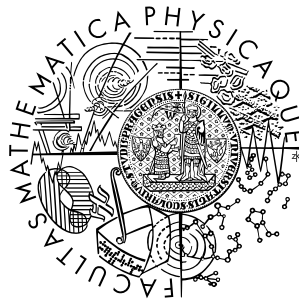


Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Michal Tuláček

Constraint solvers

Department of Theoretical Computer Science and
Mathematical Logic

Supervisor: Doc. RNDr. Roman Barták, Ph.D.

Study programme: General Computer Science

2009

`$Id: bc_prototype.tex 144 2009-07-26 12:36:50Z guffy $`

I wish to thank my supervisor Dr. R. Barták for advices and patience. My thanks also belongs to V. Kopal who helped me with correction of the English text. And last but not least my thanks belong to my coach J. Baxa who clearly stated that failure is not an option and then enforced this policy.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

I declare that I wrote my bachelor thesis independently and exclusively with the use of the cited sources. I agree with lending this thesis.

In Prague,

Michal Tuláček

Obsah

Název práce: Řešiče omezujících podmínek

Autor: Michal Tuláček

Katedra (ústav): Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Doc. RNDr. Roman Barták, Ph.D.

e-mail vedoucího: bartak@kti.mff.cuni.cz

Abstrakt: Řešič omezujících podmínek je specializovaný software, určený k řešení problémů popsanych omezujícími podmínkami. Práce podává přehled řešičů a vybrané z nich testuje z pohledu uživatelské přístupnosti a rozsahu problémů, které lze modelovat.

Klíčová slova: omezující podmínky, řešiče, benchmarky

Title: Constraint solvers

Author: Michal Tuláček

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Doc. RNDr. Roman Barták, Ph.D.

Supervisor's e-mail address: bartak@kti.mff.cuni.cz

Abstract: Constraint solver is a specialized software used to solve constraint satisfaction problems. Work lists solvers and some of them tests using the criteria of user accesibility and variety of problems which can be modeled.

Keywords: constraint satisfaction problems, constraint solvers, benchmarks

Kapitola 1

Introduction

In this thesis we compare several constraint solvers from a perspective of a user who is not experienced in a constraint programming. We focus on easiness of a learning process of each solver and we measure performance by using benchmarks which test various aspects of examined systems.

1.1 Motivation

The constraint programming is a programming paradigm which uses constraints to a describe solution rather than to program a way of achieving such solution. Constraint can be any condition which can be asserted as true or false – $X < Y$, Billy is older than Johnny, $Z = 5$, etc. As an example of a problem which can be solved using the constraint programming we use Sudoku puzzle. Sudoku is a worldwide known logical problem which is easy to explain, its difficulty can be scaled and one does not need previous training to solve Sudoku. It makes the problem easier to understand for many people and, therefore, it is very popular. Simple rules of Sudoku are: There is a given table of size nine times nine. Every field of table contains number in range one to nine. In each column all the numbers are different (this enforces that every column contains all numbers in range one to nine). In each row there are also all numbers different. Finally the same rule which applies for columns and rows also restricts three times three sized squares which are in the puzzle marked using bolder lines. The Sudoku is prefilled with a couple of values. These values help at the beginning of solving and the difficulty can be adjusted by their count and placement.

The way how to describe Sudoku puzzle in the constraint programming

is very straightforward. We define following constraints:

1. There are 81 variables which can contain values in range 1 to 9. We arrange them into a two-dimensional array with size 9×9 . (The Sudoku is a table sized 9×9 containing values in a range 1..9)
2. For all i in 1..9 is true: All values of $s_{i\bullet}$ are different (Values in each row are different)
3. For all j in 1..9 is true: All values of $s_{\bullet j}$ are different (Values in each column are different)
4. For each square is true: For all k, l such that k, l is in square values of s_{kl} are different (Values in each square are different)
5. For all prefilled values: $q_{mn} = V$ if and only if field with placed in column n and row m is prefilled and contains V .

These constraints fully describe the Sudoku puzzle problem and as the reader can see do not differ from the commonly known rules. A person solving Sudoku puzzle can use many techniques starting with randomly filling the table and looking if this is a good solution (the algorithm using this technique is called GAT – Generate and Test) and ending with generating all possible fillings and correcting the solution if something fails (this algorithm is called backtracking). The first approach can miss a correct solution. Since the second approach systematically searches the possible solutions it has to result into the correct solution; however, it can last enormous time to complete it (even on supercomputer). The secret of a successful solution is in the fact that not all numbers can be filled in a specific field. If there is prefilled value 8 at position [6,7] it means that in row 6 and in column 7 cannot be another number 8. And because of constraint (4) there also cannot be 8 in the right middle square. A person who does these observations usually writes into the destination field all possible values and as an examination of the puzzle progresses there are less and less possibilities to fill in. In an easy Sudoku after this examination there is at least one field which can be filled with only one number. After filling all such fields the solving continues in the same way until the entire table is filled. A program which use constraint programming solves it in the same way. For each variable it remembers the range of possible values (we will call it a domain). Before the program starts searching of solution it tries to eliminate as many values from the domain as

possible. It can reveal that the problem does not have a solution (if there is a variable with an empty domain) before a backtracking. It is no surprise that in a user guide to Choco system it is stated "if you know Sudoku, then you know the constraint programming."

1.2 Constraint programming

The constraint programming indeed consists of more techniques but generally it works just like the person solving Sudoku described in the previous paragraph. All constraint satisfaction problems (CSP) can be transformed in such a way that it contains only binary constraints. Common representation of the CSP is the multigraph where nodes are variables and the edges constraints on them. The value of a variable's domain is *supported* if there is not a constraint which collides with such a value. We can define *node consistency* if all values of domain of a node x satisfies all constraints $c(x, x)$ (unary constraints). The problem is node consistent (NC) if all nodes in the problem are node consistent. Similarly, the edge $e(x, y)$ is arc consistent if for all values in D_x exists a value of y such that the constraint is satisfied. This definition does not assure that if the edge $e(x, y)$ is arc consistent then the edge $e(y, x)$ is arc consistent too. CSP is arc consistent (AC) if all edges are arc consistent in both directions. The arc consistency, however, does not guarantee that the problem has solution. If the problem is not arc consistent then it has no solution for sure. For example let have variables X, Y, Z with domains $\{0, 1\}$ and constraints $X \neq Y, Y \neq Z$ and $X \neq Z$. The problem is arc consistent but it has no solution.

The general way of solving of the CSP is to transform the domains of variables to ensure the arc consistency of the problem. After the problem is in stable state (last iteration of arc consistency algorithm have not changed any domain) we test if the problem is solved. The problem is solved if each value has one sized domain. If there is an empty domain after the arc consistency algorithm we declare the problem as unsatisfiable. This part of algorithm is called *propagation*. If we have not found the solution in propagation phase we continue with *distribution* phase. In distribution phase we add an additional constraint c to the problem P and rerun the propagation on two new problems, $P \cup \{c\}$ and $P \cup \{\neg c\}$. If there exists a solution it has to be contained in at least one of the two new problems. For the usually used constraint we choose one variable x and value v from its current domain D_x . The constraint c then can be $x = v$ or $x < v$. The choosing algorithm of the

Obrázek 1.1: The MAC algorithm

```
1 procedure labelling(V,D,C)
2   if all variables from V are assigned then return V
3   select not-yet assigned variable x from V
4   for each value v from Dx do
5     (TestOK, D') := consistent(V,d,C + {x=v})
6     if TestOK = true then
7       R := labelling(V, D', C)
8       if R <> fail then return R
9   end for
10  return fail
11 end
```

proper variable x is important. Often we pick the variable with the smallest domain because it leads to possible failure more quickly. Usually the solver offers choices of the algorithm to suite all needs. This algorithm is called maintained arc consistency or MAC. Its schematic source code is in figure 1.1. The code is cited from [?]. More detailed explanations of algorithms used in CSP solving can be found in [?].

