1 CONSTRAINT SATISFACTION METHODS FOR GENERATING VALID CUTS

J. N. Hooker

Graduate School of Industrial Administration
Carnegie Mellon University
Pittsburgh, PA 15213 USA
http://www.gsia.cmu.edu/afs/andrew/gsia/jh38/jnh.html
john.hooker@andrew.cmu.edu

Abstract: Valid cuts are viewed in the operations research literature as inequalities that strengthen linear relaxations. The constraint satisfaction community has developed an alternate approach. Logical inference methods, such as the resolution method, can generate valid cuts that need not be inequalities and that are considered apart from any role in relaxations. They reduce backtracking by helping to achieve "consistency," which is analogous to integrality in a polyhedral setting. The basic theory underlying these methods is presented here. Parallels with mathematical programming are pointed out, and resolution-based algorithms for generating cuts are proposed as a unifying theme. Specific topics include k-consistency, adaptive consistency, the dependency graph, and various measures of its width, including induced width and bandwidth.

1.1 INTRODUCTION

Cutting planes have been a major research focus in the operations research community since the 1950's. They make possible the branch-and-cut solution of many integer and mixed integer programming problems that would otherwise be intractable. The rationale for their use almost always relates to the linear

relaxation of the problem: they strengthen the relaxation in order to obtain a better bound on the optimal value when the relaxation is solved.

Valid cuts serve another purpose, however, that has not been clearly distinguished in the mathematical programming literature. They can directly reduce the size of the search tree even when they are not used in a relaxation. In fact this perspective gives rise to an alternative theory of valid cuts that in some sense parallels cutting plane theory. Such a theory has been developed in the constraint programming literature. It provides several ways to reduce backtracking that are largely unknown in the mathematical programming community. This paper presents the elementary aspects of this theory and points out some of the parallels with mathematical programming. It is written for persons with a background in operations research.

1.1.1 Another Perspective on Valid Cuts

A simple example illustrates how valid cuts can prune a search tree even when they are not used in a relaxation. Suppose that a 0-1 programming problem contains the following constraints.

$$\begin{aligned} x_1 + x_{100} &\geq 1 \\ x_1 - x_{100} &\geq 0 \end{aligned} \tag{1.1}$$

There is obviously no feasible solution in which $x_1 = 0$. Suppose further that the problem is solved purely by branching on variables in the order $x_1, \ldots, x_{100}$; no relaxations are used. If the $x_1 = 0$ branch is taken first, it may be necessary to search the entire subtree of $2^{100} - 1$ nodes thereby generated in order to discover that it contains no feasible solutions. However, if the valid cut

$$x_1 \geq 1$$

is added to the constraint set, the branch $x_1 = 0$ is ruled out from the start.

A mathematical programmer is likely to view this as an example of preprocessing that fixes a variable. In fact several preprocessing techniques can be viewed as special cases of constraint satisfaction methods. As conceived in operations research, however, preprocessing tends to be a bag of tricks without a unifying theory. The aim here is to provide a theory that not only encompasses many preprocessing ideas but much more as well. The cut $x_1 \geq 1$, for example, can be viewed as making the constraint set (1.1) "strongly 2-consistent,"[1] which helps to reduce backtracking.

An additional advantage of studying cuts from this perspective is that constraints need not be in inequality form, because they need not occur in linear relaxations. The constraint satisfaction community has taken full advantage of

this greater generality. It formulates models with multi-valued discrete variables in addition to boolean and integer variables. It uses logical propositions, all-different constraints, and other non-inequality forms than can give much more succinct formulations of a problem. It may be far from evident how to formulate relaxations in such cases, but the tree-pruning power of valid cuts is still available.

1.1.2 Consistency

The basic concept for understanding the effect of valid cuts is "consistency." A consistent constraint set is not necessarily satisfiable, as the name might suggest. Rather, it is a constraint set that explicitly rules out all assignments to variables that cannot be part of a solution. For example, the constraint set (1.1) is inconsistent because no single constraint excludes the assignment $x_1 = 0$, and yet there are no feasible solutions in which $x_1 = 0$. A consistent constraint set is roughly analogous to one with an integral polyhedron, because it can be solved without backtracking.

Because it is usually impractical to obtain full consistency, various weaker forms of consistency have been defined: k-consistency, arc consistency, path consistency, adaptive consistency, etc. The basic algorithmic strategy is two-pronged: increase the degree of consistency by adding valid cuts, and choose an appropriate order in which to branch on variables. An analogous strategy is followed in mathematical programming, where cutting planes are used in combination with heuristics that decide on which variable to branch next. The constraint satisfaction approach, however, is supported by a theory that relates the branching order to problem structure. It can be shown that a lesser degree of consistency is needed to reduce backtracking for some branching orders than for others.

Valid cuts can in general be derived by logical inference. In fact, a cut is nothing other than a logical implication of the constraint set. In other words, it is a constraint that is satisfied by all feasible solutions of the constraint set. The cutting plane algorithms of integer programming, such as Chvátal's method (1973), are specialized methods of logical inference. It is shown here that the well-known resolution method of logical inference, and its generalization to multivalent variables, is a general method of deriving valid cuts. It is somewhat analogous to Chvátal's method, because it produces all valid cuts that take a certain logic form (i.e., clausal form). In addition, suitable modifications of the resolution algorithm achieve various kinds of consistency. The constraint satisfaction literature usually describes consistency-achieving algorithms in a slightly different way (from a semantic rather than a syntactic view).

The algorithms presented here are essentially those in the literature, but the resolution-based approach allows a more unified treatment.

1.1.3 The Constraint Satisfaction Field

Constraint solving began to appear in the computer science and artificial intelligence literatures in the 1970's. The idea of formulating a problem with constraints of course lies at the heart of operations research, and much of mathematics for that matter. But the computer science and AI communities have given it a different flavor because of their interest in logic programming and algorithmic science. In this environment, a declarative or constraint-based representation of a problem seemed a fresh idea that could complement procedural representations.

Some of the algorithms developed to solve constraints are analogous to those of mathematical programming. Enumerative searches similar to branch-and-bound, for example, are widely used. Constraint propagation techniques are reminiscent of some preprocessing algorithms. As shown here, techniques for achieving consistency are analogous to cutting plane algorithms. There are differences, however, that suggest that much is to be gained by synthesizing the two fields. Logic-based methods, for example, are much more evident in the constraint satisfaction literature than polyhedral methods. This literature also discusses the use of "nogoods" is a constraint-based search, which generalizes the use of Benders cuts (see Hooker, 1995). It presents a more systematic analysis of exhaustive search methods that views traditional branching as a very special case (see Ginsberg, 1993; Ginsberg and McAllester, 1994; McAllester, 1993).

The idea of a declarative formulation is already implicit in logic programming. In fact, logic programming languages such as Prolog were conceived as providing both a declarative and procedural representation of a problem. Yet there is no obvious notion of constraint solving, because the problem is solved by interpreting the Prolog statements procedurally. Jaffar and Lassez (1987) pointed out, however, that the "unification" step in the solution procedure is actually a special case of constraint solving. This provided a overarching framework for logic programming and constraint solving in which unification is extended to constraint solving in general. The result is *constraint logic programming*. The system CLP(R), for example, combines logic programming with linear constraint solving (Jaffar et al, 1992).

Constraint satisfaction techniques have since been incorporated into various programming frameworks other than classical logic programming languages. This gave rise to the field of *constraint programming* and such packages as CHIP and the ILOG Solver, which have had considerable commercial success

in solving problems very similar to some of those that have occupied the operations research profession. PROLOG IV also uses a number of constraint solving techniques in its logic programming framework. These packages are also beginning to include linear programming and other optimization modules. Conversely, the developers of AMPL are experimenting with links to the ILOG Solver (Fourer, 1997).

Standard references in constraint satisfaction include a book by Van Hentenryck (1989), now somewhat dated, and one by Tsang (1993), which is unfortunately out of print. A recent survey of the constraint programming field by van Hentenryck and Saraswat (1996) lists a large number of references. Connections between logic-based methods and mathematical programming are explored by Hooker (1994, 1994a) as well as Hooker and Osorio (1996).

1.1.4 Outline of the Paper

The paper begins by presenting the resolution method of logical inference. In its classical form it assumes that variables are binary, but it can be generalized to accommodate multivalued variables. It then presents the important idea of the dependency graph for a constraint set, which indicates how variables are coupled by the constraints. The "width" of the graph, which is defined with respect to a given ordering of the variables, indicates the degree to which the variables are coupled.

The idea of k-consistency, a form of partial consistency, is then defined. The basic result is that a k-consistent problem can be solved without backtracking if the width of its dependency graph, with respect to the branching order, is less than k. It is shown that k-resolution, a restriction of resolution, achieves k-consistency.

