Choco User Guide

Laburthe Francois, Jussien Narendra,
Rochart guillaume, Cambazard Hadrien

Creating the Problem

The central element of a choco program is the Problem object.

```
   Problem myPb = new Problem();
```

Switching from choco to palm only require to use a specific Problem : a PalmProblem.

```
   Problem myPb = new PalmProblem();
```

Variables

Actually, Three main kind of variables exist :

- IntDomainVar : It describes discrete domains where values are integers.
  - EnumIntVar : It corresponds to enumerated domains and should be used when discrete and quite small domains are needed.
  - BoundIntVar : Such domains are represented by their lower and upper bounds (propagation is only perform on the bounds). One can use them when large domains are needed.
- RealVar : It describes continous domain and use intervals to represent values.
- SetVar : It describes discrete set domains where a value of a variable is a set. Set vars are encoded with two classical bounds : the union of the all set of possible values called the envelope and the intersection of all set of possible values called the kernel.

Once the Problem has been created, variables are created through factories available on the Problem instead of the classical java constructor. The use of factories allows to redefine them in a specific Problem (as done for the PalmProblem) and ensures that constraints and variables types will remain compatible. The following example of code show how to create a finite domain variable:

```
 IntDomainVar v1 = myPb.makeEnumIntVar("var1", 1, 10);
```

v1 is an enumerated variable which is called var1 and has a discrete domain from 1 to 10. It has been created for the problem myPb.

Integer Variables

1. makeBoundIntVar(String s, int lb, int ub) : creates a finite domain variable with domain (lb .. ub), with name s.
2. makeEnumIntVar(String s, int lb, int ub) : creates a finite domain variable whose domain is approximated by bounds (lb .. ub), with name s.

The state of an IntVar can be accessed through the main following public methods on the IntVar class:

1. hasEnumeratedDomain(): checks if the variable is an enumerated or a bound one.
2. getInf(): returns the lower bound of the variable.
3. getSup(): returns the upper bound of the variable.
4. getVal(): returns the value if it is instantiated.
5. isInstantiated(): checks if the domain is reduced to a singleton.
6. canBeInstantiatedTo(int val): checks if the value val is contained in the domain of the variable.
7. getDomainSize(): returns the current size of the domain.

The domain of an IntVar can be modified through the main following public methods: Such operations are subject to the backtrack mechanism.

1. setMin(): set the lower bound of the variable.
2. setMax(): set the upper bound of the variable.
3. setVal(): set the value of the variable.

Set Variables

Set variables are still under development but a bit of api is still available.

1. makeSetVar(String name, int l, int u): creates a set domain variable with name s where l correponds to the lower bound of the inital enveloppe and b the upper bound.

The state of a SetVar can be accessed through the main following public methods on the SetVar class:

1. isInDomainKernel(int x): checks if a value x is contained in the current kernel.
2. isInDomainEnveloppe(int x): checks if a value x is contained in the current envelope.
3. getDomain(): returns the domain of the variable as a SetDomain. Iterators on envelope or kernel can than be called.
4. GetKernelDomainSize(): returns the size of the kernel.
5. GetEnveloppeDomainSize(): returns the size of the kernel.
6. GetEnveloppeInf(): returns the first available value of the envelope.
7. GetEnveloppeSup(): returns the last available value of the envelope.
8. GetKernelInf(): returns the first available value of the kernel.

9. GetKernelSup(): returns the last available value of the kernel.
10. getValue() : returns a table of integer int[] containing the current domain.

The domain of a SetVar can be modified through the main following public methods(ContradictionException can be thrown in case of an inconsistent change) :

1. setValIn(int x): set a value inside the kernel.
2. setValOut(int x): set a value outside the kernel.
3. setVal(int[] val): set the value of the variable.

Real Variables

Real variables are still under development but can be used to solve toy problems such as small systems of equations.

1. makeRealVar(String s, double lb, double ub) : creates a continous domain variable whose domain is considered as an interval [lb,ub], with name s.
2. getInf(): returns the lower bound of the variable (a double).
3. getSup(): returns the upper bound of the variable (a double).
4. isInstantiated(): checks if the domain of a variable is reduced to a canonical interval. A canonical interval indicates that the domain has reached the precision given by the user or the solver.

