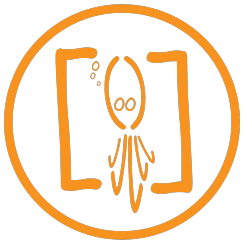
Choco solver
http://choco.emn.fr/

Tutorials



July 15, 2010

Contents

Preface

Choco is a java library for constraint satisfaction problems (CSP) and constraint programming (CP). It is built on a event-based propagation mechanism with backtrackable structures. Choco is an open-source software, distributed under a **BSD licence** and hosted by sourceforge.net. For any informations visit `http://choco.emn.fr`.

This document is organized as follows:

- Tutorials provides a fast how-to write a Choco program, a detailed example of a simple program, and several exercises with their solutions, several guided examples and some examples of Applications.

Part I

Tutorials

If you look for an easy step-by-step program in CHOCO, getting started is for you! It introduces to basic concepts of a CHOCO program (Model, Solver, variables and constraints...).

This part also presents a collection of exercises with their solutions and guided examples. It covers simple to advanced uses of CHOCO.

See also old pages: http://choco.sourceforge.net/tut_expl.html

Chapter 1

Getting started: welcome to Choco

This introduction covers the basics of writing a program in Choco

Choco is a java library for constraint satisfaction problems (CSP), constraint programming (CP) and explanation-based constraint solving (e-CP). It is built on a event-based propagation mechanism with backtrackable structures.

1.1 Before starting

Before doing anything, you have to be sure that

- you have at least Java6 installed on your environment.
- you have a IDE (like IntelliJ IDEA or Eclipse).

To install Java6 or your IDE, please refer to its specific documentation. We now assume that you have the previously defined environment.

You need to create a **New Project...** on your favorite IDE (create a new project on IntelliJ, create a new project on Eclipse). Our project name is *ChocoProgram*. Create a new class, named *MyFirstChocoProgram*, with a main method.

```
public class MyFirstChocoProgram {

    public static void main(String[] args) {

    }
}
```

1.2 Download Choco

Now, before doing anything else, you need to download the last stable version of Choco. See the download page. Once you have download choco, you need to add it to the classpath of your project.

Now you are ready to create you first Choco program.

If you want a short introduction on what is constraint programming, you can find some informations in the Documentation of choco.

When you feel ready, solve your own problem! And if you need more tries, please take a look at the exercises.

Chapter 2

First Example: Magic square

A simple magic square of order 3 can be seen as the "Hello world!" program in Choco. First of all, we need to agree on the definition of a magic square of order 3. Wikipedia tells us that :

> A **magic square** 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 normal magic square contains the integers from 1 to n^2.

So we are going to solve a problem where unknows are cells value, knowing that each cell can take its value between 1 and n^2, is different from the others and columns, diagonals and rows are equal to the same constant M (which is equal to $n * (n^2 + 1)/2$).

We have the definition, let see how to add some Choco in it.

2.1 First, the model

To define our problem, we need to create a Model object. As we want to solve our problem with constraint programming (of course, we do), we need to create a CPModel.

```
//constants of the problem:
int n = 3;
int M = n*(n*n+1)/2;

// Our model
Model m = new CPModel();
```

These objects require to import the following classes:

```
import choco.cp.model.CPModel;
import choco.kernel.model.Model;
```

At the begining, our model is empty, no problem has been defined explicitly. A model is composed of variables and constraints, and constraints link variables to each others.

- **Variables** A variable is an object defined by a name, a type and a domain. We know that our unknowns are cells of the magic square. So: `IntegerVariable cell = Choco.makeIntVar("aCell", 1, n*n);` which means that *aCell* is an integer variable, and its domain is defined from *1* to *n*n*. But we need n^2 variables, so the easiest way to define them is:

```
IntegerVariable[][] cells = new IntegerVariable[n][n];
for(int i = 0; i < n; i++){
   for(int j = 0; j < n; j++){
      cells[i][j] = Choco.makeIntVar("cell"+j, 1, n*n);
      m.addVariables(cells[i][j]);
```

```
    }
  }
```

This code requires to import the following classes:

```
import choco.kernel.model.variables.integer.IntegerVariable;
import choco.Choco;
```

We add each variables to our model: `m.addVariables(cells[i][j]);`
Now that our variables are defined, we have to define the constraints between variables.

- **Constraints over the rows** The sum of ach rows is equal to a constant M. So we need a sum operator and and equality constraint. The both are provides by the `Choco.java` class.

```
//Constraints
// ... over rows
Constraint[] rows = new Constraint[n];
for(int i = 0; i < n; i++){
   rows[i] = Choco.eq(Choco.sum(cells[i]), M);
}
```

This part of code requires the following import:

```
import choco.kernel.model.constraints.Constraint;
```

After the creation of the constraints, we need to add them to the model:

```
m.addConstraints(rows);
```

- **Constraints over the columns** Now, we need to declare the equality between the sum of each column and M. But, the way we have declare our variables matrix does not allow us to deal easily with it in the column case. So we create the transposed matrix (a 90^o rotation of the matrix) of *cells*.

> We do not introduce new variables. We just reorder the matrix to see the *column point of view*.

```
//... over columns
// first, get the columns, with a temporary array
IntegerVariable[][] cellsDual = new IntegerVariable[n][n];
for(int i = 0; i < n; i++){
   for(int j = 0; j < n; j++){
      cellsDual[i][j] = cells[j][i];
   }
}
```

Now, we can declare the constraints as before:

```
Constraint[] cols = new Constraint[n];
for(int i = 0; i < n; i++){
   cols[i] = Choco.eq(Choco.sum(cellsDual[i]), M);
}
```

And we add them to the model:

```
m.addConstraints(cols);
```

- **Constraints over the diagonals** Now, we get the two diagonals array *diags*, reordering the required *cells* variables, like in the previous step.

```
//... over diagonals
IntegerVariable[][] diags = new IntegerVariable[2][n];
for(int i = 0; i < n; i++){
    diags[0][i] = cells[i][i];
    diags[1][i] = cells[i][(n-1)-i];
}
```

And we add the constraints to the model (in one step this time).

```
m.addConstraint(Choco.eq(Choco.sum(diags[0]), M));
m.addConstraint(Choco.eq(Choco.sum(diags[1]), M));
```

- **Constraints of variables AllDifferent** Finally, we add the AllDifferent constraints, stating that each *cells* variables takes a unique value. One more time, we have to reorder the variables, introducing temporary array.

```
//All cells are differents from each other
IntegerVariable[] allVars = new IntegerVariable[n*n];
for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
        allVars[i*n+j] = cells[i][j];
    }
}
m.addConstraint(Choco.allDifferent(allVars));
```

2.2 Then, the solver

Our model is established, it does not require any other information, we can focus on the way to solve it. The first step is to create a Solver;

```
//Our solver
Solver s = new CPSolver();
```

This part requires the following imports:

```
import choco.kernel.solver.Solver;
import choco.cp.solver.CPSolver;
```

After that, the model and the solver have to be linked, thus the solver *read* the model, to extract informations:

```
//read the model
s.read(m);
```

Once it is done, we just need to solve it:

```
//solve the problem
s.solve();
```

And print the information

```
//Print the values
for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
        System.out.print(s.getVar(cells[i][j]).getVal()+" ");
    }
    System.out.println();
}
```

2.3 Conclusion

We have seen, in a few steps, how to solve a basic problem using constraint programming and Choco. Now, you are ready to solve your own problem, and if you need more tries, please take a look at the exercises. You can download the java file of the introduction: myfirstchocoprogram.zip

Chapter 3

Exercises

3.1 I'm new to CP

The goal of this practical work is twofold :

- **problem modelling** with the help of variables and constraints ;
- **mastering the syntax** of Choco in order to tackle basic problems.

Warning, constraint modelling should not be solver specific. That is why you are strongly advised to write down your model before starting its implementation within Choco.

3.1.1 Exercise 1.1 (A soft start)

Algorithm 1 (below) describes a problem which fits the minimum choco syntax requirements.

Question 1 describe the constraint network modelled in Algorithm 1.

Question 2 give the variable domains after constraint propagation.

Algorithm 1 Mysterious model.

```
// Build a model
Model m = new CPModel() ;

// Build enumerated domain variables
IntegerVariable x1 = makeIntVar("var1", 0, 5);
IntegerVariable x2 = makeIntVar("var2", 0, 5);
IntegerVariable x3 = makeIntVar("var3", 0, 5);

// Build the constraints
Constraint C1 = gt(x1, x2) ;
Constraint C2 = neq(x1, x3) ;
Constraint C3 = gt(x2, x3) ;

// Add the constraints to the Choco model
m.addConstraint(C1) ;
m.addConstraint(C2) ;
m.addConstraint(C3) ;

// Build a solver
Solver s = new CPSolver();
```

```
// Read the model
s.read(m);

// Solve the problem
s.solve() ;

// Print the variable domains
System.out.println("var1␣=" + s.getVar(x1) .getVal()) ;
System.out.println("var2␣=" + s.getVar(x2) .getVal()) ;
System.out.println("var3␣=" + s.getVar(x3) .getVal()) ;
```

(Solution)

3.1.2 Exercise 1.2 (DONALD + GERALD = ROBERT)

Associate a different digit to every letter so that the equation DONALD + GERALD = ROBERT is verified.

