# A generic solver for Constraint Satisfaction Problems

*Author:*
Jelle Bakker
s2171236

*1st supervisor:*
dr. Arnold Meijster

*2nd supervisor:*
prof. dr. Gerard R. Renardel de Lavalette

July 15, 2015

## Contents

# 1 Introduction

Constraint Satisfaction Problems (CSPs) are a class of problems that we often encounter in daily life. A CSP is a problem that consists of a given set of variables that need to satisfy a given set of constraints. Each CSP can in principle be written as a standard search problem, but this generalization ignores problem specific properties. Often, these properties can help to solve a CSP more efficiently than a brute-force search for a solution. In the literature on CSPs, several techniques have been proposed to speed-up this search process. Most of these techniques involve heuristics, and thus do not guarantee a speed improvement.

The research questions that we want to investigate in this thesis are the following:

1. Is it feasible to write a generic solver for different types of CSPs?
   (a) To answer this question, we want to try to implement a generic solver.
   (b) Moreover, on the fly we want to design a mini-programming language in which CSPs can be specified. The solver should accept CSPs written in this language.
2. Can we optimize the solving process in such a way that the number of computation steps will be reduced significantly by applying heuristic techniques?

# 2 Definition of a Constraint Satisfaction Problem

In this section, we introduce the formal definition of a CSP. We adopt the definition as given in [1].

A constraint satisfaction problem (CSP) is defined by a finite set of *variables* $X_1, X_2, ..., X_n$, a finite set of non-empty domains $D_1, D_2, ..., D_n$, and a finite set of *constraints* $C_1, C_2, ..., C_m$. Each variable $X_i$ can take on values from the domain $D_i$. A constraint $C_i$ is a predicate over a subset of the variables of the problem, which limits the values that these variables can have. There are different types of constraints:

1. A *unary constraint* is a constraint in which exactly one variable is involved, for instance $X_1 < 5$.
2. A *binary constraint* is a constraint in which exactly two variables are involved, for instance $X_2 \neq X_3$.
3. *Higher-order constraints* are constraints in which at least three variables are involved, for instance $X_4 + X_5 > X_6 \times X_7$.

A solution of CSP is a complete assignment of values to all the variables such that all the constraints are met. Of course, each variable is assigned a value from its own domain.

In this thesis, solving a CSP is considered a search process. We define a *state* of this search process as a *partial* or *complete* assignment of values to the variables of the problem. When a state does not violate any of the constraints, it is called a *consistent* or *legal* assignment. An assignment is called *complete* when all variables have been assigned a value. Hence, a *solution* is a state that is complete

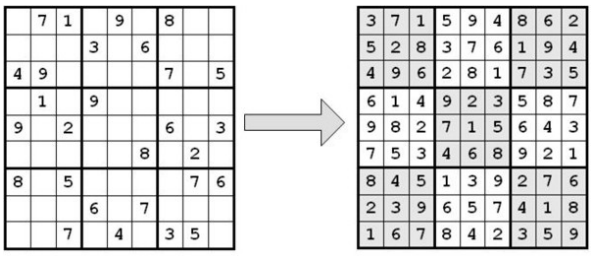

Fig. 1: A Sudoku puzzle and its solution

and consistent. There exist CSPs for which an extra *soft-requirement* must be met: the solution must maximize (or minimize) some *objective function*. This type of problems are called *optimization problems*, and are not considered in this thesis.

## 3 Some examples of Constraint Satisfaction Problems

In this section, we give some examples of CSPs that will be used as running examples throughout this thesis.

**Example 3.1.** Sudoku puzzles.
A Sudoku puzzle is a puzzle that consists of a $9 \times 9$ grid of *cells*. Each cell should be filled with a decimal digit. The grid is divided in 9 *sub-blocks* of $3 \times 3$ cells. In a solution state of a Sudoku, each decimal digit should occur exactly once in each row, column, or block. Initially, some digits in the Sudoku are given, but most of the cells are still empty. The difficulty of the puzzle typically depends on how many digits are given in advance.

A Sudoku fits perfectly in the definition of a CSP: each grid cell can be considered a variable, of which the allowed domain is $\{1,..,9\}$. The rules of a Sudoku are easily translated into a set of constraints. For example, the following mini-Sudoku

| | | | |
|---|---|---|---|
| 1 | A | B | 4 |
| C | 4 | 1 | D |
| E | 3 | 2 | F |
| 2 | G | H | 3 |

can be translated into the following formal CSP:

Variables: $A, B, C, D, E, F, G, H$
Domains: $D_A = D_B = D_C = D_D = D_E = D_F = D_G = D_H = \{1,2,3,4\}$
Constraints: $A \neq B$, $A \neq C$, $A \neq G$, $B \neq D$, $B \neq H$, $C \neq D$, $C \neq E$, $D \neq F$, $E \neq F$, $E \neq G$, $F \neq H$, $G \neq H$, $A \notin \{1,3,4\}$, $B \notin \{1,2,4\}$, $C \notin \{1,2,4\}$, $D \notin \{1,3,4\}$, $E \notin \{1,2,3\}$, $F \notin \{2,3,4\}$, $G \notin \{2,3,4\}$, $H \notin \{1,2,3\}$.

There are already a lot of Sudoku solvers currently available, but they are all written specifically for this task. The tool that is described in this thesis targets at generic CSPs, and therefore should be able to solve Sudokus as well, albeit maybe a bit less efficient than dedicated Sudoku solvers.

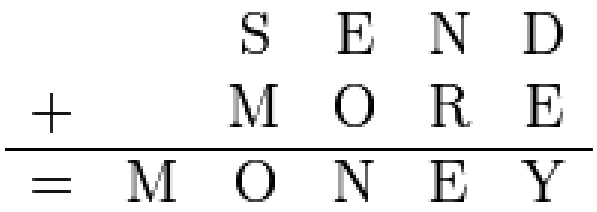


Fig. 2: Cryptarithmetic puzzle published in July 1924 by Henry Dudeney

**Example 3.2.** Cryptarithmetic puzzles.
Another type of puzzles that fit the CSP formalism perfectly are the so-called *Cryptarithmetic puzzles*. An example of such a puzzle is given in Fig. 2. These puzzles consist of $n$ words (where $n > 1$) and $n-2$ arithmetic operations between the first $n-1$ words followed by an equal sign and the last word, representing an equation. The words consist of no more than 10 distinct characters. The objective of the puzzle is to find for each letter a unique decimal digit, such that replacing the letters by the corresponding digits yields an arithmetically correct statement. For example, the solution of the puzzle SEND+MORE=MONEY is O = 0, M = 1, Y = 2, E = 5, N = 6, D = 7, R = 8, and S = 9. This corresponds with the arithmetically correct statement 9567 + 1085 = 10652.

This puzzle is easily translated into a CSP after introduction of some extra variables $X_1$, $X_2$, $X_3$, and $X_4$ which represent the carry that may occur in the addition. Note that the carry can only be 0 or 1:

| | |
|---|---|
| Variables: | $S, E, N, D, M, O, R, Y, X_1, X_2, X_3, X_4$ |
| Domains: | $D_S = D_E = D_N = D_D = D_M = D_O = D_R = D_Y = \{0,1,2,3,4,5,6,7,8,9\}$ |
| | $D_{X_1} = D_{X_2} = D_{X_3} = D_{X_4} = \{0,1\}$ |
| Constraints: | $D + E = Y + 10 \cdot X_1$, |
| | $X_1 + N + R = E + 10 \cdot X_2$, |
| | $X_2 + E + O = N + 10 \cdot X_3$, |
| | $X_3 + S + M = O + 10 \cdot X_4$, |
| | $M = X_4$, $S \neq 0$, $M \neq 0$ |

A typical way to solve the cryptarithmetic puzzle from Fig. 2 by hand goes as follows. The word MONEY starts with the letter M, and since M must be a carry (which can be 0 or 1) and leading zeroes are not allowed, it follows that M=1. Now, since M=1, we can conclude that S=9, otherwise there would not have been a carry $M = 1$. By successive reasoning in this style, one can obtain the solution of the puzzle.

Cryptarithmetic puzzles can be solved on a computer using a brute-force depth first search (DFS) in which recursively each letter is assigned a digit. The invariant of this recursive process is that the partial assignment until the current invocation of the solving procedure is consistent. As soon as an inconsistent state is reached, the algorithm performs *backtracking* to the last invocation point where a choice was made. Of course, this process starts with an empty assignment, and terminates when a complete assignment has been reached (which is the solution of the problem).

This *backtracking* process continues until a solution is found or all possible states have been tried without success. The potential size of the search space is $10^{10}$, since there can be at most ten letters, and each letter can be assigned one out of ten values. For the puzzle in Fig. 2 the state space consists

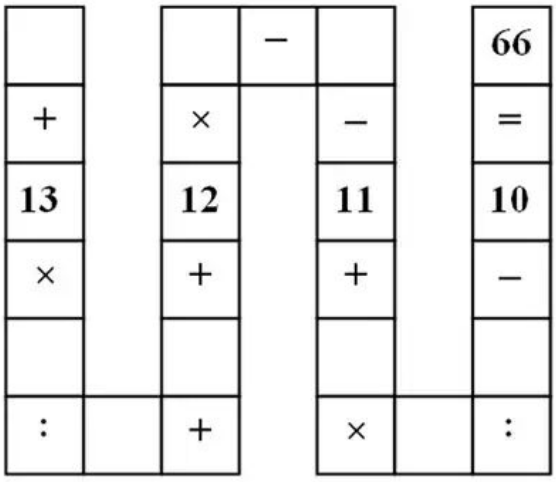


Fig. 3: The expression puzzle

of $10^8$ states, since there are only eight distinct letters.

In practice, not all possible states are reached by the recursive process, since the algorithm does not proceed in a certain direction, once it has found that a partial assignment is inconsistent. Nevertheless, the number of recursive invocations can be quite substantial. However, by using some clever *heuristics* (which we shall discuss later), the number of invocations can be reduced dramatically.

Interested readers are referred to a paper by Abbasian and Mazloom [2] in which techniques are described that are quite specific for solving cryptarithmetic puzzles.

**Example 3.3.** Expression puzzle.
Another benchmark we used is the puzzle depicted in Fig. 3. The object is to place the digits 1 to 9 in the empty cells, using each digit exactly once, to make it a valid equation. All arithmetic operations are integer operations and should return an integer. This puzzle can be translated into the following formal CSP:

| | |
|---|---|
| Variables: | $A, B, C, D, E, F, G, H, I$ |
| Domains: | $D_A = D_B = D_C = D_D = D_E = D_F = D_G = D_H = D_I = \{1,2,3,4,5,6,7,8,9\}$ |
| Constraints: | $A + (13 \cdot B)$ **div** $C + D + 12E - F - 11 + (G \cdot H)$ **div** $I - 10 = 66$, |
| | $(13 \cdot B)$ **mod** $C = 0$, $(G \cdot H)$ **mod** $I = 0$, |
| | $A \neq B$, $A \neq C$, $A \neq D$, $A \neq E$, $A \neq F$, $A \neq G$, $A \neq H$, $A \neq I$, |
| | $B \neq C$, $B \neq D$, $B \neq E$, $B \neq F$, $B \neq G$, $B \neq H$, $B \neq I$, |
| | $C \neq D$, $C \neq E$, $C \neq F$, $C \neq G$, $C \neq H$, $C \neq I$, |
| | $D \neq E$, $D \neq F$, $D \neq G$, $D \neq H$, $D \neq I$, |
| | $E \neq F$, $E \neq G$, $E \neq H$, $E \neq I$, |
| | $E \neq F$, $E \neq G$, $E \neq H$, $E \neq I$, |
| | $F \neq G$, $F \neq H$, $F \neq I$, |
| | $G \neq H$, $G \neq I$, |
| | $H \neq I$ |

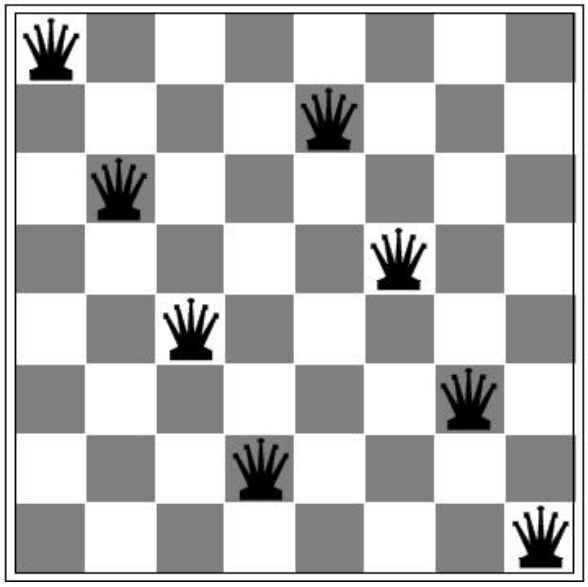

Fig. 4: A solution of the 8-queens problem

**Example 3.4.** *n*-queens problem.
The *n*-queens problem is a classic problem that is often used as an introductory programming exercise on recursion. The objective is to place $n$ queens on an $n \times n$ chessboard such that no two queens attack one another. In Fig. 4 one of the 92 solutions on a $8 \times 8$ chessboard is shown. The problem has been solved for several values of $n$. The number of solutions for $n \in [1..26]$ are currently known: the number of solutions for $n \in [1..14]$ is given in Table 1.

| $n$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| solutions | 1 | 0 | 0 | 2 | 10 | 4 | 40 | 92 | 352 | 724 | 2680 | 14200 | 73712 | 365596 |

Tab. 1: The amount of solutions for the *n*-queens problem for $n \in [1..14]$.

The recursive procedure for systematically solving the *n*-queens problem goes as follows. We start with an empty chessboard, and try to place in recursion level $k$ a queen on a square in column $k$, such that none of the queens in any column left of $k$ is attacked (i.e. the state is kept consistent). If a suitable place for a queen in column $k$ is found, then the process is recursively started for column $k+1$. Otherwise the process backtracks to (one of) the previous recursion level(s). Clearly, a solution is found in a recursive invocation for which $k = n$.

This analysis suggests the following translation of the n-queens problem into a formal CSP:

| | |
|---|---|
| Variables: | $Q_0, Q_1, \cdots, Q_n$ |
| Domains: | $D_{Q_i} = [0..n)$ for all $i \in [0..n)$ |
| Constraints: | $Q_i \neq Q_j$, for all $i,j \in [0..n) \wedge i \neq j$ (i.e. not on same row), |
| | $\lvert Q_i - Q_j \rvert \neq \lvert i - j \rvert$ , for all $i,j \in [0..n) \wedge i \neq j$ (i.e. not on same diagonal) |

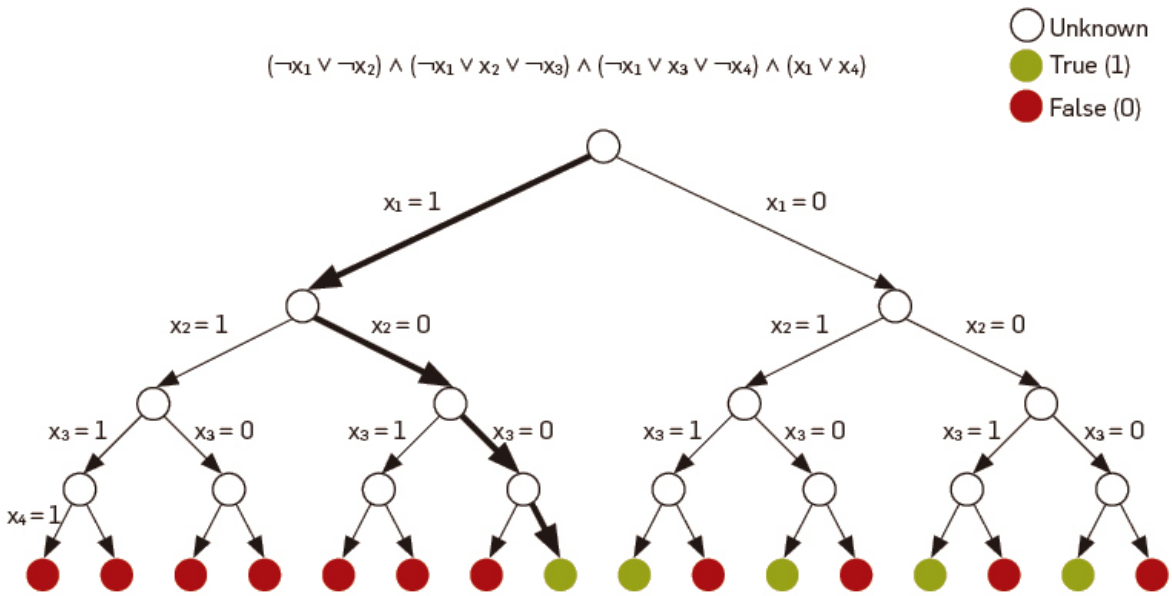


Fig. 5: Boolean Satisfiability Problem and corresponding search tree

**Example 3.5.** Boolean Satisfiability (SAT).
The last example that we consider is the *boolean satisfiability problem*. Given a formula from propositional logic, the problem concerns the question whether the atoms of the formula can be consistently replaced by the boolean values **true** or **false** in such a way that the formula evaluates to **true**.

For a formula containing $n$ different atoms, the number of possible assignments[1] is $2^n$. The satisfiability problem is known to be NP-complete. Since we can write each SAT problem easily as a formal CSP, in which the constraint is simply the logical formula itself, it is clear that solving CSPs in general is NP-complete as well.

The SAT problem can be solved using a brute-force recursive process which is depicted in Fig. 5. This particular example can be translated into the following CSP:

| | |
|---|---|
| Variables: | $X_1, X_2, X_3, X_4$ |
| Domains: | $D_{X_i} = \{\textbf{false}, \textbf{true}\}$ for all $i \in [1..4)$ |
| Constraint: | $(\neg X_1 \vee \neg X_2) \wedge (\neg X_1 \vee X_2 \vee \neg X_3) \wedge (\neg X_1 \vee X_3 \vee \neg X_4) \wedge (X_1 \vee X_4)$ |

[1] In the context of logic, such an assignment is called a *model*.

```
algorithm backtrackingSearch(csp)
  input:
    csp: representation of the CSP problem
  output: consistent complete set of (variable, value) pairs or failure
  return recursiveBacktracking({}, csp)

algorithm recursiveBacktracking(assignment, csp)
  input:
    assignment: a consistent set of (variable, value) pairs
    csp: representation of the CSP problem
  output: consistent complete assignment or failure
  if assignment is complete
    return assignment
  var ← selectUnassignedVariable(csp, assignment)
  foreach value from the domain of var
    if (var,value) is consistent with the constraints
      assignment ← assignment ∪ {(var, value)}
      result ← recursiveBacktracking(assignment, csp)
      if result ≠ failure
        return result
      assignment ← assignment \ {(var, value)}
  return failure
```

Fig. 6: Naive backtracking algorithm returning a solution if any exists, otherwise it returns *failure*.

## 4 Solving CSPs

### 4.1 Backtracking search

As we can read in [1], *backtracking search* is commonly used for solving CSPs. This is a depth-first search method in which a single variable gets assigned at each step in the backtracking process. When the algorithm wants to assign some value $x$ to a variable $v$, all constraints $c$ in which this variable $v$ is involved are checked for consistency with the value $x$ substituted for $v$. Hence, an invariant of the algorithm is that the partial assignment thus far is kept consistent. When the pair $(v, x)$ is indeed consistent with the partial assignment, the assignment $v := x$ is added to the partial assignment, and the solving procedure calls itself recursively.

If none of the values from the domain of $v$ is consistent with the partial assignment, then the algorithm returns failure at the current recursion level. This results in backtracking to previous recursion levels, where other choices for previously assigned variables are tried.

If at the top level of this backtracking process, no value can be found for the variable considered at that level, then the CSP has no solutions. On the other extreme, if at the beginning of a recursive invocation the assignment is found to be complete, then a solution is found and is returned immediately.

Note, that in each recursive forward step of this process, the number of unassigned variables decreases by one and thus the algorithm will eventually terminate. In Fig. 6, this algorithm is given in pseudocode.

The time complexity of this algorithm is clearly exponential. This is not a surprise, since we already noted that the SAT problem can be written as a CSP, and SAT is known to be NP-complete. Still, if we want to be a bit more exact about the time complexity of the algorithm, then the best we can say is that the number of recursive steps made by the algorithm is at most

$$\prod_{v \in Vars(csp)} \#Domain(v) \tag{1}$$

where $\#A$ denotes the size of a set $A$.

Note that in most cases this upper bound is not reached (by far). However, for CSPs that do not have a solution at all, or very few solutions which are discovered late by the recursive process, the number of recursive steps tend to get close to this upper bound.

## 4.2 Search heuristics

Although the number of recursive steps in (1) appears to be unacceptable at first sight, many solutions for large problems are still found by the brute-force algorithm given in Fig. 6 within a reasonable execution time. The effectiveness of the algorithm typically depends on the structure of the CSP that we try to solve. If the system has many solutions which are distributed uniformly over the search space, then the algorithm is likely to come up quickly with a solution. On the other hand, if the search space is populated sparsely with solutions, the running time may become unacceptable.

Fortunately, even for these hard problems we can still improve on the running time of the algorithm. The key lies in the following two lines of the pseudocode in Fig. 16.

*var* ← *selectUnassignedVariable*(*csp*, *assignment*)
**foreach** *value* from the domain of *var*
.....

The presented algorithm is very non-deterministic; in any step of the recursive process an unassigned variable is selected, which can be a random choice. However, the runtime of the algorithm depends on this choice, so we should choose wisely. For example, it is good to detect early that a certain branch of the search tree leads inevitably to failure.

Moreover, once a variable $v$ has been chosen at some recursion level, then the algorithm needs to iterate over the values from the domain of $v$. However, the order in which this is done, is again non-deterministic. Again, choosing a smart order may yield serious reductions in execution time.

So, the important questions to ask at each recursion level are:

- Which variable should be selected?
- Given a selected variable, in which order should we process the allowed values of this variable?

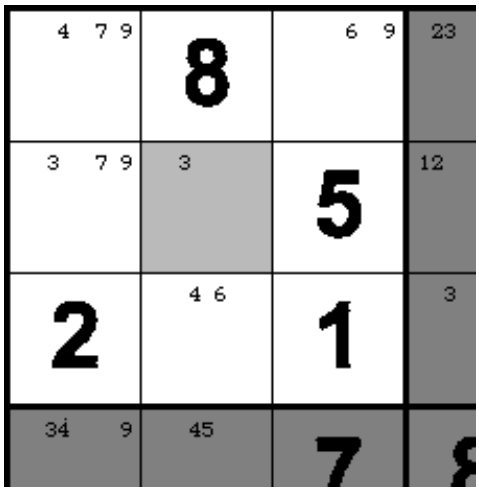


Fig. 7: Part of a sudoku puzzle containing a cell with a singleton domain

**Minimum Remaining Values heuristic**

The question "*Which variable should be selected?*" can be answered by a well-known heuristic called the *Minimum Remaining Values (MRV)* heuristic[2].

This heuristic makes the function *SelectUnassignedVariable* choose the variable which has the fewest number of allowed values in its domain.

To see that this is a good idea, consider the following trivial CSP:

Variables: $X_1, X_2, .., X_{100}$
Domains: $D_{X_i} = \{1,2,3,4,5,6,7,8,9,10\}$ for all $i \in [1..100)$,
$D_{X_{100}} = \{10\}$
Constraint: $X_i = X_{i+1}$ for all $i \in [1..100)$

Clearly, the solution of this CSP is simply $X_i = 10$ for all $i \in [1..100]$.
However, a brute-force recursion that would select variables in the order $X_1$, $X_2$, .., $X_{100}$ and tries values in sorted order would find the solution after having tried $10^{100} - 1$ inconsistent assignments.

On the other hand, an algorithm that starts at the top level of the recursion with variable $X_{100}$ will find the solution after 100 recursive steps (one step for each variable). The MRV heuristic makes sure that exactly that would happen.

This heuristic is actually used by most people who solve sudokus (or similar puzzles) by hand. For example, in Fig. 7 a situation is depicted in which the middle cell has only a single possible value left. Clearly, it is a good idea to directly assign the cell with this value.

Note that the MRV heuristic is about choosing the variable with the smallest domain, i.e. not about singleton domains per se. The important observation is that the recursive algorithm is actually traversing a search tree. Of this tree, we want nodes in the top (i.e. close to the root) to have the smallest possible branch factor, since this influences the number of recursive steps the most.

[2] In the literature, this heuristic is also known as the *Most Constrained Variable* heuristic

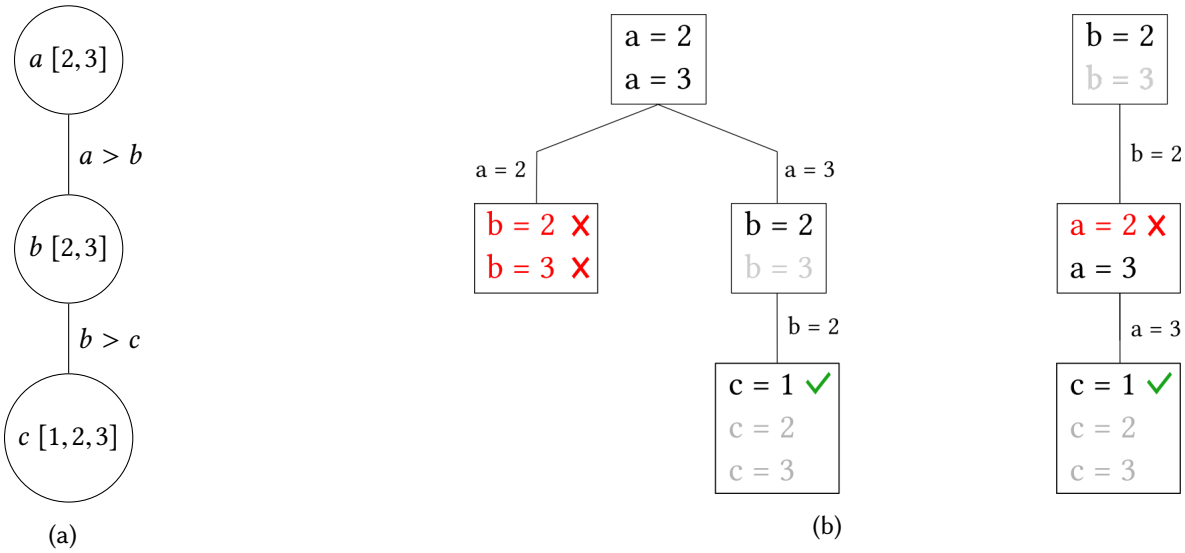


Fig. 8: Constraint graph CSP (a), search tree for MRV (b, left) and for MRV+DH (b, right).

**Degree heuristic**

The MRV heuristic does not give a definite answer which variable to choose in cases that there are at least two variables having the same minimal domain size. In principle, the system can select randomly from one of these variables, but even here we can reduce the number of recursive steps by making a smart choice. For example, consider the following simple CSP:

Variables: $a, b, c$
Domains: $D_a = \{2,3\}$, $D_b = \{2,3\}$, $D_c = \{1,2,3\}$
Constraint: $b < a, c < b$

At the top level of the recursive process, the MRV heuristic wil conclude that the algorithm should start with either the variable $a$ or $b$, since both variables have the smallest domain size.

The *degree heuristic*[3] now plays the role of a tie-breaker among the variables chosen by the MRV heuristic. The degree heuristic will choose from these variables the one that is involved in the largest number of constraints on other unassigned variables. This way, the heuristic attempts to reduce the branching factor on future choices.

The naming *degree* heuristic stems from graph theory. In the literature, it is common to depict CSPs as undirected graphs in which the nodes represent the variables, while edges denote the constraints. These graphs are called *constraint graphs*. In the above example, the variable $b$ is involved in constraints containing the variables $a$ and $c$, and therefore the graph contains an edge between $b$ and $a$, and an edge between $b$ and $c$. This graph is depicted in Fig. 8(a). Clearly, the variable $b$ has degree 2, and the variables $a$ and $b$ have degree 1. This means that the algorithm will start the solving process by trying to assign a value to the variable $b$ (rather than the variable $c$).

In Fig. 8(b), we see that indeed this choice reduces the number of recursive steps: the search tree that starts with the variable $b$ is indeed smaller than the tree that starts with the variable $a$.

