:** none  

**Preconditions:** The current variable cannot be instantiated to the value, because this
instantiation conflicts with some constraint on the problem.

**Postconditions:** none

With this function, some information can be generated as soon as an inconsistent value is detected for a variable. However, this feature is not used by any conflict handling algorithm implemented and is just kept to give more flexibility for future developments. The list of parallel variables might be required if an explanation has to be computed. That list is one of the input parameters for any explanation algorithm.

### 3.2.5 Conflict Handling - Dead End Conflict

**Input:** the variable to assign, its index, the list of all variables, the list of “parallel” variables, indexlist of instantiated and not instantiated variables, a list of the values for the current variable which were ruled out by no-goods together with the explanation, a reference to the explanation algorithm and to the learning algorithm, the initial list of constraints

**Output:** the list of variables to retract (stored in a so called shelf)

**Preconditions:** The current variable cannot take any of the values in its current domain because a conflict is detected either immediately or later during the search. The list of values ruled out by no-goods contains only values which are inconsistent and the respective explanation is given.

**Postconditions:** The list of variables to retract is empty if no information could be generated and otherwise only contains variables which have to be retracted so that the remaining partial solution can be extended to a solution.

The parallel variable list might be required by the explanation algorithm and is just forwarded if the explanation algorithm is called. Auxiliary predicates exist to assist the user in finding the best explanation which obeys to the postconditions and to handle the internal data format. It is safe to choose all variables from the current one to the chronologically last variable which also appears in the best explanation (excluding the current one of course) to be retracted.

### 3.2.6 Explanation

**Input:** index of the variable to assign, a value, a parallel variable list, the list of instantiated variables

**Output:** an explanation, that is a set of variables which are in conflict with some problem constraint

**Preconditions:** The given value is in the domain of the variable referred to by the index.

**Postconditions:** If the instantiation of the variable to the value results in a conflict, some explanation containing only instantiated variables of the current problem returned, otherwise, an empty explanation is returned.

The parallel variable list can be used in combination with the indexlist of instantiated variables to try different combinations of instantiations like quickXPlain does. The explanation returned must fulfill the condition that it is empty or if all variables in the explanation are assigned to their current value (and the current value is assigned to value) then the resulting valuation conflicts with some constraint of the problem.

### 3.2.7 Learning

**Input:** an explanation, the index of the variable that was tried to be assigned, the indexlist of instantiated variables, the value the current variable should be assigned, (from
3.3 Evaluation Methods

The final section in this chapter covers the procedures for testing and evaluating the framework. The main objective of the project is to develop a framework which allows users to combine a large number of different techniques into one search algorithm. Performance is important but plays in minor role in this research.

It is difficult to define exact measures on how easily different techniques can be combined and how well existing techniques fit into the structure of the framework. Still, an informal evaluation of the process of implementing different techniques can be made. A criteria which covers a part of the objective might be to measure how often an incompatibility between techniques occurs. If many techniques are incompatible with a number of other techniques in different modules, the framework fails to achieve the objective of flexibility and little interdependence between the modules. If however almost any technique can be combined with any other, then the chosen structure fulfills the objective.

Measuring the performance is less concerned with comparing different search algorithms - resulting from different combinations of techniques - than with measuring the overhead created by providing the flexibility. For this reason, most tests compare a standard backtracking algorithm with the framework configured to perform standard backtracking. An acceptable overhead is anything up to 100% which means that the framework can take at most twice as long as standard backtracking. The ideal and practically impossible result would be an overhead of 0% which would mean that the framework is as fast as standard backtracking while allowing users to generate and run quite different algorithms with more sophisticated techniques with a minimum of effort.

A point to consider when measuring performance is how much time is used for the setup of a problem and how much time is actually spent on searching. Setting up a problem is an effort made only once before the search starts which makes this time less decisive. In contrast, the time spent on searching determines how much of the overhead is incurred in repetetive processes.

A very important point when it comes to testing is the test data. On the one hand, it is desirable to have a large number of test samples. On the other hand, the test samples should be sufficiently diverse to represent the spectrum of possible problems well enough. Generating problems by hand allows for a good level of diversity but is extremely cumbersome when large number of problems are required. In order to find a trade off between these two goals, two problem types are used. Firstly, the “Uniform Random Problem Generator” (n.d.) is used to generate large number of similar problems. Parameters of the generator are the number of variables, their uniform domain size, the number of binary constraints between the variables and the uniform tightness of the constraints. The tightness of a constraint defines how many of all possible value combinations are forbidden by a constraint. Having $n = 10$ variables with a uniform domain size of $m = 5$, the number
of constraints can range between 0 and \( \frac{n(n-1)}{2} \) and the tightness can range between 0 and \( m^2 \).

As a second type of problems to evaluate the framework on, a few real world problems are used. The problem domain is job-shop scheduling a definition of which can be found in (Schmidt, 2001). Using such real world problems makes for a better estimation of the performance since real world problems are far more structured than randomly generated ones.
Chapter 4

Results

This chapter contains the results obtained by testing the framework. The most important objective in the development was to design and build a framework which has a number of well separated modules which do not interfere significantly with one another. The framework was designed to be usable for future research on search algorithms. Therefore, the aims include that new techniques can be implemented without changing the overall structure by using the predefined interfaces only. Techniques that follow the interface definitions should be as interchangeable as possible. Another aim was that a user who wants to solve a particular problem can do so easily without having to learn much about the internal details of the framework. Switching from one algorithm, or in other words one combination of techniques, to another was planned to be extremely easy.

A framework which offers the aforementioned flexibility definitely has some value of its own. However, to become a useful tool for research, the framework has to be competitive with other ones. If the performance of the framework is significantly worse than the performance which can be reached by implementing an algorithm directly and without offering any flexibility, then the practical use of the framework is limited. To evaluate the performance of the framework, it has been tested against standard backtracking search with and without propagation. An acceptable overhead given the great increase in flexibility was considered to be about 100%. An overhead of 100% means that, on the average, the framework takes 100% more time, that is twice as much, as a directly implemented backtracking algorithm.

As described in the previous chapter, the tests have been run using a number of randomly generated problems and several real world problems taken from the domain of job shop scheduling problems. The main focus of the evaluation was on the running times of the framework and a simple backtracking algorithm in comparison.

The next section in this chapter describes the results concerning the flexibility achieved by the framework. Since it is difficult to measure the flexibility and compatibility of modules and techniques, this evaluation has to remain fairly informal. The rest of the chapter contains the evaluation of the overhead testing on a number of different problems. An interpretation and discussion of these results is given in the next chapter.

4.1 Separability of Modules

As outlined in the previous chapter (3.3), a useful measure for the flexibility the framework offers is to examine incompatibilities between modules. The only incompatibility that occurred during in phase in the development of the framework was that using the built-in methods of ECLIPS to monitor conflicts for finding explanations is not compatible with any propagation method that does more than consistency check on instantiated variables. The reason for the incompatibility is that propagation methods can detect that a partial
solution cannot be extended to a complete solution because a particular constraint will be violated no matter how the remaining variables in its scope are instantiated. Conflict monitoring returns all constraints and consequently all variables in their scopes as the explanation, which means that uninstantiated variables might be included in the explanation. This violates the condition that an explanation may only contain instantiated variables.

The fact that the explanation module and - in the current implementation - the learning module are activated by the conflict handling module creates a further dependency. If a technique used for conflict handling does not use explanations and/or does not invoke the learning algorithm, it simply does not matter which techniques are chosen for explanation and learning.

