

# Constraints

Fahiem Bacchus  
Department. of Computer Science  
6 Kings College Road  
University Of Toronto  
Toronto, Ontario  
Canada, M5S 1A4  
fbacchus@cs.toronto.edu

January 15, 2004

# Chapter 1

## Introduction

A constraint satisfaction problem (CSP),  $\mathcal{P}$ , consists of

1. A set of variables  $\{V_1, \dots, V_n\}$ .
2. For each variable  $V_i$  a domain of possible values  $Dom[V_i]$ .
3. A set of constraints  $\{C_1, \dots, C_m\}$ .

Each variable  $V$  can be assigned a value  $v$ , denoted by  $V \leftarrow v$  if and only if  $v \in Dom[V]$ .

Intuitively, we want to find an assignment of values to the variables subject to the condition that the values *satisfy* all of the constraints. Each constraint is over some subset of the variables, and imposes a constraint on the values they can be simultaneously assigned. Even if it is easy to satisfy each individual constraint, it may be difficult to find an assignment that satisfies all of the constraints simultaneously.

More formally we make the following definitions.

A *feasible* set of assignments is a set of assignments  $\{V_1 \leftarrow v_1, \dots, V_k \leftarrow v_k\}$  such that each of the  $V_i$  is unique. That is, no variable can be assigned more than one value.

We will only be interested in feasible sets of assignments, so from here on we will assume assignments are feasible. Let  $\mathcal{A}$  be a set of assignments, associated with  $\mathcal{A}$  is a set  $VarsOf[\mathcal{A}]$ : the variables that have been assigned values in  $\mathcal{A}$ . Since a (feasible) set of assignments cannot assign distinct values to the same variable, it can have at most  $n$  elements. When it contains  $n$  assignments it is called a *complete* set of assignments.

Each constraint  $C_i$  is over some set of variables  $VarsOf[C_i]$ , and the arity of  $C_i$  is the cardinality of this set,  $\|VarsOf[C_i]\|$ . Every constraint  $C_i$  is a set of sets of assignments: if the arity of  $C_i$  is  $k$  then each element of  $C_i$  is a set of  $k$  assignments, one for each of the variables in  $VarsOf[C_i]$ . Thus our notation implies that for all  $A_1$  and  $A_2$  in  $C_i$  we must have  $VarsOf[A_1] = VarsOf[A_2] = VarsOf[C_i]$ .

We say that a set of assignments  $\mathcal{A}$  *satisfies* a constraint  $C_i$  if  $VarsOf[C_i] \subseteq VarsOf[\mathcal{A}]$  and there exists an element of  $C_i$  that is a subset of  $\mathcal{A}$ . Furthermore, we say that  $\mathcal{A}$  is *consistent* (with respect to a particular CSP), if it satisfies all constraints  $C$  such that  $VarsOf[C] \subseteq VarsOf[\mathcal{A}]$ . We denote this by  $consistent(\mathcal{A})$ .

A *solution* (a satisfying solution) to  $\mathcal{P}$  is a complete and consistent set of assignments  $\mathcal{A}$ .

**Observation** Any subset of a solution is consistent.

In general we are interested in finding solutions to a CSP. More specifically, we might be interested in

1. Finding any solution (the satisfaction problem).
2. Enumerating or counting all solutions.
3. Finding a solution that optimizes some objective function.

## 1.1 The Constraint Domain

Within this general framework there are many specialized types of constraints, or constraint domains.

A constraint domain imposes certain restrictions on

1. The values the variables can take on (i.e., the  $Dom[V_i]$  sets).
2. The type and representation of the constraints allowed.

### 1.1.1 The Constraint domain $\mathbb{R}$

Here the variable domain is the set of real numbers, and the constraints are expressions composed from real numbers, the functions  $+$ ,  $\times$ ,  $-$ , and  $/$ , the predicates  $=$ ,  $<$ ,  $>$ ,  $\neq$ ,  $\leq$ ,  $\geq$ , and the logical connective  $\wedge$  (conjunction).

**Example 1** Say you have a loan for 2 years. We have the variables  $P$ , the principle of the loan,  $B1$  the final balance after the first year,  $B2$  the final balance after the second year,  $I$  the rate of interest,  $R1$  the repayment after the first year, and  $R2$  the repayment after the second year.

