CSP4J: a black-box CSP solving API for Java

http://cspfj.sourceforge.net/

Julien Vion

CRIL-CNRS FRE 2499,
Université d'Artois
Lens, France
vion@cril.univ-artois.fr

Abstract. We propose an API, namely CSP4J for *Constraint Satisfaction Problem for Java*, that aims to solve a CSP problem part of any Java application. CSP4J is distributed online using the LGPL license [17]. We intend our API to be a "black box", i.e. to be able to solve any problem without tuning parameters or programming complex constraints. We intend CSP4J to move towards the Graal of AI: the ability to solve any problem in a reasonable time with a minimal expertise from the user.

1 Introduction

Many problems arising in the computing industry involve constraint satisfaction as an essential component. Such problems occur in numerous domains such as scheduling, planning, molecular biology and circuit design. Problems involving constraints are usually NP-Complete and need, if able, powerful Artificial Intelligence techniques to be solved in reasonable time. Problems involving constraints are usually represented by so-called constraint networks. A constraint network is simply composed of a set of variables and of a set of constraints. Finding a solution to a constraint network involves assigning a value to each variable such that all constraints are satisfied. The Constraint Satisfaction Problem (CSP) is the task to determine whether or not a given constraint network, also called CSP instance, is satisfiable. The Maximal Constraint Satisfaction Problem (Max-CSP) is the task to find a solution that satisfies as much constraints as possible, and eventually proving that a given solution is optimal, i.e. no other solution exists that can satisfy more constraints than the given one.

CSP4J has been in development since 2005 and is quickly acquiring maturity. We intend our API to be a "black box" solving CSP and Max-CSP. Given this assumption, CSP4J does not focus on problem-specific global constraints, although the Object design of CSP4J permits to develop such constraints. For example, CSP4J is shipped with the well known "all-different" global constraint including a simple specific propagator.

CSP4J proposes powerful engines based on the latest refinements of current research in AI.

– **MGAC**, a complete solver based on the well known MGAC-$dom/wdeg$ algorithm [14]. It can solve any CSP in a complete way: if given enough time, a feasible solution, if exists, will be found. If no solution exists, this engine is able to prove it.

– **MCRW**, an incomplete local search solver based on the *Min-Conflicts Hill-Climbing with Random Walks* algorithm [12]. This engine can be used to solve optimization problems that can be formalized as a Max-CSP problem in an "anytime" way: the algorithm can be stopped after a given amount of time, and the best solution found so far will be given.
– **Tabu**, an incomplete local search solver performing a Tabu search [4]. **Tabu** have similar characteristics as **MCRW**.
– **WMC**, an incomplete local search solver based on the *Breakout Method* [13], that show similar characteristics as **MCRW** and **Tabu**, although not really suited for Max-CSP problems.
– **Combo**, a complete solver based on the hybridization of MGAC-$dom/wdeg$ with WMC [20].

In order to prove the interest of our library, we developed a few test applications, all distributed online using the GPL license [16]. One of these test applications is dedicated to participate to the International CSP Solver Competitions, and tries to solve problems delivered under the XCSP 2.0 format [1]. This solver participated to the two first International CSP Solver Competitions. This "competitor" version of CSP4J is shipped with a particular constraint called "Predicate Constraint", that compiles intentional constraints as defined by the XCSP 2.0 format.

Other example applications include :

– a random problem generator and solver, which is very useful to benchmark algorithms and computers,
– a Minimal Unsatisfiable Core (MUC) extractor, able to extract a minimally unsatisfiable set of variable and constraints from a larger incoherent CSP,
– an Open-Shop solver, able to find feasible and optimal solutions to Open-Shop problems
– last but not least, a Sudoku solver

2 Solving a CSP in a black-box

In order to be able to solve any kind of problem, CSP4J focuses on two main topics: genericity and flexibility. Flexibility was obtained by the choice of an object-oriented language for its development: Java 5. The object-oriented conception of CSP4J permits to model problems using a fully object-oriented scheme.

A few classes and interfaces are in the heart of CSP4J, as described by the UML diagram on Figure 1: The $Problem$, $Variable$ and $Constraint$ classes define a CSP instance. The Solver interface is implemented by all engines provided with CSP4J.