Exploration algorithms are completely independent from other modules. The only issue is that the technique used for choosing values can create a further condition which the technique used for choosing a variable has to obey to. That is, if the domain of a variable is to be split instead of instantiating the variable, the next variable chosen should be a different one. Otherwise the same domain would be split until there is only one value left which instantiates the variable and that basically erases the effect of domain splitting (see 3.2.2). The initialisation phase can be used to determine whether both techniques for choosing variables and values can work together in an effective way or to configure the algorithms appropriately.

An effect of including constraints which can contain built-in operators of the ECL/PS e system is that the user has no control over the propagation applied to the goals containing these operators. The ECL/PS e system provides its own propagation method for constraints such as numerical comparisons (>, <, . . .) or high-level constraints like alldifferent. On the one hand, the limited control over the propagation is a downside of this. On the other hand, it usually not sensible to “artificially” modify the propagation applied to such goals if problems are to be solved as efficiently as possible. Simple consistency checks can be implemented easily if for some tests, no propagation at all is desired. Apart from this, the propagation methods work independently from the other techniques and a particular problem.

Before a summary of the modularity of the framework is given, the following section will describe how a configuration and a problem to be solved are chosen on the user level. To understand the meaning of the configuration encoding, one should consult the explanation in figure 4.1.

In order to load one or more problems from the hard disk, the predicate readProblemFromFile/2 is used. The first parameter is the filename and the second (optional) parameter is used to give the problem names a prefix so that they can be accessed as a group easily. The predicate allProblems/2 and test/2 are used to solve a set of problems with a given configuration. allProblems/2 takes a name or name pattern as its first parameter and finds all problems which have a name matching the pattern. The problems are returned in the second argument which the can be passed to test/2. The second parameter of test/2 is a list of numbers which defines the configuration. The encoding of the configuration can be looked up in figure 4.1.

To give a detailed example, imagine there are several problems in a file called ‘test-Problems.txt’ and we would like them to be solved using an algorithm which does standard backtracking. The queries would look like this:

?- readProblemFromFile('testProblems.txt', myTestProblems).
...
yes
?- allProblems(myTestProblems-_, ProblemList),
   test(ProblemList, [1,1,1,1,1,1]).
...

If after this experiment a different configuration should be used, the changes are minimal. Let the required configuration define an algorithm using the first-fail heuristic, QuickXPlain, no learning, medium propagation and backjumping. Since the problems are already loaded, the first step does not need to be repeated. The new query is:

```prolog
?- allProblems(myTestProblems-, ProblemList), test(ProblemList, [2,1,2,1,3,2]).
...
```

The first “2” changes the heuristic for choosing variables to first-fail, the second “2” sets the explanation algorithm to QuickXPlain\(^1\), the “3” changes the propagation method from “consistency checks only” to “medium strength propagation” and finally, the last “2” changes the conflict handling algorithm from standard backtracking\(^2\) to backjumping.

A few other predicates are available to solve problems with several configurations in a row or to solve a single problem in a predefined configuration to further reduce the effort.

In summary, some minor problems impede the chosen framework structure, but complications exist only between a few modules and only if certain types of techniques are used. In general, the interface definitions are sufficient. Changing the configuration of the framework is extremely easy and takes neither much time nor extensive knowledge about the framework.

### 4.2 Performance

To test the performance of the framework as opposed to the performance of a direct implementation of an algorithm, standard backtracking has been applied to a number of randomly generated problems and several real world problems. As mentioned before in the previous chapter (3.3), the random problems provide a large sample set whereas real world problems are used to determine the efficiency on problems with a structure more likely to occur in applications.

#### 4.2.1 Random Uniform Problems

As random problems are easy to generate and easy to evaluate due to their similar structure, a fairly large number of those was generated. The problems were created using the “Uniform Random Problem Generator” (, n.d.). Parameters for this generator are:

- the number of variables
- the domain size
- the number of constraints
- the tightness of the constraints
- a seed for the random number generator
- a number of instances to create

The domain size is the same for each variable just as the tightness of each constraint is the same. The problems contain only binary constraints between any two randomly chosen variables which forbid certain value combinations between those two variables. If the number of variables in a problem is \(n\) and the uniform domain size is \(m\), then

\(^1\)Note that in the first version, the explanation algorithm was irrelevant since backjumping was turned off.

\(^2\)that is, doing nothing at all
the maximum number of constraints obviously is $\frac{n(n-1)}{2}$ and the maximum number of forbidden value combinations in each constraint is $m^2$.

Problems have been generated with different number of variables, domain sizes, numbers of constraints and tightness of the constraints. Of each problem type, 100 instances were created using an arbitrary seed value for the random number generator. The problems were generated with the aim that they should not be too easy to solve. Problems with hardly any constraints or problems which are not solvable but so overconstrained that this can be determined quite quickly are not interesting for the evaluation because their runtimes are so low that they approach the borders of accuracy given by the programming language. As an informal measure for the suitability of the problems, the solutions were observed. If many instances had no solution, a new, easier problem set was created. If many problems were solved but a large number of variables took the first value in its domain, the problems were discarded as well and new, harder problem instances were generated.

As a final objective, the problems were made not to be too difficult to solve insofar as the runtime to solve each problem should not exceed approximately 3 minutes\(^3\). The testing machine was a standard desktop PC i386 1.5 GHz with 256 MB RAM, running a linux based operating system.

For the evaluation, the running times of the framework configured to do standard backtracking search and a standard backtracking algorithm which was implemented directly were measured. The labeling steps required to solve each problem were counted and the averaged results over each of the 100 problem instances are summarized in table 4.1.

In this table, the first columns identify a problem class by the number of variables, the domain size, the number of constraints and their tightness. The number of constraints and the tightness are given both as absolute values as well as relative amounts compared to the maximum number possible. The next columns list the average runtime of the directly implemented labeling algorithm and the respective standard deviation, followed by the same information about the framework performance. Two compare the runtimes, the proportional difference is given as the overhead in percent. As mentioned above, an overhead up to 100% is a satisfactory result, while 0% is the ideal case where no overhead at all is incurred. For further information, the average time that the labeling algorithm spent on each labeling step in milliseconds is given followed by the average difference between this value and the according time spent by the framework on each labeling step.

A more detailed display of the runtimes is given in the appendix where the distribution of the aforementioned values among the 100 instances in each problem class is shown.

\(^3\)If each problem takes 3 minutes to be solved, 100 instances solved twice require about 10 hours of computation
Table 4.1: Overhead test results, averaged over 100 problem instances in each row. The factor by which the framework is slower than direct labeling is given by the average runtime overhead in percent. The average time difference used per labeling step by the framework and direct labeling is given and can be compared to the total time spent per labeling step by direct labeling.

