Chapter 2

Constraint Satisfaction

When all you have is a hammer,
everything looks like a nail
Folklore

The Constraint Satisfaction is a general framework for modeling and solving combinatorial problems. Its flexibility and simplicity has led to the apparition of several Constraint Programming Languages (e.g. CLIP, Prolog III,...). Such programming languages are easy to use for relatively unexperienced users, encouraging other research communities (e.g. optimization) to look for techniques to input their problems (e.g. linear programs) in similar frameworks.

In this thesis, we exploit the generality of Constraint Satisfaction by building distributed algorithms which assume that the problems of the agents can be modeled in this framework or its simple extensions mentioned later.

We will approach Constraint Satisfaction by developing and using general solving techniques, rather than identifying tractable classes of problems for applying to them specialized algorithms (Broxvall & Jonsson 2000; Cohen *et al.* 1997). This is, otherwise, a principled approach. However, in our case agents are expected to enforce private constraints, such that in general, special tractable cases cannot be detected and solved accordingly. Nevertheless, the interest of agents in variables is public and therefore, tractable classes based on the topology of the relations between agents can be useful. The tractability of non-cyclic topologies of agents has been first exploited with distributed arc consistency (Calisti 2001). By the time this thesis is being submitted, another successful work on the tractability of non-cyclic agent links, this time based on Distributed Breakout, has also been published in (Zhang 2002).

In this thesis we only discuss general solving algorithms. We review now the most important techniques. Some of the presented approaches have the merit to throw much light on the search theory (e.g. Dynamic Backtracking), while others are famous for their efficiency (e.g. Binary Splitting).

The contribution presented in this chapter is limited to an improved algorithm for achieving Bound Consistency (BC1999 Section 2.4.1). BC1999 is inspired from the 6th arc-consistency operator (AC6) and its ideas are similar with the ones applied later to obtain AC2001 from AC2000, respectively AC3.1 from AC3 (Silaghi *et al.* 1999b; Bessière & Régin 2001; Zhang & Yap 2001).

Before or while reading this chapter, you may prefer to read Annex A

2.1 A really general definition

In annexes we introduce the Constraint Satisfaction Problems (CSPs) as special cases of Optimization Problems. However, most approaches to CSPs appear in a context that is very different from optimization. This has lead to the development of very specific appellations for the most used notions. A CSP is typically defined by a set of variables, $X = \{x_1, x_2, ..., x_n\}$. Each variable, x_i, can take its values from a domain D_i. The variables are subject to constraints which specify

consistent value combinations. Each constraint, c_i, acts on a subset X_i of X. For problems with finite domains, c_i is often represented by a subset of the Cartesian-product of the domains of the variables in X_i. However, in most contemporary approaches, the constraints are represented by symbolic predicates. For space or user comfort, this cannot be avoided for problems with large domains and for constraints over many variables.

Initially, techniques for solving CSPs were mostly designed for problems with finite small domains where no single constraint acts on more than two variables. The constraints that act on exactly two variables are called *binary constraints*. These are sufficient for simple applications. However, in this thesis we only deal with general CSPs with n-ary constraints that are represented as predicates. Some comments and proposals for explicit representations are made in Section D.1. Algorithms that can benefit from explicit representations are discussed in Chapters 5 and 13.

While in Annex A we give intuitive images about the main classes of approaches to CSPs, here we give technical details of the techniques that we use. We describe techniques and concepts from incomplete search and consistency maintaining, for problems with both finite and continuous domains. New contributions of this thesis are contained in Sections A.3.1, and 2.6.

2.2 Incomplete Search

In general, Constraint Satisfaction can model very hard problems. The complexity theory is nicely presented in (Garey & Johnson 1979). There, one can find detailed explanations of the notions of P, NP, NP-complete, NP-hard and P-space problems. Intuitively:

- P is the class of problems that can be solved within a time bounded by a polynomial function on the length of the problem description.
- NP is the class of problems for which a claimed solution can be tested within a time bounded by a polynomial function on the length of the problem description.
- NP-complete is the subclass of NP problems to which a SAT problem can be mapped within a time bounded by a polynomial function on the length of the SAT problem description. It is required that the length of the obtained problem description is bounded by a polynomial function on the length of the SAT problem description.
- NP-hard is the class of problems (not necessarily NP) to which SAT can be reduced as previously described.
- P-space is the class of problems that can be solved using a space that is bounded by a polynomial function on the length of the problem description.