The $Variable$ class: It can be used directly through its constructor. $domain$ simply contains the domain of the variable (i.e. the set of value the variable can take its value in) under the form of an array of integers.

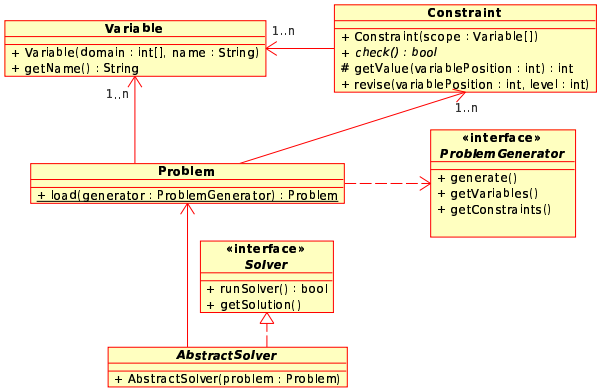

Fig. 1. UML sketch of CSP4J

```
public final class DTPConstraint extends Constraint {

  final private int duration0;
  final private int duration1;

  public DTPConstraint(final Variable[] scope,
      final int duration0, final int duration1) {
    super(scope);
    this.duration0 = duration0;
    this.duration1 = duration1;
  }

  @Override
  public boolean check() {
    final int value0 = getValue(0);
    final int value1 = getValue(1);

    return (value0 + duration0 < value1
      || value1 + duration1 < value0);
  }

}
```

Listing 1.1. The Disjunctive Temporal Constraint

```
final Predicate predicate = new Predicate();
predicate.setExpression("(X0 + X1 < X2) || (X2 + X3 < X0)");
predicate.setParameters("int X0 int X1 int X2 int X3");

final PredicateConstraint dtpConstraint =
  new PredicateConstraint(scope, predicate);
dtpConstraint.setParameters(scope[0].getName() + " " + duration0
  + " " + scope[1].getName() + " " + duration1);
try {
  dtpConstraint.compileParameters();
} catch (FailedGenerationException e) {
  System.err.println("Failed to compile constraint");
  System.exit(1);
}
```

Listing 1.2. Defining a DT Constraint with predicates

The $Constraint$ class . It consists of an abstract class that must be extended to define the constraints that define the problem. In particular, the abstract method $check()$ must be overridden. $check()$ must return whether the current tuple is allowed by the constraint. The current tuple is accessible through the $getValue(int\ variablePosition)$ method, $variablePosition$ corresponding to the position of the variable in the constraint, as defined by the *scope* in the constructor. Listing 1.1 gives an example on how to easily define a constraint. Alternatively, one could use the $PredicateConstraint$ to define such a constraint as shown on Listing 1.2. Notice, however, that source code from $PredicateConstraint$ is released amongst the Competitor test application for CSP4J under the GPL, and not directly with the CSP4J API.

If desired, one may also override the $revise(int\ variablePosition, int\ level)$ method in order to develop constraint-specific propagators. If not, a revision using the AC3rm algorithm (see section 3.1) is done.

The $Problem$ class: It defines a CSP. The $ProblemGenerator$ interface permits to define classes that will be intended to generate problems to solve. To define a problem to be solved with CSP4J, one has to implement the ProblemGenerator interface. An instance of the problem is then loaded by calling the static method $Problem.load(ProblemGenerator)$. The $ProblemGenerator$ interface only defines three methods.

- $generate()$: this method is called upon loading of the Problem, it can be used to create constraints and variables
- $Collection\langle Variable\rangle\ getVariables()$: this method must return the set of variables that defines the problem
- $Collection\langle Constraint\rangle\ getConstraints()$: this method must return the set of constraints that defines the problem

The $Solver$ interface and the $AbstractSolver$ helper class: These permit to define additional engines for CSP4J. The MGAC and MCRW engines that come with CSP4J