(Solution)

3.1.3 Exercise 1.3 (A famous example. . . a sudoku grid)

A sudoku grid is a square composed of nine squares called *blocks*. Each block is itself composed of 3x3 cells (see figure 1). The purpose of the game is to fill the grid so that each block, column and row contains all the numbers from 1 to 9 once and only once

Question 1 propose a way to model the sudoku problem with difference constraints. Implement your model with Choco.

Question 2 which global constraint can be used to model such a problem ? Modify your code accordingly.

Question 3 Test, for both models, the initial propagation step (use Choco `propagate()` method). What can be noticed ? What is the point in using global constraints ?

		7	5			3		
	4			2		1		
1				7			5	
		3	1	4		2		6
4				6	2	7		
	6	5		3				8
	7	1				6		
8								
	5		7				4	1

Figure 3.1: An exemple of a Sudoku grid

(Solution)

3.1.4 Exercise 1.4 (The knapsack problem)

Let us organise a trek. Each hiker carries a knapsack of capacity 34 and can store 3 kinds of food which respectively supply energetic values (6,4,2) for a consumed capacity of (7,5,3). The problem is to find which food is to be put in the knapsack so that the energetic value is maximal.

Question 1 In the first place, we will not consider the idea of maximizing the energetic value. Try to find a satisfying solution by modelling and implementing the problem within choco.

Question 2 Find and use the choco method to **maximise** the energetic value of the knapsack.

Question 3 Propose a Value selector heuristic to improve the efficiency of the model.

(Solution)

3.1.5 Exercise 1.5 (The n-queens problem)

The n-queens problem aims to place n queens on a chessboard of size n so that no queen can attack one another.

Question 1 propose and implement a model based on one L_i variable for every row. The value of L_i indicates the column where a queen is to be put. Use simple difference constraints and confirm that 92 solutions are obtained for $n = 8$.

Question 2 Add a redundant model by considering variables on the columns (C_i). Continue to use simple difference constraints.

Question 3 Compare the number of nodes created to find the solutions with both models. How can you explain such a difference ?

Question 4 Add to the previously implemented model the following heuristics:

- Select first the line variable L_I which has the smallest domain ;
- Select the value $j \in L_i$ so that the associated column variable C_j has the smallest domain.

Again, compare both approaches in term of number of nodes and solving time to find ONE solution for $n = 75, 90, 95, 105$.

Question 5 what changes are caused by the use of the global constraint `alldifferent` ?

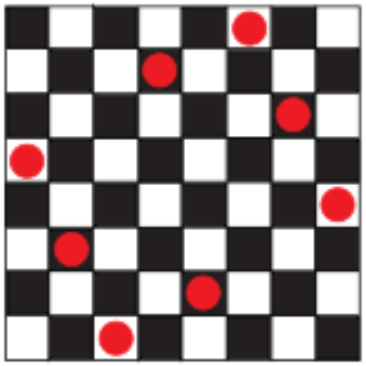


Figure 3.2: A solution of the n-queens problem for $n = 8$

(Solution)

3.2 I know CP

3.2.1 Exercise 2.1 (Bin packing, cumulative and search strategies)

Can n objects of a given size fit in m bins of capacity C ? The problem is here stated has a satisfaction problem for the sake of simplicity. Your model and heuristics will be checked by generating random instances for given n and C. The random generation must be reproducible.

Question 1 Propose a boolean model (0/1 variables).

Question 2 Let us turn this satisfaction problem into an optimization one. Use your previously stated model but increase regularly the number of containers until a feasible solution is found.

Question 3 Implement a naive lower bound. This can be done by considering the occupied size globally.

Question 4 Propose a model with integer variables based on the cumulative constraint (see choco user guide/API for details). Define an objective function to minimize the number of used bins.

Bonus question Compare different search strategies (variables/values selector) on this model for n between 10 and 15.

Take a look at the following exercise in the old version of Choco and try to transpose it on new version of Choco.

Here is **the complete code in Choco1** : BinPackingv1.zip.

```
int[] instance = getRandomPackingPb(n, capaBin, seed);
QuickSort sort = new QuickSort(instance); //Sort objects in increasing order
sort.sort();
Problem pb = new Problem();
IntDomainVar[] debut = new IntDomainVar[n];
IntDomainVar[] duree = new IntDomainVar[n];
IntDomainVar[] fin = new IntDomainVar[n];

int nbBinMin = computeLB(instance, capaBin);
for (int i = 0; i < n; i++) {
    debut[i] = pb.makeEnumIntVar("debut " + i, 0, n);
    duree[i] = pb.makeEnumIntVar("duree " + i, 1, 1);
    fin[i] = pb.makeEnumIntVar("fin " + i, 0, n);
}
IntDomainVar obj = pb.makeEnumIntVar("nbBin ", nbBinMin, n);
pb.post(pb.cumulative(debut, fin, duree, instance, capaBin));
for (int i = 0; i < n; i++) {
    pb.post(pb.geq(obj, debut[i]));
}

IntDomainVar[] branchvars = new IntDomainVar[n + 1];
System.arraycopy(debut, 0, branchvars, 0, n);
branchvars[n] = obj;

//long tps = System.currentTimeMillis();
pb.getSolver().setVarSelector(new StaticVarOrder(branchvars));
Solver.setVerbosity(Solver.SOLUTION);
pb.minimize(obj, false);
Solver.flushLogs();
// print solution
System.out.println("------------------------ " + (obj.getVal() + 1) + " bins");
if (pb.isFeasible() == Boolean.TRUE) {
    for (int j = 0; j <= obj.getVal(); j++) {
    System.out.print("Bin " + j + ": ");
    int load = 0;
    for (int i = 0; i < n; i++) {
        if (debut[i].isInstantiatedTo(j)) {
```

```
            System.out.print(i + "␣");
            load += instance[i];
            }
        }
        System.out.println("␣-␣load␣" + load);
        }
        //System.out.println("tps " + tps + " node "
         // + ((NodeLimit) pb.getSolver().getSearchSolver().limits.get(1)).getNbTot());
    }
```

(Solution)

3.2.2 Exercise 2.2 (Social golfer)

A group of golfers play once a week and are splitted into k groups of size s (there are therefore ks golfers in the club). The objective is to build a game scheduling on w weeks so that no golfer play in the same group than another one more than once (hence the name of the problem: *social golfers*). However, it may happen that two golfers will never play together. The point is only that once they have played together, they cannot play together anymore.

> You can test your model with the parameters (w, s, g) set to: $\{(11, 6, 2), (13, 7, 2), (9, 8, 8), (9, 8, 4), (4, 7, 3), (3, 6, 4)\}$.

Question 1 Propose a boolean model for this problem. Use an heuristic that consists in scheduling a golfer on every week before scheduling a new one. More precisely, a golfer can be put in the first available group of each week before considering the next golfer.

Question 2 Identify some symmetries of the problem by using every similar elements of the problem. Try to improve your model by breaking those symmetries.

	group 1	group 2	group 3	group 4
week 1	1 2 3	4 5 6	7 8 9	10 11 12
week 1	1 4 7	10 2 5	8 11 3	6 9 12
week 1	1 5 9	10 2 6	7 11 3	4 8 12

Table 3.1: A valid configuration with 4 groups of 3 golfers on 3 weeks.

