Constraint Propagation as the Core of Local Search

Nikolaos Pothitos, George Kastrinis, and Panagiotis Stamatopoulos

Department of Informatics and Telecommunications, University of Athens,
Panepistimiopolis, 157 84 Athens, Greece
{pothitos,g.kastrinis,takis}@di.uoa.gr

Abstract. Constraint programming is a powerful paradigm for solving constraint satisfaction problems, using various techniques. Amongst them, local search is a prominent methodology, particularly for large instances. However, it lacks uniformity, as it includes many variations accompanied by complex data structures, that cannot be easily brought under the same "umbrella." In this work we embrace their wide diversity by adopting propagation algorithms. Our *constraint based local search* (CBLS) system provides declarative alternative tools to express search methods, by exploiting conflict-sets of constraints and variables. Their maintenance is straightforward as it does not employ queues, unlike the state of the art CBLS systems. Thus, the propagation complexity is kept linear in the number of changes required after each assignment. Experimental results illustrate the capabilities, not only of the already implemented methods, such as hill climbing, simulated annealing, etc., but also the robustness of the underlying propagation engine.

Keywords: constraint based local search, constraint programming, constraint satisfaction problem, solver, indirect methods, metaheuristics.

1 Introduction

Constraint programming (CP) is nowadays a well-established Computer Science field, that facilitates the expression of contemporary or difficult problems and, on the other hand, solves them through generic search methods. What makes this approach unique is not only the independence between the problem description and solution processes, but also the plethora of the solving mechanisms that one may leverage on; *constraint based local search* (CBLS) is one of them.

While in constructive search we build a solution to a *constraint satisfaction problem* (CSP) from scratch and take care to satisfy every constraint after each assignment, CBLS solvers assemble a *candidate* solution, and then try to fix it, by eliminating conflicting sets of variables and constraints.

Recent work on the area includes KANGAROO, a CBLS system that appeared only in 2011 [11]. It is presented as a more efficient alternative to the COMET platform [23]. Both CBLS systems internally employ queues in the constraint propagation and externally provide high level control structures and interfaces, that permit their use by inexperienced users, although it is not easy for the

I. Maglogiannis, V. Plagianakos, and I. Vlahavas (Eds.): SETN 2012, LNAI 7297, pp. 9–16, 2012.
© Springer-Verlag Berlin Heidelberg 2012

local search method programmer to surpass the already implemented variants and access immediately the conflict sets.[1] iOpt toolkit offers many local search variants, but does not favour internal methods reprogramming [24].

Previous frameworks, like EasyLocal++ [4] and HotFrame [6] are flexible for the design of new local search methods, but the CSP description is effortful for the average user, as new C++ classes have to be built. They effectively implement local search but it is not bridged with other famous paradigms, such as constructive search. Last but not least, there are also CBLS solvers that are specialized only for specific problems like SAT [12].

A CBLS system should support the design and implementation of most local search variants by facilitating the problem description and by allowing the user/programmer to access every conflict set. Our contribution focuses on these two aspects. First of all, we provide an expressive mechanism to state CSPs by using Naxos Solver, a constraint programming platform [13], that supports constructive search as well. And second, we build generic conflict sets that are updated after each assignment. Our constructs are theoretically defined, algorithmically supported and experimentally tested for solving CSPs.

2 CSPs and Multidisciplinary Contributions

Constraint satisfaction problems appear in many areas not only in Computer Science, but in daily routine too. Common problems such as timetabling for educational institutes are now easily formulated as CSPs [20] and efficiently solved via Constraint Programming [15], while many new CSPs come from Bioinformatics [1]. A known interdisciplinary CSP is the satisfiability problem [21].

