Sat4j case studies

Learning Sat4j use and design by example

Daniel Le Berre Anne Parrain

http://www.sat4j.org/

CRIL-CNRS 8188 - Université d'Artois

Last revision: Wednesday 11th May, 2011 at 22:08

Contents

1 Introduction

The aim of that document is to allow Java programmers to make the best use of Sat4j by following on sample test cases how Sat4j designers are using the library.

The reader is expected to have a basic knowledge of Java and object oriented programming (knowing design patterns would certainly help understanding Sat4j architecture) and propositional logic.

The following examples have been tested with Sat4j 2.3.0 released on March 29, 2011.

The document is a work on progress until we cover the whole set of features of Sat4j. Comments about that document and ideas to improve it are welcome. Just drop us a line at casestudies@sat4j.org or even better use our Jira @ OW2 on the case studies component.

The source code of the examples given in the document are available as archived Eclipse projects in http://www.sat4j.org/casestudies/.

2 Knapsack

Pseudo-Boolean solvers such as Sat4j are probably not the best tools to solve KnapSack problems, since they contain only inequalities, for which Mixed Integer Linear Programming is probably best suited. However, that problem is well suited for an introduction to Sat4j since it allows us to express a simple optimization problem and to highlight the different behavior of the two classes of Pseudo-Boolean solvers available in the library, namely resolution based and cutting planes based solvers.

2.1 The problem

One of the very popular problems in operation research that fits well in the pseudo boolean optimization framework is the knapsack problem. We have a set I of n items, each of them with a weight w_i and a value v_i. We also have a container with a capacity K. The problem is to put in the container the items from I in order to maximize the value of the container while satisfying the capacity constraint. Think about how to fill a luggage when going in vacation.

Formally, we would like to solve the following optimization problem:

$$max : \sum_{i=1}^{n} v_i \times x_i \text{ subject to } \sum_{i=1}^{n} w_i \times x_i \leq K$$

where x_i are boolean variables and v_i, w_i and K are integers.

Sat4j only allows to solve minimization problems, so we need to slightly change the initial formulation into an equivalent minimization problem:

$$min : \sum_{i=1}^{n} -v_i \times x_i \text{ subject to } \sum_{i=1}^{n} w_i \times x_i \leq K$$

Let's start by creating a new class `Item` that contains all the required data: the name of the item, its weight and its value. The complete code of that class is giving in Listing 2.1.

That class is following the usual Java practices: each property has a getter because the fields are not meant to be modified, the constructor allows to quickly set each property and a `toString()` method is implemented to display an item in textual form.

For people used to test driven development, Listing 2.2 provides some test cases in JUnit 4 showing the expected behavior of the `KnapSack` class that we would like to implement using Sat4j.

```java
public class Item {

  private final int weight;
  private final int value;
  private final String name;

  public Item(String name, int weight, int cost) {
    this.name = name;
    this.weight = weight;
    this.value = cost;
  }

  public int getWeight() {
    return weight;
  }

  public int getValue() {
    return value;
  }

  public String getName() {
    return name;
  }

  @Override
  public String toString() {
    return name+"("+weight+"kg,"+value+")";
  }
}
```

Listing 2.1: A simple class representing an item

```java
public class KnapSackTest {

  private KnapSack ks;
  private Item laptop, clothes, books, presents;
  List<Item> items;

  @Before
  public void setUp() {
    laptop = new Item("macbook", 2, 1500);
    clothes = new Item("clothes", 15, 2000);
    books = new Item("books", 5, 200);
    presents = new Item("presents", 4, 1000);
    Item[] itemsArray = { laptop, clothes, books, presents };
    items = Arrays.asList(itemsArray);
    ks = new KnapSack(SolverFactory.newDefault(), 10);
  }

  @Test
  public void testWithACapacityOf10Kg() {
    List<Item> result = ks.solve(items, 10);
    assertEquals(2, result.size());
    assertTrue(result.contains(presents));
    assertTrue(result.contains(laptop));
    assertTrue(result.contains(presents));
    assertEquals(2500, ks.bestValue());
  }

  @Test
  public void testWithACapacityOf15Kg() {
    List<Item> result = ks.solve(items, 15);
    assertEquals(3, result.size());
    assertTrue(result.contains(laptop));
```

```
33      assertTrue(result.contains(presents));
34      assertTrue(result.contains(books));
35      assertEquals(2700, ks.bestValue());
36    }
37
38    @Test
39    public void testWithACapacityOf20Kg() {
40      List<Item> result = ks.solve(items, 20);
41      assertEquals(2, result.size());
42      assertTrue(result.contains(laptop));
43      assertTrue(result.contains(clothes));
44      assertEquals(3500, ks.bestValue());
45    }
46  }
```