[3] Which is also known as *Most Constraining Variable* heuristic

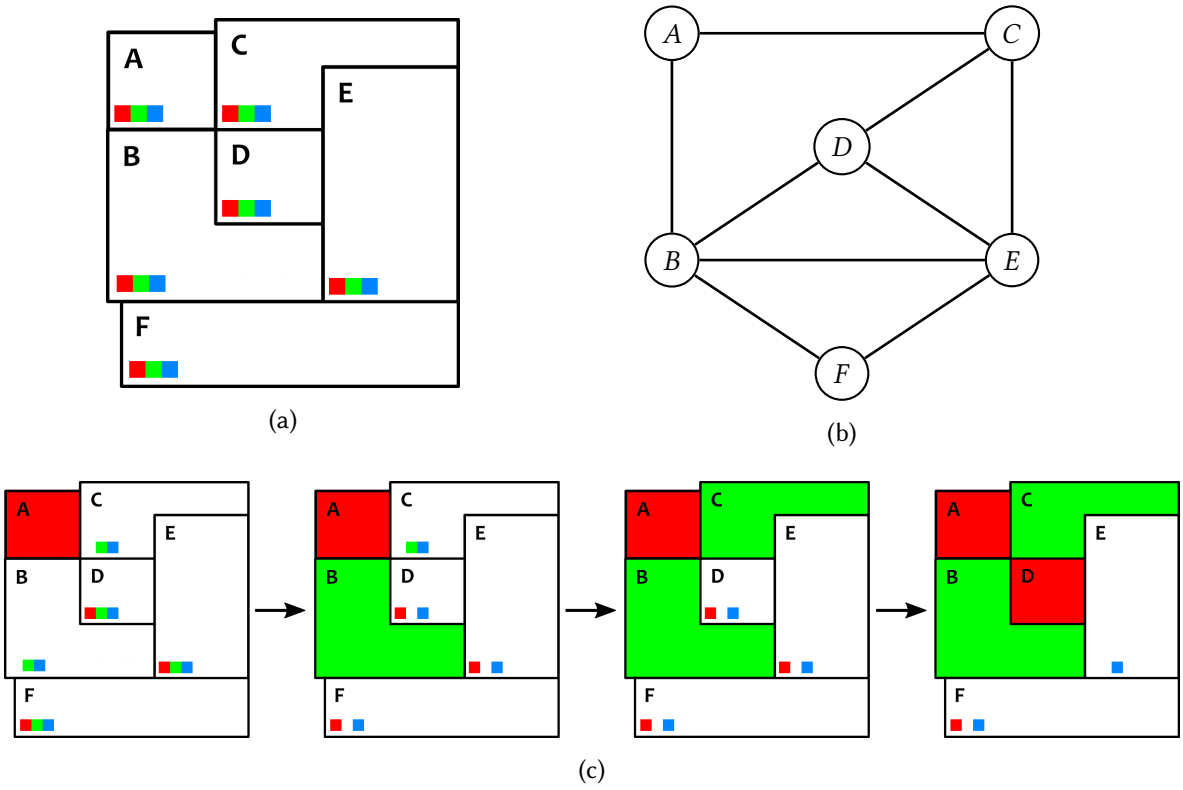


Fig. 9: Colouring problem (a), its constraint graph (b), corresponding forward checking process (c).

## 4.3 Domain reduction techniques

The methods discussed in subsection 4.2 are heuristics, since these are only guidelines to direct the search for a solution. These heuristics often reduce the number of recursive steps significantly, but they do not guarantee this reduction. In fact, in the worst case, these heuristics might even steer the search in the wrong direction, yielding even an increment of the number of recursive steps.

In this section, two other techniques are discussed: *forward checking* and *arc consistency*. These techniques are not heuristics, since they never increase the number of recursive steps. In the worst case, these techniques keep the number of recursive steps the same.

### Forward checking

The number of backtracking steps can be reduced by eliminating parts of the search tree for which it is clear that these inevitably lead to failure. In the discussion this far, given a partial assignment and a possible addition $v := x$ under consideration, the algorithm checks whether this new assignment is consistent with the partial assignment thus far. In the literature on CSPs, this is called *local consistency*.

The forward checking technique goes further than that. When some value $x$ is chosen for some variable $v$, not only local consistency is checked, but also the effect of the assignment $v := x$ on each other unassigned variable $w$ that are involved in constraints on $v$. For example, if we have a CSP

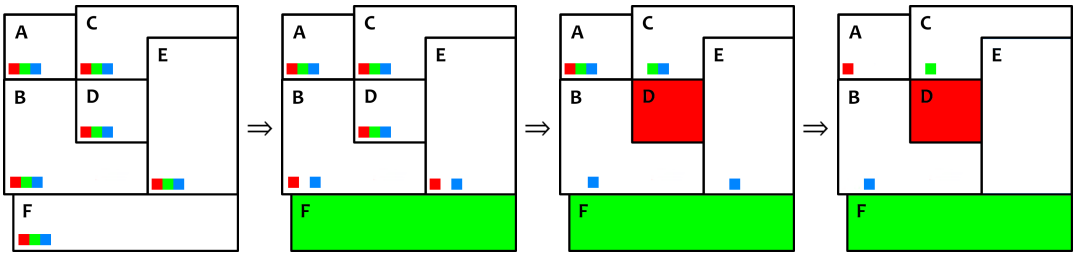


Fig. 10: Arc consistency preserved for colouring problem of Fig. 9.

with the constraint $a < b$, and the domain of $a$ is $\{1, 2, 3, 4\}$, and we consider the assignment $b := 2$, then it is clear that the algorithm can reduce the domain of $a$ directly to the singleton $\{1\}$.

This technique can reduce the number of recursive steps, since the branching factor of the search tree is determined by the sizes of the domains of the variables. Hence, we should reduce the sizes of these domains as much as possible.

If, as a result of forward checking, the domain of some variable becomes empty then it is clear that traversing the current branch of the tree does not lead to a solution; the conclusion must be that the partial assignment will lead to inconsistencies in the future.

In Fig. 9, the forward checking process is depicted graphically for a region colouring problem. The objective is to colour the regions such that no adjacent regions have the same colour, using only the colours red, green, and blue. In this figure, the algorithm decided to colour region $A$ red[4]. A consequence of this choice, is that the regions $B$ and $C$ can no longer be red, and thus this value is removed from their domains. In the next step, $B$ is coloured green, so green is removed from the domains of $D$, $E$, and $F$. And so on.

### Arc consistency

In many cases, forward checking is able to detect early that a partial assignment will lead to a failure. However, some clear future failures are not detected by forward checking. An example of this is given in Fig. 10. In this example, first the region $F$ is coloured green, and forward checking therefore removes green from the domains of $B$ and $E$. Now, let us say that variable $D$ is coloured red (even though the heuristics would choose another variable). Forward checking now removes the value red from the domains of $B$, $C$, and $E$. However, we are now in a state that is still local consistent, but it cannot lead to a solution. The problem is that the neighbouring regions $B$ and $E$ now both have the singleton domain {blue}, and thus there is no need to search any further along this line.

A stronger technique, that does discover this problem, is called *arc consistency*. The idea of arc consistency is the following. Let $v$ be a variable with the corresponding domain $V$. Let $w$ be another variable with the corresponding domain $W$, which is involved in a binary constraint $C$ on $v$ (for example $v \neq w$). Then, the constraint is arc consistent in the direction $v \to w$, if for each value assignment $v := a$ (where $a \in V$) there exists an assignment $w := b$ (where $b \in W$) such that the

[4] Note that the degree heuristic would have started with region $B$, but that does not matter for the discussion on forward checking.

constraint $C$ is satisfied. If such a $b$ does not exist for a given $a$, it means that the assignment $v := a$ will lead to a future failure. Therefore, $a$ is removed from the domain $V$.

Note that the naming of this technique refers to arcs, instead of edges. This is due to the fact that the domain $V$ is pruned based on possible values in the domain $W$. In practice, this technique is applied in both directions. So, given an edge $(v, w)$ in the constraint graph, this edge is considered as two arcs: $v \to w$ and $w \to v$.

In the example in Fig. 10, arc consistency would detect directly after the assignment $D :=$ red that there is an inconsistency for the arc $E \to B$.

## 5 Applications

The examples that we discussed this far are small toy instances of CSPs. In real life, we often encounter serious and larger problems. A standard real-life example is the scheduling problem, of which many subtypes exists. For example, a scheduling problem for a large factory with lots of different stages of production, many employees and different deadlines for finished products can be translated into a standard (albeit large) CSP. Solving this CSP by hand is infeasible, so automatic solvers can play an important role here. Of course, this specific example is also an optimization problem: given all solutions, return the optimal one according to some objective function. This function is often formulated as a constraint itself. Such constraints are so-called *soft constraints*. This thesis deals only with CSPs in which variables have finite domains and all constraints are hard, i.e. we do not consider soft constraints any further.

Some other applications of CSPs are:

- Solving all kinds of puzzles (e.g. Sudoku, Logikwiz)
- Solving systems of linear equations
- Laying out a silicon chip

These are just some examples of problems that can be translated into a CSP, but there are many more. Many problems, even though it is not directly clear at first, can be translated into the formalism of CSPs.

## 6 Goal of this project

The goal of this project is to build a tool that, given the specification of a CSP, is able to solve this CSP in a generic way. Moreover, a selection of heuristics are implemented and the results are compared with a naive backtracking implementation. A user is able to define the problem by defining the set of variables, the domains of these variables, and the constraints that bind and limit the values of the variables. Given this definition, the program must be able to compute the solutions of the problem, if these exist. The CSPs that the tool will be able to solve are CSPs with variables that have a finite domain and constraints that are all hard.

### 6.1 Definition language

In order to be able to specify a CSP in such a way that the tool is able to interpret and solve it, we need to define a small programming language in which a CSP can be specified. A grammar must be constructed that defines this language. This language should be easy to understand for a human being and sufficiently expressive to specify complicated and large CSP problems.

### 6.2 Assessment of optimizations

In order to assess the effectiveness of the discussed optimizations, we start with a naive backtracking solver in which none of these optimizations are implemented. Moreover, this version of the solver can serve as a starting point for the final solver by successively adding optimizations. In both versions of the program a counter is introduced which counts the number of recursive steps. This counter is used for comparing the effectiveness of the proposed optimizations.

We want to investigate the effectiveness of the standard heuristics Minimum Remaining Values (MRV) and the Degree Heuristic (DH). Moreover, we want to study the effectiveness of Forward Checking (FC) and Arc Consistency (AC). Besides that, we also want to investigate if we can come up with some effective new heuristics.

## 7 Implementation

In this section we will explain some of the implementation details of the solver. We will also address some design decisions that we made. The complete code can be found in appendix B.

### 7.1 Language definition

We want to develop a programming language in which CSPs can be specified. A CSP written in this language should be easy to read and learn. Any CSP can be written in the formalism as defined in section 2, but this formalism is not a convenient notation for many problems. For example, a problem in which the variables $X_0, X_1, ..., X_n$ need to have different values, would need $n(n+1)/2$ constraints of the type $X_i \neq X_j$. A much more user-friendly notation is to introduce arrays of variables, and to allow first order logic expressions (i.e. quantifications). In such a notation, the given example would be expressed by a single meta-constraint like *alldiff(X)*, which is expanded by the solver into the corresponding $n(n+1)/2$ inequalities. We dubbed this more expressive language CSP-DL (CSP Definition Language).

On the other hand, by introducing this kind of syntactic sugar, the implementation of the solver would get much more complex. Besides that, we wanted to avoid spending our time on building a compiler, and wanted to focus on solving CSPs. For this reason, we decided to split the program into two phases: the first phase accepts CSPs defined in the rich language CSP-DL, and converts it into a much simpler version of the language, that is based on the definition given in section 2. We dubbed this simpler version CSP-DLNF, where NF stands for *normal form*. The grammars for the languages

| | | |
|---|---|---|
| *csp* | → | *body* |
| *body* | → | *vars domains constraints solvespec* |
| *vars* | → | **variables :** [ *vardeflist* **:** *datatype* **;** ]* |
| *datatype* | → | **integer** \| **boolean** |
| *domains* | → | **domains :** [ *domainspec* **;** ]* |
| *domain* | → | [ **[** \| **(** ] *subdomain* [ **,** *subdomain* ]* [ **]** \| **)** ] |
| *subdomain* | → | *numExp* [ **..** *numExp* ]? |
| *integer* | → | [ 0-9 ]+ |
| *constraints* | → | **constraints :** [ *constraintspec* **;** ]* |
| *constraintlist* | → | *constraint* [ **,** *constraint* ] * |
| *solvespec* | → | **solve :** [ **all** \| integer ] |
| *varname* | → | [ a-zA-Z_ ][ a-zA-Z0-9_ ]* |
| *indexspec* | → | **[** *integer* **]** |
| *indexcalc* | → | **[** *numexp* **]** |
| *vardef* | → | *varname indexspec** |
| *varcall* | → | *varname indexcalc** |
| *functioncall* | → | [ **max** \| **min** ] **(** *numexp* **,** *numexp* **)** |
| | | \| [ **all** \| **any** ] **(** *constraintlist* **)** |
| | | \| **abs (** *numexp* **)** |
| | | \| **alldiff (** *numexp* [ **,** *numexp* ]* **)** |
| *varlist* | → | *varcall* [ **,** *varcall* ]* |
| *vardeflist* | → | *vardef* [ **,** *vardef* ]* |
| *forallspec* | → | **forall (** *varname* **in** *domain* **)** |
| *domainspec* | → | *forallspec domainspec** **end** \| *varlist* **<-** *domain* **;** |
| *constraintspec* | → | *forallspec constraintspec** **end** \| *constraint* **;** |
| *constraint* | → | *numexp* [ *relop numexp* ]? |
| *numexp* | → | *term* [ *termop term* ]* |
| *term* | → | *factor* [ *factorop factor* ]* |
| *factor* | → | *value* \| **-** *factor* |
| *value* | → | [ *integer* \| *varcall* \| *functioncall* \| **(** *numexp* **)** ] [ **∧** *factor* ]? |
| *relop* | → | **=** \| **<** \| **>** \| **<>** \| **<=** \| **>=** |
| *termop* | → | **-** \| **+** |
| *factorop* | → | ***** \| **mod** \| **div** |

Fig. 11: Grammar of CSP-DL

| | | |
|---|---|---|
| *csp* | → | *body* |
| *body* | → | *vars domains constraints solvespec* |
| *vars* | → | **variables :** [ *varlist* **:** *datatype* **;** ]* |
| *datatype* | → | **integer** \| **boolean** |
| *domains* | → | **domains :** [ *domainset* **;** ]* |
| *domain* | → | [ **[** \| **(** ] *subdomain* [ **,** *subdomain* ]* [ **]** \| **)** ] |
| *subdomain* | → | *integer* [ **..** *integer* ]? |
| *integer* | → | [ 0-9 ]+ |
| *constraints* | → | **constraints :** [ *constraint* **;** ]* |
| *constraintlist* | → | *constraint* [ **,** *constraint* ] * |
| *solvespec* | → | **solve :** [ **all** \| integer ] |
| *var* | → | **X** integer |
| *functioncall* | → | [ **max** \| **min** ] **(** *numExp* **,** *numExp* **)** |
| | | \| **abs (** *numExp* **)** |
| | | \| [ **all** \| **any** ] **(** *constraintlist* **)** |
| *varlist* | → | *var* [ **,** *var* ]* |
| *domainset* | → | *varlist* **<-** *domain* |
| *constraint* | → | *numExp* [ *relop numExp* ]? |
| *numExp* | → | *term* [ *termop term* ]* |
| *term* | → | *factor* [ *factorop factor* ]* |
| *factor* | → | *value* \| **-** *factor* |
| *value* | → | [ *integer* \| *var* \| *functioncall* \| **(** *numExp* **)** ] [ **∧** *factor* ]? |
| *relop* | → | **=** \| **<** \| **>** \| **<>** \| **<=** \| **>=** |
| *termop* | → | **-** \| **+** |
| *factorop* | → | ***** \| **mod** \| **div** |

Fig. 12: Grammar of CSP-DLNF

```
variables:
 chessboard[4] : integer;

domains:
 forall(i in [0..3])
  chessboard[i] <- [0..3];
 end

constraints:
 alldiff(chessboard);

 forall(i in [0..3])
  forall(j in [i+1..3])
   abs(chessboard[i]-chessboard[j]) <> abs(i-j);
  end
 end

solutions: 1
```

(a)

```
variables:
 X0, X1, X2, X3 : integer;

domains:
 X0, X1, X2, X3 <- [0..3];

constraints:
 X0 <> X1;
 X0 <> X2;
 X0 <> X3;
 X1 <> X2;
 X1 <> X3;
 X2 <> X3;

 abs(X0-X1) <> abs(0-1);
 abs(X0-X2) <> abs(0-2);
 abs(X0-X3) <> abs(0-3);
 abs(X1-X2) <> abs(1-2);
 abs(X1-X3) <> abs(1-3);
 abs(X2-X3) <> abs(2-3);

solutions: 1
```

(b)

Fig. 13: 4-queens problem defined in CSP-DL (a) and in CSP-DLNF (b)


Fig. 14: Pipeline of the solving process.

CSP-DL and CSP-DLNF are given in respectively Fig. 11 and Fig. 12. Note that the grammar of CSP-DLNF is a subgrammar of CSP-DL.

An example of a translation of a CSP specification written in CSP-DL into an equivalent specification in CSP-DLNF is fiven in Fig. 13. It is easy to see that `forall` and `alldiff` constraints can be translated into a format in which all cases are given explicitly (the normal-form).

This separation of concerns is depicted in Fig. 14. The first phase accepts input written in CSP-DL, and converts it into an output written in CSP-DLNF. The real solver accepts this output as its input, and converts it into data structures that represent the CSP internally. On these data structures, the real solving process is applied.

## 7.2 Solver

In this section we explain how the solving part of our tool is implemented. First the naive backtracking implementation is outlined and after that the introduction and implementation of several

```
typedef struct problem *Problem;

typedef struct problem {
  int varCount;
  int assignCount;
  int constraintCount;
  Variable *vars;
  VarSeq varSequence;
  Constraint *constraints;
  SolveSpec solvespec;
} problem;
```

(a)

```
typedef struct solutionSet *SolutionSet;
typedef struct solutionList *SolutionList;

typedef struct solutionList {
  int *values;
  SolutionList next;
} solutionList;

typedef struct solutionSet {
  SolutionList first;
  SolutionList last;
  int varAmount;
  int solutionSpace;
  int solutionCount;
} solutionSet;
```

(b)

Fig. 15: Data structures `Problem` (a) and `SolutionSet` (b)

heuristics and domain reduction techniques is evaluated.

### 7.2.1 Naive backtracking algorithm

For the implementation of the naive backtracking algorithm, we followed the pseudocode implementation given in Fig. 6 for the most part. However, our implementation differs with regard to the stop criterion. In our implementation the stop criterion is not whether a solution has been found, since the language CSP-DL allows the user to specify how many solutions (possibly all) he/she is interested in. Therefore, our version of the algorithm stops if it found the specified number of solutions, or when it traversed through the entire search space.

The concrete implementation in C of the naive backtracking algorithm is given in Fig.16. The arguments of the algorithm are a `Problem` (the CSP) and a `SolutionSet` (set of complete consistent assignments). The definition of these data structures is given in Fig. 15.

### 7.2.2 Selecting unassigned variables

For the heuristics that are used for selecting variables, we had to come up with a data structure to store a sequence of variables. This sequence represents the order in which the variables are considered by the recursive solving process. However, during this recursive process, this sequence must be rearranged continuously. When an element changes position in this sequence, it must be easy to restore it at its original location, when the recursion unfolds.

We decided to use a doubly linked list data structure for storing sequences of variables. The datatype `VarPos` (see Fig. 17) is a pointer to a node in this list. The actual data about a variable is stored in the `variable` data structure.

We use a doubly linked list because this makes it easy to move a variable to a specific position in the sequence in constant time. However, finding the right location takes linear time. Therefore,

```
void recursiveBacktracking(Problem p, SolutionSet solset) {
  Variable var;
  int *values, i;
  Domain fullDomain;

  if(p->varCount == p->assignCount) {
    addSolution(solset, p);
    return;
  }

  var = selectUnassignedVar(p);
  fullDomain = getVarDomain(var);
  values = getDomainValues(fullDomain);

  for(i = 0; i < getDomainSize(fullDomain); i++) {
    p->assignCount++;
    setVarDomain(var, getSingletonDomain(values[i]));
    if(checkLocalConsistency(var, p)) {
      recursiveBacktracking(p, solset);
    }
    freeDomain(getVarDomain(var));
    if(solset->solutionSpace == solset->solutionCount) {
      return;
    }
  }
}
```

Fig. 16: Implementation of the naive backtracking algorithm

```
typedef struct variable *Variable;
typedef struct varPos *VarPos;
typedef struct varSeq *VarSeq;

typedef struct variable {
  int index;
  int constraintDegree;
  int connectivity;
  int varConnections;
  DataType type;
  IntegerSet domain;
  IntegerSet constraints;
  VarPos sequencePos;
} variable;

typedef struct varPos {
  Variable var;
  VarPos prev, next;
} varPos;

typedef struct varSeq {
  VarPos first;
} varSeq;
```

Fig. 17: Data structure `Variable, VarPos, VarSeq`

moving a variable in the sequence has a time complexity of $O(n)$, where $n$ denotes the number of variables in the list.

Once a location has been found and a variable has been moved, restoring a variable $v$ at its old position has a constant time complexity: this operation is just a matter of rearranging the pointers *v.left*, and *v.right* and backing up some old values of these pointers.

### 7.2.3 Minimum Remaining Values

The Minimum Remaining Values (MRV) heuristic is implemented using the variable sequence that was outlined in the previous subsection. We need a method that determines at which positions in the sequence variables should be placed. For MRV this is based on the domain size of a variable. At every recursion level the first variable in the sequence (the variable with the smallest domain) $v$ is selected. This variable is assigned a value from its domain and removed from the sequence. When forward checking is applied or arc consistency maintained after this assignment, each variable $w$ from which the domain is reduced is reordered in the sequence. Moreover, before this reordering, a backup of the old sequence position of $w$ is made. This enables us to restore $w$ to its old position very fast when the assignment to $v$ turns out to be invalid.

### 7.2.4 Degree heuristic

The degree heuristic is used as a tie-breaker in combination with MRV, but the implementation does not differ much. This is because we are still dealing with a sequence of variables in which variables are ordered based on a heuristic. The only differences are the positions of the variables and the moment that the variables should be reordered. When applying MRV, the order of the variables can change after a domain reduction, but for this heuristic the sequence should also be reordered when after an assignment to a variable $v$, one or more constraints become unary. The degree of variables in these unary constraint is lowered to one.

### 7.2.5 Most Connected Variable heuristic

Another heuristic that can be helpful in many cases is the Most Connected Variable heuristic (MCV). This heuristic selects the variable that is most connected to the partial assignment thus far. To illustrate this heuristic we use the Sudoku puzzle as an example. Suppose we have three variables $v$, $w$, and $z$. The variables $v$ and $w$ are in the same subgrid and in the same row, where $z$ is not. Now suppose $v$ is assigned some value, and after this assignment $w$ and $z$ both have the same domain size. Variable $w$ is more connected to $v$ than $z$. If the assignment of $v$ leads to failure, this will be detected earlier (less recursion steps) by assigning $w$ before assigning $z$. We came up with this heuristic ourselves and implemented it in such a way that it can be applied apart or as a tie-breaker in combination with MRV.

```
int forwardChecking(Variable var, Problem p, Backup backup) {
  Queue arcQueue = emptyQueue();

  /* enqueue all arcs directed at var, so all (X --> var) */
  addVariableArcs(var, arcQueue, p);

  /* for all arcs in queue */
  while(!isEmptyQueue(arcQueue)) {
    /* dequeue arc */
    DirectedArc arc = dequeue(arcQueue);
    Constraint c = constraintOfArc(arc);
    Variable var1 = firstVarOfArc(arc);

    /* make backup before reduction */
    IntegerSet domBackup = copyIntegerSet(domainOfVar(var1));
    varPos seqBackup = *(sequencePosition(var1));

    /* check if domain of variable can be reduced by arc */
    int reduced = arcReduce(arc, p);
    freeArc(arc);

    /* if domain of variable is reduced by arc */
    if(reduced) {
      /* add backup of old domain and sequence position */
      addBackup(var1->index, domBackup, seqBackup, backup);
      /* resort variable in sequence */
      resortVarSeq(p->varSequence, sequencePosition(var1));
      int domSize = domainSizeOfVar(var1);
      /* if domain of variable became empty after reduction */
      if(domSize == 0) {
        makeArcQueueEmpty(arcQueue);
        freeQueue(arcQueue);
        /* return error */
        return 0;
      }
    } else {
      /* backup not needed */
      freeIntegerSet(domBackup);
    }
  }
  freeQueue(arcQueue);

  /* queue is empty and no error occured */
  return 1;
}
```

Fig. 18: Implementation of Forward checking

### 7.2.6 Domain reduction techniques

#### Forward checking

The FC technique is implemented in the function *forwardChecking* (see Fig. 18). Forward checking is the process which is started after assigning some value to a variable $v$. This assignment has consequences for the domains of other variables $w$ that are involved in binary constraints on $v$ and $w$. The domains of these variables $w$ need to be updated.

The forward checking process starts with building a queue (actually, a set would work just as well) of arcs $(v, w)$ that must be investigated. Next, arcs are dequeued one by one and for each arc $(v, w)$ it is checked if the domain of variable $w$ can be reduced given the new value for $v$. The method *arcReduce* checks for each value $x$ in the domain of $w$ if $x$ is consistent with the assignment of $v$. If it is not, the value $x$ is removed from the domain of $w$.

Since we are dealing with a recursive process, that we should be able to unwind, a backup of the domain of $w$ and its position in the variable sequence is made. If the domain of $w$ is reduced, this backup is added to the backups collected thus far and $w$ is relocated in the variable sequence if needed. If the domain of $w$ becomes empty, the method *forwardChecking* returns failure. If *forwardChecking* returns failure, the backups created after the assignment of $v$ are restored.

#### Arc consistency

In our implementation of arc consistency we use two different methods that both make the problem arc consistent but with different preconditions. Both methods make the given problem *strongly arc consistent*. This means that the problem is first made *node consistent*; the domains of all variables in unary constraints are made consistent with those constraints. Because this process may lead to domain reductions, it affects the process of making the problem arc consistent in which all binary constraints are made consistent.

The first function *makeArcConsistent* (see Fig. 19) does not have any preconditions. It just accepts a problem as its input, and converts the problem into a strongly arc consistent problem. So, this function can be called at any moment. The algorithm visits all arcs of the problem, and therefore it is too costly to perform during the recursive solving process. For this reason, we only use this function before the start of the recursive backtracking process.

The other function is called *mac* (see Fig. 20). This function does not investigate all arcs of the problem, but only the arcs that are affected by some assignment to a variable $v$. So, this method is called after each variable assignment. The precondition for the method *mac* is that the given problem is node consistent except for the new unary constraints in which $v$ was involved before assignment. Moreover, the problem needs to be arc consistent except for the new binary constraints in which $v$ was involved before assignment.