Algorithm 1: revise-rm(X: Variable): Boolean

```
1  domainSize ← |dom(X)|
2  foreach C | X ∈ vars(C) do
3      foreach v ∈ dom(X) do
4          if supp[C, X, v] is valid then continue
5          tuple ← seekSupport(C, X_v)
6          if tuple = ⊤ then remove v from dom(X)
7          else
8              foreach Y ∈ vars(C) do
9                  supp[C, Y, tuple[Y]] ← tuple
               /* for wdeg:                                   */
10             if dom(X) = ∅ then wght[C]++
11 return domainSize ≠ |dom(X)|
```

Algorithm 2: GAC3rm ($P = (\mathscr{X}, \mathscr{C})$: CN)

```
1  Q ← 𝒳
2  while Q ≠ ∅ do
3      pick X from Q
4      foreach Y ∈ 𝒳 | ∃C ∈ 𝒞 | X ∈ C ∧ Y ∈ C ∧ X ≠ Y do
5          if revise-rm(Y) then
6              if dom(Y) = ∅ then return false
7              Q ← Q ∪ Y
```

are classes that extends $AbstractSolver$. The $runSolver()$ method launches the resolution and returns **true** if the problem is satisfiable and **false** if it is not. The method $getSolution()$ returns the last found solution (the best solution found so far for Max-CSP). To use CSP4J as an incomplete Max-CSP solver, one has to launch $runSolver()$ from a thread to control its execution.

To illustrate how CSP4J can be used in a Java application, Listing 1.3 defines the well-known Pigeons problem, using a clique of $different$ constraints defined as predicates. Once the problem has been defined and loaded, the solving process can be launched in a few lines of code, as shown on Listing 1.4.

3 Under the hood

3.1 The MGAC engine

Generalized Arc Consistency guarantees the existence of a support of each value in each constraint. Establishing Generalized Arc Consistency on a given network P involves removing all generalized arc inconsistent values.

Many algorithms establishing Arc Consistency have been proposed in the literature. We believe that GAC3rm [9] is a very efficient and robust one. GAC3rm is a refine-

```
public class Pigeons implements ProblemGenerator {
  final private int size;
  final private List<Variable> variables;
  final private Collection<Constraint> constraints;
  final private Predicate predicate;

  public Pigeons(int size) {
    this.size = size;
    variables = new ArrayList<Variable>(size);
    constraints = new ArrayList<Constraint>();
    predicate = new Predicate();
    predicate.setExpression("X0 != X1");
    predicate.setParameters("int X0 int X1");
  }

  public void generate() throws FailedGenerationException {
    final int[] domain = new int[size - 1];
    for (int i = size - 1; --i >= 0;) { domain[i] = i; }
    for (int i = size; --i >= 0;) {
      variables.add(new Variable(domain, "V" + i));
    }
    for (int i = size; --i >= 0;) {
      for (int j = size; --j >= i + 1;) {
        constraints.add(diff(variables.get(i), variables
            .get(j)));
      }
    }
  }

  private Constraint diff(final Variable var1,
    final Variable var2) throws FailedGenerationException {
    PredicateConstraint constraint = new PredicateConstraint(
        new Variable[] { var1, var2 }, predicate);
    constraint.setParameters(var1.getName() + " "
        + var2.getName());
    constraint.compileParameters();
    return constraint;
  }

  public Collection<Variable> getVariables() {
    return variables;
  }

  public Collection<Constraint> getConstraints() {
    return constraints;
  }
}
```

Listing 1.3. The Pigeons problem

```
public static void main() throws
  FailedGenerationException, IOException {
  final Problem problem = Problem.load(10);
  final Solver solver = new MGAC(problem);
  final boolean result = solver.runSolver();
  System.out.println(result);
  if (result) {
    System.out.println(solver.getSolution());
  }
}
```

Listing 1.4. Solving the Pigeons-10 problem

Algorithm 3: MGAC(P = $(\mathscr{X}, \mathscr{C})$: CN, $maxBT$: Integer): Boolean

1 **if** $maxBT < 0$ **then throw** $Expiration$
2 **if** $\mathscr{X} = \emptyset$ **then return true**
3 **select** $(X, v) \mid X \in \mathscr{X} \wedge a \in dom(X)$
4 $P' \leftarrow GACrm(P|_{X=a})$
5 **if** $P' \neq \bot \wedge MGAC(P' \backslash X, maxBT)$ **then return true**
6 $P' \leftarrow GACrm(P|_{X \neq a})$
7 **return** $P' \neq \bot \wedge MGAC(P', maxBT - 1)$

ment of GAC3 [10]. They both admit a worst-case time complexity of $O(er^3d^{r+1})$. GAC2001 [2] admits a worst-case time complexity of $O(er^2d^r)$ and has been proved to be an optimal algorithm for establishing Generalized Arc Consistency.

