EURO J Comput Optim (2014) 2:89–97
DOI 10.1007/s13675-014-0026-3

ORIGINAL PAPER

A constraint programming primer

Gilles Pesant

Received: 30 June 2014 / Accepted: 5 July 2014 / Published online: 22 July 2014

Abstract The purpose of this short paper is to give a reader acquainted with combinatorial optimization enough background in Constraint Programming to appreciate the other papers in this special issue. We will cover modeling, inference, and search. Whenever appropriate we will make an effort to relate some of the concepts to that of other computational approaches. Those interested to learn more about this computational approach will also find some pointers to the literature on recent advances in the area.

Keywords Constraint programming · Combinatorial optimization · Inference · Search heuristic

Mathematics Subject Classification 90-01

Introduction

The origins of Constraint Programming (CP) can be traced back as early as the 1960s, to research on human-machine interfaces (Sutherland 1963; Borning 1977), artificial intelligence (Montanari 1974; Freuder 1978), and declarative programming languages (Laurière 1978; Sussman and Steele 1980; Jaffar and Lassez 1987; Colmerauer 1990). In most general terms CP is the study of computational systems based on constraints. Its advent on the operations research scene, specifically to model and solve combinatorial problems, happened in the last twenty years. In this paper we shall restrict ourselves to the latter.

Like most other solution methods in that area CP expresses the problem at hand through formal mathematical modeling. Unlike them, the language in which the model

G. Pesant (✉)
École polytechnique de Montréal and CIRRELT, Montreal, Canada
e-mail: gilles.pesant@polymtl.ca

is formulated features high level primitives that explicitly expose much of the combinatorial structure of the problem. We sacrifice the simple syntax that creates an opportunity for highly optimized monolithic solvers, but gain the rich semantics that can, and has been, exploited for both inference and search. The distinctive driving force of CP has been this direct access to structure.

One simple manifestation of this structure is that CP works directly on discrete variables instead of relying on a continuous relaxation of the model. The set of available values for a variable, called its *domain*, is stored in a data structure and updated as the computation proceeds.[1] It plays a few very important roles: it acts as liaison between inference algorithms acting on different parts of the model, it can guide the search for solutions, and it signals when search should backtrack.

But most structure is captured in individual constraints. Each on its own may represent an important characteristic of the problem such as building a Hamiltonian cycle on a graph, packing items into bins, or scheduling tasks on a cumulative resource. In CP it is not unusual that a few constraints suffice to model a complex combinatorial problem. Each type of constraint encapsulates a dedicated inference algorithm that acts on the domains by filtering out inconsistent values, thereby reducing the search space.

That search space is explored by problem decomposition, building a search tree whose branches at a node correspond to decisions that partition the current search space. These decisions can be arbitrary constraints added to the model for that subtree but typically they fix a variable to a value in its domain in the left branch and remove that value from the domain in the right branch. Branching heuristics guide how the search tree is built.

At its core CP is designed to handle feasibility problems—consider that its problem representation formalism is termed *Constraint Satisfaction Problem*. Optimization problems are traditionally approached as a succession of feasibility problems, raising the bar for the objective value at each step until the last step which is a proof of optimality. However hybrid methods, combining CP and MIP or local search, have been applied increasingly to solve optimization problems and are often more effective than their component parts.

The rest of the paper covers in more detail some important aspects of CP: inference (Sect. 2), modeling (Sect. 3), and search (Sect. 4).

Inference

In CP a problem is represented using a finite set of discrete variables $X = \{x_1, x_2, \ldots, x_n\}$ each taking its value from a finite domain, $x_i \in D_i \subset \mathbb{Z},\ 1 \leq i \leq n$, and a finite set of constraints (i.e. relations) $C = \{c_1, c_2, \ldots, c_m\}$ each expressed on a subset of the variables (called its *scope*), $c_j(x_{j_1}, x_{j_2}, \ldots, x_{j_k}) \subset \mathbb{Z}^k,\ 1 \leq j \leq m$. One must find a combination of values from the domain of each variable that simultaneously satisfies every constraint (i.e. belongs to every relation). This formalism is called

[1] A promising alternative to this representation of the search space as individual domains is the more structure-rich multiple decision diagram (Hoda et al. 2010).

the *Constraint Satisfaction Problem*. Note that we could have written domains as constraints and even combined all constraints into one, but the above formulation is closer to the mechanics of CP solvers.