In the function *mac* we also use a queue and just like in *forwardChecking* we first enqueue all arcs that were directed at $v$ before assignment (so the old binary constraints involving $v$) are enqueued. We also enqueue two arcs for each binary constraint that had arity 3 before the assignment of $v$. While dequeuing, first all newly unary constraints will be satisfied, making the problem node consistent again. This is actually the same process as performed in *forwardChecking*. The difference

```
void makeArcConsistent(Problem p) {
  int i;
  int *varIndices;
  Queue arcQueue = emptyQueue();

  /* first the CSP is made node-consistent */
  makeNodeConsistent(p);

  /* for each constraint of p */
  for(i = 0; i < p->constraintCount; i++) {
    Constraint constraint = constraintByIndex(p, i);
    /* get indices of variables occurring in constraint */
    varIndices = varIndicesOfConstraint(constraint);
    /* if constraint is a binary constraint */
    if(arityOfConstraint(constraint) == 2) {
      Variable var1 = varByIndex(p, varIndices[0]);
      Variable var2 = varByIndex(p, varIndices[1]);

      /* enqueue directed arc (var1 --> var2) */
      enqueue(arcQueue, makeArc(var1, constraint, var2));
      /* enqueue directed arc (var2 --> var1) */
      enqueue(arcQueue, makeArc(var2, constraint, var1));

      /* directional arc (var1 --> var2) is in queue */
      inArcsQueue[constraint->index][varIndices[0]] = 1;
      /* directional arc (var2 --> var1) is in queue */
      inArcsQueue[constraint->index][varIndices[1]] = 1;
    }
  }

  /* until queue is empty */
  while(!isEmptyQueue(arcQueue)) {
    DirectedArc arc = dequeue(arcQueue);
    Variable var1 = firstVarOfArc(arc);
    Constraint c = constraintOfArc(arc);
    /* arc is not in queue anymore */
    inArcsQueue[indexOfConstraint(c)][indexOfVar(var1)] = 0;

    if(arcReduce(arc, p)) {   /* if domain reduction */
      /* if domain of var became empty -> error, no solution exists */
      if(domainSizeOfVar(var1) == 0) {
        printf(
          "\nNo solutions found for the problem, \
          because variable with empty domain\n"
        );
        exit(-1);
      }
      /* because domain is reduced, domains of neighbours might be reduced */
      addVariableArcs(var1, arcQueue, p);
    }
    freeArc(arc);
  }

  freeQueue(arcQueue);
}
```

Fig. 19: Implementation of method making a CSP arc consistent

```
/*
  Function that performs constraint propagation after assignment
  If CP is set to MAC, then arc-consistency is maintained.
*/
int mac(Variable var, Problem p, Backup backup) {
  Queue arcQueue = emptyQueue();

  /* enqueue all arcs directed at var, so all (X --> var) */
  addVariableArcs(var, arcQueue, p);

  /* enqueue all new arcs (two directions of constraints that had arity 3
      before assignment of var, but became binary after) */
  addNewArcs(var, arcQueue, p);

  /* for all arcs in queue */
  while(!isEmptyQueue(arcQueue)) {
    /* dequeue arc */
    DirectedArc arc = dequeue(arcQueue);
    Constraint c = constraintOfArc(arc);
    Variable var1 = firstVarOfArc(arc);

    /* set arc 'not available' */
    inArcsQueue[indexOfConstraint(c)][indexOfVar(var1)] = 0;

    /* make backup before reduction */
    IntegerSet domBackup = copyIntegerSet(domainOfVar(var1));
    varPos seqBackup = *(sequencePosition(var1));

    /* check if domain of variable can be reduced by arc */
    int reduced = arcReduce(arc, p);
    freeArc(arc);

    /* if domain of variable is reduced by arc */
    if(reduced) {
      /* add backup of old domain and sequence position */
      addBackup(var1->index, domBackup, seqBackup, backup);
      /* resort variable in sequence */
      resortVarSeq(p->varSequence, sequencePosition(var1));
      int domSize = domainSizeOfVar(var1);
      /* if domain of variable became empty after reduction */
      if(domSize == 0) {
        makeArcQueueEmpty(arcQueue);
        freeQueue(arcQueue);
        /* return error */
        return 0;
      }
      /* apply constraint propagation for variable with reduced domain */
      addVariableArcs(var1, arcQueue, p);
    } else {
      /* backup not needed */
      freeIntegerSet(domBackup);
    }
  }
  freeQueue(arcQueue);

  /* queue is empty and no error occured */
  return 1;
}
```

Fig. 20: Implementation of method maintaining arc consistency

with the latter function is that, if a domain reduction of a variable $w$ takes place, all arcs $z \rightarrow w$ are enqueued, because the domain of each $z$ might also be reduced. This addition takes into account the effect of making a problem node consistent onto making the problem arc consistent. If a domain of a variable $w$ is reduced while enforcing node consistency, this reduction might affect the variables connected to $w$ by a binary constraint. So, the process performed by *mac* has actually the structure of a breadth first search. In the literature on CSPs, inductively enforcing arcs consistent is called *contraint propagation*.

It is pointless to enqueue arcs more than once. Therefore, we keep track of which arcs are in the queue. If an arc is already in the queue, we simply do not enqueue it again. We use a two-dimensional array to administrate this. This allows us to query or set the presence of an arc in the queue in constant time.

In *mac*, for the same reasons as in *forwardChecking*, a backup of a variable is made before reducing its domain. If a domain becomes empty, *mac* returns failure. If the queue gets empty and none of the domain reductions led to an empty domain, *mac* returns success.

#### Backing up variables

After an assignment to some variable $v$, the forward checking or arc consistency routine reduces the domains of other variables $w$. However, since the assignment to $v$ may lead to an empty domain for $w$ or failure later on in the recursive process, we should be able restore the old domain of $w$ when we encounter an empty domain or when the process unwinds the recursion. Therefore, we make a backup of the domain of $w$, which can be restored when domain reduction leads to failure or when backtracking. So, before the domain of $w$ gets reduced, we add a backup of the current domain of $w$ to a `Backup` instance, and rearrange the sequence of variables (order in which variables are picked, see subsection 7.2.2). We also make a backup of the current location of $w$ in this sequence, before relocating it. This saves a lot of searching for locations in the variable sequence when restoring variable backups.

These backups are saved in a sort of stack structure where every new backup is pushed on top of the other backups. Restoring is done by popping a backup from the stack, and restoring it as the new current state. By applying this strategy, we can undo every step in constant time, and in the right order. The last modification to a variable $w$ is undone first and this is repeated until we arrive at the state we were in before we started the constraint propagation process. It could be that a variable occurs more than once in the `Backup` instance, but its states are restored in a reverse order which will eventually result in a state of the variable that is identical to the state in which it was before the reduction process started. For the implementation of the `Backup` instance and all functions related to backing up and restoring variables, we refer to the code in Appendix B.2.

# 8 Benchmarking results

We used the following problems as a benchmark to compare the effectiveness of the (combinations of) techniques and heuristics:

- Sudoku puzzles (see section 3.1)
- A cryptarithmetic puzzle (see section 3.2)
- The expression puzzle (see section 3.3)
- The 8-queens problem (see section 3.4)

We focused on the number of recursive calls made by the recursive solving procedure. All results are presented in bar charts. In order to understand these charts, we first define the abbreviations that are used in these charts:

| | |
|---|---|
| **NONE** | The plain naive backtracking algorithm is used. |
| **NC** | Before the backtracking process starts, the problem is made *node consistent* (all domains are pruned in accordance with the unary constraints). |
| **AC** | Before the backtracking process starts, the problem is made *strongly arc consistent* (all domains are in accordance with the unary/binary constraints). |
| **MRV** | For selecting unassigned variables, the *Minimum Remaining Values* heuristic is used. |
| **MCV** | For selecting unassigned variables, the *Most Connected Variable heuristic* is used. |
| **MRV+DH** | For selecting unassigned variables, the *Minimum Remaining Values heuristic* is used together with the degree heuristic as a tie-breaker. |
| **MRV+MCV** | For selecting unassigned variables, the *Minimum Remaining Values heuristic* is used together with the *Most Connected Variable heuristic* as a tie-breaker. |
| **FC** | Besides checking local consistency relative to the current partial assignment, *Forward Checking* for binary constraints is also performed. |
| **MAC** | After each variable assignment, the remaining problem is made *strongly arc consistent.* |

## 8.1 Sudoku puzzle

In Fig. 21 the number of recursive calls needed to solve a specific Sudoku puzzle (see Appendix A.1) is given for each benchmarked combination of techniques. The vertical axis reflects the number of recursive calls, while the horizontal axis reflects the techniques used. Note that bars are missing for the naive algorithm that does not use any optimizations at all and for this algorithm applied on the problem made node consistent. We left these measurements out, because the number of calls for this algorithm is simply too large ($9^{81}$ in the worst case). Moreover, before we started each run, we made the sudoku arc consistent.[5]

[5] A human sudoku solver would actually start in the same way.

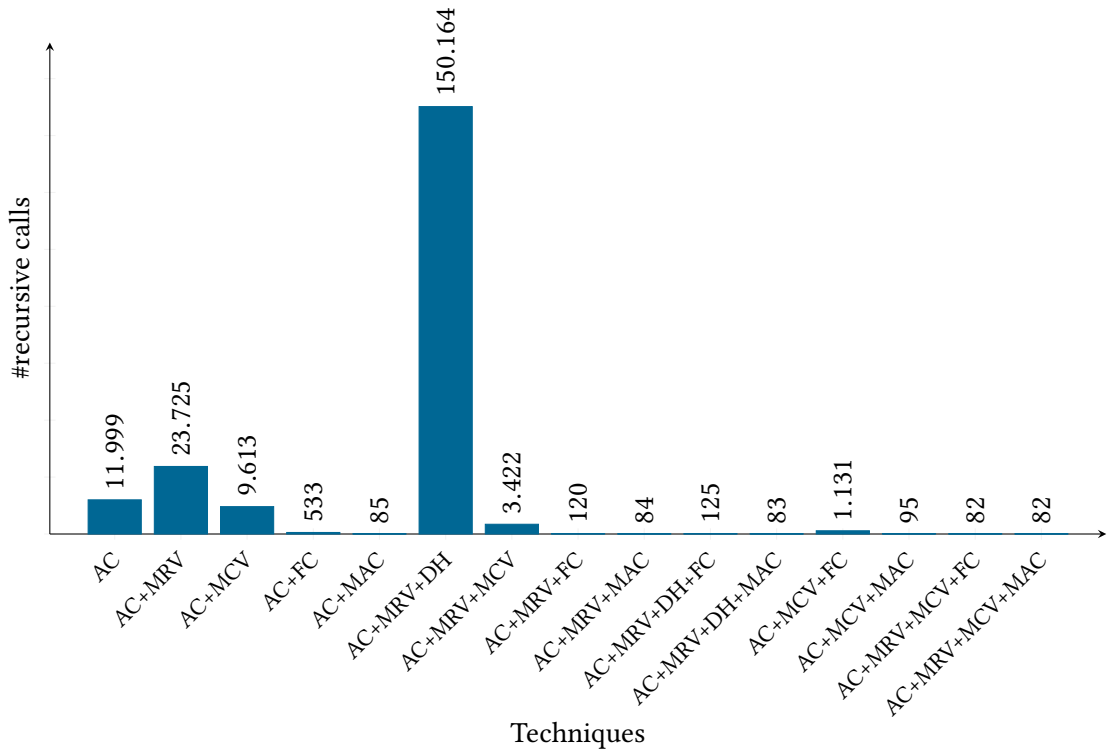


Fig. 21: The number of recursive calls for the Sudoku puzzle

From the chart, we can see that if the Minimum Remaining Values (MRV) and the degree heuristic are used without the application of Forward Checking (FC) or Maintaining Arc Consistency (MAC), then this leads to worse results. However, when MRV and the degree heuristic are applied in combination with FC or MAC, then this leads to a better result. When combined with FC this even leads to a result that is better than both techniques applied separately. The best result is obtained when the problem is kept arc consistent during the solution process. We conclude that for Sudokus the MRV and degree heuristic are only effective if we combine them with other techniques.

We also see that using our own heuristic, Most Connected Variable (MCV), leads to better results in most cases. When MCV is used as a tie-breaker in combination with MRV, the results are always better than when using only MRV or MRV+DH. We conclude that for Sudokus the MCV heuristic is always effective if we use it as a tie-breaker in combination with MRV.

## 8.2 Expression puzzle

In Fig. 22 the results are depicted for the expression puzzle introduced in section 3.3. When looking at the results we see that the first four results are the same. This is due to the fact that all domains are initially the same and none of the variables have a unary constraint that limits its domain by node-consistency. Besides that, there are no binary constraints that limit the values of the variables by arc consistency. At first sight, one might think that arc consistency can reduce the domains of $B$ and $C$ using the constraint $(13 \cdot B)$ **mod** $C = 0$, which is equivalent with $B$ **mod** $C = 0$ since the

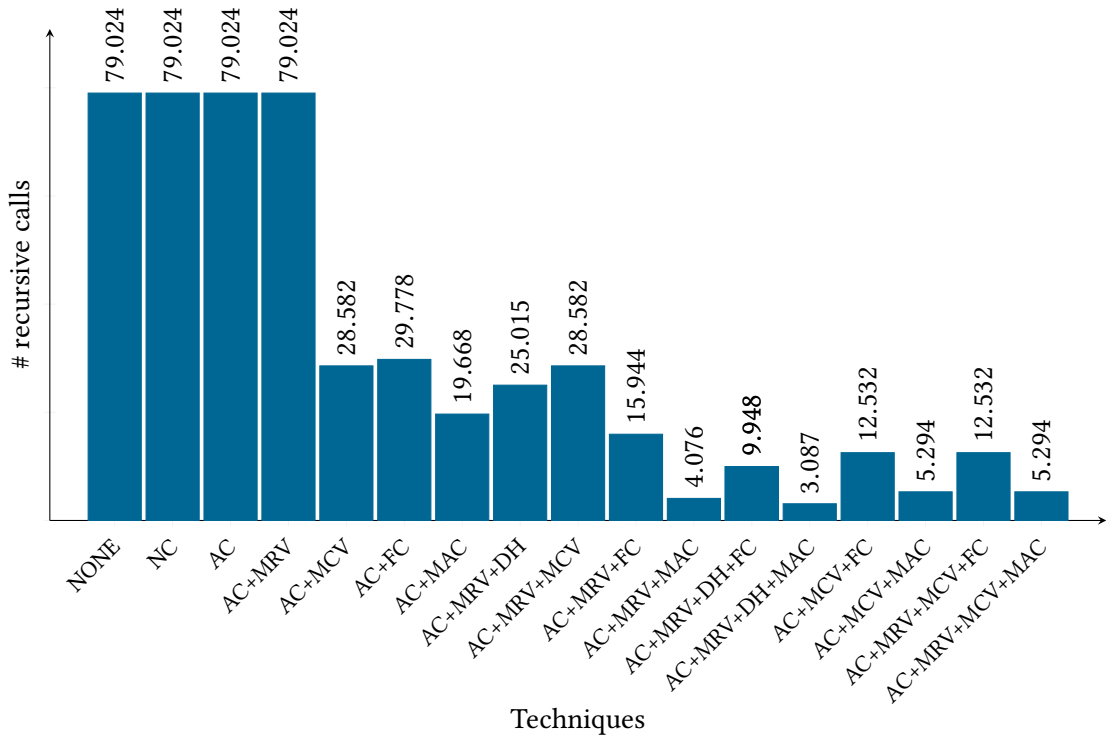


Fig. 22: The number of recursive calls for the expression puzzle

domains contain only digits. This is not the case, however, since we can satisfy the constraint by choosing $B = C$ for all values. In reality, $B = C$ is not allowed, since $B \neq C$ is one of the constraints of the puzzle. Arc consistency does not combine constraints, and therefore does not detect this.

If we do not maintain arc consistency, or apply forward checking then the domains of the unassigned variables remain the same. Hence, the MRV heuristic does not reduce the number of recursive calls. In the same way, we can explain why the combinations AC+MCV and AC+MCV+MRV do not differ.

When we look at AC+MRV+DH, we see that the result for this combination is much better. We can conclude that the degree heuristic helps to find a solution earlier in the backtracking process. This can be explained by the fact that there are two constraints for this problem (see Appendix A.2) that ensure that the subexpressions $13 * B$ **div** $C$ and $(G \cdot H)$ **div** $I$ lead to an integer number. The variables $B, C, G, H$, and $I$ thus have a higher degree than the variables $A, D, E$, and $F$. Before we can check the main constraint (the expression itself), all variables must have been assigned value (i.e. a complete assignment). But, we can reduce the search space a lot by first checking the two constraints mentioned earlier that have a lower arity. These two constraints can be checked earlier if the variables that are involved (the variables with the highest degree) are assigned first. The degree heuristic automatically takes care of this, and so we obtain a better result if we apply it.

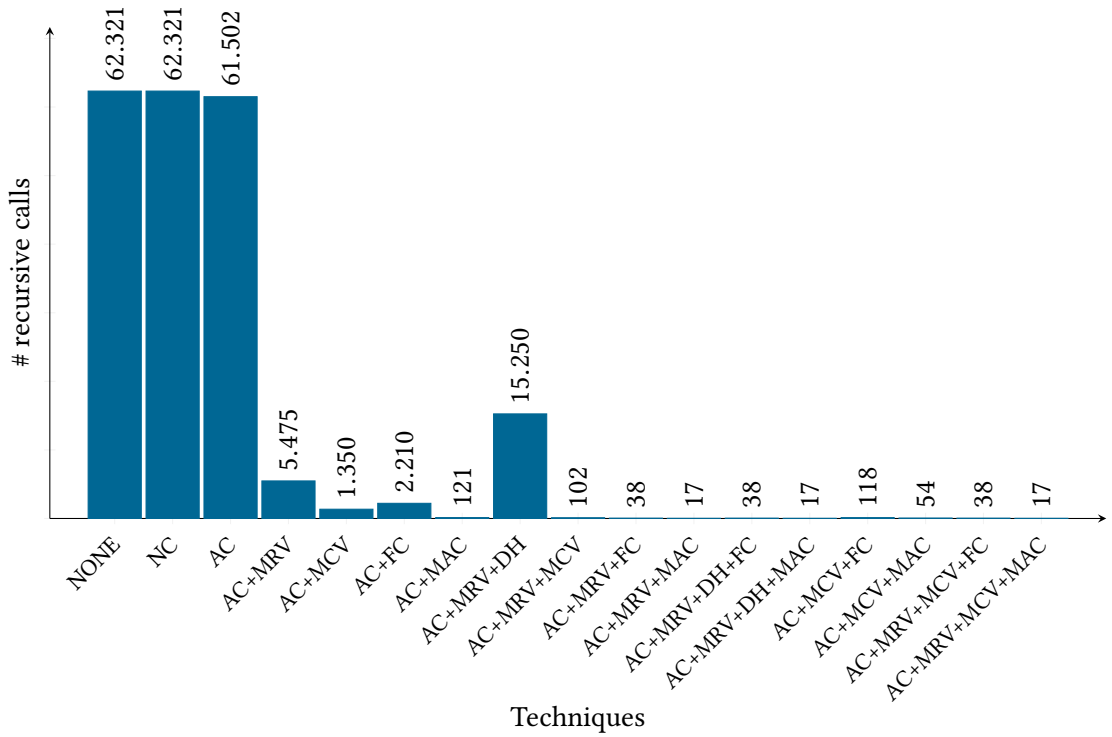


Fig. 23: The number of recursive calls for the cryptarithmetic problem SEND+MORE=MONEY

## 8.3 Cryptarithmetic puzzle

For the cryptarithmetic puzzle of Fig. 23, we see that applying arc-consistency reduces the number of recursive steps a little. This is the result of the unary constraints on $S, M$, and the last carry variable which designates that they cannot be zero. Also, a binary constraint models that $M = X_4$, which automatically reduces the domain of the carry variable $X_4$ to a singleton domain: $\{1\}$ (see Appendix A.3).

It is clear that MRV, FC and MAC reduce the search space a lot for this problem. After each assignment of a variable, each constraint with arity $k$ in which the assigned variable is involved becomes a constraint with the arity $k-1$. When a constraint becomes binary or unary we check satisfiability by applying FC or MAC. Combining all techniques and heuristics results yields a reduction of the number of recursive calls by a factor of $\frac{62321}{17} \approx 3700$.

## 8.4 8-Queens problem

In Fig. 24 we see that the number of recursive calls is the same for the naive backtracking algorithm (NONE), AC, QAC+MRV, and NC. We also see that the degree heuristic (DH) does not make any difference. This phenomenon can be explained by the fact that the 8-Queens problem is completely symmetric: all variables have initially the same domain and have the same number and types of constraints in which they are involved. Also, AC can not reduce any domain without an initial assignment of a variable that would break the symmetry.

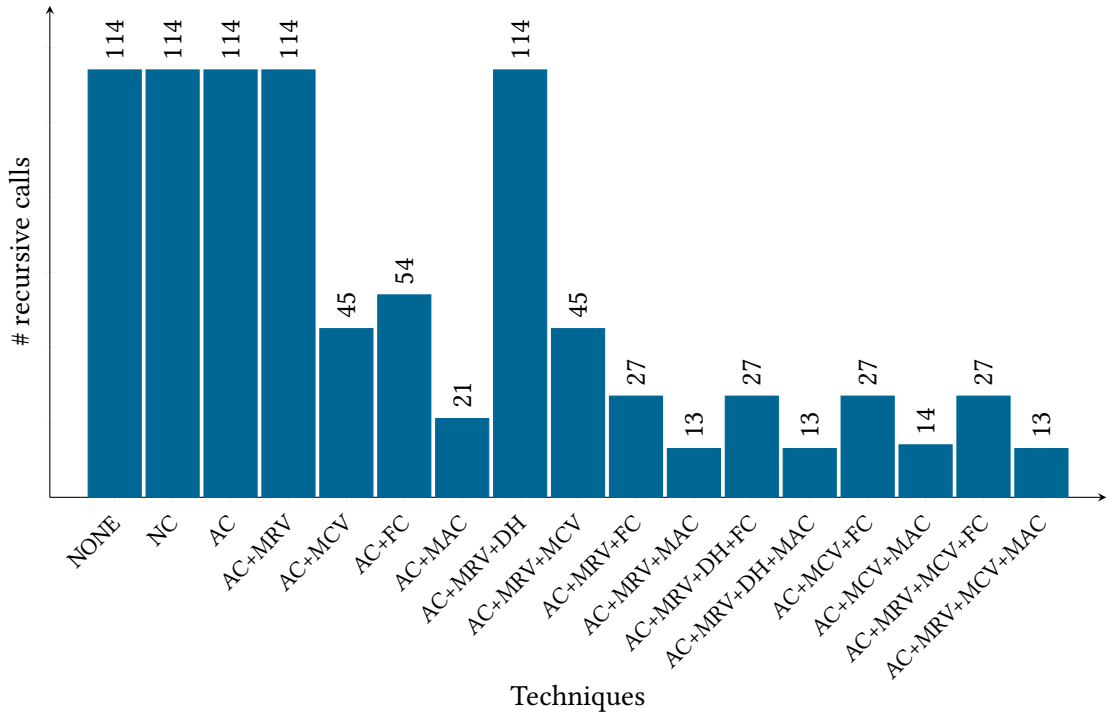


Fig. 24: The number of recursive calls for the 8-Queens problem

Moreover, any order in which variables are chosen is in accordance with the rules of DH. The reason is that after having assigned a value to a variable, the degree of all other variables is lowered with the same amount. Moreover, all variables have the same number of 'connections' with the partial assignment.

The conclusion is clear; any combination of techniques that does not reduce domain sizes, is for the $n$-queens problem non-effective. Moreover, including DH in such a combination is pointless.

## 8.5 Boolean satisfiability problem

In Fig. 25 we see the result for a variation of the boolean satisfiability problem called 3CNF. In this version of the problem, a logical formula is written as a conjunction of clauses, where each clause is a disjunction of 3 literals. Since initially there are no unary constraints, NC does not have any effect. The same holds for AC, since initially there are no binary constraints. MRV does not work initially, since all domains have the same size (i.e. 2).

The MCV heuristic is effective when no domain reductions take place and is always effective when used as a tie-breaker in combination with MRV. The best result is obtained when using MRV+DH though. It is probably a good idea to assign a variable that is involved in many clauses first, because a lot of domains will be reduced afterwards by MAC or FC. Moreover, failure is likely to be detected early.

In Fig. 26 we see the probability that a 3CNF problem with $n = 50$ variables and $m$ clauses is satisfiable as a function of the clause/symbol $(m/n)$ ratio. This image is taken from [1]. We empirically

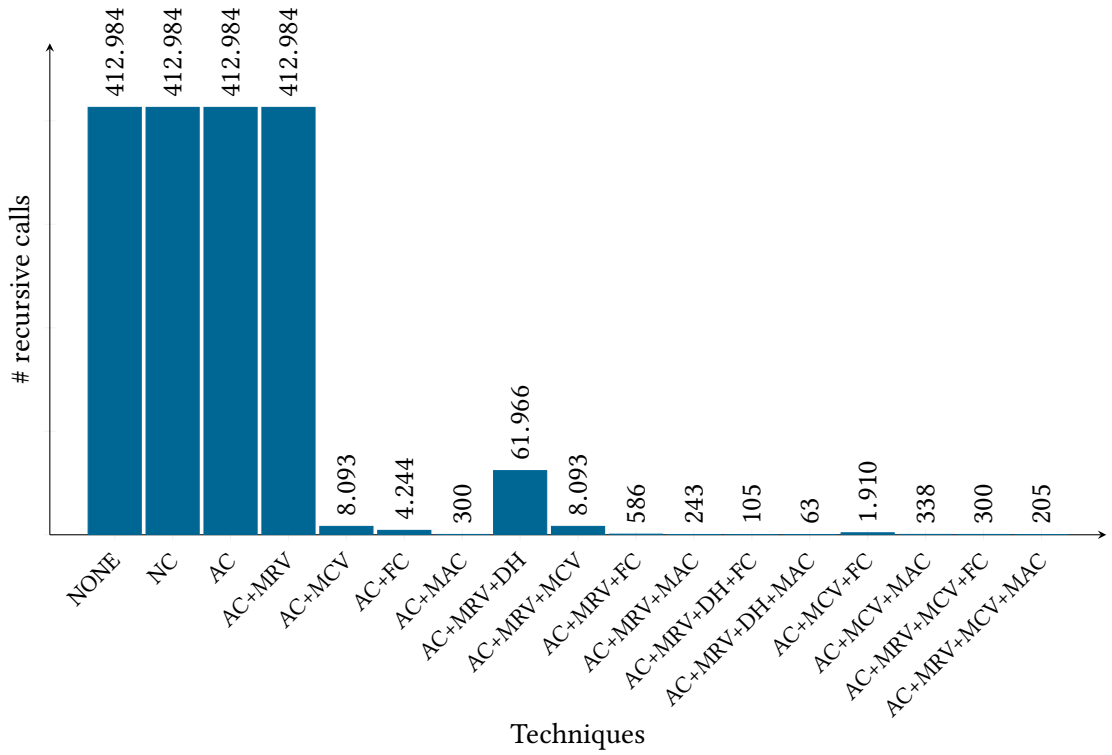


Fig. 25: The number of recursive calls for 3CNF with 40 variables, 200 clauses.

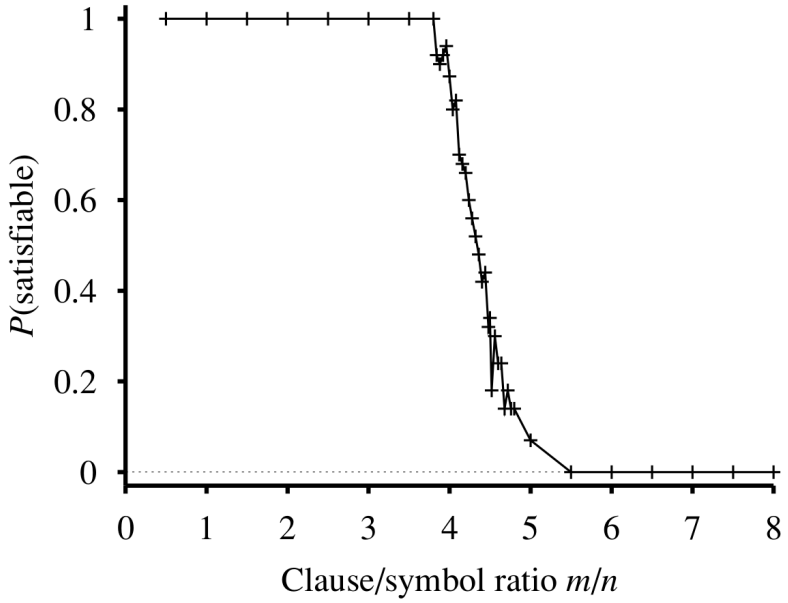


Fig. 26: The probability of success for the 3-CNF problem. Image from [1]

confirmed this graph by doing some experiments ourselves in which 3CNF CSPs were randomly generated given the values of $n$ and $m$. We generated for each ratio 100 random instances of the problem and obtained the same results. The conclusion is clear; problems with a ratio $n/m \leq 4$ are almost always satisfiable, while problems with a ration $m/n \geq 5$ are almost always not satisfiable. The problems in the range $4 < m/n < 5$ are unclear; the 50% satisfiable point is at $m/n = 4.3$.

We expect that this has an effect on the number of recursive steps of the solver for problems with differing $m/n$ ratio. We tested this for the best performing combination of techniques from Fig. 25, i.e. the case AC+MRV+DH+MAC. We tested three problems with $n = 50$ variables and chose the corresponding values of $m$ such that the ratios were respectively 3, 4.2, and 6. The results are as expected, and are given in the following table.

| $m/n$ | # recursive steps |
|---|---|
| 3 | 69 |
| 4.2 | 325 |
| 6 | 47 |

## 9 Conclusions and future work

We started this document with the following research questions:

1. Is it feasible to write a generic solver for different types of CSPs?
2. Can we optimize the solving process in such a way that the number of computation steps will be reduced significantly by applying heuristic techniques?

The first question can clearly be answered affirmative; we implemented a generic solver which is able to solve all our running examples. Moreover, we defined a small programming language in which any CSP can be specified. Especially the latter point makes this a strong case; any new CSP problem can be translated into a specification written in this language, and can thus be solved by the program.

The second question has been answered by performing several benchmark tests. Moreover, we invented a new (as far as we know) heuristic, which we dubbed Most Connected Variable (MCV). The conclusions of the benchmark tests are:

- Some heuristics only work well if used in combination with other techniques. An example of such a heuristic is Minimum Remaining Values. This heuristic always chooses the variable with the smallest domain. So, clearly it must be used in combination with some other technique that reduces domain sizes. So, without the use of forward checking or arc-consistency, using MRV is only effective as an initial variable ordering for the recursive process. When forward checking is applied and/or arc-consistency is preserved after an assignment, domain sizes are changed, and thus it makes sense to use MRV for rearranging the variable selection order. For all problems that are considered in the benchmark, applying MRV and FC together always leads to a better result than applying these techniques separately.