Constraints

Constraints are represented by dedicated objects organized in a class hierarchy. It encapsulates a filtering algorithm which are called when a propagation step occur or when external events happen on the variables belonging to the constraint, such as value removals or bounds modification.

A constraint is stated to a problem by using the method post available on the Problem object : post(Constraint c). Creating a constraint and adding it to the constraint network can be done using the Problem api. For example, adding a constraint of difference between two variables is simply written as follow:

```
myPb.post(myPb.neq(vars1, vars2));
```

Integer constraints

Basic constraints

The constraints available with choco are arithmetic constraints (equality, difference, comparisons and linear combination), user-defined binary constraints (AC4,AC3, ...), boolean operators (or, and, implies, ...).

The simplest constraints are comparisons which are defined over expressions of variables such as linear combinations. The following comparison constraints can be accessed through the Problem API:

1. neq(IntExp v1, IntExp v2) : v1 != v2.
2. eq(IntExp v1, IntExp v2) : v1 = v2.
3. leq(IntExp v1, IntExp v2) : v1 <= v2.
4. lt(IntExp v1, IntExp v2) : v1 < v2.

To construct complex expressions of variables, simple operators can be used:

1. minus(IntExp exp1, IntExp exp2): exp1 - exp2.
2. plus(IntExp exp1, IntExp exp2): exp1 + exp2.
3. mult(int coef, IntExp exp): coef * exp.
4. scalar(int[] coef, IntVar[] vars): coef[1]*vars[1] + ... + coef[n]*vars[n].
5. sum(IntVar[] vars): vars[1] + ... + vars[n].

User-defined constraints

Choco supports the statement of constraints defined by arbitrary relations. It offers the possibility of stating binary constraints with several AC Algorithm and also n-ary constraints with a weaker form of propagation.

Binary constraints

The relation defines feasible or infeasible pairs of values for the two variables involved in the constraint. Relations may be defined by two means:

1. Tables : specifying those pairs of value for which the constraints is satisfied/unsatisfied.
2. Predicates : specfying the method to be called in order to check whether a pair of value is feasible or not.

One the one hand constraints based on tables of values may become rather memory consuming in case of large domains, although relation tables may be shared by different constraints. On the other hand, predicate constraints require little memory as they do not cache thruth values, but imply some run-time overhead for calling the feasibility test. Table constraints are thus well suited for constraints over small domains; while predicate constraints are well suited for situations with large domains.

The creation of a constraint for a relation can be done through the following Problem API :

1. makePairAC(IntVar v1, IntVar v2, ArrayList pairs, boolean feas, int ac).
2. makePairAC(IntVar v1, IntVar v2, boolean[][] pairs, boolean feas, int ac).
3. relationPairAC(IntVar v1, IntVar v2, BinRelation pairs, int ac).

Parameter feas indicates whether the relation models feasible pairs of values or infeasible one (default is infeasible). Parameters pairs contains the definition of the relation. A list of int[] of size 2 in the first case, a boolean[][] in the second case or as a BinRelation in the last case. Finally parameter ac selects the algorithm for enforcing arc-consistency (default ac = 2001). Supported values for this parameter are :

1. 3 for AC3 algorithm (searching from scratch for supports on all values)
2. 4 for AC4 algorithm (maintaining a count of supports for each value)
3. 2001 for the AC2001 algorithm (maintaining the current support of each value)

The definition of a binary relation based on a predicate can be done by inheriting from the CouplesTest class. Have a look on the following example :

```
public class MyInequality extends CouplesTest{
  public boolean checkCouple(int x, int y) {
    return x != y;
  }
}
```

You can then state a constraint as the following :

```
pb.post(pb.relationPairAC(v1, v2,new MyInequality()));
```

The complete Problem API allow to easily create binary constraint :

1. infeasPairAC(IntVar v1, IntVar v2, ArrayList pairs)
2. feasPairAC(IntVar v1, IntVar v2, ArrayList pairs)
3. relationPairAC(IntVar v1, IntVar v2, BinRelation binR)
4. ...

N-Ary Constraints

The situations for binary constraints is extended to the case of relations involving more than two variables, upto a significant difference from the propagation point of view: The propagation engine maintains arc-consistency for binary constraints throughout the solving process, while for n-ary constraints, it uses a weaker propagation mechanism with a forward checking algorithm.