Definition 1. *A* CSP *consists of the following triptych* [22]:

- *Constrained* Variables *that compose the set* $\mathscr{X} = \{X_1, X_2, \ldots, X_n\}$.
- Domains *of the variables that make up the set* $\mathscr{D} = \{D_{X_1}, D_{X_2}, \ldots, D_{X_n}\}$. *In this work it is presumed that each domain is a finite set of integers.*
- Constraints *between the variables, composing the set* $\mathscr{C}$. *Each* C_i *in* $\mathscr{C}$ *is a relation between the variables of a set* $S_i \subseteq \mathscr{X}$. *Formally, we define* $C_i = (S_i, T_i)$, *where* $T_i \subseteq D_{i_1} \times D_{i_2} \times \cdots \times D_{i_q}$ *is the set with all the allowed combinations for the variables in* $S_i = \{X_{i_1}, X_{i_2}, \ldots, X_{i_q}\}$.

When every domain becomes singleton, in other words when each constrained variable "equals" a specific value, we have an *assignment.* If an assignment satisfies the constraints of the problem, it is also a *solution.*

2.1 Solving Phase and Thrashing

After a CSP description, we select a procedure to seek a solution. There are *direct search methods* that *construct* a solution step by step, by assigning a value to a variable each time, that is why they are also called *systematic* [14].

[1] We did not analyze Comet further, because it has not sufficient implementation details (see also [11]), whilst the Kangaroo executable is currently unavailable.

But when it comes to solve large-scale instances, constructive search suffers by the so-called *combinatorial explosion* that ends up in *thrashing,* because of the many steps and constraints checks required to build a solution.

2.2 Local Search and Variants

An alternative is to start with an assignment and to iteratively try to *repair* it, in order to satisfy the constraints a posteriori [8].

Definition 2. *In the general case, an* assignment *or* location L *is a non-empty subset of* $D_{X_1} \times D_{X_2} \times \cdots \times D_{X_n}$. *If the assignment contains more than one tuples, i.e. if* $|L| > 1$, *then it is called a* partial assignment.[2]

Example 1. Let $\mathscr{D} = \{D_{X_1}, D_{X_2}, D_{X_3}\}$ and $D_{X_1} = D_{X_2} = D_{X_3} = \{0, 1, 2\}$. The location $L_1 = \{1\}\times\{1, 2\}\times\{0\}$ is a partial assignment, while $L_2 = \{1\}\times\{1\}\times\{0\}$ is a complete assignment—with all the domains made singleton.

In *local search* we begin with an initial assignment L_{init} and, if it is not a solution, we modify it, so as to move on to an improved assignment L'.

Definition 3. *A* neighborhood *for a (complete or partial) assignment* L *is a set* $\mathsf{N}(L)$ *with all the* possible *successors of* L *in the search space. The* step function $\mathsf{step}(L)$ *is used to return the* specific *successor of* L, *with* $\mathsf{step}(L) \in \mathsf{N}(L)$.

Each local search variant is described by its neighborhood and step function.

Hill Climbing. A well-known variant is *hill climbing* (HC), also known as *iterative improvement* [3]. Normally, its neighborhood $\mathsf{N}(L)$ contains the locations L' which differ in one variable assignment with regard to L (*1-exchange*).

The step functions of HC variants usually employ an eval-uation function that quantifies each location quality. So, the $\mathsf{step}(L)$ function selects a location L' with $\mathsf{eval}(L') < \mathsf{eval}(L)$. To define this metric we utilize the conflict set notion.

Definition 4. *We have three* conflict set *kinds:*

- $\mathsf{CS}(\mathscr{C})$ *consists of the constraints in* $\mathscr{C}$, *violated by the current assignment.*
- $\mathsf{CS}(\mathscr{C}, X)$ *contains the constraints in* $\mathsf{CS}(\mathscr{C})$ *that refer to the variable* X.
- $\mathsf{CS}(\mathscr{X})$ *is composed of all the variables* $X \in \mathscr{X}$, *with* $\mathsf{CS}(\mathscr{C}, X) \neq \emptyset$.

An ordinary measure for $\mathsf{eval}(L)$ is $|\mathsf{CS}(\mathscr{C})|$. E.g. in the *iterative best improvement,* we select the location L' in $\mathsf{N}(L)$ with the minimum $|\mathsf{CS}(\mathscr{C})|$.