<table>
<thead>
<tr>
<th>No</th>
<th>#Vars</th>
<th>#Values</th>
<th>Constraints</th>
<th>Tightness</th>
<th>Avg. labeling steps</th>
<th>Avg time labeling (s)</th>
<th>Std.Dev labeling</th>
<th>Avg time framework (s)</th>
<th>Std.Dev framework</th>
<th>Avg runtime overhead (%)</th>
<th>Avg rt. per labeling step (ms)</th>
<th>Avg rt. difference per label. step (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>20</td>
<td>9</td>
<td>0.90</td>
<td>320</td>
<td>105.52</td>
<td>0.0870</td>
<td>0.1424</td>
<td>0.1291</td>
<td>66.78</td>
<td>0.8245</td>
<td>0.6417</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>40</td>
<td>12</td>
<td>0.8</td>
<td>1300</td>
<td>545.44</td>
<td>2.7440</td>
<td>6.6881</td>
<td>3.2511</td>
<td>18.480</td>
<td>5.0308</td>
<td>1.0675</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>10</td>
<td>25</td>
<td>0.56</td>
<td>70</td>
<td>1058.46</td>
<td>0.2150</td>
<td>0.4280</td>
<td>0.5316</td>
<td>146.797</td>
<td>0.2031</td>
<td>0.3464</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>12</td>
<td>100</td>
<td>0.95</td>
<td>40</td>
<td>7037.24</td>
<td>17.9470</td>
<td>19.6693</td>
<td>19.8940</td>
<td>216.505</td>
<td>2.5502</td>
<td>0.2751</td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>2</td>
<td>25</td>
<td>0.16</td>
<td>1</td>
<td>116.60</td>
<td>0.0060</td>
<td>0.0094</td>
<td>0.0341</td>
<td>487.931</td>
<td>0.0515</td>
<td>0.2295</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>4</td>
<td>25</td>
<td>0.16</td>
<td>7</td>
<td>38432.59</td>
<td>2.5931</td>
<td>9.5194</td>
<td>9.8203</td>
<td>278.709</td>
<td>0.0675</td>
<td>0.2096</td>
</tr>
<tr>
<td>7</td>
<td>18</td>
<td>8</td>
<td>52</td>
<td>0.34</td>
<td>28</td>
<td>130675.06</td>
<td>44.5711</td>
<td>115.4553</td>
<td>83.8014</td>
<td>215.5063</td>
<td>0.3411</td>
<td>0.3343</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
<td>2</td>
<td>7</td>
<td>0.04</td>
<td>2</td>
<td>2304.70</td>
<td>0.0513</td>
<td>0.1738</td>
<td>0.4317</td>
<td>741.521</td>
<td>0.0221</td>
<td>0.0722</td>
</tr>
<tr>
<td>9</td>
<td>22</td>
<td>5</td>
<td>70</td>
<td>0.30</td>
<td>8</td>
<td>140630.33</td>
<td>26.3719</td>
<td>87.9562</td>
<td>64.6288</td>
<td>209.1562</td>
<td>0.1875</td>
<td>0.3133</td>
</tr>
</tbody>
</table>
4.2.2 Job Shop Scheduling Problems

Job shop scheduling is a problem type taken from real world applications. Testing the framework on such problems is believed to yield a more reliable way to estimate the performance of the framework compared to standard implementations.

In job shop scheduling, a number of jobs has to be done on a certain number of machines. Jobs are divided up into a number of operations which have to be executed on a particular machine and occupy this machine a certain amount of time. Operations belonging to one job have to be executed sequentially in the correct order and no machine can process two operations at the same time. The objective is to find a schedule for each machine, that is an ordering of the operations, so that the constraints mentioned above are obeyed to and the duration until the last job is finished is minimal. This problem is strictly speaking an optimisation problem. However, by constraining the time of the last completion to the optimal time value, the problem can be treated as a constraint satisfaction problem. In the same way, the optimal solution can be found by successively tightening the time bound on the last completion.

An optimisation procedure of this kind can be added to any constraint satisfaction problem solving framework, but since it would involve a few non-trivial changes, the optimal solutions for job shop scheduling problem instances were found by restarting the search after manually tightening the bound. This way, the framework is not changed and results from previous tests are not invalidated.

As a consequence, the number of job shop scheduling problems used in the tests is significantly lower than the number of random problems. Therefore, the statistical significance of the following results is impeded, although the results can be used as a careful estimate towards the performance for real world applications.

The test samples consist partly of job shop scheduling problems taken from (Mattfeld and Vaessens, n.d.) which is a source of commonly used problem instances for benchmarking. Due to the high running times for several of the problem instances, the optimal solution could not always be confirmed. For that reason, a few other problem instances were generated manually with the objective to create easier instances for which the optimal value can be confirmed more easily. It should be noted that the arithmetic constraints imposed on the problem are propagated by ECLIPSE automatically. Propagation is done for all constraints on job shop scheduling problems in the chosen modelling and therefore the algorithms used in the framework and the direct implementation are standard backtracking plus look-ahead by propagation.

The results of the tests are summarized in table 4.2 and include a description of the problem features such as the number of jobs, machines and operations per job and if applicable, the name under which the problem instance is known generally. Each problem instance is listed in one row and so the values are not averaged.
Table 4.2: This table shows the overhead of the framework tested on job shop scheduling problems compared to a backtracking algorithm without any features except for constraint propagation. Each line corresponds to one problem instance, defined by the number of jobs, machines and operations per job. Runtimes of the standard algorithm and the framework are shown and the overhead in percent and the average runtime per labeling step are given. Problem instances taken from (Mattfeld and Væsens, n.d.) are listed with their name, other problems are listed with '-'.

<table>
<thead>
<tr>
<th>Name</th>
<th>#Jobs</th>
<th>#Machines</th>
<th>#Operations per job</th>
<th>Runtime labeling</th>
<th>Runtime framework</th>
<th>Labeling steps</th>
<th>Runtime overhead</th>
<th>Runtime per label. step (labeling)</th>
<th>Runtime difference per label. step (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ft06</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>1.10</td>
<td>3.02</td>
<td>561</td>
<td>174.55</td>
<td>1.9608</td>
<td>3.4225</td>
</tr>
<tr>
<td>la01</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>35.06</td>
<td>134.63</td>
<td>16805</td>
<td>284.00</td>
<td>2.0863</td>
<td>5.9250</td>
</tr>
<tr>
<td>la02</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>2.16</td>
<td>12.87</td>
<td>1746</td>
<td>495.83</td>
<td>1.2371</td>
<td>6.1340</td>
</tr>
<tr>
<td>la03</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>4.01</td>
<td>27.75</td>
<td>1040</td>
<td>592.02</td>
<td>3.8558</td>
<td>22.8269</td>
</tr>
<tr>
<td>la04</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>466.47</td>
<td>1267.28</td>
<td>145487</td>
<td>171.68</td>
<td>3.2063</td>
<td>5.5043</td>
</tr>
<tr>
<td>ft20</td>
<td>20</td>
<td>5</td>
<td>5</td>
<td>566.62</td>
<td>1423.28</td>
<td>204471</td>
<td>151.19</td>
<td>2.7712</td>
<td>4.1896</td>
</tr>
<tr>
<td>-</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>11.19</td>
<td>22.27</td>
<td>3340</td>
<td>99.02</td>
<td>3.3503</td>
<td>3.3174</td>
</tr>
<tr>
<td>-</td>
<td>7</td>
<td>3</td>
<td>5</td>
<td>2296.33</td>
<td>4885.39</td>
<td>1026012</td>
<td>121.75</td>
<td>2.2381</td>
<td>2.5234</td>
</tr>
<tr>
<td>-</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>0.17</td>
<td>0.38</td>
<td>53</td>
<td>123.53</td>
<td>3.2076</td>
<td>3.9623</td>
</tr>
<tr>
<td>-</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>0.16</td>
<td>0.32</td>
<td>115</td>
<td>100.00</td>
<td>1.3913</td>
<td>1.3913</td>
</tr>
<tr>
<td>-</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>1672.67</td>
<td>3423.28</td>
<td>876290</td>
<td>104.66</td>
<td>1.9088</td>
<td>1.9978</td>
</tr>
<tr>
<td>-</td>
<td>8</td>
<td>8</td>
<td>5</td>
<td>0.43</td>
<td>1.12</td>
<td>124</td>
<td>160.47</td>
<td>3.4677</td>
<td>5.5645</td>
</tr>
</tbody>
</table>
4.2.3 Performance of Different Configurations

Although not used to measure the efficiency of the framework compared to other implementations, a quick overview of performance of different configurations generated with the framework is useful. The purpose of the following tests is to give an idea of how the framework performs on a few problems with different configurations. The techniques themselves and most of their possible combinations have been discussed in the literature already and so the tests and their evaluation remain cursory.