- When applied in the wrong context (i.e. wrong problem type), some heuristics can even lead to an increment in the number of recursive computation steps. For example, from the results obtained for the Sudoku puzzle it is clear that combining MRV and DH yields a dramatic increment of the number of computation steps compared to all other combinations. This result is quite surprising, since in the literature the combination MRV and DH is almost always presented as the standard procedure for solving CSPs.

  For many problems, but not all, this is good advice. For the expression problem, the combination MRV and DH is very effective; it always leads to a better result, no matter what kind of other techniques they are combined with.

- Early pruning based on the partial assignment always leads to a reduction of the search tree. Therefore, we conclude that maintaining arc consistency is never wrong.

## 9.1 Future work: Splitting a problem into subproblems

As explained in section 4.2, it is possible to construct the constraint graph of a CSP. In this graph, the variables are represented by the vertices and constraints are represented by the edges. In standard graph notation, an edge connects two nodes. However, we often deal with constraints in which more than two variables are involved. This problem is solved by using the notion of *hyper edges*. These are edges which can connect any number of vertices. Such a generalized graph is called a *hyper graph*. More about hyper graphs can be found in [6].

The graph representation of a CSP gives more insight in the structure of the problem. For example, in the graph of Fig. 27a, it is clear that the node $a$ is a so-called *articulation point* (see [8]). An articulation point is a node that keeps the graph connected. If such a node (and the corresponding edges) is removed from the graph, then the resulting graph is no longer connected. In terms of our CSP solver, this means that once a value $x$ has been assigned to the variable $a$, the problem is actually split into smaller subproblems. In both subproblems, each occurrence of $a$ is replaced by the value $x$. If any of these subproblems does not have a solution, then we can conclude that the original problem does not have a solution containing the assignment $a := x$. This decomposition into two subproblems is depicted in Fig. 27b.

So, in order to solve the problem represented by the entire graph, one could try to detect the articulation point $a$, and iteratively try values for this variable to find a solution of the smallest subproblem. When a solution is found for this subproblem, given some assignment $a := x$, we try to solve the other subproblems under this assignment.

Using this method, it is expected that failures are detected earlier; when there exists no solution for a subproblem, there will be no solution for the original problem. Moreover, the number of recursive calls for the smaller subproblems is expected to be smaller, and the number of value-trials is expected to be reduced. On the other hand, this idea requires that we need to search for articulation points. This on its turn costs some extra computation time. Whether this is worth it, is hard to tell, and could be investigated in future work. Unfortunately, given the time constraints for this project, we were not able to investigate this ourselves.

An example in which this idea is illustrated to be effective is the following CSP:

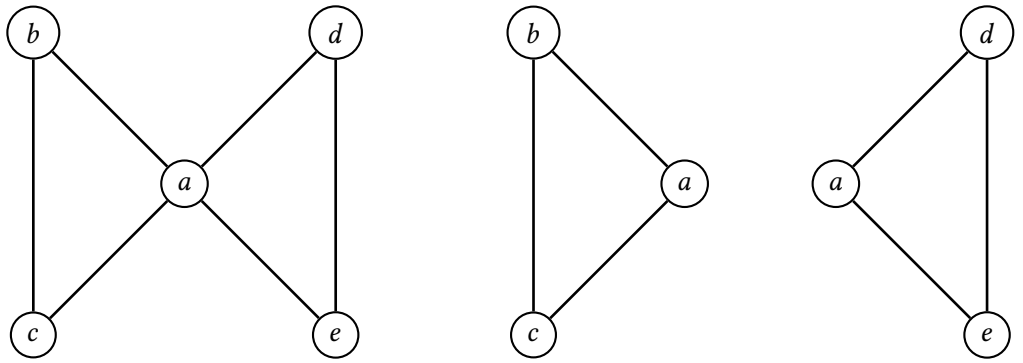


Fig. 27: Graph (a) and its tree decomposition (b)

```
variables:
  a, b, c, d, e : integer;

domains:
  a, b, c, d, e <- [1..5];

constraints:
  b = a - 4;
  c = a - 3;
  d = a - 2;
  e = a - 1;
  b < c;
  d < e;

solutions: 1
```

Clearly, the solution is $a = 5, b = 1, c = 2, d = 3$, and $e = 4$. Now, let us assume that no heuristics are used, except for the proposed technique. The naive backtracking algorithm will make $5^5 = 3125$ recursive steps in the worst case. When we apply tree decomposition and assign the articulation variable $a$ first, we would try $5 \cdot (5^2 + 5^2) = 2 \cdot 5^3 = 250$ values in the worst case. The problem is divided into two smaller subproblems which are not independent, but can be solved independently by assigning the shared variable first.

## References


[1] Russell S., Norvig P. (1995)
*Artificial Intelligence, a modern approach*, 2nd Edition

[2] Abbasian R., Mazloom M. (2009)
*Solving Cryptarithmetic Problems Using Parallel Genetic Algorithm*
Second International Conference on Computer and Electrical Engineering
`https://staff.fnwi.uva.nl/m.mazloom/Papers/mazloom,milad.pdf`

[3] Goodrich M.T., Tamassia R. (2002)
*Algorithm Design: Foundations, Analyisis and Internet Examples*
John Wiley & Sons, Inc.

[4] Tarjan R. (1972)
*Depth-first search and linear graph algorithms*
`http://langevin.univ-tln.fr/cours/PAA/extra/Tarjan-1972.pdf`

[5] Hopcroft J., Tarjan R. (1971)
*Efficient Algorithms for Graph Manipulation*
Cornell University, Ithaca NY
`http://www.akira.ruc.dk/~keld/teaching/algoritmedesign_f03/Artikler/06/Hopcroft73.pdf`

[6] Sapozhenko A.A.
*Hypergraph*
Encyclopedia of Mathematics, Springer
`http://www.encyclopediaofmath.org/index.php/Hypergraph`

[7] Mchugh J.A. (1990)
*Algorithmic Graph Theory*, ISBN 0-13-019092-6
Prentice-Hall International

[8] Weisstein E.W.
*Articulation Vertex*
MathWorld–A Wolfram Web Resource
`http://mathworld.wolfram.com/ArticulationVertex.html`

[9] Barták R. (1998)
*On-line Guide to Constraint Programming*
`http://ktiml.mff.cuni.cz/~bartak/constraints/propagation.html`

[10] Frost D.H. (1997)
*Algorithms and Heuristics for Constraint Satisfaction Problems*
University of California, Irvine
`http://www.ics.uci.edu/~csp/R69.pdf`

# A Source files CSPs

## A.1 Sudoku puzzle

```
# CSP: sudoku
variables:
  X0, X1, X2, X3, X4, X5, X6, X7, X8,
  X9, X10, X11, X12, X13, X14, X15, X16, X17,
  X18, X19, X20, X21, X22, X23, X24, X25, X26,
  X27, X28, X29, X30, X31, X32, X33, X34, X35,
  X36, X37, X38, X39, X40, X41, X42, X43, X44,
  X45, X46, X47, X48, X49, X50, X51, X52, X53,
  X54, X55, X56, X57, X58, X59, X60, X61, X62,
  X63, X64, X65, X66, X67, X68, X69, X70, X71,
  X72, X73, X74, X75, X76, X77, X78, X79, X80 : integer;

domains:
  X0, X1, X2, X3, X4, X5, X6, X7, X8 <- [1..9];
  X9, X10, X11, X12, X13, X14, X15, X16, X17 <- [1..9];
  X18, X19, X20, X21, X22, X23, X24, X25, X26 <- [1..9];
  X27, X28, X29, X30, X31, X32, X33, X34, X35 <- [1..9];
  X36, X37, X38, X39, X40, X41, X42, X43, X44 <- [1..9];
  X45, X46, X47, X48, X49, X50, X51, X52, X53 <- [1..9];
  X54, X55, X56, X57, X58, X59, X60, X61, X62 <- [1..9];
  X63, X64, X65, X66, X67, X68, X69, X70, X71 <- [1..9];
  X72, X73, X74, X75, X76, X77, X78, X79, X80 <- [1..9];

constraints:

  X3 = 5;
  X6 = 9;
  X8 = 2;

  X10 = 6;
  X11 = 4;

  X19 = 3;
  X21 = 8;
  X24 = 5;
  X25 = 6;

  X28 = 8;
  X30 = 1;
  X31 = 4;
  X35 = 9;

  X36 = 1;
  X39 = 3;
  X43 = 8;

  X50 = 5;
  X52 = 4;

  X62 = 3;

  X67 = 9;
  X69 = 6;
  X70 = 1;
```

```
  X72 = 2;
  X77 = 8;
  X79 = 5;



# Alldiff per row
  X0 <> X1;  X0 <> X2;  X0 <> X3;  X0 <> X4;  X0 <> X5;  X0 <> X6;  X0 <> X7;  X0 <>
      X8;
  X1 <> X2;  X1 <> X3;  X1 <> X4;  X1 <> X5;  X1 <> X6;  X1 <> X7;  X1 <> X8;
  X2 <> X3;  X2 <> X4;  X2 <> X5;  X2 <> X6;  X2 <> X7;  X2 <> X8;
  X3 <> X4;  X3 <> X5;  X3 <> X6;  X3 <> X7;  X3 <> X8;
  X4 <> X5;  X4 <> X6;  X4 <> X7;  X4 <> X8;
  X5 <> X6;  X5 <> X7;  X5 <> X8;
  X6 <> X7;  X6 <> X8;
  X7 <> X8;
  X9 <> X10;  X9 <> X11;  X9 <> X12;  X9 <> X13;  X9 <> X14;  X9 <> X15;  X9 <> X16;
       X9 <> X17;
  X10 <> X11;  X10 <> X12;  X10 <> X13;  X10 <> X14;  X10 <> X15;  X10 <> X16;  X10
      <> X17;
  X11 <> X12;  X11 <> X13;  X11 <> X14;  X11 <> X15;  X11 <> X16;  X11 <> X17;
  X12 <> X13;  X12 <> X14;  X12 <> X15;  X12 <> X16;  X12 <> X17;
  X13 <> X14;  X13 <> X15;  X13 <> X16;  X13 <> X17;
  X14 <> X15;  X14 <> X16;  X14 <> X17;
  X15 <> X16;  X15 <> X17;
  X16 <> X17;
  X18 <> X19;  X18 <> X20;  X18 <> X21;  X18 <> X22;  X18 <> X23;  X18 <> X24;  X18
      <> X25;  X18 <> X26;
  X19 <> X20;  X19 <> X21;  X19 <> X22;  X19 <> X23;  X19 <> X24;  X19 <> X25;  X19
      <> X26;
  X20 <> X21;  X20 <> X22;  X20 <> X23;  X20 <> X24;  X20 <> X25;  X20 <> X26;
  X21 <> X22;  X21 <> X23;  X21 <> X24;  X21 <> X25;  X21 <> X26;
  X22 <> X23;  X22 <> X24;  X22 <> X25;  X22 <> X26;
  X23 <> X24;  X23 <> X25;  X23 <> X26;
  X24 <> X25;  X24 <> X26;
  X25 <> X26;
  X27 <> X28;  X27 <> X29;  X27 <> X30;  X27 <> X31;  X27 <> X32;  X27 <> X33;  X27
      <> X34;  X27 <> X35;
  X28 <> X29;  X28 <> X30;  X28 <> X31;  X28 <> X32;  X28 <> X33;  X28 <> X34;  X28
      <> X35;
  X29 <> X30;  X29 <> X31;  X29 <> X32;  X29 <> X33;  X29 <> X34;  X29 <> X35;
  X30 <> X31;  X30 <> X32;  X30 <> X33;  X30 <> X34;  X30 <> X35;
  X31 <> X32;  X31 <> X33;  X31 <> X34;  X31 <> X35;
  X32 <> X33;  X32 <> X34;  X32 <> X35;
  X33 <> X34;  X33 <> X35;
  X34 <> X35;
  X36 <> X37;  X36 <> X38;  X36 <> X39;  X36 <> X40;  X36 <> X41;  X36 <> X42;  X36
      <> X43;  X36 <> X44;
  X37 <> X38;  X37 <> X39;  X37 <> X40;  X37 <> X41;  X37 <> X42;  X37 <> X43;  X37
      <> X44;
  X38 <> X39;  X38 <> X40;  X38 <> X41;  X38 <> X42;  X38 <> X43;  X38 <> X44;
  X39 <> X40;  X39 <> X41;  X39 <> X42;  X39 <> X43;  X39 <> X44;
  X40 <> X41;  X40 <> X42;  X40 <> X43;  X40 <> X44;
  X41 <> X42;  X41 <> X43;  X41 <> X44;
  X42 <> X43;  X42 <> X44;
  X43 <> X44;
  X45 <> X46;  X45 <> X47;  X45 <> X48;  X45 <> X49;  X45 <> X50;  X45 <> X51;  X45
      <> X52;  X45 <> X53;
  X46 <> X47;  X46 <> X48;  X46 <> X49;  X46 <> X50;  X46 <> X51;  X46 <> X52;  X46
      <> X53;
```

```
  X47 <> X48;  X47 <> X49;  X47 <> X50;  X47 <> X51;  X47 <> X52;  X47 <> X53;
  X48 <> X49;  X48 <> X50;  X48 <> X51;  X48 <> X52;  X48 <> X53;
  X49 <> X50;  X49 <> X51;  X49 <> X52;  X49 <> X53;
  X50 <> X51;  X50 <> X52;  X50 <> X53;
  X51 <> X52;  X51 <> X53;
  X52 <> X53;
  X54 <> X55;  X54 <> X56;  X54 <> X57;  X54 <> X58;  X54 <> X59;  X54 <> X60;  X54
      <> X61;  X54 <> X62;
  X55 <> X56;  X55 <> X57;  X55 <> X58;  X55 <> X59;  X55 <> X60;  X55 <> X61;  X55
      <> X62;
  X56 <> X57;  X56 <> X58;  X56 <> X59;  X56 <> X60;  X56 <> X61;  X56 <> X62;
  X57 <> X58;  X57 <> X59;  X57 <> X60;  X57 <> X61;  X57 <> X62;
  X58 <> X59;  X58 <> X60;  X58 <> X61;  X58 <> X62;
  X59 <> X60;  X59 <> X61;  X59 <> X62;
  X60 <> X61;  X60 <> X62;
  X61 <> X62;
  X63 <> X64;  X63 <> X65;  X63 <> X66;  X63 <> X67;  X63 <> X68;  X63 <> X69;  X63
      <> X70;  X63 <> X71;
  X64 <> X65;  X64 <> X66;  X64 <> X67;  X64 <> X68;  X64 <> X69;  X64 <> X70;  X64
      <> X71;
  X65 <> X66;  X65 <> X67;  X65 <> X68;  X65 <> X69;  X65 <> X70;  X65 <> X71;
  X66 <> X67;  X66 <> X68;  X66 <> X69;  X66 <> X70;  X66 <> X71;
  X67 <> X68;  X67 <> X69;  X67 <> X70;  X67 <> X71;
  X68 <> X69;  X68 <> X70;  X68 <> X71;
  X69 <> X70;  X69 <> X71;
  X70 <> X71;
  X72 <> X73;  X72 <> X74;  X72 <> X75;  X72 <> X76;  X72 <> X77;  X72 <> X78;  X72
      <> X79;  X72 <> X80;
  X73 <> X74;  X73 <> X75;  X73 <> X76;  X73 <> X77;  X73 <> X78;  X73 <> X79;  X73
      <> X80;
  X74 <> X75;  X74 <> X76;  X74 <> X77;  X74 <> X78;  X74 <> X79;  X74 <> X80;
  X75 <> X76;  X75 <> X77;  X75 <> X78;  X75 <> X79;  X75 <> X80;
  X76 <> X77;  X76 <> X78;  X76 <> X79;  X76 <> X80;
  X77 <> X78;  X77 <> X79;  X77 <> X80;
  X78 <> X79;  X78 <> X80;
  X79 <> X80;
# Alldiff per column
  X0 <> X9;  X0 <> X18;  X0 <> X27;  X0 <> X36;  X0 <> X45;  X0 <> X54;  X0 <> X63;
      X0 <> X72;
  X1 <> X10;  X1 <> X19;  X1 <> X28;  X1 <> X37;  X1 <> X46;  X1 <> X55;  X1 <> X64;
      X1 <> X73;
  X2 <> X11;  X2 <> X20;  X2 <> X29;  X2 <> X38;  X2 <> X47;  X2 <> X56;  X2 <> X65;
      X2 <> X74;
  X3 <> X12;  X3 <> X21;  X3 <> X30;  X3 <> X39;  X3 <> X48;  X3 <> X57;  X3 <> X66;
      X3 <> X75;
  X4 <> X13;  X4 <> X22;  X4 <> X31;  X4 <> X40;  X4 <> X49;  X4 <> X58;  X4 <> X67;
      X4 <> X76;
  X5 <> X14;  X5 <> X23;  X5 <> X32;  X5 <> X41;  X5 <> X50;  X5 <> X59;  X5 <> X68;
      X5 <> X77;
  X6 <> X15;  X6 <> X24;  X6 <> X33;  X6 <> X42;  X6 <> X51;  X6 <> X60;  X6 <> X69;
      X6 <> X78;
  X7 <> X16;  X7 <> X25;  X7 <> X34;  X7 <> X43;  X7 <> X52;  X7 <> X61;  X7 <> X70;
      X7 <> X79;
  X8 <> X17;  X8 <> X26;  X8 <> X35;  X8 <> X44;  X8 <> X53;  X8 <> X62;  X8 <> X71;
      X8 <> X80;
  X9 <> X18;  X9 <> X27;  X9 <> X36;  X9 <> X45;  X9 <> X54;  X9 <> X63;  X9 <> X72;
  X10 <> X19;  X10 <> X28;  X10 <> X37;  X10 <> X46;  X10 <> X55;  X10 <> X64;  X10
      <> X73;
  X11 <> X20;  X11 <> X29;  X11 <> X38;  X11 <> X47;  X11 <> X56;  X11 <> X65;  X11
      <> X74;
```

```
X12 <> X21;  X12 <> X30;  X12 <> X39;  X12 <> X48;  X12 <> X57;  X12 <> X66;  X12
    <> X75;
X13 <> X22;  X13 <> X31;  X13 <> X40;  X13 <> X49;  X13 <> X58;  X13 <> X67;  X13
    <> X76;
X14 <> X23;  X14 <> X32;  X14 <> X41;  X14 <> X50;  X14 <> X59;  X14 <> X68;  X14
    <> X77;
X15 <> X24;  X15 <> X33;  X15 <> X42;  X15 <> X51;  X15 <> X60;  X15 <> X69;  X15
    <> X78;
X16 <> X25;  X16 <> X34;  X16 <> X43;  X16 <> X52;  X16 <> X61;  X16 <> X70;  X16
    <> X79;
X17 <> X26;  X17 <> X35;  X17 <> X44;  X17 <> X53;  X17 <> X62;  X17 <> X71;  X17
    <> X80;
X18 <> X27;  X18 <> X36;  X18 <> X45;  X18 <> X54;  X18 <> X63;  X18 <> X72;
X19 <> X28;  X19 <> X37;  X19 <> X46;  X19 <> X55;  X19 <> X64;  X19 <> X73;
X20 <> X29;  X20 <> X38;  X20 <> X47;  X20 <> X56;  X20 <> X65;  X20 <> X74;
X21 <> X30;  X21 <> X39;  X21 <> X48;  X21 <> X57;  X21 <> X66;  X21 <> X75;
X22 <> X31;  X22 <> X40;  X22 <> X49;  X22 <> X58;  X22 <> X67;  X22 <> X76;
X23 <> X32;  X23 <> X41;  X23 <> X50;  X23 <> X59;  X23 <> X68;  X23 <> X77;
X24 <> X33;  X24 <> X42;  X24 <> X51;  X24 <> X60;  X24 <> X69;  X24 <> X78;
X25 <> X34;  X25 <> X43;  X25 <> X52;  X25 <> X61;  X25 <> X70;  X25 <> X79;
X26 <> X35;  X26 <> X44;  X26 <> X53;  X26 <> X62;  X26 <> X71;  X26 <> X80;
X27 <> X36;  X27 <> X45;  X27 <> X54;  X27 <> X63;  X27 <> X72;
X28 <> X37;  X28 <> X46;  X28 <> X55;  X28 <> X64;  X28 <> X73;
X29 <> X38;  X29 <> X47;  X29 <> X56;  X29 <> X65;  X29 <> X74;
X30 <> X39;  X30 <> X48;  X30 <> X57;  X30 <> X66;  X30 <> X75;
X31 <> X40;  X31 <> X49;  X31 <> X58;  X31 <> X67;  X31 <> X76;
X32 <> X41;  X32 <> X50;  X32 <> X59;  X32 <> X68;  X32 <> X77;
X33 <> X42;  X33 <> X51;  X33 <> X60;  X33 <> X69;  X33 <> X78;
X34 <> X43;  X34 <> X52;  X34 <> X61;  X34 <> X70;  X34 <> X79;
X35 <> X44;  X35 <> X53;  X35 <> X62;  X35 <> X71;  X35 <> X80;
X36 <> X45;  X36 <> X54;  X36 <> X63;  X36 <> X72;
X37 <> X46;  X37 <> X55;  X37 <> X64;  X37 <> X73;
X38 <> X47;  X38 <> X56;  X38 <> X65;  X38 <> X74;
X39 <> X48;  X39 <> X57;  X39 <> X66;  X39 <> X75;
X40 <> X49;  X40 <> X58;  X40 <> X67;  X40 <> X76;
X41 <> X50;  X41 <> X59;  X41 <> X68;  X41 <> X77;
X42 <> X51;  X42 <> X60;  X42 <> X69;  X42 <> X78;
X43 <> X52;  X43 <> X61;  X43 <> X70;  X43 <> X79;
X44 <> X53;  X44 <> X62;  X44 <> X71;  X44 <> X80;
X45 <> X54;  X45 <> X63;  X45 <> X72;
X46 <> X55;  X46 <> X64;  X46 <> X73;
X47 <> X56;  X47 <> X65;  X47 <> X74;
X48 <> X57;  X48 <> X66;  X48 <> X75;
X49 <> X58;  X49 <> X67;  X49 <> X76;
X50 <> X59;  X50 <> X68;  X50 <> X77;
X51 <> X60;  X51 <> X69;  X51 <> X78;
X52 <> X61;  X52 <> X70;  X52 <> X79;
X53 <> X62;  X53 <> X71;  X53 <> X80;
X54 <> X63;  X54 <> X72;
X55 <> X64;  X55 <> X73;
X56 <> X65;  X56 <> X74;
X57 <> X66;  X57 <> X75;
X58 <> X67;  X58 <> X76;
X59 <> X68;  X59 <> X77;
X60 <> X69;  X60 <> X78;
X61 <> X70;  X61 <> X79;
X62 <> X71;  X62 <> X80;
X63 <> X72;
X64 <> X73;
X65 <> X74;
```

```
  X66 <> X75;
  X67 <> X76;
  X68 <> X77;
  X69 <> X78;
  X70 <> X79;
  X71 <> X80;
# Alldiff per block
  X0 <> X1;  X0 <> X2;  X0 <> X9;  X0 <> X10;  X0 <> X11;  X0 <> X18;  X0 <> X19;  X0
       <> X20;
  X1 <> X2;  X1 <> X9;  X1 <> X10;  X1 <> X11;  X1 <> X18;  X1 <> X19;  X1 <> X20;
  X2 <> X9;  X2 <> X10;  X2 <> X11;  X2 <> X18;  X2 <> X19;  X2 <> X20;
  X3 <> X4;  X3 <> X5;  X3 <> X12;  X3 <> X13;  X3 <> X14;  X3 <> X21;  X3 <> X22;
      X3 <> X23;
  X4 <> X5;  X4 <> X12;  X4 <> X13;  X4 <> X14;  X4 <> X21;  X4 <> X22;  X4 <> X23;
  X5 <> X12;  X5 <> X13;  X5 <> X14;  X5 <> X21;  X5 <> X22;  X5 <> X23;
  X6 <> X7;  X6 <> X8;  X6 <> X15;  X6 <> X16;  X6 <> X17;  X6 <> X24;  X6 <> X25;
      X6 <> X26;
  X7 <> X8;  X7 <> X15;  X7 <> X16;  X7 <> X17;  X7 <> X24;  X7 <> X25;  X7 <> X26;
  X8 <> X15;  X8 <> X16;  X8 <> X17;  X8 <> X24;  X8 <> X25;  X8 <> X26;
  X9 <> X10;  X9 <> X11;  X9 <> X18;  X9 <> X19;  X9 <> X20;
  X10 <> X11;  X10 <> X18;  X10 <> X19;  X10 <> X20;
  X11 <> X18;  X11 <> X19;  X11 <> X20;
  X12 <> X13;  X12 <> X14;  X12 <> X21;  X12 <> X22;  X12 <> X23;
  X13 <> X14;  X13 <> X21;  X13 <> X22;  X13 <> X23;
  X14 <> X21;  X14 <> X22;  X14 <> X23;
  X15 <> X16;  X15 <> X17;  X15 <> X24;  X15 <> X25;  X15 <> X26;
  X16 <> X17;  X16 <> X24;  X16 <> X25;  X16 <> X26;
  X17 <> X24;  X17 <> X25;  X17 <> X26;
  X18 <> X19;  X18 <> X20;
  X19 <> X20;
  X21 <> X22;  X21 <> X23;
  X22 <> X23;
  X24 <> X25;  X24 <> X26;
  X25 <> X26;
  X27 <> X28;  X27 <> X29;  X27 <> X36;  X27 <> X37;  X27 <> X38;  X27 <> X45;  X27
      <> X46;  X27 <> X47;
  X28 <> X29;  X28 <> X36;  X28 <> X37;  X28 <> X38;  X28 <> X45;  X28 <> X46;  X28
      <> X47;
  X29 <> X36;  X29 <> X37;  X29 <> X38;  X29 <> X45;  X29 <> X46;  X29 <> X47;
  X30 <> X31;  X30 <> X32;  X30 <> X39;  X30 <> X40;  X30 <> X41;  X30 <> X48;  X30
      <> X49;  X30 <> X50;
  X31 <> X32;  X31 <> X39;  X31 <> X40;  X31 <> X41;  X31 <> X48;  X31 <> X49;  X31
      <> X50;
  X32 <> X39;  X32 <> X40;  X32 <> X41;  X32 <> X48;  X32 <> X49;  X32 <> X50;
  X33 <> X34;  X33 <> X35;  X33 <> X42;  X33 <> X43;  X33 <> X44;  X33 <> X51;  X33
      <> X52;  X33 <> X53;
  X34 <> X35;  X34 <> X42;  X34 <> X43;  X34 <> X44;  X34 <> X51;  X34 <> X52;  X34
      <> X53;
  X35 <> X42;  X35 <> X43;  X35 <> X44;  X35 <> X51;  X35 <> X52;  X35 <> X53;
  X36 <> X37;  X36 <> X38;  X36 <> X45;  X36 <> X46;  X36 <> X47;
  X37 <> X38;  X37 <> X45;  X37 <> X46;  X37 <> X47;
  X38 <> X45;  X38 <> X46;  X38 <> X47;
  X39 <> X40;  X39 <> X41;  X39 <> X48;  X39 <> X49;  X39 <> X50;
  X40 <> X41;  X40 <> X48;  X40 <> X49;  X40 <> X50;
  X41 <> X48;  X41 <> X49;  X41 <> X50;
  X42 <> X43;  X42 <> X44;  X42 <> X51;  X42 <> X52;  X42 <> X53;
  X43 <> X44;  X43 <> X51;  X43 <> X52;  X43 <> X53;
  X44 <> X51;  X44 <> X52;  X44 <> X53;
  X45 <> X46;  X45 <> X47;
  X46 <> X47;
```

```
  X48 <> X49;  X48 <> X50;
  X49 <> X50;
  X51 <> X52;  X51 <> X53;
  X52 <> X53;
  X54 <> X55;  X54 <> X56;  X54 <> X63;  X54 <> X64;  X54 <> X65;  X54 <> X72;  X54
      <> X73;  X54 <> X74;
  X55 <> X56;  X55 <> X63;  X55 <> X64;  X55 <> X65;  X55 <> X72;  X55 <> X73;  X55
      <> X74;
  X56 <> X63;  X56 <> X64;  X56 <> X65;  X56 <> X72;  X56 <> X73;  X56 <> X74;
  X57 <> X58;  X57 <> X59;  X57 <> X66;  X57 <> X67;  X57 <> X68;  X57 <> X75;  X57
      <> X76;  X57 <> X77;
  X58 <> X59;  X58 <> X66;  X58 <> X67;  X58 <> X68;  X58 <> X75;  X58 <> X76;  X58
      <> X77;
  X59 <> X66;  X59 <> X67;  X59 <> X68;  X59 <> X75;  X59 <> X76;  X59 <> X77;
  X60 <> X61;  X60 <> X62;  X60 <> X69;  X60 <> X70;  X60 <> X71;  X60 <> X78;  X60
      <> X79;  X60 <> X80;
  X61 <> X62;  X61 <> X69;  X61 <> X70;  X61 <> X71;  X61 <> X78;  X61 <> X79;  X61
      <> X80;
  X62 <> X69;  X62 <> X70;  X62 <> X71;  X62 <> X78;  X62 <> X79;  X62 <> X80;
  X63 <> X64;  X63 <> X65;  X63 <> X72;  X63 <> X73;  X63 <> X74;
  X64 <> X65;  X64 <> X72;  X64 <> X73;  X64 <> X74;
  X65 <> X72;  X65 <> X73;  X65 <> X74;
  X66 <> X67;  X66 <> X68;  X66 <> X75;  X66 <> X76;  X66 <> X77;
  X67 <> X68;  X67 <> X75;  X67 <> X76;  X67 <> X77;
  X68 <> X75;  X68 <> X76;  X68 <> X77;
  X69 <> X70;  X69 <> X71;  X69 <> X78;  X69 <> X79;  X69 <> X80;
  X70 <> X71;  X70 <> X78;  X70 <> X79;  X70 <> X80;
  X71 <> X78;  X71 <> X79;  X71 <> X80;
  X72 <> X73;  X72 <> X74;
  X73 <> X74;
  X75 <> X76;  X75 <> X77;
  X76 <> X77;
  X78 <> X79;  X78 <> X80;
  X79 <> X80;

  solutions: 1
```