In reading previous paragraphs, a reader could think that the constraint programming is used only as an academic toy for solving Sudoku and for other applications which are useless in a real life. In fact the constraint programming is used in various applications. A few examples of many contain scheduling, an image recognition, financial modeling, planning, vehicle routing, a configuration, computer networks and bioinformatics. The constraint programming was also successfully used at NASA in Deep Space 1 experiment. Deep Space 1 was a space probe using 12 cutting-edge technologies which were never tested in space before. One of these technologies was a remote agent used to plan actions of a space vehicle while only general commands were sent to agent. Agent used for planning constraint solver [?].

1.3 Constraint solvers

A programmer who wants to solve problem using constraint programming can use in their algorithms ideas described in previous paragraphs or use specialized software, a constraint solver. Constraint solver is a system which uses constraint programming techniques to solve a given problem. There are many solvers available both commercial and freeware. A short list of available systems can be found in table ???. More detailed list is maintained by Roman Barták in On-line guide to constraints programming at [?].

1.4 Related work

There exists a paper "A Comparative Study of Eight Constraint Programming Languages Over the Boolean and Finite Domains" by A. Fernandez and others [?] which covers similar area; however, this paper does not focus on the user experience with the solvers. They compared solvers by their performance on various benchmarks and discussed an implementation of self referential quiz in each solver. We are not benchmarking the same sets of solvers and we do not use similar methodology which means this thesis conclusions cannot replace or update results from [?].

Every year there is held the International Constraint Solver Competition where authors of solvers can compete. The solvers can be submitted in two categories – complete and incomplete. Complete solvers can prove that the instance of problem is satisfiable or not (or find and prove the optimum). As stated in the rules [?] for each solver there is a *Boolean capability vector* which indicates which constraints the solver can handle. Solvers with the same capability vector can be naturally compared. Solvers with different capabilities can be compared on instances which belong to the intersection of their capabilities, provided it is non-empty. During the competition the solvers are run in the sandbox environment on a Linux cluster. The task is to compute solution of as many benchmarks as possible in the smallest time amount.

There exists a library [?] of the constraint satisfaction problems which can be used in benchmarking and comparing of solver capabilities. The library contains varying problems in several fields – optimisation problems, combinatorial problems and so on. We encourage the reader to try to implement several problems in the chosen solver as a part of learning of modelling in the solver.

1.5 Outline of the thesis

We described our motivation for this thesis and listed some constraint solvers. In the second chapter we define methodology used to examine some of mentioned solvers. The examination consists of two parts – performance tests and usability tests. In third chapter we will define benchmarks used to performance tests. The fourth chapter describes in details each solver, mentions a little from their history but mainly focuses on usability and easiness of learning and using the solver. In the fifth chapter we discuss the perfor-

mance tests results and compare the solvers. Finally, in the sixth chapter we state a conclusion of the whole examination process and present a decision graph "which solver use in which situation".

Kapitola 2

Methodology

In the introduction we explained what constraint solvers are and presented several examples of them. In the rest of the thesis we focus on seven of them – ILOG OPL, SICStus Prolog, Mozart, ECLⁱPS^e, Gecode, Choco and Minion. The first two of them are professional commercial solutions and the others are freely available open source products. A purpose of this thesis is to help new users with choosing of the right solver. Therefore we study difficulty of learning and using of each solver and their performance and abilities. We test solvers which use various programming languages and paradigms. An imperative paradigm is represented by a C++ library Gecode and a Java library Choco. Users experienced in a logical programming might find interesting SICStus Prolog or ECLⁱPS^e. Mozart is implementation of Oz, a multi-paradigm programming language. With these solvers user can use their current experience and just learn an API of the constraint library. The other solvers are configured by a solver specific problem description language. This fact is both advantage and disadvantage. As for disadvantage, we must accept that user cannot use their experience with existing programming languages and has to learn new concepts; however, a specialized language for describing constraint problems can be more accessible for users who do not have any programming experiences but they need to solve the given problem. A general overview of the examination follows.

First, we examine all solvers from a perspective of a user experienced in the given programming language but inexperienced in using of the solver. In case of OPL and Minion we expect that user has general computers knowledge and is able to describe a given problem in constraints. The first examination tries to answer a question how difficult it is to learn to use

the solver. We model problems described in the third chapter and look for constraints which cannot be modelled and we describe possible solutions.

Secondly, a quality of documentation is also important criterion. A solver can be the best of all, but it is useless if the user cannot understand the usage. The quality of documentation is perceived subjectively and cannot be measured exactly. This means that any evaluation is only informational, although it should be considered. As a documentation we accept a user guide as well as all other available guides, documents, web pages or a doxygen style documentation. An existence of user forums or mailing lists is also an important part of learning of new systems. We mention a couple of existing solvers.

Last but not least, we aim for debugging. There are two areas which can be debugged - correctness of program and correctness of model. The correctness of program means that the program does what it should do, that it handles all inputs as a programmer expects and so on. The correctness of model stands for an accurate description of given model. User should be able to inspect variables, visualise a decision tree of search and more other information. We discuss the ways how a solver informs about mistakes (and how much descriptive the information is), the tools provided with solver to debug the program and similarly the tools which can be used to debug model correctness.

When user masters the solver and uses it to solve real problems, the time and space efficiency of used algorithms matters. We neither examine source codes, nor analyze time complexity of used algorithms. Instead we measure the time needed to load and the time to solve the problem. If a solver cannot provide such information we measure only a total time. We also measure an amount of consumed memory during the program execution. All measurements is performed several times and averaged to avoid randomness. A robustness test is also performed. In the robustness test we set limit one hour and try to compute the biggest magic sequence in given time. We use models created in the process described in the previous paragraph. In [?] authors have sent models to the solvers' authors and have given them a chance to modify them to achieve the best performance of their solvers. We focus on first-time users of a solver, so we use our own models which are not perfect and, more importantly, not tuned for any particular solver. Apart from OPL, all solvers are tested on Debian 4.0 Linux with kernel 2.6.18 on Pentium 4, 3GHz (single core with hyper threading). Due to licence problems, OPL is tested on Windows XP SP2 on Intel Core Duo, 1.6GHz.

For comparison we test on this platform also ECLⁱPS^e which step should give us relative comparison with solvers tested on another computer.

Kapitola 3

Benchmarky

V této kapitole si definujeme benchmarky, které budou dále použity k otestování vlastností jednotlivých řešičů. Jedná se o pět různých problémů, které jsou dobře známy a zdokumentovány. My si na těchto benchmarkcích ukážeme jak různý přístup řešičů k modelování problému tak budeme měřit kolik spotřebují strojového času a prostředků na vyřešení těchto problémů. Až na sebereferenční kvíz jsou všechny benchmarky škálovatelné a dají se použít také pro testy robustnosti řešiče. My budeme tuto robustnost testovat na magické sekvenci. Budeme měřit, jak velkou magickou sekvenci je ještě systém schopen spočítat do daného časového limitu. Poslední benchmark – umístění skladů – je příklad optimalizační úlohy. Řešič bude muset najít optimální řešení na základě dané objektivní funkce. Ke každému benchmarku uvedeme jeho stručný popis, model problému s omezujícími podmínkami a ukázkovou implementaci. Pro ukázkovou implementaci použijeme programovací jazyk Essence. Základy tohoto jazyka popíšeme v první části této kapitoly.