There are two constraints:

1.  $B1 = P + P \times I - R1$
2.  $B2 = B1 + B1 \times I - R2$

There are an infinite number of solutions to this CSP, one of them is

$$\{P \leftarrow 100, I \leftarrow 0.1, R1 \leftarrow 50, B1 \leftarrow 60, B2 \leftarrow 66, R2 \leftarrow 0\}.$$

A non solution is

$$\{P \leftarrow 100, I \leftarrow 0.1, R1 \leftarrow 0, B1 \leftarrow 60, B2 \leftarrow 66, R2 \leftarrow 0\}.$$

If we add the additional constraints:

- $B2 = 0, I = 0.1, R1 + R2 = 200.$

Then there are a whole set of solutions each with a different value of  $P$  and distribution between  $R1$  and  $R2$ .

It is not difficult to see that if we allow a more general set of functions in our constraints the constraint domain  $\mathcal{R}$  encompasses much of what has been studied in mathematics. Even without additional functions, this domain allows us to specify problems like finding the roots of polynomials:

**Example 2** Let  $X$  and  $Y$  be the two variables, and the constraints be

- $Y = 100 \times X \times X \times X + 4 \times X + 6$
- $Y = 0$

There are at most three solutions—the three roots of this polynomial. Of course, not all of these roots need be solutions, as only the real roots lie in the domain of  $X$ .

### 1.1.2 The linear constraint domain

Here the domains of the variables is once again the reals. However, now we restrict the constraints to be linear equalities/in-equalities over the variables. For example the constraint could be the system of equations:

$$\begin{aligned} 1 + X &= 2Y + Z \\ Z - X &= 3 \\ X + y &= 5 + Z \end{aligned}$$

One nice feature of this constraint domain is that it has been well studied, and we can easily find a characterization of the complete set of solutions by simply solving this set of equations. Furthermore, linear programming algorithms can even find points in this set of solutions that optimize linear functions defined over the variables.

### 1.1.3 Linear constraints over Integer Domains

Here we have the same kind of linear equality/in-equality constraints as in the previous case, but we restrict some of the variables to take on integer values. This case has been well studied in operations research, it is the well known the integer linear programming problem.

### 1.1.4 The Finite Domain constraint domain

Here we restrict the domains of the variables to be finite sets. Once we do this we can include any kind of constraints. We can represent these constraints either

**Intensionally** The constraint is represented by a expression or program that evaluates to true or false dependent on whether or not the assignment satisfies the constraint.

**Extensionally** The constraint is represented explicitly as satisfying sets of assignments. (Note that since the variable domains are finite, the constraints must also be finite). This representation however can have size exponential in the number of variables the constrain is over.

Finite domain constraints have been the main subject of study in AI, and they will be our main focus in the course.

## 1.2 Example Constraint Problems

### 1.2.1 E-Commerce

Combinatorial Auctions are an important opportunity for e-commerce. Say Company A manufactures and sells a number of industrial products, products that are the essential inputs for a range of industrial processes.

Say that company B is engaged in a manufacturing process that requires as input a number of the products sold by company A. However, for company B these products are only useful together. If they do not get all of the inputs the remaining inputs are worthless to them.

Company A may sell to a number of other companies besides company B, and some of these other companies might also want to purchase some of the products that B wants, but they might not want all of them. How is company A to maximize its income?

One way is for company A to set up a combinatorial auction. The bidders make bids on bundles of goods, and the winners of these bids get their entire bundle. The question is how does company A decide who wins the auction?

This can be set up as a CSP as follows:

- Variables  $\{B_1, \dots, B_n\}$ . One variable per bid.
- Domains, every domain is the set  $\{0, 1\}$ .  $B_i \leftarrow 0$  means that  $B_i$  wins their bid and gets the product they requested.
- Constraints. We impose a constraint between every bid  $B_i, B_j$  whose bundle has a non-empty intersection (i.e., these two customers both want the same item). The constraint prohibits the joint assignment  $\langle B_i \leftarrow 1, B_j \leftarrow 1 \rangle$ .

Also we have an optimization criteria. Associated with each bid  $B_i$  is a price  $p_i$ , and the auctioneer wants to maximize

$$\sum_{i=1}^n p_i B_i,$$

(where by multiplying by  $B_i$  we mean that we are multiplying by the value of the variable).