The GAC3rm algorithm is described in Algorithm 2. Every variable of the CN is put in a queue in order to be revised one by one using Algorithm 1. If an effective revision is done (i.e. at least one value is removed from the variable), all neighbors of the variable are put in the queue. The algorithm continues until a fix-point is reached, i.e. no more value can be removed in the CN. A neighbor variable is one that shares at least one constraint with the current variable.

Residual supports ($supp[C, X, v]$) are used during the revision in order to speed up the search. Contrary to GAC2001, if the residue is no longer valid, the search for a valid tuple is restarted from scratch, which allow us to keep the residues from one call to another, even after a backtrack. Although GAC3rm by itself is not optimal, [9] shows that maintaining GAC3rm during search (see below) is more efficient than maintaining GAC2001.

The MGAC algorithm [14] aims at solving a CSP instance and performs a depth-first search with backtracking while maintaining (generalized) arc consistency. More precisely, at each step of the search, a variable assignment is performed followed by a filtering process called constraint propagation which corresponds to enforcing generalized arc-consistency.

Recent implementations of MGAC use a binary (2-way) branching scheme [7]: at each node of the search tree, a variable X is selected, a value $a \in$ dom(X) is selected,

Algorithm 4: initP($P = (\mathscr{X}, \mathscr{C})$: CN): Integer

```
1 foreach X ∈ 𝒳 do
2     select v ∈ dom(X) | countConflicts(P|_{X=v}) is minimal
3     P ← P|_{X=v}
4 return countConflicts(P)
```

and two edges are considered: the first one corresponds to $X = a$ and the second one to $X \neq a$.

Algorithm 3 corresponds to a recursive version of the MGAC algorithm (using binary branching). A CSP instance is solved by calling the $MGAC$ function: it returns **true** iff the instance is satisfiable. $P|_{X=a}$ denotes the constraint network obtained from P by restricting the domain of X to the singleton $\{a\}$ whereas $P|_{X\neq a}$ denotes the constraint network obtained from P by removing the value a from the domain of X. $P \backslash X$ denotes the constraint network obtained from P by removing the variable X.

The heuristics that allows the selection of the pair (X, a) has been recognized has a crucial issue for a long time. Using different variable ordering heuristics to solve a CSP instance can lead to drastically different results in terms of efficiency.

In [3], it is proposed to associate a counter, denoted $wght[C]$, with any constraint C of the problem. These counters are used as constraint weighting. Whenever a constraint is shown to be unsatisfied (during the constraint propagation process), its weight is incremented by 1 (see line 11 of Algorithm 1).

The weighted degree of a variable X is then defined as the sum of the weights of the constraints involving X and at least another uninstantiated variable. The adaptive heuristic $dom/wdeg$ [3] involves selecting first the variable with the smallest ratio current domain size to current weighted degree. As search progresses, the weight of hard constraints become more and more important and this particularly helps the heuristic to select variables appearing in the hard part of the network. This heuristic has been shown to be quite efficient [19].

3.2 Local Search algorithms

Although there also has been some interest in using Local Search techniques to solve the CSP problem [12, 4, 5, 18], these algorithms have not be studied a fraction as much as MGAC. Contrary to systematic backtracking algorithms like MGAC, local search techniques are incomplete by nature: if a solution exists, it is not guaranteed to be found, and the absence of solution can usually not be proved. However, on very large instances, local search techniques have been proved to be the best practical alternative. We also found that local learch algorithms are far more efficient than MGAC on quite small, dense instances.