3.1 Essence

Essence je programovací jazyk pro modelování kombinatorických problémů. Umí popsat problém v omezujících podmínkách dostatečně srozumitelně a přehledně, aby i člověk, který tento jazyk nikdy neviděl, byl schopen určit, co daný program dělá. Program v Essence sestává ze tří částí. První část definuje verzi jazyka, druhá část použité proměnné a konečně třetí omezující podmínky na těchto proměnných. Použité proměnné mohou být typu Integer, Boolean a vektor/matice. Jazyk podporuje sum and loop over variable with given domain. Program může být rozdělen na část, která definuje model

a parametry reprezentující konkrétní zadání. V ukázkách v této kapitole je uvedena pouze definice modelu. Soubory s definicí parametrů můžete najít na příloženém CD. Pro překlad z jazyka Essence do zadání pro solvery slouží nástroj Tailor. Ten umí v aktuální verzi vytvořit z Essence zdrojový soubor pro řešič Minion, C++ source using Gecode as a solver a zdrojový soubor pro řešič Gecode/FlatZinc. Nástroj Tailor je detailněji popsán v sekci ??.

3.2 N-královen

Tento benchmark vychází z klasické šachistické úlohy, rozmístit na šachovnici 8 královen tak, aby se navzájem neohrožovaly. My si tuto úlohu zobecníme pro šachovnici o libovolném počtu sloupců. Cílem tedy je do tabulky o rozměrech $n \times n$ rozmístit n královen tak, aby se neohrožovaly. To znamená, že pro libovolné dvě královny platí, že nejsou ve stejném sloupci ani řádku a dokonce ani na stejné diagonále.

Pokud tento problém modelujeme, brzy zjistíme, že je výhodné ho modelovat pomocí pole proměnných délky n , kde každá proměnná nabývá hodnot od jedné do n . Dle zadání totiž nesmí být v jednom sloupci dvě královny a zároveň musí být v každém sloupci alespoň jedna královna. Stačí tedy pro každý sloupec určit, v jakém jeho řádku bude umístěná královna. Zároveň musí platit, že všechny hodnoty musí být různé, protože nelze umístit dvě královny na jeden řádek. Nakonec musíme vyřešit podmínku diagonál. Dvě královny jsou na stejné diagonále, pokud je mezi nimi stejný počet sloupců i řádků. V našem modelu to tedy znamená, že nesmí platit vztah: $|Q(i) - Q(j)| = |i - j|$

3.2.1 Model CSP

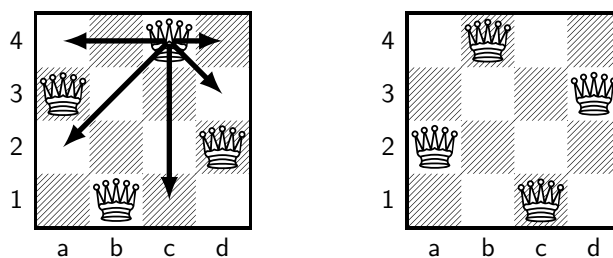
- Proměnné a domény: $q_1, \dots, q_n \in \{1, \dots, n\}$, q_i odpovídá řádku ve kterém je umístěná královna ve sloupci i
- Podmínky:
 - Žádné dvě královny nejsou na stejném řádku: $\forall i, j \in \{1, \dots, n\} : q_i \neq q_j$
 - Žádné dvě královny nejsou na stejné diagonále: $\forall i, j \in \{1, \dots, n\} : |q_i - q_j| \neq |i - j|$
 - Pro omezení symetrie: $q_1 < q_n$

Obrázek 3.1: Implementation of N-Queens Problem in Essence

```
1 language ESSENCE' 1.b.a
2 find queens: matrix indexed by [int(1..n)] of int(1..n)
3 such that
4 forall i: int(1..n). forall j: int(i+1..n).
5 alldiff(queens),
6 | queens[i] - queens[j] | != | i - j |
```

The Essence implementation is in figure 3.1.

Obrázek 3.2: Solutions of 4-queens problem



3.3 Magická sekvence

Magická sekvence je posloupnost čísel, pro kterou platí, že číslo na pozici k (číslujeme od nuly) určuje počet výskytů čísla k v posloupnosti. Například posloupnost (21200) je magickou sekvencí délky pět, protože platí výše uvedená podmínka – číslo nula je v posloupnosti právě dvakrát a proto je na první pozici dvojka, jednička je právě jedna a proto má posloupnost na pozici jedna právě číslo jedna a tak dále.

3.3.1 Model CSP

Model pro magickou sekvencí délky k :

- Proměnné a domény: Prvky magické sekvence: $m_0, \dots, m_{k-1} \in \{0, \dots, k\}$.
- Podmínky:

Obrázek 3.3: Implementation of Magic Sequence Problem in Essence

```
1 language ESSENCE' 1.b.a
2 find s : matrix indexed by [int(0..n-1)] of int(0..n)
3 such that
4   forall i : int(0..n-1).
5     ( s[i] = (sum j : int(0..n-1). (s[j] = i)))
```

- Prvek na pozici i odpovídá počtu výskytů i v posloupnosti: $\forall i \in 0, \dots, k-1 : m_i = \sum_{m_j=1} 1$.

V případě, že nelze v řešiči použít podmínku $m_i = \sum_{m_j=1} 1$, dá se sestavit alternativní model

- Proměnné a domény:
 - Prvky magické sekvence: $m_0, \dots, m_{k-1} \in \{0, \dots, k\}$,
 - pomocné proměnné: $\forall i, j \in 0, \dots, k-1 : aux_{ij}$.
- Podmínky:
 - aux_{ij} má hodnotu 1 pokud m_j má hodnotu i , jinak má aux_{ij} hodnotu 0: $\forall i, j \in \{0, \dots, k-1\} : (aux_{ij} = 1) \Leftrightarrow (m_j = i)$,
 - Prvky magické sekvence odpovídají součtu příslušných pomocných proměnných: $\forall i \in \{0, \dots, k-1\} : m_i = \sum_{j=0}^{k-1} aux_{ij}$.

The Essence implementation is in figure 3.3.

3.4 Seberefrenční kvíz

Seberefrenční kvíz je kvíz, kde odpovědi na otázky závisí na odpovědích na ostatní otázky v takovémto kvízu. Typickými otázkami v takovémto testu jsou například:

1. První otázkou na kterou je odpověď A je: A: 1, B: 2, C: 3, D: 4, E: na žádnou otázku není odpověď A
2. Odpovědí na tuto otázku je: A: A, B: B, C: C, D: D, E: E

Při řešení takovýchto kvízů se použijí zejména reifikované podmínky, tedy podmínky ve tvaru $(C \Leftrightarrow x) \& x \in \{0, 1\}$. Konstrukce takovýchto kvízů je popsána v článku [?]. Zadání testu je následující:

1. The first question whose answer is A is:
(A) 4 (B) 3 (C) 2 (D) 1 (E) none of above
2. The only two consecutive questions with identical answers are:
(A) 3 and 4 (B) 4 and 5 (C) 5 and 6 (D) 6 and 7 (E) 7 and 8
3. The next question with answer A is:
(A) 4 (B) 5 (C) 6 (D) 7 (E) 8
4. The first even numbered question with answer B is:
(A) 2 (B) 4 (C) 6 (D) 8 (E) 10
5. The only odd numbered question with answer C is:
(A) 1 (B) 3 (C) 5 (D) 7 (E) 9
6. A question with answer D:
(A) comes before this one, but not after this one (B) comes after this one, but not before this one (C) comes before and after this one (D) does not occur at all (E) none of the above
7. The last question whose answer is E is:
(A) 5 (B) 6 (C) 7 (D) 8 (E) 9
8. The number of questions whose answers are consonants is:
(A) 7 (B) 6 (C) 5 (D) 4 (E) 3
9. The number of questions whose answers are vowels is:
(A) 0 (B) 1 (C) 2 (D) 3 (E) 4
10. The answer to this question is:
(A) A (B) B (C) C (D) D (E) E