## A.2 Expression puzzle

```
# CSP: sudoku
variables:
  X0, X1, X2, X3, X4, X5, X6, X7, X8 : integer;

domains:
  X0, X1, X2, X3, X4, X5, X6, X7, X8 <- [1..9];

constraints:
  X0 <> X1; X0 <> X2; X0 <> X3; X0 <> X4; X0 <> X5; X0 <> X6; X0 <> X7; X0 <> X8;
  X1 <> X2; X1 <> X3; X1 <> X4; X1 <> X5; X1 <> X6; X1 <> X7; X1 <> X8;
  X2 <> X3; X2 <> X4; X2 <> X5; X2 <> X6; X2 <> X7; X2 <> X8;
  X3 <> X4; X3 <> X5; X3 <> X6; X3 <> X7; X3 <> X8;
  X4 <> X5; X4 <> X6; X4 <> X7; X4 <> X8;
  X5 <> X6; X5 <> X7; X5 <> X8;
  X6 <> X7; X6 <> X8;
  X7 <> X8;

  X0 + 13 * X1 div X2 + X3 + 12 * X4 - X5 - 11 + X6 * X7 div X8 - 10 = 66;
  (13 * X1) mod X2 = 0;
```

```
  (X6 * X7) mod X8 = 0;

  solutions: 1
```

## A.3 Cryptarithmetic puzzle

```
# CSP: cryptarithmetic puzzle SEND + MORE = MONEY

variables:
  # S = X0, E = X1, N = X2, D = X3, M = X4, O = X5, R = X6, Y = X7
  X0, X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11 : integer;



domains:
  X0, X1, X2, X3, X4, X5, X6, X7 <- [0..9];
  X8, X9, X10, X11 <- [0..1];

constraints:
  X3 + X1 = X7 + 10*X8;          # D + E = Y + 10 * CARRY1
  X2 + X6 + X8 = X1 + 10*X9;     # N + R + CARRY1 = E + 10 * CARRY2
  X1 + X5 + X9 = X2 + 10*X10;    # E + O + CARRY2 = N + 10 * CARRY3
  X0 + X4 + X10 = X5 + 10*X11;   # S + M + CARRY3 = O + 10 * CARRY4
  X11 = X4;                      # 0 + CARRY4 = M
  X11 <> 0;
  X0 <> 0;
  X4 <> 0;

  #alldiff for all characters S,E,N,D,M,O,R,Y
  X0 <> X1; X0 <> X2; X0 <> X3; X0 <> X4; X0 <> X5; X0 <> X6; X0 <> X7;
  X1 <> X2; X1 <> X3; X1 <> X4; X1 <> X5; X1 <> X6; X1 <> X7;
  X2 <> X3; X2 <> X4; X2 <> X5; X2 <> X6; X2 <> X7;
  X3 <> X4; X3 <> X5; X3 <> X6; X3 <> X7;
  X4 <> X5; X4 <> X6; X4 <> X7;
  X5 <> X6; X5 <> X7;
  X6 <> X7;

solutions: 1
```

## A.4 8-Queens problem

```
#CSP: 8-queens problem
variables:
  X0, X1, X2, X3, X4, X5, X6, X7 : integer;

domains:
  X0, X1, X2, X3, X4, X5, X6, X7 <- [0..7];

constraints:
  X0 <> X1; X0 <> X2; X0 <> X3; X0 <> X4; X0 <> X5; X0 <> X6; X0 <> X7;
  X1 <> X2; X1 <> X3; X1 <> X4; X1 <> X5; X1 <> X6; X1 <> X7;
  X2 <> X3; X2 <> X4; X2 <> X5; X2 <> X6; X2 <> X7;
  X3 <> X4; X3 <> X5; X3 <> X6; X3 <> X7;
  X4 <> X5; X4 <> X6; X4 <> X7;
  X5 <> X6; X5 <> X7;
```

```
  X6 <> X7;


  abs(X0-X1) <> abs(0-1); abs(X0-X2) <> abs(0-2); abs(X0-X3) <> abs(0-3); abs(X0-X4)
      <> abs(0-4); abs(X0-X5) <> abs(0-5); abs(X0-X6) <> abs(0-6); abs(X0-X7) <> abs
      (0-7);
  abs(X1-X2) <> abs(1-2); abs(X1-X3) <> abs(1-3); abs(X1-X4) <> abs(1-4); abs(X1-X5)
      <> abs(1-5); abs(X1-X6) <> abs(1-6); abs(X1-X7) <> abs(1-7);
  abs(X2-X3) <> abs(2-3); abs(X2-X4) <> abs(2-4); abs(X2-X5) <> abs(2-5); abs(X2-X6)
      <> abs(2-6); abs(X2-X7) <> abs(2-7);
  abs(X3-X4) <> abs(3-4); abs(X3-X5) <> abs(3-5); abs(X3-X6) <> abs(3-6); abs(X3-X7)
      <> abs(3-7);
  abs(X4-X5) <> abs(4-5); abs(X4-X6) <> abs(4-6); abs(X4-X7) <> abs(4-7);
  abs(X5-X6) <> abs(5-6); abs(X5-X7) <> abs(5-7);
  abs(X6-X7) <> abs(6-7);


solutions: 1
```

# B Code files

## B.1 Grammar

### Flex file CSP-DL (flex.fl)

```
%option noyywrap

%{

#include <stdio.h>
#include <stdlib.h>
#include "grammar.h"

int lineNr = 1;
static int lastNewLine = 0;
static int chars = 0;
static int col = 0;

static int getLength (int number){
  int i=1;
  while(number>9){
    i++;
    number/=10;
  }
  return i;
}

static void newLine() {
  lineNr++;
  lastNewLine = chars+1;
  col = 0;
}

static void column(int plus) {
  col += plus;
  chars += plus;
```

```
}

static int acceptToken(int token) {
  column(yyleng);
  return token;
}

static void printError(){
  fprintf(stderr, "\nError in line %d: ", lineNr);
  fseek(yyin, lastNewLine, SEEK_SET);
  int i = 1;
  char line[200];
  fprintf(stderr, "%s", fgets(line, 200, yyin));
  for(i=0; i<col+(15)+getLength(lineNr); i++){
    fprintf(stderr,  " ");
  }
  fprintf(stderr, "^\n");
  fprintf(stderr, "Illegal character (%s) detected at column %d.\n", yytext, col+1);
  exit(-1);
}

%}


WS          [ \t\v\f]+
STRING      '([^']*)'
DEC         [0-9]+
IDENTIFIER  [a-zA-Z_][a-zA-Z0-9_]*


%%

"#"[^\n]*     { column(yyleng); }

\n            { column(yyleng); newLine(); }

"variables"   { return acceptToken(VARSTOK); }
"in"          { return acceptToken(INTOK); }
"end"          { return acceptToken(ENDTOK); }
"integer"     { return acceptToken(INTTYPE); }
"boolean"     { return acceptToken(BOOLTYPE); }
"domains"     { return acceptToken(DOMAINSTOK); }
"forall"      { return acceptToken(FORALLTOK); }
"alldiff"     { return acceptToken(ALLDIFFTOK); }
"constraints" { return acceptToken(CONSTRAINTSTOK); }
"solutions"   { return acceptToken(SOLVETOK); }
".."          { return acceptToken(RANGETOK); }

"["           { return acceptToken(BRACKOPEN); }
"]"           { return acceptToken(BRACKCLOSE); }
"-"           { return acceptToken(MINUSTOK); }
"+"           { return acceptToken(PLUSTOK); }
"*"           { return acceptToken(STARTOK); }
"^"           { return acceptToken(POWTOK); }
","           { return acceptToken(COMMATOK); }
";"           { return acceptToken(SEMITOK); }
":"           { return acceptToken(COLONTOK); }
"="           { return acceptToken(ISTOK); }
"<"           { return acceptToken(SMALLERTOK); }
">"           { return acceptToken(GREATERTOK); }
```

```
"<>"          { return acceptToken(NEQTOK); }
"<="          { return acceptToken(LEQTOK); }
">="          { return acceptToken(GEQTOK); }
"mod"         { return acceptToken(MODTOK); }
"div"         { return acceptToken(DIVTOK); }
"abs"         { return acceptToken(ABSTOK); }
"max"         { return acceptToken(MAXTOK); }
"min"         { return acceptToken(MINTOK); }
"any"         { return acceptToken(ANYTOK); }
"all"         { return acceptToken(ALLTOK); }
"("           { return acceptToken(PARENTOPEN); }
")"           { return acceptToken(PARENTCLOSE); }
"<-"          { return acceptToken(ARROWTOK); }

{IDENTIFIER}  { return acceptToken(VARNAME); }
{DEC}         { return acceptToken(DECTOK); }
{WS}          { column(yyleng); }

.             { printError(); }
%%

/*
int main(int argc, char** argv) {
  if(argc == 2) {
    yyin = fopen(argv[1], "r");
  }
  yylex();
  if(argc == 2) {
    fclose(yyin);
  }
  return 0;
}*/
```

### File specifying grammar CSP-DL (grammar.g)

```
%start parser, problem;
%token  VARSTOK, DOMAINSTOK, CONSTRAINTSTOK, SOLVETOK, ALLTOK,
    INTTYPE, BOOLTYPE, ARROWTOK,
    BRACKOPEN, BRACKCLOSE, MINUSTOK, PLUSTOK, STARTOK, POWTOK, COMMATOK,
    SEMITOK, COLONTOK, ISTOK, SMALLERTOK, GREATERTOK, NEQTOK, LEQTOK, GEQTOK, MODTOK,
    DIVTOK, PARENTOPEN, PARENTCLOSE, VARNAME, DECTOK, RANGETOK, MAXTOK, MINTOK,
        ABSTOK, ANYTOK,
    FORALLTOK, ALLDIFFTOK, INTOK, ENDTOK;
%options "generate-lexer-wrapper";
%lexical yylex;

%top{
}

{
  #include <stdio.h>
  #include <stdlib.h>

  extern char * yytext;
  extern int lineNr;

  void LLmessage(int token) {
    printf("Parse error: line %d, unexpected token %s\n", lineNr, yytext);
    exit(EXIT_FAILURE);
  }
```

```
  int main(int argc, char** argv) {
    parser();
    return 0;
  }

}

problem          : body
                 ;


body             : vars
                   domains
                   constraints
                   solvespec
                 ;


vars             : VARSTOK COLONTOK [vardeflist COLONTOK datatype SEMITOK]*
                 ;


datatype         : INTTYPE
                 | BOOLTYPE
                 ;


domains          : DOMAINSTOK COLONTOK [domainspec]*
                 ;


domain           : [BRACKOPEN | PARENTOPEN]
                   subdomain
                   [COMMATOK subdomain]*
                   [BRACKCLOSE | PARENTCLOSE]
                 ;


subdomain        : numexp [RANGETOK numexp]?
                 ;


integer          : DECTOK
                 ;


constraints      : CONSTRAINTSTOK COLONTOK [constraintspec]*
                 ;


constraintlist   : constraint [COMMATOK constraint]*
                 ;


solvespec        : SOLVETOK COLONTOK [ALLTOK | integer]
                 ;

varname          : VARNAME
                 ;
```

```
indexspec       : BRACKOPEN integer BRACKCLOSE
                ;

indexcalc       : BRACKOPEN numexp BRACKCLOSE
                ;

vardef          : varname indexspec*
                ;

varcall         : varname indexcalc*
                ;

functioncall    : [
                    [MAXTOK | MINTOK] PARENTOPEN numexp COMMATOK numexp PARENTCLOSE
                  |
                    [ALLTOK | ANYTOK] PARENTOPEN constraintlist PARENTCLOSE
                  |
                    ABSTOK PARENTOPEN numexp PARENTCLOSE
                  |
                    ALLDIFFTOK PARENTOPEN numexp [COMMATOK numexp]* PARENTCLOSE
                  ]
                ;

varlist         : varcall [COMMATOK varcall]*
                ;

vardeflist      : vardef [COMMATOK vardef]*
                ;

forallspec      : FORALLTOK PARENTOPEN varname INTOK domain PARENTCLOSE
                ;

domainspec      : forallspec domainspec* ENDTOK | varlist ARROWTOK domain SEMITOK
                ;

constraintspec  : forallspec constraintspec* ENDTOK | constraint SEMITOK
                ;

constraint      : numexp [relop numexp]?
                ;


numexp          : term [termop term]*
                ;

term            : factor [factorop factor]*
                ;

factor          : value | MINUSTOK factor
                ;

value           : [integer | varcall | functioncall | PARENTOPEN numexp PARENTCLOSE]
    [POWTOK factor]?
                ;

relop           : ISTOK | SMALLERTOK | GREATERTOK | NEQTOK | LEQTOK | GEQTOK
                ;

termop          : MINUSTOK | PLUSTOK
```

```
                ;

factorop        : STARTOK | MODTOK | DIVTOK
                ;
```

### Flex file CSP-DLNF (flex.fl)

```
%option noyywrap

%{

#include <stdio.h>
#include <stdlib.h>
#include "grammar.h"

int lineNr = 1;
static int lastNewLine = 0;
static int chars = 0;
static int col = 0;

static void showtoken() {
  printf ("|%s| ", yytext);
}

static int getLength (int number){
  int i=1;
  while(number>9){
    i++;
    number/=10;
  }
  return i;
}

static void newLine() {
  lineNr++;
  lastNewLine = chars+1;
  col = 0;
}

static void column(int plus) {
  col += plus;
  chars += plus;
}

static int acceptToken(int token) {
  column(yyleng);
  return token;
}

static void printError(){
  fprintf(stderr, "\nError in line %d: ", lineNr);
  fseek(yyin, lastNewLine, SEEK_SET);
  int i = 1;
  char line[200];
  fprintf(stderr, "%s", fgets(line, 200, yyin));
  for(i=0; i<col+(15)+getLength(lineNr); i++){
    fprintf(stderr,  " ");
  }
  fprintf(stderr, "^\n");
  fprintf(stderr, "Illegal character (%s) detected at column %d.\n", yytext, col+1);
```

```
  exit(-1);
}

%}


WS      [ \t\v\f]+
STRING    '([^']*)'
DEC     [0-9]+
VARPREFIX   X



%%

"#"[^\n]*       { column(yyleng); }

\n              { column(yyleng); newLine(); }

"variables"     { return acceptToken(VARSTOK); }
"integer"       { return acceptToken(INTTYPE); }
"boolean"       { return acceptToken(BOOLTYPE); }
"domains"       { return acceptToken(DOMAINSTOK); }
"constraints"   { return acceptToken(CONSTRAINTSTOK); }
"solutions"     { return acceptToken(SOLVETOK); }
".."            { return acceptToken(RANGETOK); }

"["             { return acceptToken(BRACKOPEN); }
"]"             { return acceptToken(BRACKCLOSE); }
"-"             { return acceptToken(MINUSTOK); }
"+"             { return acceptToken(PLUSTOK); }
"*"             { return acceptToken(STARTOK); }
"^"             { return acceptToken(POWTOK); }
","             { return acceptToken(COMMATOK); }
";"             { return acceptToken(SEMITOK); }
":"             { return acceptToken(COLONTOK); }
"="             { return acceptToken(ISTOK); }
"<"             { return acceptToken(SMALLERTOK); }
">"             { return acceptToken(GREATERTOK); }
"<>"            { return acceptToken(NEQTOK); }
"<="            { return acceptToken(LEQTOK); }
">="            { return acceptToken(GEQTOK); }
"mod"           { return acceptToken(MODTOK); }
"div"           { return acceptToken(DIVTOK); }
"abs"           { return acceptToken(ABSTOK); }
"max"           { return acceptToken(MAXTOK); }
"min"           { return acceptToken(MINTOK); }
"any"           { return acceptToken(ANYTOK); }
"all"           { return acceptToken(ALLTOK); }
"("             { return acceptToken(PARENTOPEN); }
")"             { return acceptToken(PARENTCLOSE); }
"<-"            { return acceptToken(ARROWTOK); }

{VARPREFIX}     { return acceptToken(VARTOK); }
{DEC}           { return acceptToken(DECTOK); }
{WS}            { column(yyleng); }

.               { printError(); }
%%
```

### File specifying grammar CSP-DLNF (grammar.g)

```
%start parser, problem;
%token  VARSTOK, DOMAINSTOK, CONSTRAINTSTOK, SOLVETOK, ALLTOK,
    INTTYPE, BOOLTYPE, ARROWTOK,
    BRACKOPEN, BRACKCLOSE, MINUSTOK, PLUSTOK, STARTOK, POWTOK, COMMATOK,
    SEMITOK, COLONTOK, ISTOK, SMALLERTOK, GREATERTOK, NEQTOK, LEQTOK, GEQTOK, MODTOK,
    DIVTOK, PARENTOPEN, PARENTCLOSE, VARTOK, DECTOK, RANGETOK, MAXTOK, MINTOK, ABSTOK
        , ANYTOK;
%options "thread-safe generate-lexer-wrapper generate-symbol-table";
%datatype "Problem", "problem.h";
%lexical yylex;

%top{
#include "problem.h"
}

{
  #include <stdio.h>
  #include <stdlib.h>
  #include <assert.h>
  #include "datatypes.h"
  #include "problem.h"
  #include "variable.h"
  #include "constraint.h"
  #include "solve.h"

  extern char * yytext;
  extern int lineNr;

  void LLmessage(struct LLthis *llthis, int token) {
    printf("Parse error: line %d, unexpected token %s\n", lineNr, yytext);
    exit(EXIT_FAILURE);
  }

}

problem<Problem>                : body(&LLretval)
                                ;


body(Problem *p)                : vars<v>           { *p = emptyProblem();
                                                      setVarsOfProblem(*p, v);
                                                    }
                                  domains(*p)
                                  constraints<c>    { setConstraintsOfProblem(*p, c);
                                       }
                                  solvespec<s>      { (*p)->solvespec = s; }
                                ;


vars<VarList>                                       { LLretval = NULL; }
                                : VARSTOK COLONTOK
                                  [
                                    varlist<l>
                                    COLONTOK
                                    datatype<t>     { LLretval = addVars(LLretval, t,
                                         l); }
                                    SEMITOK
                                  ]*
                                ;
```

```
datatype<DataType>              : INTTYPE           { LLretval = INTEGER; }
                                | BOOLTYPE          { LLretval = BOOLEAN; }
                                ;


domains(Problem p)              : DOMAINSTOK
                                  COLONTOK
                                  [ domainspec(p)
                                    SEMITOK
                                  ]*
                                ;


domain<IntegerSet>                                  { int min = 0;}
                                : [
                                    BRACKOPEN
                                  |
                                    PARENTOPEN      { min = 1;}
                                  ]
                                  subdomain<sd>     { LLretval = emptyIntegerSet();
                                                      sd.min += min;
                                                      addIntervalToSet(LLretval, sd.
                                                          min, sd.max);
                                                    }
                                  [
                                    COMMATOK
                                    subdomain<sd>   { addIntervalToSet(LLretval, sd.
                                        min, sd.max); }
                                  ]*
                                  [
                                    BRACKCLOSE
                                  |
                                    PARENTCLOSE     { removeIntegerFromSet(LLretval,
                                        sd.max); }
                                  ]
                                ;


subdomain<Tuple>                                    { int max; }
                                : integer<min>      { max = min; }
                                  [ RANGETOK
                                    integer<max>
                                  ]?
                                                    { LLretval = (Tuple){min, max}; }
                                ;


integer<int>                    : DECTOK            { LLretval = atoi(yytext); }
                                ;


constraints<ConstraintList>     : CONSTRAINTSTOK
                                  COLONTOK          { LLretval = NULL; }
                                  [
                                    constraint<c>   { LLretval = addConstraint(
                                        LLretval, c); }
                                    SEMITOK
                                  ]*
```

```
                                ;


constraintlist<ConstraintList>  : constraint<c>     { LLretval = addConstraint(NULL,
    c); }
                                  [
                                    COMMATOK
                                    constraint<c>   { LLretval = addConstraint(
                                        LLretval, c); }
                                  ]*
                                ;


solvespec<SolveSpec>            : SOLVETOK COLONTOK
                                  [
                                    ALLTOK          { LLretval.type = SOLVEALL; }
                                  |
                                    integer<max>    { LLretval.type = SOLVENR;
                                                      LLretval.max = max;
                                                    }
                                  ]
                                ;


var<int>                        : VARTOK
                                  integer<LLretval>
                                ;

functioncall<FunctionCall>                          { int argc; FunctionName name;
    void *argv; }
                                : [
                                    [ MAXTOK        { name = MAX; }
                                    | MINTOK        { name = MIN; }
                                    ]
                                    PARENTOPEN
                                    numExp<arg1>
                                    COMMATOK
                                    numExp<arg2>
                                    PARENTCLOSE
                                                    {
                                                      argc = 2;
                                                      NumExp *ar = safeMalloc(argc*
                                                          sizeof(NumExp));
                                                      ar[0] = arg1;
                                                      ar[1] = arg2;
                                                      argv = ar;
                                                    }
                                  |
                                    ABSTOK
                                    PARENTOPEN
                                    numExp<arg1>
                                    PARENTCLOSE
                                                    {
                                                      name = ABS;
                                                      argc = 1;
                                                      NumExp *ar = safeMalloc(argc*
                                                          sizeof(NumExp));
                                                      ar[0] = arg1;
                                                      argv = ar;
                                                    }
```

```
                                  | [ ALLTOK          {name = ALL;}
                                    | ANYTOK        {name = ANY;}
                                    ]
                                    PARENTOPEN
                                    constraintlist<l>
                                    PARENTCLOSE
                                                    {
                                                      argc = 1;
                                                      ConstraintList *p;
                                                      p = safeMalloc(argc*sizeof(
                                                          ConstraintList));
                                                      p[0] = l;
                                                      argv = p;
                                                    }
                                  ]
                                                    { LLretval = newFunctionCall(name
                                                        ,argc,argv); }
                                ;

varlist<VarList>                : var<v>            { LLretval = newVarList(v, NULL);
     }
                                  [ COMMATOK
                                    var<v2>         { addVar(LLretval, v2); }
                                  ]*
                                ;


domainspec(Problem p)            : varlist<vl>
                                  ARROWTOK
                                  domain<d>
                                                    {
                                                      setDomainsOfVars(p, vl, d);
                                                      freeVarList(vl);
                                                      freeIntegerSet(d);
                                                    }
                                ;


constraint<Constraint>          : numExp<e1>        { LLretval = newValConstraint(e1)
    ; }
                                  [
                                    relop<op>
                                    numExp<e2>      { setOperatorOfConstraint(
                                        LLretval, op);
                                                      setSecondExp(LLretval, e2);
                                                    }
                                  ]?
                                ;


numExp<NumExp>                  : term<t1>          { LLretval = newNumExp(t1); }
                                  [
                                    termop<op>
                                    term<t2>        { LLretval = addTerm(LLretval, op
                                        , t2); }
                                  ]*
                                ;

term<Term>                      : factor<f>         { LLretval = newTerm(f); }
                                  [
```

```
                                    factorop<o>
                                    factor<f2>      { LLretval = addFactor(LLretval,
                                        o, f2); }
                                  ]*
                                ;

factor<Factor>                  :   value<v>        { LLretval = newValueFactor(v); }
                                |   MINUSTOK
                                    factor<f>       { LLretval = newMinusFactor(f); }
                                ;

value<Value>                    : [ integer<i>      { LLretval = newIntVal(i); }
                                  | var<id>         { LLretval = newVarVal(id); }
                                  | functioncall<fc>{ LLretval = newFuncVal(fc); }
                                  | PARENTOPEN
                                    numExp<exp>
                                    PARENTCLOSE     { LLretval = newNumExpVal(exp); }
                                  ]
                                  [
                                    POWTOK factor<f>  { LLretval->exponent = f; }
                                  ]?
                                ;

relop<RelOperator>              : ISTOK             { LLretval = IS; }
                                | SMALLERTOK        { LLretval = SMALLER; }
                                | GREATERTOK        { LLretval = GREATER; }
                                | NEQTOK            { LLretval = NEQ; }
                                | LEQTOK            { LLretval = LEQ; }
                                | GEQTOK            { LLretval = GEQ; }
                                ;

termop<TermOperator>            : MINUSTOK          { LLretval = MINUS; }
                                | PLUSTOK           { LLretval = PLUS; }
                                ;

factorop<FactorOperator>        : STARTOK           { LLretval = MUL; }
                                | MODTOK            { LLretval = MOD; }
                                | DIVTOK            { LLretval = DIV; }
                                ;
```

## B.2 Backing up variables

### Header file for data structure Backup (backup.h)

```
#ifndef BACKUP_H
#define BACKUP_H

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "variable.h"
#include "problem.h"

typedef struct backup *Backup;
typedef struct backupList *BackupList;

typedef struct backupList {
  int index;
```

```
  IntegerSet domain;
  varPos sequencePos;
  BackupList next;
} backupList;

typedef struct backup {
  BackupList first;
  BackupList last;
} backup;

Backup emptyBackup();
void freeBackup(Backup db);
void restoreBackup(Backup db, Problem p);
void addBackup(int varIndex, IntegerSet domain, varPos sequencePos, Backup db);

void printBackup(Backup b);


#endif
```

### Implementation data structure Backup (backup.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "backup.h"
#include "solve.h"

Backup emptyBackup() {
  Backup db = safeMalloc(sizeof(backup));
  db->first = NULL;
  db->last = NULL;
  return db;
}

void freeBackupList(BackupList dbl) {
  if(dbl != NULL) {
    freeBackupList(dbl->next);
    free(dbl);
  }
}

void freeBackup(Backup db) {
  freeBackupList(db->first);
  free(db);
}


void restoreBackup(Backup db, Problem p) {
  BackupList list = db->first;
  while(list != NULL) { /* foreach (variable, domain) tuple -> restore */
    Variable v = p->vars[list->index];
    IntegerSet new = domainOfVar(v);
    setDomainOfVar(v, list->domain);
    if(varInSequence(list->sequencePos.var, p->varSequence)) {
      restoreVarSeq(p->varSequence, list->sequencePos);
    }
    freeIntegerSet(new);

    list = list->next;
```

```
  }
  freeBackup(db);
}


void addBackup(int varIndex, IntegerSet domain, varPos seqPos, Backup backup) {
  BackupList new = safeMalloc(sizeof(backupList));
  new->index = varIndex;
  new->domain = domain;
  new->sequencePos = seqPos;
  new->next = backup->first;
  backup->first = new;
  if(backup->last == NULL) {
    backup->last = new;
  }
}

void printBackup(Backup b) {
  addLog("BEGIN BACKUP\n");
  BackupList list = b->first;
  while(list != NULL) {
    Variable v = list->sequencePos.var;
    int varIndex = indexOfVar(v);
    int prevIdx = -1;
    int nextIdx = -1;
    if(list->sequencePos.prev != NULL) {
      prevIdx = indexOfVar(list->sequencePos.prev->var);
    }
    if(list->sequencePos.next != NULL) {
      nextIdx = indexOfVar(list->sequencePos.next->var);
    }
    addLog("Variable X%d, prev: %d, next: %d\n", varIndex, prevIdx, nextIdx);
    list = list->next;
  }
  addLog("END BACKUP\n");
}
```

## B.3 Constraints

### Header file for data structure Constraint (constraint.h)

```
#ifndef CONSTRAINT_H
#define CONSTRAINT_H

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>

struct Constraint;
typedef struct value *Value;
typedef struct term *Term;
typedef struct numExp *NumExp;
typedef struct factor *Factor;
typedef struct functionCall *FunctionCall;
typedef struct constraintList *ConstraintList;
typedef struct constraint *Constraint;
```

