CONSTRAINTS SATISFACTION PROBLEM (CSP)

Seminary

Student: Chaves Noelia
<table>
<thead>
<tr>
<th>Contents</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Definition</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Objective</td>
<td>2</td>
</tr>
<tr>
<td><strong>2 Constraint Satisfaction: An Emerging Paradigm</strong></td>
<td>3</td>
</tr>
<tr>
<td>2.1 The Constraint Satisfaction Problem: Representation and Reasoning</td>
<td>3</td>
</tr>
<tr>
<td>2.1.1 The magic square problem</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Solvers and Libraries for CP</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Planning and Scheduling Activities for Earth Surveillance and Observation Satellites</td>
<td>4</td>
</tr>
<tr>
<td>2.4 CHOCO</td>
<td>5</td>
</tr>
<tr>
<td>2.4.1 Variables</td>
<td>5</td>
</tr>
<tr>
<td>2.4.2 Constraints</td>
<td>6</td>
</tr>
<tr>
<td>2.4.3 The solver</td>
<td>8</td>
</tr>
<tr>
<td>2.4.4 Advanced uses of Choco</td>
<td>12</td>
</tr>
<tr>
<td>2.4.5 Constraints (Model)</td>
<td>14</td>
</tr>
<tr>
<td>2.4.6 Branching strategies (Solver)</td>
<td>15</td>
</tr>
<tr>
<td>2.4.7 Variable selectors (Solver)</td>
<td>15</td>
</tr>
<tr>
<td>2.4.8 Value iterators (Solver)</td>
<td>15</td>
</tr>
<tr>
<td>2.4.9 Value selector (Solver)</td>
<td>15</td>
</tr>
<tr>
<td>2.4.10 The magic square problem</td>
<td>16</td>
</tr>
<tr>
<td>2.4.11 The output</td>
<td>19</td>
</tr>
<tr>
<td>2.5 Works or projects related to CSP</td>
<td>19</td>
</tr>
<tr>
<td>2.6 Conclusions</td>
<td>19</td>
</tr>
</tbody>
</table>
CONTENTS

Bibliography 21
CHAPTER 1

Introduction

1.1 Definition

Constraint programming is a powerful paradigm for modeling and solving combinatorial optimization problems, that draws on a wide range of techniques from artificial intelligence, computer science, databases, programming languages, and operations research (See [4]).

Constraint programming is currently applied with success to many domains, such as scheduling, planning, vehicle routing, configuration, networks, and bioinformatics (See [4]).

The basic idea in constraint programming is that the user states the constraints and a general purpose constraint solver is used to solve them.

Constraints are just relations, and a constraint satisfaction problem (CSP) states which relations should hold among the given decision variables.

Constraint solvers take a real-world problem, represented in terms of decision variables and constraints, and find an assignment to all the variables that satisfies the constraints.

While defining a set of constraints may seem a simple way to model a real-world problem, finding a good model that works well with a chosen solver is not always easy. Thus much care must be devoted to choosing a good model and also to devising solvers that can exploit the features of the chosen model.

A fundamental challenge in constraint programming is to understand the computational complexity of problems involving constraints. In their most general form, constraint satisfaction problems (CSPs) are NP-Hard$^1$.

---

$^1$If there is a polynomial algorithm for any NP-hard problem, then there are polynomial algorithms for all problems in NP. See: http://en.wikipedia.org/wiki/NP-hard.
1.1 **Definition**

To counter this pessimistic result, much work has been done on identifying restrictions on constraint satisfaction problems such that solving an instance can be done efficiently; that is, in polynomial time in the worst-case.

1.1.1 **Objective**

Our objective is to understand how is used inside of the solvers, which solvers there are, how is applied to planning and scheduling for activities for earth surveillance and observation satellite and examples of projects on this topic.
2.1 The Constraint Satisfaction Problem: Representation and Reasoning

The classic definition of a Constraint Satisfaction Problem (CSP) is as follows:

A CSP $P$ is a triple $P = (X, D, C)$ where $X$ is an $n$-tuple of variables $X = (x_1, x_2, \ldots, x_n)$, $D$ is a corresponding $n$-tuple of domains $D = (D_1, D_2, D_3, \ldots, D_n)$ such that $x_i \in D_i$, and $C$ is a $t$-tuple of constraints $C = (C_1, C_2, C_3, \ldots, C_t)$.

