INCOP: An Open Library for INcomplete Combinatorial OPtimization version 1.0

Bertrand Neveu and Gilles Trombettoni

Projet COPRIN, CERMICS-I3S-INRIA
Route des lucioles, BP 93, 06902 Sophia Antipolis France
Bertrand.Neveu@sophia.inria.fr, Gilles.Trombettoni@sophia.inria.fr

Abstract. We present a new library, `INCOP`, which provides incomplete algorithms for optimizing combinatorial problems. This library offers classical local search methods such as simulated annealing, tabu search as well as a population based method, Go With the Winners. Several problems have been encoded, including Constraint Satisfaction Problems, graph coloring, frequency assignment.
`INCOP` is an open C++ library. The user can easily add new algorithms and encode new problems. The neighborhood management has been studied carefully. First, an original parameterized move selection allows us to easily implement most of the existing meta-heuristics. Second, different levels of incrementality can be specified for the configuration cost computation, which highly improves efficiency.
`INCOP` has shown great performances on our sample of well-known benchmarks. It outperforms most of the existing concurrent tools. The challenging `flat300_28` graph coloring instance has been colored in 30 colors for the first time by a standard Metropolis algorithm.

1 Introduction

Discrete optimization problems can be solved by two majors types of methods, complete and incomplete ones. Complete algorithms are based on a tree search with a Branch and Bound schema. Several commercial software tools propose such methods. When an optimization problem can be modeled by linear constraints and a linear criterion, MIP packages can be used. Otherwise, constraint programming tools, such as `IlogSolver` or `Chip`, can be used.

When the search space becomes too large, these systematic search techniques are often outperformed by incomplete methods. Incomplete algorithms cannot prove the optimality of a solution, but can often give rapidly a good solution. The most common incomplete methods are based on local search which tries to make local changes to *one* configuration for improving its cost. Other incomplete methods explore the search space by managing a population of configurations, that is, *several* configurations simultaneously.

To efficiently implement complete algorithms requires a great effort and therefore commercial tools have been built and are successfully used. Conversely, it is easier to implement an incomplete method and no important effort has been made to build a commercial tool. `IlogSolver` has recently added a local search module, but it is included in the whole library and cannot be used separately.

This lack of tool has recently led many researchers to build their own incomplete search method libraries. A lot of projects have been presented these last years[23]. We can cite `iOpt` [24] by British Telecom, `SCOOP` [18] by SINTEF, `Localizer++` [14] at Brown University, `Hotframe` [22] at University of Braunschweig, `Discropt` [19] at State University of New York. Philippe Galinier and Jin-Kao Hao [7] also proposed a framework for local search.

However, at the moment, all these libraries are not free or not available. We have found only one free library available on the Web: `EasyLocal++`, at University of Udine [3], that implements local search methods.

Initially, we wanted to test a population-based method and compare it with local search methods in the same implementation. Indeed, a fair comparison between methods can only be done in the same software platform. We remarked that the implementation itself, the data structures used, and the degree of incrementality were very important. In that purpose, we decided to provide a free library, implementing the main local search meta-heuristics and efficient population-based methods.

Main features of the `INCOP` library

Without loss of generality, our library handles only minimization problems.

For the moment, we have implemented only one type of configuration (i.e., solution) represented by a fixed set of integer variables, with a priori known domains of values. Important combinatorial optimization problems can be encoded in this framework. It is straightforward to represent traveling salesman problems (TSP) and constraint satisfaction problems (CSP). Graph coloring and certain frequency assignment problems can also be viewed as specific CSPs. Other types of configurations can be added in the library.

Contributions. Since local search makes displacements in the configuration space, we have to propose a move component. For efficiency reasons, `INCOP` allows the algorithms to incrementally compute the cost of a configuration (i.e., the criterion to be optimized) when a move has been performed. In addition, `INCOP` has the following features:

- An original parameterized way for selecting a neighbor configuration is provided, a kind of candidate list strategy [10], has been embedded in `INCOP`.
- A population-based method (`GWW`) is implemented, with a very efficient version hybridized with local search (`GWW-grw`).
- It is possible to add a new method and to implement a new combinatorial optimization problem.
- The most popular local search metaheuristics are implemented such as Hill Climbing, `GSAT` [20], Simulated Annealing [1], Tabu Search [8].