The API for creating such constraints is the following ones:

1. makeTupleFC(IntVar[] vs, ArrayList tuples, boolean feas)
2. relationTuple(IntVar[] vs, LargeRelation rela)

Defining a specific n-ary relation without storing the tuples can be done as on the following example :

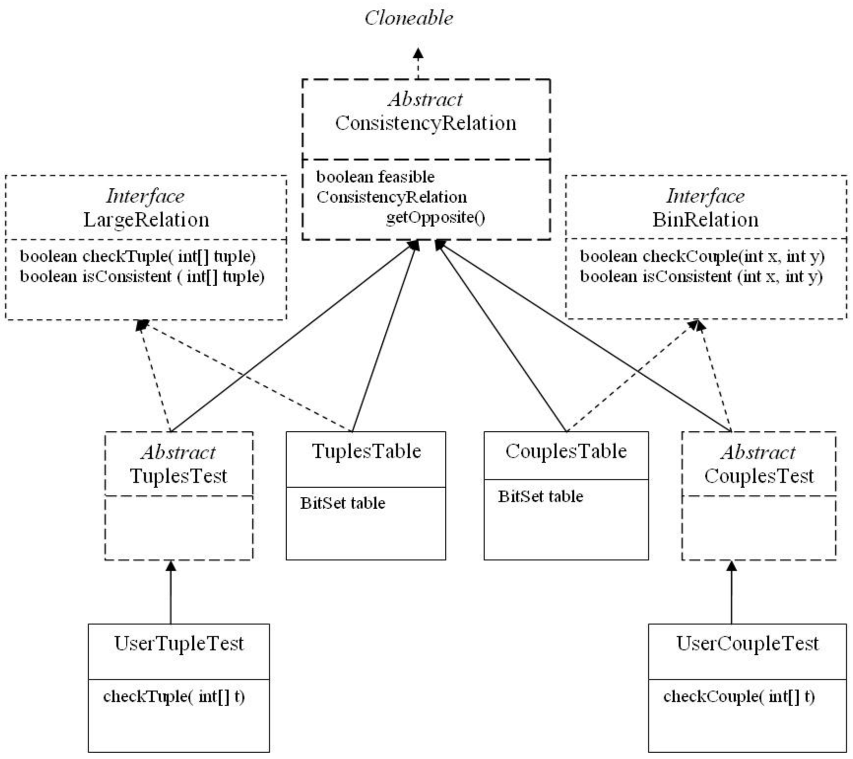
```
 public class NotAllEqual extends TuplesTest {

       public boolean checkTuple(int[] tuple) {
          for (int i = 1; i < tuple.length; i++) {
             if (tuple[i - 1] != tuple[i]) return true;
          }
          return false;
       }
```

```
   }
```

Otherwise, the tuples are given in an ArrayList as int[] table given the compatible/incompatible values. One can then state the constraint to the problem :

```
pb.post(pb.relationTuple(new IntVar[]{x, y, z}, new NotAllEqual()))
```

Finally, the structure of the Consistency relations can be seen in more details on the following picture :



Advanced constraints

Choco includes several global constraints. Those constraints allows to filter efficiently some inconsistant values. For instance, if several variables should be affected to different values, using a global constraint can offer some additionnal filtering rules (for instance a should be in [1,4], b in [1,4], c in [3,4] and d in [3,4], then one can deduce that a and b cannot be instantiated to 3 or 4; such rule cannot be infered by simple binary constraints).

Here are described som of those constraints :

1. pb.allDifferent(IntVar[] vars) creates a constraint ensuring that all pairs of variable have distinct values (which is useful for some matching problems);
2. pb.occurrence(IntVar[] vars, int value, IntVar occurrence) creates a constraint to

ensure that occurrence will be instantiated to the number of occurrences of valu ein the list of variables vars; this is a specilisation of the following constraint;

3. pb.globalCardinality(IntVar[] vars, int[] low, int[] up) creates a constraint ensuring that the number of occurrences of the value 1 in all the variables vars is between low[0] and up[0], and generallay the number of occurrences of the value i in vars is between low[i-1] and up[i-1].

Some other global constraints can be added to Choco in future releases. One can find all the API and constraints available on the Javadoc API.