Listing 2.2: Some test cases showing the expected behavior of the KnapSack class

2.2 Solving the problem in Java

One of the easiest way to solve such a problem with Sat4j is to directly encode everything in Java.

2.2.1 Using the DependencyHelper class

We need to provide two information to Sat4j: the objective function and the capacity constraint. Since we are working directly on Java objects, we can use the `DependencyHelper` class for that purpose. That class aims at expressing the constraints and the objective functions directly on Java objects. The capacity constraints is an "At Most" constraint, with weighted literals. The `DependencyHelper` class relies on the `WeightedObject` class to provide weighted literals to both the capacity constraint and the objective function. Let's build a new class `KnapSack` with a method `solve()` that solves a knapsack problem for a given list of items and a specific capacity. That class contains a `DependencyHelper` field as shown in Listing 2.5 that will be detailed later. The complete listing of the method `solve()` is given in Listing 2.3.

The first step is to create an array of `WeightedObject` for both the constraint and the objective function. The `DependencyHelper` class is generic, i.e. you are required to parameter it with the type of the elements you want to put in the constraints. As a consequence, the `WeightedObject` class is also generic. A limitation of Java prevents the use of generic type in arrays, as such it is needed to silent some warnings using the `@SuppressWarnings` annotation (lines 2–5).

Those arrays are filled in with `WeightedObject` objects associating respectively an item to its weight and to its value. The `newWO()` method is a convenience factory method from `WeightedObject` class to be used with the `import static` of Java 5.

The capacity constraint is created using the `DependencyHelper.atMost()` method (line 14). The objective function is set line 15.

Sat4j can compute if a solution to that problem exists using the `hasSolution()` method (line 16). The method is expected to return a subset of the input list of items. Sat4j has its own classes to manage lists, that are more efficient than the one that ships with Java because the order of the elements on those lists are not preserved across operations. We will have to move the content of the result from Sat4j into a classical `java.util.List`: this is

```
1  public List<Item> solve(List<Item> items, long capacity) {
2    @SuppressWarnings("unchecked")
3    WeightedObject<Item>[] weightedObjects = new WeightedObject[items
4        .size()];
5    @SuppressWarnings("unchecked")
6    WeightedObject<Item>[] objective = new WeightedObject[items.size()];
7    int index = 0;
8    for (Item item : items) {
9      weightedObjects[index] = newWO(item, item.getWeight());
10     objective[index] = newWO(item, -item.getValue());
11     index++;
12   }
13   try {
14     List<Item> result = new ArrayList<Item>();
15     helper.atMost("Capacity", BigInteger.valueOf(capacity),
16         weightedObjects);
17     helper.setObjectiveFunction(objective);
18     if (helper.hasASolution()) {
19       IVec<Item> sol = helper.getSolution();
20       for (Iterator<Item> it = sol.iterator(); it.hasNext();) {
21         result.add(it.next());
22       }
23     }
24     return result;
25   } catch (ContradictionException e) {
26     return Collections.emptyList();
27   } catch (TimeoutException e) {
28     return Collections.emptyList();
29   }
30 }
```

Listing 2.3: Solving the knapsack problem using a DependencyHelper object

the purpose of lines 17–20.

There are two possible cases that will prevent the code to execute normally:

- If the constraint added to the helper is trivially unsatisfiable, i.e. there is no way to satisfy it, then a `ContradictionException` is launched. It should not occur in our case.
- If the solver reaches a timeout before finding a solution, then a `TimeoutException` is launched.