Contents

Section 2 presents a first object-oriented architecture of the library. This section also illustrates how to add a new meta-heuristic, a new problem or define a new neighborhood. Section 3 details the contributions of `INCOP`: the incremental configuration cost computations, our parameterized move selection, and the implemented population-based algorithms. Experimental results obtained on graph coloring and radio link frequency assignments instances are reported in Section 4 and highlight the good performances of `INCOP`. The related work is presented in Section 5.

2 Architecture

We have chosen an object oriented design and implemented the library in C++, using virtual methods and data structures provided by the STL. Although no language specific to incomplete algorithms has been developed (such as `OPL` [14] or `SALSA` [13]), we will show that it is not difficult to define new meta-heuritics or new problems.

`INCOP` contains six main classes:

- The class `Configuration` specifies the encoding of a solution.
- Specializing the class `Move` defines a new neighborhood for configurations.
- The class `NeighborhoodSearch` allows a configuration to search for a neighbor in a parameterized way (see Section 3.3).
- Specializing the class `IncompleteAlgorithm` defines a new algorithm.
- Specializing the class `Metaheuristic` defines a new metaheuristic.
- Specializing the class `OpProblem` adds a new problem.

Note that algorithms and problems are separated in two class hierarchies. It is then possible to add a new problem and solve it with the yet implemented meta-heuristics, or conversely, to add a new algorithm and test it on the already implemented problems.

We should notice that an original neighborhood management is embedded in `INCOP`. The attribute `tabconflicts` in subclasses of `Configuration`, and methods of the class `OpProblem` can offer a high level of incrementality to meta-heuristics for the configuration cost evaluation. Attributes in the class `NeighborhoodSearch` allow a parameterized move selection (see Section 3.3).

2.1 Defining a configuration

In the current version, the class `Configuration` is only specialized for representing a `MAX-CSP` solution, minimizing the number (or a weigted sum) of constraint violations. The current class hierarchy is the following:

```
Configuration
    CSPconfiguration
         IncrCSPconfiguration
         FullincrCSPconfiguration
```

An object from the class CSPconfiguration has three main attributes. The solution is encoded in the attribute config as a vector of values, one for every variable in the CSP instance. Another attribute valuation contains the cost of the configuration.

A third attribute, called conflicts, is defined in subclasses of CSPconfiguration to store the number of conflicts of the values. The section 3.1 details how the attribute conflicts is updated when a move is performed.

2.2 Defining a new neighborhood

The neighborhood is mainly defined in two classes: NeighborhoodSearch and Move. Objects of the class Move store a dynamic information associated to the latest modification of a configuration (a move), allowing the problem to incrementally update the cost of the configuration.

For adding a new neighborhood, we have to define a subclass of this class which currently contains two subclasses. The subclass SwapMove defines a swap used in a permutation configuration: the values of two variables are exchanged. Such a move is mainly defined by the 2 swapped variables variable1 and variable2.

```
class SwapMove : public Move {
  public :
     int variable1;
     int variable2;
     SwapMove();
     ...
};
```

The subclass CSPMove yields the simplest way to define a neighborhood in CSPs: two neighbors differ in the value of one variable.

An object of the class NeighborhoodSearch can also be used to refine the neighborhood for CSPs. Based on Minton's work [15], INCOP yields variants of this neighborhood definition: two boolean attributes are defined for that purpose.

- var_conflict: indicator of restriction to the variables participating to a conflict.
- min_conflict: indicator of restriction to the values of a variable that minimize the conflicts for this variable (Min-conflicts heuristics [15]).

One could either be more selective in the determination of the possible neighbors or less selective and implement large neighborhoods (as in LNS).

2.3 Incomplete algorithms

IncompleteAlgorithm is the root class. It is subdivided in LSAlgorithm and GWWAlgorithm.

```
IncompleteAlgorithm
   LSAlgorithm
   GWWAlgorithm
        ThresholdGWWAlgorithm
        NoThresholdGWWAlgorithm
```

The class `LSAlgorithm` implements local search algorithms that perform a walk for a single configuration. This class has three main attributes:

– a number of moves to be performed before the end of the algorithm,
– a pointer `nbsearch` to an object of the class `NeighborhoodSearch`, allowing the algorithm to pass from a configuration to another in a parameterized way,
– a pointer `mheur` to a `Metaheuristic` object.