The tests were carried out solving the first 10 problems of some of the randomly generated problems, in particular the problems used in 4.1 with 15 variables (A) and those with 18 variables and 8 values (B). Furthermore, ten problems from another set of randomly generated problems with 25 variables, 16 values, 55 constraints (18.33%) and tightness 90 (35.16%) (C), have been tested. The runtimes of each problem and each configuration are listed in 4.3.

A configuration of the framework is given as a 6-tuple of numbers, see fig. 4.1 for details. The first number defines which exploration algorithm is used for choosing variables. The first choice - standard backtracking - chooses an arbitrary variable⁴. The other two choices implement the heuristics first-fail and most-constrained. The second number defines the algorithm used to choose values for variables. At the moment, only one, which chooses the smallest value in the domain first, is implemented.

The third number stands for the explanation algorithm. Whether the number is relevant depends on the conflict handling algorithm, defined by the last number. If no conflict handling is done, no explanations are computed. The fourth choice sets the learning algorithm. Apart from 'no learning', three degrees of learning can be chosen. They differ in the amount of subsumption checking done (see 3.1.11). Stronger learning algorithms spend more time on merging no-goods after a new one is added, but the merged no-goods are likely to be more general and therefore prune the search space better.

The fifth number defines the propagation algorithm used. No propagation reduces the algorithm to consistency checking. More powerful versions use increasingly powerful built-in predicates of ECLIPS. As mentioned before, the sixth and last number stands for the conflict handling. 'None' means that, in case of conflicts, nothing is done but retracting the chronologically last variable.

Figure 4.1: Decomposing the encoding of a framework configuration

---

⁴more precisely, the first variable in the list of all problem variables is chosen
It is expected that the results will not indicate that one particular combination is superior to all others, as the efficiency of different algorithms is always heavily problem dependent. A possible bias towards a certain combination might be the result of the limited problem sample set. Again, the purpose of the tests is to illustrate different algorithms at work, nothing else.
Table 4.3: The table shows the runtimes of the framework in different configurations for 30 problems taken from random problems (A: 15 variables, B: 18 variables, C: 25 variables). The column 'total runtime' contains the sum of runtimes for 10 problems each in each configuration. The results for the problems in group C are shown in the next table 4.4. Refer to 4.2.3 in order to understand the configuration encoding.

<table>
<thead>
<tr>
<th>ID</th>
<th>Configuration</th>
<th>Total runtime (s)</th>
<th>Runt. prob 1 (s)</th>
<th>Runt. prob 2 (s)</th>
<th>Runt. prob 3 (s)</th>
<th>Runt. prob 4 (s)</th>
<th>Runt. prob 5 (s)</th>
<th>Runt. prob 6 (s)</th>
<th>Runt. prob 7 (s)</th>
<th>Runt. prob 8 (s)</th>
<th>Runt. prob 9 (s)</th>
<th>Runt. prob 10 (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>111111</td>
<td>1913.5886</td>
<td>25.8900</td>
<td>1667.2700</td>
<td>4.6899</td>
<td>73.9800</td>
<td>41.4599</td>
<td>2.7900</td>
<td>0.0199</td>
<td>54.8500</td>
<td>5.6499</td>
<td>36.9899</td>
</tr>
<tr>
<td></td>
<td>111121</td>
<td>1.8995</td>
<td>0.2999</td>
<td>0.3500</td>
<td>0.1199</td>
<td>0.1400</td>
<td>0.3699</td>
<td>0.0800</td>
<td>0.0799</td>
<td>0.1600</td>
<td>0.1699</td>
<td>0.1300</td>
</tr>
<tr>
<td></td>
<td>112122</td>
<td>12.4394</td>
<td>2.1399</td>
<td>3.1300</td>
<td>0.5299</td>
<td>0.7799</td>
<td>3.2000</td>
<td>0.2999</td>
<td>0.1399</td>
<td>1.3099</td>
<td>0.8899</td>
<td>0.2000</td>
</tr>
<tr>
<td></td>
<td>112222</td>
<td>7.6095</td>
<td>0.9000</td>
<td>1.1899</td>
<td>0.4300</td>
<td>0.5799</td>
<td>2.5200</td>
<td>0.2699</td>
<td>0.1399</td>
<td>0.7200</td>
<td>0.6700</td>
<td>0.1899</td>
</tr>
<tr>
<td></td>
<td>112422</td>
<td>13.0894</td>
<td>5.7200</td>
<td>1.3299</td>
<td>0.4400</td>
<td>0.6099</td>
<td>2.8000</td>
<td>0.2799</td>
<td>0.1399</td>
<td>0.8100</td>
<td>0.7599</td>
<td>0.1999</td>
</tr>
<tr>
<td></td>
<td>212222</td>
<td>3.5397</td>
<td>0.3000</td>
<td>0.1499</td>
<td>0.4900</td>
<td>0.2200</td>
<td>0.2100</td>
<td>0.1399</td>
<td>0.2599</td>
<td>0.4800</td>
<td>0.6600</td>
<td>0.6300</td>
</tr>
<tr>
<td></td>
<td>312222</td>
<td>8.7196</td>
<td>1.4900</td>
<td>0.2200</td>
<td>0.2300</td>
<td>0.3299</td>
<td>0.9500</td>
<td>1.3200</td>
<td>3.0099</td>
<td>0.3899</td>
<td>0.1400</td>
<td>0.6399</td>
</tr>
<tr>
<td></td>
<td>312422</td>
<td>7.9196</td>
<td>0.4500</td>
<td>0.2199</td>
<td>0.2400</td>
<td>0.3300</td>
<td>0.6699</td>
<td>2.9500</td>
<td>1.8900</td>
<td>0.3899</td>
<td>0.1400</td>
<td>0.6399</td>
</tr>
<tr>
<td></td>
<td>112232</td>
<td>10.0894</td>
<td>0.2999</td>
<td>0.4700</td>
<td>0.5299</td>
<td>3.3699</td>
<td>2.1900</td>
<td>0.2899</td>
<td>0.2200</td>
<td>0.2999</td>
<td>1.8600</td>
<td>0.5599</td>
</tr>
<tr>
<td></td>
<td>311111</td>
<td>622.8800</td>
<td>115.2300</td>
<td>213.7400</td>
<td>3.0400</td>
<td>5.3000</td>
<td>10.6000</td>
<td>18.5500</td>
<td>111.4500</td>
<td>15.9600</td>
<td>77.7600</td>
<td>51.2500</td>
</tr>
<tr>
<td>B</td>
<td>111111</td>
<td>287.1797</td>
<td>32.4000</td>
<td>83.1800</td>
<td>53.8099</td>
<td>1.6900</td>
<td>24.6700</td>
<td>4.5600</td>
<td>0.0799</td>
<td>0.2500</td>
<td>4.3899</td>
<td>82.1500</td>
</tr>
<tr>
<td></td>
<td>111121</td>
<td>18.8796</td>
<td>1.6499</td>
<td>6.0799</td>
<td>1.8200</td>
<td>0.7200</td>
<td>2.3400</td>
<td>0.8699</td>
<td>0.5400</td>
<td>0.5499</td>
<td>0.8100</td>
<td>3.4700</td>
</tr>
<tr>
<td></td>
<td>111112</td>
<td>168.7198</td>
<td>165.9300</td>
<td>474.5000</td>
<td>288.0900</td>
<td>11.5300</td>
<td>156.7500</td>
<td>27.6799</td>
<td>0.5700</td>
<td>1.2399</td>
<td>30.7000</td>
<td>529.7300</td>
</tr>
<tr>
<td></td>
<td>112112</td>
<td>2201.4497</td>
<td>236.4200</td>
<td>574.3900</td>
<td>426.2600</td>
<td>12.9200</td>
<td>187.0700</td>
<td>35.7200</td>
<td>0.5899</td>
<td>1.9500</td>
<td>33.4899</td>
<td>692.6399</td>
</tr>
<tr>
<td></td>
<td>112122</td>
<td>275.7896</td>
<td>20.1900</td>
<td>142.3799</td>
<td>23.4600</td>
<td>4.8900</td>
<td>30.2399</td>
<td>7.3999</td>
<td>1.2100</td>
<td>1.1100</td>
<td>7.5000</td>
<td>37.4099</td>
</tr>
<tr>
<td></td>
<td>112232</td>
<td>725.4294</td>
<td>63.2999</td>
<td>338.9000</td>
<td>42.8000</td>
<td>14.3699</td>
<td>95.1199</td>
<td>26.2500</td>
<td>2.1999</td>
<td>2.2899</td>
<td>27.1099</td>
<td>113.0900</td>
</tr>
<tr>
<td></td>
<td>112422</td>
<td>243.5297</td>
<td>17.96</td>
<td>122.02</td>
<td>20.8500</td>
<td>4.4299</td>
<td>26.0700</td>
<td>7.1300</td>
<td>1.2100</td>
<td>1.1099</td>
<td>7.1700</td>
<td>35.5799</td>
</tr>
<tr>
<td></td>
<td>211121</td>
<td>8.3996</td>
<td>0.6800</td>
<td>2.0000</td>
<td>0.7199</td>
<td>0.5300</td>
<td>0.7500</td>
<td>0.6500</td>
<td>0.8699</td>
<td>0.5499</td>
<td>0.6099</td>
<td>1.0400</td>
</tr>
<tr>
<td></td>
<td>311121</td>
<td>22.6198</td>
<td>5.3300</td>
<td>1.0900</td>
<td>1.0600</td>
<td>0.6599</td>
<td>2.8900</td>
<td>2.6900</td>
<td>1.2300</td>
<td>3.0699</td>
<td>5.5500</td>
<td>1.0500</td>
</tr>
</tbody>
</table>
Table 4.4: This table is a continuation of table 4.3 and contains the problem instances in group C.