Here is **the complete code in Choco1** : SocialGolferv1.zip

```
Problem pb = new Problem();
int numplayers = g * s;

// golfmat[i][j][k] : is golfer k playing week j in group i ?
IntDomainVar[][][] golfmat = new IntDomainVar[g][w][numplayers];
for (int i = 0; i < g; i++) {
    for (int j = 0; j < w; j++)
         for (int k = 0; k < numplayers; k++)
            golfmat[i][j][k] = pb.makeEnumIntVar("("+i+"_"+j+"_"+k+")", 0, 1);
}

//every week, every golfer plays in one group
for (int i = 0; i < w; i++) {
    for (int j = 0; j < numplayers; j++) {
        IntDomainVar[] vars = new IntDomainVar[g];
        for (int k = 0; k < g; k++) {
            vars[k] = golfmat[k][i][j];
        }
```

```
        pb.post(pb.eq(pb.scalar(vars, getOneMatrix(g)), 1));
    }
}

//every group is of size s
for (int i = 0; i < w; i++) {
    for (int j = 0; j < g; j++) {
        IntDomainVar[] vars = new IntDomainVar[numplayers];
        System.arraycopy(golfmat[j][i], 0, vars, 0, numplayers);
        pb.post(pb.eq(pb.scalar(vars, getOneMatrix(numplayers)), s));
    }
}

//every pair of players only meets once
// Efficient way : use of a ScalarAtMost
for (int i = 0; i < numplayers; i++) {
    for (int j = i + 1; j < numplayers; j++) {
        IntDomainVar[] vars = new IntDomainVar[w * g * 2];
        int cpt = 0;
        for (int k = 0; k < w; k++) {
            for (int l = 0; l < g; l++) {
                vars[cpt] = golfmat[l][k][i];
                vars[cpt + w * g] = golfmat[l][k][j];
                cpt++;
            }
        }
        pb.post(new ScalarAtMostv1(vars, w * g, 1));
    }
}

//break symetries among weeks
//enforce a lexicographic ordering between every pairs of week
for (int i = 0; i < w; i++) {
    for (int j = i + 1; j < w; j++) {
        IntDomainVar[] vars1 = new IntDomainVar[numplayers * g];
        IntDomainVar[] vars2 = new IntDomainVar[numplayers * g];
        int cpt = 0;
        for (int k = 0; k < numplayers; k++) {
            for (int l = 0; l < g; l++) {
                vars1[cpt] = golfmat[l][i][k];
                vars2[cpt] = golfmat[l][j][k];
                cpt++;
            }
        }
        pb.post(pb.lex(vars1, vars2));
    }
}

//break symetries among groups
for (int i = 0; i < numplayers; i++) {
    for (int j = i + 1; j < numplayers; j++) {
        IntDomainVar[] vars1 = new IntDomainVar[w * g];
        IntDomainVar[] vars2 = new IntDomainVar[w * g];
        int cpt = 0;
        for (int k = 0; k < w; k++) {
            for (int l = 0; l < g; l++) {
                vars1[cpt] = golfmat[l][k][i];
                vars2[cpt] = golfmat[l][k][j];
                cpt++;
            }
        }
```

```
            pb.post(pb.lex(vars1, vars2));
        }
    }

    //break symetries among players
    for (int i = 0; i < w; i++) {
        for (int j = 0; j < g; j++) {
            for (int p = j + 1; p < g; p++) {
                IntDomainVar[] vars1 = new IntDomainVar[numplayers];
                IntDomainVar[] vars2 = new IntDomainVar[numplayers];
                int cpt = 0;
                for (int k = 0; k < numplayers; k++) {
                    vars1[cpt] = golfmat[j][i][k];
                    vars2[cpt] = golfmat[p][i][k];
                    cpt++;
                }
                pb.post(pb.lex(vars1, vars2));
            }
        }
    }

    //gather branching variables
    IntDomainVar[] staticvars = new IntDomainVar[g * w * numplayers];
    int cpt = 0;
    for (int i = 0; i < numplayers; i++) {
        for (int j = 0; j < w; j++) {
            for (int k = 0; k < g; k++) {
                staticvars[cpt] = golfmat[k][j][i];
                cpt++;
            }
        }
    }
    pb.getSolver().setVarSelector(new StaticVarOrder(staticvars));

    pb.getSolver().setTimeLimit(120000);
    Solver.setVerbosity(Solver.SOLUTION);
    pb.solve();
    Solver.flushLogs();
```

(Solution)

3.2.3 Exercise 2.3 (Golomb rule)

under development

(Solution)

3.3 I know CP and Choco

3.3.1 Exercise 3.1 (Hamiltonian Cycle Problem Traveling Salesman Problem)

Given a graph $G = (V, E)$, an *Hamiltonian cycle* is a cycle that goes through every nodes of G once and only once. This exercise first intorduces a naive model to solve the Hamiltonian Cycle Problem. A second part tackles with the well known Traveling Salesman Problem.

Let $V = \{2, ..., n\}$ be a set of cities index to cover, and let d be a single warehouse duplicated into two indices 1 and $n+1$. Notice the duplication distinguishes the source from the sink while there is only one warehouse. Finally, let us denote by $V_d = V \cup \{1, n+1\}$ the set of nodes to cover by a tour. Thus, the two following problems are defined:

- find an Hamiltonian path covering all the cities of V

- find an Hamiltonian cycle of minimum cost that covers all the cities of V.

Question 1 [Hamiltonian Cycle Problem]:

We first consider the satisfaction problem. Formally, a directed graph $G = (V, E)$ represents the topology of the cities and the unfolded warehouse. There is an arc $(i, j) \in E$ iff there exists a directed road from $i \in V$ to $j \in V$. Furthermore, every arc $(1, i)$ and $(i, n + 1)$, with $i \in V$, belongs to E. Such a problem has to respect the following constraints:

- each node of V_d is reached exactly once,
- there is no subcycle containing nodes of V. In other words, the single cycle involved in G is hamiltonian and contains arc $(n + 1, 1)$.

Question 1.a The first constraint can directly be modelled using those proposed by Choco. On the other way, the second one requires to implement a constraint. This can be done through the following steps (see the provided skeleton):

- strictly specify your constraint signature,
- formalise the underlying subproblemand information that need to be maintained,
- ignore in a first time the Choco event based mechanism and implement your filtering algorithm directly within the `propagate()` method,
- once your algorithm has been checked, try to reformulate your constraint through an event based implementation with the following methods: `awakeOnInst()`, `awakeOnSup()`, `awakeOnInf()`, `awakeOnBounds()`, `awakeOnRem()`, `awakeOnRemovals()`.

Question 1.b Now, propose a search heuristic (both on variables and values) that incrementaly builds the searched path from the source node. For this purpose, you have to respectively implement java classes that inherit from `IntVarSelector` and `ValSelector`.

Question 2 [Traveling Salesman Problem]:

We now consider the optimisation view of the Hamiltonian Cycle Problem. A quantitative information is now associated with each arc of G given by a cost function $f : E \leftarrow \mathbb{Z}_+$. Then, the graph G is now defined by the triplet (V_d, E, f) and we have to find an Hamiltonian path of minimum cost in G.

For this purpose, we provide a skeleton of a Choco global constraint that dynamically maintains a lower bound evaluation of the searched path cost. Here, an evaluation of a minimum spanning tree of G is proposed. *Be careful*: take into account the partial assignment of the variables associated with the cities.

Question 2.a find an upper bound on the cost of the Hamiltonian path,

Question 2.b back-propagate lower/upper bounds informations on the required/infeasible arcs of G.

(Solution)

3.3.2 Exercise 3.2 (Shop scheduling)

Given a set of n tasks T and m disjunctive resources R, the problem is to find a plan to assign tasks to resources so that for every instant t, each resource $r \in R$ executes at most one task. Each task $T_i \in T$ is defined by:

- a starting date $s_i = [s_i^-, s_i^+] \in \mathbb{Z}_+$,
- an ending date $e_i = [e_i^-, e_i^+] \in \mathbb{Z}_+$,
- a duration $d_i = e_i - s_i$,
- a resource $r_i = \{res_1, \ldots, res_m\} \subseteq R$,

- a set of tasks, $preds_i \subseteq T$ that need to be processed before the start of T_i.

Let us consider the following satisfaction problem : Can one find a schedule of tasks T on the resources R that

- satisfies all the precedence constraints:

$$e_j \leq s_i, \qquad (\forall T_i \in T, \forall T_j \in preds_i)$$

- the last processed task ends before a given date D ?:

$$e_i \leq D, \qquad (\forall T_i \in T)$$

Then consider the optimization version : We now aim at finding the scheduling that satisfies all the constraints and that minimizes the date of the last task processed. You will be given for this :

- a class structure where you have to describe your model (`AssignmentProblem`),
- a class structure describing a Task (`Task`),
- a class structure (`BinaryNonOverlapping`) which defines a Choco constraint. This constraint takes two tasks as parameters and has to verify whether at any time t those tasks will be processed by the same resource or not.
- a class structure (`MandatoryInterval`) which describes for a given task, the time window it has to be processed in.

Question 1 How would you model the job scheduling problem ? Make use of the constraint `BinaryNonOverlapping`.

Question 2 Implement your model as if the constraint `BinaryNonOverlapping` was implemented.

Question 3 Sketch the mandatory processing interval of a task.

Question 4 Implement the constraint `BinaryNonOverlapping`:

- Implement the following reasoning : if two tasks have to be processed on the same resource and their mandatory intervals intersect, throw a failure.
- Now, implement the condition : if two tasks have a mandatory interval intersection, they must be scheduled on different resources.
- Finally, implement the following reasoning : If two tasks have to be processed by the same resource, then the starting and ending dates of every task ought to be updated functions to their mandatory intervals.

Question 5 Implement an variable selection heuristic on the decision variable of the problem.

Question 6 Propose a model which minimize the end date of the last assigned task.

Question 7 Can you find a way to improve the BinaryNonOverlapping constraint.

Bonus Question Find a lower bound on the end date of the last processed task.

Solution

Chapter 4

Solutions

4.1 I'm new to CP

4.1.1 Solution of Exercise 1.1 (A soft start)

(Problem)

***Question 1**: describe the constraint network related to code*

The model is defined as :

- $V = \{x_1, x_2, x_3\}$: the set of variables,
- $D = \{[0,5], [0,5], [0,5]\}$: the set of domain
- $C = \{x_1 > x_2, x_1 \neq x_3, x_2 > x_3\}$: the set of constraints.