Current meta-heuristics are implemented as subclasses of the abstract class `Metaheuristic`, including `TabuSearch`, `GreedySearch`, `SimulatedAnnealing`. Other can be added easily. The subclasses mainly contain a specific method `acceptance` deciding whether a tested move is accepted and specific data structures. For example, the class `TabuSearch` has a field to store its tabu list.

The class `GWWAlgorithm` is an important feature of `INCOP` detailed in Section 3.2. It implements algorithms that treat several configurations simultaneously.

Adding a new meta-heuristic

For implementing a new meta-heuristic, one has to define a subclass of `Metaheuristic`, with its data, and three methods:

– `reinit`: initialization of the metaheuristic data,
– `acceptance`: acceptance condition of a candidate move,
– `executebeforemove`: updating the meta-heuristic data (like the temperature of simulated annealing or the tabu list) before executing a move.

To illustrate this part, we present the code of the Threshold Accepting meta-heuristic [6]: a move is accepted if the deterioration of the current configuration evaluation is less than a threshold; this threshold is going down linearly from `thresholdinit` to 0 during the search (a walk of length `walklength`). At each move, the threshold is thus decremented by `delta`.

```
class ThresholdAccepting: public Metaheuristic {
  public :
    double thresholdinit;
    double delta;
    double thresholdaccept;
    ThresholdAccepting(double maxthreshold, int walklength);
    int acceptance (Move * move, Configuration* config);
```

```
    void executebeforemove(Move*  move, Configuration * configuration,
                           OpProblem * problem);
    void reinit();
};

ThresholdAccepting::ThresholdAccepting
                 (double maxthreshhold,int walklength) {
  thresholdinit=maxthreshhold; delta=thresholdinit/walklength;}

int ThresholdAccepting::acceptance (Move * move, Configuration*  config) {
  return ((move->valuation - configvaluebefore) <  thresholdaccept);}

void ThresholdAccepting::reinit() {
   thresholdaccept = thresholdinit;}

void ThresholdAccepting::executebeforemove (Move * move,
 Configuration * configuration, OpProblem * problem) {
   if (thresholdaccept >= delta)  thresholdaccept -= delta; }
```

Then, the metaheuristics can directly be used by filling the `mheur` attribute of a LSAlgorithm algo object.

```
algo->mheur = new ThresholdAccepting (100., algo->walklength);
```

It is also possible to integrate this metaheuristics in the list of methods known at the command line level. One has therefore to slightly modify the file incoputil.cc : one has to add the method in the known method list by :

– adding `liste_methodes.push_back("threshold");` in definir_liste_methodes
– adding the metaheuristic name in the partial method list used in algo_creation for separating simple local search algorithms from GWW algorithms
– adding in algo_marche an entry for the reader of its arguments and another for installing the metaheuristic in the `mheur` attribute.

You can see how it has been done in incoputil.cc to do the same thing for implementing a new metaheuristic.

2.4 The problem hierarchy

The problems that can be implemented in the library are discrete optimization problems as, for example, Traveling Salesman Problem. The constraint satisfaction problems (CSP) are transformed into optimization MAX-CSP problems for which the number of violated constraints (or more generally a given criterion computed on these constraint violations) is minimized. A solution is obtained when a configuration is found with a cost equal to zero. We have implemented several CSP problems, including graph coloring and frequency assignment problems.

```
OpProblem : optimization problem
   CSProblem : Constraint Satisfaction Problem
      BinaryCSProblem : Binary CSP
         ColorCSProblem  : Graph Coloring
         CelarCSProblem  : Celar frequency assignment (add. criterion)
         ExtensionBinaryCSProblem : for example random CSP (handled asMax-CSP)
         CliqueProblem : Find a clique of given size in a graph
         NQueenProblem   : N-queens problem
   TSProblem : Traveling Salesman problem
```

Adding a new problem

If the problem has binary constraints, we can inherit the `constraints` datastructure from the class `BinaryCSProblem`. If necessary, a special neighborhood can be created as a subclass of `Move`.

The criterion to be optimized is specific to a given problem. Three methods of the class `OpProblem` allows the computation of the configuration cost. The method `config_evaluation` evaluates the cost of an initial configuration; `move_evaluation` performs the incremental evaluation of a move; the methods `incr_update_conflicts` or `full_incrupdate_conflicts` update the conflict datastructure of a configuration when a move is executed, depending on the incrementality chosen.