Proposition 2.1 *In general, proving feasibility for a Constraint Satisfaction Problem is NP-complete.*

Proof. It results by direct reduction from SAT. Any SAT problem is a CSP, therefore CSP is NP-hard. There exist several encoding schemas for transforming general CSPs to SAT. The most simple solution is to introduce a boolean variable for each possible assignment of each variable. More fancy (but perhaps less efficient) approaches map each variable x in $log_2(||D_x||)$ boolean variables, where D_x is the domain of x (Hoos 1999). □

Constraint Satisfaction Problems may be used to approach very large problems with real-time requirements. For large problems, complete systematic techniques can be very slow even when the problem has many solutions. The user is then forced to admit that he may have to stop the search before the satisfiability is decided. Especially in these conditions, the completeness promise does no longer offer much. Incomplete local search is often an alternative which may be successful in real time. Local techniques act by scanning *neighborhoods* of a current valuation, trying to improve it with minimum effort like the local optimization (see Annex A). Regular local search algorithms are defined by the next features:

- *The search space.* The set of possible valuations of the variables in the CSP.
- *The neighborhoods.* Define which valuations in the search space can be considered to belong to the neighborhood of a given valuation.
- *Selection rule.* Deciding which valuation out of the neighborhood is selected next. Some history can be used with this rule.
- *The restart rule.* Deciding when to use an alternative selection rule for choosing a next valuation.

```
procedure Local-Search() do
    s ← random valuation of variables;
    for ever do
        N ← neighborhood(s);
        if ¬restartRule(s,N) then
            c ← selection(s,N);
        else
            c ← alternative-selection(s,N);
        s ← c;
        if(solution(s)) return s;
        if(abandon()) return failure;
```

Algorithm 1: A Generic Local Search Algorithm.

Depending on peculiarities of these features, several local search algorithms have been classified into further families such as: GSAT (Selman *et al.* 1992), Min-Conflict (Minton *et al.* 1990), Hill-Climbing, Random-Walk, and Tabu-Search (Glover 1986) (Bartak 1988). A general Local Search algorithm is given in Algorithm 1.

```
procedure Breakout() do
    Until current state is solution do
        if current state is not a local minimum then
            make any local change that reduces the total cost;
        else
            increase weights of all currently stored conflicts;
```

Algorithm 2: The Breakout Algorithm.

Clause weight is obtained when the function selection(s,N) stores information about the constraints that were hard to satisfy in local minima and avoids conflicting them in the future. (Selman & Kautz 1993) fixes with *clause weight* versions of GSAT which perform very badly (Jónsson & Ginsberg 1993) and whose peculiarity consist in returning often to the same local minima. A version of *clause weight*, the Breakout algorithm, is given in Algorithm 2 (Morris 1993) and a distributed counterpart is described in (Hirayama & Toyoda 1995; Yokoo & Hiramaya 1996). An extrema is Tabu Search (Glover 1986; Galinier & Hao 1997) where moves/configurations to avoid are stored for a while.

The alternative-selection(s,N) function is often random but some versions of Local Search use ideas from genetic algorithms, namely combining some of the best solutions found so far. The restartRule(s,N) function may also use some random computations. When the probability that restartRule(s,N) returns *true* decreases slowly in time, the obtained algorithm is called *Simulated Annealing* (Kirkpatrick *et al.* 1983). Most often Local Search simply detects local minima. abandon() decides when to renounce search.

Local Search can be used to approach optimization problems where the objective function minimizes the number of unsatisfied predicates in a set of predicates. When local search techniques fail for real-time problems, they may still be able to approach a valuation which conflicts a reduced number of constraints.

```
procedure Fill() do
    Until current state is solution do
        if current state is not a local minimum then
            make any local change that reduces the cost;
        else
            increase stored cost of current state;
```

Algorithm 3: The Fill Algorithm.

