# Lab Unit 7

## Set covering, set partitioning and set packing problems

## *Problemas de recubrimiento, particionamiento y empaquetamiento*



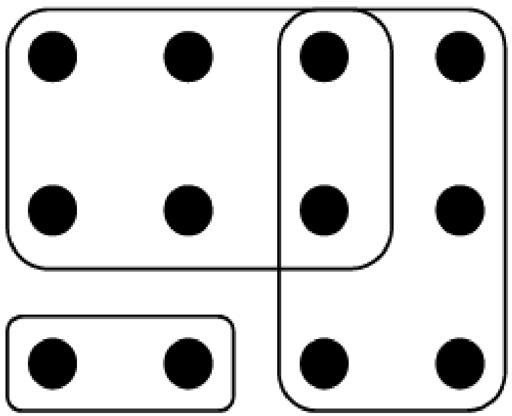
## Contents Part 1

- The set covering problem
- The set partitioning problem
- The set packing problem
- Use  of Choco

**http://choco-solver.org**



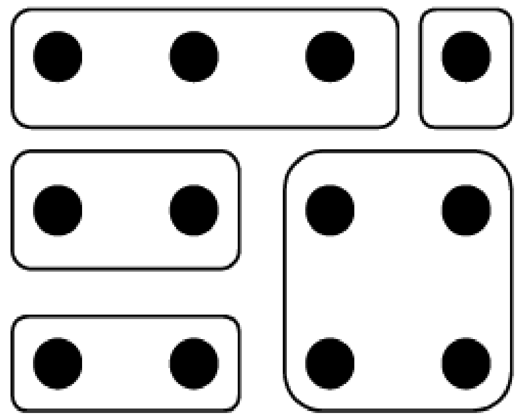

# Remember…



Covering

All items in **at least one** partition

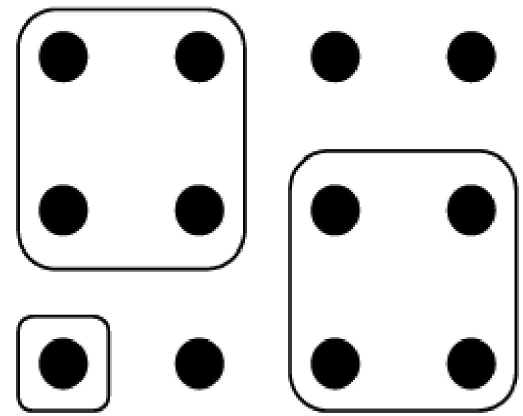
$\forall$**item i:** |covered(i)| ≥ 1



Partitioning

All items **in one and only one** partition

|covered(i)| = 1



Packing

All items in **zero or at most one** partition

|covered(i)| ≤ 1

## What's Choco?

- Choco is a **free** and **open-source** software for **constraint programming**
- It is a Java library, aiming at describing hard combinatorial problems in the form of CSPs and solving them with constraint programming techniques
- The user **models** the problem in a declarative way (in Java). Then, **Choco solves** the problem by alternating constraint **filtering algorithms** with a **search** mechanism
- Choco is among the **fastest** CP solvers on the market