Note that it would be nice to reset the dependency helper at the beginning of the method `solve()` in order to be able to reuse several times that method on different lists of items or different capacity. However, there is no easy way to do it in Sat4j 2.3.0. A `reset()` method will be available in the `DependencyHelper` class in release 2.3.1.

2.2.2 Optimization versus decision problems

By default, the engines used in Sat4j are solving decision problems, not optimization problems. A decision problem, implementing the `IProblem` interface in Sat4j, means that the expected answer to the problem is yes/no (the answer to `IProblem.isSatisfiable()` or `DependencyHelper.hasASolution()`). In general, the answer yes is usually completed by an assignment, a model, used as a certificate of satisfiability. Here, the certificate is the set of items to put in the container. Solving a decision problem for the knapsack problem is simply choosing a set of items that satisfies the capacity constraint. To find the optimal set of items, the one that maximizes the value of the container, one need to solve an optimization problem.

```
boolean isSatisfiable = false;
IOptimizationProblem optproblem = ..
try {
  // while it is possible to find a better solution
  while (optproblem.admitABetterSolution()) {
    if (!isSatisfiable) {
      isSatisfiable = true;
      // can now start optimization
    }
    // make sure to prevent equivalent solutions to be found again
    optproblem.discardCurrentSolution();
  }
  if (isSatisfiable) {
    return OPTIMUM_FOUND;
  } else {
    return UNSATISFIABLE;
  }
} catch (ContradictionException ex) {
  assert isSatisfiable;
  // discarding a solution may launch a ContradictionException.
  return OPTIMUM_FOUND;
} catch (TimeoutException ex) {
  if (isSatisfiable)
      return UPPER_BOUND;
  return UNKNOWN;
}
```

Listing 2.4: finding an optimal solution in Sat4j

Optimization problems, implementing the `IOptimizationProblem` interface in Sat4j, contain two additional methods: `admitABetterSolution()` and `discardCurrentSolution()`. The idea is to solve optimization problems by iterating over candidate solutions until no better solution can be found. A pseudo-code explaining how optimization works in Sat4j is given in Listing 2.4.

That approach is convenient from a solver designer point of view because that way, the solver is often able to compute an upper bound of the optimal solution when it is not able to find the optimal solution. Sat4j uses a lot the decorator design pattern. The pattern is used in the `PseudoOptDecorator` class to transform any pseudo boolean satisfaction solver into a pseudo boolean optimizer. However, from a user point of view, it is not really nice to have to handle the optimization loop in the client code, so we also provide a utility class `OptToPBSatAdapter` based on the adapter design pattern to allow the end user to compute an optimal solution by a simple call to the `isSatisfiable()` method in the `IProblem` interface.

```
  private OptToPBSATAdapter optimizer;
  private DependencyHelper<Item, String> helper;

  public KnapSack(IPBSolver solver, int timeout) {
    optimizer = new OptToPBSATAdapter(new PseudoOptDecorator(solver));
    optimizer.setTimeout(timeout);
    optimizer.setVerbose(true);
    helper = new DependencyHelper<Item, String>(optimizer, false);
  }
```

Listing 2.5: Setting up the solver and the DependencyHelper object

Let go back to our Knapsack problem. The Listing 2.5 details the fields and constructor of the `KnapSack` class. It needs two fields: one for the optimizer, that will work with its own internal representation of the problem, and a `DependencyHelper` object to allow us to express

the constraints on Java objects. Line 5 defines the optimizer from a pseudo boolean satisfaction solver (`IPBSolver`) decorated by a `PseudoOptDecorator` and a `OptToPBSatAdapter`.

The aim of the `DependencyHelper` class is to provide a convenient way for the Java programmer to feed an `IPBSolver` with constraints on Java objects. Line 8 connects the optimizer and the `DependencyHelper`. As a consequence, a solution computed by the `DependencyHelper` object will be an optimal solution.