<table>
<thead>
<tr>
<th>ID</th>
<th>Configuration</th>
<th>Total runtime (s)</th>
<th>Runt. prob 1 (s)</th>
<th>Runt. prob 2 (s)</th>
<th>Runt. prob 3 (s)</th>
<th>Runt. prob 4 (s)</th>
<th>Runt. prob 5 (s)</th>
<th>Runt. prob 6 (s)</th>
<th>Runt. prob 7 (s)</th>
<th>Runt. prob 8 (s)</th>
<th>Runt. prob 9 (s)</th>
<th>Runt. prob 10 (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>111111</td>
<td>692.0697</td>
<td>0.0100</td>
<td>0.0099</td>
<td>15.0300</td>
<td>0.0199</td>
<td>0.0000</td>
<td>0.0100</td>
<td>674.1900</td>
<td>0.1000</td>
<td>2.6900</td>
<td>0.0099</td>
</tr>
<tr>
<td></td>
<td>111121</td>
<td>1.6500</td>
<td>0.1699</td>
<td>0.1600</td>
<td>0.1599</td>
<td>0.1699</td>
<td>0.1600</td>
<td>0.1599</td>
<td>0.1699</td>
<td>0.1700</td>
<td>0.1700</td>
<td>0.1600</td>
</tr>
<tr>
<td></td>
<td>112112</td>
<td>555.3199</td>
<td>0.0099</td>
<td>0.0099</td>
<td>1.2400</td>
<td>0.0099</td>
<td>0.0200</td>
<td>0.0099</td>
<td>526.5800</td>
<td>0.0299</td>
<td>27.4000</td>
<td>0.0099</td>
</tr>
<tr>
<td></td>
<td>112212</td>
<td>5.8297</td>
<td>0.0099</td>
<td>0.0099</td>
<td>0.0699</td>
<td>0.0199</td>
<td>0.0199</td>
<td>0.0099</td>
<td>0.2300</td>
<td>0.0299</td>
<td>5.4099</td>
<td>0.0199</td>
</tr>
<tr>
<td></td>
<td>112312</td>
<td>6.0698</td>
<td>0.0099</td>
<td>0.0099</td>
<td>0.0699</td>
<td>0.0199</td>
<td>0.0199</td>
<td>0.0099</td>
<td>0.2300</td>
<td>0.0299</td>
<td>5.6500</td>
<td>0.0199</td>
</tr>
<tr>
<td></td>
<td>112412</td>
<td>0.8597</td>
<td>0.0000</td>
<td>0.0099</td>
<td>0.1200</td>
<td>0.0199</td>
<td>0.0199</td>
<td>0.0099</td>
<td>0.3299</td>
<td>0.0299</td>
<td>0.3199</td>
<td>0.0099</td>
</tr>
<tr>
<td></td>
<td>112222</td>
<td>3.2599</td>
<td>0.3099</td>
<td>0.3099</td>
<td>0.3100</td>
<td>0.3099</td>
<td>0.3099</td>
<td>0.3000</td>
<td>0.3099</td>
<td>0.4700</td>
<td>0.3199</td>
<td>0.3199</td>
</tr>
<tr>
<td></td>
<td>111122</td>
<td>9.3197</td>
<td>0.0199</td>
<td>0.0199</td>
<td>0.1100</td>
<td>0.0299</td>
<td>0.0299</td>
<td>0.0199</td>
<td>0.2200</td>
<td>0.0599</td>
<td>8.7899</td>
<td>0.0199</td>
</tr>
<tr>
<td></td>
<td>311212</td>
<td>0.3598</td>
<td>0.0199</td>
<td>0.0200</td>
<td>0.0199</td>
<td>0.1399</td>
<td>0.0599</td>
<td>0.0200</td>
<td>0.0199</td>
<td>0.1999</td>
<td>0.0199</td>
<td>0.0199</td>
</tr>
<tr>
<td></td>
<td>211212</td>
<td>9.4992</td>
<td>0.0299</td>
<td>0.0099</td>
<td>0.1090</td>
<td>0.0399</td>
<td>0.0299</td>
<td>0.0300</td>
<td>0.2299</td>
<td>0.0600</td>
<td>8.9299</td>
<td>0.0299</td>
</tr>
</tbody>
</table>
Chapter 5

Discussion

In this chapter, the results reported in the previous chapter are discussed and evaluated. The first part concentrates on an evaluation of the framework in terms of the objectives set during the project whereas the second section compares the results to previous research. This chapter ends with a conclusion and summary of the limitations of the framework and possible extensions to be made in future.