Kvíz můžeme modelovat jako tabulku s pěti sloupci (A,B,C,D,E) a deseti řádky booleovských proměnných, pro každou odpověď v každé otázce jednu. Hodnota v řádku i a sloupci j je 1, pokud je odpověď na otázku i rovna j . Protože pro každou otázku platí, že je na ní pouze jedna přípustná odpověď, přidáme do modelu podmínku "v každém řádku je právě jedna hodnota 1". Tento test má právě jedno řešení:

Tabulka 3.1: Solution of Self Referential Quiz

| Question | A | B | C | D | E |
|----------|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 |
| 4 | 0 | 1 | 0 | 0 | 0 |
| 5 | 1 | 0 | 0 | 0 | 0 |
| 6 | 0 | 1 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 1 |
| 8 | 0 | 1 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 1 |
| 10 | 0 | 0 | 0 | 1 | 0 |

3.4.1 Model CSP

- Proměnné a domény: $s_{1|1}, s_{1|2}, \dots, s_{10|4}, s_{10|5} \in 0, 1$.
- Podmínky:
 - V každém řádku je právě jedna hodnota rovna 1:
 $\forall i \in \{1, \dots, 10\} : \left(\sum_{j \in \{1, \dots, 5\}} s_{i|j} \right) = 1$,
 - otázka 1, A až D:
 $\forall i \in \{1, \dots, 4\} : (s_{1|i} = 1) \Leftrightarrow (s_{4-i+1|1} = 1 \wedge (\forall j \in \{1, \dots, 4-i\} : s_{j|1} = 0))$,
 - otázka 1, E:
 $(s_{1|5} = 1) \Leftrightarrow (\forall j \in \{1, \dots, 4\} : s_{j|1} = 0)$,
 - otázka 2:
 $\forall i \in \{1, \dots, 5\} : (s_{2|i} = 1) \Leftrightarrow (\forall j \in \{1, \dots, 5\} : s_{(3+i-1)|j} = s_{3+i|j})$,
 - otázka 3:
 $\forall i \in \{1, \dots, 5\} : (s_{3|i} = 1) \Leftrightarrow (s_{4+i-1|1} = 1 \wedge (\forall j \in \{4..2+i\} : s_{j|1} = 0))$,
 - otázka 4:
 $\forall i \in \{1, \dots, 5\} : (s_{4|i} = 1) \Leftrightarrow (s_{2i|2} = 1 / (\forall j \in \{1..i-1\} : s_{2j|2} = 0))$,

- otázka 5:
 $\forall i \in \{1, \dots, 5\} : (s_{5|i} = 1) \Leftrightarrow (s_{2i-1|3} = 1)$
- otázka 6, A:
 $\forall i \in \{1, \dots, 5\} : (s_{6|1} = 1) \Leftrightarrow (\exists j \in \{1, \dots, 5\} : s_{j|4} = 1 \wedge \forall j \in \{7, \dots, 10\} : s_{j|4} = 0)$
- otázka 6, B:
 $\forall i \in \{1, \dots, 5\} : (s_{6|2} = 1) \Leftrightarrow (\exists j \in \{7, \dots, 10\} : s_{j|4} = 1 \wedge \forall j \in \{1, \dots, 5\} : s_{j|4} = 0)$
- otázka 6, C:
 $\forall i \in \{1, \dots, 5\} : (s_{6|3} = 1) \Leftrightarrow (\exists j \in \{1, \dots, 5, 7, \dots, 10\} : s_{j|4} = 1)$
- otázka 6, D:
 $\forall i \in \{1, \dots, 5\} : (s_{6|4} = 1) \Leftrightarrow (\forall j \in \{1, \dots, 10\} : s_{j|4} = 0)$
- otázka 6, E:
 $\forall i \in \{1, \dots, 5\} : (s_{6|5} = 1) \Leftrightarrow (s_{6|4} = 1)$
- otázka 7:
 $\forall i \in \{1, \dots, 5\} : (s_{7|i} = 1) \Leftrightarrow (s_{i+4|5} = 1) \wedge (\forall j \in \{i+4+1\dots 10\} : s_{j,5} = 0)$
- otázka 8:
 $\forall i \in \{1, \dots, 5\} : (s_{8|i} = 1) \Leftrightarrow \left(\sum_{j=1}^{10} (s_{j|2} + s_{j|3} + s_{j|4}) = 7 - i + 1 \right)$
- otázka 9:
 $\forall i \in \{1, \dots, 5\} : (s_{0|i} = 1) \Leftrightarrow \left(\sum_{j=1}^{10} (s_{j|1} + s_{j|5}) = i - 1 \right).$

The Essence implementation is in figure 3.4.

3.5 Quasigroup with holes

Kvazigrupa neboli latinský čtverec je tabulka o velikosti $n \times n$ vyplněná čísly v rozsahu $0 - n$ tak, že se žádné číslo v řádku ani ve sloupci neopakuje. Je možné zadat i dodatečnou podmínku na prvky kvazigrupy, například, že prvky na diagonále musí být sudé apod. Úkolem je pak pro danou částečně vyplněnou kvazigrupu najít zbylé prvky tak, aby byla plně vyplněna a splňovala všechny podmínky na ní kladené. Takovéto zadání se nazývá problém doplnění kvazigrupy – quasigroup completion problem (QCP). Bohužel toto nemusí vést vždy ke stejným výsledkům. Některá zadání totiž není možné splnit, některá jsou překvapivě jednoduchá apod.

Obrázek 3.4: Implementation of Self Referential Quiz in Essence

```
1 language ESSENCE' 1.b.a
2 find s : matrix indexed by [int(1..10), int(1..5)] of bool
3 such that
4 $ the is only one answer to each question and there is not any unanswered question
5 forall row : int(1..10). ((sum col : int(1..5). s[row,col]) = 1),
6 $ Question 1
7 $ A to D
8 forall col : int(1..4). ( (s[1,col] = 1) <=> ( (s[(4-col+1),1] = 1) /\ ( forall row :
   int(1..(4-col)). (s[row,1] = 0) ) ) ),
9 $ E
10 (s[1,5] = 1) <=> (forall row : int(1..4). (s[row,1] = 0)),
11
12 $ Question 2
13 forall col : int(1..5). ( (s[2,col] = 1) <=> ( forall col2: int(1..5). (s[3+col-1,
   col2] = s[3+col,col2]) ) ),
14 $ Question 3
15 forall col : int(1..5). ( (s[3,col] = 1) <=> ( (s[(4+col-1),1] = 1) /\ ( forall row :
   int(4..2+col). s[row,1] = 0 ) ) ),
16 $ Question 4
17 forall col : int(1..5). ( (s[4,col] = 1) <=> ( (s[col*2,2] = 1) /\ ( forall row : int
   (1..(col-1)). s[row*2,2] = 0 ) ) ),
18 $ Question 5
19 forall col : int(1..5). ( (s[5,col] = 1) <=> (s[2*col-1,3]=1) ),
20 $ Question 6
21 (s[6,1] = 1) <=> ( ( exists row : int(1..5). s[row,4] = 1 ) /\ ( forall row : int
   (7..10). s[row,4] = 0 ) ),
22 (s[6,2] = 1) <=> ( ( exists row : int(7..10). s[row,4] = 1 ) /\ ( forall row : int
   (1..5). s[row,4] = 0 ) ),
23 (s[6,3] = 1) <=> ( ( exists row : int(7..10). s[row,4] = 1 ) /\ ( exists row : int
   (1..5). s[row,4] = 1 ) ),
24 (s[6,4] = 1) <=> ( forall row : int(1..10). s[row,4] = 0 ),
25 (s[6,5] = 1) <=> (s[6,4] = 1),
26 $ Question 7
27 forall col : int(1..5). ( (s[7,col] = 1) <=> ( (s[col+4,5] = 1) /\ ( forall row : int
   (col+4+1..10). s[row,5] = 0 ) ) ),
28 $ Question 8
29 forall col: int(1..5). ( (s[8,col] = 1) <=> ( ( sum row: int(1..10). (s[row,2] + s[
   row,3] + s[row,4]) ) = (7-col+1) ) ),
30 $ Question 9
31 forall col: int(1..5). ( (s[9,col] = 1) <=> ( ( sum row: int(1..10). (s[row,1] + s[
   row,5]) ) = (col-1) ) )
32 $ Constraints for question 10 are useless
```