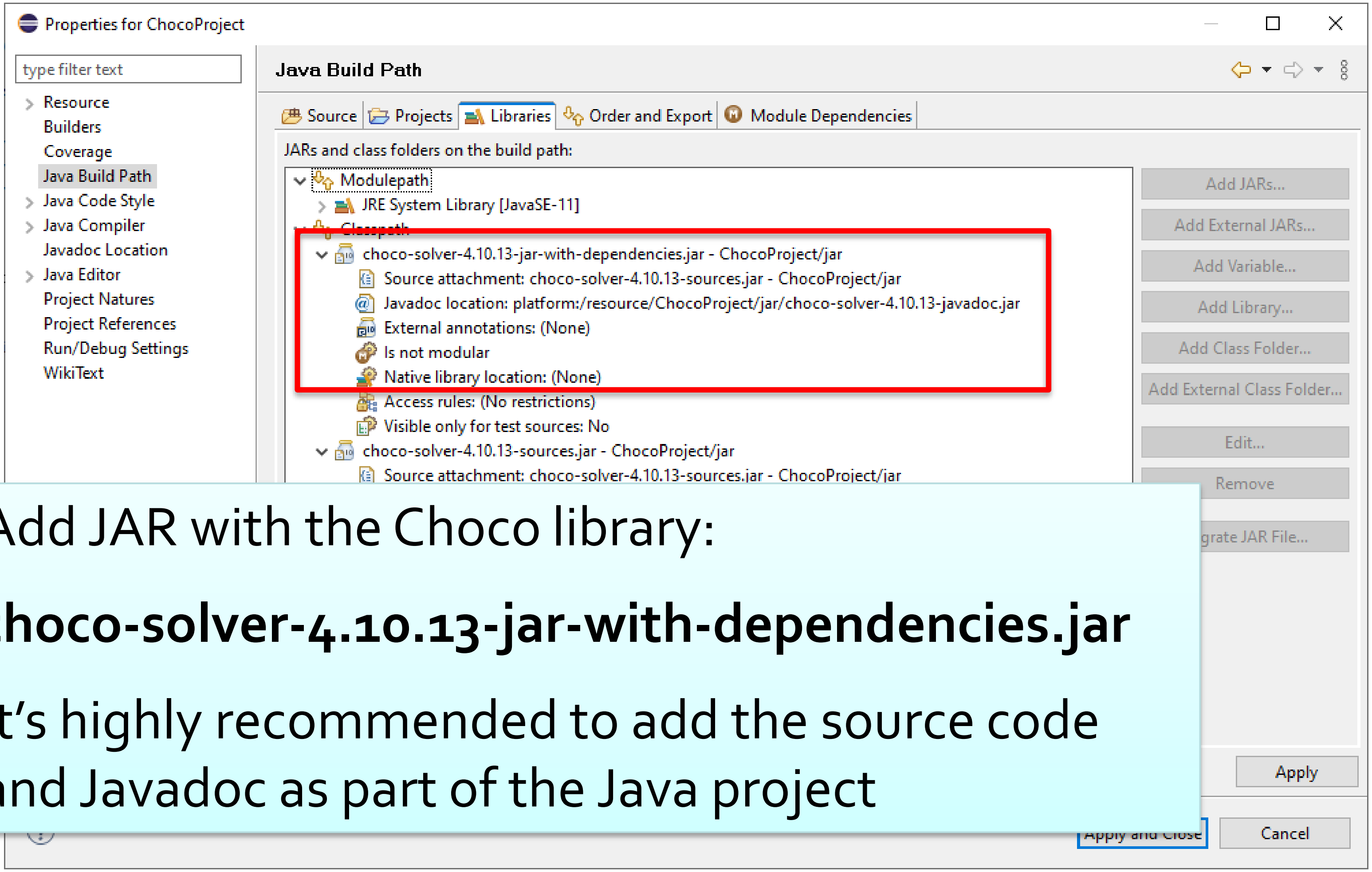
## Very well supported

- User Guides, tutorials and learning documentation
  - https://choco-solver.org/docs/
  - https://choco-solver.org/tutos/
  - https://github.com/chocoteam/choco-solver/releases

- Even with a forum (google group):
  - https://groups.google.com/g/choco-solver



# **Creation** of a **Java project, as usual**




- Add JAR with the Choco library:

  **choco-solver-4.10.13-jar-with-dependencies.jar**

- It's highly recommended to add the source code and Javadoc as part of the Java project

## 1. Create a **model**

```
Model model = new Model(“My example”);
```

## 2. Create the **variables** with their **domains**

```
// one int (bounded) variable with domain [0..5]
IntVar x = model.intVar(“var”,0,5);
// one array of 5 variables with domain [-2..8]
IntVar[] vs = model.intVarArray(“vector”,5,-2,8);
// one 5x6 matrix of variables with domain [10..15]
IntVar[][] m = model.intVarMatrix(“mt”,5,6,10,15);
IntVar x = model.intVar(0);  // fixed (i.e. constant): x = 0
```



3. There are many **different types** of **variables** (check the manual!)

- Enumerated (*rather than bounded*)

```
IntVar v = model.intVar(“v1”,1,4,false); //not boundedDomain
IntVar v = model.intVar(“v2”,new int[]{1,7,8});
```

- Boolean

```
BoolVar bs = model.boolVar(“bool”);  // or boolVarArray(size)
```

- Real, with a given precision (e.g. precision= 0.001d)

```
RealVar x = model.realVar(“y”,0.2d,5.6d,precision);
```

- Set

```
SetVar y = model.setVar(“y”, new int[]{}, new int[]{1,2,3,5,12});
// a subset of values with possible values: {}, {2}, {1,3,5} ...
```



3. **Variable** as a function of another variable (**view**)

