INCOP: An Open Library for INcomplete Combinatorial OPtimization

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 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 carefully studied. 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 well-known benchmarks. 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, that cannot prove the optimality of a solution, but 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.

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 [12]. We can cite `iOpt` [14] by British Telecom, `SCOOP` [10] by SINTEF, `Localizer++` [6] at Brown University, `Hotframe` [13] at University of Braunschweig, `Discropt` [11] at State University of New York. Philippe Galinier and Jin-Kao Hao [4] 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 [2], that implements local search methods.

Initially, we wanted to test a new population-based method and compare it with local search methods in the same implementation. In that purpose, we decided to provide a free library, implementing the main local search metaheuristics and efficient population-based methods.

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. The main classes are: `OpProblem`, `Algorithm`, `Configuration`, `Move` , `NeighborhoodSearch`, `Metaheuristic`. It is then not difficult to define new meta-heuritics, new neighborhoods or new problems by defining subclasses.

The most popular local search metaheuristics are implemented such as Hill Climbing, `GSAT`, Simulated Annealing, Tabu Search. For adding a new metaheuristic, one has to define a subclass of `Metaheuristic`, with its data, an `acceptance` condition of a candidate move and a `executebeforemove` method for updating the meta-heuristic data (like the temperature of simulated annealing or the tabu list) before executing a move.

A configuration is represented by a fixed set of integer variables, with a priori known domains of values. Important combinatorial optimization problems, as traveling salesman problems (TSP) can be encoded in this framework. Constraint Satisfaction Problems (CSP) are transformed into MAX-CSP optimization problems for which the number of violated constraints (or more generally a criterion computed on these violations) is minimized. We have implemented several CSPs, including graph coloring and frequency assignment problems.

Adding a new problem. The criterion to be optimized is specific to a given problem. Three methods compute this criterion. `config_evaluation` evaluates the cost of an initial configuration; `move_evaluation` performs the incremental evaluation of a move; `update_conflicts` updates the `conflicts` data structure of a configuration when a move is executed.

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 local search algorithms.

3.1 Incrementality

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 evaluation is performed very often, it is crucial to rapidly evaluate the impact of a move on the whole configuration cost. We provide 3 manners to manage the conflicts, implemented by 3 classes.

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 `conflicts`; we need to compute only the contribution of the new value.
3. In `FullincrCSPconfiguration`, all the contributions of all possible values are stored in `conflicts`; the evaluation of a move is immediate.

The incremental evaluations are performed by the two following methods: `move_evaluation` is called when a move is tested, and `update_conflicts` when a move is performed. With full incrementality, the computation effort is mainly done in `update_conflicts`. It is fruitful when a lot of moves must be tested before accepting one. When the problem is sparse as in most of graph coloring instances, the updating is not costly. It only concerns the values of the few variables linked by a constraint with the currently changed variable. Full incrementality can save an order of magnitude in computing time. The memory required is also reasonable for coloring problems: the size of the conflict data structure is $N \times D$, where N is the number of nodes and D the number of colors.

3.2 Go With the Winners algorithms

The population-based algorithms implemented in `INCOP` are variants of the Go With the Winners algorithm [3]. 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. `GWW-grw` [9], a hybridization of `GWW` with a simple walk algorithm, has given very good results.

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, a kind of candidate list strategy [5], has been embedded in `INCOP`. First, the method `is_feasible` gives a feasibility 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, 3 parameters are used:

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 indicates how to select a configuration: either the best feasible or any feasible among the `Max_neighbors` visited neighbors.

These parameters allow us to implement many different classical behaviors as searching the best neighbor in the entire neighborhood or the first acceptable neighbor in a sample of K neighbors.

4 Experiments

We have performed experiments on difficult instances mainly issued from two categories of problems encoded as weighted MAX-CSPs: difficult graph coloring instances proposed in the DIMACS challenge, and CELAR frequency assignment problems[1]. All the tests have been performed on a `PentiumIII 935 Mhz`.

4.1 Graph coloring instances

Incomplete algorithms succeeded in coloring `flat300_28` in 31 colors [8].We colored it in 31 colors in a few minutes and in 30 colors in 1.6 hour using a Metropolis algorithm (i.e., simulated annealing with constant temperature [1]) and a neighborhood implementing the Min-conflicts heuristics [7].

	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 (number of colors, time); the results with `INCOP` in the right side (number of colors, number of conflicts (average on 10 trials), cpu time (average on 10 trials), success rate, algorithm and neighborhood used.

4.2 CELAR frequency assignment instances

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

Table 2. Results on `CELAR` frequency assignment benchmarks. The results of the best known algorithms are in the left side; the results with INCOP in the right side.

The constraints are of the form $|x_i - x_j| = \delta$ or $|x_i - x_j| > \delta$. The objective function is a weighted sum of violated constraints.

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

5 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 move evaluations. We have obtained it by maintaining a conflict data structure. Finally, we hope that our parameterized move selection process will 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. We have, for the first time, colored `flat300_28` with 30 colors.

References

1. D. T. Connolly. An improved annealing scheme for the QAP. *European Journal of Operational Research*, (46):93–100, 1990.
2. 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.
3. Tassos Dimitriou and Russell Impagliazzo. Towards an analysis of local optimization algorithms. In *Proc. STOC*, 1996.
4. Philippe Galinier and Jin-Kao Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3(4):379–397, 1999.
5. F. Glover and M.Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.
6. L. Michel and P. Van Hentenryck. Localizer++ : An open library for local search. Technical Report CS-01-02, Brown University, 2001.
7. 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.
8. 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.
9. Bertrand Neveu and Gilles Trombettoni. When local search goes with the winners. In *Proc. of CPAIOR'03 workshop*, 2003.
10. P. K. Nielsen. *SCOOP 2.0 Reference Manual*. SINTEF Report 42A98001, 1998.
11. V. Phan and S. Skiena. Coloring graphs with a general heuristic search engine. In *Computational Symposium of Graph Coloring and Generalizations*, 2002.
12. S. Voß and D. Woodruff. *Optimization Software Class Libraries*. Kluwer, 2002.
13. S. Voß and D.L. Woodruff. Hotframe: A heuristic optimization framework. [12], pages 81–154.
14. C. Voudouris and R.Dorne. Integrating heuristic search and one-way constraints in the iOpt toolkit. [12], pages 177–192.