Constraint Programming and the CSP

Constraint Programming (CP) [8] is a declarative style of programming where problems are modelled as a CSP, i.e., as a set of variables that have to be assigned values from those variables' domains to satisfy a set of constraints. Values might typically be integers drawn from finite domains, real numbers from ranges, or more complex entities like sets or graphs. We will only be considering integers.

Definition 1. *A* constraint satisfaction problem *(CSP) is a triple* (V, D, C) *where* V *is a set of* n variables $\{v_1, \ldots, v_n\}$*;* $D = \{dom(v_1), \ldots, dom(v_n)\}$ *is a collection of* domains*, each a totally ordered set of integer values; and* $C = \{c_1, \ldots, c_e\}$ *is a set of* e constraints*, each with a* scope *of variables* $scope(c) = (v_{c_1}, \ldots, v_{c_k})$ *and a* relation $rel(c) \subseteq dom(v_{c_1}) \times \ldots \times dom(v_{c_k})$*. An assignment of value* $x \in dom(v)$ *to variable* $v_i \in V$ *is denoted by* (v_i, x)*. A constraint* $c \in C$ *is* satisfied *by an assignment* $\{(v_{c_1}, x_{c_1}), \ldots, (v_{c_k}, x_{c_k})\}$ *when* $scope(c) = (v_{c_1}, \ldots, v_{c_k})$ *and* $(x_{c_1}, \ldots, x_{c_k}) \in rel(c)$*. A set of assignments* $\{(v_1, x_1), \ldots, (v_n, x_n)\}$ *involving every variable in the problem is a* solution *when it satisfies all the constraints in* C*.*

A constraint solver finds a solution to a CSP via a process of *constraint propagation* and search. Constraint propagation is an inferencing process that takes place when a variable is initialised or loses values. Propagation maintains a level of consistency, such as arc-consistency [6], across the variables, removing values from domains that cannot occur in any solution (i.e., removing unsupported values). We use the definitions of (Generalized) arc-consistency ((G)AC) due to [2]:

Definition 2. *Given a CSP* (V, D, C)*, a constraint* $c \in C$ *with* $scope(c) = (v_{c_1}, \ldots, v_{c_k})$*, and a variable* $v \in scope(c)$*, a value* $x \in dom(v)$ *is* consistent *with respect to* c *(alternatively,* supported *by* c*) iff there exists a satisfying assignment* $\alpha = \{(v_{c_1}, a_1), \ldots, (v_{c_k}, a_k)\}$ *for* c *such that* $(v, x) \in \alpha$ *and* $\forall i,\ a_i \in dom(v_{c_i})$*. The domain* $dom(v)$ *is (generalized) arc-consistent on* c *iff all values in* $dom(v)$ *are consistent with respect to* c*, and the CSP is (generalized) arc-consistent when all variable domains are (generalized) arc-consistent on all constraints in* C*.*

Arc-consistency can be established on a CSP using an algorithm such as AC3 [6]. For sake of exposition we assume all constraints in C are binary and directed, such that a binary relation is represented by the relation and its inverse, consequently we have a symmetric pair of constraints c with $scope(c) = (v_a, v_b)$ and $scope(c') = (v_b, v_a)$ with $rel(c') = rel^{-1}(c)$. For example, if we have the constraint $c_{xy} = x < y$ then we also have the constraint $c_{yx} = y > x$. At the heart of AC3 is the *revise* function, which takes a binary constraint c as its argument and delivers a Boolean result. The function removes from $dom(v_a)$ all values that have no support in $dom(v_b)$ w.r.t. the constraint c, and returns true if any removals take place. Initially all constraints are added to a set S. Constraints are iteratively removed from S and revised. If $revise(c_{km})$ returns true then S becomes $S \cup \{c_{ik} | c_{ik} \in C \wedge i \neq k \wedge i \neq m\}$. This step can be considered as the

propagation of a domain reduction on variable v_k to variables constrained by v_k. The iteration terminates when S is empty or a variable's domain becomes empty. When S is empty the arc-consistency algorithm has reached a fixed point (i.e., a further application of the arc-consistency process will have no effect on the domains of the variables) and the problem has been made arc-consistent. The AC3 algorithm is of $O(e \cdot d^3)$ time complexity, where e is the number of constraints and d the size of the largest domain, however other algorithms can achieve a time bound of $O(e \cdot d^2)$ [13, 1].

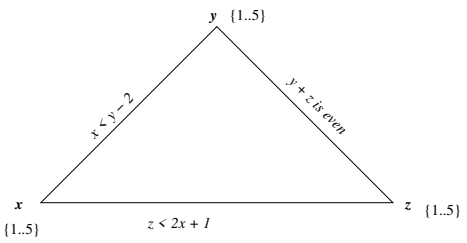

Fig. 1. A binary constraint satisfaction problem.