- A simple way to model and shortcut the creation of a variable and its constraint

```
IntVar x = model.intOffsetView(y,2);  // x = y + 2
IntVar x = model.intMinusView(y);      // x = -y
IntVar x = model.intScaleView(y,3);    // x = 3*y
IntVar x = model.intOffsetView(model.intScaleView(y,2),5);
// x = 2*y + 5
```

# 4. Create the **constraints** (check the manual!)

```
Constraint a = model.arithm(x,“<”,0);  // <,>,!=,=,<=,>=

Constraint b = model.arithm(y,“>”,z,“+”,42);

Constraint c = model.arithm(x,“>”,y,“-”,z);

Constraint c = model.or(boolvar1, boolvar2, boolvar3);
// boolvar1 OR boolvar2 OR boolvar3
```

absolute, allDifferent, allDifferent, allDifferentExcept0, allDifferentUnderCondition, allEqual, among, and, and, arithm, arithm, arithm, arithm, atLeastNValues, atMostNValues, binPacking, bitsIntChanneling, boolsIntChanneling, circuit, circuit, circuit, clausesIntChanneling, costRegular, count, count, cumulative, cumulative, cumulative, cumulative, cumulative, diffN, distance, distance, div, element, element, element, getDomainUnion, globalCardinality, intValuePrecedeChain, intValuePrecedeChain, inverseChanneling, inverseChanneling, keySort, knapsack, lexChainLess, lexChainLessEq, lexLess, lexLessEq, max, max, max, mddc, member, member, min, min, min, mod, multiCostRegular, multiCostRegular, not, notAllEqual, notMember, notMember, nValues, or, or, path, path, regular, scalar, scalar, scalar, scalar, sort, square, subCircuit, subPath, sum, sum, sum, sum, sum, sum, sum, table, table, table, table, times, times, times, tree, tree

allDifferent, allDisjoint, allEqual, disjoint, element, element, intersection, intersection, inverseSet, max, max, member, member, member, min, min, nbEmpty, nbEmpty, notEmpty, notMember, notMember, offSet, partition, setBoolsChanneling, setBoolsChanneling, setsIntsChanneling, setsIntsChanneling, subsetEq, sum, sumElements, sumElements, symmetric, symmetric, union, union

ifOnlyIf, ifThen, ifThen, ifThenElse, ifThenElse, reification, reifyXeqC, reifyXeqY, reifyXgtC, reifyXinS, reifyXleY, reifyXltC, reifyXltY, reifyXltYC, reifyXneC, reifyXneY



# 4. Create the **constraints**

```
Constraint sc1 = model.scalar(IntVar[] VARS, int[] COEFFS,
String OPERATOR, int SCALAR);
```

$$\sum_{\forall i} VARS[i] * COEFFS[i] \quad \text{"OPERATOR"} \quad SCALAR$$

```
Constraint sc2 = model.scalar(IntVar[] VARS, int[] COEFFS,
String OPERATOR, IntVar SCALAR_VAR);
```

$$\sum_{\forall i} VARS[i] * COEFFS[i] \quad \text{"OPERATOR"} \quad SCALAR\_VAR$$

where "**OPERATOR**" is {"=", "!=", ">","<",">=","<="}

```
model.sum(IntVar[] VARS, String OPERATOR, int SUM);

model.sum(IntVar[] VARS, String OPERATOR, IntVar SUM_VAR);
// like scalar method with all COEFFS==1
```



5. **Post** the **constraints** (otherwise, they won't be satisfied)

```
model.post(constraint);
model.post(model.arithm(x,"+",y,"<",5));
```

or, in a **more convenient** way:

```
model.allDifferent(vars).post();
model.sum(vars,"<",3).post();
model.scalar(vars,coeffs,"=",y).post();

Constraint a = model.arithm(x,"<",0);
Constraint b = model.arithm(y,">",42);
model.ifThen(a,b); //automatic posting; also ifThenElse
```


## 5. **Post** the **constraints** (otherwise, they won't be satisfied)

```
model.times(x,y,z).post();      // x * y = z
model.div(x,y,z).post();        // x / y = z
model.mod(x,y,z).post();        // x % y = z
```

## and **other constraints** (methods of model)…

```
atLeastNValues(IntVar[] VARS, IntVar NVALUES, boolean AC);

atMostNValues(IntVar[] VARS, IntVar NVALUES, boolean
STRONG);

count(IntVar VALUE, IntVar[] VARS, IntVar LIMIT);

cumulative(Task[] TASKS, IntVar[] HEIGHTS, IntVar CAPACITY);

distance(IntVar VAR1, IntVar VAR2, String OP, IntVar VAR3);
```