## 1.2.2 The Golomb Ruler Problem

The Golomb ruler problem is the problem of finding the shortest ruler with marks at integer location such that every pair of distinct marks can measure a different distance.

Consider a set of  $N$  natural numbers (including 0) such that if  $i, j, k, l$ , are numbers in the set such that  $i < j, k < l$ , and  $(i, j) \neq (k, l)$ , then  $(j - i) \neq (l - k)$ . That is, the difference between any pair of numbers is distinct. These numbers are the marks on the ruler.

The Golomb ruler problem is to find, for various values of  $N$ , a set containing  $N$  marks satisfying the condition above such that we minimize the maximum number in the set. That is we want to find the shortest ruler.

The *decision* problem version is, given a fixed length  $L$ , and the number of marks  $N$  is there a layout of the marks within the range  $0 - L$  that forms a golomb ruler. This can be set up as a CSP as follows:

- Variables,  $\{M_1, \dots, M_N\}$ .
- $Dom[M_1] = 0, Dom[M_N] = \{1, \dots, L\}$ .
- Constraints, for every  $i, j, k, l$ , such that  $(i, j) \neq (k, l)$  and  $(j, i) \neq (k, l)$ , we have the constraint  $|j - i| \neq |k - l|$ .

Now we can solve the optimization version of the problem by successively solving the problems with increasing values of  $L$  until we find a problem that has a solution.

**Question 1** *What feature of the Golomb ruler problem allows us to solve the optimization problem by iterating over this sequence of decision problem?*

## 1.2.3 Satisfiability

The question as to whether or not a set of 3-SAT clauses are satisfiable is easily translated into a finite domain CSP. Say that we have the boolean variables  $\{X_1, \dots, X_n\}$  and the collection of 3-clauses  $\{C_1, \dots, C_m\}$  (an example 3-clause might be  $\neg X_1 \vee X_{10} \vee \neg X_{11}$ ), we can construct the following CSP:

- Variables,  $\{X_1, \dots, X_n\}$ —we use the same set of variables.
- Domains. Each variable has  $\{\text{TRUE}, \text{FALSE}\}$  as its domain.
- Constraints. Each 3-clause becomes a constraint over the three variables that appear in the clause. The constraint prohibits the unsatisfying assignment. For example for the 3-clause  $\neg X_1 \vee X_{10} \vee \neg X_{11}$  there would be a constraint between  $X_1, X_{10}$ , and  $X_{11}$  which prohibits the assignment  $\{X_1 \leftarrow \text{TRUE}, X_{10} \leftarrow \text{FALSE}, X_{11} \leftarrow \text{TRUE}\}$ .

## 1.3 Local Consistency/Constraint Propagation

For most CSPs, especially finite domain CSPs finding an assignment that satisfies a single constraint is not difficult. The difficulty lies in satisfying all of the constraints simultaneously.

The example of satisfiability shows that the question of whether or not a solution for a finite domain CSP exists is an NP-complete problem, so in general finding a simultaneous assignment to all of the constraints can be computationally difficult.

One useful way of reasoning with constraints is to apply local propagation to achieve local consistency. This process does not usually solve the CSP, but it can provide some useful information about it. Additionally, for some specialized problems achieving local consistency can solve the problem.

There are a number of simple examples of propagation:

- Bounds propagation when dealing with real or integer valued variables. For example, if we have that  $0 \leq X \leq 3.3$ ,  $0 \leq Y \leq 2.0$ , and the constraint  $X \leq Y$ , then we can propagate this constraint to restrict the domain of  $X$  to the range  $0 \leq 2.0$ . Many more elaborate “interval” calculations can be identified, e.g., when  $X < Y \times Z$  and we have bounds on  $Y$  and  $Z$ .
- Often constraint problems contain functionally determined values. For example, in modeling a combinatorial circuit as a constraint problem, we might have that the variable  $O$  is the output of the AND of  $I_1$  and  $I_2$ . Thus if we know the value of  $I_1$  and  $I_2$  we can propagate these values through the constraint to determine the value of  $O$ .

For finite domain constraints we can treat propagation more systematically.