Navíc je úloha zaplnění latinského čtverce NP-úplná, nemůžeme si tedy být nikdy jistí, zda je problém pouze moc náročný pro řešič a nebo zda řešič nedává odpověď proto, že řešení neexistuje. Proto se místo klasického QCP používá jeho modifikace – kvazigrupa s dírami (QWH). Nejprve si předem vygenerujeme kvazigrupu, která splňuje danou podmínku. Pak z této kvazigrupy odstraníme některé hodnoty, které pak necháme dopočítat solver. Tím máme zaručeno, že řešení existuje. Generováním vhodných zadání pro QWH se zabýval D. Achlioptas a kol., kteří ukázali, že těžkost problému souvisí s tzv. páteří [?]. Páteř je množina prvků kvazigrupy, které se vyskytují ve všech řešeních. Pokud se podíl páteře blíží k 0%, znamená to, že existuje mnoho různých řešení a tedy řešič může nějaké řešení najít "náhodou". Pokud je oproti tomu podíl páteře blízký 100%, existuje velmi málo řešení a vedou k němu všechny podmínky. Dá se tedy očekávat, že zajímavé kvazigrupy budou takové, jejichž páteř má podíl okolo 50%. Experimenty pak prokázaly [?], že nejnáročnější jsou problémy s podílem páteře mezi 30 a 35%.

My použijeme kvazigrupy bez dodatečné podmínky na prvky, které si vygenerujeme pomocí programů lsencode od Carly Gomezové [citace? odkaz?] a walksat od Henryho Kautze.

3.5.1 Model CSP

Model pro kvazigrupu řádu n . Předvyplněné hodnoty jsou v poli $data_{ij}$:

- Proměnné a domény: $q_{11}, \dots, q_{nn} \in \{1, \dots, n\}$.
- Podmínky:
 - Prvky v jednom řádku jsou různé: $\forall i \in \{1, \dots, n\} : \forall j, k \in \{1, \dots, n\} : q_{ij} \neq q_{ik}$,
 - prvky v jednom sloupci jsou různé: $\forall i \in 1, \dots, n : \forall j, k \in 1, \dots, n : q_{ji} \neq q_{ki}$
 - některé prvky mají hodnotu danou zadáním: $data_{ij}$ definováno $\Leftrightarrow (q_{ij} = data_{ij})$

The Essence implementation is in figure 3.5.

Obrázek 3.5: Implementation of Quasigroup With Holes Problem in Essence

```
1 language ESSENCE' 1.b.a
2 letting nDomain be domain int(1..n)
3 find qcp : matrix indexed by [nDomain,nDomain] of nDomain
4 such that
5   forall i : nDomain. alldiff(qcp[i, nDomain]),
6   forall i : nDomain. alldiff(qcp[nDomain,i])
```

3.6 Výstavba skladů

Představme si, že máme hypotetické obchodní společnosti pomoci s rozhodnutím, jaké postavit sklady pro své prodejny. Hlavním kritériem je samozřejmě cena, jakou bude tato výstavba stát. Naším cílem je tedy tuto cenu minimalizovat. Na druhou stranu musíme vyhovět všem omezujícím podmínkám, na problém kladených. Společnost má již postavené prodejny a vytipované lokality pro své sklady. Každý možný sklad má definovanou svou maximální kapacitu – nejvyšší možný počet obsluhovaných obchodů. Každý obchod musí mít přidělený právě jeden sklad. Cena za výstavbu skladu je fixní ale náklady na zásobení prodejny z daného skladu je pro každou takovou dvojici různá. Řešič na začátku dostane jako zadání počet prodejen, počet možných skladů s jejich kapacitami a matici c , kde prvek c_{ij} určuje cenu za zásobování prodejny i ze skladu j .

3.6.1 Model CSP

Máme k dispozici W skladů, přičemž otevření každého skladu stojí openCost . Dále máme S prodejen. Máme danou maximální kapacitu skladu w , kde w_i je maximální počet prodejen, které mohou odebírat zboží od skladu $i \in \{1, \dots, W\}$. Nakonec máme zadanou cenu supplyCost dopravy zboží tak, že supplyCost_{ij} znamená cenu dopravy ze skladu i do prodejny j .

- Proměnné a domény:
 - Celková cena – hodnota objektivní funkce: $\text{totalCost} \in \mathbb{N}$,
 - počet otevřených skladů: $\text{numberOpen} \in \{0, W\}$,
 - indikace, zda je sklad otevřen: $\text{open}_1, \dots, \text{open}_W \in \{0, 1\}$,
 - indikace který sklad zásobuje kterou prodejnu: $\text{supplier}_1, \dots, \text{supplier}_S \in \{1, \dots, W\}$,

- pomocná proměnná indikující cenu dopravy do prodejny: $\text{cost}_1, \dots, \text{cost}_S \in \mathbb{N}$,
 - celková cena za dopravu: $\text{costSum} \in \mathbb{N}$.
- Podmínky:
 - Objektivní funkce: $\text{totalCost} = \text{costSum} + \text{numberOpen} \cdot \text{openCost}$,
 - celková cena za dopravu: $\text{costSum} = \sum_i \text{cost}_i$,
 - počet otevřených skladů: $\text{numberOpen} = \sum_i \text{open}_i$,
 - kapacita skladu nesmí být překročena: $\forall i \in \{1, \dots, W\} : w_i \geq \sum_{\text{supplier}_j=i} 1$,
 - pokud sklad dodává do nějaké prodejny, je otevřený: $\forall i \in \{1, \dots, W\} : (\text{open}_i = 1) \Leftrightarrow \left(\left(\sum_{\text{supplier}_j=i} 1 \right) > 0 \right)$,
 - cena za dopravu ze skladu do prodejny: $\forall i \in 1, \dots, S, \forall j \in \{1, \dots, W\} : (\text{supplier}_i = j) \Rightarrow (\text{cost}_i = \text{supplyCost}_{ij})$.

The Essence implementation is in figure 3.6.