1. Once the model has been created, we have to **solve** it

```
Solver solver = model.getSolver();
solver.findSolution(STOPCriterion);  // nodes, time, etc.
```

or simply: `solver.solve();  // returns T/F`

```
if (solver.solve())
{
    // do something, i.e. print out variable values
}
```

optional

Or **alternatively**

```
solver.findAllSolutions(STOPCriterion); // attempts to find
all feasible solutions
```

2. Finding the **optimal solution** (COP; check the manual!)

```
Solution sol =
solver.findOptimalSolution(objectiveVar,maximize);  // true
= maximize; false = minimize
```

or

```
solver.findAllOptimalSolutions(objectiveVar,maximize);
```

If we want to deal with **multi-objective optimisation**

```
solver.findParetoFront(objectiveVars,maximize)   // now with
a collections of objectiveVars
```

when a solution is found, a cut is posted, and at least one of the objective variables must be better in the next solution

3. Once the model has been solved, we **get** the **instantiated values**

```
var.getLB();        // lower bound of var
var.getUB()         // upper bound of var
var.getValue();     // instantiated value of var
```

e.g. to **print all** the values of integer variables

```
if (solver.solve())
{
    for (Variable var: model.getVars())
       if (var instanceof IntVar)
       {
          System.out.println(((IntVar) var).getName() + ": " +
          ((IntVar) var).getValue());
       }}}
```

access the values **before** the search tree has been closed, or use **Solution** recording instead

# 4. Retrieving **statistics** and **extra info**

- **Printing the tree search during the resolution**

```
solver.showDecisions();  // invoke before solving
// Note: it slows down the search process
```

- **Printing statistics (invoke after solving)**

```
solver.printShortStatistics();

solver.printStatistics();

solver.printCSVStatistics();  // comma-separated stats
```

5.**Optionally**, changing the **search strategy** (check the manual!)

```
IntVar x = model.intVar(“X”,0,5);
IntVar y = model.intVar(“Y”,0,10);
model.arithm(x,“+”,y,“<“,5).post();
model.arithm(x,“+”,y,“>”,2).post();
```

**After solving by default** x=3, y=0. But with other strategies…

| **`setSearch(Strategy)`, where `Strategy`** | **Description** | **x** | **y** |
|---|---|---|---|
| `setSearch(minDomLBSearch(x,y))` | Starting from the variable of smallest domain size to its lower bound, first x and then y | 0 | 3 |
| `setSearch(minDomUBSearch(x,y))` | Starting from the variable of smallest domain size to its upper bound | 4 | 0 |

Requires **import static org.chocosolver.solver.search.strategy.Search.*;**



# The **knapsack** problem

*Given a set of type of objects, each with a weight and a reward, determine the number of objects of each type to include in a collection so that the total weight is less than or equal to a given limit and the total reward is as large as possible.*

```
public class Knapsack
{
    private int maxNumObjects = 10;
    private int knapsackMaxCapacity = 250;

    private Model model;
    private Solver solver;

    // input data
    private int[] rewards;      // reward per object
    private int[] weights;      // weight per object

    // variables
    private IntVar[] objectsOfType; // collection of objects to be chosen
    private IntVar totalReward; // total reward to be maximised
    private IntVar totalWeight; // total weight used
```

# The **knapsack** problem

```
public Knapsack()
{
    model = new Model("Knapsack example");
    solver = null;
}
```

```
public void buildModel()
{
    rewards = new int[maxNumObjects];
    weights = new int[maxNumObjects];

    objectsOfType = new IntVar[maxNumObjects];

    for (int i = 0; i < maxNumObjects; i++)  // Data values
    {
        // defining the rewards per object
        rewards[i] = k10_maxRewards[i];

        // defining the weights per object
        weights[i] = k10_maxWeights[i];

        // creating the maxNumObjects objects
        int upDomain = k10_maxOccurrences[i];

        objectsOfType[i] = model.intVar("object_" + (i+1), 0, upDomain);
    }

    // capacity constraint: objects[i] * weights[i] = totalWeight <= knapsackMaxCapacity (the weights don't exceed the max capacity)
    totalWeight = model.intVar("TotalWeightUsed", 0, knapsackMaxCapacity);
    model.scalar(objectsOfType, weights, "=", totalWeight).post();  // totalWeight <= knapsackMaxCapacity by the definition of the domain

    // objective variable
    totalReward = model.intVar("TotalReward", 0, 1000);
    // reward constraint: objects[i] * rewards[i] = totalReward
    model.scalar(objectsOfType, rewards, "=", totalReward).post();

    // Or analogously in a much more simplified way:
    //model.knapsack(objectsOfType, totalWeight, totalReward, weights, rewards).post();
}
```