***Question 2**: give the variable domains after constraint propagation.*

- From $x_1 = [0,5]$ and $x_2 = [0,5]$ and $x_1 > x_2$, we can deduce tha : the domain of x_1 can be reduce to $[1,5]$ and the domain of x_2 can be reduce to $[0,4]$.
- Then, from $x_2 = [0,4]$ and $x_3 = [0,5]$ and $x_2 > x_3$, we can deduce that : the domain of x_2 can be reduce to $[1,4]$ and the domain of x_3 can be reduce to $[0,3]$.
- Then, from $x_1 = [1,5]$ and $x_2 = [1,4]$ and $x_1 > x_2$, we can deduce that : the domain of x_1 can be reduce to $[2,5]$.

We cannot deduce anything else, so we have reached a **fix point**, and here is the domain of each variables:

$$x_1 : [2,5], \quad x_2 : [1,4], \quad x_3 : [0,3].$$

4.1.2 Solution of Exercise 1.2 (DONALD + GERALD = ROBERT)

(Problem)

Source code: ExDonaldGeraldRobert.zip

```
// Build model
Model model = new CPModel();

// Declare every letter as a variable
IntegerVariable d = makeIntVar("d", 0, 9, Options.V_ENUM);
IntegerVariable o = makeIntVar("o", 0, 9, Options.V_ENUM);
IntegerVariable n = makeIntVar("n", 0, 9, Options.V_ENUM);
IntegerVariable a = makeIntVar("a", 0, 9, Options.V_ENUM);
IntegerVariable l = makeIntVar("l", 0, 9, Options.V_ENUM);
IntegerVariable g = makeIntVar("g", 0, 9, Options.V_ENUM);
IntegerVariable e = makeIntVar("e", 0, 9, Options.V_ENUM);
```

```
IntegerVariable r = makeIntVar("r", 0, 9, Options.V_ENUM);
IntegerVariable b = makeIntVar("b", 0, 9, Options.V_ENUM);
IntegerVariable t = makeIntVar("t", 0, 9, Options.V_ENUM);

// Declare every name as a variable
IntegerVariable donald = makeIntVar("donald", 0, 1000000,Options.V_BOUND);
IntegerVariable gerald = makeIntVar("gerald", 0, 1000000,Options.V_BOUND);
IntegerVariable robert = makeIntVar("robert", 0, 1000000,Options.V_BOUND);

// Array of coefficients
int[] c = new int[]{100000, 10000, 1000, 100, 10, 1};

// Declare every combination of letter as an integer expression
IntegerExpressionVariable donaldLetters = scalar(new IntegerVariable[]{d,o,n,a,l,d}, c);
IntegerExpressionVariable geraldLetters = scalar(new IntegerVariable[]{g,e,r,a,l,d}, c);
IntegerExpressionVariable robertLetters = scalar(new IntegerVariable[]{r,o,b,e,r,t}, c);

// Add equality between name and letters combination
model.addConstraint(eq(donaldLetters, donald));
model.addConstraint(eq(geraldLetters, gerald));
model.addConstraint(eq(robertLetters, robert));
// Add constraint name sum
model.addConstraint(eq(plus(donald, gerald), robert));
// Add constraint of all different letters.
model.addConstraint(allDifferent(new IntegerVariable[]{d,o,n,a,l,g,e,r,b,t}));

// Build a solver, read the model and solve it
Solver s = new CPSolver();
s.read(model);
s.solve();

// Print name value
System.out.println("donald␣=␣" + s.getVar(donald).getVal());
System.out.println("gerald␣=␣" + s.getVar(gerald).getVal());
System.out.println("robert␣=␣" + s.getVar(robert).getVal());
```

4.1.3 Solution of Exercise 1.3 (A famous example. . . a sudoku grid)

(Problem)

Source code: ExSudoku.zip

Question 1: *propose a way to model the sudoku problem with difference constraints. Implement your model with choco solver.*

```
int n = instance.length;
// Build Model
Model m = new CPModel();

// Build an array of integer variables
IntegerVariable[][] rows = makeIntVarArray("rows", n, n, 1, n,Options.V_ENUM);

// Not equal constraint between each case of a row
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
        for (int k = j; k < n; k++)
            if (k != j) m.addConstraint(neq(rows[i][j], rows[i][k]));
}

// Not equal constraint between each case of a column
for (int j = 0; j < n; j++) {
    for (int i = 0; i < n; i++)
```

```
        for (int k = 0; k < n; k++)
            if (k != i) m.addConstraint(neq(rows[i][j], rows[k][j]));
}

// Not equal constraint between each case of a sub region
for (int ci = 0; ci < n; ci += 3) {
    for (int cj = 0; cj < n; cj += 3)
        // Extraction of disequality of a sub region
        for (int i = ci; i < ci + 3; i++)
            for (int j = cj; j < cj + 3; j++)
                for (int k = ci; k < ci + 3; k++)
                    for (int l = cj; l < cj + 3; l++)
                        if (k != i || l != j) m.addConstraint(neq(rows[i][j], rows[k][l]));
}

//...

// Call solver
Solver s = new CPSolver();
s.read(m);
CPSolver.setVerbosity(CPSolver.SOLUTION);
s.solve();
CPSolver.flushLogs();
printGrid(rows, s);
```

***Question 2**: which global constraint can be used to model such a problem ? Modify your code to use this constraint.*

The *allDifferent* constraint can be used to remplace every disequality constraint on the first Sudoku model. It improves the efficient of the model and make it more "readable".

```
// Build model
Model m = new CPModel();
// Declare variables
IntegerVariable[][] cols = new IntegerVariable[n][n];
IntegerVariable[][] rows = makeIntVarArray("rows", n, n, 1, n,Options.V_ENUM);

// Channeling between rows and columns
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
        cols[i][j] = rows[j][i];
}

// Add alldifferent constraint
for (int i = 0; i < n; i++) {
    m.addConstraint(allDifferent(cols[i]));
    m.addConstraint(allDifferent(rows[i]));
}

// Define sub regions
IntegerVariable[][] carres = new IntegerVariable[n][n];
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++)
        for (int k = 0; k < 3; k++)
            carres[j + k * 3][i] = rows[0 + k * 3][i + j * 3];
            carres[j + k * 3][i + 3] = rows[1 + k * 3][i + j * 3];
            carres[j + k * 3][i + 6] = rows[2 + k * 3][i + j * 3];
}

// Add alldifferent on sub regions
for (int i = 0; i < n; i++) {
    Constraint c = allDifferent(carres[i]);
```

```
    m.addConstraint(c);
}

//...

// Call solver
Solver s = new CPSolver();
s.read(m);
CPSolver.setVerbosity(CPSolver.SOLUTION);
s.solve();
printGrid(rows, s);
```

***Question 3**: Test for both model the initial propagation step (use choco* `propagate()` *method). What can be noticed ? What is the point in using global constraints ?*

The sudoku problem can be solved just with the propagation. FIXME explanation. The global constraint provides a more efficient filter algorithm, due to more complex deduction.

4.1.4 Solution of Exercise 1.4 (The knapsack problem)

(Problem)

Source code: ExKnapSack.zip

***Question 1** : In the first place, we will not consider the idea of maximizing the energetic value. Try to find a satisfying solution by modelling and implementing the problem within choco.*

```
Model m = new CPModel();

obj1 = makeIntVar("obj1", 0, 5,Options.V_ENUM);
obj2 = makeIntVar("obj2", 0, 7,Options.V_ENUM);
obj3 = makeIntVar("obj3", 0, 10,Options.V_ENUM);
c = makeIntVar("cost", 1, 1000000,Options.V_BOUND);

int capacity = 34;
int[] volumes = new int[]{7, 5, 3};
int[] energy = new int[]{6, 4, 2};

m.addConstraint(leq(scalar(volumes, new IntegerVariable[]{obj1, obj2, obj3}), capacity));
m.addConstraint(eq(scalar(energy, new IntegerVariable[]{obj1, obj2, obj3}), c));

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

s.solve();

System.out.println("("+s.getVar(obj1).getVal()+","+s.getVar(obj2).getVal()+","
                   +s.getVar(obj3).getVal()+")␣cost␣=␣"+ s.getVar(c).getVal());
```

***Question 2** : Find and use the choco method to maximise the energetic value of the knapsack.* Replace `s.solve()` by:

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

***Question 3** : Propose a Value selector heuristic to improve the efficiency of the model.*

It can be improved using the following value selector strategy. It iterates over decreasing values of every domain variables:

```
s.setValIntIterator(new DecreasingDomain());
```

4.1.5 Solution of Exercise 1.5 (The n-queens problem)

(Problem)