Or as shown in [2]

- A finite set of variables
- Domains – a finite set of values for each variable
- A finite set of constraints
  - is an arbitrary relation over the set of variables
  - can be defined extensionally (a set of compatible tuples) or intentionally (formula)

And a solution to CSP is a complete assignment of variables satisfying all the constraints.

Following, we give an example that is possible to resolve with CSP.

2.1.1 The magic square problem

A magic square (see [3]) of order $n$ is an arrangement of $n^2$ numbers, usually distinct integers, in a square, such that the $n$ numbers in all rows, all columns, and both diagonals sum to the same constant. A standard magic square contains the integers from 1 to $n^2$. 
2.3 Planning and Scheduling Activities for Earth Surveillance and Observation Satellites

Figure 2.1: Magic

The constant sum in every row, column and diagonal is called the magic constant or magic sum $M$. The magic constant of a classic magic square depends only on $n$ and has the value: $M(n) = n(n^2 + 1)/2$.

For more information see http://en.wikipedia.org/wiki/Magic_square.

2.2 Solvers and Libraries for CP

There are many solvers, but we will list some:

- **MINION**
  - Windows, Linux
  - Availability: Free

- **Choco**
  - http://choco.sourceforge.net/
  - Linux
  - Availability: Free

- **KOALOG**
  - Windows, Linux
  - Availability: Commercial

We can see others in http://csplib.org/.

2.3 Planning and Scheduling Activities for Earth Surveillance and Observation Satellites

We defined the following terms in [2](See http://ktiml.mff.cuni.cz/ bartak/constraints/):
The planning task is to find out a sequence of actions that will transfer the initial state of the world into a state where the desired goal is satisfied.

The scheduling task is to allocate known activities to available resources and time respecting capacity, precedence (and other) constraints.

We can find the following example in space domain (See [1]):

Constraints to be satisfied are listed:

- Temporal constraints:
  - visibility windows for observations and downloads
  - durations of observations and downloads
  - no overlapping between observations and between downloads;
  - (time-dependent) transition times between observations;
  - precedences: observation precedes download.

- Resource constraints: memory, energy, instrument temperature, total On time, total number of On/Offs . . .

- Logical activation constraints:
  - download requires observation (before);
  - observation requires instrument ON (during);
  - stereo observations (either both or none).

Another example is displayed in [5]:

They present an example inspired from current practice at the European Space Agency (ESA) mission control center. The planning problem consists in deciding data transmission commands from a satellite orbiting Mars to Earth within given ground station visibility windows.

2.4 CHOCO

We choose ‘CHOCO’ to understand how the solver works internally. We will show how it works.

The Choco Model allows to describe a problem in an easy and declarative way: it simply records the variables and the constraints defining the problem.

2.4.1 Variables

A Variable is defined by a type (integer, real, or set variable), a name, and the values of its domain.

Constants

A constant is a variable with a fixed domain.
2.4 CHOCO

Expression variables

Expression variables represent the result of combinations between variables of the same type made by operators.

Decision/non-decision variables

In choco each variable added to a model is a decision variable, i.e. is included in the default search strategy\(^3\).

A variable can be stated as a non decision one if its value can be computed by side-effect. To specify non decision variables, one can:

- exclude them from the search strategies
- specify non-decision variables (adding Options.V_NO_DECISION to their options) and keep the default search strategy.

Objective variable

You can define an objective variable directly within the model, only one variable can be defined as an objective.

2.4.2 Constraints

Choco provides a large number of simple and global constraints and allows the user to easily define its own new constraint (see [3]).

Adding a constraint automatically adds its variables to the model (explicit declaration of variables addition is not mandatory). Thus, a variable not involved in any constraints will not be declared in the Solver during the reading step.

Binary constraints

Constraints involving two integer variables. Example: eq.

Ternary constraints Constraints

Constraints involving three integer variables. Example: distanceEq.

Constraints involving real variables

Constraints involving two real variables. Example: eq.

Constraints involving set variables

Example: members.

\(^3\)composition of branching strategies.
2.4 CHOCO

Constraints in extension and relations

Choco supports the statement of constraints defining arbitrary relations over two or more variables. Such a relation may be defined by three means:

- feasible table: the list of allowed tuples of values
- infeasible table: the list of forbidden tuples of values
- infeasible table: the list of forbidden tuples of values

Reified constraints

The truth value of a constraint is a boolean that is true if and only if the constraint holds. To reify a constraint is to get its truth value. This mechanism can be used for example to model a MaxCSP problem where the number of satisfied constraints has to be maximized. It is also intended to give the freedom to model complex constraints combining several reified constraints, using some logical operators on the truth values, such as in: \((x \neq y) \lor (z \leq 9)\).