The Breakout algorithm has a complete version (see Algorithm 3). Unfortunately the complete version called *The Fill Algorithm* has exponential space worst case requirements (Morris 1993). The Fill Algorithm requests to store all local minima and to associate a weight to each of them. One can infer that distributed versions of Distributed Breakout could be made complete if one stores all detected conflicts.

2.3 Complete Search

Most techniques that are asynchronous and complete, are based on the framework introduced now. In the previous section we got acquainted with incomplete techniques based on Local Search. These techniques are quick in practice for the problems that they can solve. ((Selman & Kautz 1993) mentions one order of magnitude gain in problem size with GSAT versus Davis-Putnam Algorithm). However, Local Search algorithms cannot prove unsatisfiability and perform less well on large problems with few solutions. As pointed in Section 2.2, some Local Search Algorithms based on conflict storing can offer completeness, but they usually require exponential space.

The most simple but typically inefficient complete algorithm is the *Generate and Test*. *Generate and Test* performs by enumerating all the elements in the Cartesian product of the domains of the problems. Each of these elements is tested separately until a solution is found or the search space is exhausted. Most used algorithms try to reuse the results of the tests in order to prune several elements at once.

The complete algorithms with polynomial space requirements are characterized by a systematic analysis of the search space. Such algorithms are often called *systematic search techniques*. Since they often split the search space in a divide and conquer manner and backtrack upon failure, these techniques are also referred to as *backtracking algorithms*. Overviews of the classic backtracking algorithms are given in (Kondrak 1994a; Tsang 1993). The classic formalism used for presenting and proving correctness is based on the search tree view of the trace as in Section A.2.2.1. (Stallman & Sussman 1977) proposes the dependency directed backtracking (DDB), a backtracking based algorithm storing conflicts that lead to dead-ends. These conflicts are stored under the form of nogoods.

Definition 2.1 (Stallman&Sussman,1977) *A NOGOOD assertion explicitly lists the state assumptions that conspired to produce the contradiction.*

While this definition is general and covers all the aspects in this thesis, most algorithms developed for problems with finite domains use the next definition given in (Ginsberg & McAllester 1994).

Definition 2.2 (Ginsberg&McAllester,1994) *A nogood is an expression of the form:*

$$(x_1 = v_1) \wedge ... \wedge (x_k = v_k) \rightarrow x \neq v.$$

Here, $(x_k = v_k)$ are the antecedents and $x \neq v$ is the conclusion of the nogood.

The most recent search algorithms are built using the new formalism introduced in the seminal work of Ginsberg (Ginsberg 1993a). This formalism allows to simply build and prove new complex systematic algorithms by keeping track of the already eliminated space using nogoods. In Algorithm 4 we see the reformulation of the Chronological Backtracking given in Annex A, as presented in (Ginsberg 1993a).

I denotes the set of variables and V_x is the domain of values for the variable x. P is a set of assignments of type (x, v), where x is a variable and v is a value. $\overline{P}$ denotes the set of variables involved in assignments found in P, $\overline{P} = \{x | \exists v, (x, v) \in P\}$.
$\hat{\epsilon}$ is an *elimination mechanism* for the problem. It takes as parameters a set of assignments, P, and a variable x and returns a set of eliminated values with explanations of the eliminations, a subset of P.

1. Set $P = \emptyset$. P is a partial solution to the CSP. Set $E_x = \emptyset$ for each $x \in I$; E_x is the set of values that have been eliminated for the variable x.
2. If $\overline{P} = I$, so that P assigns a value to every element in I, is a solution to the original problem. Return it. Otherwise, select a variable $x \in I - \overline{P}$. Set $E_x = \hat{\epsilon}(P, x)$, the values that have been eliminated as possible choices for x.
3. Set $S = V_x - E_x$, the set of remaining possibilities for x. If S is nonempty, choose an element $v \in S$. Append (x, v) to P, thereby setting x's value to v, and return to step 2.
4. If S is empty, let (y, v_y) be the last entry in P; if there is no such entry, return failure. Remove (y, v_y) from P, add v_y to E_y, set $x = y$ and return to step 3.

Algorithm 4: Chronological Backtracking based on nogoods.