Simulated Annealing. The above practice is prone to be trapped into *local minima* of the evaluation function. In this case we need a *meta-heuristic* to escape the current local minimum by making a random step. *Simulated annealing* (SA) was introduced in 1983 as one of the first meta-heuristics [10].

SA permits random steps—to skip local minima—while a parameter called *temperature* is high; as time passes by and temperature drops, the method becomes less tolerant in random steps, especially if their evaluation is poor.

---

[2] In Definition 2 we extended the more traditional definition: "*partial assignment* is an assignment where not all variables are given values."

3 An Augmented CSP Schema

To cope with the needs of a generic CBLS system, that integrates the above methods and supports the design of new ones, an enhanced CSP outline is suggested by extending the variable notion. At first, the variables in $\mathscr{X}$, specified in the CSP description, are marked as *non-intermediate;* then $\mathscr{X}$ is augmented by adding into it the implied *intermediate* variables. These variables do not alter the CSP semantics; they are completely auxiliary, as they satisfy the following.

Property 1. If a constrained variable $Y \in \mathscr{X}$ is *intermediate* (in other words, if it is *invariant* or *dependent*) *with regard to* the variables $\{X_{i_1}, X_{i_2}, \ldots, X_{i_m}\}$, it holds that $|D_{X_{i_1}}| = 1 \wedge |D_{X_{i_2}}| = 1 \wedge \cdots \wedge |D_{X_{i_m}}| = 1 \implies |D_Y| = 1$.

Example 2. Let $\mathscr{X} = \{X, Y\}$ and $\mathscr{C} = \{Y = X + 1\}$, where Y is intermediate. Due to Property 1, the variable Y is intermediate *with regard to* X, because if we instantiate X (i.e. if we set $D_X = \{3\}$), the domain D_Y must be also made singleton ($D_Y = \{4\}$) in order to satisfy the constraint.

In some cases Property 1 may hold for non-intermediate variables too. Thus, it is *not* Property 1 that makes a variable intermediate; CP solvers automatically generate intermediate variables, to produce amalgamated constraints.

3.1 Intermediate Variables in the N-Queens Problem

Take for example the N-Queens problem; the goal here is to place N queens on a $N \times N$ chessboard, so that no two queens "attack" each other.

Problem 1. In the normal chessboard we have $\mathscr{X} = \{X_1, X_2, \ldots, X_8\}$. Each X_i corresponds to the queen on the ith *line.* The values in D_{X_i} are the possible *columns* to place the queen on. The constraint *"no X_i attacks X_j"* is decomposed into the constraints triplet $X_i \neq X_j \wedge X_i + i \neq X_j + j \wedge X_i - i \neq X_j - j$ [19].

The above three sub-constraints, imply that no two queens share the same column, minor diagonal, and principal diagonal respectively. These can be stated in most CP solvers [5,9] as three *individual* constraints, while the compound statement *"X_i does not attack X_j"* is only used in theoretical bibliography, where ad hoc and not generic constraints often occur. Besides, the inequality constraint ($\neq$) is reusable in many CSPs [7], while the complex constraint *"X_i does not attack X_j"* is not generic. Figure 1 depicts the automatically generated *intermediate* variables X_i', X_i'', X_j', and X_j'' that are eventually added into $\mathscr{X}$.

3.2 Issues in Local Search

Intermediate variables facilitate the CSPs specification and adapt smoothly to direct search methods. However, they do not "fit" well in local search, as it does not usually incorporate constraint propagation [18], that is mainly integrated in systematic search [17]. Actually, in local search, when we assign a value to a

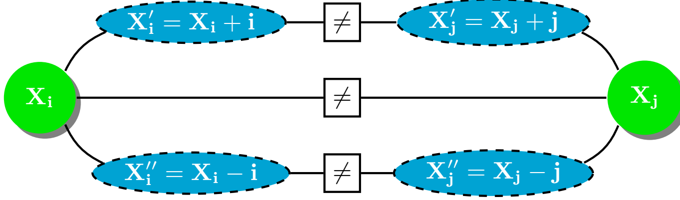

Fig. 1. Non-intermediate and intermediate (*dotted*) variables in the constraint network

variable, the assignment is not propagated to its dependent intermediate variables; their assignment is dependent exclusively to the heuristics and the declared neighborhood and step functions (cf. Definition 3).