Choco provides a generic constraint `reifiedConstraint` to reify any constraint into a boolean variable expressing its truth value:

- `Constraint reifiedConstraint(IntegerVariable b, Constraint c);`
- `Constraint reifiedConstraint(IntegerVariable b, Constraint c1, Constraint c2);`

Handling complex expressions

In order to build complex combinations of constraints, Choco also provides a simpler and more direct API with the following logical meta-constraints taking constraints in arguments:

- `and`, `or`, `implies`, `ifOnlyIf`, `ifThenElse`, `not`, `nand`, `nor`

For example, the following expression \(((x = 10 \times y) \lor (z \leq 9)) \iff \text{alldifferent}(a, b, c)\)

could be expressed in Choco by:

```java
Constraint exp = ifOnlyIf( or( eq(x, mult(10, y)), leq(z, 9) ), alldifferent(new IntegerVariable[]a,b,c) );
```

Such an expression is internally represented as a tree whose nodes are operators and leaves are variables, constants and constraints.

The language available on expressions currently matches the language used in the Constraint Solver Competition 2008 of the CPAI workshop.

At the solver level, there exists two different ways to represent expressions:

- by extension: the first way is to handle expressions as constraints in extension. The expression is then used to check a tuple in a dynamic way just like a n-ary
2.4 CHOCO

relation that is defined without listing all the possible tuples. The expression is then propagated using the GAC3rm algorithm. This is very powerful as arc-consistency is achieved on the corresponding constraints.

- by decomposition: the second way is to decompose the expression automatically by introducing intermediate variables. By doing so, the level of pruning decreases but expressions of larger arity involving large domains can be represented.

The way to represent expressions is decided at the modeling level. Representation by extension is the default. Representation by decomposition can be set instead.

Global constraints

Choco includes several global constraints. Those constraints accept any number of variables and offer dedicated filtering algorithms which are able to make deductions where a decomposed model would not. For instance, constraint alldifferent(a, b, c, d) with \( a, b \in [1, 4] \) and \( c, d \in [3, 4] \) allows to deduce that \( a \) and \( b \) cannot be instantiated to 3 or 4; such rule cannot be inferred by simple binary constraints.

Value constraints

Constraints that put a restriction on how values can be distributed among a collection of variables. Example of counting distinct values: allDifferent.

Boolean constraints

Logical operations on boolean expressions. Example: and.

Packing constraints (capacitated resources)

Constraints involving items to be packed in bins without overlapping. More generally, any constraints modelling the concurrent assignment of objects to one or several capacitated resources. Example of packing problems: equation.

Scheduling constraints (time assignment)

Constraints involving tasks to be scheduled over a time horizon. Example of temporal contraints: disjoint (tasks) precedence.

2.4.3 The solver

The Choco Solver is mainly focused on resolution part: reading the Model, defining the search strategies and the resolution policy.

To create a Solver, one just needs to create a new object as follow:

```java
Solver solver = new CPSolver();
```

This instruction creates a Constraint Programming (CP) Solver object.
2.4 CHOCO

The solver gives an API to read a model. The reading of a model is compulsory and must be done after the entire definition of the model.

```java
solver.read(model);
```

The reading is divided in two parts: variables reading and constraints reading.

The variables are declared in a model with a given type `IntegerVariable`, `SetVariable`, `RealVariable` and, possibly, with a given domain type (e.g. bounded or enumerated domains for integer and set variables).

**Bound variables** are related to large domains which are only represented by their lower and upper bounds.

The level of consistency achieved by most constraints on these variables is called bound-consistency.

On the contrary, the domain of an enumerated variable is explicitly represented and every value is considered while pruning. Basic constraints are therefore often able to achieve arc-consistency on enumerated variables.

### Constraints reading

Once the solver variables are created when reading the model, the solver then iterates over the constraints of the model. At this step, auxiliary solver variables and constraints may be generated. The created constraints are then added to the internal constraint network. Each solver constraint encapsulates a filtering algorithm which is called, during the search, when a propagation step occurs or when an external event (e.g., value removal or bound modification) happens on some variable of the constraint.

### Solve a problem

Table below presents the different API offered by Solver to launch the problem resolution. All these methods return a Boolean object standing for the problem feasibility status of the solver:

- `Boolean.TRUE` if at least one feasible solution has been computed,
- `Boolean.FALSE` if the problem is proved to be infeasible,
- `null` otherwise, i.e. when a search limit has been reached before.