```
#include "variable.h"
#include "problem.h"



typedef enum {
  NUMNEG, VALUE
} FactorType;

typedef enum {
  INTVAL, VARVAL, FUNCVAL, NUMEXP
} ValueType;

typedef enum {
  MAX, MIN, ABS, ANY, ALL
} FunctionName;

typedef enum Rel{
  IS, SMALLER, GREATER, GEQ, LEQ, NEQ
} RelOperator;

typedef enum {
  MINUS, PLUS
} TermOperator;

typedef enum {
  MUL, DIV, MOD
} FactorOperator;


typedef union {
  int intval;
  FunctionCall funcCall;
  int varIndex;
  NumExp numexp;
} ValueData;

typedef struct value {
  ValueType type;
  ValueData data;
  Factor exponent;
} value;

typedef union {
  Factor factor;
  Value value;
} FactorData;

typedef struct factor {
  FactorType type;
  FactorData data;
} factor;


typedef struct functionCall {
  FunctionName name;
  int argc;
  void *argv;
} functionCall;
```

```
typedef struct term {
  void *data;
  Term next;
  FactorOperator factorop;
} term;

typedef struct numExp {
  void *data;
  NumExp next;
  TermOperator termop;
} numExp;


typedef struct constraint {   /* exp1 [op exp2]? */
  int index;
  int arity;
  IntegerSet vars;
  NumExp exp1;

  /* optional: if exp2 != NULL. constraint: exp1 [op exp2]? */
  RelOperator op;
  NumExp exp2;
} constraint;

typedef struct constraintList {
  Constraint constraint;
  ConstraintList next;
} constraintList;

Constraint newValConstraint(NumExp e1);
Constraint newBoolConstraint(NumExp e1, RelOperator o, NumExp e2);

void setOperatorOfConstraint(Constraint c, RelOperator relop);
void setSecondExp(Constraint c, NumExp exp2);
void setIndexOfConstraint(Constraint c, int index);
void addVarToConstraint(Constraint c, Variable v);
void removeVarFromConstraint(Constraint c, Variable v);

NumExp firstExp(Constraint c);
NumExp secondExp(Constraint c);
RelOperator operatorOfConstraint(Constraint c);
int indexOfConstraint(Constraint c);

void freeConstraint(Constraint c);
void printConstraint(Constraint c);

ConstraintList addConstraint(ConstraintList list, Constraint constraint);
void freeConstraintList(ConstraintList cl);
void printConstraintList(ConstraintList cl);
int *varIndicesOfConstraint(Constraint c);

NumExp newNumExp(void *data);
NumExp addTerm(NumExp old, TermOperator termop, Term t);
void freeNumExp(NumExp numexp);
void printNumExp(NumExp numExp);
void printRelOperator(RelOperator op);

Term newTerm(void *data);
Term addFactor(Term old, FactorOperator factorop, Factor f);
void freeTerm(Term t);
```

```
void printTerm(Term t);
void printTermOperator(TermOperator op);
int singletonTerm(Term t, Problem p);

Factor newFactor(FactorType type);
Factor newValueFactor(Value v);
Factor newMinusFactor(Factor next);
void freeFactor(Factor f);
void printFactor(Factor f);
void printFactorOperator(FactorOperator op);
int singletonFactor(Factor f, Problem p);
FactorType typeOfFactor(Factor f);
Factor subFactorOfFactor(Factor f);
Value subValueOfFactor(Factor f);

FunctionCall newFunctionCall(FunctionName name, int argc, void *argv);
FunctionName nameOfFuncCall(FunctionCall fc);
int argCountOfFunctionCall(FunctionCall fc);
void *argumentsOfFuncCall(FunctionCall fc);
void freeFunctionCall(FunctionCall f);

Value newValue(ValueType type);
Value newIntVal(int intval);
Value newVarVal(int varIndex);
Value newFuncVal(FunctionCall fc);
Value newNumExpVal(NumExp numexp);
void freeValue(Value v);
void printValue(Value v);
int singletonValue(Value v, Problem p);


Term newTerm(void *data);
Term addFactor(Term old, FactorOperator factorop, Factor f);
void freeTerm(Term t);
void printTerm(Term t);

NumExp newNumExp(void *data);
NumExp addTerm(NumExp numExp, TermOperator termop, Term t);
void freeNumExp(NumExp numexp);
int singletonNumExp(NumExp n, Problem p);
int matchVarExpression(Variable v, NumExp exp);


/* Functions regarding calculation expressions */
int max(NumExp exp1, NumExp exp2, Problem p);
int min(NumExp exp1, NumExp exp2, Problem p);
int absVal(NumExp exp1, Problem p);
int calcFuncVal(FunctionCall fc, Problem p);
int calcValue(Value v, Problem p);
int calcFactor(Factor f, Problem p);
int calcTerm(Term t, Problem p);
int calcExp(NumExp exp, Problem p);

/* Functions regarding properties of constraints */
int arityOfConstraint(Constraint c);
int determinable(Constraint c, Problem p);

/* Functions regarding truth of constraints */
int satisfiable(Constraint c, Problem p);
int or(ConstraintList list, Problem p);
```

```
int and(ConstraintList list, Problem p);
RelOperator switchOperator(RelOperator relop);
int checkRelation(int left, RelOperator relop, int right);
int checkConstraint(Constraint c, Problem p);



#endif
```

### Implementation data structure Constraint (constraint.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "datatypes.h"
#include "variable.h"
#include "constraint.h"
#include "solve.h"
#include <limits.h>

extern FILE *logFile;

Constraint newValConstraint(NumExp e1) {
  Constraint c = safeMalloc(sizeof(constraint));
  c->arity = 0;
  c->exp1 = e1;
  c->exp2 = NULL;
  c->vars = emptyIntegerSet();
  return c;
}

Constraint newBoolConstraint(NumExp e1, RelOperator relop, NumExp e2) {
  Constraint c = newValConstraint(e1);
  c->op = relop;
  c->exp2 = e2;
  return c;
}

void setOperatorOfConstraint(Constraint c, RelOperator relop) {
  c->op = relop;
}

void setSecondExp(Constraint c, NumExp exp2) {
  c->exp2 = exp2;
}

void setIndexOfConstraint(Constraint c, int index) {
  c->index = index;
}

void addVarToConstraint(Constraint c, Variable v) {
  addIntegerToSet(c->vars, indexOfVar(v));
}

void removeVarFromConstraint(Constraint c, Variable v) {
  removeIntegerFromSet(c->vars, indexOfVar(v));
}

NumExp firstExp(Constraint c) {
  return c->exp1;
```

```
}

NumExp secondExp(Constraint c) {
  return c->exp2;
}

RelOperator operatorOfConstraint(Constraint c) {
  return c->op;
}

int indexOfConstraint(Constraint c) {
  return c->index;
}

void freeConstraint(Constraint c) {
  freeNumExp(c->exp1);
  if(c->exp2 != NULL) {
    freeNumExp(c->exp2);
  }
  freeIntegerSet(c->vars);
  free(c);
}

ConstraintList addConstraint(ConstraintList list, Constraint constraint) {
  ConstraintList temp = list;
  ConstraintList new = safeMalloc(sizeof(constraintList));
  new->constraint = constraint;
  new->next = NULL;
  if(list == NULL) {
    return new;
  }
  while(temp->next != NULL) {
    temp = temp->next;
  }
  if(temp->next == NULL) {
    temp->next = new;
  }
  return list;
}

void freeConstraintList(ConstraintList cl) {
  if(cl != NULL) {
    freeConstraintList(cl->next);
    free(cl);
  }
}

NumExp newNumExp(void *data) {
  NumExp numExp = safeMalloc(sizeof(numExp));
  numExp->data = data;
  numExp->next = NULL;
  return numExp;
}

void freeNumExp(NumExp numexp) {
  if(numexp->next == NULL) {
    freeTerm(numexp->data);
  } else {
    freeNumExp(numexp->data);
    freeNumExp(numexp->next);
```

```
  }
  free(numexp);
}

int matchVarExpression(Variable v, NumExp exp) {
  Term t;
  Factor f;
  if(exp->next != NULL) {
    return 0;
  }
  t = exp->data;
  if(t->next != NULL) {
    return 0;
  }
  f = t->data;
  if(f->type == NUMNEG) {
    return 0;
  }
  Value val = f->data.value;
  return (val->type == VARVAL && val->data.varIndex == v->index);
}

NumExp addTerm(NumExp old, TermOperator termop, Term t) {
  NumExp new = newNumExp(old);
  new->termop = termop;
  new->next = newNumExp(t);
  return new;
}

int singletonNumExp(NumExp n, Problem p) {
  if(n == NULL) {
    return 0;
  }
  if(n->next == NULL) {
    return singletonTerm(n->data, p);
  }
  return singletonNumExp(n->data, p)*singletonNumExp(n->next, p);
}

Term newTerm(void *data) {
  Term t = safeMalloc(sizeof(term));
  t->next = NULL;
  t->data = data;
  return t;
}

void freeTerm(Term t) {
  if(t->next == NULL) {
    freeFactor(t->data);
  } else {
    freeTerm(t->next);
    freeTerm(t->data);
  }
  free(t);
}

Term addFactor(Term old, FactorOperator factorop, Factor f) {
  Term new = newTerm(old);
  new->factorop = factorop;
  new->next = newTerm(f);
```

```
  return new;
}

int singletonTerm(Term t, Problem p) {
  if(t->next == NULL) {
    return singletonFactor(t->data, p);
  }
  return singletonTerm(t->data, p)*singletonTerm(t->next, p);
}

Factor newFactor(FactorType type) {
  Factor f = safeMalloc(sizeof(factor));
  f->type = type;
  return f;
}

void freeFactor(Factor f) {
  if(f->type == NUMNEG) {
    freeFactor(f->data.factor);
  } else { /* VALUE */
    freeValue(f->data.value);
  }
  free(f);
}

FactorType typeOfFactor(Factor f) {
  return f->type;
}

Factor subFactorOfFactor(Factor f) {
  return f->data.factor;
}

Value subValueOfFactor(Factor f) {
  return f->data.value;
}

FunctionCall newFunctionCall(FunctionName name, int argc, void *argv) {
  FunctionCall f = safeMalloc(sizeof(functionCall));
  f->name = name;
  f->argc = argc;
  f->argv = argv;
  return f;
}

void freeFunctionCall(FunctionCall f) {
  int i;
  ConstraintList *lists;
  NumExp *exps;
  if(f->name == ANY || f->name == ALL) {
    lists = (ConstraintList *) f->argv;
    for(i = 0; i < f->argc; i++) {
      freeConstraintList(lists[0]);
    }
  }
  else {
    exps = (NumExp *) f->argv;
    for(i = 0; i < f->argc; i++) {
      freeNumExp(exps[i]);
    }
```

```
  }
  free(f->argv);
  free(f);
}

FunctionName nameOfFunctionCall(FunctionCall fc) {
  return fc->name;
}

void *argsOfFunctionCall(FunctionCall fc) {
  return fc->argv;
}

int argCountOfFunctionCall(FunctionCall fc) {
  return fc->argc;
}

Factor newValueFactor(Value v) {
  Factor f = newFactor(VALUE);
  f->data.value = v;
  return f;
}

Factor newMinusFactor(Factor next) {
  Factor f = newFactor(NUMNEG);
  f->data.factor = next;
  return f;
}

int singletonFactor(Factor f, Problem p) {
  if(f->type == NUMNEG) {
    return singletonFactor(f->data.factor, p);
  }
  return singletonValue(f->data.value, p);
}


Value newValue(ValueType type) {
  Value v = safeMalloc(sizeof(value));
  v->type = type;
  v->exponent = NULL;
  return v;
}

void freeValue(Value v) {
  if(v->type == NUMEXP) {
    freeNumExp(v->data.numexp);
  } else if(v->type == FUNCVAL) {
    freeFunctionCall(v->data.funcCall);
  }
  if(v->exponent != NULL) {
    freeFactor(v->exponent);
  }
  free(v);
}

int singletonValue(Value v, Problem p) {
  if(v->type == VARVAL) {
    if(!singletonDomain(domainOfVar(varByIndex(p, v->data.varIndex)))) {
      return 0;
```

```
    }
  } else if(v->type == NUMEXP) {
    if(!singletonNumExp(v->data.numexp, p)) {
      return 0;
    }
  } else if(v->type == FUNCVAL) {
    int i;
    if(v->data.funcCall->name == ANY || v->data.funcCall->name == ALL) {
      ConstraintList *lists = v->data.funcCall->argv;
      ConstraintList list = lists[0];
      while(list != NULL) {
        if(!singletonNumExp(firstExp(list->constraint), p)) {
          return 0;
        }
        NumExp exp2  = secondExp(list->constraint);
        if(exp2 != NULL && !singletonNumExp(exp2, p)) {
          return 0;
        }
        list = list->next;
      }
    } else {
      NumExp *numExps = v->data.funcCall->argv;
      for(i = 0; i < v->data.funcCall->argc; i++) {
        if(!singletonNumExp(numExps[i], p)) {
          return 0;
        }
      }
    }
  }
  return (v->exponent == NULL || singletonFactor(v->exponent, p));
}

Value newIntVal(int intval) {
  Value v = newValue(INTVAL);
  v->data.intval = intval;
  return v;
}

Value newVarVal(int varIndex) {
  Value v = newValue(VARVAL);
  v->data.varIndex = varIndex;
  return v;
}

Value newFuncVal(FunctionCall fc) {
  Value v = newValue(FUNCVAL);
  v->data.funcCall = fc;
  return v;
}

Value newNumExpVal(NumExp numexp) {
  Value v = newValue(NUMEXP);
  v->data.numexp = numexp;
  return v;
}

void printRelOperator(RelOperator op) {
  char *relops[] = {"=", "<", ">", ">=", "<=", "<>"};
  addLog(" %s ", relops[op]);
}
```

```
void printTermOperator(TermOperator op) {
  char *termops[] = {"-", "+"};
  addLog(" %s ", termops[op]);
}

void printFactorOperator(FactorOperator op) {
  char *factorops[] = {"*", "/", "%"};
  addLog(" %s ", factorops[op]);
}

void printValue(Value v) {
  char *functions[] = {"MAX", "MIN", "ABS", "ANY", "ALL"};
  int i;
  switch(v->type) {
    case NUMEXP:
      addLog("(");
      printNumExp(v->data.numexp);
      addLog(")");
      break;
    case INTVAL:
      addLog("%d", v->data.intval);
      break;
    case FUNCVAL:
      addLog("%s(", functions[v->data.funcCall->name]);
      if(v->data.funcCall->name == ANY || v->data.funcCall->name == ALL) {
        ConstraintList *lists = v->data.funcCall->argv;
        ConstraintList list = lists[0];
        while(list != NULL) {
          printNumExp(firstExp(list->constraint));
          if(secondExp(list->constraint) != NULL) {
            printRelOperator(operatorOfConstraint(list->constraint));
            printNumExp(secondExp(list->constraint));
          }
          list = list->next;
        }
      } else {
        NumExp *numExps = v->data.funcCall->argv;
        printNumExp(numExps[0]);
        for(i = 1; i < v->data.funcCall->argc; i++) {
          addLog(", ");
          printNumExp(numExps[i]);
        }
      }
      addLog(")");
      break;
    default: /* VARVAL */
      addLog("X%d", v->data.varIndex);
      break;
  }
  if(v->exponent != NULL) {
    addLog("^");
    printFactor(v->exponent);
  }
}

void printFactor(Factor f) {
  if(f->type == NUMNEG) {
    addLog("-(");
    printFactor(f->data.factor);
```

```
    addLog(")");
  } else {  /* VALUE */
    printValue(f->data.value);
  }
}

void printTerm(Term t) {
  if(t->next == NULL) {
    printFactor(t->data);
  } else {
    addLog("(");
    printTerm(t->data);
    addLog(")");
    printFactorOperator(t->factorop);
    addLog("(");
    printTerm(t->next);
    addLog(")");
  }
}

void printNumExp(NumExp numExp) {
  if(numExp->next == NULL) {
    printTerm(numExp->data);
  } else {
    addLog("(");
    printNumExp(numExp->data);
    addLog(")");
    printTermOperator(numExp->termop);
    addLog("(");
    printNumExp(numExp->next);
    addLog(")");
  }
}

void printConstraint(Constraint c) {
  printNumExp(c->exp1);
  if(c->exp2 != NULL) {
    printRelOperator(c->op);
    printNumExp(c->exp2);
  }
  addLog("\n");
}

void printConstraintList(ConstraintList cl) {
  while(cl !=  NULL) {
    addLog("\t\t");
    printConstraint(cl->constraint);
    cl = cl->next;
  }
}

int *varIndicesOfConstraint(Constraint c) {
  return valuesOfSet(c->vars);
}

/* returns the maximum of numerical expressions exp1 and exp2 */
int max(NumExp exp1, NumExp exp2, Problem p) {
  int a, b;
  a = calcExp(exp1, p);
  b = calcExp(exp2, p);
```

```
  return (a > b ? a : b);
}

/* returns the minimum of numerical expressions exp1 and exp2 */
int min(NumExp exp1, NumExp exp2, Problem p) {
  int a, b;
  a = calcExp(exp1, p);
  b = calcExp(exp2, p);
  return (a < b ? a : b);
}

/* returns the absolute value of numerical expression exp1 */
int absVal(NumExp exp1, Problem p) {
  int val = calcExp(exp1, p);
  return (val >= 0 ? val : -val);
}

/* calculates the value that is returned by FunctionCall fc */
int calcFuncVal(FunctionCall fc, Problem p) {
  FunctionName funcName = nameOfFunctionCall(fc);
  void *arguments = argsOfFunctionCall(fc);
  int argCount = argCountOfFunctionCall(fc);
  if(funcName == ANY || funcName == ALL) {  /* nr of arguments is 1 */
    ConstraintList *lists = arguments;
    if(funcName == ANY) {
      return or(lists[0], p);
    }
    return and(lists[0], p);
  }
  /* not ANY or ALL -> MAX, MIN or ABS */
  NumExp *numExps = arguments;
  if(argCount == 2) { /* MAX or MIN */
    if(funcName == MIN) {
      return min(numExps[0], numExps[1], p);
    }
    return max(numExps[0], numExps[1], p);
  }
  /* argCount == 1 -> ABS */
  return absVal(numExps[0], p);
}

/* calculates the value of Value v */
int calcValue(Value v, Problem p) {
  int base;
  if(v->type == VARVAL) {
    base = domainMinimumOfVar(varByIndex(p, v->data.varIndex));
  } else if(v->type == NUMEXP) {
    base = calcExp(v->data.numexp, p);
  } else if(v->type == FUNCVAL) {
    base = calcFuncVal(v->data.funcCall, p);
  } else {
    base = v->data.intval;
  }
  if(v->exponent != NULL) {
    return pow(base, calcFactor(v->exponent, p));
  }
  return base;
}

/* calculates the value of Factor f */
```

```
int calcFactor(Factor f, Problem p) {
  if(typeOfFactor(f) == NUMNEG) {
    return -calcFactor(subFactorOfFactor(f), p);
  }
  return calcValue(subValueOfFactor(f), p);
}

/* calculates the value of Term t */
int calcTerm(Term t, Problem p) {
  if(t->next == NULL) {
    return calcFactor(t->data, p);
  }
  int val1 = calcTerm(t->data, p);
  int val2 = calcTerm(t->next, p);
  if(t->factorop == MUL) {
    return (val1 * val2);
  }
  /* div or mod */
  if(val2 == 0) {
    addLog("Division by zero: abort\n");
    exit(-1);
  }
  if(t->factorop == DIV) {
    return (val1 / val2);
  }
  return (val1 % val2);
}

/* calculates the value of NumExp exp */
int calcExp(NumExp exp, Problem p) {
  if(exp->next != NULL) {
    int val1 = calcExp(exp->data, p);
    int val2 = calcExp(exp->next, p);
    if(exp->termop == PLUS) {
      return (val1 + val2);
    }
    return (val1 - val2);
  }
  return calcTerm(exp->data, p);
}

int arityOfConstraint(Constraint c) {
  return sizeOfSet(c->vars);
}

/* checks if all variables are assigned a value */
int determinable(Constraint c, Problem p) {
  return (arityOfConstraint(c) == 0);
}

/* checks if the constraint c is satisfiable */
int satisfiable(Constraint c, Problem p) {
  int *varIndices = varIndicesOfConstraint(c);
  IntegerSet fullDomain;
  Variable unassignedVar = NULL;
  int *values;
  int i, j;
  int satisfied = 0;
  int arity = arityOfConstraint(c);
```

```
  /* search for variable that is not yet assigned */
  for(i = 0; i < arity; i++) {
    unassignedVar = varByIndex(p, varIndices[i]);
    fullDomain = domainOfVar(unassignedVar);
    if(sizeOfSet(fullDomain) != 1) {
      break;
    }
  }

  /* all variables have a singleton domain -> one possibility */
  if(unassignedVar == NULL || i == arity) {
    return checkConstraint(c, p);
  }

  /* a variable that is not assigned yet is found */
  values = valuesOfSet(fullDomain);
  for(j = 0; j < sizeOfSet(fullDomain); j++) {
    setDomainOfVar(unassignedVar, createSingletonDomain(values[j]));
    satisfied = satisfiable(c, p);
    freeIntegerSet(domainOfVar(unassignedVar));
    if(satisfied) {
      break;
    }
  }

  /* reset old domain of the ith variable */
  setDomainOfVar(unassignedVar, fullDomain);
  return satisfied;
}



/* checks if there is at least one constraint true in list */
int or(ConstraintList list, Problem p) {
  while(list != NULL) {
    if(checkConstraint(list->constraint, p)) {
      return 1;
    }
    list = list->next;
  }
  return 0;
}

/* checks if all constraints are true in list */
int and(ConstraintList list, Problem p) {
  while(list != NULL) {
    if(!checkConstraint(list->constraint, p)) {
      return 0;
    }
    list = list->next;
  }
  return 1;
}


/* switches the comparison operator if possible */
RelOperator switchOperator(RelOperator relop) {
  if(relop == GEQ) {
    return LEQ;
  }
```

```
  if(relop == LEQ) {
    return GEQ;
  }
  if(relop == GREATER) {
    return SMALLER;
  }
  if(relop == SMALLER) {
    return GREATER;
  }
  return relop;
}

/*
  checks if the comparison operator relop between left and right is applicable
*/
int checkRelation(int left, RelOperator relop, int right) {
  if(relop == NEQ) {
    return (left != right);
  }
  if(relop == SMALLER) {
    return (left < right);
  }
  if(relop == GREATER) {
    return (left > right);
  }
  if(relop == GEQ) {
    return (left >= right);
  }
  if(relop == LEQ) {
    return (left <= right);
  }
  return (left == right); /* relop == IS */
}

int checkConstraint(Constraint c, Problem p) {
  int val1 = calcExp(firstExp(c), p);
  if(secondExp(c) != NULL) {
    int val2 = calcExp(secondExp(c), p);
    return checkRelation(val1, operatorOfConstraint(c), val2);
  }
  return val1;
}
```

## B.4 Variables

### Header file for data structure Variable (variable.h)

```
#ifndef VARIABLE_H
#define VARIABLE_H

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct domainsBackupList *DomainsBackupList;
typedef struct domainsBackup *DomainsBackup;
typedef struct varList *VarList;
typedef struct varPos *VarPos;
```

```
typedef struct varSeq *VarSeq;
typedef struct variable *Variable;

#include "datatypes.h"
#include "constraint.h"


typedef struct variable {
  int index;
  int constraintDegree;
  int connectivity;
  int varConnections;
  DataType type;
  IntegerSet domain;
  IntegerSet constraints;
  VarPos sequencePos;
} variable;

typedef struct varList {
  int varIndex;
  DataType varType;
  VarList next;
} varList;

typedef struct varPos {
  Variable var;
  VarPos prev, next;
} varPos;

typedef struct varSeq {
  VarPos first;
} varSeq;


Variable newVariable(int varIndex, DataType d);
void setDomainOfVar(Variable v, IntegerSet domain);
void freeVariable(Variable v);
void addConstraintToVar(Variable var, Constraint c);
void removeConstraintFromVar(Variable v, Constraint c);

int indexOfVar(Variable v);
DataType dataTypeOfVar(Variable v);
int constraintAmountOfVar(Variable v);
int *constraintIndicesOfVar(Variable v);
int isAssigned(Variable v, Problem p);
int varInSequence(Variable v, VarSeq sequence);
IntegerSet domainOfVar(Variable v);
VarPos sequencePosition(Variable v);
void lowerDegree(Variable v);
void plusDegree(Variable v);
int degreeOfVar(Variable v);
void lowerConnectivity(Variable v);
void plusConnectivity(Variable v);

IntegerSet createSingletonDomain(int value);
int domainSizeOfVar(Variable v);
void printDomainOfVar(Variable v);
int singletonDomain(IntegerSet d);
int *domainValuesOfVar(Variable v);
int domainMinimumOfVar(Variable v);
```

```
int domainMaximumOfVar(Variable v);

VarList newVarList(int varIndex, VarList next);
void freeVarList(VarList vl);
void addVar(VarList vl, int varIndex);
VarList addVars(VarList vl, DataType d, VarList toAdd);
Variable *varListToArray(VarList vl);

void printVar(Variable v);


VarSeq emptyVarSeq();
void freeVarSeq(VarSeq sequence);
void restoreVarSeq(VarSeq sequence, varPos backup);
void resortVarSeq(VarSeq sequence, VarPos position);

void assertSequenceSorted(VarSeq sequence);

VarPos insertVarInSequence(VarSeq sequence, Variable v);
void removeVarFromSequence(VarSeq sequence, VarPos position);
VarPos firstVarPosition(VarSeq sequence);
Variable varAtPosition(VarSeq sequence, VarPos position);
void printVarSequence(VarSeq sequence);


int mrvPlusDegreeOrdered(Variable first, Variable second);
int mrvOrdered(Variable first, Variable second);
int mostConnectedOrdered(Variable first, Variable second);
int mrvPlusConnectedOrdered(Variable first, Variable second);
int skipTest(Variable first, Variable second);

int nodeReduce(Variable v, Constraint c, Problem p);
int arcReduce(DirectedArc arc, Problem p);

void plusVarConnections(Variable v);

#endif
```

### Implementation data structure Variable (variable.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <limits.h>
#include "datatypes.h"
#include "problem.h"
#include "variable.h"
#include "solve.h"

extern FILE *logFile;
extern int (*varOrdered)(Variable, Variable);

/* BEGIN functions Variable */
Variable newVariable(int varIndex, DataType d) {
  Variable v = safeMalloc(sizeof(variable));
  v->index = varIndex;
  v->constraintDegree = 0;
  v->connectivity = 0;
  v->varConnections = 0;
  v->type = d;
```

```
  v->domain = emptyIntegerSet();
  v->constraints = emptyIntegerSet();
  v->sequencePos = NULL;
  return v;
}

void setDomainOfVar(Variable v, IntegerSet domain) {
  v->domain = domain;
}

void freeVariable(Variable v) {
  if(v != NULL) {
    freeIntegerSet(v->domain);
    freeIntegerSet(v->constraints);
    free(v);
  }
}

void addConstraintToVar(Variable v, Constraint c) {
  addIntegerToSet(v->constraints, indexOfConstraint(c));
}

void removeConstraintFromVar(Variable v, Constraint c) {
  removeIntegerFromSet(v->constraints, indexOfConstraint(c));
}

int indexOfVar(Variable v) {
  return v->index;
}

DataType dataTypeOfVar(Variable v) {
  return v->type;
}

int constraintAmountOfVar(Variable v) {
  return v->constraints->size;
}

int *constraintIndicesOfVar(Variable v) {
  return v->constraints->values;
}

int isAssigned(Variable v, Problem p) {
  return !varInSequence(v, varSeqOfProblem(p));
}

IntegerSet domainOfVar(Variable v) {
  return v->domain;
}

VarPos sequencePosition(Variable v) {
  return v->sequencePos;
}

void setSeqPosOfVar(Variable v, VarPos pos) {
  v->sequencePos = pos;
}

void lowerDegree(Variable v) {
  v->constraintDegree--;
```

```
}

void plusDegree(Variable v) {
  v->constraintDegree++;
}

int degreeOfVar(Variable v) {
  return v->constraintDegree;
}

void lowerConnectivity(Variable v) {
  v->connectivity--;
}

void plusConnectivity(Variable v) {
  v->connectivity++;
}

/* END functions Variable */

/* BEGIN functions Domain */

IntegerSet createSingletonDomain(int value) {
  IntegerSet singleton = emptyIntegerSet();
  addIntegerToSet(singleton, value);
  return singleton;
}

void printDomainOfVar(Variable v) {
  IntegerSet domain = v->domain;
  int i;
  int *values = valuesOfSet(domain);
  addLog("[");
  if(sizeOfSet(domain) > 0) {
    addLog("%d", values[0]);
  }
  for(i = 1; i < sizeOfSet(domain); i++) {
    addLog(", %d", values[i]);
  }
  addLog("]");
}

int singletonDomain(IntegerSet domain) {
  return (sizeOfSet(domain) == 1);
}

int domainSizeOfVar(Variable v) {
  return sizeOfSet(v->domain);
}

int *domainValuesOfVar(Variable v) {
  return valuesOfSet(v->domain);
}

int domainMinimumOfVar(Variable v) {
  return minimumOfSet(v->domain);
}

int domainMaximumOfVar(Variable v) {
  return maximumOfSet(v->domain);
```

```
}

/* END functions Domain */

/*
 * creates a new VarList and sets the first item
*/
VarList newVarList(int varIndex, VarList next) {
  VarList vl = safeMalloc(sizeof(varList));
  vl->varIndex = varIndex;
  vl->next = next;
  return vl;
}