Obrázek 3.6: Implementation of Locating Warehouses Problem in Essence

```
1 language ESSENCE' 1.b.a
2
3 given Capacity : matrix indexed by [WarehousesRANGE] of int(0..numberOfStores)
4 given StoreWarehouseCost : matrix indexed by [StoresRANGE, WarehousesRANGE] of CostRANGE
5 letting CostRANGE be domain int(0..maxCost)
6 letting StoresRANGE be domain int(0..numberOfStores-1)
7 letting WarehousesRANGE be domain int(0..numberOfWarehouses-1)
8
9 find
10   TotalCost : CostRANGE,
11   Open : matrix indexed by [WarehousesRANGE] of int(0..1),
12   NumberOpen : int(0..numberOfWarehouses),
13   Supplier : matrix indexed by [StoresRANGE] of WarehousesRANGE,
14   Cost : matrix indexed by [StoresRANGE] of CostRANGE,
15   SumCost : CostRANGE
16
17 minimising TotalCost
18
19 such that
20   TotalCost = SumCost + NumberOpen * warehouseCost,
21   SumCost = sum j : StoresRANGE. (Cost[j]),
22   NumberOpen = sum j : WarehousesRANGE. (Open[j]),
23
24   forall i : WarehousesRANGE.
25     (Capacity[i] >= (sum j : StoresRANGE. (Supplier[j] = i))),
26
27   forall i : WarehousesRANGE.
28     (((sum j : StoresRANGE. (Supplier[j] = i)) > 0) => (Open[i] = 1)),
29
30   forall i : WarehousesRANGE.
31     (((sum j : StoresRANGE. (Supplier[j] = i)) = 0) => (Open[i] = 0)),
32
33   forall i : StoresRANGE. forall j : WarehousesRANGE. ( (Supplier[i] = j) => (Cost[i] =
      StoreWarehouseCost[i,j]) )
```

Kapitola 4

Constraint solvers

4.1 Mozart/Oz

`$Id: mozart.tex 145 2009-07-26 12:48:21Z tutchek $`

Mozart je implementací multiparadigmatického jazyka Oz. Oz je funkcionální jazyk, který má vestavěnou podporu pro vícevláknové aplikace, podporu pro paralelizaci a mimo jiné obsahuje také vestavěnou podporu pro řešení problémů s omezujícími podmínkami. Jak bylo zmíněno dříve, jde o jazyk multiparadigmatický, lze v něm tedy psát i imperativní programy stejně jako logické programy (podobné prologu). Dále je možné definovat třídy včetně dědičnosti a praocvat s od nich odvozenými objekty. Tento jazyk byl navržen pro co nejvyšší variabilitu při použití, protože programátor může využít naráz kombinaci funkcionálního, logického i imperativního programování, OOP a další najednou. Ve standardní distribuci je dodáván jako samostatný překladač, který vytvoří nativní programy pro daný operační systém. Mimo toho umí také běžet v interaktivním módu, kdy se jednotlivé bloky příkazů posílají přímo kompilátoru. Jako IDE se používá běžně systém EMACS, se kterým je Mozart/Oz dodáván.

Podobně jako v jiných funkcionálních jazycích je do proměnné možné přiřadit pouze jednu hodnotu za dobu jejího života. Proto má každá proměnná vedle hodnoty ještě stav. V případě, že má dojít k akci nad proměnnou do které není nic přiřazeno, ač by to příkaz vyžadoval dojde k pozastavení vlákna dokud nebude do proměnné přiřazeno. To umožňuje provést následující program:

```
a = 5
```

```

if a > b then c = 5 else c = 6
b = 4
// V c nyní bude hodnota 5

```

Jazyk obsahuje vestavěnou podporu pro paralelní výpočty, tedy na úrovni jazyka je možné vytvářet vlákna a synchronizační primitiva. Jazyk navíc podporuje předat výpočet na jiný počítač skrze protokol TCP/IP. Lze tedy využít cluster počítačů pro zrychlení náročných výpočtů.

4.1.1 Popis řešiče omezujících podmínek

Při řešení problémů s omezujícími podmínkami obsahuje vestavěný řešič problémů s konečnou doménou (finite domain). Pro tyto účely se množinou s konečnou doménou rozumí konečná množina přirozených čísel s nulou. Řešič obsahuje omezení na nejvyšší možnou hodnotu proměnné. Výpočetní model pro propagaci podmínek se jmenuje prostor (space) a je tvořen několika propagátory napojenými na constraint store. Constraint store obsahuje konjunkci základních podmínek, tedy podmínek ve tvaru $x = n$ nebo $x \in D$. Tedy může vypadat například $x = 6 \& y \in 1..12 \& z = y$. Propagátory obsahují ostatní podmínky, tedy například $x > y$ nebo $a^2 + b^2 = c^2$. Propagátor pro podmínku c je samostatný výpočetní agent, který se snaží omezit domény proměnných, které jsou obsažené v c . Řešením je pak takové přiřazení hodnot proměnným, které splní všechny podmínky dané propagátory.

Example 1 *Mějme proměnné X a Y a následující podmínky: $X \in \{0..9\}$, $Y \in \{0..9\}$, $X + Y = 9$, $2X + 4Y = 24$.*

1. *Constraint store obsahuje: $X \in \{0..9\}$, $Y \in \{0..9\}$. Propagátory: $X + Y = 9$ a $2X + 4Y = 24$.*
2. *První propagátor nemůže udělat nic, druhý ale může změnit constraint store tak, že obsahuje $X \in \{0..8\}$, $Y \in \{2..6\}$.*
3. *První propagátor nyní může změnit constraint store na $X \in \{3..7\}$, $Y \in \{2..6\}$.*
4. *Druhý propagátor nyní změní constraint store na $X \in \{4..6\}$, $Y \in \{3..4\}$.*
5. *První propagátor změní constraint store na $X \in \{5..6\}$, $Y \in \{3..4\}$.*

6. Druhý propagátor nakonec změní constraint store, tak že obsahuje $X = 6$, $Y = 3$.

Propagace může být buď intervalová nebo doménová. Zatímco intervalová propagace pouze mění okraje domény, doménová také odstraňuje vnitřní hodnoty domény. Přestože doménová propagace vypadá jako lepší technika, používá se spíše intervalová pro její výpočetní jednoduchost.

Již pro jednoduché problémy ale propagace nemusí vést k výsledku. Vezměme například problém: $x \neq y$, $x \neq z$, $y \neq z$, $x \in \{0..1\}$, $y \in \{0..1\}$, $z \in \{0..1\}$. Tento problém nelze více propagovat, všechny podmínky lze splnit ale žádná proměnná není přiřazena hodnota. V této chvíli lze použít distribuci. Problém P je distribuován pomocí podmínky C , pokud vyřešíme nové problémy $P \cup \{\neg C\}$ a $P \cup \{C\}$. Alespoň v jednom z nových problémů se buď nachází řešení problému P a nebo je problém neřešitelný. V naší reprezentaci se tohoto faktu dá využít pro space. Pokud se systém dostane do stabilního stavu, kdy žádná podmínka není nesplněna a přitom není možné přiřadit žádné proměnné hodnotu, tak vybereme proměnnou x a hodnotu n takovou, že je konzistentní se všemi podmínkami nad x . Tím získáme dva nové space, jeden $S \cup x = n$ a jeden $S \cup x \neq n$. Na obou space pak pustíme propagaci. Pokud se dostane opět do stabilního stavu, znova distribuujeme a takto postupujeme až do vyřešení problému a nebo konstatování, že je problém neřešitelný. Tato metoda je úplná, tedy pokud existuje alespoň jedno řešení, pak budou nalezena všechna.

Je možné volit z různých distribučních strategií. Volba správné strategie může výrazně ovlivnit dobu výpočtu. Pro většinu problémů je dobré zvolit first-fail strategii, tedy volit proměnnou s nejmenší možnou doménou. Pokud ale žádná dodávaná distribuční strategie nevyhovuje, je možné implementovat své distribuční strategie, plně na míru problému.

Pro hledání řešení optimalizačních problémů se dají využít dvě techniky. Naivní technika spočívá v postupném zvyšování výsledku objektivní funkce a následném hledání řešení, které tomuto výsledku bude odpovídat. Pro tento postup existuje v jazyce konstrukce, která před splňováním ostatních podmínek postupně zvyšuje danou proměnnou. Tento postup však postrádá obecnost a Mozart nabízí lepší postup – techniku branch and bound. Tato technika vyžaduje, aby uživatel sestavil porovnávací funkci. Porovnávací funkce jako argumenty přijímá předchozí a aktuální řešení problému. Funkce pak zpravidla zavolá objektivní funkci a porovná její výsledky. Tato technika oproti naivní výrazně zrychlí výpočet.