Adaptive consistency, a kind of local consistency, is also defined, and another restriction of resolution is shown to achieve it. The resolution cuts added to work toward adaptive consistency are roughly analogous to separating cuts used in branch-and-cut algorithms. Achieving adaptive consistency adds arcs to the dependency graph, resulting in the "induced graph," whose width is the induced width of the original graph. It is shown that the worst-case complexity of a constraint satisfaction problem varies exponentially with the induced width of its dependency graph.

It may be more effective in practice to find a search order that reduces the "bandwidth" rather than the width or induced width, because the bandwidth is the maximum number of levels one must backtrack when an infeasible node is encountered. The bandwidth is an upper bound on the induced width, and an algorithm for minimizing it is presented.

Finally, it should be noted that the constraint satisfaction literature reviewed here is concerned with finding feasible solutions, not with optimization. Feasibility is of course an important issue in its own right. The most obvious way to adapt constraint satisfaction techniques to optimization is to treat any known bounds on the objective function value as constraints.

There is also a deeper connection with optimization that is not explored here. It is noted below that the induced width of a graph is the smallest k for which the graph is a partial k-tree. But an optimization problem whose dependency graph is a partial k-tree can be solved by nonserial dynamic programming in time that is at worst exponential in k (Bertele and Brioschi, 1972; Arnborg and Proscurowski, 1986; Chhajed and Lowe, 1994). The objective function is regarded as partially separable, and variables that occur in the same component are represented as coupled in the dependency graph.

1.2 LOGICAL PRELIMINARIES

As noted above, the resolution method of logical inference provides a general scheme for generating valid cuts that is somewhat analogous to Chvátal's method. When all the variables are binary, the classical resolution method for propositional logic is adequate. This because any constraint in binary variables is equivalent to a formula of propositional logic. When some of the variables are multi-valued, resolution must be generalized to apply in a logic of discrete variables.

1.2.1 Propositional Logic

Propositional logic consists of *atomic propositions* x_j that are combined with such connectives as 'and,' 'or' and 'not' to create longer formulas. The truth value of a formula is determined in the obvious way by the truth values of its atomic propositions.

The formulas of interest here are those in *clausal form* (or *conjunctive normal form*, CNF), meaning that they are conjunctions of logical clauses. A *clause* is a disjunction of *literals*, which are atomic propositions or their negations. For example, the following clause contains three literals,

$$x_1 \vee \neg x_2 \vee x_3, \tag{1.2}$$

where $\vee$ means 'or' and $\neg$ means 'not.' The classical satisfiability problem (SAT) of propositional logic, which was the original NP-complete problem, is to determine whether a given formula in clausal form can be true.

A formula F *implies* formula G if any truth assignment to atomic propositions that makes F true also makes G true. It is easy to see that a clause C implies a clause D if and only if C *absorbs* D; i.e., all the literals of C occur in

D. A set S of formulas implies G if the conjunction of formulas in S implies G. Formulas F and G are *equivalent* if they imply each other. There is no loss of generality in dealing exclusively with clausal form, because any given formula is equivalent to some CNF formula.

In fact, any given constraint that contains only binary variables is equivalent to a CNF formula. The values of the variables can be interpreted as 'true' and 'false.' The constraint can therefore be viewed as a proposition that is true when it is satisfied. As such, it is equivalent to some CNF formula.

In particular, a 0-1 inequality is equivalent to a CNF formula when the values 0 and 1 are interpreted as 'false' and 'true,' respectively. For example, the inequality

$$4x_1 + 2x_2 + x_3 \geq 3 \tag{1.3}$$

is 'true' whenever x_1 is 'true' (i.e., $x_1 = 1$) and whenever x_2, x_3 are 'true.' The inequality is therefore equivalent to the conjunction of the clauses,

$$x_1 \vee x_2$$
$$x_1 \vee x_3.$$

In general, a 0-1 inequality can be written $ax \geq \alpha$ with $a \geq 0$ (replacing x_j with $1 - x_j$ if necessary). The inequality implies the clause

$$\bigvee_{j \in J} x_j$$

if and only if

$$\sum_{j \notin J} a_j < \alpha.$$

An inequality is equivalent to the set of all the clauses it implies (many of which are redundant, of course).

1.2.2 Discrete Variable Logic

Propositional logic is easily extended to a *discrete variable logic* that represents constraints with multivalued variables. It contains all the formulas of propositional logic, except that the atomic formulas x_j are replaced with predicates of the form

$$y_j \in Y,$$

where y_j is a multivalent discrete variable. Each variable y_j has a finite *domain* D_j of values it may assume, and $Y \subset D_j$. If each D_j contains two values, the logic reduces to ordinary propositional logic. It may be useful to introduce additional predicates, such as

$$\text{all-different}(y_1, \ldots, y_m),$$

which states that $y_1, \ldots, y_m$ must all take different values.

A *multivalent clause* has the form

$$\bigvee_{j=1}^{n} (y_j \in Y_j), \tag{1.4}$$

where each $Y_j \subset D_j$. If Y_j is empty, the term $(y_j \in Y_j)$ can be omitted from (1.4), but it is convenient to suppose that (1.4) contains a term for each j. If $y_j = D_j$ for some j, then (1.4) is a tautology. Note that the literals of a multivalent clause contain no negations. This brings no loss of generality, because $\neg(y_j \in Y_j)$ can be written $y_j \in D_j \setminus Y_j$.

A formula F of discrete variable logic implies another formula G if all the assignments of values to variables that make F true also make G true. One multivalent clause $\bigvee_j (y_j \in Y_{1j})$ implies another $\bigvee_j (y_j \in Y_{2j})$ if and only if the former absorbs the latter; i.e., $Y_{1j} \subset Y_{2j}$ for each j. Two formulas are equivalent if they imply each other. Any formula of discrete variable logic is equivalent to a conjunction of multivalent clauses.

A constraint involving discrete multivalued variables can be viewed as a proposition that is true when it is satisfied by the values of its variables. Any such constraint is therefore equivalent to a conjunction of multivalent clauses.

This is true in particular of linear inequalities in bounded integer variables. For example, if each y_j has domain $\{0, 1, 2\}$, then

$$4y_1 + 2y_2 + y_3 \geq 3$$

is equivalent to the conjunction of the multivalent clauses,

$$\begin{aligned} &(y_1 \in \{1, 2\}) \vee (y_2 \in \{2\}) \vee (y_3 \in \{1, 2\}) \\ &(y_1 \in \{1, 2\}) \vee (y_2 \in \{1, 2\}). \end{aligned}$$

In general, suppose that inequality $ay \geq \alpha$ has $a \geq 0$ with each $y_j \in \{0, 1, \ldots, M_j\}$ (replacing y_j with $M_j - y_j$ if necessary). Then $ay \geq \alpha$ implies the clause

$$\bigvee_{j=1}^{n} (y_j \in Y_j)$$

if and only if

$$\sum_{j=1}^{n} a_j \max_{v \in D_j \setminus Y_j} \{v\} < \alpha.$$

1.2.3 Resolution in Propositional Logic

The *resolution method* is a complete inference method for formulas in clausal form, up to absorption. This means that given any clause C that is implied by a set S of clauses, the resolution method derives from S a clause that absorbs C. Because any constraint set in binary variables is equivalent to a clause set, resolution can generate all valid clausal cuts for such a constraint set. We will also see that resolution can modified so that it achieves various kinds of consistency.

Classical resolution was developed by Quine (1952,1955) for propositional logic and extended by Robinson (1965) to predicate logic. Various forms of it are widely used in logic programming and theorem proving systems. Its worst-case complexity was first investigated by Tseitin (1968) and shown by Haken (1985) to be exponential. In practice, however, one would use resolution to generate only a few cuts rather than carry it to completion.

Resolution for propositional logic is defined as follows. Suppose two clauses are given for which exactly one atomic proposition occurs positively in one and negatively in the other.

$$\begin{array}{l} x_1 \vee x_2 \vee x_3 \\ \neg x_1 \vee x_2 \vee \neg x_4 \end{array} \tag{1.5}$$

The *resolvent* of the clauses is the disjunction of all literals that appear in either, except the two that are complements of each other:

$$x_2 \vee x_3 \vee \neg x_4. \tag{1.6}$$

Resolution reasons by cases. The atom x_1 is either true or false. If it is true, then $x_2 \vee \neg x_4$ must be true. If it is false, then $x_2 \vee x_3$ must be true. In either case, (1.6) is true.