For example, the graph coloring instances are implemented by the following `ColorCSProblem` class. We use a full incrementality for the move evaluations.[1].

```
class ColorCSProblem : public BinaryCSProblem {
  public :
    ColorCSProblem (int nvar, int nconst, int nbcol);
    int config_evaluation (CSPconfiguration * config);
    void fullincr_update_conflicts (FullincrCSPConfiguration *config,
                                    Move * move);
    void init_domains (int nbvar, int nbcol);
    Configuration* create_configuration();
};
```

Graph coloring instances can be encoded as MAX-CSP instances: variables are the vertices in the graph to be colored; the number `nbcol` of colors with which the graph must be colored yields domains ranging from 1 to `nbcol`; vertices linked by an edge must be colored with different colors: the corresponding variables must take different values. Coloring a graph in `nbcol` colors amounts in minimizing the number of violated constraints and finding a solution with cost 0. This modelization allows us to reuse a large part of the code defined in the `BinaryCSProblem` class.

The moves are also the same as for standard CSPs: changing a color of a node amounts in modifying a value of a CSP variable. The conflict datastructure and the `move_evaluation` method are also inherited from `BinaryCSProblem`.

[1] In addition, a function first reads the data from a file in Dimacs format, calls the `ColorCSProblem` constructor, and fills the constraints datastructure.

Only the methods config_evaluation and fullincr_update_conflicts are not straighforward to be written. They are detailed below.

The create_configuration method specifies which incrementality mode we use (here the full incremental mode).

```
Configuration* ColorCSProblem::create_configuration()
{return (new FullincrCSPConfiguration (nbvar,domainsize ));}
```

The config_evaluation method computes the cost of a configuration and fills the conflict data structure.

```
int ColorCSProblem::config_evaluation (Configuration* configuration)
{int value=0;
 configuration->init_conflicts();
 for(int i=0; i<nbvar; i++)
   for (int j=i+1; j< nbvar; j++)
     if (constraints[i][j])
       {if (configuration->config[i]==configuration->config[j])
          value++;
       configuration->incr_conflicts(i,configuration->config[i],
                                     configuration->config[j],1);
       configuration->incr_conflicts(j,configuration->config[i],
                                     configuration->config[i],1);
       }
 return value;
}
```

The method updating the tabconflicts conflict data structure is implemented as follows.

```
void ColorCSProblem::fullincr_update_conflicts
                     (FullincrCSPConfiguration* configuration,Move* move)
{int changed_variable = ((CSPMove*)move)-> variable;
  int changed_value = ((CSPMove*)move)-> value;
 for (vector<int>::iterator vi= connections[changed_variable].begin();
                       vi!= connections[changed_variable].end(); vi++)
   {configuration->tabconflicts[*vi][configuration->config[changed_variable]]--;
    configuration->tabconflicts[*vi][changed_value]++;
   }
}
```

This code illustrates the main datastructures used. The array config contains the current value assignment of every variable in a configuration. The conflicts appear in the two-dimensional array conflicts indexed by one variable and one value index. The binary constraints, i.e., the links between vertices, are redundantly stored in 2 structures for efficiency reasons. A two-dimensional array constraints indicates if two variables i and j are linked by a constraint.

3 Contributions

This section details original features of INCOP. First, the incremental configuration cost computations offered by our library improve efficiency. Second, efficient population-based algorithms can be used to tackle the most difficult instances. Third, an original parameterized move selection can lead to easily create new variants of existing algorithms. More important, we hope that deeply studying the new facilities offered by this move selection will lead to new classes of local search methods [9].

3.1 Incrementality

It is important to incrementally update the `conflicts` structure of `Configuration` objects (see Section 2.1). Indeed, the contribution of any variable value to the evaluation of a configuration cost is the number of constraints violated by this value (considering the current value of the other variables). Since this operation is performed very often, it is fruitfull to rapidly evaluate the impact of a move on the whole configuration cost:

1. In `CSPconfiguration`: the conflicts are not stored: one needs to compute the number of constraints violated by the old and the new values.
2. In `IncrCSPconfiguration`: the contribution of the current value is stored in `tabconflicts`; we need to compute only the contribution of the new value.
3. In `FullincrCSPconfiguration`: all the contributions of all possible values are stored in `tabconflicts`; the evaluation of a move is immediate.

This is implemented by two methods of the class `OpProblem` (the problem to be solved).

– The method `move_evaluation` is called at the end of the method `next_move`, that is, when a move is tested.
– The methods `incr_update_conflicts` or `fullincr_update_conflicts` are called every time a move is performed.