5.1 Evaluation of Results

5.1.1 Modularity

The main issues found during the evaluation of the framework were that the learning module depends on the conflict handling module and that the built-in operators in ECL\`PS\(\text{e}\) have their own propagation methods which cannot be controlled by the user.

While the dependency between conflict handling, learning and explanation algorithms is not a fatal flaw - any combination of techniques results in a working search algorithm - it still can be seen as an imperfection. A possible solution to overcome the first issue has been presented in 3.1.9. In the proposed revision of the structure, the framework module calls the explanation algorithm and passes the results on to the conflict handling and learning algorithms as they require. Using the initialisation phase, the framework can know whether any explanations have to be computed at all.

The issue about the propagation of built-in operators can be overcome in theory by implementing a new propagation algorithm. Doing this for a simple consistency check such as simple backtracking does would be easy. However, if more powerful methods are required, it would take a substantial effort and it seems inadequate to ignore the built-in features of the ECL\`PS\(\text{e}\) language. In fact, as it has already been stated in the previous chapter, it is not sensible under normal circumstances to modify the propagation methods associated with certain operators. The propagation provided by ECL\`PS\(\text{e}\) for these operators and goals is tailored to the specific constraints. Therefore, the lack of control is balanced by the opportunities that arise from having specific propagation methods for certain goals.

The incompatibility between conflict monitoring as an explanation algorithm and propagation methods can be ignored since this only concerns a single technique in the explanation module and implementing other methods is not overly difficult.

In summary, the framework structure is adequate and well suitable to combine a large number of different techniques. An implementor of more techniques only has to obey to the interface definitions and a user of the system can change the configuration of the framework completely by simply changing numbers in the calling query entered in ECL\`PS\(\text{e}\). Due to the fact that the framework structure has been chosen with respect
to a number of current techniques which enhance constructive searching and since the
techniques which were implemented fit into the structure directly, it can be expected that
more techniques can be added without major difficulties. The framework therefore gives
users and researchers significant flexibility and independence of techniques. The goal of
modularity has been achieved.

5.1.2 Performance

Random Problems

The tests on randomly generated problems (4.1) show that the overhead incurred when
using the framework in standard backtracking ranges from very small values (11%) to
extremely large value (742%) depending on the problem type. The former result is ex-
traordinarily good since the acceptable overhead of 100% is beaten by factor 9. The latter
result however is unsatisfactory and in that case the framework failed to achieve the ob-
jective. The obvious question is which characteristics of the problems cause such a big
difference.

Observing the results table, one can see that all problem types which an overhead of
less than 100% had 8 or more values per variable. The problem class with 10 variables
had 10 values per variable as well but the overhead was 147% and therefore higher than
the objective. In contrast to the other problems with 8 or more values, this problem class
has a fairly low number of constraints and a higher degree of tightness. The problem class
with 10 variables has 56% of all possible constraints and the relative tightness is 70%.
Most other classes with an overhead of less than 100% have a high relative number of
constraints. As a result, it can be induced that the framework performs competitively for
problems with 8 values or more per variable if the tightness is not too high or the relative
number of constraints is high enough.

The by far worst results (overhead 488% and 742%) were obtained for problems with
only two values per variable. The problem with the higher overhead has a very small
amount of constraints (only 4%) combined with a tightness of 50% which agrees to the
interpretation above saying that a high relative number of constraints and a lower degree of
tightness is helpful for good performance. A possible explanation for the high overhead for
problems with few variables is that in those cases, the exploration algorithm for choosing
a value is often called for the first time for each variable rather than being resatisfied
most of the time. In the current implementation, the exploration algorithm retrieves the
domain of that variable as a list and repeatedly calls the “member/2” predicate. The
standard backtracking algorithm instead uses the single predicate “indomain/1”. For the
framework it is necessary to retrieve the value for a variable before the instantiation is
attempted because the interfaces prescribes that if the instantiation fails, the value has to
be set tentatively. Relaxing this condition - by using the initialisation phase to determine
whether the tentative instantiation is actually required - might reduce the overhead.

An interesting result is that the average runtime difference per labeling step between
the framework and the direct implementation remains fairly constant for problems with
more than 6 variables. The average time spent per labeling step obviously depends on the
number of values since more values mean that more options can be tried before another
labeling step begins or backtracking is done. The fact that, no matter what the mean
time per labeling step is, the framework seems to need an almost constant amount of
approximately 0.2-0.3 milli seconds per labeling step more than the direct implementation
indicates that the performance can be predicted fairly well if the number of labeling steps
and the runtime of a standard algorithm is known. Another indication for good predi-
catibility is that the standard deviation of the runtimes for the framework and the direct
implementation are fairly similar if the different mean runtime is taken into consideration.
5.1. EVALUATION OF RESULTS

The proportion of standard deviation divided by mean runtime is quite similar for both in all cases.

In summary, it can be concluded that the framework performs with almost the same efficiency as a direct implementation in a single node in the search space where different values for a variable are tried. When a new variable is chosen or when backtracking occurs, the framework is slower due to the number of features that might be activated in such a case. In terms of the objective to reach an overhead of 100%, the objective has been achieved in a number of cases where the problem parameters (values, constraints and tightness) are appropriate but in other cases, especially problems with binary variables, the objective could not be achieved. It is likely that improvements on the implementation result in a qualitative change of the overhead towards the better.

Job Shop Scheduling

The tests show that the runtime overhead of the framework ranges between 100% and 200% about three quarters of the time which is more overhead than the objective prescribes. However, the worst overhead is 592% and therefore less than the highest overhead encountered using the randomly generated problems.

The runtimes per labeling step are usually much higher than the ones for uniform random problems 4.1, which is because of the propagation which runs in the background. The problem characteristics listed (#jobs, #machines and #operations per job) are not sufficient to describe how difficult a job shop scheduling problem is, since for the instances la01 to la04, these parameters are the same while the overhead ranges from 171% to 592%. Of course, the difficulty of a single problem depends greatly on chance since the (optimal) solution can be at any point in a large search space. Therefore, although two problems have the same structure, the solution of the first one might be found very early while the solution of the second one might be found very late.

It becomes apparent that all problem instances for which a high overhead of more than 200% was incurred can be solved using few labeling steps only - that is less than 5000. Problem instances which require more than 100000 labeling steps always had an overhead of less than 200%. It is an unexpected outcome that the overhead for the arbitrarily generated problems was always well in the around range of 100%, with only one deviation to 160%, independent of the number of required labeling steps. One possible interpretation is that the problem instances taken from the problem database by Mattfeld and Vaessens (n.d.) have been designed with particular focus on creating certain properties which make them more interesting problems. Another reason for the difference might be the difference in the number of jobs used in both types of problems. The self-made problems have a lower number of jobs than most database-problems. Even though a high number of jobs does not imply a high overhead, an possible conclusion might be that a high number of jobs is likely to yield a higher overhead.