<table>
<thead>
<tr>
<th>Solver API</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>solve()</code> or <code>solve(false)</code></td>
<td>runs backtracking until reaching a first feasible solution or the proof of infeasibility or a search limit.</td>
</tr>
<tr>
<td><code>nextSolution()</code></td>
<td>Can only be called after a <code>solve()</code> or a <code>nextSolution()</code> call that has returned <code>Boolean.TRUE</code>. Runs backtracking, from the solution leaf reached by the previous <code>solve()</code> or <code>nextSolution()</code> call, until reaching a new feasible solution or proving no such new solution exists or reaching a search limit.</td>
</tr>
<tr>
<td><code>isFeasible()</code></td>
<td>Returns the feasibility status of the solver.</td>
</tr>
</tbody>
</table>
The following API are also useful to manipulate a Solver object:

<table>
<thead>
<tr>
<th>Solver API</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>propagate()</td>
<td>Launches propagation by running, in turn, the domain reduction algorithms of the constraints until it reaches a fix point. Throws a ContradictionException when a contradiction is detected, i.e. a variable domain is emptied. This method is called at each node of the tree search constructed by the solving methods above.</td>
</tr>
</tbody>
</table>

Search Strategy

A key ingredient of any constraint approach is a clever search strategy. In backtracking or branch-and-bound approaches, the search is organized as an enumeration tree, where each node corresponds to a subspace of the search, and each child node is a subdivision of its father node’s space. The tree is progressively constructed by applying a series of branching strategies that determine how to subdivide space at each node and in which order to explore the created child nodes. Branching strategies play the role of achieving intermediate goals in logic programming.

Standard backtracking or branch-and-bound approaches in constraint programming develop the enumeration tree in a Depth-First Search (DFS) manner:

1. evaluate a node: run propagation
2. if a failure occurs or if the search space cannot be separated then backtrack: evaluate the next pending node
3. otherwise branch: divide the search space and evaluate the first child node.

With Choco, the search process of the CPSolver does not currently allow to explore the tree in a different manner, using Best-First Search for example. In addition, the common way of dividing the search space in CP-based backtracking/B&B algorithms is to assign a variable to a value or to forbid this assignment. Choco provides such a branching strategy and the tools to easily customize the variable and value selection heuristics within this strategy. However, Choco makes possible to implement more complex branching strategies (e.g. constraint branching or dichotomy branching).

**Overriding the default search strategy**
2.4 CHOCO

**Branching, variable selection and value selection strategies.** Basically, a search strategy in Choco is a composition of branching strategy objects, each defined on a given set of decision variables. The most common branching strategies are based on the assignment of a selected variable to one or several selected values (one assignment in each branch).

**Default strategies.** Choco allows overriding the default search strategy, but when no search strategy is specified, default search strategies apply to all the decision variables of the solver. If the model has decision variables of different types, then these default branchings are evaluated in this order: first, the set decision variables are considered until they are all instantiated, then branching occurs on the pool of integer decision variables, and last on the pool of real decision variables.

**Decision variables.** Branchings apply to decision variables only. A branching can occur (i.e. the tree node can be separated according to this strategy) if and only if there exists a decision variable in its scope that is still not instantiated. The non-decision variables are also called implied variables because it is expected that, all variables – including these – will be instantiated (i.e. they will form a solution) by propagation as soon as all the decision variables will be instantiated.

**Pre-defined search strategies**

**Branching strategy** defines the way to branch from a tree search node. The branching strategies currently available in Choco are the following: AssignInterval, AssignOrForbidIntVarVal, etc.

**Variable selector** defines the way to choose a non-instantiated variable on which the next decision will be made. The variable selectors currently available in Choco are the following:

- CompositeIntVarSelector (implementing interface VarSelector<IntDomainVar>)
- MaxDomSet (implementing interface VarSelector<SetVar>)
- CyclicRealVarSelector (implementing interface VarSelector<RealVar>)

**Value iterator** Once the variable has been chosen, the solver has to compute its value. The first way to do it is to schedule all the values once and to give an iterator to the solver. (Example: DecreasingDomain, etc).

**Value selector** The second way to do it is to compute the next value at each call. (Example: MaxVal, etc).

**Why is it important to define a search strategy ?**

The search strategy should not be underestimated. A well-suited search strategy can reduce: the execution time, the number of expanded nodes, the number of backtracks.
2.4 CHOCO

Restarts

Restart means stopping the current tree search, then starting a new tree search from the root node. Restarting makes sense only when coupled with randomized dynamic branching strategies ensuring that the same enumeration tree is not constructed twice. The branching strategies based on the past experience of the search, such as DomOver-WDegBranching, are more accurate in combination with a restart approach.