If the clauses (1.5) are written as inequalities, the resolvent is actually a rank 1 cutting plane for the polytope described by them and $0 \leq x_j \leq 1$. This is seen by taking the linear combination below, where the first two inequalities represent the clauses (1.5), and the weights are indicated in parentheses:

$$\begin{array}{rrrrrr} x_1 & +x_2 + x_3 & & \geq 1 & (1/2) \\ (1-x_1) + x_2 & & +(1-x_4) & \geq 1 & (1/2) \\ & x_3 & & \geq 0 & (1/2) \\ & & (1-x_4) & \geq -1 & (1/2) \end{array}$$

The combination yields $x_2 + x_3 + (1 - x_4) \geq 1/2$, which becomes a rank 1 cut when the $1/2$ is rounded up. It is equivalent to the resolvent (1.6). Further connections between resolution and cutting planes are presented by Hooker (1989) and Hooker and Fedjki (1990).

```
Let S be a set of clauses and set S' = S.
While S' contains two clauses with a resolvent
R that is absorbed by no clause in S' {
    Remove all clauses in S' absorbed by R.
    Add R to S'.
}
```

Figure 1.1 The resolution algorithm for propositional logic.

The resolution method keeps generating resolvents for as long as possible, or until an empty clause is obtained, in which case the clauses are inconsistent. (The resolvent of x_j and $\neg x_j$ is the empty clause.) The precise algorithm appears in Fig. 1.1.

When the algorithm is done, S' contains all the strongest possible implications of S. That is, S' contains every *prime implication*, which is a clause implied by S that is absorbed by no other clausal implication of S. This means that resolution generates all clausal implications of S, up to absorption. For if S implies a clause C, then some prime implication of S absorbs C.

Theorem 1 (Quine) *The resolution method generates precisely the prime implications of a set of clauses and is therefore a complete inference method. In particular, it generates the empty clause if and only if the set is unsatisfiable.*

Prime implications are somewhat analogous to facet-defining inequalities, because they are the strongest possible clausal cuts. The concepts do not coincide, however, even in the realm of inequality constraints. Two different inequalities, only one of which defines a facet, may represent the same prime implication. For example, the 0-1 inequalities $x_1 + x_2 \geq 1$ and $2x_1 + x_2 \geq 1$ are valid cuts for the inequality system $x_1 + x_2 \geq 1,\ 0 \leq x_j \leq 1$, and both are logically equivalent to the prime implication $x_1 \vee x_2$. But only $x_1 + x_2 \geq 1$ defines a facet.

Resolution can be generalized so as to obtain all valid inequality cuts for a system of 0-1 inequalities, up to logical equivalence (Hooker, 1992). Barth (1995) specialized this approach to obtain cut generation techniques for inequalities of the form $\sum_{j \in J} x_j \geq k$. These inequalities seem to be a useful compromise between 0-1 inequalities and logical clauses, because they retain some of the expressiveness of the former and are yet amenable to logic processing.

1.2.4 Resolution in Discrete Variable Logic

Resolution is easily extended to the logic of discrete variables. Given a set of multivalent clauses,

$$\{\bigvee_{j=1}^{n}(y_j \in Y_{ij}) \mid i \in I\}, \tag{1.7}$$

the *resolvent on* y_k of these clauses is

$$(y_k \in \bigcap_{i\in I} Y_{ik}) \vee \bigvee_{j\neq k}(y_j \in \bigcup_{i\in I} Y_{ij}). \tag{1.8}$$

Ordinary bivalent resolution is a special case. The clauses in the set (1.7) are the *parents* of the resolvent (1.8).

For example, the first three clauses below resolve on y_1 to produce the fourth. Here each y_j has domain $\{1,2,3,4\}$.

$$\begin{gathered}(y_1 \in \{1,4\}) \vee (y_2 \in \{1\})\\ (y_1 \in \{2,4\}) \vee (y_2 \in \{2,3\})\\ (y_1 \in \{3,4\}) \vee (y_2 \in \{1\})\\ (y_1 \in \{4\}) \vee (y_2 \in \{1,2,3\})\end{gathered}$$

It is pointless to resolve the first three clauses on y_2, because this produces the tautology,

$$(y_1 \in \{1,2,3,4\}) \vee (x_2 \in \{1\}).$$

The multivalent algorithm is parallel to the classical algorithm and appears in Fig. 1.2.

```
Let S be a set of multivalent clauses and set S' = S.
While S' contains a subset of clauses with a resolvent
R that is absorbed by no clause in S' {
    Remove all clauses in S' absorbed by R.
    Add R to S'.
}
```

Figure 1.2 The resolution algorithm for discrete variable logic.

Note that there is no point is resolving on y_k if $Y_{ik} = \emptyset$ for some parent clause i, because the parent will imply the resolvent.

Prime implications can be defined for multivalent clauses in analogy with ordinary clauses. $\bigvee_{j=1}^{n}(y_j \in Y_j)$ is the empty clause if each $Y_j = \emptyset$. The following is proved by Hooker and Osorio (1996).

Theorem 2 *The multivalent resolution method generates precisely the prime implications of a set of multivalent clauses and is therefore a complete inference method. In particular, it generates the empty clause if and only if the set is unsatisfiable.*

The proof of the theorem shows that it suffices in principle to generate resolvents only of pairs of clauses.

1.3 THE DEPENDENCY GRAPH

The *dependency graph* for a set of variables and constraints, also called the *primal graph*, indicates the extent to which the variables are coupled by the constraints. The graph contains a vertex j for each variable x_j, and two vertices i, j are connected by an edge when x_i and x_j occur in the same constraint.

1.3.1 An Example

Consider for example the propositional satisfiability problem,

$$\begin{array}{rrrrrl} & & & x_4 \vee x_5 \vee x_6 & & (a) \\ & & x_3 & \vee x_5 & & (b) \\ x_1 \vee & x_2 \vee x_3 & & & & (c) \\ \neg x_1 \vee & \neg x_2 \vee x_3 & & & & (d) \\ \neg x_1 \vee & x_2 & \vee x_4 & & & (e) \\ x_1 \vee & \neg x_2 & \vee x_4 & & & (f) \end{array} \tag{1.9}$$

The dependency graph appears in Fig. 1.3.

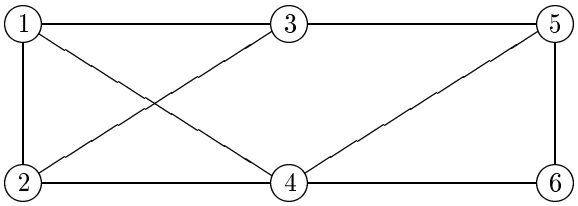

Figure 1.3 Dependency graph for a propositional satisfiability problem.

It stands to reason that the density of the dependency graph would have some relation to the difficulty of the constraint satisfaction problem. A sparse graph indicates that the problem can be decoupled to some extent.

Actually the difficulty of the problem is related more to the *width* of the constraint graph than its density. The width of a vertex, with respect to a total ordering of the vertices, is the number of adjacent vertices that precede it in the ordering. For instance, if the vertices of Fig. 1.3 are ordered as indicated, the widths of vertices $1, \ldots, 6$ are $0, 1, 2, 2, 2, 2$. The width of a graph with respect to an ordering is the maximum width of the vertices (2 in the example). The width of a graph *simpliciter* is its minimum width under all possible orderings. A graph of width 1 is a tree or forest.

The width of a graph is related to the amount of backtracking required to find a feasible solution by branching. This is discussed below. A related concept, the *induced width*, is related to the complexity of finding a feasible or optimal solution. This is discussed in Section 1.5.3.

1.3.2 Constraints and Satisfaction

Before exploring the connection between width and backtracking, it is necessary to clarify what it means to satisfy or violate a constraint.

Let $x_1, \ldots, x_n$ be the variables that occur in a constraint satisfaction problem. An *assignment* of values to variables has the form

$$(x_{j_1}, \ldots, x_{j_p}) = (v_{j_1}, \ldots, v_{j_p}), \tag{1.10}$$

where each $v_j \in D_{j_t}$. The assignment is *partial* if $\{x_{j_1}, \ldots, x_{j_p}\}$ is a proper subset of $\{x_1, \ldots, x_n\}$. Otherwise the assignment is *complete*.

By convention, a partial assignment can satisfy or violate a constraint only if it assigns values to all of the variables that occur in the constraint. For example, the partial assignment $x_1 = 1$ does not satisfy the 0-1 inequality $4x_1 + 2x_2 + x_3 \geq 3$, because x_2, x_3 have not been assigned values. Similarly, $(x_1, x_2) = (0, 0)$ does not violate the inequality.