Another issue is that the conflict sets are made inefficient to maintain, due to the significant number of redundant (intermediate) variables in them. This makes it also difficult for the heuristics to select the next move to go on, out from a cumbersome set—with the intermediate variables included in it.

3.3 Conflicting Assignments and Violated Constraints

In the new schema, we make the conflict sets transparent to invariants.

Proposition 1. *If a constraint which involves the variables $X_{i_1}, \ldots, X_{i_m}$ is violated, then it holds* $\mathsf{depend}(X_{i_1}) \cup \cdots \cup \mathsf{depend}(X_{i_m}) \subseteq \mathsf{CS}(\mathscr{X})$*, where* $\mathsf{depend}(X)$ *is the set of* non-intermediate *variables that the intermediate X depends on. If X is not intermediate, then we suppose* $\mathsf{depend}(X) = \{X\}$*.*

With the above proposition we can identify a composite constraint, even if we do not know about its inherent sub-constraints and intermediate variables. For instance, in this work we label each constraint with the *non-intermediate* variables it depends on. Consequently, the conflict sets $\mathsf{CS}(\mathscr{C})$ and $\mathsf{CS}(\mathscr{C}, X)$ contain tuples of variables that are involved into the corresponding violated constraints.

4 Unqueued Constraint Propagation

In light of this theoretical background we designed lightweight algorithms for the assignment propagation and conflict sets maintenance. The assignment of a value to a variable is the focal point of our framework.

Figure 2 illustrates what happens when we ASSIGN a *value* to X. If X is already bound to *another* value, then there is a conflict; we build the *conflictTuples* and add them to $\mathsf{CS}(\mathscr{C})$. Note that $\mathsf{CS}(\mathscr{C}, X)$ and $\mathsf{CS}(\mathscr{X})$ are also updated.

Each intermediate variable has its own *supportTuples* containing the sets of the variables that support its current assignment. When there is a conflict, $X.\mathit{supportTuples}$ collides with *supportVars*, i.e. the variables that fired the assignment. Hence, the conflict set here is $X.\mathit{supportTuples} \times \{\mathit{supportVars}\}$. We used the Cartesian product, because we may have multiple *supportTuples* for an

```
procedure X.ASSIGN(value, supportVars)
if X is assigned another value then
    conflictTuples ← {{}}
    X.GETSUPPORT(conflictTuples)
    for each Y ∈ supportVars do
        Y.GETSUPPORT(conflictTuples)
    end for
    Add conflictTuples to CS(𝒞)
else
    Commit the assignment of value to X
    Add supportVars to X.supportTuples
    for each c ∈ 𝒞, with X involved do
        c.FIXEDCONS( )
    end for
end if
end procedure
```

```
procedure X.GETSUPPORT(csTuples)
for each tuple ∈ X.supportTuples do
    suppTuple ← {{}}
    for each Y ∈ tuple do
        suppY ← {{}}
        Y.GETSUPPORT(suppY)
        suppTuple ← suppTuple × suppY
    end for
    csTuples ← csTuples × suppTuple
end for
if X is non-intermediate then
    csTuples ← csTuples × {{X}}
end if
end procedure
```

Fig. 2. Algorithms to propagate assignments and to update conflict sets

assignment. But before inserting the product into $\mathsf{CS}(\mathscr{C})$, we must "dig" into it to find the non-intermediate variables it depends on, in view of Proposition 1; this is performed via GETSUPPORT, a recursive function in Fig. 2.