Restarts can be set using the API available on the Solver. It performs a search with restarts controlled by the number of backtracks.

Limiting Search Space

The Solver class provides ways to limit the tree search controlled by different criteria. These limits have to be specified before the resolution. They are updated and checked each time a new node is created. Once a limit is reached, the search stops even if no solution is found.

- time limit: concerns the elapsed time from the beginning of the search.
- node limit: concerns the number of opened nodes.
- backtrack limit: concerns the number of performed backtracks.
- fail limit: concerns the number of contradiction encountered.

2.4.4 Advanced uses of Choco

How does the propagation engine work?

Once the Model and Solver have been defined, the resolution can start. It is based on decisions and filtering orders, this is the propagation engine. In this part, we’re going to present how the resolution is guided in Choco. A resolution instruction (solve(), solveAll(), maximize(...) or minimize(...)) always starts by setting options based on resolution policy, then generates the search strategies and ends by running the search loop.

How does a search loop work?

The search loop is the conductor of the engine. It goes down and up in the branches in order to cover the tree search, call the filtering algorithm, etc. Here is the organigram of the search loop (See figure 2.2).

Basically, the search loop is divided in 5 steps: INITIAL PROPAGATION (highlighted in red), OPEN NODE (highlighted in green), DOWN BRANCH (highlighted in violet), UP BRANCH (highlighted in orange) and RESTART (highlighted in blue).

Propagate

The main and unique PropagationEngine of Choco is ChocoEngine. This engine stores events occurring on variables, variable events, and specific calls to constraint filtering algorithm, constraint events, in order to reach a fix point or to detect contradictions. Events are stored in queues (FIFO). On a call to Solver:propagate() or during a resolution step, the consistency of a model is computed: stored events are popped and
2.4 CHOCO

Figure 2.2: Organigram of the search loop

propagated (apply side-effects). The propagation of a single event can create new ones, feeding the system until fix point or contradiction.

If the propagation of an event leads to a contradiction, the propagation engine stops the process. In both cases, the search loop takes up with the new state.

Seven priorities

It is important to know that events declare a parameter named priority. The priority of a constraint’s event depends on the constraint priority. And the priority of a variable’s event is given by the maximum priority of the variable’s constraints. To each priority corresponds a rank. In the propagation engine, there are as many queues as ranks. During the propagation loop, the rank is used to push the event on the corresponding queue.

There are seven priorities: UNARY, BINARY, TERNARY, LINEAR, QUADRATIC, CUBIC and VERY_SLOW.

By default, priorities and ranks are defined as follow:

<table>
<thead>
<tr>
<th>Priority</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary</td>
<td>1</td>
</tr>
<tr>
<td>binary</td>
<td>2</td>
</tr>
<tr>
<td>ternary</td>
<td>3</td>
</tr>
<tr>
<td>linear</td>
<td>4</td>
</tr>
<tr>
<td>quadratic</td>
<td>5</td>
</tr>
<tr>
<td>cubic</td>
<td>6</td>
</tr>
<tr>
<td>very slow</td>
<td>7</td>
</tr>
</tbody>
</table>
2.4 CHOCO

Constraint event

A call to the main filtering algorithm can be added during the resolution by posting a constraint event.

Variable event

The resolution goal is to instantiate variables in order to find solutions. Instantiation of a variable is done applying modification on its domain. Each time a modification is applied on a domain, an event is posted, storing informations about the action done. This event will be given to the related constraints of the modified variable, to check consistency and propagate this new information to the other variables.

An event given as a parameter to the engine is then pushed into a unique queue, waiting to be treated. There are seven different queues where an event can be pushed, it depends on the priority of the event.

2.4.5 Constraints (Model)

Choco has constraints, that we can use them. But we are going to name only one, like example:

allDifferent (constraint)

allDifferent(?x1, ...,xn?) states that the arguments have pairwise distinct values:

\( x_i \neq x_j, \forall i \neq j \)
2.4.6 Branching strategies (Solver)

A list of branching strategies\(^2\) are available, but we are going to name only one as example:

**AssignInterval (Branching strategy)**

AssignInterval is a binary branching assigning two distinct intervals to a real variable. Following the interval bisection rule, the interval representing the domain of the selected variable is split into two parts at its midpoint. In the first branch, the variable upper bound is set to the midpoint; in the second branch, the variable lower bound is set to the midpoint.