A local search algorithm works on *complete assignments*: each variable is assigned with some value, then the assignment is iteratively *repaired* until a solution is found. A repair generally involves changing the value assigned to a variable so that as few constraints as possible are violated [12]. The initial variable assignments may be randomly generated. However, in order to make the first repairs more significant, we use

Algorithm 5: initγ(P = $(\mathscr{X}, \mathscr{C})$: CN)

```
1 foreach X ∈ 𝒳 do
2     foreach v ∈ dom(X) do
3         γ(X, v) ← 0
4         foreach C ∈ 𝒞 | X ∈ vars(C) do
5             if ¬check(C|_{X=v}) then γ(X, v) ← γ(X, v) + wght[C]
```

Algorithm 6: updateγ(X : Variable, v_{old}: Value)

```
1 foreach C ∈ 𝒞 | X ∈ vars(C) do
2     foreach Y ∈ vars(C) | X ≠ Y do
3         foreach v_y ∈ dom(Y) do
4             if check(C|_{Y=v_y}) ≠ check(C|_{Y=v_y ∧ X=v_old}) then
5                 if check(C|_{Y=v_y}) then
6                     γ(Y, v_y) ← γ(Y, v_y) − wght[C]
7                 else
8                     γ(Y, v_y) ← γ(Y, v_y) + wght[C]
```

Algorithm 4 to build the initial variable assignment. The algorithm tries to minimize the number of conflicting constraints after initialization. $countConflicts(P)$ returns the number of falsified constraints involving only assigned variables.

Designing efficient local search algorithms for CSP requires the use of clever data structures and powerful incremental algorithms in order to keep track of the efficiency of each repair. [4] proposes to use a data structure $\gamma(X, v)$ which at any time contains the number of conflicts a repair would lead to. Algorithms 5 and 6 describes the management of γ ($check(C)$ controls whether C is satisfied by the current assignments of $vars(C)$). Since each assignation has an impact only on the constraints involving the selected variable, we can count conflicts incrementally at each iteration with a worst-time complexity of $O(\Gamma_{max} rd)$.

There are many cases where no value change can improve the current assignment in terms of constraint satisfaction. In this case, we have reached a *local minimum*. The main challenge over local search techniques is to find the best way to avoid or escape local minima and carry on the search. A $maxIterations$ parameter is given to each local search algorithm. It mostly allows to define a restart strategy: if no solution is found after a fixed number of iterations, the search is restarted with a new initial assignment. The best value of $maxIterations$ is highly dependant on the nature of the problem. This comes against our view of a "black box" CSP solver, and future progress on CSP4J will be aimed to eliminate that kind of parameter. However, default values are given to each algorithms and we found them to be quite robust.

The MCRW Engine With a probability p, the repair is chosen randomly instead of being selected into the set of repairs that improves the current assignment. The first al-

Algorithm 7: MCRW($P = (\mathscr{X}, \mathscr{C})$: CN, $maxIterations$: Integer): Boolean

```
1  nbConflicts ← initP(P) ; initγ(P) ; nbIterations ← 0
2  while nbConflicts > 0 do
3      select X randomly | X is in conflict
4      if random[0,1] < p then
5          select v ∈ dom(X) randomly
6      else
7          select v ∈ dom(X) | γ(X,v) is minimal
8      v_old ← current value for X
9      if v ≠ v_old then
10         P ← P|_{X=v}
11         nbConflicts ← γ(X,v)
12         updateγ(X, v_old)
13         if nbIterations++ > maxIterations then throw Expiration
14 return true
```

Algorithm 8: Tabu($P = (\mathscr{X}, \mathscr{C})$: CN, $maxIterations$: Integer): Boolean

```
1  nbConflicts ← initP(P) ; initγ(P) ; nbIterations ← 0
2  init TABU randomly
3  while nbConflicts > 0 do
4      select (X,v) ∉ TABU∨ meets the aspiration criteria | γ(X,v) is minimal
5      v_old ← current value for X
6      insert (X, v_old) in TABU and delete oldest element from TABU
7      P ← P|_{X=v}
8      nbConflicts ← γ(X,v)
9      updateγ(X, v_old)
10     if nbIterations++ > maxIterations then throw Expiration
11 return true
```

gorithm implementing this technique was described in [12] and we call it *Min-Conflicts Random Walk* (MCRW). Algorithm 7 performs a MCRW local search. At each iteration, a variable in conflict is selected (line 3). A variable X is in conflict if any constraint involving X is in conflict. Then, with a probability p, a random value (line 5) or, with a probability $1 - p$, the best value (line 7) is selected. p is one additional parameter we aim to eliminate in further versions of CSP4J. Again, the default value ($p = 0.04$) is quite robust for most problems.