Conceptually variables and constraints are arranged in a network with the former as vertices and the latter as hyper-edges, incident with the variables in their scope. Vertices are labeled with the domain of the corresponding variable and these labels are updated as the computation proceeds. Looking locally at a particular hyper-edge (constraint), we potentially modify the domains of the incident vertices (variables) by removing values which cannot be part of any solution because they would violate that individual constraint. This *local consistency* step can be performed efficiently. The modification of a vertex's label triggers the re-evaluation of all incident hyper-edges, which in turn may modify other labels. This process stops when either all domain modifications have been dealt with or the empty domain is obtained, in which case no solution exists. Note that we never add values to a domain but only remove some inconsistent ones, and since domains are finite this process must terminate. The overall behavior is called *constraint propagation*. It is important to realize that each constraint has its own specialized *filtering algorithm* (also called *propagator*) to filter out values and that these algorithms interact through the domains of their shared variables.

One major strength of CP is what happens during local consistency, ranging from straightforward deletions to sophisticated inference. Several levels of consistency have been defined to describe formally which values are left in domains once local consistency is achieved—we describe the most common. They are all based on the concept of support. With respect to a given constraint, a *support* for value $d \in D_i$ being assigned to variable x_i consists of a satisfying combination of values for each other variable in that constraint's scope. The local consistency step removes unsupported values. If the values in a support are restricted to belong to the domain of the respective variables then removing every unsupported value achieves *domain consistency* (also called *generalized arc consistency* in some of the literature, for historical reasons). This is the best we can do while removing individual values from domains. In case it is too computationally expensive (typically we avoid the exponential cost of explicitly considering all combinations), we can opt for weaker but more time efficient local consistencies. Let $D^{\mathbb{R}} = \{d \in \mathbb{R} : \min(D) \leq d \leq \max(D)\}$, the smallest real interval containing every value in domain D. If instead the values in a support are restricted to belong to the relaxed domain $D^{\mathbb{R}}$ of the respective variables then we will talk of *domain* ($\mathbb{R}$) *consistency*. We see here an analogy with the use of continuous relaxations in integer programming: working with continuous domains lowers the computational effort, for example by allowing simple algebraic manipulations on numerical constraints over the reals. Going further, if instead of removing every unsupported value we only remove unsupported boundary values (i.e. the smallest and largest values in the domain) then we achieve *bounds* ($\mathbb{R}$) *consistency*, often simply called *bounds consistency*. This consistency level has been integrated in MIP solvers as part of *node preprocessing*. For some of the more complex combinatorial structures the above local consistency levels are sometimes out of reach and therefore we settle for inference rules that remove some unsupported values but whose consistency level is not characterized.

As mentioned before the filtering algorithms can be quite sophisticated but usually they are kept to a low polynomial time complexity. Because in depth-first search the state of the computation changes little from one call of the filtering algorithm to the next (see Sect. 4) one often designs them to be *incremental*, saving past work in data structures and re-establishing consistency from the current state instead of starting from scratch every time.

So the concept of filtering domains to achieve some local consistency is really based on feasibility reasoning, removing values that cannot lead to a feasible solution. But when solving a combinatorial optimization problem, there has also been a proposal to filter based on optimality reasoning. *Reduced-cost filtering* (Focacci et al. 1999) translates part of the CP model into a linear program and retrieves reduced costs which are used to remove values that cannot lead to an optimal solution, similar to *variable fixing* in MIP solvers.

Modeling

CP offers a very rich modeling language with families of constraints defined for many of the usual combinatorial structures. A comprehensive catalog of constraints proposed in the literature (but not necessarily supported by CP solvers) is being maintained[2]—we present here some of its main representatives and refer the reader to that catalog for details and pointers to the relevant literature.

Numerical constraints. Of course there are linear constraints, and in particular knapsack constraints, but one can also write nonlinear constraints, including those featuring trigonometric functions, absolute values, exponentials, etc. Generally we infer reduced domains from them using a unified approach that achieves bounds consistency but in some special cases such as knapsack constraints we may choose to achieve stronger domain consistency.

Functional constraints. The ability to describe a functional relationship between a pair of variables is a useful modeling construct. It has taken two syntactic forms in CP: the `element` constraint or, more subtly, indexing by a variable. In this way auxiliary variables may be defined relative to the main decision variables. For example they can represent the cost of each decision in the objective. Consider the Traveling Salesman Problem modeled in CP with one decision variable s_i for each city i, representing its successor in the tour. If c_{ij} gives the cost of traveling directly from city i to city j then we can define auxiliary variables through functional constraints $a_i = c_{is_i},\ 1 \le i \le n$ (or equivalently $\texttt{element}(s_i, c_i, a_i),\ 1 \le i \le n$) and write the objective as $\min z$ with $z = \sum_{i=1}^{n} a_i$ (it could also be written in a single step as $\min \sum_{i=1}^{n} c_{is_i}$). Domain consistency is maintained on such constraints.