1.4 CONSISTENCY AND BACKTRACKING

The importance of search order has long been recognized in the constraint satisfaction literature, and one of the ideas invoked to help understand this importance is that of *k-consistency*. The basic result is that if the dependency graph of a "strongly k- consistent" problem has width less than k under some ordering of the variables, then a feasible solution can be found without backtracking by branching on the variables in that order. It is therefore useful to make the width of a problem small and the degree of consistency high. The width can be reduced by reordering the variables. The degree of consistency can be increased by adding cuts.

A set of constraints is *k-satisfiable* if for every set of k variables occurring in the constraints, there is some assignment of values to these variables that

violates no constraint. If the constraints contain n variables altogether, they are *satisfiable* if they are n-satisfiable.

1.4.1 k-Consistency

Consistency is poorly named, because it suggests satisfiability but is neither necessary nor sufficient for satisfiability. Rather, a consistent constraint set is essentially one in which all implicit constraints have been made explicit. If there are partial assignments that violate no constraints but are part of no solution, the problem is inconsistent.

More precisely, a problem is inconsistent when some partial assignment $(x_{j_1}, \ldots, x_{j_k}) = (v_1, \ldots, v_k)$ violates no constraints, but there is no solution value of $(x_1, \ldots, x_n)$ in which $(x_{j_1}, \ldots x_{j_k}) = (v_1, \ldots v_k)$. The constraints implicitly exclude the partial assignment, but no constraint explicitly rules it out. Such an assignment is said in the constraint satisfaction literature to be a "redundant (compound) label" because it will have to be discarded in the search for a feasible solution. It will be described here as a *redundant assignment*.

Consistency allows one to find a feasible solution without backtracking. If no redundant assignments are encountered, no backtracking is necessary, because it is never necessary to go back a reset the value of a variable. At any node of an enumeration tree, certain variables $x_{j_1}, \ldots, x_{j_k}$ have been fixed by the branches taken between that node and the root. Consistency ensures that if the values so far assigned violate no constraints, then the next variable can be assigned some value without violating any constraints. So, one can obtain a feasible solution (if one exists) just by assigning each variable a value that, in combination with values already assigned, violates no constraints.

Any constraint set can in principle be made consistent by applying an inference algorithm that brings out all the implicit constraints. But the computational challenge is usually formidable, as it amounts to computing all the prime implications. In practice one must normally settle for a limited type of consistency known as k-consistency. But as noted earlier, this suffices to avoid backtracking if the width of the dependency graph (under the relevant ordering) has width less than k.

A constraint set is *1-consistent* if any given variable x_j can be assigned any value in its domain without violating any constraints.[2] The constraint set is k-consistent ($k > 1$) if any assignment to $k-1$ variables that violates no constraints can be extended to k variables without violating any constraints. That is, for every partial assignment $(x_{j_1}, \ldots, x_{j_{k-1}}) = (v_{j_1}, \ldots, v_{j_{k-1}})$ that violates no constraints and every variable $x_j \notin \{x_{j_1}, \ldots, x_{j_{k-1}}\}$, x_j can be given some value v_j in its domain for which $(x_{j_1}, \ldots, x_{j_{k-1}}, x_j) = (v_{j_1}, \ldots, v_{j_{k-1}}, v_j)$ violates no constraints.

It is important to note that k-consistency does not mean that any assignment to $k-1$ variables can be extended to a feasible solution. That is, k-consistency does not imply $(k+1)$-consistency and may permit redundant assignments to $k-1$ variables. For instance, the clause set

$$\begin{aligned} &x_1 \vee x_2 \vee x_3 \\ &x_1 \vee x_2 \vee \neg x_3 \\ &x_1 \vee \neg x_2 \vee x_3 \\ &x_1 \vee \neg x_2 \vee \neg x_3 \end{aligned}$$

is trivially 2-consistent because no assignment to two variables violates any constraints. Nonetheless, the assignment $x_1 =$ false cannot be extended to a feasible solution and is therefore redundant.

Conversely, a k-consistent problem need not be $(k-1)$-consistent. To adapt an example of Freuder (1982), the following constraint set is 3-consistent but not 2-consistent.

$$\begin{aligned} &\neg x_1 \\ &\neg x_3 \\ &x_1 \vee \neg x_2 \\ &\neg x_2 \vee x_3. \end{aligned}$$

It is 3-consistent because the only assignment to the pairs (x_1, x_2), (x_1, x_3) and (x_2, x_3) that violates no constraints is (F, F), and it can be extended to $(x_1, x_2, x_3) = (F, F, F)$. The problem is not 2-consistent because the assignment $x_2 = T$ violates no constraints but cannot be extended to a satisfying assignment for (x_1, x_2), or (x_2, x_3) for that matter. Because k-consistency does not entail $(k-1)$-consistency, it convenient to define a *strongly k-consistent* problem to be one that is t-consistent for $t = 1, \ldots, k$.

The example (1.9) is 1-consistent because there are no unit clauses; each variable x_j can be assigned true or false without violating any constraints. It is 2-consistent because for any pair of variables (x_i, x_j), either assignment to x_i can be extended to an assignment to (x_i, x_j) that falsifies no clauses. For instance, if $x_2 = F$, then x_5 can be true without violating any clauses, and the remaining variables can be assigned either value because they do not occur in a two-literal clause with x_2. It is easy to check that the problem is 2-consistent and 3-consistent as well. It is not 4-consistent, however, because if $(x_2, x_3, x_4) = (F, F, F)$, then x_1 cannot be assigned a value without violating a constraint.

As noted earlier, neither consistency nor satisfiability implies the other. The constraint set consisting of the unit clause x_1 is 1-satisfiable but not 1-consistent. Conversely, a problem in which the variables have empty domains is (strongly) 1-consistent but unsatisfiable. There is, however, the following connection between the two concepts.

```
Let C be a collection of constraints, and let the variables
    that occur in these constraints be ordered x_1, ..., x_n.
Let D_{x_j} be the domain of x_j.
For j = 1, ..., n {
   If there is a v_j ∈ D_{x_j} for which (x_1, ..., x_j) = (v_1, ..., v_j)
      violates no constraints in C then select such a v_j
   Else stop; C is unsatisfiable.
}
(x_1, ..., x_n) = (v_1, ..., v_n) satisfies C.
```

Figure 1.4 Zero-step lookahead algorithm.

Lemma 3 *A problem that is 1-satisfiable and strongly k-consistent is k-satisfiable.*

This is true because 1-satisfiability implies that x_1, for instance, can be given a value v_1 that violates no constraints. By 2-consistency, this can be extended to an assignment $(x_1, x_2) = (v_1, v_2)$ that violates no constraints, which can by 3-consistency be further extended, and so on until a solution is obtained.

1.4.2 k-Consistency and Backtracking

A strongly n-consistent constraint set can obviously be solved without backtracking, in the manner already described. But strong k-consistency for smaller k is sufficient if the width of the constraint graph is smaller than k with respect to some ordering of the variables. If one fixes the variables in the given order, then possible values for the next variable will never depend on the values of more than $k-1$ of the previously fixed variables. Strong k-consistency therefore suffices.

The precise algorithm appears in Fig. 1.4. It might be called a zero-step lookahead algorithm because one can determine by looking no further than the current node of the search tree which value the corresponding variable should be given.

Theorem 4 *If a constraint satisfaction problem is strongly k-consistent, and its dependency graph has width less than k with respect to some ordering, then for this ordering the zero-step lookahead algorithm finds a solution if and only if one exists.*

Proof. If the constraints are unsatisfiable then clearly the algorithm will at some point fail to find a value v_j for which $(x_1, \ldots, x_j) = (v_1, \ldots, v_j)$ violates

no constraints. Suppose, then, that the constraints are satisfiable. At any point in the algorithm a solution $(x_1, \ldots, x_{j-1}) = (v_1, \ldots, v_{j-1})$ has been found that does not violate any constraints (or else the algorithm would have already terminated). Because the dependency graph has width less than k with respect to the ordering $x_1, \ldots, x_n$, constraints involving x_j do not involve x_t for $t \leq j - k$. This means that $(x_1, \ldots, x_j) = (v_1, \ldots, v_j)$ violates no constraints if $(x_t, \ldots, x_j) = (v_t, \ldots, v_j)$ violates no constraints, where $t = \max\{1, j - k + 1\}$. But because it is given that $(x_t, \ldots, x_{j-1}) = (v_t, \ldots, v_{j-1})$ violates no constraints, strong k-consistency implies that there is a $v_j \in D_{x_j}$ for which $(x_t, \ldots, x_j) = (v_t, \ldots, v_j)$ violates no constraints. The algorithm therefore continues until $j = n$ and a solution is found.