The Tabu engine: Previous repairs are recorded so that we can avoid repairs that lead back to an already visited assignment. A limited number of repairs is remembered, and older ones are forgotten, allowing us to always have a fairly high number of repairs available at each iteration. The size of the Tabu List is arbitrary fixed before search. Note that the *aspiration criterion* allows to select a repair in the Tabu list if it permits to achieve a new best assignment. There have been previous works that mention the

Algorithm 9: WMC($P = (\mathscr{X}, \mathscr{C})$: CN, $maxIterations$: Integer): Boolean

```
1  nbConflicts ← initP(P) ; initγ(P) ; nbIterations ← 0
2  while nbConflicts > 0 do
3      select (X, v) | γ(X, v) is minimal
4      v_old ← current value for X
5      if γ(X, v) ≥ γ(X, v_old) then
6          foreach C ∈ 𝒞 | C is in conflict do
7              wght[C]++ ; nbConflicts++
8              foreach Y ∈ vars(C) do
9                  foreach w ∈ dom(Y) do
10                     if ¬check(C|_{Y=w}) then γ(Y, w)++
11     else
12         P ← P|_{X=v}
13         nbConflicts ← γ(X, v)
14         updateγ(X, v_old)
15     if nbIterations++ > maxIterations then throw Expiration
16 return true
```

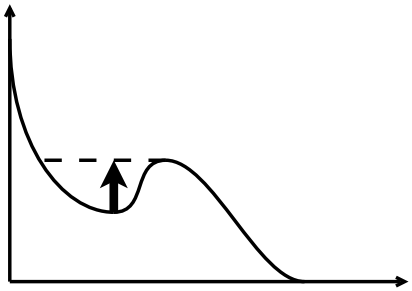

Fig. 2. Escaping from a local minimum

efficiency of Tabu search for Constraint Optimization problems (Max-CSP) [4, 5]. Algorithm 8 performs a Tabu search. The size of the Tabu list is one additional parameter we aim to eliminate in further versions of CSP4J. Again, the default value (30) is quite robust for most problems.

The WMC Engine Another efficient way to escape from local minima, called the Breakout method, has also been proposed [13]. We use this method to design a local search algorithm aimed to find solutions to satisfiable CSPs.

The resulting algorithm, *Weighted Min-Conflicts* (WMC) is described in Algorithm 9. Line 5 detects local minima. When a local minimum is encountered, all conflicting constraints are weighted (line 12). Note that a main advantage of WMC over Tabu search or MCRW is that it involves no parameter outside of $maxIterations$.

Algorithm 10: Hybrid(P = $(\mathscr{X}, \mathscr{C})$: CN, $maxIter$: Integer, α: Float): Boolean

```
1  maxTries ← 1 ; maxBT ← maxIter × 8n/ed
2  repeat
3      startTime ← now()
4      repeat ⌊maxTries⌋ times
5          try
6              return WMC(P, maxIter)
7          catch Expiration
8      WMCDuration ← now() − startTime
9      startTime ← now()
10     try
11         return MGAC(P, maxBT)
12     catch Expiration
13     MGACDuration ← now() − startTime
14     maxTries ← α × maxTries
15     maxBT ← α × maxBT × WMCDuration/MGACDuration
```

Incrementing the weight of constraints permits to effectively and durably escape from local minima, as illustrated by Figure 2. Incrementing the constraints "fills" the local minimum until another parts of the search space are reached. Constraints that are heavily weighted are expected to be the "hardest" constraints to satisfy. By weighting them, there importance is enhanced and the algorithm will try to satisfy them in priority.