If ASSIGN is called by the user/programmer, *supportVars* is empty and the assignment is permitted in any case. ASSIGN may be also called inside FIXEDCONS, a constraint-specific procedure which imposes *fixed consistency.*

Definition 5. *A constraint* $C_i = (S_i, T_i)^3$ *is* fixed consistent, *iff for each unassigned variable* $X_{i_m} \in S_i$, *there exist* at least two *values* $v_1, v_2 \in D_{X_{i_m}}$ *that satisfy it, i.e., formally,* $\left|T_i \cap \left(D_{i_1} \times \cdots \times \{v\} \times \cdots \times D_{i_q}\right)\right| \geq 1,\ v \in \{v_1, v_2\}$.

FIXEDCONS imposes fixed consistency w.r.t. C_i, by making singleton every variable in S_i that has only one value supported by the rest of the variables in S_i. Apparently, each constraint type has its own FIXEDCONS implementation.

5 Empirical Results

On top of the above algorithmic ground, we implemented several local search variants, including hill climbing and simulated annealing, outlined in Sect. 2.2. We integrated them into NAXOS SOLVER, a library for an object-oriented programming environment, written in C++ [13].

We used the above methods to find a solution to the N-Queens (Problem 1), on a Dell computer with an Intel Pentium D 2.8 GHz dual-core processor and 1 GB of memory, running Ubuntu 8.04.[4] Figure 3 depicts the performance of the

[3] See also Definition 1.

[4] The experiments code is available at `http://di.uoa.gr/~pothitos/setn2012` with other instances, such as *graph coloring,* also solved.

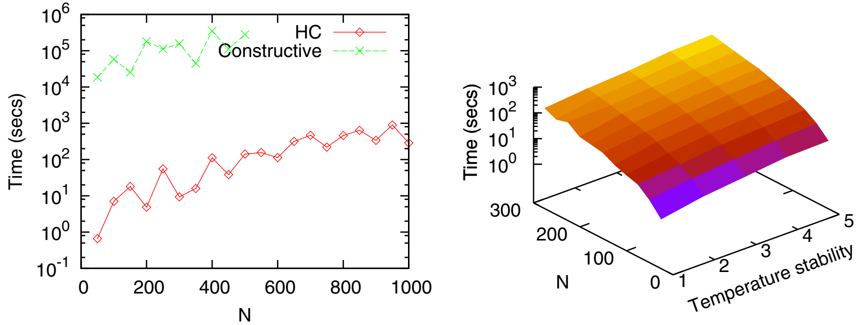

Fig. 3. Solving N-Queens using Constructive Search, Hill Climbing (*left*), and Simulated Annealing (*right*)

two methods, while the problem scales. In the *left* subfigure, constructive search time is also shown; besides, Naxos Solver is capable of using it to solve exactly the same instances, as we did not need to modify the problem descriptions. Nevertheless, local search is orders of magnitude faster than constructive search. ECL^{i}PS^e 5.10 (Interval Constraints) constructive search gave even slower results.

Simulated Annealing performance depends on how fast the "temperature" drops. A low temperature means less random moves. In Fig. 3 the *temperature stability* factor defines for how many steps the temperature will remain the same. In these instances a rapid temperature decrease gives better results.

6 Conclusions and Future Directions

Our goal is to provide a freely available flexible platform—implemented as a C++ library—for both the specification of a problem and the design of local search methods. Beyond facilitating the compound constraints expression, intermediate variables were the key feature for propagating assignments.

A future direction is to enrich the available methods, by adopting e.g. genetic algorithms. It will be also interesting and easy, to describe a methodology for exploiting Naxos hybrid framework to mix direct and indirect methods [2].

Acknowledgements. Nikolaos Pothitos is financially supported by a Bodossaki Foundation Ph.D. scholarship.

The work infrastructure was supported by the Special Account Research Grants of the National and Kapodistrian University of Athens, in the context of the project "C++ Libraries for Constraint Programming" (project no. 70/4/4639).