Value occurrence constraints. These are perhaps the most important constraints, both from a historical and practical perspective. They restrict the number of occurrences of each value being assigned to a set of variables, typically by giving lower and upper bounds for each value. The most general member of this family is the global cardinality constraint (`gcc`) but other noteworthy members are `among`, concerned

[2] Global Constraint Catalog: http://sofdem.github.io/gccat/.

with occurrences from a given subset of values, and the ubiquitous `alldifferent`, restricting every value to occur at most once. Given their importance and the trade-off between the strength of the consistency level and the computational effort to achieve it, several filtering algorithms have been proposed for different consistency levels including domain and bounds consistency.

Value distribution constraints. The previous family could also be seen as constraining the discrete distribution of values among variables. One can also restrict the discrete distribution in a less detailed way, by acting on its mean and on the deviation of each value from the mean. The `spread` constraint is defined on a set of variables and on two others respectively equal to the mean value of the previous ones and on their standard deviation. The `deviation` constraint is similar except that the latter variable corresponds to the sum of the absolute value of the difference between each variable and the mean. These are very useful when seeking balance or fairness in solutions with respect to a given feature. Bounds consistency algorithms have been proposed, and more recently one achieving domain consistency for both constraints.

Sequencing constraints. The previous two families of constraints featured sets of variables but sometimes variables are ordered (e.g. representing a sequence of decisions over time) and one wishes to restrict the possible combinations of consecutive values taken by these variables. The `sequence` constraint (also known as `sliding_distribution`) constrains the number of occurrences of each value, much like the `gcc`, but only inside a sliding window of a given width over the sequence of variables. The `regular` constraint states that the respective values taken by the finite sequence of variables, seen as a "word", should belong to the regular language described by a given automaton. Many intricate sequencing rules can thus be enforced by specifying an automaton, often in a concise way. Domain consistency is achieved on the latter constraint.

Scheduling constraints. Resource scheduling has been a remarkably successful application area of CP. Given a set of tasks, each with a duration, requiring a resource that cannot be shared, the `disjunctive` constraint only allows a schedule for the tasks such that they do not overlap in time. The `cumulative` constraint generalizes this context to a resource of a given capacity and a certain amount of resource required by each task. Many rules from the OR literature (e.g. edge finding, timetabling, energetic reasoning) have been adapted, encapsulated in constraints, and applied dynamically during search to reduce domains. The end-result cannot be characterized by the consistency levels previously defined.

Packing constraints. Putting objects of various sizes into one or several given containers so that they do not overlap arises in many contexts. In one dimension this corresponds to a bin packing problem for which constraints such as `pack` and `multiknapsack` have been implemented. For higher dimensions there are the `diffn` and `geost` constraints. Filtering algorithms applied to these constraints include constructive disjunction and the sweep line technique from computational geometry. Here again the consistency levels we discussed do not apply.

Extensional constraints. Finally constraints for which no apparent structure can be exploited may be stated in extension as a set of allowed or forbidden tuples from a relation of given arity. They are often referred to as `table` constraints in CP solvers. Several efficient algorithms achieving domain consistency have been proposed for such extensional constraints.

Objective function. For optimization problems we typically define an additional variable, say z, representing the objective value and write a constraint to link it to the objective function (for an example of this, see the paragraph on functional constraints in this section). Whenever a solution is found, thereby fixing z to some value v, the constraint $z < v$ (if we are minimizing) is automatically added in order to constrain the search to improving solutions until none can be found, proving the optimality of the latest solution. Because filtering algorithms are a multiway process, domain changes for the decision variables can trigger changes in the objective variable's domain and changes in the latter, typically when we add a stronger bound, can in turn impact the decision variables.

Search

Since constraint propagation may stop with indeterminate variables (i.e. whose domain still contains several values), the solution process requires search. It usually takes the form of tree search in which one branches to resolve that indeterminacy either by shrinking variable domains directly or by triggering more constraint propagation through the addition of a constraint. In the following we present how CP organizes search by contrasting it with how MIP solvers do it.