/*
 *
*/
void freeVarList(VarList vl) {
  if(vl != NULL) {
    freeVarList(vl->next);
    free(vl);
  }
}

/*
 * creates a Variable, given  to a VarList
*/
void addVar(VarList vl, int varIndex) {
  VarList end = vl;
  while(end->next != NULL) {
    end = end->next;
  }
  end->next = newVarList(varIndex, NULL);
}

/*
 *
*/
VarList addVars(VarList vl, DataType d, VarList toAdd) {
  VarList tempList = toAdd;
  while(tempList != NULL) {
    tempList->varType = d;
    tempList = tempList->next;
  }
  if(vl == NULL) {
    return toAdd;
  }
  tempList = vl;
  while(tempList->next != NULL) {
    tempList = tempList->next;
  }
  tempList->next = toAdd;
  return tempList;
}


/* END functions VarSet */


/* BEGIN functions Variable */
```

```
/*
 *
*/
void printVar(Variable v) {
  addLog("X%d : %s\n", v->index, (v->type == INTEGER ? "integer" : "boolean"));
  addLog("\tDomain(%d): ", sizeOfSet(v->domain));
  printDomainOfVar(v);
  addLog("\n");
  addLog("\tDegree(%d): \n", v->constraintDegree);
  addLog("\tConnectivity(%d): \n", v->connectivity);
  addLog("\n");
}


/* END functions Variable */


/* BEGIN Functions VarSeq */

/*
 *
*/
VarSeq emptyVarSeq() {
  VarSeq seq = safeMalloc(sizeof(varSeq));
  seq->first = NULL;
  return seq;
}

/*
 *
*/
VarPos newVarPos(Variable v, VarPos prev, VarPos next) {
  VarPos new = safeMalloc(sizeof(varPos));
  new->var = v;
  new->prev = prev;
  new->next = next;
  if(prev != NULL) {
    prev->next = new;
  }
  if(next != NULL) {
    next->prev = new;
  }
  return new;
}

/*
 *
*/
void freeVarSeqRec(VarPos current) {
  if(current != NULL) {
    freeVarSeqRec(current->next);
    free(current);
  }
}

/*
 *
*/
void freeVarSeq(VarSeq sequence) {
```

```
  if(sequence != NULL) {
    freeVarSeqRec(sequence->first);
    free(sequence);
  }
}

int countVarSeq(VarSeq sequence) {
  VarPos pos = sequence->first;
  int cnt = 0;
  while(pos != NULL) {
    cnt++;
    pos = pos->next;
  }
  return cnt;
}


/*
 *
*/
void removeVarFromSequence(VarSeq sequence, VarPos position) {
  int cnt = countVarSeq(sequence);

  if(!varInSequence(position->var, sequence)) {
    return;
  }

  /* 'knot' previous prev and next */
  if(position->prev != NULL) {
    position->prev->next = position->next;
  } else {  /* position is first position */
    sequence->first = position->next;
  }
  if(position->next != NULL) {
    position->next->prev = position->prev;
  }
  position->next = NULL;
  position->prev = NULL;
  assert(cnt == countVarSeq(sequence)+1);
}

/*
 *
*/
int varInSequence(Variable v, VarSeq sequence) {
  VarPos position = sequencePosition(v);
  if(position == NULL) {
    return 0;
  }
  if(position->prev != NULL || position->next != NULL) {
    return 1;
  }
  return (sequence->first == position);
}

/*
 * Inserts a variable v in sequence of variables
*/
VarPos insertVarInSequence(VarSeq sequence, Variable v) {
  /* insert at first position */
```

```
  sequence->first = newVarPos(v, NULL, sequence->first);
  /* tell variable it's new position */
  setSeqPosOfVar(v, sequence->first);
  /* resort sequence */
  resortVarSeq(sequence, sequence->first);
  return sequencePosition(v);
}

void insertAfter(VarSeq seq, VarPos pos, VarPos prev) {
  /* set new neighbours */
  pos->prev = prev;

  /* tell new neighbours you are living here */
  if(pos->prev != NULL) {
    pos->next = pos->prev->next;
    pos->prev->next = pos;
  } else {
    pos->next = seq->first;
    seq->first = pos;
  }
  if(pos->next != NULL) {
    pos->next->prev = pos;
  }
}

/* resorts variable at VarPos in the VarSequence of variables */
void resortVarSeq(VarSeq sequence, VarPos position) {
  VarPos next = position->next;
  VarPos prev = position->prev;
  /* shift to right as long as needed */
  while(next != NULL && !varOrdered(position->var, next->var)) {
    prev = next;
    next = next->next;
  }
  /* shift to left as long as needed */
  while(prev != NULL && !varOrdered(prev->var, position->var)) {
    next = prev;
    prev = prev->prev;
  }
  /* remove at current position */
  removeVarFromSequence(sequence, position);
  /* insert at new position */

  insertAfter(sequence, position, prev);
}


void assertSequenceSorted(VarSeq sequence) {
  VarPos first = sequence->first;
  VarPos second = first->next;
  while(second != NULL) {
    assert(varOrdered(first->var, second->var));
    first = second;
    second = second->next;
  }
}


/*
 * Function that restores the sequence to an earlier version
```

```
 * given a backup of a position of a variable in the sequence
*/
void restoreVarSeq(VarSeq sequence, varPos backup) {
  int cnt = countVarSeq(sequence);
  int inSeq = varInSequence(backup.var, sequence);
  /* remove variable from current position */
  removeVarFromSequence(sequence, backup.var->sequencePos);

  /* insert at new position */
  insertAfter(sequence, backup.var->sequencePos, backup.prev);
  resortVarSeq(sequence, backup.var->sequencePos);
  assert(countVarSeq(sequence) == cnt + !inSeq);
}

/*
 * Function that returns the first VarPos, given a VarSequence
*/
VarPos firstVarPosition(VarSeq sequence) {
  return sequence->first;
}

/*
 * Given a sequence and position in sequence
 * -> return var at that position
 * VarSequence as argument because abstraction
 * (implementation could change)
*/
Variable varAtPosition(VarSeq seq, VarPos pos) {
  if(pos == NULL) {
    return NULL;
  }
  return pos->var;
}

/*
 * This function prints, given a VarSequence, for each variable
 * - the domains of the variable
 * - the degree (constraints with unassigned variables) of the variable
*/
void printVarSequence(VarSeq sequence) {
  VarPos pos = sequence->first;
  addLog("BEGIN Sequence of variables: \n");
  while(pos != NULL) {
    printVar(pos->var);
    pos = pos->next;
  }
  addLog("END Sequence of variables: \n");
}

/*
 * Checks if variable first should be selected before variable second
 * when MRV (without degree heuristic as a tie-breaker) is applied
*/
int mrvOrdered(Variable first, Variable second) {
  int domSize1 = domainSizeOfVar(first);
  int domSize2 = domainSizeOfVar(second);
  return (domSize1 <= domSize2);
}

/*
```

```
 * Checks if variable first should be selected before variable second
 * when MRV + degree heuristic is applied
*/
int mrvPlusDegreeOrdered(Variable first, Variable second) {
  int domSizeFirst = domainSizeOfVar(first);
  int domSizeSecond = domainSizeOfVar(second);
  if(domSizeFirst < domSizeSecond) {
    return 1;
  } else if(domSizeFirst == domSizeSecond) {
    return (first->constraintDegree >= second->constraintDegree);
  }
  return 0;
}

int mostConnectedOrdered(Variable first, Variable second) {
  /*return (first->connectivity >= second->connectivity);*/
  int connectivity1 = first->connectivity;
  int connectivity2 = second->connectivity;
  /*double conns1 = first->varConnections
  double conns2 = second->varConnections
  connectivity1 /= conns1;
  connectivity2 /= conns2;*/
  return (connectivity1 >= connectivity2);
}

int mrvPlusConnectedOrdered(Variable first, Variable second) {
  int domSizeFirst = domainSizeOfVar(first);
  int domSizeSecond = domainSizeOfVar(second);
  if(domSizeFirst < domSizeSecond) {
    return 1;
  } else if(domSizeFirst == domSizeSecond) {
    return mostConnectedOrdered(first, second);
  }
  return 0;
}


/*
 * Function that always agrees with order of variables first and second
 * in variable sequence
*/
int skipTest(Variable first, Variable second) {
  return 1;
}

/* END Functions VarSequence */


int nodeReduce(Variable v, Constraint c, Problem p) {
  IntegerSet domain = domainOfVar(v);
  int *values = valuesOfSet(domain);
  int changed = 0;
  int remove, i;
  if(arityOfConstraint(c) != 1) {
    return 0;
  }
  for(i = 0; i < sizeOfSet(domain); i++) {
    setDomainOfVar(v, createSingletonDomain(values[i]));
    remove = !checkConstraint(c, p);
    freeIntegerSet(domainOfVar(v));
```

```
    if(remove) {
      changed = 1;
      removeNthIntegerFromSet(domain, i);
      i--;
    }
  }
  setDomainOfVar(v, domain);
  return changed;
}

int arcReduce(DirectedArc arc, Problem p) {
  Variable v = firstVarOfArc(arc);
  Constraint c = constraintOfArc(arc);
  IntegerSet domain = domainOfVar(v);
  int *values = valuesOfSet(domain);
  int changed = 0;
  int remove, i;
  /* arity of constraint must be 2 (binary, in case of arc consistency check) or
      arity must be 1 (after assignment var and forwardchecking for all arcs directed
          at var) */
  assert(arityOfConstraint(c) <= 2);
  for(i = 0; i < sizeOfSet(domain); i++) {
    setDomainOfVar(v, createSingletonDomain(values[i]));
    remove = !satisfiable(c, p);
    freeIntegerSet(domainOfVar(v));
    if(remove) {
      changed = 1;
      removeNthIntegerFromSet(domain, i);
      i--;
    }
  }
  setDomainOfVar(v, domain);
  return changed;
}

void plusVarConnections(Variable v) {
  v->varConnections++;
}
```

## B.5 Several datatypes

### Header file for several datatypes (datatypes.h)

```
#ifndef DATATYPES_H
#define DATATYPES_H

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct integerList *IntegerList;
typedef struct integerSet *IntegerSet;
typedef struct listItem *List;
typedef struct queue *Queue;
typedef struct stack *Stack;

typedef enum {
  INTEGER, BOOLEAN
```

```
} DataType;

typedef struct integerSet {
  int size;
  int space;
  int *values;
} integerSet;

typedef struct integerList {
  int val;
  IntegerList next, prev;
} integerList;

typedef struct Tuple {
  int min;
  int max;
} Tuple;

typedef struct listItem {
  void *data;
  List next;
} listItem;

typedef struct queue {
  List first;
  List last;
} queue;

typedef struct stack {
  List first;
} stack;

void *safeMalloc(size_t size);
void *safeCalloc(int amount, size_t size);
void *safeRealloc(void *oldPtr, size_t size);

IntegerList newIntegerList(int val, IntegerList next);
IntegerList copyIntegerList(IntegerList orig);
void printIntegerList(IntegerList list);
void freeIntegerList(IntegerList list);

IntegerSet emptyIntegerSet();
IntegerSet copyIntegerSet(IntegerSet orig);
void freeIntegerSet(IntegerSet set);
void addIntegerToSet(IntegerSet set, int value);
void addIntervalToSet(IntegerSet set, int min, int max);
void removeIntegerFromSet(IntegerSet set, int value);
void removeNthIntegerFromSet(IntegerSet set, int n);
IntegerSet intersect(IntegerSet set1, IntegerSet set2);
IntegerSet except(IntegerSet total, IntegerSet toBeRemoved);

int minimumOfSet(IntegerSet d);
int maximumOfSet(IntegerSet d);
int sizeOfSet(IntegerSet set);
int *valuesOfSet(IntegerSet set);
int valueInSet(IntegerSet set, int value);

List newListItem(void *data, List next);
void freeList(List l);
```

```
Queue emptyQueue();
int isEmptyQueue(Queue q);
void freeQueue(Queue q);
void enqueue(Queue q, void *item);
void *dequeue(Queue q);

Stack emptyStack();
int isEmptyStack(Stack s);
void freeStack(Stack s);
void push(Stack s, void *item);
void *pop(Stack s);
void *top(Stack s);

#endif
```

### Implementation several datatypes (datatypes.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <limits.h>
#include "datatypes.h"
#include "problem.h"
#include "variable.h"
#include "constraint.h"
#include "solve.h"
#include "string.h"

void *safeMalloc(size_t size) {
  void *ret = malloc(size);
  assert(ret != NULL);
  return ret;
}

void *safeCalloc(int amount, size_t size) {
  void *ret = calloc(amount, size);
  assert(ret != NULL);
  return ret;
}

void *safeRealloc(void *oldPointer, size_t size) {
  void *ret = realloc(oldPointer, size);
  assert(ret != NULL);
  return ret;
}


/* BEGIN functions regarding datatype IntegerList */


IntegerList newIntegerList(int val, IntegerList next) {
  IntegerList intList = safeMalloc(sizeof(integerList));
  intList->val = val;
  intList->next = next;
  if(next != NULL) {
    next->prev = intList;
  }
  return intList;
}
```

```
IntegerList copyIntegerList(IntegerList orig) {
  IntegerList copy = safeMalloc(sizeof(integerList));
  copy->val = orig->val;
  copy->prev = NULL;
  copy->next = copyIntegerList(orig->next);
  copy->next->prev = copy;
  return copy;
}

void printIntegerList(IntegerList list) {
  while(list != NULL) {
    printf("%d ", list->val);
    list = list->next;
  }
  printf("\n");
}

void freeIntegerList(IntegerList list) {
  if(list != NULL) {
    freeIntegerList(list->next);
    free(list);
  }
}

/* END functions regarding datatype IntegerList */


/* BEGIN functions regarding datatype IntegerSet */

IntegerSet emptyIntegerSet() {
  IntegerSet set = safeMalloc(sizeof(integerSet));
  set->size = 0;
  set->space = 8;
  set->values = safeMalloc(set->space*sizeof(int));
  return set;
}

void doubleSetSpace(IntegerSet set) {
  set->space *= 2;
  set->values = safeRealloc(set->values, set->space*sizeof(int));
}

IntegerSet copyIntegerSet(IntegerSet orig) {
  IntegerSet copy = safeMalloc(sizeof(integerSet));
  copy->size = orig->size;
  copy->space = orig->space;
  copy->values = safeMalloc(copy->space*sizeof(int));
  copy->values = memcpy(copy->values, orig->values, copy->size*sizeof(int));
  return copy;
}

void freeIntegerSet(IntegerSet set) {
  free(set->values);
  free(set);
}

void printIntegerSet(IntegerSet set) {
  int i;
  printf("[");
  if(set->size > 0) {
```

```
    printf("%d", set->values[0]);
    for(i = 1; i < set->size; i++) {
      printf(", %d", set->values[i]);
    }
  }
  printf("]");
}

void checkSorted(IntegerSet set) {
  int i;
  for(i = 1; i < set->size; i++) {
    if(set->values[i] < set->values[i-1]) {
      printf("not sorted: ");
      printIntegerSet(set);
      printf("\n");
      exit(-1);
    }
  }
}

int findIndex(int *array, int size, int value) {
  int begin = 0;
  int end = size-1;

  while(begin <= end) {
    int mid = (begin + end) / 2;
    if(array[mid] > value) {
      end = mid-1;
    } else if(array[mid] < value) {
      begin = mid+1;
    } else {  /* array[mid] == value */
      return mid;
    }
  }
  /* begin > end -> value not found */
  return -(begin+1);
}

/*
 * Function that shifts values of array of IntegerSet set
 * The interval [start, start+len] is shifted to [dest, dest+len]
*/
void shiftValues(IntegerSet set, int start, int len, int dest) {
  int i;
  while(dest+len >= set->space) {
    doubleSetSpace(set);
  }
  if(start < dest) {
    for(i = len-1; i >= 0; i--) {
      set->values[dest+i] = set->values[start+i];
    }
  } else if(start > dest) {
    for(i = 0; i < len; i++) {
      set->values[dest+i] = set->values[start+i];
    }
  } /* else -> start == dest -> nothing to do */
}

static void printSet(IntegerSet set) {
  int i;
```

```
  for(i = 0; i < sizeOfSet(set); i++) {
    printf(" %d ", set->values[i]);
  }
  printf("\n");
}


void addIntegerToSet(IntegerSet set, int value) {
  int idx = findIndex(set->values, set->size, value);
  if(idx >= 0) {
    /* item is already available in set */
    return;
  }
  /* idx < 0 -> not in set, but should be inserted at position -(idx+1) */
  idx = -(idx+1);
  /* shift all values from index [idx, set->size-1] to [idx+1, set->size] */
  shiftValues(set, idx, set->size - idx, idx+1);
  /* put value at right index */
  set->values[idx] = value;
  set->size++;
}

void removeNthIntegerFromSet(IntegerSet set, int n) {
  int start, len, dest;
  if(n >= set->size) {
    /* index not available */
    return;
  }
  /* shift all values from indices [n+1, set->size-1] to [n, set->size-2] */
  start = n+1;
  dest = n;
  len = set->size-start;
  shiftValues(set, start, len, dest);
  set->size--;
}

void removeIntegerFromSet(IntegerSet set, int value) {
  int idx = findIndex(set->values, set->size, value);
  if(idx >= 0) {
    removeNthIntegerFromSet(set, idx);
  }
}

void addIntervalToSet(IntegerSet set, int min, int max) {
  int sizePartToShift;
  int i;
  int intervalSize = max-min+1;
  int intervalBegin = findIndex(set->values, set->size, min);
  int beginPartToShift = findIndex(set->values, set->size, max+1);
  int shiftDest;

  if(intervalBegin < 0) {
    intervalBegin = -(intervalBegin+1);
  }
  if(beginPartToShift < 0) {
    beginPartToShift = -(beginPartToShift+1);
  }

  sizePartToShift = set->size - beginPartToShift;
  set->size = intervalBegin + intervalSize + sizePartToShift;
```

```
  shiftDest = intervalBegin + intervalSize;

  /* shift values in set that occur after interval after sequence */
  shiftValues(set, beginPartToShift, sizePartToShift, shiftDest);

  /* add all values of interval to set */
  for(i = 0; i < intervalSize; i++) {
    set->values[intervalBegin+i] = min+i;
  }
  checkSorted(set);
}

IntegerSet intersect(IntegerSet set1, IntegerSet set2) {
  IntegerSet returnSet;
  if(set1->size > set2->size) {
    IntegerSet tmp = set1;
    set1 = set2;
    set2 = tmp;
  }
  returnSet = copyIntegerSet(set1); /* copy smallest IntegerSet */
  int i = 0;
  while(i < returnSet->size) {
    if(!valueInSet(set2, returnSet->values[i])) {
      removeNthIntegerFromSet(returnSet, i);  /* remove value at index i */
    } else {
      i++;                            /* i-th value should remain in set */
    }
  }
  return returnSet;
}

IntegerSet except(IntegerSet total, IntegerSet toBeRemoved) {
  IntegerSet returnSet = copyIntegerSet(total);
  int i;
  for(i = 0; i < toBeRemoved->size; i++) {
    removeIntegerFromSet(returnSet, toBeRemoved->values[i]);
  }
  return returnSet;
}

int minimumOfSet(IntegerSet set) {
  if(set->size == 0) {
    fprintf(stderr, "Set is empty -> no minimum value can be returned\n");
    exit(-1);
  }
  return set->values[0];
}

int maximumOfSet(IntegerSet set) {
  if(set->size == 0) {
    fprintf(stderr, "Domain is empty -> no minimum value can be returned\n");
  }
  return set->values[set->size-1];
}

int sizeOfSet(IntegerSet set) {
  return set->size;
}

int *valuesOfSet(IntegerSet set) {
```

```
  return set->values;
}

int valueInSet(IntegerSet set, int value) {
  int idx = findIndex(set->values, set->size, value);
  return (idx >= 0);
}

/* END functions regarding datatype IntegerSet */

/* BEGIN Functions List */

void freeList(List l) {
  while(l != NULL) {
    List tmp = l->next;
    free(l);
    l = tmp;
  }
}

List newListItem(void *data, List next) {
  List l = safeMalloc(sizeof(listItem));
  l->data = data;
  l->next = next;
  return l;
}

/* END Functions List */


/* BEGIN Functions Queue */

Queue emptyQueue() {
  Queue q = safeMalloc(sizeof(queue));
  q->first = NULL;
  q->last = NULL;
  return q;
}

int isEmptyQueue(Queue q) {
  return (q->first == NULL);
}

void freeQueue(Queue q) {
  freeList(q->first);
  free(q);
}

void enqueue(Queue q, void *item) {
  if(q->last != NULL) {
    q->last->next = newListItem(item, NULL);
    q->last = q->last->next;
  } else {  /* Queue is empty */
    q->first = newListItem(item, NULL);
    q->last = q->first;
  }
}

void *dequeue(Queue q) {
  if(q->first == NULL) {
```

```
    fprintf(stderr, "Error @dequeue: Queue is empty\n");
    exit(-1);
  }
  List first = q->first;
  void *ret = first->data;
  q->first = first->next;
  if(q->first == NULL) {
    q->last = NULL;
  }
  free(first);
  return ret;
}

/* END Functions Queue */


/* BEGIN Functions Stack */

Stack emptyStack() {
  Stack s = safeMalloc(sizeof(stack));
  s->first = NULL;
  return s;
}

int isEmptyStack(Stack s) {
  return (s->first == NULL);
}

void freeStack(Stack s) {
  freeList(s->first);
  free(s);
}

void push(Stack s, void *item) {
  s->first = newListItem(item, s->first);
}

void *pop(Stack s) {
  if(s->first == NULL) {
    fprintf(stderr, "Error @pop: Stack is empty\n");
    exit(-1);
  }
  void *ret = s->first->data;
  List toFree = s->first;
  s->first = s->first->next;
  free(toFree);
  return ret;
}

void *top(Stack s) {
  if(s->first == NULL) {
    fprintf(stderr, "Error @top: Stack is empty\n");
    exit(-1);
  }
  return s->first->data;
}
```

## B.6 CSP Problem

### Header file for data structure Problem (problem.h)

```
#ifndef PROBLEM_H
#define PROBLEM_H

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct problem *Problem;
typedef struct directedArc *DirectedArc;

#include "variable.h"
#include "constraint.h"

typedef enum {
  SOLVEALL, SOLVENR
} SolveType;

typedef struct SolveSpec {
  SolveType type;
  int max;
} SolveSpec;

typedef struct directedArc {
  Variable from;
  Constraint constraint;
  Variable to;
} directedArc;

typedef struct problem {
  int varCount;       /* nr of variables for this Problem */
  int assignCount;      /* nr of variables assigned */
  int constraintCount;    /* nr of constraints for this Problem */
  Variable *vars;       /* array of variables for this Problem */
  VarSeq varSequence;
  Constraint *constraints;  /* array of constraints for this Problem */
  SolveSpec solvespec;    /* how many solutions should be determined */
} problem;

void setVarsOfProblem(Problem p, VarList vl);
void setConstraintsOfProblem(Problem p, ConstraintList constraints);
void setDomainsOfVars(Problem p, VarList varIndices, IntegerSet d);

Variable varByIndex(Problem p, int index);
void setVarAtIndex(Problem p, int index, Variable v);
Constraint constraintByIndex(Problem p, int index);
void setConstraintAtIndex(Problem p, int index, Constraint c);
VarSeq varSeqOfProblem(Problem p);

void printProblem(Problem p);
/*void freeProblemVars(Problem p);*/
Problem emptyProblem();
void freeProblem(Problem p);



DirectedArc makeArc(Variable from, Constraint c, Variable to);
```

```
void freeArc(DirectedArc arc);
Variable firstVarOfArc(DirectedArc arc);
Variable secondVarOfArc(DirectedArc arc);
Constraint constraintOfArc(DirectedArc arc);



#endif
```

### Implementation data structure Problem (problem.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "datatypes.h"
#include <limits.h>
#include "datatypes.h"
#include "problem.h"
#include "variable.h"
#include "constraint.h"
#include "solve.h"

extern FILE *logFile;

static void setConstraintOfNumExp(NumExp exp, Constraint c, Problem p);
static void setConstraintOfFactor(Factor exp, Constraint c, Problem p);


static void setConstraintOfValue(Value v, Constraint c, Problem p) {
  if(v->type == VARVAL) {
    Variable var = varByIndex(p, v->data.varIndex);
    addConstraintToVar(var, c);
    addVarToConstraint(c, var);
  } else if(v->type == NUMEXP) {
    setConstraintOfNumExp(v->data.numexp, c, p);
  } else if(v->type == FUNCVAL) {
    int i;
    if(v->data.funcCall->name == ANY || v->data.funcCall->name == ALL) {
      ConstraintList *lists = v->data.funcCall->argv;
      ConstraintList l = lists[0];
      while(l != NULL) {
        setConstraintOfNumExp(firstExp(l->constraint), c, p);
        if(secondExp(l->constraint) != NULL) {
          setConstraintOfNumExp(secondExp(l->constraint), c, p);
        }
        l = l->next;
      }
    } else {
      NumExp *numExps = v->data.funcCall->argv;
      for(i = 0; i < v->data.funcCall->argc; i++) {
        setConstraintOfNumExp(numExps[i], c, p);
      }
    }
  }
  if(v->exponent != NULL) {
    setConstraintOfFactor(v->exponent, c, p);
  }
}

static void setConstraintOfFactor(Factor f, Constraint c, Problem p) {
```

```
  if(f->type == NUMNEG) {
    setConstraintOfFactor(f->data.factor, c, p);
  } else {
    setConstraintOfValue(f->data.value, c, p);
  }
}

static void setConstraintOfTerm(Term t, Constraint c, Problem p) {
  if(t->next == NULL) {
    setConstraintOfFactor(t->data, c, p);
  } else {
    setConstraintOfTerm(t->data, c, p);
    setConstraintOfTerm(t->next, c, p);
  }
}

static void setConstraintOfNumExp(NumExp exp, Constraint c, Problem p) {
  if(exp->next == NULL) {
    setConstraintOfTerm(exp->data, c, p);
  } else {
    setConstraintOfNumExp(exp->data, c, p);
    setConstraintOfNumExp(exp->next, c, p);
  }
}

static void freeProblemVars(Problem p) {
  int i;
  for(i = 0; i < p->varCount; i++) {
    freeVariable(p->vars[i]);
  }
  free(p->vars);
}

static void freeProblemConstraints(Problem p) {
  int i;
  for(i = 0; i < p->constraintCount; i++) {
    freeConstraint(p->constraints[i]);
  }
  free(p->constraints);
}

void linkConstraintsAndVars(Problem p) {
  int i;
  for(i = 0; i < p->constraintCount; i++) {
    Constraint c = constraintByIndex(p, i);
    setConstraintOfNumExp(firstExp(c), c, p);
    if(secondExp(p->constraints[i]) != NULL) {
      setConstraintOfNumExp(secondExp(c), c, p);
    }
  }
}

Problem emptyProblem() {
  Problem p = safeMalloc(sizeof(problem));
  p->varCount = 0;
  p->assignCount = 0;
  p->constraintCount = 0;
  /*p->connected = NULL;
  p->hasConstraint = NULL;*/
  p->vars = NULL;
```

```
  p->varSequence = NULL;
  p->constraints = NULL;
  p->solvespec.type = SOLVENR;
  p->solvespec.max = 1;
  return p;
}


void freeProblem(Problem p) {
  freeProblemVars(p);
  freeProblemConstraints(p);
  freeVarSeq(p->varSequence);
  free(p);
}


VarSeq varSeqOfProblem(Problem p) {
  return p->varSequence;
}


DirectedArc makeArc(Variable from, Constraint c, Variable to) {
  DirectedArc arc = safeMalloc(sizeof(directedArc));
  arc->from = from;
  arc->constraint = c;
  arc->to = to;
  return arc;
}

void freeArc(DirectedArc arc) {
  free(arc);
}

Variable firstVarOfArc(DirectedArc arc) {
  return arc->from;
}

Variable secondVarOfArc(DirectedArc arc) {
  return arc->to;
}

Constraint constraintOfArc(DirectedArc arc) {
  return arc->constraint;
}


void setVarsOfProblem(Problem p, VarList vl) {
  int i;
  Variable *vars;
  VarList tmp = vl;
  int varCount = 0;
  while(tmp != NULL) {
    varCount++;
    tmp = tmp->next;
  }
  vars = safeMalloc(varCount * sizeof(Variable));
  tmp = vl;
  for(i = 0; i < varCount; i++) {
    vars[i] = newVariable(tmp->varIndex, tmp->varType);
    tmp = tmp->next;
```