4.1.2 Nástroje pro podporu modelování

Mozart nabízí uživateli interaktivní nástroj Explorer, který umožňuje prozkoumat strom řešení včetně dílčích rozhodnutí, které řešič během výpočtu učinil. Explorer je možné používat také v interaktivním módu a ručně určovat, které větve řešení prozkoumat.

Kolečka označují uzly rozhodovacího stromu, ve kterých proběhlo rozhodnutí, kosočtverce označují nalezení úspěšného řešení a čtverec větev kde není řešení. Světlejší barvou jsou uvedené uzly, které je možné ještě dále expandovat. Na uvedeném obrázku tedy máme jedno řešení, dvě neúspěšná řešení a pět okamžiků, kdy došlo k rozhodnutí z nichž lze ještě ve třech hledat další řešení. Explorer umožňuje také export nakresleného stromu ve formátu PostScript.

4.1.3 Implementace benchmarků

Každý implementovaný benchmark je samostatný solver. Po zavolání příslušného solveru je vrácena funkce, která je posléze předána funkci realizující hledání řešení. Jde buď o funkce SearchOne resp. SearchAll, které naleznou jedno nebo všechna řešení problému a vrátí je v seznamu. Pro spuštění exploreru a interaktivní hledání řešení lze pak zavolat ExploreOne resp. ExploreAll.

4.2 Choco

`$Id: choco.tex 152 2009-07-26 22:46:48Z tutchek $`

Choco is a constraint solver which is implemented as a library in Java programming language. It is distributed as a JAR package having a Javadoc documentation included. It is quite easy to install it even for a beginners in Java and it lasts about five minutes in commonly used IDEs. Since the Java is used, the Choco solver is available for various platforms and operating systems. As far as it is not our goal to describe possibilities of host operating system, we are not about to further discuss the Java features. Choco is being developed at Ecole des Mines de Nantes ve Francii and it is freely available for downloading from SourceForge server. The main number of current version is 2. Choco divides the problem solution into two parts – a model and a solver itself. The model contains variables and constraints given in task. Afterwards, the solver is given the model as an input and it tries to find a solution. Variables in the model can be integers, real numbers or

sets. Then the solver is able to find a solution for the current model. A user can get information from the solver whether the problem has a solution or it contains a conflict. There is an interface for resolving solutions themselves, whereby one can ask for the first, the following or all existing solutions. If we define a variable equal to a value of an objective function, the solver can this variable either minimize, or maximize. Furthermore, the solver allow us to choose a strategy which might perfectly fit the given problem. The variables of solver depends on the variables of the model and one can resolve values only through the variables of the solver.

4.2.1 Solver description

As it has been already mentioned in the previous section, the solving of problem is divided into two separated tasks – to define a model and to deploy the model to a well-configured solver. The model as well as the solver are classes, both of which are instanced by a user into objects. First we describe the model and after that we look at the solver.

Model

The model is a instance of class `CPModel`. In Choco the variables of the model are represented as objects of the following types: `IntegerVariable`, `RealVariable` and `SetVariable`. Those variables, generally, are not created using the keyword `new`, but in Choco there are factory methods for this purpose. One has to register those variables first by calling a function `CPModel::addVariable`, or `CPModel::addVariables` when adding an array of variables at once. While registering variables into a model, we can set additional properties to the variables, for example to set whether it is a decision variable, or a variable containing a result of an objective function. It is not necessary to set those properties every time; however, they might rapidly improve the computation. Alternatively, we can define those properties in the solver, which way is described later in the following section.

Once we have registered the variables, a definition of constraints follows. It is possible either to use a large number of build-in constraints, or to define our own constraints. The constraints which are available natively in the solver are listed in the table [??](#). Each of the constraint fits in one the following groups: basic constraints (`true`, `false`, relation operators), basic expressions (goniometric functions, powers, sums), other constraints (`abs`,

`div`, `max`, ...), reified constraints (`and`, `or`, `ifOnlyIf`) and global constraints (`allDifferent`, `occurrenceMax`, ...). Furthermore, there are constraints available which might be used for modeling geometric constraints, scheduling constraints and constraints for a sequention of variables which is accepted by finite automaton.

Apart from the build-in constraints, is it possible to define our own constraints. The first way is to define a constraint $p(x, y)$ as a set of compatible values (a, b) , where p is satisfied if $x = a$ and $y = b$, or, eventually, as a set of incompatible values. In that case the set is define as a table of value. Besides, we define the constraint as a predicate, which has to be satisfied, whereby the constraint is an instance of a class derived from a class `BinRelation` with a method `checkCouple` having implemented. This function takes two values as params and returns boolean value whether the condition was satisfied or not. Simirarly, we can define constraints over tuples. For all such constraints (either binary or tuple) we can specify desired algorithm for arc consistence. There is `AC3`, `AC2001`, `AC3rm` and `AC3` available for binary constraints and `AC32`, `AC3rm`, `AC2001` and `AC2008` tuple constraints. A description of each of the algorithm can be found in [najít citaci pro algoritmy AC].

Solver

A solver is an instance of class `CPSolver`, which tries to find a solution according to the model from the previous section. The solver starts with reading the variables of the model and converting them into variables of the solver (`IntegerVariable` into `IntDomainVar`, `RealVariable` into `RealVar` and `SetVariable` into `SetVar`). Afterwards, it reads the constraints of the model and creates constraints of the solver, which are based on previously read constraints. Then the solver uses a search strategy and search for solutions. Since the chosen strategy is a key factor for the speed of solving, one can configure its various options. A user can specify a selector and an iterator. Where the selector specifies which variable is about to be taken in next solver's desicion and the iterator chooses each of available values and iterates over them. In a standard distribution of Choco there are basic selectors such as *variable with minimum domain*, *variable with maximum domain* and so on. Whilst the iterators can try values in ascendant or descendant order. An alternative to iterator is a value selector, which returns next available value when required. As for value selector, we can use, for example, minimal value in a domain, random value in a domain and so on. We can choose

different user-defined strategies for various group of variables so as to follow the specified problem in the best way. In that case we define the solver's behaviour through so-called goals. A goal contains a definition of a strategy, that means a selector for certain variables and an iterator over values.

While solving large-scale problems, it might be enormously time demanding, take too much system resources and so on. To avoid this we can define solver limits. In the solver we can set a time limit, a limit for a number of nodes, a depth of backtracking, a number of fails or a limit for CPU time. Apart from that, a user can define their own limits.

Once the solver has read the model and the strategies are defined, it starts solving the problem. The solver offers an interface for accessing either each solutions (`solve`, `nextSolution`), or to get all the solutions at once. Moreover, we can specify a variable which the solver tries to minimize or maximize. Since the result is held in variables of the solver and not in user-defined variables of the model, it is required to resolve the solver's variables by calling a function `CPSolver::getVar`, which accepts a variable of a model and returns a variable of a solver.

4.2.2 Debugging support

Choco does not include any tools for graphic visualisation of search tree such as systems Mozart or Gecode do; however, in Choco it is possible to print out a log of the solving process. One can configure several levels how detailed information is logged varying from nothing to a complete list of what the Choco does internally. `***ZPRACUJVIS***`

4.2.3 Subjective description

The system has a good documentation, although it is quite jumble. Even though an accurate reader finds virtually everything they look for. A documentation for developers is generated by JavaDoc system. Due to that fact it is available as a hinting tool for many users of common Java IDEs, that definitely helps for better understanding of the solver. Since the development of the solver is maintained at SourceForge server, it is quite easy to access source codes as well as a history of versions via revision control system Subversion. One can find there also a technical support forum, where the authors answers the users' questions. The reaction time is very low and the answers are of high quality so most of the problems are quickly fixed.