### 1.3.1 Local Consistency in Finite Domain CSPs

First we restrict ourselves to *binary* CSPs. Binary CSPs are CSPs where every constraint is over at most two variables. For binary constraints we can construct a *constraint network* from a constraint problem.

The constraint network is a graph formed by making each of the variables a node and making each binary constraint an edge between the two variables (nodes) it constrains.

On the constraint network 3 different types of local consistency can be defined.

- Node consistency.
- Arc consistency.
- Path consistency.

First some notation. If  $C$  is a constraint between variables  $V_1$  and  $V_2$  we write it as  $C_{V_1, V_2}$ , that is we use the variables it is over as a subscript. Furthermore if the assignment  $\{V_1 \leftarrow x, V_2 \leftarrow y\}$  satisfies  $C_{V_1, V_2}$  (i.e.,  $\{V_1 \leftarrow x, V_2 \leftarrow y\} \in C_{V_1, V_2}$ ) we write  $C_{V_1, V_2}(x, y)$ . Note the order of the values agrees with the order of the variables in the subscript. Note also that when considered to be a set of set of assignments  $C_{R, Q} = C_{R, Q}$ , however due to our dependence on order in our notation we have  $C_{R, Q}(x, y) \equiv C_{Q, R}(y, x)$ .

## Node Consistency

Node consistency is concerned with unary constraints only. It requires for every variable  $V$ , and every value  $x \in Dom[V]$ , that  $x$  satisfy all unary constraints on  $V$ .

For example, if  $Dom[V] = \{0, 1, 2, 3, 4\}$  and we have the unary constraint  $V \leq 3$ , then the CSP would not be node consistent.

We can make a CSP node consistent by simply removing all inconsistent domain elements. In our example, we would simply remove the element 4 from the domain of  $V$  to obtain  $Dom[V] = \{0, 1, 2, 3\}$ .

## Arc Consistency

Arc consistency is concerned with the local effects of each binary constraint. Say that  $C_{V_1, V_2}$  is a constraint (between variables  $V_1$  and  $V_2$  as per our notation). Then for every value  $x \in Dom[V_1]$  there needs to be a value  $y \in Dom[V_2]$  such that  $C_{V_1, V_2}(x, y)$  (i.e, we must have that  $\{V_1 \leftarrow x, V_2 \leftarrow y\} \in C_{V_1, V_2}$ . If there is no such “supporting” value for  $x$  then it is clear that the assignment  $V_1 \leftarrow x$  can never appear in any solution of the CSP. Thus we may remove it. This can be accomplished with the procedure *Revise*. Its inputs are two variables,  $V_1$  and  $V_2$  and the constraint between them  $C_{V_1, V_2}$

```
Revise(V1, V2, C)
  DELETE := false;
  for x ∈ Dom[V1]
    SUPPORTED := false
    for y ∈ Dom[V2] while !SUPPORTED
      if C(x, y)
        SUPPORTED := true
    if !SUPPORTED
      Dom[V1] := Dom[V1] - x
      DELETE := true
  return DELETE
```

We say that  $x \in Dom[V]$  supports  $y \in Dom[V']$  if  $C_{V, V'}(x, y)$ . Note that  $x$  supports  $y$  if and only if  $y$  supports  $x$ .

If we run this procedure on the arc between  $V_1$  and  $V_2$  we will achieve consistency in one direction only. That is every value in  $Dom[V_1]$  will have a support in  $Dom[V_2]$  but there may still be values  $y \in Dom[V_2]$  for which there is no  $x \in Dom[V_1]$  such that  $C_{V_1, V_2}(x, y)$ . Furthermore, by deleting values from  $Dom[V_1]$  we may now cause values in the domains of other variables to become unsupported.

To achieve arc consistency we must ensure that we achieve a stable state where every value has a support in every constraint. This can be achieved by an iterative process that converges to stability.