Once the Chronological Backtracking takes its new form, it is easier to define heuristics inspired from Local Search (Ginsberg 1993a; McAllester 1993; Ginsberg & McAllester 1994; Bliek 1998a). In Algorithm 5 (Bliek 1998a) is given a generalization of the previous polynomial space algorithms developed based on nogoods. Note that GPB works with a complete set of assignments, X. GPB also maintains a partial order on variables, S, instead of the classic total order (defining a relation $<_S$). The set of nogoods stored by GPB is denoted Γ. The *live domain* of a variable y is the set of values of y that are not consequence of any nogood in Γ.

procedure *GPB* **do**
  Until X *is a solution or the empty nogood is derived* **do**
4.1 select a nogood γ corresponding to a constraint violation;
4.2 Backtrack γ;
4.3 change X to be acceptable for Γ;

procedure *Backtrack* (γ) **do**
  Select a conclusion variable y of γ respecting S;
4.4 Add γ to Γ and remove from Γ all nogoods
        which have y as antecedent variable;
  Modify S so that the transition conditions are satisfied;
  if *the live domain of* y *is empty* **then**
4.5 infer a new nogood ρ and;
    Backtrack ρ;

Algorithm 5: General Partial Order Dynamic Backtracking.

Definition 2.3 ((Bliek 1998b)) *An assignment is acceptable for a set of nogoods if all the antecedents are matched and none of the conclusions are violated.*

We recall now the transition conditions required for getting a new order S' from the previous order S when a new nogood with conclusion $y \neq v$ is added in GPB. Denoting $Z = \{z | y <_S z\}$, and by Γ and Γ' the set of ordered nogoods before and after the transition:

1. only ordering conditions of the type $q <_S z$ with $z \in Z$ may disappear,
2. for every variable x for which $x <_{S'} y$, we have for all $z \in Z$, $x <_{S'} z$ and
3. for every ordered nogood in Γ' with antecedent variables x_j and conclusion variable y, we have $x_j <_{S'} y$.

Instances of General Partial Order Dynamic Backtracking actually approach GSAT in the sense that they allow incremental changes in the current valuation. However, the allowed changes are limited and can lead to worsening of the efficiency (Backer 1994). Special freedom of moving in the search space is given by the Limited Discrepancy Search. Limited Discrepancy Search is a search algorithm allowing for backtracking to early nodes in the search tree but ensuring that only a limited number of assignments are changed (Harvey & Ginsberg 1995).

The backtracking algorithms are called *constructive search*, since they perform by invariantly maintaining a consistent partial valuation. This is different from *generate and test* and *local search* algorithms. A distributed version is proposed in Section 10.4.

In Annex A, Section A.3, I describe two interesting versions of GPB: the most well known version, Dynamic Backtracking (DB), respectively GOB, the version used in the tests of (Silaghi *et al.* 2000a).

2.4 Local Consistency

Complete search is generally exponential in the size of the problem and expensive. When people deal with problems, they only seldom use extensive search of all possibilities. Most often, humans rely on inferences. Sometimes, local search is used for hard problems. We usually employ extensive search in our daily thinking only when inferences are not possible, and we want completeness.

Definition 2.4 *Local consistency is a technique for reducing the size of a problem by exploiting conclusions of logic inferences made on sub-parts of the problem (subsets of the constraints).*

2.4.1 Bound Consistency

Most well known (LP) local consistency is the arc consistency introduced by Waltz. It is also the first notion defined for constraint satisfaction. Consistency notions can be simpler or more complex, weaker or stronger. We discuss here a weaker and simpler notion called Bound Consistency. Bound Consistency is very important for problems with large domains and it has been especially studied for numeric problems. Bound Consistency can be defined only for problems with ordered domains (Silaghi *et al.* 1999a).

2.4.1.1 Definition of Bound-Consistency

For a constraint q and a set D of domains for the variables involved in q, we define the notion of support as follows (Mohr & Henderson 1986).