2.2.3 Some useful methods

We need to define a few other methods to complete our `Knapsack` class. They are detailed in Listing 2.6. The first one computes the value of the container. Since we minimized the negation of the value, we need to negate the value of the solution found by the solver (Line 2).

```
1  public long bestValue() {
2    return -helper.getSolutionCost().longValue();
3  }
4
5  public boolean isOptimal() {
6    return optimizer.isOptimal();
7  }
8
9  public void printStats() {
10   System.out.println(optimizer.toString(""));
11   optimizer.printStat(new PrintWriter(System.out, true), "");
12 }
```

Listing 2.6: Some useful utility methods

Since the helper class timeouts silently, we need to check is the solution found is optimal or if it is only an upper bound of the optimal solution. The optimizer can provide us that answer (Line 6). Finally, we can ask the solver to display some details about its configuration (Line 10) and some statistics about the search (Line 11).

2.2.4 Solving some randomly generated knapsack problems

It is now quite easy to solve some knapsack problems using our `KnapSack` class. The Listing 2.7 provides a simple example program to solve random knapsack problems. The first step is to create a list of items to put in a container. We can randomly generate ours to check that it works fine (Line 7). The main decision we have to take is which pseudo boolean solver we would like to use. Two families of solvers are available in Sat4j: resolution based ones and cutting planes based ones. The former usually perform well on problems containing a lot of clauses and few pseudo boolean constraints, while the latter performs better on problems containing only pseudo boolean constraints. Those solvers are available from the `org.sat4j.pb.SolverFactory` class using the `newResolution()` and `newCuttingPlanes()` static methods. Here we decided to choose a cutting planes based solver. The timeout is fixed to 500 seconds. We solve our knapsack problem with a capacity of 100 kg. We display the solution found and whether or not that solution is optimal. Then we display some information about the solver used and some statistics about the search.

The Listing 2.8 provides a sample output for that program. The randomly generated items appear first. The solver being in verbose mode, some information about upper bounds found are also displayed on the console. Here the solver required three main steps to compute the optimal solution. Note that the information from the solver provides a negative value for the objective function, while the value of the container is positive later on. The content of the container is displayed. The solver reports that the solution is optimal. The

```java
public class Main {
  public static final int N = 10;
  public static void main(String[] args) {
    Random rand = new Random();
    List<Item> items = new ArrayList<Item>(N);
    for (int i=1;i<=N;i++) {
      items.add(new Item("i"+i,rand.nextInt(50)+1,rand.nextInt(1000)+1));
    }
    System.out.println(items);
    KnapSack ks = new KnapSack(SolverFactory.newCuttingPlanes(),500);
    List<Item> sol = ks.solve(items,100);
    System.out.println(sol);
    System.out.println("Value: "+ks.bestValue());
    System.out.println("Size: "+sol.size());
    System.out.println("Optimal? "+ks.isOptimal());
    ks.printStats();
  }
}
```

Listing 2.7: Using our knapsack solving class in Java

```
[i1(8kg,479), i2(34kg,767), i3(1kg,846), i4(1kg,614), i5(9kg,760), i6(41kg,785), i7(5kg
    ,987), i8(18kg,778), i9(47kg,740), i10(23kg,874)]
c Current objective function value: -5644(0.0080s)
c Current objective function value: -6105(0.015s)
[i1(8kg,479), i2(34kg,767), i3(1kg,846), i4(1kg,614), i5(9kg,760), i7(5kg,987), i8(18kg
    ,778), i10(23kg,874)]
Value: 6105
Size: 8
Optimal? true
Optimization to Pseudo Boolean adapter
Pseudo Boolean Optimization
Cutting planes based inference (org.sat4j.pb.core.PBSolverCP)
--- Begin Solver configuration ---
...
```

Listing 2.8: Output given by our knapsack solver for 10 randomly generated items

remaining information provides the detailed setup on the engine used. Note that the fact that our optimizer is using an adapter and an optimization decorator appears in the solver description.

2.2.5 Experimenting with Sat4j

From that simple example, there are several experiments that can be done to understand the notion of NP-completeness.