Set constraints

Choco supports the statement of constraints among sets :

```
pb = new Problem();
pb.post(mod.eqCard(vars[i],3));
```

The following set constraints are available :

1. member(SetVar sv1, int val): states that the variable sv1 contains value val.
2. notMember(SetVar sv1, int val): states that the variable sv1 does not contain value val.
3. setDisjoint(SetVar sv1, SetVar sv2): states that *sv1* and *sv2* are disjoint sets ; e.g. that *sv1* and *sv2* contain no common values.
4. setInter(SetVar sv1, SetVar sv2, SetVar inter): states that the inter set variable is the intersection of set variables *sv1* and *sv2* ; e.g. states that inter contains exactly those values contained in both sets *sv1* and *sv2.*
5. eqCard(SetVar sv, int val): states that the cardinality of the set variable is equal to value val.
6. geqCard(SetVar sv, int val): states that the cardinality of the set variable is greater or equal equal than value val.
7. leqCard(SetVar sv, int val): states that the cardinality of the set variable is equal than value val.

To deal with integer variables, the following mixed constraint are available :

1. member(SetVar sv1, IntVar var).
2. notMember(SetVar sv1, IntVar var).
3. eqCard(SetVar sv, IntVar iv).
4. geqCard(SetVar sv, IntVar iv).
5. leqCard(SetVar sv, IntVar iv).

Real constraints

Search and Branching

1. Search for one or all solutions

Once the problem is modelled and created thanks to the API described in previous sections, one may want to solve it ! If only one solution is needed, this is quite easy. The following code solves the problem and displays the solution:

```
if (pb.solve() == Boolean.TRUE) {
    for(int i = 0; i < pb.getNbIntVars(); i++) {
        System.out.println(pb.getIntVar(i) + " = " + ((IntDomainVar)
pb.getIntVar(i)).getVal());
    }
}
```

Notice that the *solve()* method returns a Boolean object instead of a primitive boolean because its value may be null meaning that a limit has been reached. If one wants several solutions, the incremental solve API can be used : *nextSolution* search for another solution in the search tree :

```
if (pb.solve() == Boolean.TRUE) {
    do {
        for(int i = 0; i < pb.getNbIntVars(); i ++) {
            System.out.println(pb.getIntVar(i) + " = " + ((IntDomainVar)
pb.getIntVar(i)).getVal());
        }
    } while(pb.nextSolution() == Boolean.TRUE);
}
```

a. The Solver

The *Problem* is the central element of a choco model as it allows the creation of variables and constraints. But the control of the search process without using predefined tools is made on the *Solver* class. The following code will do exactly what you get by calling the solve() method but you may access this time to the solver once it has been generated to parameterize it in more details. We will use this piece of code in section 3 and 4 to show how the search space can be controlled in details.

```
Solver s = pb.getSolver();
s.setFirstSolution(true);
s.generateSearchSolver(pb);
// insert here the code to parameterize the solver in details (adding goals, new limits,
//etc …)
s.launch();
Boolean isFeasible = pb.isFeasible();
```

2. *Optimization*

Optimization is done in Choco according to a variable denoting the objective value. The objective function is then expressed as a constraint over this variable and the rest of the problem. The API concerning optimization proposes to minimize/maximize this objective variable (instead of a call to pb.solve()) :

1. minimize(IntVar obj, boolean restart).

2. maximize(IntVar obj, boolean restart).

Parameter restart is a boolean indicating whether the solver will restart the search after each solution found or if it will keep backtracking from the leaf of the last solution found.

Look at the following knapsack example where a scalar product over three variables is maximized :

```
Problem pb = new Problem();
IntVar obj1 = pb.makeEnumIntVar("obj1",0,7);
IntVar obj2 = pb.makeEnumIntVar("obj1",0,5);
IntVar obj3 = pb.makeEnumIntVar("obj1",0,3);
IntVar cost = pb.makeBoundIntVar("cout",0,1000000);
int capacite = 34;
int[] volumes = new int[]{7,5,3};
int[] energie = new int[]{6,4,2};
// capacity constraint
pb.post(pb.leq(pb.scalar(volumes,new IntVar[]{obj1,obj2,obj3}),capacite));

// objective function
pb.post(pb.eq(pb.scalar(energie,new IntVar[]{obj1,obj2,obj3}),cost));

pb.maximize(c,false);
```

3. *Limiting the search space*

Limits may be imposed on the search algorithm to avoid spending too much time in the exploration. The limits are updated and checked each time a new node is created. The API is given on the Solver class:

1. *setTimeLimit(int timeLimit):* stops the search algorithm after timeLimit milliseconds have been spent searching.
2. *setNodeLimit(int nodeLimit):* stops the search algorithm after nodeLimit nodes have been expanded.