The full incrementality embedded in `FullIncrCSProblem` instances implies that the computation effort is mainly performed in the `fullincr_update_conflicts` method. It is fruitfull when a lot of moves must be tested before accepting one. When the problem is sparse, e.g., in most of the graph coloring instances, the updating is not costly. Regarding the value of variable v changed by the current move, the updating only concerns the values of the few variables linked with v by a constraint. In practice, this full incremental way can save an order of magnitude in computing time. The memory required is also reasonable for coloring problems: the size of the conflict datastructure is $N \times D$, where N is the number of nodes and D is the number of colors. The good experimental results reported in Section 4 highlight the interest of this feature.

3.2 Go With the Winners algorithms

The population-based algorithms implemented in INCOP are variants of the Go With the Winners algorithm [4]. Several configurations are handled simultaneously. Every configuration, named particle, performs a random walk and, periodically, the worst particles are redistributed on the best ones. To ensure improvements in the population, a threshold is lowered during the search and no move passing above the threshold is allowed.

The hybridization with local search is straightforward: instead of performing a random walk, every particle performs a local search. In the library, the `GWWAlgorithm` object owns an attribute, named `walkalgorithm`, which contains a `LSAlgorithm` object. Therefore no extra code is necessary. One of these hybridizations, `GWW-grw` [17], a hybridization of `GWW` with a simple walk algorithm, has given very good results (see Section 4).

It would be possible to implement in the library classical genetic algorithms, or memetic algorithms (hybrids of genetic algorithms and local search). Note, however, that `GWW` and `GWW-grw` present several advantages as compared to genetic algorithms. First, they are simpler. Second, the crossover operator of genetic algorithms is generally not relevant in very structured problems such as CSPs. Third, incremental move selections cannot be performed with the crossover operator in genetic algorithms, making these algorithms not competitive.

3.3 Selection of a move

An atomic step in local search algorithms is the way neighbors of the current configuration are tested. An original generic move selection has been embedded in INCOP.

Several parameters specify the move selection. They belong to objects of the class `NeighborhoodSearch`. The class `LSAlgorithm` has an attribute which contains an object of the class `NeighborhoodSearch`.

First, one must specify the method `is_feasible` giving a feasability condition for the move. For instance, in `GWW` algorithms, the configuration cost must stay under the current threshold.

In order to finely tune the intensification effort of the search, the user must specify 3 parameters for searching a move in the neighborhood of the current configuration: `Min_neighbors`, `Max_neighbors` and `No_acceptation`.

1. We first test `Min_neighbors` neighbors in order to select the *best* one.
2. If none has been accepted by the meta-heuristics, we test other neighbors until *one* is accepted or a sample of `Max_neighbors` is exhausted.
3. Finally, if no neighbor among these `Max_neighbors` has been accepted, the `No_acceptation` parameter with value `no_move`, `one_neighbor` or `best-neighbor` indicates how to select a configuration:
   – `no_move`: the current configuration is not modified,
   – `one_neighbor`: select any feasible neighbor among `Max_neighbors` (without taking into account the meta-heuristic acceptance criterion),

– `best_neighbor`: select the best feasible neighbor.

In existing algorithms, `Min_neighbors` is generally set to 0 (e.g., in Metropolis), or to `Max_neighbors` (e.g., in a tabu search). It makes a trade-off between performance and effort for making a bigger progress. This parameterized function allows us to implement many different classical behaviors. For example:

– Search the best neighbor in the entire neighborhood:
  `Min_neighbors` = `Max_neighbors` = neighborhood size
– Search the first acceptable neighbor in a sample of K neighbors:
  `Min_neighbors` = 0, `Max_neighbors` = K.

Remark `Min_neighbors` and `No_Acceptation` are two parameters which are rarely explicit in existing algorithms. Deeper studies are necessary to evaluate their impact on performance, and the reasons behind. We have already observed very interesting behaviors with variants of existing algorithms just obtained by this move selection. For example, our first experiments seem to indicate that fixing `No_acceptation` to `one_neighbor` is a good way to escape from local minima.

The algorithm `Configuration_move` (see Algorithm 1) is implemented as a method of the class `LSAlgorithm`.

4 Experiments

We have performed experiments on difficult instances mainly issued from two categories of problems encoded as weighted MAX-CSPs problems: graph coloring instances proposed in the DIMACS challenge ten years ago, and frequency assignment problems[2]. All the tests have been performed on a `PentiumIII 935 Mhz` with a Linux operating system.