1. Modify the code in Listing 2.7 to provide both the number of items and the timeout (in seconds) on the command line.
2. Run the program with different number of items (ranging from 50 to 1000 for example) and various timeout (from 5 to 500 seconds for instance). Pay attention that the capacity of the container might be either proportional to the number of initial items or randomly generated to provide more interesting results with larger set of items. Check that the solver sometimes answers optimally and sometimes not. Furthermore, when the set of items becomes bigger, the number of non optimal solutions found by the solver should also increase.
3. Sat4j allows to create a solver using the `SolverFactory.getSolverByName()` method. Modify again the main program to be able to define the solver to use on the

command line (as third parameter). The solver will be given by name, e.g. `Resolution`, `CuttingPlanes` or `Both`.

4. Try the solvers `Resolution`, `CuttingPlanes` or `Both` with various combination of number of items and timeout. You should note that cutting planes based solvers are more efficient on that specific problem. `Both` means that both a resolution and a cutting planes based solver are running in parallel. Depending of the problems, that solution may outperform the use of a specific kind of solver.

2.3 Building a PBO problem file without Sat4j

The other option is to translate our knapsack problem into a pseudo boolean optimization problem using the standard input format defined for the pseudo boolean evaluation. The main advantage of such approach is to be able to reuse a wide range of pseudo boolean optimization solvers. The main drawback is that you have to take care of the mapping between the opb file and the java objects.

2.3.1 Building the PBO file from the initial problem

A complete and precise description of the OPB format is available from the PB evaluation web site. The main features are summarized below:

- The number of variables and constraints should be given on the first line of the file. This is not an issue in our case since the number of variables equals the number of items and we have only one constraint.
- The objective function must be a minimization function. We already saw in the previous section that we simply have to negate the item values to change our maximization function into a minimization function.
- The variable names must be of the form `xDDD` where DDD is a number between 1 and the number of declared variables.
- Constraints must be either equalities or "at least" constraints. So, similarly to the objective function, we need to negate our "at most" constraint to obtain an "at least" one.

The Listing 2.9 provides the complete code for expressing our problem using the OPB format. It is a class method since it's code does not depend of instance members of the `KnapSack` class. We simple create a file whose name is given as a parameter and feed it using a `PrintStream` object. We first declare the number of variables and constraints in the header of the file. Then we declare the minimization function. Finally, we express the capacity cosntraint.

The Listing 2.10 provides a sample OPB file for the problem displayed in listing 2.8. One can check the value and the weight of the various items. There are all negated because of the rewriting of the problem. It is thus possible to solve that pseudo-boolean optimization problem with any compatible solver.

```
1 * #variable= 10 #constraint= 1
2 min:  -479 x1  -767 x2  -846 x3  -614 x4  -760 x5  -785 x6  -987 x7  -778 x8  -740 x9
      -874 x10 ;
3  -8 x1  -34 x2  -1 x3  -1 x4  -9 x5  -41 x6  -5 x7  -18 x8  -47 x9  -23 x10  >= -100 ;
```

Listing 2.10: A knapsack problem in OPB format

```
1   public static void generateOpbFile(List<Item> items, long capacity,
2       String filename) throws FileNotFoundException {
3     int nbvars = items.size();
4     int nbconstrs = 1;
5     PrintStream out = new PrintStream(new File(filename));
6     out.printf("* #variable= %d #constraint= %d\n", nbvars, nbconstrs);
7     // need to translate a maximization into a minimization
8     out.print("min: ");
9     int index = 1;
10    for (Item item : items) {
11      out.printf(" -%d x%d ", item.getValue(), index++);
12    }
13    out.println(";");
14    // need to translate an at most into an at least
15    index = 1;
16    for (Item item : items) {
17      out.printf(" -%d x%d ", item.getWeight(), index++);
18    }
19    out.printf(" >= -%d ;", capacity);
20  }
```

Listing 2.9: Generating an OPB file by hand

```
1  public class GenerateOPB {
2    public static final int N = 10;
3    public static void main(String[] args) {
4      Random rand = new Random();
5      List<Item> items = new ArrayList<Item>(N);
6      for (int i=0;i<N;i++) {
7        items.add(new Item("i"+i,rand.nextInt(50)+1,rand.nextInt(1000)+1));
8      }
9      System.out.println(items);
10     try {
11       KnapSack.generateOpbFile(items, 100, "ks.opb");
12     } catch (FileNotFoundException e) {
13       System.err.println("Cannot create file: "+e.getMessage());
14     }
15   }
16 }
```

Listing 2.11: Using the OPB generation feature

Listing 2.12 provides the command line and the output of Sat4j default SAT engine (Resolution) on that problem. Note that the optimal solution is found (Line 68), and that the optimal solution value is identical (Line 70). The "value line" (starting with a v, line 69) provides a solution. Only the items associated with positive literals in the solution (the one not prefixed by a -) should be added to the container. One can check that such solution is the same than the one found with the pure Java solution.