The Combo engine It is well known that the main drawback of systematic backtracking strategies such as MGAC is that an early bad choice may lead to explore a huge sub-tree that could be avoided if the heuristic had lead to focus on a rather small, very hard or even inconsistent sub-problem. In this case, the solver is said to be subject to "thrashing": it rediscovers the same inconsistencies multiple times. On the other hand, it is important to note that some instances are not inherently very difficult. These often show a "heavy tailed" behavior when they are solved multiple times with some randomization [6]. The $dom/wdeg$ heuristic was designed to avoid thrashing by focusing the search on one hard sub-problem [3, 18]. This technique is reported to work quite well on structured problems.

On the other hand, the main drawback of local search algorithms is quite straightforward: their inability to prove the unsatisfiability of problems and the absence of guarantee, even on satisfiable problems, that a solution will be found. The development of hybrid algorithms, hopefully earning the best from each world, has been devised as a great challenge in both satisfiability and constraint satisfaction problems [15].

Constraint weighting used by $dom/wdeg$ heuristic and WMC work in a similar way. Both help to identify hard sub-problems. [11] reports that statistics earned during a failed run of local search can be successfully as an oracle to guide a systematic algorithm in the search of a solution or to extract an incoherent core. We propose to use directly the weights of the constraints obtained at the end of a WMC run to initiate

$dom/wdeg$ weights. We devise a simple hybrid algorithm, described by Algorithm 10 based on this assumption.

4 Conclusion and perspectives

We presented CSP4J, an API for Java 5, intended to solve CSPs as part on any Java application, in a "black-box" scheme. We introduced clues on CSP4J usage and given some examples of use, the we presented the five engines shipped with CSP4J and their respective interest.

We will continue to develop CSP4J, by optimizing the algorithms as well as refining them according to the latest refinements of fundamental research is Constraint Programming, and especially SAT and CSP solving. Next developments of CSP4J will focus on preprocessing, especially using promising algorithms such as Dual Consistency [8]. We will also try to eliminate any user-supplied parameter from our algorithms and will focus towards merging the advantages of all engines so that no expertise at all should be needed from the user, in the spirit of CLP(FD) used in Prolog interpreters.

References

1. Second International CSP Solvers Competition. http://www.cril.univ-artois.fr/CPAI06, 2006.
2. C. Bessiere, J.C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.
3. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
4. P. Galinier and J.K. Hao. Tabu search for maximal constraint satisfaction problems. *Proceedings of CP'97*, pages 196–208, 1997.
5. P. Galinier and J.K. Hao. A General Approach for Constraint Solving by Local Search. *Journal of Mathematical Modelling and Algorithms*, 3(1):73–88, 2004.
6. C.P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24:67–100, 2000.
7. J. Hwang and D.G. Mitchell. 2-way vs d-way branching for CSP. In *Proceedings of CP'05*, pages 343–357, 2005.
8. C. Lecoutre, S. Cardon, and J. Vion. Conservative Dual Consistency. In *Proceedings of AAAI'07, to appear*, 2007.
9. C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'2007)*, pages 125–130, 2007.
10. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
11. B. Mazure, L. Sais, and E. Gregoire. Boosting complete techniques thanks to local search methods. *Annals of Mathematics and Artificial Intelligence*, 22:319–331, 1998.
12. S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint-satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.
13. P. Morris. The breakout method for escaping from local minima. In *Proceedings of AAAI'93*, pages 40–45, 1993.

14. D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94*, pages 10–20, 1994.
15. B. Selman, H. Kautz, and D. McAllester. Ten challenges in propositional reasoning and search. *Proc. IJCAI'97*, 1997.
16. R.M. Stallman. GNU General Public License. *GNU Project–Free Software Foundation, http://gnu.org/licenses*, 1991.
17. R.M. Stallman. GNU Lesser General Public License. *GNU Project–Free Software Foundation, http://gnu.org/licenses*, 1999.
18. J.R. Thornton. *Constraint weighting local search for constraint satisfaction*. PhD thesis, Griffith University, Australia, 2000.
19. M. R. C. van Dongen, editor. *Proceedings of CPAI'05 workshop held with CP'05*, volume II, 2005.
20. J. Vion. Hybridation de prouveurs CSP et apprentissage. In *Actes des troisièmes Journées Francophones de Programmation par Contraintes (JFPC '07), to appear*, 2007.