4.1 Graph coloring instances

We have selected difficult graph coloring instances from the two most difficult categories in the catalogue: the Leighton's graphs with 450 vertices `le450_15c` (16680 edges), `le450_15d` (16750 edges), `le450_25c` (17425 edges) and `le450_25d` (17343 edges), and the more dense `flat300_28` instance with 300 vertices and 21695 edges. In all instances, a best solution is hidden with respectively, 15, 25 and 28 colors.

The `flat300_28` instance has never been colored in 28 colors (in 29 or 30 colors neither). To our knowledge, only three incomplete algorithms succeeded in coloring it in 31 colors: the distributed version of the Impasse algorithm [16], a tabu search [5], and a heuristic mixing a genetic algorithm and local search [7]. We succeeded in coloring it in 31 colors in a few minutes and in 30 colors in 1.6 hours on average using a Metropolis algorithm (i.e., simulated annealing with

[2] Thanks to the "Centre d'ELectronique de l'ARmement".

```
algorithm Configuration_move(x : a configuration) Returns: a configuration
    i ← 0
    best? ← (Min_neighbors > 1) or (No_acceptation=best-neighbor)
    best-cost ← +∞ ; one-x ← ⊥
    accepted? ← false
    while (i < Min-Neighbors) or (i < Max-Neighbors and not(accepted?)) do
        x' ← next_move(x)
        if is_feasible(x') then
            if acceptance?(x, x') then accepted? ← true
            if best? and (cost(x') < best-cost) then
                x-best ← x'
                best-cost ← cost(x')
            end
            if (No_acceptation=one-neighbor) then one-x ← x'
        end
        i ← i + 1
    end
    if accepted? then
        if best? then
            return x-best
        else
            return x'
        end
    end
    if No-Acceptation=no-move or one-x=⊥ then
        return x
    end
    if No-Acceptation=best-neighbor then return x-best
    if No-Acceptation=one-neighbor then return one-x
end.
```

Algorithm 1: The `Configuration_move` method

constant temperature as proposed by [6]) and the min-conflict neighborhood implementing the Min-conflicts heuristics.

We report these results on table 1: for every instance, 10 tries have been performed for a given number of colors. We report the average cpu-time, the average number of conflicts at the end of a try and the success rate.

4.2 Comparison with other libraries

We have also compared `INCOP` with other local search libraries on the same graph coloring instances. We report the results published by their authors: `iOpt` [24] implemented in Java, `Easylocal++` [3], Galinier and Hao in [7]. These comparisons have been made on DSJC random graphs from DIMACS challenge with a number of colors greater than the chromatic number in order to conform with the already published results. Recently, some results on these graphs have been published with `Discropt` [19]. Since the allowed time is limited to 100 s, the

	nb-col	time	nb-col	conflicts	time	success	algo	neighb.
le450_15c	15	min	15	0	1.1 min	10/10	GWW-grw	var-conflict
le450_15d	15	min	15	1.4	1 min	5/10	GWW-grw	var-conflict
le450_25c	25	min	25	1.5	55 min	1/10	Metropolis	var-conflict
le450_25d	25	min	25	1.3	58 min	1/10	Metropolis	var-conflict
flat300_28	31	h	31	0.3	4 min	9/10	Metropolis	min-conflict
flat300_28	31	h	**30**	1.6	1.6 h	5/10	Metropolis	min-conflict

Table 1. Results on graph coloring benchmarks. The best results of known algorithms are reported in the left side of the table (number of colors, time); the results with INCOP in the right side (number of colors, number of conflicts (average), cpu time (average), success rate, algorithm and neighborhood used.

graphs are colored with a greater number of colors than the experiments presented below. This approach is thus not competitive. The results reported for INCOP in table 2 are the average cpu-times on 10 tries for solving each instance (the method was successfull for all the tries). We have used a Metropolis algorithm with constant temperature and a var-conflict neighborhood. We think that the good results we have obtained are in great part due to the full incrementality performed by our library when exploring the neighborhood. The only system which mentions the same kind of incrementality is the system by Galinier and Hao [7] which obtains comparable results.