2.3.2 Building the OPB file with Sat4j

In many occasion, we need to convert problems expressed in Java into a more standard input format such as the one used in the various solver competitions. As such, Sat4j has a built in mechanism to encode problems described in Java in OPB files. This is the responsibility of class `OPBStringSolver`. The idea is to use that specific solver instead of a regular solver in our code. It is important that such solver receives both the various calls to `addXxxx()` to create the constraints and the call to `isSatisfiable()` to properly generate the OPB file. The latter will throw a `TimeoutException` since it cannot solve the problem. The user is expected to call the `toString()` method on that solver to retrieve the OPB file. The Listing

```
$ java -jar lib/org.sat4j.pb.jar ks.opb
c SAT4J: a SATisfiability library for Java (c) 2004-2010 Daniel Le Berre
c This is free software under the dual EPL/GNU LGPL licenses.
c See www.sat4j.org for details.
c version 2.3.0.v20110329
c java.runtime.name Java(TM) SE Runtime Environment
c java.vm.name    Java HotSpot(TM) 64-Bit Server VM
c java.vm.version 19.1-b02-334
c java.vm.vendor  Apple Inc.
c sun.arch.data.model 64
c java.version    1.6.0_24
c os.name   Mac OS X
c os.version    10.6.7
c os.arch   x86_64
c Free memory     82872800
c Max memory    129957888
c Total memory    85000192
c Number of processors  2
c Pseudo Boolean Optimization
c --- Begin Solver configuration ---
c org.sat4j.pb.constraints.
    CompetResolutionPBLongMixedWLClauseCardConstrDataStructure@299209ea
c Learn all clauses as in MiniSAT
c claDecay=0.999 varDecay=0.95 conflictBoundIncFactor=1.5 initConflictBound=100
c VSIDS like heuristics from MiniSAT using a heap lightweight component caching from
    RSAT taking into account the objective function
c Expensive reason simplification
c Armin Biere (Picosat) restarts strategy
c Glucose learned constraints deletion strategy
c timeout=2147483s
c DB Simplification allowed=false
c --- End Solver configuration ---
c solving ks.opb
c reading problem ...
c ... done. Wall clock time 0.015s.
c #vars     10
c #constraints  1
c constraints type
c org.sat4j.pb.constraints.pb.MaxWatchPbLong => 1
c SATISFIABLE
c OPTIMIZING...
c Got one! Elapsed wall clock time (in seconds):0.018
o -5644
c Got one! Elapsed wall clock time (in seconds):0.036
o -6105
c starts    : 3
c conflicts   : 5
c decisions   : 16
c propagations    : 51
c inspects    : 58
c shortcuts   : 0
c learnt literals : 1
c learnt binary clauses : 1
c learnt ternary clauses  : 1
c learnt constraints  : 3
c ignored constraints : 0
c root simplifications  : 0
c removed literals (reason simplification)  : 0
c reason swapping (by a shorter reason) : 0
c Calls to reduceDB : 0
c number of reductions to clauses (during analyze)  : 0
c number of learned constraints concerned by reduction  : 0
c number of learning phase by resolution  : 0
c number of learning phase by cutting planes  : 0
c speed (assignments/second)  : 5666.666666666667
c non guided choices  0
c learnt constraints type
c constraints type
c org.sat4j.pb.constraints.pb.MaxWatchPbLong => 1
s OPTIMUM FOUND
v x1 x2 x3 x4 x5 -x6 x7 x8 -x9 x10
c objective function=-6105
c Total wall clock time (in seconds): 0.046
```