We demonstrate arc-consistency with the example of Figure 1, taken from [10]. We have three constrained integer variables x, y and z, each with an integer domain $\{1..5\}$, and binary constraints $c_{xy} : x < y - 2$, $c_{yz} : y + z$ *is even*, and $c_{zx} : z < 2x + 1$. Since constraints are binary we can represent the problem as a constraint graph, where nodes are vertices and edges are constraints. Initially the constraint c_{xy} is revised with respect to x and the values $\{3..5\}$ are removed from $dom(x)$. Then c_{xy} is revised w.r.t. y and $dom(y)$ becomes $\{4, 5\}$. c_{yz} is then revised w.r.t. y with no effect and then revised w.r.t. z, again with no effect. Revising c_{zx} w.r.t. z reduces $dom(z)$ such that it becomes $\{1..4\}$, consequently the constraint c_{yz} is added into the set of constraints pending revision. Constraints c_{zx} is revised w.r.t. x and then c_{yz} w.r.t. to y, both with no effect. The revision set at that point is empty and arc-consistency has been established with variable domains $dom(x) = \{1, 2\}$, $dom(y) = \{4, 5\}$, and $dom(z) = \{1..4\}$.

Solving a CSP may involve search, i.e., we might need to try different values for variables in order to determine if a solution exists. Typically a constraint program will begin by establishing arc-consistency, and then repeatedly select a variable and assign it a value from its domain (*instantiate* it). This effectively reduces that variable's domain to a singleton, and arc-consistency is then re-established. If this succeeds another instantiation is made, but if it fails we backtrack by undoing the most recent instantiation. This is called MAC, for maintaining arc-consistency [9]. Figure 2 shows a constraint program for the problem in Figure 1 using the choco constraint programming toolkit for the Java language [5], and it finds the first solution $x = 1$, $y = 4$, and $z = 2$.

```
import choco.Problem;
import choco.ContradictionException;
import choco.integer.*;

public class BMStut {

  public static void main(String[] args)  throws ContradictionException {
    Problem pb        = new Problem();
    IntDomainVar x    = pb.makeEnumIntVar("x",1,5); // x in {1..5}
    IntDomainVar y    = pb.makeEnumIntVar("y",1,5); // y in {1..5}
    IntDomainVar z    = pb.makeEnumIntVar("z",1,5); // y in {1..5}
    IntDomainVar even = pb.makeEnumIntVar("even",new int[] {2,4,6,8,10});

    pb.post(pb.gt(pb.minus(y,2),x));           // x < y - 2
    pb.post(pb.gt(pb.plus(pb.mult(2,x),1),z)); // z < 2x + 1
    pb.post(pb.eq(even,pb.plus(y,z)));         // y + z is even

    pb.solve();                                // solve using MAC
  }
}
```

Fig. 2. A JChoco constraint program for the CSP of Figure 1.

Constraint toolkits tend to be based around the AC5 algorithm [12], allowing propagators to be specialised for specific constraints resulting in improved efficiency and adaptability. AC5 amends set S from AC3 to contain triples of the form (v, c, δ) where $v \in \mathit{scope}(c)$ and δ is the set of values lost by v, consequently the remaining variables in the scope of c can be revised more efficiently. In an object-oriented toolkit language a constraint has associated propagation methods that should be implemented, and these methods are activated when a domain event occurs on a variable involved in that constraint. Domain events can be the initialisation of a variable, an increase in the lower bound, a decrease in the upper bound, the removal of a value between the bounds, or the instantiation of that variable. This is an exhaustive list, however some toolkits allow only one event, and that is that one or more values have been lost and the constraint programmer must then determine what action to take. To give examples of using a toolkit with specialised constraints, when modelling a routing problem we might have a constrained integer variable for each location to be visited, with a domain of values corresponding to the index of the next location to be visited (the so-called "single-successor" model). The subtour elimination constraint [4] might then be used to ensure that only legal tours are produced, and Régin's all-different constraint [7] could be added to increase domain filtering. Side constraints could be added to ensure that some locations are visited before others, as in a pick up and delivery problem. For a job shop scheduling problem we might have a model that uses 0/1 variables to decide the order between pairs of activities that share a resource, and we might increase propagation by adding the edge finding constraint [3].

The constraint programming approach is general and practical for modelling and solving problems, and provides a framework for the combination of problem specific algorithms in one solver. This allows us to solve many classes of problems efficiently and to model even more problems via the addition of side constraints.

References

1. C. Bessière and J-C Régin. Refining the basic constraint propagation algorithm. In *IJCAI*, pages 309–315, 2001.
2. Christian Bessière. Constraint propagation. In *Handbook of constraint programming*. Elsevier, 2006. Chapter 3.
3. J. Carlier and E. Pinson. Adjustment of heads and tails for the jobshop scheduling problem. *European Journal of Operational Research*, 78:146–161, 1994.
4. Yves Caseau and Francois Laburthe. Solving small TSPs with constraints. In *Proceedings International Conference on Logic Programming*, pages 1–15, 1997.
5. Choco. http://www.choco-constraints.net/ home of the choco constraint programming system, 2008.
6. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
7. J-C. Régin. A filtering algorithm for constraints of difference in CSP's. In *Proceedings AAAI'94*, pages 362–367, 1994.
8. Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier, 2007.
9. D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of ECAI-94*, pages 125–129, 1994.
10. Barbara M. Smith. A Tutorial on Constraint Programming. Technical Report 95.14, University of Leeds, 1995.
11. Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
12. P. van Hentenryck, Y. Deville, and C-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.
13. Zhang Yuanlin and Roland H. C. Yap. Makingg AC-3 an optimal algorithm. In *IJCAI*, pages 316–321, 2001.