Problem	Nb col	INCOP	Easylocal++	Galinier	IOPT
Machine		Pentium III 935	Pentium III 750	Sparc II 333	Pentium III 866
Method		Metropolis	Tabu	Tabu	Tabu or SA
125.1	6	0.005	0.031		166 (SA)
125.1	5	0.36			44 (T)
125.5	18	0.18	2.65		557 (T)
125.9	44	0.97	13.5		2904 (T)
250.1	9	0.04	0.61		
250.5	30	1.42	52.2	15	
250.5	29	7.81		85	
250.9	75	3.07	118.		
500.1	14	0.18	7.05		
500.5	54	2.43	457		
500.5	52	14.7		14	
500.5	51	96		47	
500.9	140	8.75	1856		

Table 2. Results on DSJC random graphs (CPU time in sec)

4.3 N-queen problem

On the Easylocal++ Web page, two examples of encoded problems are provided: n-queens and course time tabling problems. We have downloaded the Easylocal++ library with the n-queens problem encoding and run it on our machine. We have then implemented in INCOP this problem with the same neighborhood (swap in a permutation encoding). We have obtained comparable solving times using the same algorithm (Hill-climbing): for example, 10 tries of instances of size 3000, randomly initialized, are solved in average in 20.5 s with `Easylocal++` and in 21.6 s with `INCOP`.

4.4 Celar frequency assignment instances

We have also selected the three most difficult instances of the `CELAR` frequency assignment problems: the `celar6`, the `celar7` and the `celar8` instances. These instances are realistic since they have all been built from different sub-parts of a real problem. The `celar6` has 200 variables and 1322 constraints; the `celar7` has 400 variables and 2865 constraints; the `celar8` has 916 variables and 5744 constraints.

The variables are the frequencies to be assigned a value which belongs to a predefined set of allowed frequencies (domain size about 40). The constraints are of the form $|x_i - x_j| = \delta$ or $|x_i - x_j| > \delta$ (to avoid interferences). Our encoding is standard and creates only the even variables in the CSP along with only the inequalities[3].

The objective function to be minimized is a weighted sum of violated constraints. The constraints belong to four categories with different costs of violation:

- Constraints of `celar8` have a weight 1, 2, 3 or 4.
- Constraints of `celar6` have a weight 1, 10, 100 or 1000.
- Constraints of `celar7` have a weight 1, 100, 10000 or 10^6.

[2] and [12] describe complete algorithms based on graph-based decomposition and dynamic programming to solve the `celar6` instance. The other two instances are really challenging and have never been proven by a complete algorithm[4]. The algorithm in [11] is based on a very tuned genetic algorithm with a mutation operator encoding a MIP. The algorithm described in [25] is a local search taking the conflicts into account to select neighbors. The good results obtained by `GWW-grw` implemented in `INCOP` are reported in table 3.

	bound	**best found**	**bound (average)**	**time**	**succes**	**algo**
`celar6`	3389	min [2, 12]	3389 (3405.7)	9 min	4/10	`GWW-grw`
`celar7`	343592	min [11, 25]	343596 (343657)	4.5 h	1/10	`GWW-grw`
`celar8`	262	min [11, 25]	262 (267.4)	33 min	2/10	`GWW-grw`

Table 3. Results on `celar` frequency assignment benchmarks. The results of the best known algorithms are reported in the left side of the table; the results with INCOP in the right side.

[3] A bijection exists between odd and even variables. A simple propagation of the equalities allows us to deduce the values of the odd variables.

[4] This can be explained by the size of `celar8` and by the criterion of `celar7`.

5 Related Work

As mentioned in the introduction, many libraries have been proposed these last years. It is not possible to present in details all of them. Some are limited to local search methods, others present local search and population based algorithms such as genetic algorithms hybridized with local search. `Templar` permits distributed algorithms and parallelism. A survey of all these existing tools is presented in [23]. We can summarize the main features of these libraries in the following table.

Library	**language**	**meta-heuritics**	**miscellaneous**
`Templar`	C++	tabu, SA, genetic algos	parallelism
`Neighbor Searcher`	C++ Templates	local search + VNS	
`Hotframe`	C++ easy extension	local search	availability anounced
`Esylocal++`	C++	local search	open; available
`Iopt`	Java	local search + memetic	
`Discropt`	C++	local search	availability anounced
`Localizer++`	C++	local search	

Table 4.

The main issue of most of these libraries is rather simplicity and extensibility than efficiency. It is difficult to compare the performances since results have been published by the libraries on different problems. We have analyzed in Section 4.1 the published results on graph coloring problems. They are often not very competitive.