Node selection. Node selection in CP often proceeds in a depth-first fashion. Jumping around freely among nodes at the frontier of the search tree would be too time- or space-consuming: consider that each constraint may have an internal data structure maintained by its filtering algorithm so the state of the computation can be quite large to store or long to restore from scratch. For that reason, and also because in essence a feasibility problem is being solved, rarely in CP would we select the next node to expand based on a bound on the objective. A moderate form of jumping that has gained popularity is *limited discrepancy search* (Harvey and Ginsberg 1995): leaves of the search tree are visited in order of increasing number of discrepancies, where a discrepancy in a path to a leaf corresponds to not following the recommendation of the value-selection heuristic (see below). This way we favor leaves reached by listening to the heuristic advice most of the time. A variant of limited discrepancy search breaks ties among equal discrepancy numbers by favoring discrepancies high in the search tree, the insight being that the heuristic is less informed at the top of the search tree (fewer decisions have been made; the search space is larger) so going against it is not as ill-advised. Of course the underlying assumption here is that we have a good heuristic to guide us. Another popular approach to recover from a bad decision made high in the search tree is *randomized restarts* (Gomes et al. 1998): once search has gone on for a while it is restarted from the top of the tree, having introduced some randomness in the heuristics to avoid repeating the same tree exploration.

Variable selection. Variable selection has been the subject of much research in CP. Some generic *variable-selection heuristics* have been proposed but most CP solvers also make it easy for users to design their own heuristic tailored to their problem. An early design principle, the *Fail-First Principle*, recommends to branch so that failure comes sooner than later. Though unintuitive on the surface, it recognizes the fact that search will take a wrong turn every once in a while and then escaping the failed subtree as quickly as possible will be very important (remember that we are likely using depth-first search). This principle's main instantiation has been the *smallest-domain-first* heuristic, which selects a variable with the fewest values remaining in its domain: because there aren't many choices left to fix that variable, there is a certain urgency to that decision and, in case we are in a failed subtree, the low fanout at that node may help build a small subtree. Other heuristics stemming from that principle include: *weighted degree* (Boussemart et al. 2004), favoring a variable appearing in many constraints that have often been instrumental in detecting a dead end during search so far; *impact-based search* (Refalo 2004), favoring a variable whose assignments cause the largest reduction in the size of the Cartesian product of the domains (inspired by pseudo-cost branching in MIP solvers); *activity-based search* (Michel and Hentenryck 2012), favoring a variable whose domain is reduced most often (inspired by the VSIDS heuristic in SAT solvers). Thus variable-selection heuristics are generally concerned with feasibility and not so much with optimization, with the notable exception of the *regret* heuristic which favors a variable with the largest difference between the best and second-best values in its domain according to the objective function.

Branching direction. Branching at a node can be enumerative, with a branch for each value in the domain of the selected variable. Alternatively when the domains are large *domain splitting* can be performed by partitioning the domain of a given variable among two or more branches. But branching is more often binary, fixing a variable to a value in its domain in the left branch and removing that value from the domain in the right branch. Thus the usual branching direction is the one fixing the variable. *Value-selection heuristics* have also generated interest in CP but to a lesser degree, perhaps underestimating their importance. Proceeding in simple lexicographic order is still popular. One could argue that the design principle here should be "Succeed First" and some of the previously mentioned heuristics integrating the choice of variable and value have followed it: impact-based search favors the value causing the smallest reduction in the size of the Cartesian product of the domains; regret favors the best value according to the objective function. Another heuristic, *counting-based search* (Pesant et al. 2012), follows the latter principle to choose both variable and value: it favors an assignment that appears most often in solutions to a constraint. This heuristic is also an instance of the combinatorial structure of individual constraints being exploited for search.

A noteworthy use of value selection is to break some value symmetries dynamically during search: among yet-unassigned interchangeable values, only one will be selected and the others will never be branched on at that node.

Learning from search. An important ingredient of several of the most popular variable-selection heuristics is learning from past search to guide future branching decisions. And if restarts are used the learnt information is kept. Weighted degree learns a constraint's weight by incrementing it every time its filtering algorithm empties a domain. Activity-based search proceeds similarly with a variable's weight every time its domain is reduced. Impact-based search stores the observed impact of branching decisions as search proceeds and uses it in other parts of the search tree after backtracking instead of computing impacts at every node.

A recent and very promising combination of the high-level modeling capability of CP and the efficiency of SAT solvers is *Lazy Clause Generation* (Ohrimenko et al. 2009). Every time a dead end is reached, a short explanation for it is learned and added as a clause to the SAT model. Here again the combinatorial structure of constraints can be exploited to generate such explanations.