Source code: ExQueen.zip

Question 1 *: propose and implement a model based on one L_i variable for every row...*

```
Model m = new CPModel();

IntegerVariable[] queens = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    queens[i] = makeIntVar("Q" + i, 1, n,Options.V_ENUM);
}

for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        int k = j - i;
        m.addConstraint(neq(queens[i], queens[j]));
        m.addConstraint(neq(queens[i], plus(queens[j], k))); // diagonal
        m.addConstraint(neq(queens[i], minus(queens[j], k))); // diagonal
    }
}

Solver s = new CPSolver();
s.read(m);
CPSolver.setVerbosity(CPSolver.SOLUTION);
int timeLimit = 60000;
s.setTimeLimit(timeLimit);
s.solve();
CPSolver.flushLogs();
```

Question 2 *: Add a redundant model by considering variable on the columns (C_i). Continue to use simple difference constraints.*

```
Model m = new CPModel();

IntegerVariable[] queens = new IntegerVariable[n];
IntegerVariable[] queensdual = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    queens[i] = makeIntVar("Q" + i, 1, n,Options.V_ENUM);
    queensdual[i] = makeIntVar("QD" + i, 1, n,Options.V_ENUM);
}

for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        int k = j - i;
        m.addConstraint(neq(queens[i], queens[j]));
        m.addConstraint(neq(queens[i], plus(queens[j], k))); // diagonal
        m.addConstraint(neq(queens[i], minus(queens[j], k))); // diagonal
    }
}

for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        int k = j - i;
        m.addConstraint(neq(queensdual[i], queensdual[j]));
        m.addConstraint(neq(queensdual[i], plus(queensdual[j], k))); // diagonal
        m.addConstraint(neq(queensdual[i], minus(queensdual[j], k))); // diagonal
    }
}
m.addConstraint(inverseChanneling(queens, queensdual));

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

s.setVarIntSelector(new MinDomain(s,s.getVar(queens)));
```

```
CPSolver.setVerbosity(CPSolver.SOLUTION);
s.setLoggingMaxDepth(50);
int timeLimit = 60000;
s.setTimeLimit(timeLimit);
s.solve();
CPSolver.flushLogs();
```

Question 3 *: Compare the number of nodes created to find the solutions with both models. How can you explain such a difference ?*

The channeling permit to reduce more nodes from the tree search... FIXME

Question 4 *: Add to the previous implemented model the following heuristics,*

- *Select first the line variable (L_i) which has the smallest domain ;*
- *Select the value $j \in L_i$ so that the associated column variable C_j has the smallest domain.*

Again, compare both approaches in term of nodes number and solving time to find ONE solution for $n = 75, 90, 95, 105$.

Add the following lines to your program (after the reading of the model):

```
s.setVarIntSelector(new MinDomain(s,s.getVar(queens)));
s.setValIntSelector(new NQueenValueSelector(s.getVar(queensdual)));
```

The variable selector strategy (`MinDomain`) already exists in Choco. It iterates over variables given and returns the variable ordering by creasing domain size. The value selector strategy has to be created as follow:

```
public class NQueenValueSelector implements ValSelector {

    // Column variable
    protected IntDomainVar[] dualVar;

    // Constructor of the value selector,
    public NQueenValueSelector(IntDomainVar[] cols) {
        this.dualVar = cols;
    }

    // Returns the "best val" that is the smallest column domain size OR -1
    // (-1 is not in the domain of the variables)
    public int getBestVal(IntDomainVar intDomainVar) {
        int minValue = 10000;
        int v0 = -1;
        IntIterator it = intDomainVar.getDomain().getIterator();
        while (it.hasNext()){
            int i = it.next();
            int val = dualVar[i - 1].getDomainSize();
            if (val < minValue) {
                minValue = val;
                v0 = i;
            }
        }
        return v0;
    }
}
```

Question 5 *: what changes are caused by the use of the global constraint* ***alldifferent*** *?*

```
Model m = new CPModel();

IntegerVariable[] queens = new IntegerVariable[n];
IntegerVariable[] queensdual = new IntegerVariable[n];
IntegerVariable[] diag1 = new IntegerVariable[n];
```

```
IntegerVariable[] diag2 = new IntegerVariable[n];
IntegerVariable[] diag1dual = new IntegerVariable[n];
IntegerVariable[] diag2dual = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    queens[i] = makeIntVar("Q" + i, 1, n,Options.V_ENUM);
    queensdual[i] = makeIntVar("QD" + i, 1, n,Options.V_ENUM);
    diag1[i] = makeIntVar("D1" + i, 1, 2 * n,Options.V_ENUM);
    diag2[i] = makeIntVar("D2" + i, -n, n,Options.V_ENUM);
    diag1dual[i] = makeIntVar("D1" + i, 1, 2 * n,Options.V_ENUM);
    diag2dual[i] = makeIntVar("D2" + i, -n, n,Options.V_ENUM);
}

m.addConstraint(allDifferent(queens));
m.addConstraint(allDifferent(queensdual));
for (int i = 0; i < n; i++) {
    m.addConstraint(eq(diag1[i], plus(queens[i], i)));
    m.addConstraint(eq(diag2[i], minus(queens[i], i)));
    m.addConstraint(eq(diag1dual[i], plus(queensdual[i], i)));
    m.addConstraint(eq(diag2dual[i], minus(queensdual[i], i)));
}
m.addConstraint(inverseChanneling(queens,queensdual));

m.addConstraint(allDifferent(diag1));
m.addConstraint(allDifferent(diag2));
m.addConstraint(allDifferent(diag1dual));
m.addConstraint(allDifferent(diag2dual));

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

s.setVarIntSelector(new MinDomain(s,s.getVar(queens)));
s.setValIntSelector(new NQueenValueSelector(s.getVar(queensdual)));

CPSolver.setVerbosity(CPSolver.SOLUTION);
int timeLimit = 60000;
s.setTimeLimit(timeLimit);
s.solve();
CPSolver.flushLogs();
```

4.2 I know CP

4.2.1 Solution of Exercise 2.1 (Bin packing, cumulative and search strategies)

(Problem)

Source code: BinPackingv2.zip

4.2.2 Solution of Exercise 2.2 (Social golfer)

(Problem)

Source code: SocialGolferv2.zip

4.2.3 Solution of Exercise 2.3 (Golomb rule)

under development

(Problem)

4.3 I know CP and Choco2.0

4.3.1 Solution of Exercise 3.1 (Hamiltonian Cycle Problem Traveling Salesman Problem)

(Problem)
Source code: ExTSP.zip

4.3.2 Solution of Exercise 3.2 (Shop scheduling)

(Problem)
under development

Chapter 5

Applications with global constraints

5.1 Placement and use of the Geost constraint

The global constraint **geost**(k, O, S, C) handles in a generic way a variety of geometrical constraints C in space and time between polymorphic $k \in \mathbb{N}$ dimensional objects O, each of which taking a shape among a set of shapes S during a given time interval and at a given position in space. Each shape from S is defined as a finite set of shifted boxes, where each shifted box is described by a box in a k-dimensional space at the given offset with the given sizes.

More precisely a *shifted box s= shape(sid,t[],l[])* is an entity defined by a shape id *sid*, an shift offset *s.t[d]*, $0 \leq d < k$, and a size *s.l[d]*> 0, $0 \leq d < k$. All attributes of a shifted box are integer values. Then, a *shape* from S is a collection of shifted boxes sharing all the same shape id. Note that the shifted boxes associated with a given shape may or may not overlap. This sometimes allows a drastic reduction in the number of shifted boxes needed to describe a shape. Each *object o= object(id, sid,x[], start, duration,end)* from O is an entity defined by a unique object id *o.id* (an integer), a shape id *o.sid*, an origin *o.x[d]*, $0 \leq d < k$, a starting time *o.start*, a duration *o.duration*> 0, and a finishingxs time *o.end*.

All attributes *sid, x[0],x[1],...,x[k-1], start, duration, end* correspond to domain variables. Typical constraints from the list of constraints C are for instance the fact that a given subset of objects from O do not pairwise overlap. Constraints of the list of constraints C have always two first arguments A_i and O_i (followed by possibly some additional arguments) which respectively specify:

- A list of dimensions (integers between 0 and k-1), or attributes of the objects of O the constraint considers.
- A list of identifiers of the objects to which the constraint applies.

5.1.1 Example and way to implement it

We will explain how to use *geost* although a 2D example. Consider we have 3 objects o_0, o_1, o_2 to place them inside a box B (3x4) such that they don't overlap (see Figure bellow). The first object o_0 has two potential shapes while o_1 and o_2 have one shape. Given that the placement of the objects should be totally inside B this means that the domain of the origins of objects are as follows (we start from 0 this means that the placement space is from 0 to 2 on x and from 0 to 3 on y:

- o_0: x in 0..1, y in 0..1,
- o_1: x in 0..1, y in 0..1,
- o_2: x in 0..1, y in 0..3.

We describe now how to solve this problem by using Choco.

Build a CP model.