4.3 Minion

\$Id: minion.tex 143 2009-07-26 10:01:45Z tutchek \$

Minion je řešič omezujících podmínek, který funguje jako samostatná aplikace, která na vstupu dostane popis problému a na výstupu vrátí řešení. Jako mnoho ostatních řešičů je k dispozici na serveru sourceforge.net jako open-source program. Oproti ostatním řešičům, které fungují jako knihovny pro nějaký programovací jazyk a je možné je použít jako subsystém pro řešení omezujících podmínek přímo uvnitř programu, Minion pracuje samostatně. Popis problému dostává ve speciálním datovém souboru, který je rozdělen na definici proměnných, definic podmínek na nich a případnou definici objektivní funkce. Výstup řešiče pak může uživatel přečíst na standardním výstupu. Samotný jazyk pro popis problému by se dal nejlépe popsat jako "assembler pro omezující podmínky". Škálovat problém není možné bez změny vstupního souboru, protože ten je navržený jak na míru problému tak jeho parametrů – zatímco v jiných řešičích stačí pro změnu z problému čtyř královen na osm královen jenom přepsat konstantu 4 na 8, zde je nutné přepsat většinu vstupního souboru. Množina podporovaných podmínek je konečná a není možné přidat novou podmínku bez zásahu do kódu samotného řešiče. Uživatele jistě zaujme, že není možné specifikovat podmínku $a = \sum w_i x_i$, ale je nutné ji nahradit dvojicí $a \leq \sum w_i x_i$, $a \geq \sum w_i x_i$. Podmínky také není možné řetězit. Pokud je řetězení podmínek pro model podstatné, je nutné zavést auxiliary proměnnou, přes kterou se podmínka sváže. Minion podporuje čtyři typy proměnných – bool, bound, discrete a sparsebound. Bool proměnné jsou proměnné s doménou $\{0, 1\}$. Bound proměnné jsou interně uloženy pouze okraji intervalu, který reprezentují. Discrete proměnné také reprezentují zadaný interval, ale oproti bound proměnným mohou obsahovat v intervalu díru. Konečně sparsebound reprezentuje doménu, která sestává z převážně osamocených hodnot, které jsou definovány ve vstupním souboru. V průběhu výpočtu se pro sparsebound proměnné upravují opět pouze okraje intervalu. Z těchto proměnných je pak možné vytvářet vektory, matice a n-tice.

Formát vstupu pro minion není moc human-friendly. Již pro malé problémy roste počet pomocných proměnných nad rozumné meze a vstup se stává nepřehledným. Proto se vyplatí vstupní soubor generovat automaticky. Buď pomocí ad hoc generátorů, které si uživatel napíše např. jako součást svého programu, který posléze volá minion a nebo pomocí nástroje Tailor popsaného v sekci ??.

4.3.1 Popis řešiče omezujících podmínek

Jak bylo zmíněno v předchozí sekci, minion je samostatný spustitelný program, který jako parametr dostane název vstupního souboru (případně dostane vstup na standardním vstupu) a na standardní výstup případně do souboru vypíše řešení. Umí pracovat pouze s celými čísly, problém tedy musí být zakódován tak, aby se jimi dal vyjádřit. Formát vstupního souboru prochází mezi verzemi změnami, ale řešič by měl být schopen pracovat se všemi staršími formáty. Aktuální verze formátu vstupního souboru má číslo 3. Každý soubor obsahuje na prvním řádku specifikaci formátu, v tomto případě tedy `MINION 3`. Dále následují v libovolném pořadí a s libovolnou možností opakování jednotlivé sekce souboru. Soubor může obsahovat sekci s definicí proměnných, sekci s definicí n-tic, sekci s definicí podmínek a konečně sekci s definicí parametrů hledání řešení. Podmínky se nechovají jako klasické funkce, ale jako predikáty, které musí být splněny a nemají návratovou hodnotu. To vede k tomu, že abychom složili efekty více podmínek, musíme zavést pomocnou proměnnou. Například mějme podmínku $|x| < y$ a k dispozici jen podmínky $X = |Y|$ a $X < Y$. Pro podmínku $|x| < y$ pak musíme zavést pomocnou proměnnou a a do modelu přidat následující dvě podmínky $a = |x|$, $a < y$. Přehled podporovaných podmínek řešiče Minion lze nalézt v tabulce ???. Parametry vyhledávání lze ovlivnit buď přímo v definičním souboru, například pořadí proměnných apod. a nebo skrze parametry příkazové řádky.

4.3.2 Nástroje pro podporu modelování

Minion umožňuje vypsát na standardní výstup prohlédávací strom, ve kterém je možné sledovat činnost solveru na předloženém modelu. Vlastní nástroj pro visualisaci řešení systém neobsahuje. Protože je ruční modelování v systému Minion poměrně obtížné, uživatel pravděpodobně použije pro tvorbu modelu nástroj Tailor. Ten umožňuje jak překlad programu v Essence' do formátu Minion, tak přímo spustit minion a sledovat řešení z Tailoru.

4.3.3 Subjektivní hodnocení solveru

Autoři na stránkách solveru uvádí, že je Minion nejrychlejší solver, který je k dispozici. Toto necháme na posouzení čtenáři po vlastních pokusech se systémem případně po zhodnocení výsledků našich benchmarků. Rozhodně se ukázalo, že v případě kdy systému nedojde paměť z důvodu veli-

kosti problému, je i pro velké instance problému schopen okamžité odpovědi. Největší překážkou je tedy vstupní formát, ale tento problém je elegantně řešitelný zmíněným programem Tailor. Jako samostatný program, který je možné zavolat s definicí problému a on na standardní výstup vrátí řešení je použitelný jako řešič i pro takové programy jako například Bash skripty. Jediný požadavek na použití je schopnost sestavit vstupní soubor. Co se týče dokumentace, je k dispozici referenční příručka, která obsahuje popis všech podmínek a vstupního formátu souboru. Také je součástí příručky jemný úvod do převodu CSP modelu na vstupní soubor Minionu.

4.4 Gecode

`$Id: gecode.tex 157 2009-07-28 07:26:22Z gufy $`

Gecode is a C++ library for solving problems with constraints. It allows to model a problem which contains integers, boolean variables and finite integer sets. Gecode is a free open source software as many of the mentioned solvers are. The library is distributed as source codes and for Windows OS there is also an installer with pre-compiled libraries. Optionally a user needs a Qt library, which is used in a visualization graphic tool Gist. Apart from basic constraints, Gecode has also constraints for scheduling, finite automata, graphs and so on. We describe those constraints further in the following section. Gecode often uses the feature of C++ to overload functions and operators, which leads to a fact that many equal constraints for varying data types have the same name. In addition, the overloading of operators helps to ease the work with expressions. A disadvantage might be a lack of explicit consciousness about a state of used operators, therefore a user cannot be sure if an operator that is used for a variable is just a common operator, or a part of constraint. A key man of Gecode solver is Christian Schulte, who also participated in a development of the system Mozart/Oz.

4.4.1 Solver description

A problem is modeled as a class derived from a class Space. In this particular class there are defined variables and constraints. Variables are objects of one of the following type: IntVar for integers, BoolVar for boolean variables and SetVar for finite integer sets. Compared to other solvers the boolean variables aren't just integers with a domain 0, 1. It is not even allowed to declare a constraint $b = i$ having a boolean variable b and an integer i . When a relation