Definition 2.5 *Given a variable x involved in q, a value v in the domain of x is said to have supports in q iff $q \wedge (x{=}v)$ has at least a solution with values in D.*

Example 2.1 *Let $x, y \in [0, 2]$ and let a constraint, q, be defined as $x{+}1{-}y{=}0$. $x{=}1$ finds supports in q, namely $y{=}2$ is the support for $x{=}1$. Instead, $x{=}2$ has no support since $y{<}3$. Actually, only x from $[0, 1]$ have supports in q, and their supports are $y{=}x{+}1$.*

The next two definitions are taken from analysis course books.

Definition 2.6 *The infimum of a totally ordered set D is defined as:*

$$\inf(D) = max\{x|\forall y \in D, x \leq y\}$$

Definition 2.7 *The supremum of a totally ordered set D is defined as:*

$$\sup(D) = min\{x|\forall y \in D, x \geq y\}$$

Definition 2.8 (Bound Consistency) *A CSP with discreet totally ordered domains, P, is Bound Consistent iff for every variable x with domain D_x, both* $\sup(D_x)$ *and* $\inf(D_x)$ *find supports for each constraint in P, separately.*

Example 2.2 *The problem* $x, y \in \{1, 2, 3\}, x + y = 4$*, is Bound Consistent.* $\inf(\{1, 2, 3\})$*=1.* $\sup(\{1, 2, 3\})$*=3.*
x=1 *has as support* y=3.
x=3 *has as support* y=1.
y=3 *has as support* x=1.
y=1 *has as support* x=3.

2.4.1.2 Algorithm for achieving Bound Consistency

An algorithm that uses the Definition 2.8 to reduce discrete problems to bound consistent equivalents is exponential in the maximal constraint arity, a, (Puget 1998). By m we denote the number of constraints of a CSP, by d the maximal domain size, and by n the number of variables. The Algorithm 6 (Silaghi *et al.* 1999b; 2000c) enforces bound consistency. It requires storing for each boundary of each checked constraint the tuple in the constraint that supports the boundary. The supporting tuples are stored in the structures:

- *UpSupport[*x,q*]* for the supporting tuple in the constraint q for the upper bound of variable x,
- *LowSupport[*x,q*]* for the supporting tuple in the constraint q for the lower bound of variable x.

Since each tuple has size a, this requires a space of $O(a^2m)$, more exactly $2a^2m$.

```
function BC1999(Queue)
    while q=ExtractConstraint(Queue) do
        if !CheckBounds(Queue,q) then
            return 0
    return 1
```

Algorithm 6: BC1999

Initially, all the constraints are inserted in the queue. A constraint is queued when the domain of one of its variables changes. A revision step consists of verifying that the values supporting the boundaries of the current constraint are still valid. For any support falling outside the current intervals of the variables (lines 7.1 and 7.2), the constraint is scanned so that a new support can be identified. If the scanning is performed according to a static order of the domains and of the constraints' variables for a given constraint, then it is only needed to scan beyond the old support.

The *ScanForUpper* and *ScanForLower* are similar to the basic step of the procedure described in length in Section 5.2.1.1. Rough pseudo-codes of two alternative implementations are given in Algorithm 8, respectively in Algorithm 9 and Algorithm 10. While the complexity evaluated next for the two algorithms is the same, Algorithm 9 is more optimized and is closer to the version actually implemented in the work described here.[1]

[1]The version I implemented is based on bitmap representations and is slightly more complex and optimized, (e.g. doing additional tests at several abstraction levels).

```
function CheckBounds(Q,q)
    for all variables, x, in the constraint q do
        changed←0;
7.1     if OutOfBound(UpSupport[x,q]) then
            Box←ScanForUpper(x,q);
            changed←1;
7.2     if OutOfBound(LowSupport[x,q]) then
            Box←ScanForLower(x,q);
            changed←1;
        if Empty(Box,x) then
            return 0;
        if changed then
            Insert(Q,AffectedConstraints({x}));
7.3         UpdateVariableRange(Box,x);
    return 1;
```

Algorithm 7: Check Bounds

Theorem 2.2 *BC1999 terminates in less than mdn revision steps where d and n stand for the maximum domain size, respectively the number of variables.*

Proof. After each reduction of a domain there will be at most $m-1$ constraints checked without a subsequent domain reduction. If, after checking all the $m-1$ constraints involved, there is no domain reduction, consistency is reached and the algorithm stops. There are at most dn domain reductions possible. Therefore there will be maximum mdn steps before failure or consistency is reached. □