Listing 2.12: Solving the OPB file using Sat4j on the command line

```
1  public static void generateOpbFileBis(List<Item> items, long capacity,
2      String filename) throws FileNotFoundException {
3    IPBSolver solver = new OPBStringSolver();
4    KnapSack ks = new KnapSack(solver, 500);
5    // creates the constraints in the solver
6    List<Item> result = ks.solve(items, 100);
7    // a TimeoutException has been launched by the solver,
8    // so the list should be empty
9    assert result.isEmpty();
10   PrintStream out = new PrintStream(new File(filename));
11   out.println(solver.toString());
12   out.close();
13 }
```

Listing 2.13: Generating an OPB file using OPBStringSolver

2.13 provides the alternate code for generating an OPB file, using the `solve()` method this time. Note that we need to keep track of the solver (Line 3) to be able to use the `toString()` method later (Line 11). Note that such approach works fine here because we silently return an empty list when the solver throws a `TimeoutException`.

We believe that such approach is the best way to deal with regular standard format with Sat4j:

- No need to learn the various output formats ;
- The built in input/output formats are regularly updated to allow us to participate to the competitions ;
- Some built-in helper tools such as `DependencyHelper` provide you a basic translation of logical gates into CNF.

2.3.3 Exercices

1. Modify the Listing 2.11 to generate opb files using a various number of items and various capacities. Generate various benchmarks, especially some very large ones (several thousands items).
2. Try several engine from Sat4j pseudo on those benchmarks.
3. Try several other PBO solvers on large instances (at least 1000 items):
   - bsolo (branch and bound, Linux binary)
   - SCIP (Mixed Integer Linear Programming, Branch and Cut)
   - ...

A Solution to exercices

A.1 Solutions for chapter 2

```
1  public class MainImproved {
2    public static void main(String[] args) {
3      if (args.length!=3) {
4        System.out.println("Usage: nbitems timeout enginename");
5        return;
6      }
7      int n = Integer.valueOf(args[0]);
8      int timeout = Integer.valueOf(args[1]);
9      IPBSolver solver = SolverFactory.instance().createSolverByName(args[2]);
10     Random rand = new Random();
11     List<Item> items = new ArrayList<Item>(n);
12     for (int i=0;i<n;i++) {
13       items.add(new Item("i"+i,rand.nextInt(50)+1,rand.nextInt(1000)+1));
14     }
15     System.out.println(items);
16     KnapSack ks = new KnapSack(solver,timeout);
17     List<Item> sol = ks.solve(items,n*10);
18     System.out.println(sol);
19     System.out.println("Value: "+ks.bestValue());
20     System.out.println("Size: "+sol.size());
21     System.out.println("Optimal? "+ks.isOptimal());
22     ks.printStats();
23   }
24 }
```

Listing A.1: An improved main class for the knapsack problem

B Frequently Asked Questions

B.1 General

B.1.1 How to compile Sat4j from source?

Sat4j is using Maven 3 as main build system. As such, the easiest way to build Sat4j from source is to use Maven 3. The source code of the project is available from OW2 forge using a subversion repository. We expect the reader to be familiar with those tools. Getting started guide and tutorials are available from those tools web sites.

The first step is thus to grab the source code from OW2 forge and then to build the library using maven. This is exactly what does the command line below:

```
1 svn checkout svn://svn.forge.objectweb.org/svnroot/sat4j/maven/trunk SAT4J
2 cd SAT4J
3 mvn clean package
```

After running that command, the various jar files are available in the subprojects `target` directories.

It is also possible to build Sat4j using ant.

```
1 svn checkout svn://svn.forge.objectweb.org/svnroot/sat4j/maven/trunk SAT4J
2 cd SAT4J
3 ant
```

In that case, the script builds the core and pseudo packages and make them available in `dist/CUSTOM` directory.

B.2 Sat4j Core

B.3 Sat4j Pseudo

B.4 Sat4j MaxSat

Sat4j is an open source software in Java™ developed by Daniel Le Berre and Anne Parrain, researchers in the Centre de Recherche en Informatique de Lens, at Artois University, Lens, France. Sat4j is distributed under both the Eclipse Public License and the GNU Lesser General Public License. For more information about Sat4j, visit www.sat4j.org.