For example, to stop the search after 30 seconds, just add the following line ((before a call to pb.solve()):

```
pb.getSolver().setTimeLimit(30000);
```

To define your own limits/statistics (notice that a limit object can be used only to get statistics about the search), you can create a limit object by extending the *AbstractGlobalSearchLimit* class or implementing directly the interface *IGlobalSearchLimit*. Limits are managed at each node of the tree search and are updated each time a node is open or closed. Notice that limits are therefore time consuming. Implementing its own limit need only to specify to the following interface :

```
/**
```

```
 * The interface of objects limiting the global search exploration
 */
public interface IGlobalSearchLimit extends Entity {
  /**
   * resets the limit (the counter run from now on)
   * @param first true for the very first initialization, false for subsequent ones
   */
  public void reset(boolean first);

  /**
   * notify the limit object whenever a new node is created in the search tree
   * @param solver the controller of the search exploration, managing the limit
   * @return true if the limit accepts the creation of the new node, false otherwise
   */
  public boolean newNode(AbstractGlobalSearchSolver solver);

  /**
   * notify the limit object whenever the search closes a node in the search tree
   * @param solver the controller of the search exploration, managing the limit
   * @return true if the limit accepts the death of the new node, false otherwise
   */
  public boolean endNode(AbstractGlobalSearchSolver solver);
}
```

Look at the following example to see a concrete implementation of the previous interface. We define here a limit on the depth of the search (which is not found by default in choco). *The getWorldIndex()* is used to get the current world, i.e the current depth of the search or the number of choices which have been done from *baseWorld*.

```
public class DepthLimit extends AbstractGlobalSearchLimit {

  public DepthLimit(AbstractGlobalSearchSolver theSolver, int theLimit) {
    super(theSolver,theLimit);
    unit = "deep";
  }

  public boolean newNode(AbstractGlobalSearchSolver solver) {
        nb = Math.max(nb, this.getProblem().getWorldIndex() –
                this.getProblem().getSolver().getSearchSolver().baseWorld);
        return (nb < nbMax);
  }

  public boolean endNode(AbstractGlobalSearchSolver solver) {
        return true;
  }

  public void reset(boolean first) {
        if (first) {
           nbTot = 0;
        } else {
```

```
        nbTot = Math.max(nbTot, nb);
      }
      nb = 0;
 }
```

Once you have implemented your own limit, you need to tell the search solver to take it into account. Instead of using a call to the *solve()* method*,* you have to create the search solver by yourself and add the limit to its limits list such as in the following code :

```
Solver s = pb.getSolver();
s.setFirstSolution(true);
s.generateSearchSolver(pb);
s.getSearchSolver().limits.add(new DepthLimit(s.getSearchSolver(),10));
s.launch();
```

4. Define your own tree search

A key ingredient of any constraint approach is a clever branching strategy. The construction of the search tree is done according to a serie of *Branching* objects (that plays the role of achieving intermediate goals in logic programming). The user may specify the sequence of branching objects to be used to build the search tree. We will first present in this section how to define your own branching object and how to compose it with other goals. We will start with a very simple and common way to branch by choosing values for variables and specially how to define its own variable/value selection strategy. We will then focus on more complex branching such as dichotomic or n-ary choices. Finally we will show how to control the search space in more details with well known strategy such as LDS (Limited discrepancy search).

