SOLVING SOME OPTIMIZATION PROBLEMS USING choco LIBRARY

Lect. Radu Boriga, PhD candidate,

Titu Maiorescu University, Bucharest

choco *is a Java library for constraint satisfaction problems, constraint programming and explanation - based constraint solving. It is built on an event-based propagation mechanism with backtrackable structures which is optimized using backjumping and backmarking techniques. In this article we present how to model some optimization problems by constraint satisfaction problems and how to solve them using* choco.

1. CONSTRAINT SATISFACTION PROBLEM (CSP)

Although the works devoted to programming constraints have emerged since the 70s, the most clear definition of such programming was given by Eugene Freuder 1997: *" Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it."*

A *constraint* is a logical relationship between several variables of a problem, each variable tak ing values in a given domain. It can be noted that, in general, constraints may specify partial information, are non-directional, are declarative, are additive and are rarely independent.

From the properties listed above we can see that *modeling* a problem by using constraint programming lies in determining a finite number of variables with finite domeins and a finite set of constraints between them.

Finding a solution for this problem consists in determining some acceptable values for each variable so as not to violate any constraint. Depending on the nature of the problem, solving it can mean to identify either a single solution or all solutions, or to identify an optimal solution, if it has been defined an objective function, too.

Formally, a *Constraint Satisfaction Problem (CSP)* is defined ([1], [2]) by a triplet *(X,D,C)* such as:

1. **Variables:** $X = \{x_1, x_2, ..., x_n\}$ is the set of variables of the problem.
2. **Domain:** D is a function which associates to each variable x_i its domain $D(x_i)$, i.e., the possible values that can be assigned to $x_i (1 \le i \le n)$;
3. **Constraints:** $C = \{c_1, c_2, ..., c_n\}$ is the set of constraints. Each constraint c_j is a relation between a subset of variables which restricts the domain of each one $(1 \le j \le n)$.

We say that a *constraint is satisfied* if the tuple of the values of its variables belongs to the relation describing the constraint. Thus, solving a CSP consists in finding a tuple on the set of variables such that each constraint is satisfied.

2. ABOUT choco

choco is *an efficient constraint system* for research and development and a readable constraint system for teaching ([4], [6]). It was started in 1999 within the OCRE project, a French national initiative for an open constraint solver for both teaching and research involving researchers and practitionners from Nantes (Ecole des Mines), Montpellier (LIRMM), Toulouse (INRA and ONERA) and Bouygues SA. Its first implementation was in CLAIRE ([6]). In 2003, choco went through its premiere major modification when it has been implemented into the *Java* programming language. The objective was to ensure a greater portability and an easier takeover for newcomers.

In September 2008, when the second version was published, choco is being taken a step further: it offers a clear *separation between the model and the solving machinery* (providing both modelling tools and innovative solving tools), a complete refactoring improving its general performance, and a more userfriendly API for both newcomers and experienced CP practionne rs ([6],[5]).

As a *problem modeler* `choco` is able to manipulate a wide variety of variable types (*integer*, *set* and *real*) and to accept over 70 constraints:

- all classical *arithmetical constraints* (equal, not equal, less or equal, greater or equal, etc.);
- *boolean operations* between constraints;
- *table constraints* defining the sets of tuples that (do not) verify the intended relation for a set of variables;
- a large set of useful classical *global constraints* including the `alldifferent` constraint, the `global cardinality` constraint, the `nvalue` constraint, the `element` constraint, the `cumulative` constraint;
- most recent implementations of global constraints, including the `tree` constraint and the `geost` constraint.

As a *solver* `choco` provides several implementations of the various *domain types* (enumerated, bounded, list-based and integer variables) and several algorithms for *constraint propagation* (algorithms for table constraints, full and bound alldifferent, parameterized cumulative, etc.). It can either be used in *satisfaction mode* (computing one solution, all solutions or iterating them) or in *optimization mode* (maximisation and minimisation). Search can be parameterized using a set of predefined variable and value selection heuristics.

Finally, when converting the model into a solver-specific problem, `choco` can enter into a pre-processor mode that will perform some automatic improvements in the model.

3. THE KNAPSACK PROBLEM

3.1 The Binary Knapsack Problem

Firstly, we review the terms of the *Binary Knapsack Problem*: "Considering a set of n items, for each item we have associated a profit p_j and a weight w_j $(1 \le j \le n)$. The objective is to pick some of the items, with maximal total profit, while obeying that the maximum total weight of the chosen items must not exceed the weight W which can be loaded in a knapsack. Moreover, any item can be fully charged or not at all." Generally, the coefficients are scaled to become integers, and they are almost always assumed to be positive.