\[
\begin{align*}
B_1 : x & \in [x_1, m], \\
B_2 : x & \in [m, x_2],
\end{align*}
\]

with \(m = (x_1 + x_2)/2\)

2.4.7 Variable selectors (Solver)

Choco have variable selectors\(^4\) currently that we can use them. An example is:

**CompositeIntVarSelector (Variable selector)**

CompositeIntVarSelector(h1, h2) selects a constraint according to heuristic h1, then selects an integer variable involved in the constraint according to heuristic h2:

\[h_2(\text{support}(h1))\]

2.4.8 Value iterators (Solver)

Choco have value iterators\(^5\), that we can use them. An example is:

**DecreasingDomain (Value iterator)**

DecreasingDomain selects the integer variable largest value: \(\max(D(x))\)

2.4.9 Value selector (Solver)

Choco have value selectors\(^6\) that we can use them. An example is:

**MidVal (Value selector)**

---

\(^2\)heuristic controlling the execution of a search loop at a point where the control flow may be split between different branches.

\(^4\)heuristic specifying how to choose a variable at a fix point.

\(^5\)heuristic specifying how to choose a value from a chosen variable, through an iterator, at a fix point.

\(^6\)heuristic specifying how to choose a value from a chosen variable at a fix point.
2.4 CHOCO

MidVal selects the closest value (equal or greater) to the integer variable domain midpoint:

\[ \min(v \mid v \geq (x+x_1/2) \]

2.4.10 The magic square problem

We developed the magic square problem in Choco.

The program

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package chocoseminario;
import choco.Choco;
import choco.cp.model.CPModel;
import choco.kernel.model.variables.integer.IntegerVariable;
import choco.kernel.model.constraints.Constraint;
import choco.cp.solver.CPSolver;
import java.text.MessageFormat;

/**
 * @author noelia
 */
public class ChocoSeminario {

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    // TODO code application logic here
    // Constant declaration
    int n = 3; // Order of the magic square
    int magicSum = n * (n * n + 1) / 2; // Magic sum
    // Build the model
    CPModel m = new CPModel();

    // Creation of an array of variables
    IntegerVariable[][] var = new IntegerVariable[n][n];
    // For each variable, we define its name and the boundaries of its
    // domain
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            var[i][j] = Choco.makeIntVar("var_" + i + "_" + j, 1, n * n);
            // Associate the variable to the model.
        }
    }
}
```
m. addVariable(var[i][j]);
}

// All cells of the matrix must be different
for (int i = 0; i < n * n; i++) {
    for (int j = i + 1; j < n * n; j++) {
        Constraint c = (Choco.neq(var[i / n][i % n], var[j / n][j % n]));
        m. addConstraint(c);
    }
}

// System.out.print(3%3);
// All rows sum has to be equal to the magic sum
for (int i = 0; i < n; i++) {
    m. addConstraint(Choco.eq(Choco.sum(var[i]), magicSum));
}

IntegerVariable[][] varCol = new IntegerVariable[n][n];
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // Copy of var in the column order
        varCol[i][j] = var[j][i];
    }
    // All columns sum is equal to the magic sum
    m. addConstraint(Choco.eq(Choco.sum(varCol[i]), magicSum));
}

IntegerVariable[] varDiag1 = new IntegerVariable[n];
IntegerVariable[] varDiag2 = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    varDiag1[i] = var[i][i]; // Copy of var in varDiag1
    varDiag2[i] = var[(n - 1) - i][i]; // Copy of var in varDiag2
}
// All diagonals sum has to be equal to the magic sum
m. addConstraint(Choco.eq(Choco.sum(varDiag1), magicSum));
m. addConstraint(Choco.eq(Choco.sum(varDiag2), magicSum));

// Build the solver
CPSolver s = new CPSolver();

// Read the model
s.read(m);
// Solve the model
s.solve();
// Print the solution
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        System.out.print(MessageFormat.format("{0}", s.getVar(var[i][j]).getVal()));
    }
}
2.4 CHOCO

    System.out.println();
    }
    }
    }
    }
    }
2.6 Conclusions

2.4.11 The output

The output of the magic square problem is shown in the figure 2.4.

2.5 Works or projects related to CSP

There are projects related to CSP, but we will list some (see [5]):

- RAX-PS, EUROPA
- IXTeT
- ASPEN
- OMPS

2.6 Conclusions

We understood how CSP is used inside of a particular solver (Choco). Many solvers we can find. There are projects developed in CSP.