The runtime difference per labeling step ranged between 3 ms and 6 ms for most problem instances which is longer by factor 10 than for the uniform random problems. Of course, the propagation done in job shop scheduling problems causes for the difference. However, since the runtime difference is fairly balanced again, this could be used for runtime predictions if the runtime of a standard backtracking algorithm with the same propagation and the number of labeling steps is known.

The high overhead of more than 500% for one problem lead to an investigation of the reasons for this. The bottleneck which slowed down the framework so greatly was identified to be one of the subroutines used in the framework in every labeling step. Changing this subroutine lead to enormous speed-ups for the jobshop scheduling problems.
The performance achieved on job shop scheduling problems was increased significantly, reducing the overhead from 592% to 40% in one case. The former number is a quite unsatisfactory value, the latter a result which is better than aimed at. For all jobshop scheduling problems, the overhead was always between 10% and 40%. The average overhead on job shop scheduling problems has been reduced from 214.89% to 24.96%.

After the great improvement of the results on jobshop scheduling problems, the tests for some of the random problems had been repeated as well to see whether the improvement would affect those overheads as positively. Unfortunately, the overheads did not decrease as dramatically as the ones for the jobshop problems. The problems where the framework suffered from an overhead of 740% could be solved with an overhead of 720% using the improved version. Some overheads in the range between 200% and 500% were decreased a bit more than only by 20%. Some of the problems where the overhead was low, that is below 100%, were solved with a slightly increased overhead after the change. Summarizing the effect of the change on the random problems, the high overheads were reduced slightly while some of the low overheads were increased insignificantly. The average overhead did not change remarkably. The fact that high overheads were reduced counterbalances the minor loss of efficiency for problems with low overheads. The change of the framework has a slight positive effect, but not nearly as dramatically as the effect on job shop scheduling problems.

To summarize the job shop scheduling results, the objective to keep the overhead below 100% has not been achieved in the original version of the framework although for some instances, the overhead was not significantly higher than that. The worst case encountered still is better than the worst overhead found for random problems which could increase the predictability of the framework for such problems. Using the improved version of the framework, the overheads were significantly lower, that is below 40% in all cases. This shows that investing a little effort on improving the framework can have great effects. The results achieved in the newer version were far more than just satisfactory.

Different Configurations

The tests using different configurations of the framework show that different problems are solved most efficiently with different configurations, as is to be expected. The total time used to solve 10 problems in each configuration (see table 4.3) can be used as an indicator for the general performance of a configuration on the given set of problems. However, the same configuration can be good on the average for all 10 problems, but inefficient for some particular problems. Therefore, studying the runtimes for singular problems is useful.

In general, the framework solves tasks a lot faster if propagation is used with a speed-up factor between 5 and 100 for many problems. The problem sets A and B show that propagation alone is usually the best choice for these problems. Conflict handling (together with QuickXPlain) is slower by factor 10 on the average on these problem sets. Using learning techniques in addition improves the performance again, but the low runtimes of plain propagation are not reached. Using stronger propagation than the weak setting increases the runtimes for some problems significantly, only in a few instances could the runtime be decreased. Using the first-fail heuristic to choose variables, the performance could usually be increased further by a factor of 2 approximately. Using the most-constrained heuristic increased the runtimes on the average. Two tests in problem set B compare the performance of conflict handling alone using conflict monitoring and QuickXPlain, the latter being slower in these cases.

Problem set C is different from A and B in three matters. Firstly, using propagation was slower on the average than using conflict handling and strong learning in combination. Using QuickXPlain instead of conflict monitoring resulted in better performance here, too. The best result of all configurations was achieved using the most-constrained heuristic.
5.2. COMPARISON WITH PREVIOUS RESEARCH

which again deviates from the results of problem set A and B. It is interesting to note that using the weak and medium learning did not result in better performance here on the average. Observing the individual problems, however, shows that weak or medium learning is better than the weak propagation on most of these problems.

In conclusion, the tests show that the framework can be used to solve problems exceedingly faster when special techniques are switched on. As can be expected, different problems require different configurations to be solved in the fastest way possible. Each technique can be used for some problem to solve it most efficiently, including different levels of learning strength, which refers to the amount of no-good merging done.

5.2 Comparison with Previous Research

The main issue with previous work is that some of the modern techniques like learning are not supported or at least difficult to include in search algorithms using the frameworks developed by Beck and Fox (1998) or Jussien and Lhomme (2003). The framework developed in this project offers a much higher degree of flexibility including learning algorithms. The module structure is more detailed than in the framework by Jussien and Lhomme (2003) which gives the user more opportunities for optimisation and the successful implementation of the framework proves the soundness of the concept.

A comparison on performance is hardly possible, because the former framework is not implemented but merely a conceptual design and the latter has not been tested on the same problems on the same machine. The flexibility in terms of local search and hybrid approaches which are offered by the SALSA language (Laburthe and Caseau, 1998) or the ToOLS library (de Givry and Jeannin, 2003) is not given by the framework on the one hand, but this is not the scope of the project. On the other hand, the SALSA language and ToOLS library offer much less support for look-back algorithms, computation of explanations and learning.

The framework developed in this project comes with a number of implemented algorithms for each module so that users can solve problems directly without any further work and use the existing algorithms as a base for future development.

5.3 Conclusion

The structure proposed for partitioning search algorithms has proven to be adequate to incorporate a large number of techniques with a minimal number of interdependencies in constructive search algorithms. Several techniques were implemented and tested on different problems. In the course of the project, the standard CSP formalism was extended to handle arbitrary constraints and to make use of the operator- and goal-specific propagation provided by ECL/PS.

The performance of the framework is acceptable or better for problems with sufficiently many values per variable. For problems in which the variables only have binary domains, the overhead generated by the framework still is undesireably high. Future improvements might qualitatively improve this matter. The performance achieved on solving job shop scheduling problems is very good if the improved version of the framework is used. Even though only a small test sample set was used, the results with the improved version promise a very good and competitive performance of the framework on real world problems. Without the improvement, the objective could not be achieved, however.

The effect of using different configurations to solve problems has been shown. Propagation is one of the most powerful mechanisms to improve speed while other techniques can improve the performance further in some cases or even beat propagation in terms of their positive effect on performance. Combining many techniques does not necessarily
lead to a faster search algorithm on some problems because the overhead generated by some techniques cannot be outweighed by the additional benefits they provide if other techniques are active as well.

Even though the overhead could not be kept lower than 100\% all of the time, the project was successful. The great amount of flexibility given to users is likely to outweigh the drawback in terms of performance. The initialisation phase which is used to compute an appropriate setup given a combination of algorithms provides the largest contribution to decrease the overhead. Making this part more sophisticated and at the same time implement optimised versions of the special techniques is likely to result in significantly better results. Almost any technique can be combined with any other in another module and the framework structure allows for implementing further techniques rather easily.

5.3.1 Review on Deliverables
The deliverables as stated in the research proposal were:

\begin{itemize}
\item implementation of the generic search framework
\item implementation of at least two algorithms for each of the modules of the generic search algorithm
\item a performance analysis for several feasible combinations of module-algorithms
\item documentation for all program code
\end{itemize}

The generic framework has been implemented completely. The number of algorithms implemented for each module are as follows:

\begin{itemize}
\item Exploration(Variable) : 3
\item Exploration(Value) : 1
\item Propagation : 4
\item Conflict Handling: : 2
\item Explanation : 2
\item Learning : 3
\end{itemize}

Since for propagation, the built-in features of ECL\textsuperscript{PS} have been used, the high number of algorithms was easy to achieve. Having only one algorithm for choosing a value for a variable is counterbalanced by the larger number of techniques for other modules.