To begin the implementation we build a CP model:

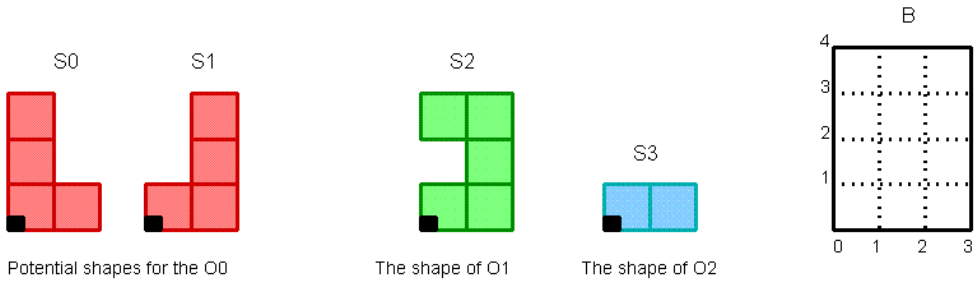


Figure 5.1: Geost objects and shapes

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

Set the Dimension.

Then we first need to specify the dimension k we are working in. This is done by assigning the dimension to a local variable that we will use later:

```
int dim = 2;
```

Create the Objects.

Then we start by creating the objects and store them in a vector as such:

```
Vector<GeostObject> objects = new Vector<GeostObject>();
```

Now we create the first object o_0 by creating all its attributes.

```
int objectId = 0; // object id
IntegerVariable shapeId = Choco.makeIntVar("sid", 0, 1); // shape id (2 possible values)
IntegerVariable coords[] = new IntegerVariable[dim]; // coordinates of the origin
coords[0] = Choco.makeIntVar("x", 0, 1);
coords[1] = Choco.makeIntVar("y", 0, 1);
```

We need to specify 3 more Integer Domain Variables representing the temporal attributes (start, duration and end), which for the current implementation of `geost` are not working, however we need to give them dummy values.

```
IntegerVariable start = Choco.makeIntVar("start", 0, 0);
IntegerVariable duration = Choco.makeIntVar("duration", 1, 1);
IntegerVariable end = Choco.makeIntVar("end", 1, 1);
```

Finally we are ready to add the object 0 to our *objects* Vector:

```
objects.add(new GeostObject(dim, objectId, shapeId, coords, start, duration, end));
```

Now we do the same for the other object o_1 and o_2 and add them to our *objects* vector.

Create the Shifted Boxes.

To create the shapes and their shifted boxes we create the shifted boxes and associate them with the corresponding shapeId. This is done as follows, first we create a Vector called *sb* for example

```
Vector<ShiftedBox> sb = new Vector<ShiftedBox> ();
```

To create the shifted boxes for the shape 0 (that corresponds to o_0), we start by the first shifted box by creating the sid and 2 arrays one to specify the offset of the box in each dimension and one for the size

of the box in each dimension:

```
int sid = 0;
int[] offset = {0,0};
int[] sizes = {1,3};
```

Now we add our shiftedbox to the *sb* Vector:

```
sb.add(new ShiftedBox(sid, offset, sizes));
```

We do the same with second shifted box:

```
sb.add(new ShiftedBox(0, new int[]{0,0}, new int[]{2,1}));
```

By the same way we create the shifted boxes corresponding to second shape S_1:

```
sb.add(new ShiftedBox(1, new int[]{0,0}, new int[]{2,1}));
sb.add(new ShiftedBox(1, new int[]{1,0}, new int[]{1,3}));
```

and the third shape S_2 consisting of three shifted boxes:

```
sb.add(new ShiftedBox(2, new int[]{0,0}, new int[]{2,1})) ;
sb.add(new ShiftedBox(2, new int[]{1,0}, new int[]{1,3}));
sb.add(new ShiftedBox(2, new int[]{0,2}, new int[]{2,1}));
```

and finally the last shape S_3

```
sb.add(new ShiftedBox(3, new int[]{0,0}, new int[]{2,1}));
```

Create the constraints.

First we create a Vector called ectr that will contain the external constraints.

```
Vector <ExternalConstraint> ectr = new Vector <ExternalConstraint>();
```

In order to create the non-overlapping constraint we first create an array containing all the dimensions the constraint will be active in (in our example it is all dimensions) and lets name this array *ectrDim* and a list of objects *objOfEctr* that this constraint will apply to (in our example it is all objects).

> Note that in the current implementation of `geost` only the non-overlapping constraint is available. Moreover, *ectrDim* should contain all dimensions and *objOfEctr* should contain all the objects, i.e. the non-overlapping constraint applies to all the objects in all dimensions.

After that we add the constraint to a vector *ectr* that contains all the constraints we want to add. The code for these steps is as follows:

```
int[] ectrDim = new int[dim];
for(i = 0; i < dim; i ++)
    ectrDim[i] = i;
int[] objOfEctr = new int[3];
for(i = 0; i < 3; i ++)
    objOfEctr[i] = objects.elementAt(i).getObjectId();
```

All we need to do now is create the non-overlapping constraint and add it to the *ectr* vector that holds all the constraints. this is done as follows:

```
//Constants.NON_OVERLAPPING indicates the id of the non-overlapping constraint
NonOverlapping n = new NonOverlapping(Constants.NON_OVERLAPPING, ectrDim, objOfEctr);
ectr.add(n);
```

Create the `geost` constraint and add it to the model.

```
Constraint geost = Choco.geost(dim, objects, sb, ectr);
m.addConstraint(geost);
```

Solve the problem.

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

The full java code can be found here: geostexp.java

5.1.2 Support for Greedy Assignment within the geost Kernel

Motivation and functionality description.

Since, for performance reasons, the `geost` kernel offers a mode where he tries to fix all objects during one single propagation step, we provide a way to specify a preferred order on how to fix all the objects in one single propagation step. This is achieved by:

- Fixing the objects according to the order they were passed to the `geost` kernel.
- When considering one object, fixing its shape variable as well as its coordinates:
  - According to an order on these variables that can be explicitly specified.
  - A value to assign that can either be the smallest or the largest value, also specified by the user.

> Note that the use of the greedy mode assumes that no other constraint is present in the problem.

This is encoded by a term that has exactly the same structure as the term associated to an object of `geost`. The only difference consists of the fact that a variable is replaced by an expression _ (*The character _ denotes the fact that the corresponding attribute is irrelevant, since for instance, we know that it is always fixed*), $\min(I)$ (respectively, $\max(I)$), where I is a strictly positive integer. The meaning is that the corresponding variable should be fixed to its minimum (respectively maximum value) in the order I. We can in fact give a list of vectors $v_1, v_2, \ldots, v_p$ in order to specify how to fix objects $o_{(1+pa)}, o_{(2+pa)}, \ldots, o_{(p+pa)}$.

This is illustrated by Figure bellow: for instance, Part(I) specifies that we alternatively:

- fix the shape variable of an object to its maximum value (i.e., by using max(1)), fix the x-coordinate of an object to its its minimum value (i.e., by using min(2)), fix the y-coordinate of an object to its its minimum value (i.e., by using min(3)) and
- fix the shape variable of an object to its maximum value (i.e., by using max(1)), fix the x-coordinate of an object to its its maximum value (i.e., by using max(2)), fix the y-coordinate of an object to its its maximum value (i.e., by using max(3)).

In the example associated with Part (I) we successively fix objects $o_1, o_2, o_3, o_4, o_5, o_6$ by alternatively using strategies (1) `object(_,max(1),x[min(2),min(3)])` and (2) `object(_,max(1),x[max(2),max(3))`.