a. Building a sequence of branching objects

Adding a new goal is made through the problem solver *(Solver s = pb.getSolver())* with the *addGoal(AbstractBranching b)* method of the solver. As for the addition of your own limit, don't call the *solve()* method but instead, build the solver by yourself add your sequence of branching and call the *launch()* method of the solver. The following example add three branching objects on integer variables vars1, vars2 and set variables svars. The first two branching are both *AssignVar* (see next section) but uses two different variable/values selection strategies:

```
Solver s = pb.getSolver();
pb.getSolver().generateSearchSolver(pb);
s.attachGoal(new AssignVar(new MinDom(pb,vars1), new IncreasingDomain()));
s.addGoal(new AssignVar(new DomOverDeg(pb,vars2),new DecreasingDomain());
s.addGoal(new AssignSetVar(new MinDomSet(pb,svars), new MinEnv(pb)));
s.launch();
```

b. Variable/value selection for AssignVar branching

Choco provides means of composing search trees by specifying the heuristics used for selecting variables and values on integer variables in case of *AssignVar* branchings. An AssignVar branching takes as input the variable and value selection strategy which are based on the following interfaces:

1. *IIntVarSelector* : Interface for the variable selection
2. *IValIterator* : Interface that provide a way of describing an iteration scheme on the domain of a variable
3. *IValSelector* : Interface for a value selection

Default branching heuristics

The default branching heuristic used by Choco is to choose the variable with current minimum domain size first and to take its values in increasing order. The default branchings currently supported by choco are available in the packages **choco.integer.search** for integer variables (**choco.set.search** for set variables). Concrete examples of the previous interfaces are the classes *MinDomain, MostConstrained, DomOverDeg, RandomIntVarSelector* ...

If you only want to use one single goal but with customized value and variable heuristics, you can use the API available on the *Solver* class (before calling the *solve()* method) as shown on the following example :

```
pb.getSolver().setVarSelector(new RandomIntVarSelector(pb));
```

Changing the values enumeration/selection can be done in the same way:

```
// select the value in increasing order
pb.getSolver().setValIterator(new DecreasingDomain());
// or select a random value
pb.getSolver().setValSelector(new RandomIntValSelector());
```

How to define it own variable selection

You may extend this small library of branching schemes and heuristics by defining your own concrete classes of *IIntVarSelector*, *IValIterator* and *IValSelector*.

We give here an example of an *IntVarSelector* with the implementation of a static variable ordering :

```
public class StaticVarOrder implements IIntVarSelector {
/**
* the sequence of variables that need be instantiated
*/
protected IntDomainVar[] vars;

public StaticVarOrder(IntVar[] vars) {
        this.vars = vars;
}

public IntDomainVar selectIntVar() {
	for (int i = 0; i < vars.length; i++) {
		if (!vars[i].isInstantiated()) {
			return vars[i];
		}
	}
	return null;
}
}
```

Notice on this example that you only need to implement the method *selectIntVar* which belongs to the contract of *IIntVarSelector*. Once the branching is finished, it returns *null* and the next branching (if one exists) is taken by the search algorithm to continue the search, otherwise, the search stops as all variable are instantiated. To avoid the loop over the variables of the branching, a backtrackable integer (StoredInt) could be used to remember the last instantiated variable and to directly select the next one in the table. Notice that backtrackable structures could be used in any of the code presented in this chapter to speedup the computation of dynamic choices.

c. Beyond Variable/value selection, how to define its own Branching object

A branching object is based on the *IntBranching* interface where each alternative is labelled with an integer.

```
/**
 * IntBranching objects are specific branching objects where each branch is labelled with an integer.
 * This is typically useful for choice points in search trees
 */
public interface IntBranching extends Branching {

/**
   * selecting the object under scrutiny (that object on which an alternative will be set)
   * @return the object on which an alternative will be set (often  a variable)
   */
   public Object selectBranchingObject();

  /**
   * performs the action, so that we go down a branch from the current choice point
   * @param x the object on which the alternative is set
   * @param i the label of the branch that we want to go down
   */
  public void goDownBranch(Object x, int i) throws ContradictionException;

  /**
   * performs the action, so that we go down up the current branch to the father choice point
   * @param x the object on which the alternative has been set at the father choice point
   * @param i the label of the branch that has been travelled down from the father choice point
   */
   public void goUpBranch(Object x, int i) throws ContradictionException;

  /**
   * Computes the search index of the first branch of the choice point
   * @param x the object on which the alternative is set
   * @return the index of the first branch
   */
   public int getFirstBranch(Object x);

  /**
   * Computes the search index of the next branch of the choice point
   * @param x the object on which the alternative is set
   * @param i the index of the current branch
   * @return the index of the next branch
   */
   public int getNextBranch(Object x, int i);

  /**
   * Checks whether all branches have already been explored at the current choice point
   * @param x the object on which the alternative is set
   * @param i the index of the last branch
   * @return true if no more branches can be generated
   */
   public boolean finishedBranching(Object x, int i);
```