References

1. Barahona, P., Krippahl, L., Perriquet, O.: Bioinformatics: A challenge to constraint programming. In: Van Hentenryck, P., Milano, M. (eds.) Hybrid Optimization. Springer Opt. and Its Applications, vol. 45, pp. 463–487. Springer, New York (2011)

2. Chatzikokolakis, K., Boukeas, G., Stamatopoulos, P.: Construction and Repair: A Hybrid Approach to Search in CSPs. In: Vouros, G.A., Panayiotopoulos, T. (eds.) SETN 2004. LNCS (LNAI), vol. 3025, pp. 342–351. Springer, Heidelberg (2004)
3. Cohen, W., Greiner, R., Schuurmans, D.: Probabilistic hill-climbing. In: Hanson, S.J., Petsche, T., Kearns, M., Rivest, R.L. (eds.) Comp. Learn. Theory and Natural Learn. Syst., vol. II, pp. 171–181. The MIT Press, Cambridge (1994)
4. Di Gaspero, L., Schaerf, A.: EasyLocal++: An object-oriented framework for the flexible design of local-search algorithms. S/W Pract. Exp. 33(8), 733–765 (2003)
5. ECL^{i}PS^e constraint programming system (2011), `http://eclipseclp.org`
6. Fink, A., Voß, S.: HotFrame: A heuristic optimization framework. In: Voß, S., Woodruff, D. (eds.) Opt. S/W Cl. Lib., vol. 18, pp. 81–154. Kluwer, Boston (2002)
7. Gent, I.P., Walsh, T.: CSPLib: A Benchmark Library for Constraints. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 480–481. Springer, Heidelberg (1999), `http://CSPLib.org`
8. Hoos, H.H., Tsang, E.: Local search methods. In: Rossi, et al. (eds.) [16], ch. 5, pp. 135–167
9. Ilog Solver (2010), `http://ilog.com/products/cp`
10. Kirkpatrick, S., Gelatt Jr., C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220(4598), 671–680 (1983)
11. Hakim Newton, M.A., Pham, D.N., Sattar, A., Maher, M.: Kangaroo: An Efficient Constraint-Based Local Search System Using Lazy Propagation. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 645–659. Springer, Heidelberg (2011)
12. Pham, D.N., Thornton, J., Sattar, A.: Building structure into local search for SAT. In: Veloso, M.M. (ed.) IJCAI 2007, pp. 2359–2364. AAAI Press, Menlo Park (2007)
13. Pothitos, N.: Naxos Solver (2011), `http://di.uoa.gr/~pothitos/naxos`
14. Prosser, P., Unsworth, C.: Limited discrepancy search revisited. Journal of Experimental Algorithmics 16(1), 1.6:1–1.6:18 (2011)
15. Qu, R., Burke, E.K., McCollum, B., Merlot, L.T.G., Lee, S.Y.: A survey of search methodologies and automated system development for examination timetabling. Journal of Scheduling 12(1), 55–89 (2009)
16. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Foundations of Artificial Intelligence. Elsevier Science, Amsterdam (2006)
17. Sabin, D., Freuder, E.C.: Contradicting Conventional Wisdom in Constraint Satisfaction. In: Borning, A. (ed.) PPCP 1994. LNCS, vol. 874, pp. 10–20. Springer, Heidelberg (1994)
18. Schulte, C., Stuckey, P.J.: Efficient constraint propagation engines. ACM Transactions on Programming Languages and Systems 31(1), 2:1–2:43 (2008)
19. Smith, B.M.: Modelling. In: Rossi, et al. (eds.) [16], ch. 11, pp. 377–406
20. Stamatopoulos, P., Viglas, E., Karaboyas, S.: Nearly optimum timetable construction through CLP and intelligent search. Int'l J. on AI Tools 7(4), 415–442 (1998)
21. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. Constraints 14(2), 254–272 (2009)
22. Tsang, E.: A glimpse of constraint satisfaction. AI Review 13(3), 215–227 (1999)
23. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. Logic Programming. The MIT Press, Cambridge (2009)
24. Voudouris, C., Dorne, R., Lesaint, D., Liret, A.: iOpt: A Software Toolkit for Heuristic Search Methods. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 716–729. Springer, Heidelberg (2001)