Theorem 2.3 *The complexity of the BC1999 algorithm is $O(ma^2d(n + d^{a-1}))$, where m stands for the number of constraints, a for the maximal arity, d for the maximal domain size and n for the number of variables.*

Proof. The tuples of the constraints are scanned only once each, $O(nd^a)$, for each pair of bounds[2], $O(a)$, this results in a number of $O(amd^a)$ tests. The cost of a test is $O(a)$, leading to a complexity of $O(a^2md^a)$ operations. The only values that have to be reconsidered as supports are those of the changed variables. However, at each revision step, all the supports of a constraint boundaries (eventually one less[3]) have to be tested (each support needs $2a$ comparisons), resulting in a complexity of $O(mdna^2)$. □

One difference between BC1999 and the AC2001/3.1 described in (Bessière & Régin 2001; Zhang & Yap 2001), is that the modification of a domain D_k requires the constraints to be tested also for the reduction of the already reduced bounds of D_k. Such a situation is not possible in AC2001/3.1, where the corresponding values in the reduced D_k have been already tested in previous passes.

BC1999 has been inspired from AC6 (Bessière 1994), from which it removes a storages of supports. To be noted that the same modification has been done in (Bessière & Régin 2001; Zhang & Yap 2001) for developing AC2001 respectively AC3.1. As it has been argued later for AC2001 and AC3.1, the advantage of BC1999 over the direct use of the definition as in AC3, is that it guarantees optimal complexity with little space overhead and with a simple implementation.

2.5 Numeric Constraints

A wide range of problems can be modeled as constraint satisfaction problems (CSPs) with numerical constraints. A numerical CSP (NCSP), (V, C, D) is stated as a set of variables V taking their values

[2]At most one value of each dimension may be scanned twice.

[3]If only one bound of only one variable is reduced, then the support of the other bound of the same variable don't have to be checked.

```
//A discrete interval (pair of bounds) has the type IN_+^2
function ScanForUpper-1(x:IN, q:Constraint)→ (IN_+^2)^arity(q)
    a ← arity(q);
    t ← PrevActive(UpSupport[x, q], x, q);
    for ever do
        if (feasible(t,q)) then
            UpSupport[x, q] ← t;
            return UpdateUpperBoundCrtSearchNode(x,t.x);
        t ← PrevActive(t, x, q);
        if (t=∅) then
            return ∅;
function ScanForLower-1(x:IN, q:Constraint)→ (IN_+^2)^arity(q)
    a ← arity(q);
    t ← NextActive(LowSupport[x, q], x, q);
    for ever do
        if (feasible(t,q)) then
            LowSupport[x, q] ← t;
            return UpdateUpperBoundCrtSearchNode(x,t.x);
        t ← NextActive(t, x, q);
        if (t=∅) then
            return ∅;
//x is the first variable function NextActive(t:IN_+^a, x:IN, q:Constraint)→ IN_+^a
    t.(a−1) + +;
    i ← the first index of a value out of bounds;
    if i<a ∧ t.i> high bound for i then
        if i=0 then
            return ∅;
        t.(i−1) + +;
        if t.(i−1)> high bound for i then
            return NextActive(t,x,q);
    set in t all variables after i^th to the lowest element in the current search space;
function PrevActive(t:IN_+^a, x:IN, q:Constraint)→ IN_+^a
    t.(a−1) − −;
    i ← the first index of a value out of bounds;
    if i < ∧ t.i< low bound for i then
        if i=0 then
            return ∅;
        t.(i−1) − −;
        if t.(i−1)< low bound for i then
            return PrevActive(t,x,q);
    set in t all variables after i^th to the highest element in their current search space;
```