```
  }
  freeVarList(vl);
  p->varCount = varCount;
  p->vars = vars;
}

void setDomainsOfVars(Problem p, VarList toSet, IntegerSet d) {
  while(toSet != NULL) {
    freeIntegerSet(domainOfVar(p->vars[toSet->varIndex]));
    setDomainOfVar(p->vars[toSet->varIndex], copyIntegerSet(d));
    toSet = toSet->next;
  }
}

void setConstraintsOfProblem(Problem p, ConstraintList cl) {
  int i;
  ConstraintList tmp = cl;
  p->constraintCount = 0;
  while(tmp != NULL) {
    p->constraintCount++;
    tmp = tmp->next;
  }

  p->constraints = safeMalloc(p->constraintCount * sizeof(Constraint));
  tmp = cl;
  for(i = 0; tmp != NULL; i++) {
    p->constraints[i] = tmp->constraint;
    setIndexOfConstraint(tmp->constraint, i);
    tmp = tmp->next;
  }

  freeConstraintList(cl);
  linkConstraintsAndVars(p);
}

Variable varByIndex(Problem p, int index) {
  return p->vars[index];
}

void setVarAtIndex(Problem p, int index, Variable v) {
  p->vars[index] = v;
}

Constraint constraintByIndex(Problem p, int index) {
  return p->constraints[index];
}

void setConstraintAtIndex(Problem p, int index, Constraint c) {
  p->constraints[index] = c;
}
```

## B.7 Solving

### Header file solving methods and datastructures (solve.h)

```
#ifndef SOLVE_H
#define SOLVE_H

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "variable.h"
#include "problem.h"
#include <stdarg.h>

typedef struct solutionSet *SolutionSet;
typedef struct solutionList *SolutionList;

typedef struct solutionList {
  int *values;
  SolutionList next;
} solutionList;

typedef struct solutionSet {
  SolutionList first;
  SolutionList last;
  int varAmount;
  int solutionSpace;
  int solutionCount;
} solutionSet;

SolutionSet newSolutionSet(int varAmount, int solutionSpace);
void addSolution(SolutionSet solset, Problem p);
int solutionsLeft(SolutionSet solset);
void freeSolutionSet(SolutionSet solset);
void printSolution(int *solution, int varCount);

void addLog(const char * format, ...);
int init(Problem p);
Problem backtrack(Problem p);
SolutionSet solve(Problem p);


#endif
```

### Implementation solving methods and datastructures (solve.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <sys/resource.h>
#include "datatypes.h"
#include "problem.h"
#include "variable.h"
#include "constraint.h"
#include "backup.h"
#include "solve.h"

#define ON 1
```

```
#define OFF 0
#define NODE 1
#define ARC 2
#define FC 1
#define MAC 2

#define LOG OFF

/* which techniques should be applied while solving CSP */
#define MAKECONSISTENT    OFF   /* OFF/NODE/ARC */
#define MRV               OFF   /* ON/OFF */
#define DEGREE_HEURISTIC  OFF   /* ON/OFF */
#define MOSTCONNECTED     OFF   /* ON/OFF */
#define CP                OFF   /* OFF/FC/MAC */

/* function pointers that are set based on the applied techniques/heuristics */
int (*varOrdered)(Variable, Variable);
int (*propagationSuccess)(Variable, Problem, Backup);
static int **inArcsQueue;

/* contains file descriptor of logFile: also referred to in other files */
extern FILE *logFile;

/* counts the amount of backtracking calls */
unsigned long long stateCount = 0;

/* adds content to the logFile, if logging is enabled */
/* src: http://www.cplusplus.com/reference/cstdio/vprintf/ */
void addLog(const char * format, ...) {
  if(LOG) {
    va_list args;
    va_start (args, format);
    vfprintf (logFile, format, args);
    va_end (args);
  }
}

/*
  This function adds all directed arcs (X --> var) to a given queue
  an arc is unique by X, var and the constraint that connects these two
  (can be more than 1 constraint, so more than one arc representing (X --> var)
*/
void addVariableArcs(Variable var, Queue arcQueue, Problem p) {
  int i;
  int *varIndices;

  /* get indices of constraints in which var is involved */
  int *constraintIndices = constraintIndicesOfVar(var);
  int varAssigned = isAssigned(var, p);
  Constraint c;

  /* for all these constraints */
  for(i = 0; i < constraintAmountOfVar(var); i++) {
    c = constraintByIndex(p, constraintIndices[i]);

    /* if constraint is binary */
    if(arityOfConstraint(c) == 1 + !varAssigned) {
      varIndices = varIndicesOfConstraint(c);

      /* var2 becomes other variable in binary constraint */
```

```
      Variable var2 = varByIndex(p, varIndices[0]);
      if(var2 == var) {
        var2 = varByIndex(p, varIndices[1]);
      }

      /* if (constraint, var2) representing directed arc not already in queue */
      if(!inArcsQueue[c->index][var2->index]) {
        /* enqueue directed arc (var2 --> var) */
        DirectedArc arc = makeArc(var2, c, var);
        enqueue(arcQueue, arc);

        /* mark this arc as available in the queue */
        inArcsQueue[indexOfConstraint(c)][indexOfVar(var2)] = 1;
      }
    }
  }
}

/*
  This function adds all constraints for var with arity 3 that became binary
  after assigning var to a given queue.
*/
void addNewArcs(Variable var, Queue arcQueue, Problem p) {
  int i;
  int *varIndices;

  /* get indices of constraints in which var is involved */
  int *constraintIndices = constraintIndicesOfVar(var);
  Constraint c;

  /* for all these constraints */
  for(i = 0; i < constraintAmountOfVar(var); i++) {
    c = constraintByIndex(p, constraintIndices[i]);

    /* if constraint is binary */
    if(arityOfConstraint(c) == 2) {
      varIndices = varIndicesOfConstraint(c);

      /* var2, var3 becomes variable left in (now) binary constraint */
      Variable var2 = varByIndex(p, varIndices[0]);
      Variable var3 = varByIndex(p, varIndices[1]);

      /* if (constraint, var2) representing directed arc not already in queue */
      if(!inArcsQueue[c->index][var2->index]) {
        /* enqueue directed arc (var2 --> var) */
        DirectedArc arc = makeArc(var2, c, var);
        enqueue(arcQueue, arc);

        /* mark this arc as available in the queue */
        inArcsQueue[indexOfConstraint(c)][indexOfVar(var2)] = 1;
      }
      if(!inArcsQueue[c->index][var3->index]) {
        /* enqueue directed arc (var3 --> var) */
        DirectedArc arc = makeArc(var3, c, var);
        enqueue(arcQueue, arc);

        /* mark this arc as available in the queue */
        inArcsQueue[indexOfConstraint(c)][indexOfVar(var3)] = 1;
      }
    }
```

```
  }
}

/*
  This function ensures that all variables of CSP p are consistent
  regarding their unary constraints.
  If a value in the domain of a variable is not consistent
  with one of the unary constraints it is deleted.
  When all inconsistent values are removed, the constraint is also removed,
  such that it is not checked redundantly later on in the process.
*/
void makeNodeConsistent(Problem p) {
  int i;
  int *varIndices;
  Constraint constraint;
  Variable var;

  /* for each constraint of CSP p */
  for(i = 0; i < p->constraintCount; i++) {
    constraint = p->constraints[i];
    /* if constraint is unary */
    if(arityOfConstraint(constraint) == 1) {
      /* get indices of variables occurring in constraint */
      varIndices = varIndicesOfConstraint(constraint);
      /* get variable which is involved in this constraint */
      var = varByIndex(p, varIndices[0]);
      /* if domain of variable var is reduced */
      if(nodeReduce(var, constraint, p)) {
        /* if domain of var became empty -> error, no solution exists */
        if(domainSizeOfVar(var) == 0) {
          addLog(
            "No solutions found for the problem, \
            because variable X%d has an empty domain\n",
            var->index
          );
          printf(
            "\nNo solutions found for the problem, \
            because variable with empty domain\n"
          );
          exit(-1);
        }
      }
      /* remove unary constraint so that it is no longer checked */
      removeConstraintFromVar(var, constraint);
    }
  }
}

/*
  This function ensures that all domains of the variables of CSP are consistent
  regarding their binary constraints. After execution
  the CSP p is fully arc-consistent or strongly 2-consistent.
*/
void makeArcConsistent(Problem p) {
  int i;
  int *varIndices;
  Queue arcQueue = emptyQueue();

  /* first the CSP is made node-consistent */
  makeNodeConsistent(p);
```

```
  /* for each constraint of p */
  for(i = 0; i < p->constraintCount; i++) {
    Constraint constraint = constraintByIndex(p, i);
    /* get indices of variables occurring in constraint */
    varIndices = varIndicesOfConstraint(constraint);
    /* if constraint is a binary constraint */
    if(arityOfConstraint(constraint) == 2) {
      Variable var1 = varByIndex(p, varIndices[0]);
      Variable var2 = varByIndex(p, varIndices[1]);

      /* enqueue directed arc (var1 --> var2) */
      enqueue(arcQueue, makeArc(var1, constraint, var2));
      /* enqueue directed arc (var2 --> var1) */
      enqueue(arcQueue, makeArc(var2, constraint, var1));

      /* directional arc (var1 --> var2) is in queue */
      inArcsQueue[constraint->index][varIndices[0]] = 1;
      /* directional arc (var2 --> var1) is in queue */
      inArcsQueue[constraint->index][varIndices[1]] = 1;
    }
  }

  /* until queue is empty */
  while(!isEmptyQueue(arcQueue)) {
    DirectedArc arc = dequeue(arcQueue);
    Variable var1 = firstVarOfArc(arc);
    Constraint c = constraintOfArc(arc);
    /* arc is not in queue anymore */
    inArcsQueue[indexOfConstraint(c)][indexOfVar(var1)] = 0;

    if(arcReduce(arc, p)) {   /* if domain reduction */
      /* if domain of var became empty -> error, no solution exists */
      if(domainSizeOfVar(var1) == 0) {
        addLog(
          "No solutions found for the problem, \
          because variable X%d has an empty domain\n",
          indexOfVar(var1)
        );
        printf(
          "\nNo solutions found for the problem, \
          because variable with empty domain\n"
        );
        exit(-1);
      }
      /* because domain is reduced, domains of neighbours might be reduced */
      addVariableArcs(var1, arcQueue, p);
    }
    freeArc(arc);
  }

  freeQueue(arcQueue);
}

/*
  Checks if the assignment to variable var is consistent
  with current partial assignment.
*/
int checkLocalConsistency(Variable var, Problem p) {
  int i;
```

```
  int *constraintIndices = constraintIndicesOfVar(var);
  Constraint c;

  /* for all constraints connected to variable var */
  for(i = 0; i < constraintAmountOfVar(var); i++) {
    c = constraintByIndex(p, constraintIndices[i]);
    /* check if inconsistent with partial solution */
    if(determinable(c, p) && !checkConstraint(c, p)) {
      return 0;
    }
  }
  return 1;
}

/*
  This function checks if the current partial solution after assignment of var
  is consistent and (if techniques applied) removes invalid values from domains
  of unassigned variables by constraint propagation
*/
int isConsistent(Variable var, Problem p, Backup b) {
  if(checkLocalConsistency(var, p)) {
    return propagationSuccess(var, p, b);
  }
  return 0;
}

/*
  Function that makes the queue empty and sets all arcs as 'not available'
  for queue containing directed arcs.
*/
void makeArcQueueEmpty(Queue arcQueue) {
  /* for all arcs in queue */
  while(!isEmptyQueue(arcQueue)) {
    DirectedArc arc = dequeue(arcQueue);
    Constraint c = constraintOfArc(arc);
    Variable var1 = firstVarOfArc(arc);
    /* set arc (var1 --> otherVar) by constraint c 'not available' */
    inArcsQueue[indexOfConstraint(c)][indexOfVar(var1)] = 0;
    freeArc(arc);
  }
}

/*
  Function that can be used as placeholder if no constraint propagation
  is applied.
*/
int skipPropagationTest(Variable var, Problem p, Backup backup) {
  return 1;
}

int forwardChecking(Variable var, Problem p, Backup backup) {
  Queue arcQueue = emptyQueue();

  addLog("forward checking based on assignment of variable X%d\n", var->index);

  /* enqueue all arcs directed at var, so all (X --> var) */
  addVariableArcs(var, arcQueue, p);

  /* for all arcs in queue */
  while(!isEmptyQueue(arcQueue)) {
```

```
    /* dequeue arc */
    DirectedArc arc = dequeue(arcQueue);
    Constraint c = constraintOfArc(arc);
    Variable var1 = firstVarOfArc(arc);

    /* set arc 'not available' */
    inArcsQueue[indexOfConstraint(c)][indexOfVar(var1)] = 0;

    /* make backup before reduction */
    IntegerSet domBackup = copyIntegerSet(domainOfVar(var1));
    varPos seqBackup = *(sequencePosition(var1));

    /* check if domain of variable can be reduced by arc */
    int reduced = arcReduce(arc, p);
    freeArc(arc);

    /* if domain of variable is reduced by arc */
    if(reduced) {
      /* add backup of old domain and sequence position */
      addBackup(var1->index, domBackup, seqBackup, backup);
      /* resort variable in sequence */
      resortVarSeq(p->varSequence, sequencePosition(var1));
      addLog(
        "X%d = %d -> Domain limited of variable X%d.\n",
        indexOfVar(var), domainMinimumOfVar(var), indexOfVar(var1)
      );
      int domSize = domainSizeOfVar(var1);
      /* if domain of variable became empty after reduction */
      if(domSize == 0) {
        makeArcQueueEmpty(arcQueue);
        freeQueue(arcQueue);
        /* return error */
        return 0;
      }
    } else {
      /* backup not needed */
      freeIntegerSet(domBackup);
    }
  }
  freeQueue(arcQueue);

  /* queue is empty and no error occured */
  return 1;
}

/*
  Function that performs constraint propagation after assignment
  If CP is set to MAC, then arc-consistency is maintained.
*/
int mac(Variable var, Problem p, Backup backup) {
  Queue arcQueue = emptyQueue();

  addLog("Maintaining arc consistency after assignment of variable X%d\n", var->index
      );

  /* enqueue all arcs directed at var, so all (X --> var) */
  addVariableArcs(var, arcQueue, p);

  /* enqueue all new arcs (two directions of constraints that had arity 3
      before assignment of var, but became binary after) */
```

```
  addNewArcs(var, arcQueue, p);

  /* for all arcs in queue */
  while(!isEmptyQueue(arcQueue)) {
    /* dequeue arc */
    DirectedArc arc = dequeue(arcQueue);
    Constraint c = constraintOfArc(arc);
    Variable var1 = firstVarOfArc(arc);

    /* set arc 'not available' */
    inArcsQueue[indexOfConstraint(c)][indexOfVar(var1)] = 0;

    /* make backup before reduction */
    IntegerSet domBackup = copyIntegerSet(domainOfVar(var1));
    varPos seqBackup = *(sequencePosition(var1));

    /* check if domain of variable can be reduced by arc */
    int reduced = arcReduce(arc, p);
    freeArc(arc);

    /* if domain of variable is reduced by arc */
    if(reduced) {
      /* add backup of old domain and sequence position */
      addBackup(var1->index, domBackup, seqBackup, backup);
      /* resort variable in sequence */
      resortVarSeq(p->varSequence, sequencePosition(var1));
      addLog(
        "X%d = %d -> Domain limited of variable X%d.\n",
        indexOfVar(var), domainMinimumOfVar(var), indexOfVar(var1)
      );
      int domSize = domainSizeOfVar(var1);
      /* if domain of variable became empty after reduction */
      if(domSize == 0) {
        makeArcQueueEmpty(arcQueue);
        freeQueue(arcQueue);
        /* return error */
        return 0;
      }
      /* apply constraint propagation for variable with reduced domain */
      addVariableArcs(var1, arcQueue, p);
    } else {
      /* backup not needed */
      freeIntegerSet(domBackup);
    }
  }
  freeQueue(arcQueue);

  /* queue is empty and no error occured */
  return 1;
}

Variable selectUnassignedVar(Problem p) {
  VarPos position = firstVarPosition(p->varSequence);
  Variable v = varAtPosition(p->varSequence, position);
  return v;
}


int init(Problem p) {
  int i, j;
```

```
  p->assignCount = 0;

  int possibilities = 0;
  int maxDomSize = 0;
  int *domainCounts = safeCalloc(1, sizeof(int));
  for(i = 0; i < p->varCount; i++) {
    int domSize = domainSizeOfVar(varByIndex(p, i));
    possibilities += domSize;
    if(domSize > maxDomSize) {
      domainCounts = safeRealloc(domainCounts, (domSize+1)*sizeof(int));
      for(j = maxDomSize+1; j <= domSize; j++) {
        domainCounts[j] = 0;
      }
      maxDomSize = domSize;
    }
    domainCounts[domSize]++;
  }
  addLog("\n");
  addLog(
    "# Before init: %d possible values and %d ",
    possibilities, (domainCounts[0] == 0)
  );
  for(i = 1; i <= maxDomSize; i++) {
    if(domainCounts[i]) {
      addLog("* %d^%d ", i, domainCounts[i]);
    }
  }
  addLog("combinations\n");
  addLog(
    "# Applying substitution for single-value domains \
    and checking applicable constraints:\n"
  );

  switch(MAKECONSISTENT) {
    case NODE:
      makeNodeConsistent(p);
      break;
    case ARC:
      makeArcConsistent(p);
      break;
  }

  switch(CP) {
    case FC:
      propagationSuccess = forwardChecking;
      break;
    case MAC:
      propagationSuccess = mac;
      break;
    default:
      propagationSuccess = skipPropagationTest;
      break;
  }

  possibilities = 0;
  for(i = 0; i <= maxDomSize; i++) {
    domainCounts[i] = 0;
  }
  for(i = 0; i < p->varCount; i++) {
    int domSize = domainSizeOfVar(varByIndex(p, i));
```

```
    possibilities += domSize;
    domainCounts[domSize]++;
  }
  addLog(
    "# After init: %d possible values and %d ",
    possibilities, (domainCounts[0] == 0)
  );
  for(i = 1; i <= maxDomSize; i++) {
    if(domainCounts[i]) {
      addLog("* %d^%d ", i, domainCounts[i]);
    }
  }
  addLog("combinations\n");
  free(domainCounts);
  addLog("# After init: domains\n");
  for(i = 0; i < p->varCount; i++) {
    addLog("X%d: ", i);
    printDomainOfVar(varByIndex(p, i));
    addLog("\n");
  }
  return 1;
}


SolutionSet newSolutionSet(int varAmount, int solutionSpace) {
  SolutionSet set = safeMalloc(sizeof(solutionSet));
  set->first = NULL;
  set->last = NULL;
  set->solutionSpace = solutionSpace;
  set->solutionCount = 0;
  set->varAmount = varAmount;
  return set;
}

 void printSolution(int *solution, int varCount) {
  int i;

  if(varCount == 81) {
    for(i = 0; i < varCount; i++) {
      printf("%d  ", solution[i]);
      if((i+1) % 9 == 0) {
        printf("\n\n");
      }
    }
    printf("\n");
    return;
  }

  for(i = 0; i < varCount; i++) {
    printf("X%d = %d;\n", i, solution[i]);
  }
}

SolutionList newSolution(Problem p) {
  int i;
  SolutionList list = safeMalloc(sizeof(solutionList));
  list->values = safeMalloc(p->varCount*sizeof(int));
  for(i = 0; i < p->varCount; i++) {
    list->values[i] = domainMinimumOfVar(varByIndex(p, i));
  }
```

```
  list->next = NULL;
  return list;
}

void addSolution(SolutionSet solset, Problem p) {
  if(solset->solutionCount == solset->solutionSpace) {
    fprintf(stderr, "No more solutions can be added to this set\n");
    exit(-1);
  }
  if(solset->solutionCount == 0) {
    solset->first = newSolution(p);
    solset->last = solset->first;
  } else {
    solset->last->next = newSolution(p);
    solset->last = solset->last->next;
  }
  solset->solutionCount++;
}

int solutionsLeft(SolutionSet solset) {
  return (solset->solutionSpace - solset->solutionCount != 0);
}

void freeSolutionList(SolutionList list) {
  if(list != NULL) {
    freeSolutionList(list->next);
    free(list->values);
    free(list);
  }
}

void freeSolutionSet(SolutionSet set) {
  freeSolutionList(set->first);
  free(set);
}

void addVarToConstraints(Variable var, Problem p) {
  int i;
  int *constraintIndices = constraintIndicesOfVar(var);

  /* for every constraint c of var */
  for(i = 0; i < constraintAmountOfVar(var); i++) {
    Constraint c = constraintByIndex(p, constraintIndices[i]);

    /* if arity of constraint is unary, becomes binary */
    if(arityOfConstraint(c) == 1) {
      int *varIndices = varIndicesOfConstraint(c);
      Variable var2 = varByIndex(p, varIndices[0]);
      /* other variable gets constraint on other variable (var) */
      plusDegree(var2);
      /* var2 is still in sequence -> resort */
      resortVarSeq(p->varSequence, sequencePosition(var2));
    }

    int j;
    for(j = 0; j < arityOfConstraint(c); j++) {
      int *varIndices = varIndicesOfConstraint(c);
      Variable var2 = varByIndex(p, varIndices[j]);
      lowerConnectivity(var2);
    }
```

```
    /* add var to c */
    addVarToConstraint(c, var);
  }
}

void resetVar(Problem p, Variable v, varPos pos, IntegerSet dom) {
  setDomainOfVar(v, dom);
  addVarToConstraints(v, p);
  restoreVarSeq(p->varSequence, pos);
}

void removeVarFromConstraints(Variable var, Problem p) {
  int i;
  int *constraintIndices = constraintIndicesOfVar(var);

  /* for every constraint c of var */
  for(i = 0; i < constraintAmountOfVar(var); i++) {
    Constraint c = constraintByIndex(p, constraintIndices[i]);
    /* remove var from c */
    removeVarFromConstraint(c, var);

    /* if arity of constraint was binary, became unary */
    if(arityOfConstraint(c) == 1) {
      int *varIndices = varIndicesOfConstraint(c);
      Variable var2 = varByIndex(p, varIndices[0]);
      /* other variable loses constraint on other variable (var) */
      lowerDegree(var2);
      /* var2 is still in sequence -> resort */
      resortVarSeq(p->varSequence, sequencePosition(var2));
    }

    int j;
    for(j = 0; j < arityOfConstraint(c); j++) {
      int *varIndices = varIndicesOfConstraint(c);
      Variable var2 = varByIndex(p, varIndices[j]);
      plusConnectivity(var2);
    }
  }
}

void recursiveBacktracking(Problem p, SolutionSet solset) {
  Variable var;
  varPos sequencePos;
  int *values, i;
  IntegerSet fullDomain;

  stateCount++;
  if(p->varCount == p->assignCount) {
    addSolution(solset, p);
    return;
  }

  var = selectUnassignedVar(p);

  sequencePos = *sequencePosition(var);
  fullDomain = domainOfVar(var);
  values = valuesOfSet(fullDomain);
  removeVarFromConstraints(var, p);
  removeVarFromSequence(p->varSequence, var->sequencePos);
```

```
  for(i = 0; i < sizeOfSet(fullDomain) && solutionsLeft(solset); i++) {
    addLog("Trying value %d for variable X%d.\n", values[i], var->index);
    p->assignCount++;
    setDomainOfVar(var, createSingletonDomain(values[i]));
    Backup backup = emptyBackup();
    if(isConsistent(var, p, backup)) {
      recursiveBacktracking(p, solset);
    }
    restoreBackup(backup, p);
    freeIntegerSet(domainOfVar(var));
    p->assignCount--;
  }
  resetVar(p, var, sequencePos, fullDomain);
}

int checkConstantConstraints(Problem p) {
  int i;
  for(i = 0; i < p->constraintCount; i++) {
    if(arityOfConstraint(p->constraints[i]) == 0 &&
       !satisfiable(p->constraints[i], p)) {
      addLog("constant constraint %d is not satisfiable:\n", i);
      printConstraint(p->constraints[i]);
      return 0;
    }
  }
  return 1;
}


SolutionSet solve(Problem p) {
  int i;

  /* set right function pointers */
  if(MRV) {
    varOrdered = mrvOrdered;
    if(DEGREE_HEURISTIC) {
      varOrdered = mrvPlusDegreeOrdered;
    } else if(MOSTCONNECTED) {
      varOrdered = mrvPlusConnectedOrdered;
    }
  } else if(MOSTCONNECTED) {
    varOrdered = mostConnectedOrdered;
  } else {
    varOrdered = skipTest;
  }

  inArcsQueue = safeMalloc(p->constraintCount * sizeof(int *));
  for(i = 0; i < p->constraintCount; i++) {
    inArcsQueue[i] = safeCalloc(p->varCount, sizeof(int));
  }

  SolutionSet solset = newSolutionSet(p->varCount,
      (p->solvespec.type == SOLVEALL ? -1 : p->solvespec.max));

  p->varSequence = emptyVarSeq();

  addLog("\n#######################################################\n");

  if(checkConstantConstraints(p)) {
```

```
    /* for all constraints of problem */
    for(i = 0; i < p->constraintCount; i++) {
      Constraint c = constraintByIndex(p, i);
      /* if arity of c > 1 */
      if(arityOfConstraint(c) > 1) {
        int *varIndices = varIndicesOfConstraint(c);
        int j;
        /* for all variables of c */
        for(j = 0; j < arityOfConstraint(c); j++) {
          /* degree++ */
          plusDegree(varByIndex(p, varIndices[j]));

          int k;
          for(k = j+1; k < arityOfConstraint(c); k++) {
            plusVarConnections(varByIndex(p, varIndices[j]));
            plusVarConnections(varByIndex(p, varIndices[k]));
          }
        }
      }
    }

    init(p);
    int i;
    for(i = 0; i < p->varCount; i++) {
      insertVarInSequence(p->varSequence, varByIndex(p, i));
    }
    recursiveBacktracking(p, solset);
  }

  for(i = 0; i < p->constraintCount; i++) {
    free(inArcsQueue[i]);
  }
  free(inArcsQueue);

  fprintf(logFile, "backtracking points:  %llu\n", stateCount);

  printf("backtracking points:  %llu\n", stateCount);

  return solset;
}
```

## B.8 Main file

### main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <assert.h>
#include "datatypes.h"
#include "solve.h"
#include <limits.h>

FILE *logFile;

extern void parser(Problem LLuserData, Problem *LLretval);
```

```
void Main() {
  Problem p = NULL;

  parser(p, &p);

  addLog("Accepted\n");

  SolutionSet solset = solve(p);
  SolutionList solution = solset->first;

  int i = 0;
  while(solution != NULL) {
    printf("\n########## SOLUTION #%d ##########\n", i+1);
    printSolution(solution->values, p->varCount);
    solution = solution->next;
    i++;
  }

  if(solset->solutionCount == 0) {
    printf("\nNo solution could be found\n");
  } else if(p->solvespec.type == SOLVEALL) {
    printf("\nFound %d solution(s).\n", solset->solutionCount);
  } else if(solset->solutionCount < p->solvespec.max) {
    /* p->solvespec.type == SOLVENR */
    printf("\nNo more than %d solution(s) "
    "could be found.\n", solset->solutionCount);
  }

  freeSolutionSet(solset);

  freeProblem(p);
}

void timedMain() {
  clock_t start, stop;
  start = clock();
  Main();
  stop = clock();
  printf ("Cpu time: %.3f seconds\n", ((double)stop - (double)start)*1.0e-6);
}

int main(int argc, char *argv[]) {
  if(argc == 2) {
    stdin = fopen(argv[1], "r");
  }
  logFile = fopen("csp.log", "w");

  if (logFile == NULL) {
    fprintf(stderr, "Error opening log file!\n");
    exit(1);
  }

  timedMain();

  fclose(logFile);

  return EXIT_SUCCESS;
}
```

## B.9 Makefile

### Makefile

```
CC=gcc
CFLAGS=-Wall -O6 -g -pg
OBJS=grammar.o datatypes.o backup.o solve.o constraint.o variable.o problem.o lex.yy.
    o main.o
LIBS=-lm

all: grammar.c lex.yy.c ${OBJS}
  gcc -Wall -pg -O6 -o csp ${OBJS} ${LIBS}
grammar.c: grammar.g
  LLnextgen grammar.g
lex.yy.c: flex.fl
  flex flex.fl
clean:
  rm -f *~
  rm -f *.o
```