The AssignVar branching typically implements the computation of the branching object (*selectBranchingObject()*) by delegating it to its VarSelector as well as first and next branch computation are delegated to its Value Selector or Iterator. So in this case, the value chosen is used to label the branch. Finally the AssignVar branching simply implements the *goDownBranch(Object x, i)* by assigning the value *i* to variable *x* (cast in this case as an *IntVar*) and propagating this choice :

```
public void goDownBranch(Object x, int i) throws ContradictionException {
   (IntDomainVar) x.setVal(i);
   manager.problem.propagate();
 }

 public void goUpBranch(Object x, int i) throws ContradictionException {
   ((IntDomainVar) x).remVal(i);
   manager.problem.propagate();
 }
```

Notice that in this implementation, we choose to propagate the removal of a value when this value has proved to lead to a failure. We could have chosen to leave the *goUpBranch(Object x, int i)* empty and to keep branching with the next values.

An example of a dichotomic branching

In the case of a dichotomic branching, we use a predefined AbstractBinIntBranching where there is only two alternatives at each choice point. So no value selection strategy is used and the branch selection is simply implemented as :

```
public abstract class AbstractBinIntBranching extends AbstractIntBranching {
  public int getFirstBranch(Object x) {
    return 1;
  }

  public int getNextBranch(Object x, int i) {
    return 2;
  }

  public boolean finishedBranching(Object x, int i) {
    return (i == 2);
  }
}
```

On this basis, *goDownBranch(Object x, int i)* is the single method that needs to be implemented. It computes the middle point of the domain and branches first on the right side by updating the lower bound of the variable and then on the left side by updating the upper bound.

```
/**
 * A dichotomic branching example
 */
public class DichotomicBranching extends AbstractBinIntBranching implements
IntBranching {

    protected IVarSelector varSelector;

    public DichotomicBranching(IVarSelector varSel) {
        this.varSelector = varSel;
    }

   /**
    * delegates to the var selector the choice of the  branching variable
    */
    public Object selectBranchingObject() {
        return varSelector.selectVar();
    }


    public void goDownBranch(Object x, int i) throws ContradictionException {
        IntDomainVar var = (IntDomainVar) x;
        // we compute the bound to split
        int bound = (var.getSup() - var.getInf()) / 2 + var.getInf() + 1;

        switch (i) {
           case 1: {
              var.setInf(bound);
              manager.problem.propagate();break;
           }
           case 2: {
              var.setSup(bound - 1);
              manager.problem.propagate();break;
           }
        }
    }

}
```

Branching by posting constraints

You can easily implement complex branching by posting a constraint instead of acting on the domain of a variable as we only did at that time. It is indeed useful to implement n-ary branching (on more than one variable). Just for example, the previous code can be changed by the following where a “greater than equal” and “less than equal” constraint are posted instead of calling the *updateInf* and *updateSup* methods :

```
public void goDownBranch(Object x, int i) throws ContradictionException {
        IntDomainVar var = (IntDomainVar) x;
        int bound = (var.getSup() - var.getInf()) / 2 + var.getInf() + 1;

        switch (i) {
           case 1: {
              manager.problem.post(manager.problem.geq(var, bound));
              manager.problem.propagate();break;
           }
           case 2: {
              manager.problem.post(manager.problem.leq(var, bound-1));
              manager.problem.propagate();break ;
           }
        }
    }
```

d. Building a limited tree search

LDS (Limited discrepancy search)

Harvey and Ginsberg describe an interesting tree search algorithm called limited discrepancy search (LDS) that exploits the existence of good heuristics. LDS is based on the assumption that a solution constructed according to a good value-ordering heuristic is unlikely to contain many mistakes. For a path in the search tree, the number of nodes where the chosen value differs from that specified by the value-ordering heuristic is called the *discrepancy count*. When LDS is forced to backtrack, it examines new paths in increasing order of the discrepancy count.