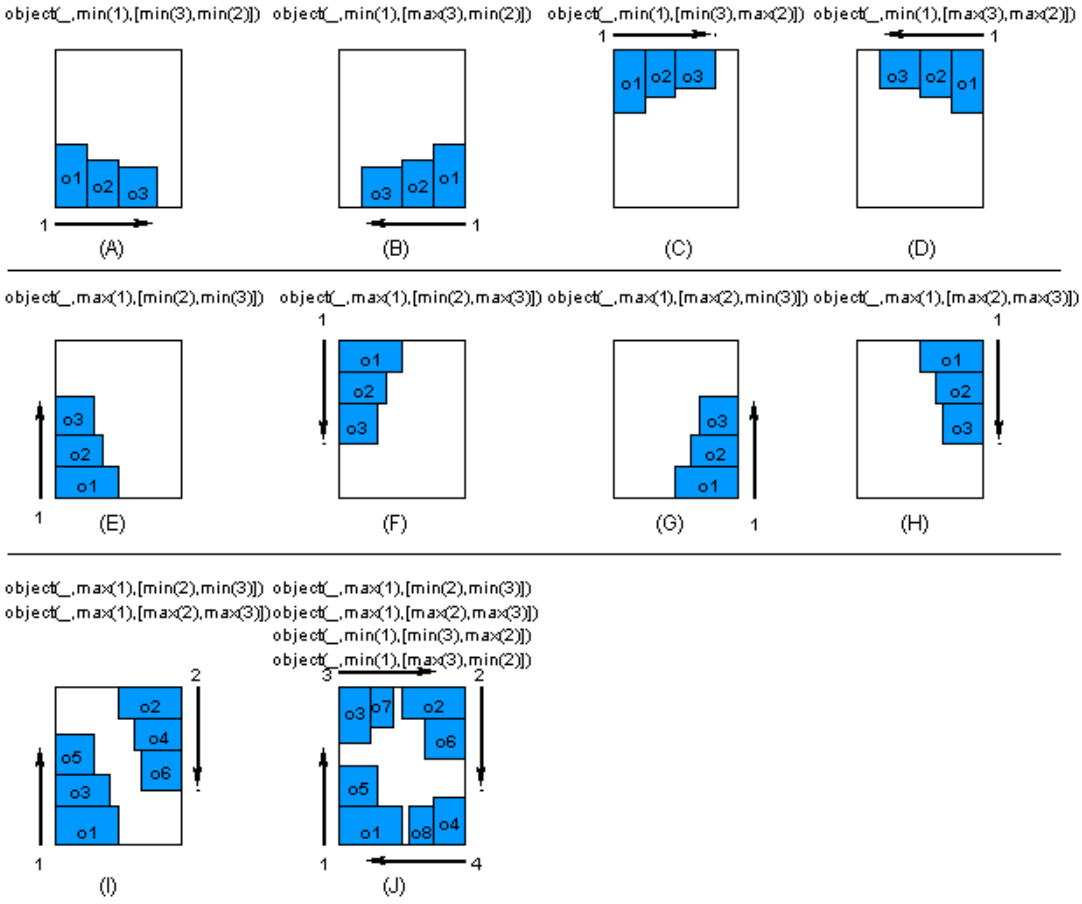


Figure 5.2: Greedy placement

Implementation.

The greedy algorithm for fixing an object o is controlled by a vector v of length $k+1$ such that:

- The shape variable $o.sid$ should be set to its minimum possible value if $v[0] < 0$, and to its maximum possible value otherwise.
- abs($v[1]$)-2 is the most significant dimension (the one that varies the slowest) during the sweep. The values are tried in ascending order if $v[1] < 0$, and in descending order otherwise.
- abs($v[2]$)-2 is the next most significant dimension, and its sign indicates the value order, and so on.

For example, a term `object(_,min(1),[max(3),min(4),max(2)])` is encoded as the vector $[-1, 4, 2, -3]$.

Second example.

We will explain although a 2D example how to take into account of greedy mode. Consider we have 12 identical objects $o_0, o_1, \ldots, o_{11}$ having 4 potential shapes and we want to place them in a box B (7x6) (see Figure bellow). Given that the placement of the objects should be totally inside B this means that the domain of the origins of objects are as follows $x \in [0, 5]$, $y \in [0, 4]$. Moreover, suppose that we want use two strategies when greedy algorithm is called: the term `object(_,min(1),[min(2),min(3)])` for objects $o_0, o_2, o_4, o_6, o_8, o_{10}$, and the term `object(_,max(1),[max(2),max(3)])` for objects $o_1, o_3, o_5, o_7, o_9, o_{11}$. These strategies are encoded respectively as $[-1, -2, -3]$ and $[1, 2, 3]$.

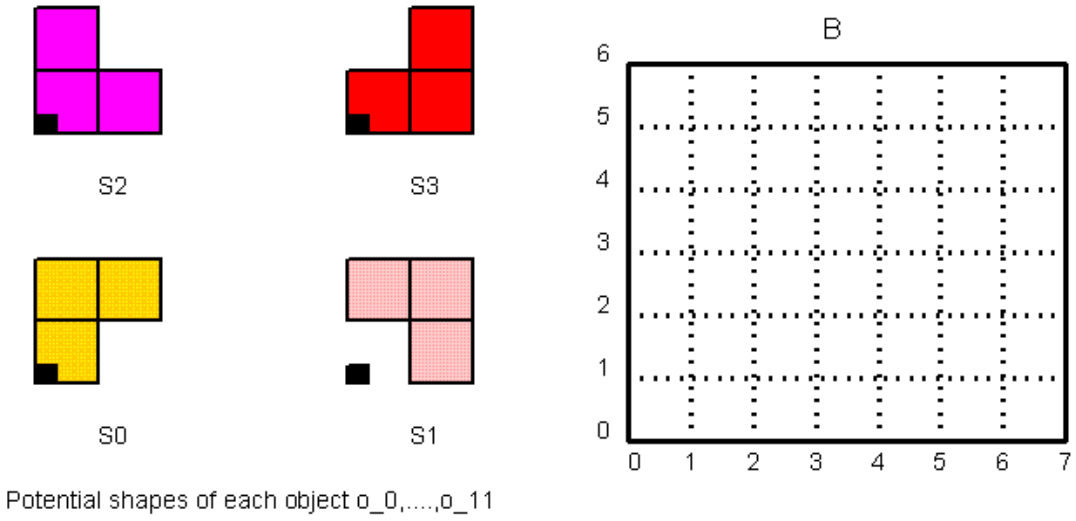


Figure 5.3: A second Geost instance.

We comment only the additional step w.r.t. the preceding example. In fact we just need to create the list of controlling vectors before creating the geost constraint. Each controlling vector is an array:

```
Vector <int[]> ctrlVs = new Vector<int[]>();
int[] v0 = {-1, -2, -3};
int[] v1 = {1, 2, 3};
ctrlVs.add(v0);
ctrlVs.add(v1);
```

and then create the `geost` constraint, by adding the list of controlling vectots as an another argument, as follows:

```
Constraint geost = Choco.geost(dim, objects, sb, ectr,ctrlVs);
m.addConstraint(geost);
```

The placement obtained using the preceding strategies is displayed in the following figure (right side).

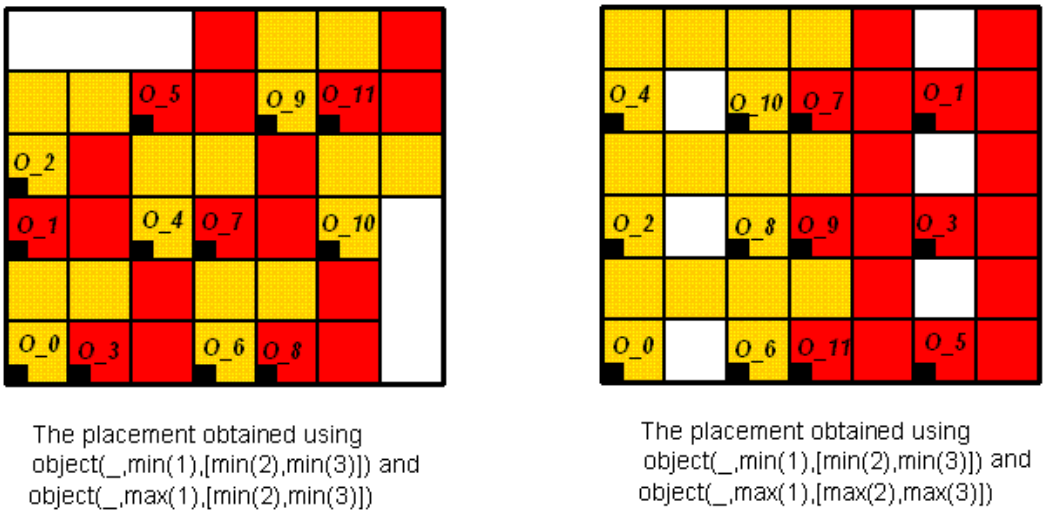


Figure 5.4: A solution placement

5.2 Scheduling and use of the cumulative constraint

This tutorial is the Choco2 version of this one

We present a simple example of a scheduling problem solved using the cumulative global constraint.

The problem is to maximize the number of tasks that can be scheduled on a single resource within a given time horizon.

The following picture summarizes the instance that we will use as example. It shows the resource profile on the left and on the right, the set of tasks to be scheduled. Each task is represented here as a rectangle whose height is the resource consumption of the task and whose length is the duration of the task. Notice that the profile is not a straight line but varies in time.