The performance analysis for several combinations of techniques has been done in a brief manner. The reason for this is that the performance of different algorithms was regarded to be of little significance to evaluate the framework itself. It is deemed more important to compare the framework to different ways of solving problems and therefore the overhead tests have been conducted.

Finally, all program code has been commented extensively. In conclusion, the requirements given by the deliverables have been fulfilled.

5.4 Limitations
The framework has been developed to support constructive search algorithms for solving Constraint Satisfaction Problems. For that reason, different approaches like repair based methods or genetic algorithms are not supported and are probably very difficult to implement using the framework. Definitely the features offered by the framework cannot be applied to solve CSPs using such methods.

The computational overhead remains the only drawback of the framework. This could limit its practical applicability slightly although the theoretical range of applications of the framework is as large as intended. Further work (5.5) might reduce the overhead further.
5.5. FUTURE WORK

Even though a lot of different techniques has been implemented using the framework structure, it is possible that other methods either currently existing or being developed in future do not easily fit into the structure even though the design aimed at avoiding such difficulties. This question obviously can only be answered by time.

The framework does not offer its own simplified language for solving CSPs such as some other frameworks do and as a result, someone trying to implement new algorithms needs to be proficient enough in logical programming and the features provided by ECL\textsuperscript{i}PS\textsuperscript{e}.

Overall, the framework has no limitations which were not to be expected, except for a higher overhead than intended. In the second chapter 2.1.2, the CSP model and its common limitations have been explained. The framework provides the facilities to process any problem included in this formalism. In addition, constraints are treated more generally than the original model prescribes. Instead of tuples of consistent values, constraints can be any predicate. The downside of this extension is of course, that it lies with the user to create and use only appropriate constraint goals, that is goals which when called to not create an infinite loop or other undesired effects.

5.5 Future Work

The framework can be improved and/or extended in several ways. These are issues that are not vital for the framework to work but which could improve its performance and user ergonomy.

Most important, the performance can be improved. Neither for the framework body nor for implementations of special techniques has a considerable effort to design an optimal implementation in terms of performance been made. The coding was done with the objective to make the algorithms perform their tasks correctly, not necessarily in the fastest and most efficient way. Therefore it can be expected that attempts to improve the performance will be successful and a significant decrease in the overhead can possibly be achieved. In fact, the slight improvement which resulted in dramatic changes of the overhead shows that little effort can have significant results.

Related to improving the performance, more work should be done on investigating what exactly causes the high overhead incurred when solving random problems with few values per variable. Superficial tests, which have been done already, do not suggest a particular module or part of the framework to be responsible for the overhead.

Next, several more techniques can be implemented to test the framework. The capability of being used for future research on search algorithms is of course one of the main purposes for the framework to exist. Implementing more algorithms would, however, also strengthen or weaken the claim that the structure of the framework is appropriate to manage any technique used for constructive search.

Testing the framework in different configurations and comparing those to direct implementations of the same algorithms would greatly increase the significance of the overhead tests. Testing the framework on more job shop scheduling problems and other real world problems would add more to the confidence into the framework and its capabilities and limitations.

The user interface of the framework is simplistic at best currently. Users will find it inconvenient to use it to solve problems. For that reason, some effort should be invested into designing a proper graphical user interface where loaded problems can be managed, different techniques can be combined by selecting items from lists and results are shown in a way easier to understand than what ECL\textsuperscript{i}PS\textsuperscript{e} does by itself. The current version of the framework only supports numerics values for variables which is not a limitation by itself. However, it would be more convenient to be able to use other terms like names as
values so that a solution can be interpreted more easily. Such a feature is provided by an ECL\textsuperscript{i}PS\textsuperscript{e} library but it still would have to be added to the framework.

The representation module could be extended to handle different formats of problem definitions. For the randomly generated problems and the job shop scheduling problems, a translating program was written to rewrite problems in the form the framework understands with the current representation module. If standard forms of problem definition exist, the representation module could be extended to cope with these, thus easing the effort of importing problems.

Auxiliary variables which can support designing a constraint model for problems are not supported by the framework yet. Treating these is slightly different from problem/decision variables since they are not assigned a value by exploration, but only by propagation usually. However, if the propagation is not strong enough or a unique solution is not given, some labeling must be done to instantiate all variables and completely solve the problem. The user should have some control to define which value - like the smallest one in the domain - the auxiliary variables should take at the end of the labeling.

An interesting idea to improve searching is given in (Sakkout et al., 1996), where the authors propose to adaptively change the strength of propagation algorithms during the search depending on the internal state. This idea could be extended to adaptively change any technique used in a configuration. Definitely, providing such a flexibility on top of what is already provided could increase the overhead but it is also possible that such methods would reduce the total time used for solving large and complicated problems. The current version of the framework does not support adaptivity directly.

In several systems for solving constrained optimisation or constraint satisfaction problems, the exploration phase consists of one technique to narrow down the set of candidates to choose as the next variable to assign followed by a second technique to break ties. More general, one could think of a pipeline of techniques or even more general, a system of techniques which choose a variable or a value. Since the user has the full programming language of ECL\textsuperscript{i}PS\textsuperscript{e} at his disposal to write an exploration algorithm, he can in theory implement any of these systems. However, it would be preferable if the framework itself would support such pipelines of techniques. The changes necessary involve changing the interface to exploration algorithms slightly so that not a single variable but a list of these is returned. Until the list is reduced to a single element, more techniques taken from the pipeline could be used.
Appendix A

Test Results
Figure A.1: Problem #1: 5 Variables 20 Values

Figure A.2: Problem #2: 6 Variables 40 Values

Figure A.3: Problem #3: 10 Variables 10 Values
Figure A.4: Problem #4: 15 Variables 12 Values

Figure A.5: Problem #5: 18 Variables 2 Values

Figure A.6: Problem #6: 18 Variables 4 Values
APPENDIX A. TEST RESULTS

Figure A.7: Problem #7: 18 Variables 8 Values

Runtime histogram for random problem #7 - Framework

Runtime histogram for random problem #7 - Labeling

Figure A.8: Problem #8: 20 Variables 2 Values

Runtime histogram for random problem #8 - Framework

Runtime histogram for random problem #8 - Labeling

Figure A.9: Problem #9: 22 Variables 5 Values

Runtime histogram for random problem #9 - Framework

Runtime histogram for random problem #9 - Labeling
Figure A.10: Problem #1: 5 Variables 20 Values  #2: 6 Variables 40 Values

Figure A.11: Problem #3: 10 Variables 10 Values  #4: 15 Variables 12 Values

Figure A.12: Problem #5: 18 Variables 2 Values  #6: 18 Variables 4 Values
Figure A.13: Problem #7 18 Variables 8 Values    #8: 20 Variables 2 Values

Figure A.14: Problem #9: 22 Variables 5 Values
References

(n.d.).
URL: http://www.lirmm.fr/bessiere/generator.html


URL: citeseer.csail.mit.edu/degivry03tools.html


URL: citeseer.ist.psu.edu/laburthe98salsa.html


Mattfeld, D. C. and Vaessens, R. J. (n.d.).
URL: http://people.brunel.ac.uk/mastjhb/jeb/orlib/jobshopinfo.html
REFERENCES