The example (1.9) illustrates the principle. Suppose the variables are ordered $x_1, \ldots, x_6$. Because the width of the dependency graph (Fig. 1.3) is 2 and the problem is 3-consistent, the problem is soluble without backtracking. In the backtrack-free search tree of Fig. 1.5, for instance, the zero lookahead algorithm first tries in each iteration to set $x_j = 0$ and then, if necessary, $x_j = 1$. It obtains the solution $(x_1, \ldots, x_6) = (0, 0, 1, 0, 1, 0)$. However, if one uses the ordering $(x_3, x_4, x_1, x_2, x_5, x_6)$, then the dependency graph has width 3, and a backtrack-free search is not guaranteed. In fact, zero-step lookahead fails to find a solution. A search by branching results in the search tree of Fig. 1.6.

1.4.3 Binary Problems

Much of the constraint satisfaction literature focuses on "binary" problems, in which each constraint contains at most two variables. In this context a 1-consistent problem is called *node consistent*, and a 2-consistent problem is called *arc consistent*. Theorem 4 implies that an arc consistent binary problem can be solved without backtracking, for any ordering of the variables, if the dependency graph is a tree.

For binary problems one can also define *path consistency*, which implies that if $(x_0, x_j) = (v_0, v_j)$ violates no constraints, then for any path $x_0, x_{t_1}, \ldots, x_{t_k}$ in the dependency graph (where $x_{t_k} = x_j$), there is an assignment $(x_0, x_{t_1}, \ldots, x_{t_k}) = (v_0, v_{t_1}, \ldots, v_{t_k})$ for which each pair $(x_{t_{p-1}}, x_{t_p}) = (v_{t_{p-1}}, v_{t_p})$ violates no constraints. As with other types of consistency, the motivation for achieving path consistency is to remove redundant assignments. Binary problems will not be further discussed here because optimization problems encountered in practice tend not to be binary.

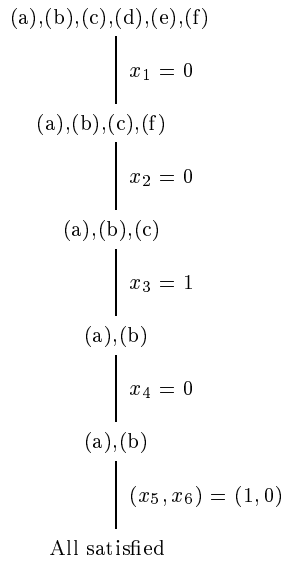

Figure 1.5 A solution of a satisfiability subproblem by zero-step lookahead. At each node are indicated the clauses that have not yet been satisfied (in the sense that no literals in the clause have been fixed to true).

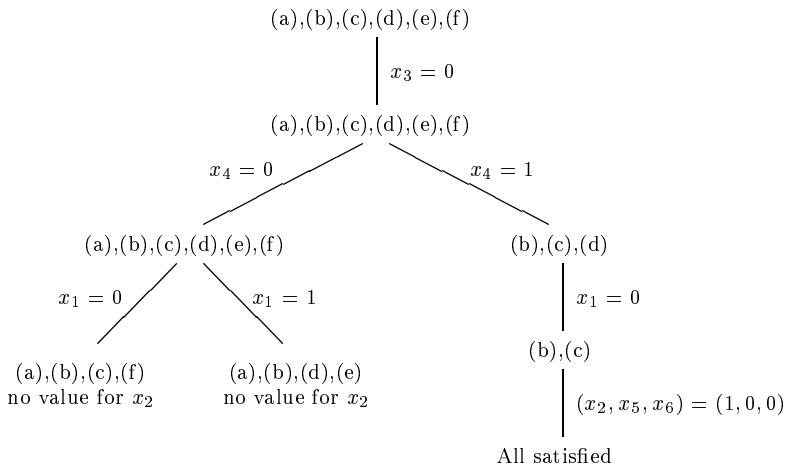

Figure 1.6 A solution of a satisfiability subproblem by branching. Backtracking can (and does) occur because the width of the dependency graph is 4 and the problem is only 3-consistent.

1.4.4 Achieving k-Consistency

Because it is generally impractical to derive all valid cuts, the art of cut generation is to produce a limited number of useful constraints. The concept of k-consistency suggests one way to do this: generate the cuts needed to make a problem k-consistent for small k. In the ideal case, k would exceed the width of the dependency graph with respect to the branching order. But in practice it may be advantageous to achieve k-consistency for a k that is too small to eliminate backtracking, or to work toward a higher degree of consistency without actually achieving it.

The constraint satisfaction literature dwells primarily on achieving various types of consistency in binary problems, such as node and arc consistency. But at least one algorithm has been proposed for making general constraint satisfaction problem k-consistent (Cooper, 1989). It simply accumulates a list of infeasible partial assignments to k variables and notes which assignments to $k-1$ variables have the property that all of their extensions are infeasible.

The process is actually a semantic interpretation of what happens in the resolution algorithm. Consider for instance the clauses

$$\begin{array}{ll} x_1 \vee x_2 \vee x_3 & \\ \neg x_1 \qquad \vee x_3 \vee x_4 & \end{array} \tag{1.11}$$

Because they are violated by the assignments $x = (F, F, F, F)$ and $x = (T, F, F, F)$, the assignment $(x_2, x_3, x_4) = (F, F, F)$ cannot be extended. This redundant assignment can be ruled out by adding the constraint

$$x_2 \vee x_3 \vee x_4,$$

which is precisely the resolvent of the two clauses (1.11).

It is not hard to see that k-consistency can be achieved for clauses by deriving all resolvents with fewer than k literals. If a problem is not k-consistent, then there are two assignments

$$\begin{array}{l} (x_{j_1}, \ldots, x_{j_k}) = (v_{j_1}, \ldots, v_{j_{k-1}}, T) \\ (x_{j_1}, \ldots, x_{j_k}) = (v_{j_1}, \ldots, v_{j_{k-1}}, F), \end{array} \tag{1.12}$$

each of which falsifies some clause, such that

$$(x_{j_1}, \ldots, x_{j_{k-1}}) = (v_{j_1}, \ldots, v_{j_{k-1}}) \tag{1.13}$$

falsifies no clause. Clearly the clauses falsified by (1.12) have a resolvent containing fewer than k literals. No other clause absorbs the resolvent because (1.13) falsifies no clause. So, when all resolvents shorter than k literals have been derived, the problem is k-consistent, and indeed strongly k-consistent.

For a set S of logical clauses, let k-resolution be the algorithm that a) adds to C all resolvents of clauses in S that contain fewer than k literals and that are not already absorbed by clauses in s, and b) repeats this procedure until no further such resolvents can be added. We have shown,

Theorem 5 *The k-resolution algorithm achieves k-consistency for a given set of clauses.*

Note that deriving all resolvents having fewer than k literals is not the same as carrying out all resolution proofs of clauses having fewer than k literals. The latter process may involve deriving resolvents with k or more literals. In fact it computes the projections of the clause set onto all sets of $k-1$ or fewer variables. It therefore removes all redundant assignments to $k-1$ or fewer variables, which as observed earlier is a larger task than achieving k-consistency.

k-consistency can be achieved for any constraint set with bivalent variables by applying k-resolution to a clause set to which it is equivalent. The clause set may be impracticably large, however. Barth (1995) reports, for example, that the single 0-1 inequality

$$\begin{aligned} &300x_3 + 300x_4 + 285x_5 + 285x_6 + 265x_8 + 265x_9 + 230x_{12} \\ &\quad +230x_{13} + 190x_{14} + 200x_{22} + 400x_{23} + 200x_{24} + 400x_{25} \\ &\quad +200x_{26} + 400x_{27} + 200x_{28} + 400x_{29} + 200x_{30} + 400x_{31} \leq 2700 \end{aligned} \tag{1.14}$$

is equivalent to a set of 117,520 nonredundant clauses. In practice, one would normally apply k-resolution to a limited number of clauses implied by the constraint. One need only consider clauses with at most k literals, and possibly only a small fraction of these would be generated. In addition, many common constraints are essentially logical propositions that are equivalent to one or a few clauses.

Multivalent generalization can be used to obtain k-consistency for multivalent clauses. The k-resolution algorithm is as before, and the proof is similar to the bivalent case.

Theorem 6 *The k-resolution algorithm achieves k-consistency for a given set of multivalent clauses.*

The multivalent k-resolution algorithm can in principle achieve k-consistency for any constraint set (whose variables have finite domains) if it is applied to the equivalent set of multivalent clauses.

1.5 ADAPTIVE CONSISTENCY AND K-TREES

One can achieve the degree of consistency necessary to avoid backtracking in a global or a local manner. A global approach has been described: achieve

k-consistency for a sufficiently large k before starting the search for a solution. A local approach is to add precisely those constraints are needed to achieve consistency at each node of the search tree. This is roughly analogous to a branch-and-cut algorithm in integer programming, where one adds separating cuts at each node of the search tree.