Hybrid approaches. A growing number of applications of constraint programming do not use straight CP tree search but hybridize it with some other combinatorial optimization approach. CP-based *Branch and Price* (Junker et al. 1999) uses CP to generate columns (i.e. solve the pricing subproblem). It takes advantage of the fact that column generation can be cast as a feasibility problem since one must generate negative reduced cost columns but not necessarily the most negative one. It also brings flexibility to handle several potentially complex constraints in the subproblem. For CP it provides a proven solving approach for very large instances. Logic-based *Benders Decomposition* (Hooker 2005) has been a very successful hybridization framework for CP, which has been used alternatively in the master problem and in the subproblem. *Large Neighborhood Search* (Shaw 1998) is conceptually a local search method that explores a neighborhood through CP tree search. Operationally at each iteration some fraction of the variables in the CP model are fixed to their value in the current solution while the others are constrained to their initial domain, defining the neighborhood to search over.

Conclusion

This year marks the twentieth anniversary of Constraint Programming's main conference and next year will mark the same anniversary for its main journal, *Constraints*. The field has made it to adulthood, choosing to go down some paths and abandoning others along the way. It seems clear that solving combinatorial problems will remain a strong drive and its association with other approaches toward the same goal is bringing new and exciting ways to achieve it.

References

Borning A (1977) ThingLab—an object-oriented system for building simulations using constraints. In: Reddy R (ed) IJCAI, William Kaufmann, pp 497–498

Boussemart F, Hemery F, Lecoutre C, Sais L (2004) Boosting systematic search by weighting constraints. In: de Mántaras RL, Saitta L (eds) ECAI. IOS Press, Amsterdam, pp 146–150
Colmerauer A (1990) An introduction to prolog III. Commun ACM 33(7):69–90
Focacci F, Lodi A, Milano M (1999) Cost-based domain filtering. In: Jaffar (1999), pp 189–203
Freuder EC (1978) Synthesizing constraint expressions. Commun ACM 21(11):958–966
Gomes CP, Selman B, Kautz HA (1998) Boosting combinatorial search through randomization. In: Mostow J, Rich C (eds) AAAI/IAAI. AAAI Press, Menlo Park, pp 431–437
Harvey WD, Ginsberg ML (1995) Limited discrepancy search. In: IJCAI (1), Morgan Kaufmann, Burlington, pp 607–615
Hoda S, van Hoeve WJ, Hooker JN (2010) A systematic approach to MDD-based constraint programming. In: Cohen D (ed) CP, Lecture notes in computer science, vol 6308. Springer, Berlin, pp 266–280
Hooker JN (2005) A hybrid method for planning and scheduling. Constraints 10(4):385–401
Jaffar J (ed) (1999) Principles and practice of constraint programming—CP'99, 5th international conference, Alexandria, Virginia, USA, October 11–14, 1999, Proceedings. Lecture notes in computer science, vol 1713. Springer, Berlin
Jaffar J, Lassez JL (1987) Constraint logic programming. In: POPL. ACM Press, New York, pp 111–119
Junker U, Karisch SE, Kohl N, Vaaben B, Fahle T, Sellmann M (1999) A framework for constraint programming based column generation. In: Jaffar (1999), pp 261–274
Laurière JL (1978) A language and a program for stating and solving combinatorial problems. Artif Intell 10(1):29–127
Michel L, Hentenryck PV (2012) Activity-based search for black-box constraint programming solvers. In: Beldiceanu N, Jussien N, Pinson E (eds) CPAIOR. Lecture notes in computer science, vol 7298. Springer, Berlin, pp 228–243
Montanari U (1974) Networks of constraints: fundamental properties and applications to picture processing. Inf Sci 7:95–132
Ohrimenko O, Stuckey PJ, Codish M (2009) Propagation via lazy clause generation. Constraints 14(3): 357–391
Pesant G, Quimper CG, Zanarini A (2012) Counting-based search: branching heuristics for constraint satisfaction problems. J Artif Intell Res (JAIR) 43:173–210
Refalo P (2004) Impact-based search strategies for constraint programming. In: Wallace M (ed) CP, Lecture notes in computer science, vol 3258. Springer, Berlin, pp 557–571
Shaw P (1998) Using constraint programming and local search methods to solve vehicle routing problems. In: Maher MJ, Puget JF (eds) CP, Lecture notes in computer science, vol 1520. Springer, Berlin, pp 417–431
Sussman GJ, Steele GL (1980) CONSTRAINTS—a language for expressing almost-hierarchical descriptions. Artif Intell 14(1):1–39
Sutherland IE (1963) Sketchpad, a man–machine graphical communication system. Outstanding dissertations in the computer sciences, Garland Publishing, New York