Algorithm 8: Iterative versions of scans for supports: *ScanForUpper* and *ScanForLower*

in (subsets of) $\mathbb{R}$ and subject to constraints C. In practice, the constraints can be equalities or inequalities of arbitrary type and arity, usually expressed using arithmetic expressions. The goal is to assign values to the variables so that all the constraints are satisfied. Such an assignment is then called a solution. Interval constraint-based solvers (e.g. Numerica (Van Hentenryck 1998),

```
function ScanForUpper-2(x:ℕ, q:Constraint)→ (ℕ²₊)^arity(q)
    a ← arity(q);
    t ← UpSupport[x, q];
    t.(a−1) − −;
    init[0..(a-1)]←true;
    r=UpRecursiveCheck(x,q,t);
    if (r) then
        UpSupport[x, q] ← t;
        return UpdateUpperBoundCrtSearchNode(x,t.x);
    else
        return ∅;
function ScanForLower-2(x:ℕ, q:Constraint)→ (ℕ²₊)^arity(q)
    t ← LowSupport[x, q];
    t.(a−1) + +;
    init[0..(a-1)]←true;
    r=LowRecursiveCheck(x,q,t);
    if (r) then
        LowSupport[x, q] ← t;
        return UpdateLowerBoundCrtSearchNode(x,t.x);
    else
        return ∅;
```

Algorithm 9: Recursive versions of scans for supports: *ScanForUpper* and *ScanForLower*

Solver (ILOG 1999)) take as input a numerical CSP, where the domains of the variables are intervals over the reals, and generate a set of boxes which *conservatively* enclose each solution (no solution is lost).

2.6 Random Problem Generation

It is common practice in the CSP community to test algorithms on discrete problems randomly generated. There exist a standard accepted procedure only for generating binary random problems.

2.6.1 Random binary discrete CSP generators

Random binary problems are generated in such a way that there exists a single constraint for any given pair of distinct variables. Given a number of variables and a domain size, a binary discrete problem can be randomly generated using the following two parameters:

- density: The percentage of constraints that are generated among the total number of constraints that are possible, $\frac{2m}{n(n-1)}$. m is the number of generated constraints and n is the number of variables.
- tightness: The ratio of infeasible tuples of a given constraint given the total number of its tuples, $\frac{i}{d^2}$. i is the number of infeasible tuples and d is the domain size.

Some researchers generate all the constraints with the same tightness by dispersing a predefined number of feasible tuples in the constraint matrix. In the tests presented in this thesis, the constraints are generated by setting the elements of the constraint matrices to 0 rather than to 1, with a probability equal to the tightness.

For generating Valued CSPs (CSPs where the elements of the constraint matrices can be any real number) these matrix elements can be generated using random generators with predefined distributions of the values. In Annex Section A.4 I propose a technique for building such generators.

```
function UpRecursiveCheck(x:ℕ, q:Constraint, t : ℕ^a)
    if (¬init[x]) then
        t.x= upper bound for x;
    else
        init[x]=false;
        if t.x> upper bound for x then
            t.x= upper bound for x; set higher variables to their upper bound;
        if t.x< lower bound for x then
            set higher variables to their upper bound; return false;
    Until t.x< low bound for x do
        y = Next(x) //the variables are statically ordered, first is the shrinking one;
        if y=∅ then
            if feasible(t) then
                return true;
        else
            if UpRecursiveCheck(y,q,t) then
                return true;
        t.x − −;
    return false;
function LowRecursiveCheck(x:ℕ, q:Constraint, t : ℕ^a)
    if (¬init[x]) then
        t.x= low bound for x;
    else
        init[x]=false;
        if t.x< lower bound for x then
            t.x= lower bound for x; set higher variables to their lower bound;
        if t.x> higher bound for x then
            set higher variables to their lower bound; return false;
    Until t.x> upper bound for x do
        y = Next(x);
        if y=∅ then
            if feasible(t) then
                return true;
        else
            if LowRecursiveCheck(y,q,t) then
                return true;
        t.x + +;
    return false;
```

Algorithm 10: Recursive procedures called by: *ScanForUpper* and *ScanForLower*

2.7 Summary

In this chapter we have seen some basic concepts for Constraint Satisfaction, such as degrees of consistency, complete and incomplete search techniques, and random problem generation.

BC1999 is an algorithm for computing Bound Consistency for n-ary discrete CSPs with optimal complexity, and AC3-like efficiency and simplicity. Recently, AC2001 and AC3.1 have applied successfully the same technique for achieving arc-consistency.