Like k-consistency, adaptive consistency may be computationally prohibitive if obtained to a degree that eliminates backtracking. In practice one can balance the expense of generating constraints to achieve consistency against the expense of traversing a large search tree.

1.5.1 Adaptive Consistency and Backtracking

The concept of adaptive consistency is motivated as follows. Suppose that the variables are given the branching order $x_1, \ldots, x_n$. At any node in the search tree, the first $t-1$ variables $x_1, \ldots, x_{t-1}$ have been fixed by branches taken between the root and that node. A value for x_t is sought that, together with the values already assigned, violates no constraints. But only certain of the first $t-1$ variables, perhaps variables $x_{j_1}, \ldots, x_{j_k}$, are adjacent to x_t in the problem's dependency graph. No backtracking is necessary if some value of x_t, together with the values assigned to these k variables, violates no constraint. For this it suffices that every assignment to these k variables can be extended to an assignment to these variables and x_t. This is adaptive consistency.

In general, a constraint set S has *adaptive consistency for variable* x_t with respect to an ordering $x_1, \ldots, x_n$ if the following is true: if $x_{j_1}, \ldots, x_{j_k}$ are the variables in $\{x_1, \ldots, x_{t-1}\}$ that are adjacent to x_t in the dependency graph for S, then for every assignment $(x_{j_1}, \ldots, x_{j_k}) = (v_1, \ldots, v_k)$ that violates no constraint in S, there is a $v \in D_{x_t}$ for which $(x_{j_1}, \ldots, x_{j_k}, x_t) = (v_1, \ldots, v_k, v)$ violates no constraint. S has adaptive consistency with respect to the ordering if it has adaptive consistency for every variable. The following should now be obvious.

Theorem 7 (Dechter and Pearl, 1988) *If a constraint set has adaptive consistency with respect to an ordering, then a solution can be found with the zero-step lookahead algorithm of Fig. 1.4.*

For example, consider the clausal constraint set (1.9), whose dependency graph appears in Fig. 1.3. Recall that the ordering $x_3, x_4, x_1, x_2, x_5, x_6$ generated the search tree of Fig. 1.6, which contains backtracking. One should therefore expect that the constraint set would not have adaptive consistency with respect to this ordering, and it does not. The variable x_2, for instance, is adjacent in the dependency graph to three variables preceding it in the ordering: x_1, x_3, and x_4. Two assignments to these variables, namely $(x_1, x_3, x_4) = (F, F, F)$

and (T, F, F), falsify no clauses but cannot be extended to assignments to (x_1, x_2, x_3, x_4) without falsifying a clause. So there is no adaptive consistency for x_2.

1.5.2 Achieving Adaptive Consistency

Adaptive consistency can be achieved for a given variable in a set of multivalent clauses by a restriction of the multivalent resolution algorithm. For a given ordering on the variables of a clause set S, let $x_{j_1}, \ldots, x_{j_k}$ be the variables that precede x_t in this ordering and that are adjacent to x_t in the dependency graph for S. Then what might be called the *adaptive resolution* algorithm proceeds as follows: a) derive from clauses in S all resolvents on x_t whose variables belong to $\{x_{j_1}, \ldots, x_{j_k}\}$ and that are not already absorbed by clauses in S, b) add the resolvents to S, and c) repeat this procedure until no such resolvents can be derived.

Theorem 8 *The adaptive resolution algorithm achieves adaptive consistency for any given variable x_t in a set of multivalent clauses, with respect to any given ordering of the variables.*

The addition of resolvents can change the dependency graph. Because the variables in a resolvent may not all occur in the same parent, every pair of them should be connected by an arc.

One can achieve adaptive consistency with respect to a given ordering by achieving it for each variable, one at a time. It is necessary, however, to treat the variables in reverse order, beginning with the last variable x_n and working toward x_1. Otherwise the constraint added to achieve adaptive consistency for one variable may change the constraint set in such a way as to nullify the work already done for another variable. In fact, the added constraint may change the dependency graph, because it may relate variables that did not previously occur in a common constraint.

Working backward avoids this problem for the following reason. Suppose that a resolvent R was added to achieve consistency for x_t. A resolvent R' later added to achieve consistency for any x_i preceding x_t may change the constraint set and the dependency graph, but it cannot destroy the adaptive consistency achieved for x_t. This is because R' contains only variables that precede x_i and therefore does not contain x_t. So any assignment to the variables in R that violates a constraint after the addition of R' did so before the addition of R'. This means that adaptive consistency for x_t is undisturbed.

As an example consider again the clausal set (1.9) and the ordering x_3, x_4, x_1, x_2, x_5, x_6. Adaptive consistency is achieved for each variable as shown in Table 1.1. Consider x_2, for instance. x_1, x_3, x_4 are the variables prior to x_2 in

Table 1.1 Achieving adaptive consistency for each variable.

Variable	*Previous variables adjacent in dependency graph*	*Resolvents generated*	*Arcs added to dependency graph*
x_6	x_4, x_5	none	none
x_5	x_3, x_4	none	none
x_2	x_1, x_3, x_4	$x_1 \vee x_3 \vee x_4$ $\neg x_1 \vee x_3 \vee x_4$	(x_3, x_4)
x_1	x_3, x_4	$x_3 \vee x_4$	none
x_4	x_3	none	none
x_3	none	none	none

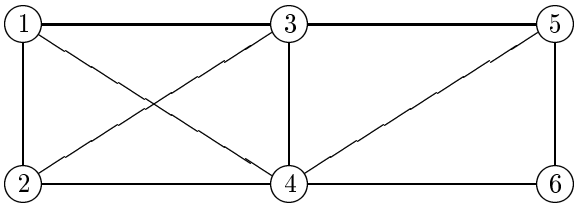

Figure 1.7 Dependency graph after achieving adaptive consistency.

the ordering that are adjacent to x_2. Clauses (c) and (f) in (1.9) are resolved on x_2 to obtain $x_1 \vee x_3 \vee x_4$, whose variables belong to $\{x_1, x_3, x_4\}$, and (d) and (e) are similarly resolved to obtain $\neg x_1 \vee x_3 \vee x_4$. These two resolvents are themselves resolved in the next stage to achieve adaptive consistency for x_1. One arc is added to the dependency graph in the process, resulting in the graph of Fig. 1.7

1.5.3 Induced Width and k-Trees

It was seen above that constraints added to achieve adaptive consistency can add arcs to the dependency graph. The new arcs can actually increase the width. The maximum width that can result is the *induced width*, which is important because it is closely related to the worst-case complexity of solving the problem.

The induced width for a given ordering $x_1, \ldots, x_n$ of the variables is defined as follows. Beginning with x_n, remove the nodes from the graph one at a time. When each x_t is removed, add new arcs as needed to connect all remaining nodes adjacent to x_t. The dependency graph, augmented by the new arcs, is the *induced graph*, whose width is the *induced width* relative to the ordering. The induced width of a graph *simpliciter* is the minimum induced width over all orderings.

The graph of Fig. 1.3 has width 2 for the ordering $x_1, \ldots, x_6$ but induced width 3. When x_6 is removed, no arcs are added, but arc (x_3, x_4) is added when x_5 is removed. No further arcs are added, and the induced graph is that of Fig. 1.7. The induced width is the width of the induced graph, which is 3.

The idea of a k-tree is closely related to the induced width of a graph. A *k-tree* is a graph G such that for some ordering $j_1, \ldots, j_n$ of the vertices, each vertex is connected to k others at the time of removal if the vertices are removed from G in reverse order. More precisely, there is a sequence of graphs $G_k, \ldots, G_n$ such that a) G_k is a complete graph, b) $G_n = G$, c) each G_t $(t = k, \ldots, n-1)$ is obtained by removing from G_{t+1} the vertex j_{t+1} and all arcs adjacent to j_{t+1} in G_{t+1}, and d) each vertex j_t $(t = k+1, \ldots, n)$ is adjacent in G_t to exactly k vertices. Any subgraph of a k-tree is a *partial k-tree*.

Lemma 9 *The induced width of a graph is the smallest k for which the graph is a partial k-tree.*

Proof. Let w be the induced width of G. Then there is an ordering of vertices $j_1, \ldots, j_n$ for which the induced graph $\bar{G}$ has width w. Let w_t be the number of vertices in $\{j_1, \ldots, j_{t-1}\}$ that are adjacent to j_t in $\bar{G}$. Create a graph G' whose vertices and arcs are those of $\bar{G}$, plus arcs from each vertex j_t $(t = k+1, \ldots, n)$ to any $w - w_t$ vertices in the list $j_1, \ldots, j_{t-1}$. Then G' is a w-tree, which means that G (a subgraph of G') is a partial w-tree. Furthermore, if G were a partial k-tree for $k < w$, then the induced width of G would be at most k and therefore less than w.