A specific constraint that makes things easier…

# The **knapsack** problem

```
public void solve()
{
    solver = model.getSolver();
    Solution solution = solver.findOptimalSolution(totalReward, true);  // true = maximize

    if (solution != null)
    {
        solver.printStatistics();
        System.out.println(solution);
    }
}
```

```
public static void main(String[] args)
{
    Knapsack knpsack = new Knapsack();
    knpsack.buildModel();
    knpsack.solve();
}
```

```
////////////////////////////////////////// DATA /////////////////////////////
static int[] k10_maxOccurrences     = {6, 9, 5, 3, 7, 7, 6, 9, 7, 2};
static int[] k10_maxWeights         = {5, 8, 7, 3, 4, 8, 10, 3, 4, 6};
static int[] k10_maxRewards         = {4, 8, 7, 5, 3, 10, 1, 2, 5, 7};
```



## Task 1. Covering all characteristics

## Flight crew assignment

*(from Hillier and Lieberman, 2015)*

An **airways** company has to **assign** 3 **crews** based in San Francisco **to** the 11 current **flights** listed in the first column of the next table. The other 12 columns show the 12 feasible sequences of flights for a crew. (The numbers in each column indicate the order of the flights)

**Exactly three** of the **sequences need** to be chosen (one per crew) to cover every flight, although it is permissible to have more than one crew per sequence on a flight (we'd need to pay as if they were working). **Note**: three crews in total, not three crews per flight

The **cost** of **assigning** a **crew** to a particular sequence of **flights** is given in K$ in the bottom row of the table

# Task 1. Covering all characteristics – Flight crew assignment

| | Feasible Sequence of Flights | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Flight** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
| 1. San Francisco to Los Angeles | 1 | | | 1 | | | 1 | | | 1 | | |
| 2. San Francisco to Denver | | 1 | | | 1 | | | 1 | | | 1 | |
| 3. San Francisco to Seattle | | | 1 | | | 1 | | | 1 | | | 1 |
| 4. Los Angeles to Chicago | | | | 2 | | | 2 | | 3 | 2 | | 3 |
| 5. Los Angeles to San Francisco | 2 | | | | | 3 | | | | 5 | 5 | |
| 6. Chicago to Denver | | | | 3 | 3 | | | | 4 | | | |
| 7. Chicago to Seattle | | | | | | | 3 | 3 | | 3 | 3 | 4 |
| 8. Denver to San Francisco | | 2 | | 4 | 4 | | | | 5 | | | |
| 9. Denver to Chicago | | | | | 2 | | | 2 | | | 2 | |
| 10. Seattle to San Francisco | | | 2 | | | | 4 | 4 | | | | 5 |
| 11. Seattle to Los Angeles | | | | | | 2 | | | 2 | 4 | 4 | 2 |
| Cost, \$1,000's | 2 | 3 | 4 | 6 | 7 | 5 | 7 | 8 | 9 | 9 | 8 | 9 |

**Goal**: **minimise** the **total cost** of the crew assignments that **cover all** the 11 flights



## Task 2. Temporal planning for delivering parcels

A **van** is in the depot at **12:00** and needs to deliver five parcels, being back no later than **20:00**. **Moving** from one place to another always **takes 30 minutes**

There are **different** opening hours and duration for delivery

| Parcel | Customers' opening hours | Duration for delivery |
|---|---|---|
| P1 | 9:00-13:30; 17:00-20:30 | 30 minutes |
| P2 | 12:00-14:30; 16:30-18:30 | 60 minutes |
| P3 | 14:00-17:00; 18:00-20:30 | 30 minutes |
| P4 | 9:00-17:00; 18:30-20:30 | 120 minutes |
| P5 | 12:00-14:30; 18:30-19:30 | 60 minutes |

**Goal**: find the (optimal) temporal plan



**Model** and **solve** the two previous tasks **using Choco**


- Model the variables, constraints and the optimisation function (if necessary)
- Create a Java application and use the services that Choco provides to define all the elements of the subsequent CSP
- Solve the problem and analyse the results