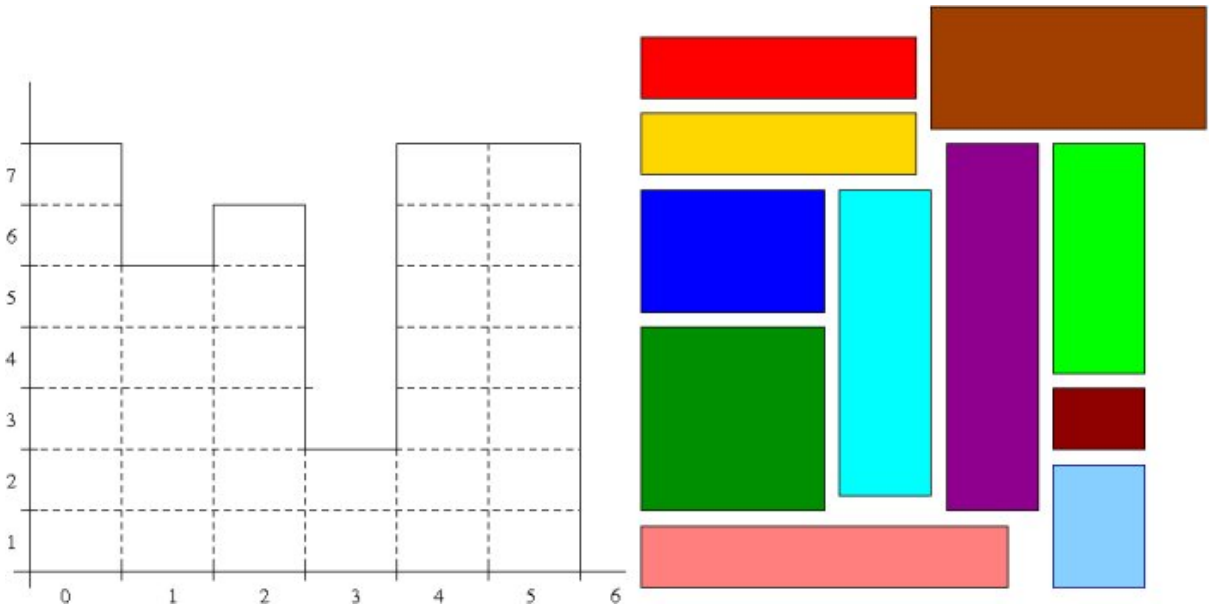


Figure 5.5: A cumulative scheduling problem instance

This tutorial might help you to deal with :

- The profile of the resource that varies in time whereas the API of the cumulative only accepts a

constant capacity

- The objective function that implies optional tasks which is a priori not allowed by cumulative
- A search heuristic that will first assign the tasks to the resource and then try to schedule them while maximizing the number of tasks

The first point is easy to solve by adding fake tasks at the right position to simulate the consumption of the resource. The second point is possible thanks to the ability of the cumulative to handle variable heights. We shall explain it in more details soon.

Let's have a look at the source code and first start with the representation of the instance. We need three fake tasks to decrease the profile accordindly to the instance capacity. There is otherwise 11 tasks. Their heights and duration are fixed and given in the two following int[] tables. The three first tasks correspond to the fake one are all of duration 1 and heights 2, 1, 4.

```
CPModel m = new CPModel();
// data
int n = 11 + 3; //number of tasks (include the three fake tasks)
int[] heights_data = new int[]{2, 1, 4, 2, 3, 1, 5, 6, 2, 1,3, 1, 1, 2};
int[] durations_data = new int[]{1, 1, 1, 2, 1, 3, 1, 1, 3, 4,2, 3, 1, 1};
```

The variables of the problem consist of four variables for each task (start, end, duration, height). We recall here that the scheduling convention is that a task is active on the interval [start, end-1] so that the upper bound of the start and end variables need to be 5 and 6 respectively. Notice that start and end variables are BoudIntVar variables. Indeed, the cumulative is only performing bound reasonning so it would be a waste of efficiency to declare here EnumVariables. Duration and heights are constant in this problem. However, our plan is to simulate the allocation of a task to the resource by using variable height. In other word, we will define the height of the task i as a variable of domain $\{0, \texttt{height[i]}\}$. The height of the task takes its normal value if the task is assigned to the resource and 0 otherwise. The duration is really constant and is therefore created as a ConstantIntVar.

Moreover, we add a boolean variable per task to specify if the task is assigned to the resource or not. The objective variable is created as a BoundIntVar.

```
IntegerVariable capa = constant(7);
IntegerVariable[] starts = makeIntVarArray("start", n, 0, 5, Options.V_BOUND);
IntegerVariable[] ends = makeIntVarArray("end", n, 0, 6, Options.V_BOUND);

IntegerVariable[] duration = new IntegerVariable[n];
IntegerVariable[] height = new IntegerVariable[n];
for (int i = 0; i < height.length; i++) {
    duration[i] = constant(durations_data[i]);
    height[i] = makeIntVar("height␣" + i, new int[]{0, heights_data[i]});
}

IntegerVariable[] bool = makeIntVarArray("taskIn?", n, 0, 1);
IntegerVariable obj = makeIntVar("obj", 0, n, Options.V_BOUND, Options.V_OBJECTIVE);
```

We then add the constraints to the model. Three constraints are needed. First, the cumulative ensures that the resource consumption is respected at any time. Then we need to make sure that if a task is assigned to the cumulative, its height can not be null which is done by the use of boolean channeling constraints. Those constraints ensure that :

$$\texttt{bool[i]} = 1 \iff \texttt{height[i]} = \texttt{heights\_data[i]}$$

We state the objective function to be the sum of all boolean variables.

```
//post the cumulative
m.addConstraint(cumulative(starts, ends, duration, height, capa, ""));
//post the channeling to know if the task is scheduled or not
for (int i = 0; i < n; i++) {
    m.addConstraint(boolChanneling(bool[i], height[i], heights_data[i]));
```

```
}

//state the objective function
m.addConstraint(eq(sum(bool), obj));
```

Finally we fix the fake task at their position to simulate the profil:

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

//set the fake tasks to establish the profile capacity of the resource
try {
    s.getVar(starts[0]).setVal(1); s.getVar(ends[0]).setVal(2); s.getVar(height[0]).setVal(2);
    s.getVar(starts[1]).setVal(2); s.getVar(ends[1]).setVal(3); s.getVar(height[1]).setVal(1);
    s.getVar(starts[2]).setVal(3); s.getVar(ends[2]).setVal(4); s.getVar(height[2]).setVal(4);
} catch (ContradictionException e) {
    System.out.println("error, no contradiction expected at this stage");
}
```

We are now ready to solve the problem. We could call a maximize(false) but we want to add a specific heuristic that first assigned the tasks to the cumulative and then tries to schedule them.

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

We now want to print the solution and will use the following code :

```
System.out.println("Objective : " + (s.getVar(obj).getVal() - 3));
for (int i = 3; i < starts.length; i++) {
   if (s.getVar(height[i]).getVal() != 0)
      System.out.println("[" + s.getVar(starts[i]).getVal() + " - "
                         + (s.getVar(ends[i]).getVal() - 1) + "]:"
                         + s.getVar(height[i]).getVal());
}
```

Choco gives the following solution :

```
Objective : 9
[1 - 2]:2
[0 - 0]:3
[2 - 4]:1
[4 - 4]:5
[5 - 5]:6
[0 - 2]:2
[0 - 3]:1
[3 - 5]:1
[0 - 0]:1
```

This solution could be represented by the following picture :

Notice that the cumulative gives a necesserary condition for packing (if no schedule exists then no packing exists) but this condition is not sufficient as shown on the picture because it only ensures that the capacity is respected at each time point. Specially, the tasks might be splitted to fit in the profile as in the previous solution. The complete code can be found here.

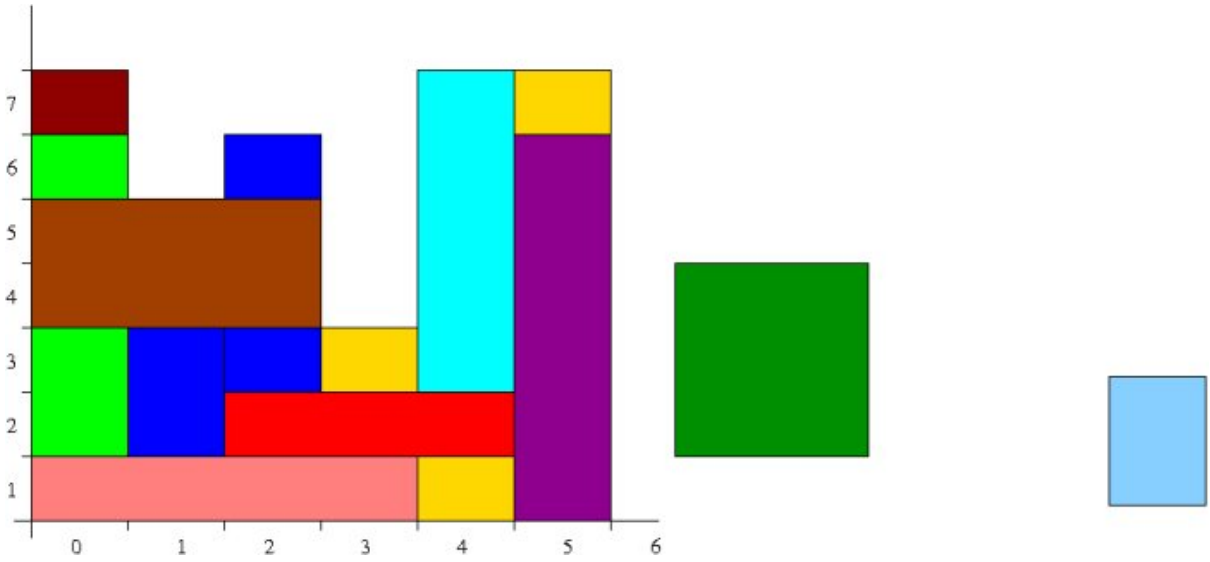


Figure 5.6: Cumulative profile of a solution.

Bibliography

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.