Consider what happens when we remove  $x \in Dom[V_1]$ . All variables that are constrained with  $V_1$  may now contain values that are unsupported. However, say that we have just run  $Revise(V_1, V_2)$ , it is also clear that the removal of  $x$  from  $Dom[V_1]$  will not affect any value in  $Dom[V_2]$ — $x$  was removed because it had no supporting value in  $Dom[V_2]$ , hence no value in  $Dom[V_2]$  depends on  $x$  for support. These observations allow us to restrict the amount of work we need to do to achieve arc-consistency, and they yield the algorithm AC-3:

```

AC-3()
  EnforceNodeConsistency()
  Queue := {(Vi,Vj) | there is an arc (vi,vj) (i ≠ j)}
  while Queue is not empty
    (V1,V2) := remove an arc from Queue
    if Revise(V1,V2)
      Queue := Queue ∪ (vi,V1) | there is an arc (vi,V1),
                                     vi ≠ V1,
                                     vi ≠ V2

```

Note that in the algorithm Queue is viewed as being a set, an arc is only added to it if that arc was not already present.

**Example 3** Consider the CSP:

- Variables:  $\{V_1, \dots, V_5\}$ .
- Domains:  $Dom[V_1] = Dom[V_2] = \{1, 2, 3\}$  and  $Dom[V_3] = Dom[V_4] = Dom[V_5] = \{1, 2\}$ .
- Constraints:  $V_3 > V_1, V_3 > V_2, V_3 > V_4, V_3 > V_5,$  and  $V_5 > V_4$ .

**Question 2** Draw the constraint network and achieve arc consistency in it.

**Observation** If arc-consistency reduces the domain of some variable to the empty set, the CSP has no solution. If arc-consistency reduces the domain of each variable to a single value, these values are a solution. Otherwise the CSP might still have no, one, or many solutions.

**AC-4** A number of improved algorithms for achieving arc-consistency have been developed. They revolve around the idea of using improved data structures that allow one to avoid having to scan and rescan all of the variable domains. AC-4 is a worst case optimal algorithm (although in practice it often does not perform as well as AC-3).

We use two data structures. First, **Supports** $[V,x]$ , this is a set associated with each value of each variable. The set contains triples,  $(V', y, C_{V,V'})$ , specifying the values supported by  $x$  of  $V$ . For each value  $y$  supported by  $x$  the triple notes the value, the variable whose domain contains the value, and the constraint between the variables. Second, **SupportCount** $[V,x,C_{V,V'}]$ , a support count for each value of each variable. We maintain a separate support count for each constraint that  $V$  participates in. If any of these support counts drops to zero we know that  $x$  now has lost all of its support on the domain of  $V'$ , and that we must now delete  $x$ .

```

AC-4()
  EnforceNodeConsistency()
  Set all Supports[V,x] = ∅
  Set all SupportCount[V,x,C] = 0
  ;;
  ;;Set up supporting values.
  ;;
  for all arcs (Vi,Vj) (in both directions)
    for x ∈ Dom[Vi]
      for y ∈ Dom[Vj]
        if Cij(x,y)
          SupportCount[Vi,x,Cij]++

```

```

        Supports[Vj,y] := Supports[Vj,y] ∪ (Vi,x,Cij)
    if SupportCount[Vi,x,Cij] == 0
        FailList := FailList ∪ (Vi,x)
        Dom[Vi] := Dom[Vi] - x
;;
;;Remove unsupported values
;;
while FailList is not empty
    (V,x) := remove an unsupported value from FailList
    for (y,V',C) ∈ Supports[V,x]
        SupportCount[y,V',C]--
        if SupportCount[y,V',C] == 0
            FailList := FailList ∪ (V',y)
            Dom[V'] := Dom[V'] - y

```

As with AC-3, FailList is being treated as a set (i.e., no duplicates are allowed).

It can be noted by keeping a count we know when we have to trigger more extensive processing. AC-4 has been the basis for the algorithms AC-6 and AC-7 which embody certain optimizations over and above AC-4.

However, it should also be noted that the space requirements for the **Supports** sets is  $O(ned)$  where  $n$  is the number of variables,  $d$  is the maximum domain size, and  $e$  are the number of edges in the constraint graph (number of constraints). For some CSPs this can be excessive.

**AC-2000 and AC-2001** Recently simple modifications that make AC-3 asymptotically optimal (i.e., of the same order of complexity as AC-4) have been identified (Refining the Basic Constraint Propagation Algorithm, by Christian Bessiere and Jean-Charles Regin. IJCAI-2001 pages 309–315).

### 1.3.2 Generalized Arc Consistency GAC

The previous definition of arc consistency was specific to binary constraints. However, arc consistency is easily generalize to the non-binary case.