Starting with the solution described in ([3]), it's easy to model this problem by the next *CSP*:

1. **Variables:** $X = \{x_1, x_2, ..., x_n\}$
2. **Domain:** $D(x_i) = \{0,1\}, \forall i \in \{1,2,...,n\}$
3. **Constraints:** $\sum_{k=1}^{n} x_k w_k \le W$
4. **Goal:** to maximize $\sum_{k=1}^{n} x_k p_k$

Based on the *CSP* above mentioned, we can model and solve the problem in a few steps using choco:

1. **creating a new model:**

```
Model m = new CPModel();
```

2. **creating the variables:**

```
IntegerVariable[] x = makeIntVarArray("x", n, 0, 1, "cp:enum");
IntegerVariable c = makeIntVar("c", 1, 1000000, "cp:binary");
```

3. **creating the constraints:**

```
m.addConstraint(leq(scalar(w, x), W));
m.addConstraint(eq(scalar(p, x), c));
```

4. **creating a new solver and loading the model into it:**

```
Solver s = new CPSolver();
s.read(m);
```

5. **solving the problem:**

```
s.maximize(s.getVar(c), false);
```

6. **printing all the solution:**

```
System.out.println("Maximum benefit: " + s.getVar(c).getVal());
  for (int i = 0; i < n; i++)
   System.out.println("Object " + (i + 1) + ": " +
s.getVar(x[i]).getVal());
```

3.2 The Bounded Knapsack Problem

The *Bounded Knapsack Problem* specifies for each item j $(1 \le j \le n)$, additionally, an upper bound u_j (which may be a positive integer) on the number of times item j can be selected.

It's easy to see that the only a minor difference appears when we are creating the variable s $x_1,\ldots,x_n$:

```
IntegerVariable[] x = makeIntVarArray("x", n);
    for(int i=0;i<n;i++)
  x[i]=makeIntVar("x"+i,0,u[i],"cp:enum");
```

3.3 The Unbounded Knapsack Problem

In the *Unbounded Knapsack Problem* (sometimes called the *Integer Knapsack Problem*) we does not put any upper bounds on the number of times an item may be selected . Anyway, it's clear that each item j can be used at most $[W/w_j]$ times $(1 \le j \le n)$.

It's easy to see that, again, the only a minor difference appears when we are creating the v ariables $x_1,\ldots,x_n$:

```
IntegerVariable[] x = makeIntVarArray("x", n);
    for(int i=0;i<n;i++)
  x[i]=makeIntVar("x"+i,0,W/w[i],"cp:enum");
```

4. A PAYMENT METHOD OF AN AMOUNT USING A MINIMUM NUMBER OF COINS

Let's assume that we have an amount S and n types of coins with values $v_1, v_2,\ldots,v_n$. We want to pay the amount S using a minimum number of coins. As the Knapsack Problem, this problem has a binary, a bounded or an unbouded version. T o avoid unnecessary exposure charge, we will consider the unbounded version.

Starting with the solution described in ([3]), it's easy to model this problem by the next *CSP*:

1. **Variables:** $X = \{x_1, x_2,\ldots, x_n\}$
2. **Domain:** $D(x_i) = \{0,[S / v_i]\}, \forall i \in \{1,2,\ldots, n\}$
3. **Constraints:** $\sum_{k=1}^{n} x_k v_k = S$
4. **Goal:** to minimize $\sum_{k=1}^{n} x_k$

Based on the *CSP* above mentioned, we can model and solve the problem in a few steps using choco:

1. **creating a new model:**

```
Model m = new CPModel();
```

2. **creating the variables:**

```
IntegerVariable[] x = makeIntVarArray("x", n);
for(int i=0;i<n;i++)
  x[i]=makeIntVar("x"+i,0,S/v[i],"cp:enum");
IntegerVariable nc = makeIntVar("c", 1, 1000000, "cp:bound");
```

3. **creating the constraints:**

```
m.addConstraint(eq(scalar(v, x), S));
m.addConstraint(eq(sum(x), nc));
```

4. **creating a new solver and loading the model into it:**

```
Solver s = new CPSolver();
s.read(m);
```

5. **solving the problem:**

```
s.minimize(s.getVar(c), false);
```

6. **printing all the solution:**

```
System.out.println("Minimum number of coins: " + s.getVar(nc).getVal());
 for (int i = 0; i < n; i++)
    System.out.println("Coin " + (i + 1) + ":" + s.getVar(x[i]).getVal());
```

7. CONCLUSIONS

Using constraint satisfaction problems for solving some optimization problems, or, moreover, for solving problems which requires exhaustive searches, has multiple advantages:

- the programmer doesn't need to implement an exhausitve search algorithm;
- the source code becomes smaller and more readable;
- the optimization is made automatically, due to the implicit using of backjumping and backmarking algorithms, but it can be defined by the programmer, too.

8. REFERENCES

[1] Bartak, R. - *On-line guide to constraint programming*, http://kti.mff.cuni.cz/bartak/constraints/index.html, 1998
[2] Dechte, R. - *Constraint Processing*, Morgan Kaufmann, 2003
[3] Georgescu, H. - *Tehnici de programare*, Editura Universităţii din Bucureşti, 2005
[4] Laburthe, F. and Jussein, H. – choco's *website*, http://choco.emn.fr, 2006.
[5] Ryser, H. - *Combinatorial Mathematics*, Math. Assoc. Amer., Buffalo, NY, 1963.
[6] The choco team - choco*: an Open Source Java Constraint Programmin g Library*, http://choco.emn.fr, 2008.