1.5.4 Induced Width and Complexity

The worst-case complexity of achieving adaptive consistency is related exponentially to the induced width. The time required to generate the resolvents

that achieve adaptive consistency for x_t is essentially the time required to determine which assignments to $(x_{j_1}, \ldots, x_{j_k}, x_t)$ violate some constraint. The latter is at worst proportional to d^{k+1}, where d is the maximum size of the variable domains D_{x_j}. So the total time to achieve adaptive consistency is at worst proportional to nd^{k+1}. This also bounds the time required to solve the problem, because backtracking is no longer necessary.

Theorem 10 *A constraint satisfaction problem can be solved in time that is at worst proportional to nd^{k+1}, where n is the number of variables, d the size of the largest variable domain D_{x_j}, and k the induced width of the dependency graph with respect to some ordering.*

1.6 MINIMUM WIDTH ORDERINGS

The previous section aimed to decrease backtracking by achieving strong k-consistency for some k, ideally where k exceeds the width of the dependency graph. A complementary approach is to choose a branching order that reduces the width.

1.6.1 Finding a Minimum-Width Ordering

There is a simple greedy algorithm that finds the minimum width ordering of the variables $x_1, \ldots, x_n$. It chooses the variable x_{j_n} with the smallest degree (number of adjacent nodes). After removing x_{j_n} and all its incident arcs from the dependency graph, it chooses $x_{j_{n-1}}$ similarly, and so forth until all nodes have been removed.

Theorem 11 (Freuder, 1982) *The above greedy algorithm finds a minimum width ordering $x_{j_1}, \ldots, x_{j_n}$.*

The proof, omitted here, is nontrivial.

In the dependency graph of Fig. 1.3, the greedy algorithm removes nodes in the order $x_6, x_5, \ldots, x_1$. Therefore $x_1, \ldots, x_6$ is a minimum width ordering, which corresponds to a width of 2.

1.6.2 Minimum Bandwidth Orderings

The strategy of ordering variables so as to minimize width has two weaknesses. One is that it although it eliminates backtracking if the width is less than k for a k-consistent problem, it may have no beneficial effect if the width is only slightly greater. When the width is less than k, at any point in a tree search the variable x_t to be fixed next interacts with at most $k-1$ variables that are fixed higher in the tree, and k-consistency ensures that x_t can be given a value

without violating constraints. However, if the width is k or greater, x_t can interact with k or more variables, and there may be no feasible value for x_t. In this case one must backtrack to where the last of these k or more variables is fixed. If it was fixed at a much higher point in the tree (i.e., much closer to the root), a width of k or greater is consistent with a good deal of backtracking.

This reasoning suggests that the variables with which a given variable interacts should occur within a small range of elevations in the tree, so that one is never obliged to backtrack very far. That is, variables ought to be ordered so as to minimize the *bandwidth*. For a given ordering $x_{j_1}, \ldots, x_{j_n}$ the bandwidth is the maximum of $|t - u|$ over all arcs (x_{j_t}, x_{j_u}) in the dependency graph.

Minimizing bandwidth also addresses a second weakness of minimum width orderings. This is the fact that the width provides no bound on the complexity of solving the problem. Indeed it might appear that one should minimize *induced* width, because induced with provides such a bound. However, one can bound the complexity *and* limit how far one backtracks by minimizing induced width, due to the following fact.

Theorem 12 (Zabih, 1990) *For any given ordering of the variables, the bandwidth of the dependency graph is greater than or equal to the induced width.*

This suggests minimizing bandwidth as an attractive ordering heuristic.

The reasoning behind Theorem 12 is simple enough. It begins with a lemma.

Lemma 13 *The bandwidth of a graph with respect to an ordering is an upper bound on the width.*

This is because if the bandwidth is k, the nodes that precede any node x_t in the ordering and that are adjacent to x_t in the dependency graph must be within distance k of x_t in the ordering. So at most k nodes are before and adjacent to x_t.

Suppose now that a problem has bandwidth k with respect to some ordering, and that nodes are removed from the dependency graph in reverse order to determine which additional arcs are in the induced graph. When x_t is removed, all adjacent nodes before it in the ordering are within a distance k of x_t. So adding arcs between these nodes cannot increase the bandwidth. This means that the bandwidth of the induced graph is the same as the bandwidth of the original graph. But the bandwidth of the induced graph is by Lemma 13 greater than or equal to the width of the induced graph, and Theorem 12 follows.

1.6.3 Finding a Minimum Bandwidth Ordering

Dynamic programming algorithms proposed by Saxe (1980) and improved by Gurari and Sudburough (1984) can be used to find a minimum bandwidth

ordering of variables. Following Tsang (1993), a tree search version of the algorithm is presented here. The complexity is exponential in the worst case, but the algorithm uses clever pruning strategies that can reduce the search substantially.

The algorithm actually checks whether there is an ordering of variables for which the dependency graph has bandwidth of at most k. It assumes that the graph is connected, which is innocuous, because components of a disconnected graph can be handled separately.

The algorithm builds an ordered list of variables one by one. At a given node of the search tree variables $x_1, \ldots, x_d$ have been ordered. If x_t is the first of these variables that is adjacent to some unordered node, the algorithm stores only the sublist $x_t, \ldots, x_d$. The remainder of the list is implicit in another data structure that is maintained: the set of all arcs from $x_t, \ldots, x_d$ to unordered nodes.

The algorithm branches on which node x_j is to be added to the end of the list $x_t, \ldots, x_d$. It need only consider unordered x_j's that are adjacent to x_t; because the graph is connected, any x_t that is not adjacent to x_t (and therefore adjacent to none of $x_1, \ldots, x_t$) is adjacent to some later variable in the list, and it can be considered at that point without consequence.

For each candidate x_j the algorithm checks a simple necessary condition for the possibility of extending the list further if x_j is added now. To do this it updates the data structure by adding x_j to the end of the list $x_t, \ldots, x_d$ and deleting nodes from the beginning of the list that are no longer adjacent to an unordered node (because x_j was the only unordered node to which they were adjacent). The resulting list can be written $x_{t'}, \ldots, x_d, x_j$. Now if this list contains p nodes, $x_{t'}$ must be adjacent to no more than $k - p + 1$ unordered nodes. Otherwise some node will perforce be further than k from $x_{t'}$ in any extension of this ordering. Similarly $x_{t'+1}$ must be adjacent to no more than $k-p+2$ unordered nodes, and so forth. If any of these conditions are violated, there is no need to branch to a child node at which x_j is added to the list.

The precise algorithm appears in Figs. 1.8 and 1.9. The dynamic programming nature of the algorithm is captured by traversing the tree in a breadth-first fashion. This is achieved by treating the list A of active nodes as a first-in first-out queue. The algorithm can be applied repeatedly for $k = 1, 2$, etc. The smallest k for which an ordering is found is the minimum bandwidth.

An application of the algorithm to the graph of Fig. 1.3 is depicted in Fig. 1.10. Here k is set to 2. The possible first members of the ordered list, namely $x_1, \ldots, x_6$, are enumerated in the first level of the tree. All but x_6 can be excluded. For instance, the list (x_1) cannot be extended because x_1 is adjacent to 3 unordered nodes, and $3 > k - p + 1 = 2$. Because all of the leaf nodes

```
Let G be the dependency graph of a constraint set.
To be determined is whether there is an ordering of variables
   for which G has bandwidth ≤ k.
Let A be the set of active nodes (V, E) of the search tree,
   where initially V = E = ∅.
Initialize done(V, E) = false for all ordered subsets V
   of nodes of G and all subsets E of arcs of G.
While A is nonempty {
   Remove a node (V, E) from A, and let V = {x_1, ..., x_d}.
   For each node x_j with (x_1, x_j) ∈ E (or , if E = ∅,
   for each node x_j of G) {
      Let (V', E') = Child(V, E, x_j).
      If E' = ∅ then
         Stop; there is an ordering with bandwidth ≤ k.
      Else if not done(V', E') and Check(V', E', x_j) then {
         Let done(V', E') = true.
         Add (V', E') to A.
      }
   }
}
```

Figure 1.8 Tree search algorithm for checking whether a dependency graph has bandwidth $\leq k$ for some ordering.

```
Function Child(V, E, x_j) {
   Create a child of (V, E) in the search tree by adding
      x_j to the ordered list V and making adjustments
      accordingly.
   Remove from E each arc of the form (x_t, x_j).
   Let V = (x_1, ..., x_d).
   Let i' be the largest i(≤ d) for which x_i is incident to
      no arc in E.
   For each arc in G of the form (x_j, x_t) with x_t ∉ V
     Add (x_j, x_t) to E.
   Return ((x_i', ..., x_d, x_j), E).
}
Function Check(V, E, x_j).  {
   Check whether it appears possible to extend the ordered
      list V by adding x_j to the end.
   If |V| > k then return false.
   Else {
      Let V = (x_1, ..., x_d).
      For i = 1, ..., d {
         If E contains more than k − |V| + i arcs of the
            form (x_i, x_t) then return false.
      }
   }
   Return true.
}
```

Figure 1.9 Procedures Child and Check for the bandwidth algorithm.