**Definition 1.3.1** [Generalized Arc Consistent (GAC)]

A value  $x$  of a variable  $V$  is generalized arc consistent in a constraint  $C$  if either  $V \notin \text{VarsOf}[C]$ , or  $V \in \text{VarsOf}[C]$  and there exists an assignment for all of the other variables in  $C$  such that  $V \leftarrow x$  along with these assignments is consistent.

A value  $x$  of a variable  $V$  is generalized arc consistent if it is generalized arc consistent along every constraint.

A CSP is generalized arc consistent if all of its values are generalized arc consistent.

It is not difficult to see that if the constraint  $C$  is binary, then this definition is identical to the previous definition of binary arc consistency. GAC can be enforced by pruning all values that are not GAC using an algorithm very similar to AC3. However, in this case the time complexity of the algorithm grows exponentially with the arity of the constraint. An algorithms like AC4 can also be developed, but then space complexity (specifically the size of the support sets) grows exponentially with the arity of the constraint.

### 1.3.3 Path Consistency

Once again in the case of binary constraints we can define a higher form of arc consistency, called path consistency.

**Observation** If there is no constraint between  $V_1$  and  $V_2$  we can view the constraint as being the *universal* constraint. That is, we can view the “constraint” as permitting all possible values.

The next level of local consistency that has been studied in the literature, but that has not seen much practical application is path consistency. It is the condition that for every pair of values  $x \in \text{Dom}[V_1]$  and  $y \in \text{Dom}[V_2]$  such that  $C_{V_1, V_2}(x, y)$ , and third variable  $V_3$  there must exist a value  $z \in \text{Dom}[V_3]$  such that  $C_{V_1, V_3}(x, z)$  and  $C_{V_2, V_3}(y, z)$ . Note that if there is no constraint between  $V_3$  and either of  $V_1$  or  $V_2$  the condition is either vacuous or it reduces to simple arc-consistency.

However, to achieve path-consistency we must remove pairs of values  $(x, y)$  that do not have a path-support  $z$ . This cannot be done by removing values from the domains of variables, rather we must augment the binary constraints so that they rule these inconsistent pairs.

In practice path consistency has limited use since

- It is computationally cumbersome to augment constraints, whereas removing domain values can be done efficiently.
- The complexity of achieving path-consistency is cubic, rather than quadratic. This often makes it more expensive than is worthwhile.
- Augmenting constraints has less computational value than pruning domain values.

Note that this condition over triples of variables implies the following condition (which is where the name path consistency comes from).

A path of length  $m$  through the nodes  $(V_0, V_1, \dots, V_m)$  is path consistent if and only if for any values  $x \in \text{Dom}[V_0]$  and  $y \in \text{Dom}[V_m]$ , such that  $C_{V_0, V_m}(x, y)$  there exists a sequence of values  $z_1 \in \text{Dom}[V_1], \dots, z_{m-1} \in \text{Dom}[V_{m-1}]$  such that  $C_{V_0, V_1}(x, z_1), C_{V_1, V_2}(z_1, z_2), \dots, C_{V_{m-1}, V_m}(z_{m-1}, y)$ .

**Question 3** Prove that if the above condition on triples of variables holds then this notion of consistency along any path also holds. (Note that this only makes sense when dealing with binary constraints).

### 1.3.4 $(i, j)$ consistency

A generalization of node, arc, and path consistency is the notion of  $(i, j)$  consistency. It is defined quite simply:

A CSP is  $(i, j)$  consistent if whenever we choose consistent values for any set of  $i$  variables, we can find values for any set of  $j$  additional values such that the values for all of the  $i + j$  variables are consistent.

Remember that “consistent” means that every constraint all of whose variables are fully instantiated is satisfied. Note that this definition is not limited to binary constraints as were the previous notions of arc and path consistency.

- Binary Arc-Consistency is (1,1) consistency.
- Path-consistency is (2,1) consistency.
- $(i,j)$  consistency for  $i > 1$  requires us to augment the constraints in the CSP to achieve it.
- $(1,i)$  which is called  $i$  inverse consistency can be achieved by pruning domain values only. This case will prove to be useful when we talk about backtracking search.

We will revisit  $i$ - $j$  consistency when we discuss graphical properties of the constraint network.