6 Conclusion

This paper has presented a new C++ library for incomplete combinatorial optimization. We have implemented several local search, and original and efficient population-based algorithms. A great effort has been done for the neighborhood management. An important issue is the incrementality in the configuration evaluations. We have obtained it by maintaining a data structure specific to conflicts. Another characteristics is our parameterized move selection process which will hopefully improve existing meta-heuristics

We think that no incomplete algorithm can efficiently solve all the problems. So it is important to test rapidly different algorithms, different neighborhoods. Such a library permits it and we have obtained good results for `CELAR` frequency assignment problems with `GWW-grw` and for graph coloring problems with `GWW-grw` or Metropolis, with a `min-conflict` or a `var-conflict` neighborhood, depending on the instance. We have, for the first time, colored the `flat300_28` instance with 30 colors.

The first version of `INCOP` is available upon request.

References

1. D. T. Connolly. An improved annealing scheme for the QAP. *European Journal of Operational Research*, (46):93–100, 1990.
2. S. de Givry, G. Verfaillie, and T. Schiex. Bounding the optimum of constraint optimization problems. In *Proc. of Constraint Programming CP97*, number 1330 in LNCS, 1997.
3. L. DiGaspero and A. Schaerf. Easylocal++ : An object oriented framework for flexible design of local search algorithms. Technical Report UDMI/13, Universita degli Studie di Udine, 2000.
4. Tassos Dimitriou and Russell Impagliazzo. Towards an analysis of local optimization algorithms. In *Proc. STOC*, 1996.
5. Raphaël Dorne and Jin-Kao Hao. Tabu search for graph coloring, t-colorings and set t-colorings. In I.H. Osman S. Voss, S.Martello and C. Roucairol, editors, *Meta-heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 77–92. Kluwer Academic Publishers, 1998.
6. G. Dueck and T. Scheuer. Threshold accepting : a general purpose algorithm appearing superior to simulated annealing. *Journal of Computational Physics*, 90:161–175, 1990.
7. Philippe Galinier and Jin-Kao Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3(4):379–397, 1999.
8. F. Glover. Future paths for integer programming and links to artificial intelligence. In *Computers and Operation Research*, volume 13, pages 533–549, 1986.
9. F. Glover. Personal communication, 2003.
10. F. Glover and M.Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.
11. A. Kolen. A genetic algorithm for frequency assignment. Technical report, Universiteit Maastricht, 1999.
12. A. Koster, C. Van Hoesel, and A. Kolen. Solving frequency assignment problems via tree-decomposition. Technical Report 99-011, Universiteit Maastricht, 1999.
13. François Laburthe and Yves Caseau. SALSA: A language for search algorithms. In *Proc. CP'98*, number 1520 in LNCS, pages 310–324, 1998.
14. L. Michel and P. Van Hentenryck. Localizer++ : An open library for local search. Technical Report CS-01-02, Brown University, 2001.
15. S. Minton, M. Johnston, A. Philips, and P. Laird. Minimizing conflict: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.
16. C. Morgenstern. Distributed coloration neighborhood search. In D. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability*, volume 26 of *dimacs*, pages 335–357. American Mathematical Society, 1996.
17. Bertrand Neveu and Gilles Trombettoni. When local search goes with the winners. In *Proc. of CPAIOR'03 workshop*, 2003.
18. P. K. Nielsen. *SCOOP 2.0 Reference Manual*. SINTEF Report 42A98001, 1998.
19. V. Phan and S. Skiena. Coloring graphs with a general heuristic search engine. In *Computational Symposium of Graph Coloring and Generalizations*, 2002.
20. B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proc. AI'92*, pages 440–446, 1992.
21. S. Voß and D. Woodruff. *Optimization Software Class Libraries*. Kluwer, 2002.
22. S. Voß and D.L. Woodruff. Hotframe: A heuristic optimization framework. [21], pages 81–154.

23. Stefan Voß and David L. Woodruff. Optimization software class libraries. [21], pages 1–24.
24. C. Voudouris and R.Dorne. Integrating heuristic search and one-way constraints in the iOpt toolkit. [21], pages 177–192.
25. C. Voudouris and E. Tsang. Solving the radio link frequency assignment problem using guided local search proceedings. In *NATO Symposium on Radio Length Frequency Assignment, Sharing and Conservation Systems (Aerospace)*, 1998.