A tree search with a fixed limited discrepancy according to a given branching can be implemented as a specific Branching which maintains the number of discrepancy of the current path using a backtrackable integer. All methods are delegated to the branching which is limited (attribute *delegate*) except the *getNextBranch* and *finishedBranching* which counts the discrepancy and check it does not violates the limit allowed. The *LimitedSearch* class constitutes a kind of meta-branching which applies a limited discrepancy search to its *delegate* branching.

```
public class LimitedSearch extends AbstractIntBranching implements IntBranching {

   IntBranching delegate;
   IStateInt discrepancyCount;
   int maxDiscrepancy;

   public LimitedSearch(Problem pb, IntBranching delegate, int nbViolsMax) {
      this.delegate = delegate;
      discrepancyCount = new StoredInt(pb.getEnvironment(), 0);
      maxDiscrepancy = nbViolsMax;
   }

   public int getNextBranch(Object x, int i) {
      discrepancyCount.add(1);
      return delegate.getNextBranch(x, i);
   }

   public boolean finishedBranching(Object x, int i) {
      if (discrepancyCount.get() < maxDiscrepancy)
         return delegate.finishedBranching(x, i);
      return true;
   }

   public Object selectBranchingObject() {
      return delegate.selectBranchingObject();
   }

   public int getFirstBranch(Object x) {
      return delegate.getFirstBranch(x);
   }

   public void goDownBranch(Object x, int i) throws ContradictionException {
      delegate.goDownBranch(x, i);
   }

   public void goUpBranch(Object x, int i) throws ContradictionException {
      delegate.goUpBranch(x, i);
   }

}
```

Limited Depth first search

Another way to limit the tree search exploration is to limit the depth of the tree. Once the limited depth is reached, we want the solver to branch according to the heuristic without backtracking at all. Every methods of the *IntBranching* class is simply implemented by delegating it to its *delegate* attribut. The only method to change is the *finishedBranching* which returns true as soon as the limit of depth has been reached. Indeed, As for the LDS above, after this point, no alternatives will be try as the branching always tells the solver it is finished without asking the *delegate*.

```
public class DepthLimitedSearch extends AbstractIntBranching implements
IntBranching {
   IntBranching delegate;
   int maxDepth;

   public DepthLimitedSearch (IntBranching delegate, int maxDepth) {
      this.delegate = delegate;
      this.maxDepth = maxDepth;
   }

   public boolean finishedBranching(Object x, int i) {
        int startDepth = this.getProblem().getSolver().getSearchSolver().baseWorld;
        if (manager.problem.getWorldIndex() -  startDepth < maxDepth)
          return delegate.finishedBranching(x, i);
        return true;
   }

  public Object selectBranchingObject() {
      return delegate.selectBranchingObject();
   }

   public int getFirstBranch(Object x) {
      return delegate.getFirstBranch(x);
```

```
    }

    public int getNextBranch(Object x, int i) {
       return delegate.getNextBranch(x, i);
    }

    public void goDownBranch(Object x, int i) throws ContradictionException {
       delegate.goDownBranch(x, i);
    }

    public void goUpBranch(Object x, int i) throws ContradictionException {
       delegate.goUpBranch(x, i);
    }
}
```

How to create its own constraint

The constraint hierarchy

A constraint must respect the interface Constraint to be handled by Choco. As a lot of low level methods (to manage the constraint network using a system of listeners) are included in this interface, an abstract class AbstractConstraint provide an implementation of all listeners.

For integer variables, the easiest way to implement your own constraint is to inherit from one of the following classes :

- AbstractUnIntConstraint, AbstractBinIntConstraint, AbstractTernIntConstraint : A default implementation for constraint stating one, two or three variables.
- AbstractLargeIntConstraint : A default implementation for constraint stating more than 3 variables.

In the same way, SetConstraint can inherit from :

- AbstractUnSetConstraint, AbstractBinSetConstraint, AbstractTernSetConstraint.
- AbstractLargeSetConstraint.

Moreover, Constraints stating on integer and set variables can be written by inheriting from AbstractMixedConstraint.

Backtrackable structures

Updating domains of variable (indexes and links with propagation)

The event system handled by the propagation mechanism (constawake, asynchronous)