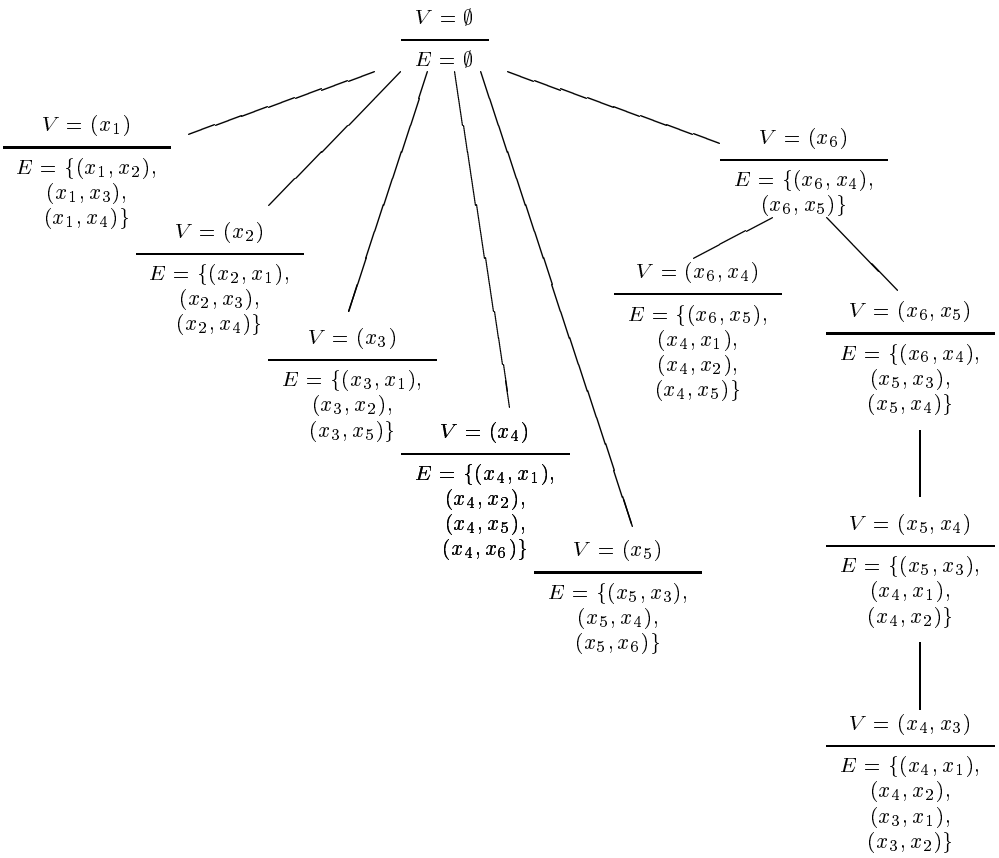

Figure 1.10 Execution of the bandwidth algorithm showing that there is no ordering with bandwidth $k = 2$.

fail the condition applied in `Check`, the tree proves that there is no ordering with bandwidth $k = 2$. If $k = 3$, the ordering $x_1, \ldots, x_6$ is found right away.

Notes

1. Technically, to achieve strong 2-consistency, the cut $x_1 \geq 1$ must be used to reduce the domain of x_1 from $\{0, 1\}$ to $\{1\}$ rather than as an explicit constraint. This is because the cut has only one variable.

2. The definition of 1-consistency is not analogous with that if k-consistency for $k > 1$. An analogous definition would say that a constraint set is 1-consistent if every variable either has an empty domain or can be assigned some value in its domain without violating any constraints.

References

Arnborg, S., and A. Proskurowski (1986). Characterization and recognition of partial k-trees, *SIAM Journal on Algebraic and Discrete Mathematics* **7**, 305-314.

Barth, P. (1995). *Logic-Based 0-1 Constraint Solvingin Constraint Logic Programming*, Dordrecht: Kluwer.

Bertele, U., and F. Brioschi (1972). *Nonserial Dynamic Programming*, New York: Academic Press.

Chvátal, V. (1973). Edmonds polytopes and a hierarchy of combinatorial problems, *Discrete Mathematics* **4**, 305-337.

Chhajed, D., and T. J. Lowe (1994). Solving structured multifacility location problems efficiently, *Transportation Science* **28**, 104-115.

Cooper, M. C. (1989). An optimal k-consistency algorithm,*Artificial Intelligence* **41**, 89-95.

Dechter, R., and J. Pearl (1988). Tree-clustering schemes for constraint processing, *Proceeedings, National Conference on Artificial Intelligence (AAAI)*, 150-154.

Fourer, R. (1997). Proposed new AMPL features (web page), http://achille.cs.bell-labs.com/cm/cs/what/AMPL/NEW/FUTURE/logic.html.

Freuder, E. C. (1982). A sufficient condition for backtrack-free search, *Journal of the ACM* **29**, 24-32.

Ginsberg, M. L. (1993). Dynamic backtracking, *Journalof Artificial Intelligence Research* **1**, 25-46.

Ginsberg, M. L., and D. A. McAllester (1994). GSAT anddynamic backtracking, *Second Workshop on Principles and Practice of Constraint Programming*, 216-225.

Gurari, E., and I. Sudborough (1984). Improved dynamic programming algorithms for bandwidth minimization and the mincut linear arrangement problem, *Journal of Algorithms* **5**, 531-546.

Haken, A. (1985). The intractability of resolution, *Theoretical Computer Science* **39**, 297-308.

Hooker, J. N. (1989). Input proofs and rank one cutting planes, *ORSA Journal on Computing* **1**, 137-145.

Hooker, J. N. (1992). Generalized resolution for 0-1 linear inequalities, *Annals of Mathematics and Artificial Intelligence* **6**, 271-286.

Hooker, J. N. (1994). Logic-based methods for optimization, in A. Borning, ed., *Principles and Practice of Constraint Programming,Lecture Notes in Computer Science* **874**, 336-349.

Hooker, J. N. (1994a). Tutorial in logic-based methods for optimization, *CSTS Newsletter*, Fall issue.

Hooker, J. N. (1995). Logic-based Benders decomposition, available on http://www.gsia.cmu.edu/afs/andrew/gsia/jh38/jnh.html.

Hooker, J. N. and C. Fedjki (1990). Branch-and-cut solution of inference problems in propositional logic, *Annals of Mathematics and AI* **1**, 123-140.

Hooker, J. N., and M. A. Osorio (1996). Mixed logical/linear programming, available at http://www.gsia.cmu.edu/afs/andrew/gsia/jh38/jnh.html.

Jaffar, J., and J. L. Lassez (1987). Constraint logic programming, *Proceesings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL87)*, 111-119.

Jaffar, J., S. Michaylov, P. Stuckey and R. Yap (1992). A abstract machine for CLP(R^+), *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 128-139.

McAllester, D. A. (1993). Partial order backtracking, manuscript, MIT AI Laboratory, 545 Technology Square, Cambridge, MA 02139 USA.

Quine, W. V. (1952). The problem of simplifying truth functions, *American Mathematical Monthly* **59**, 521-531.

Quine, W. V. (1952). A way to simplify truth functions, *American Mathematical Monthly* **62**, 627-631.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle, *Journal of the ACM* **12**, 23-41.

Saxe, J. (1980). Dynamic programming algorithms forrecognizing small bandwidth graphs in polynomial time, *SIAM Journal on Algebraic and Discrete Methods* **1**, 363-369.

Tsang, E. (1993). *Foundations of Constraint Satisfaction*, London: Academic Press.

Tseitin, G. S. (1968). On the complexity of derivations in the propositional calculus, in A. O. Slisenko, ed., *Structures in Constructive Mathematics and Mathematical Logic, Part II* (translated from Russian) 115-125.

Van Hentenryck, P. (1989). *Constraint Satisfaction in Logic Programming*, Cambridge, MA: MIT Press.

Van Hentenryck, P., and V. Saraswat (1996). Constraint programming, *ACM Computing Surveys*, December.

Zabih, R. (1990). Some applications of graph bandwidth toconstraint satisfaction problems, *Proceedings, National Conference on Artificial Intelligence (AAAI)*, 46-51.