ARL-CR-0787 ● Nov 2015

US Army Research Laboratory

Constraint Optimization Literature Review

prepared by Peter J Schwartz
ORSA Corporation
1003 Old Philadelphia Road, #103
Aberdeen, MD

under contract W91CRB-11D-0007

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-CR-0787 ● Nov 2015

US Army Research Laboratory

Constraint Optimization Literature Review

prepared by Peter J Schwartz
ORSA Corporation
1003 Old Philadelphia Road, #103
Aberdeen, MD

under contract W91CRB-11D-0007

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
November 2015	Final	June 2013–December 2013

4. TITLE AND SUBTITLE
Constraint Optimization Literature Review

5a. CONTRACT NUMBER
W91CRB-11D-0007

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Peter J Schwartz

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
ORSA Corporation
1003 Old Philadelphia Rd., #103
Aberdeen, MD 21001

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
ATTN: RDRL-CIH-S
Aberdeen Proving Ground, MD 21005-5067

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
ARL-CR-0787

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The Constraint Optimization Problem (COP) is a commonly used mathematical formalism that can express many real-world situations. When the COP contains discrete variables, however, the number of combinations of variable assignments to consider is exponential in the number of variables. For example, adding just 20 binary variables to a COP multiplies the number of combinations to consider by over one million. This means that even if standard hardware and software efficiency techniques can provide orders of magnitude in increased speed, they can become quickly overwhelmed as problems become larger. A variety of techniques have been developed to address the complexity of COPs. Unfortunately, the COP is nondeterministic polynomial time hard (NP-hard) in general, meaning that for any algorithm there exists a problem instance for which the runtime is exponential in the size of the problem input. This report reviews the literature on COPs.

15. SUBJECT TERMS
high-performance computing, mobile ad hoc network, optimization, constraint, satisfaction

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Peter J Schwartz
Unclassified	Unclassified	Unclassified	UU	44	19b. TELEPHONE NUMBER (Include area code) 410-306-1313

Contents

INTENTIONALLY LEFT BLANK.

1. Introduction

The Constraint Optimization Problem (COP) is a commonly used mathematical formalism that can express many real-world situations. When the COP contains discrete variables, however, the number of combinations of variable assignments to consider is exponential in the number of variables. For example, adding just 20 binary variables to a COP multiplies the number of combinations to consider by over one million. This means that even if standard hardware and software efficiency techniques can provide orders of magnitude in increased speed, they can become quickly overwhelmed as problems become larger.

A variety of techniques have been developed to address the complexity of COPs. These techniques combine search algorithms with constraint propagation to try to find good solutions quickly without testing each possible combination of variable assignments. Some approaches are systematic and guaranteed to produce an optimal solution while others give up optimality in an effort to find near-optimal solutions faster.

Unfortunately, COP is nondeterministic polynomial time hard (NP-hard) in general, meaning that for any algorithm there exists a problem instance for which the runtime is exponential in the size of the problem input. In practical terms, this means that even modestly sized problems can be too complex to solve in a reasonable amount of time even for the most sophisticated algorithms.

This report reviews the literature on COPs. Section 2 defines the Constraint Satisfaction Problem (CSP) and extends it to define several COP variants. Section 3 describes a variety of algorithms and techniques used to solve CSPs and COPs. Section 4 summarizes the findings. Someone who needs to solve a CSP or COP should find this report useful for choosing an effective representation or an efficient algorithm.

2. Constraint Optimization Problems

2.1 Constraint Satisfaction Problems

The CSP (Montanari 1974) is a very common problem representation in the artificial intelligence literature (Wallace 1996) and serves as the basis of the COP. It is useful for describing problems in which a set of decisions must be made, but because these decisions can interact with one another, not all combinations of choices are valid. The problem is to find any combination of choices that is valid or to prove that no such combination exists, and this problem is NP-complete (i.e.,

when it is both in NP and NP-hard), in general. In a CSP, each decision to be made is called a variable, each possible choice for each decision is called a value, the interactions between decisions are called constraints, and combinations of choices are called assignments. These terms are further described in the following definitions.

Definition 2-1. CSP: Given a set of variables V, a set of domains D, where each domain $D_i \in D$ is a set of possible values for a variable $v_i \in V$, and a set of constraints C over the variables of V, produce an assignment of values to variables that satisfies all of the constraints.

Definition 2-2. Constraint: Given a CSP with variables V, domains D, and constraints C, a constraint is a relation over the set of possible assignments to a subset of the variables of V. This subset of variables is called the scope of the constraint. Each possible assignment to the variables in the scope is called a tuple. A constraint is therefore a relation that defines which tuples over the scope of the constraint satisfy it and which tuples violate it. A constraint c_1 is tighter than another constraint c_2 if the set of complete assignments that satisfy c_1 is a subset of the set of complete assignments that satisfy c_2.

Definition 2-3. Assignment: Given a CSP with variables V, domains D, and constraints C, an assignment is a mapping of variables of V to values in the corresponding domains of D. A complete assignment specifies a value for every variable of V, and a partial assignment specifies values for a subset of the variables of V. A solution is a complete assignment that does not violate any of the constraints of C.

In a special case of CSP, every variable has a domain with only 2 values, TRUE and FALSE. This type of variable is known as a Boolean variable. A CSP of Boolean variables is known as a Boolean satisfiability (SAT) problem.

Definition 2-4. SAT: Given a set of variables V, each with the domain {TRUE, FALSE}, and a set of constraints C over the variables of V, produce an assignment of values to variables that satisfies all of the constraints.

The constraints of an instance of SAT are generally described as a set of Boolean clauses.

Definition 2-5. Boolean clause: A clause is a disjunction of literals. A literal is either a Boolean variable or its negation. If B is a Boolean variable, the negation of B is written $\neg B$, and $\neg B$ evaluates to the logical opposite of B (that is, if B = TRUE, then $\neg B$ = FALSE, and if B = FALSE, then $\neg B$ = TRUE). If B and C are Boolean variables, the disjunction of B and C is written $B \vee C$, and $B \vee C$ evaluates to the

logical "or" of B and C (that is, $B \vee C$ = TRUE if B = TRUE or if C = TRUE, and $B \vee C$ = FALSE if B = FALSE and C = FALSE).

Like CSP, SAT is NP-complete, in general. One attribute of all NP-complete problems is the fact that it is possible to convert an instance from any NP-complete representation to any other NP-complete representation in polynomial time. In particular, any finite-domain CSP (even those with variable domains of more than 2 values) can be transformed into an equivalent instance of SAT in polynomial time. This can be accomplished by creating one Boolean variable for each value in the domain of each variable in the original CSP. For example, suppose the original CSP contains a variable x with the domain {A, B, C}. First, create one Boolean variable for each value in the domain; call them x_A, x_B, and x_C. For each Boolean variable x_i, x_i = TRUE in the SAT transformation represents $x = i$ in the original CSP.

A set of clauses in the SAT transformation can force exactly one value to be assigned to each variable of the original CSP. In this example, this clause requires at least one of the values to be assigned: $x_A \vee x_B \vee x_C$.

It is possible to prevent more than one of the values from being assigned by adding one clause for each pair of values: $\neg x_A \vee \neg x_B$, $\neg x_A \vee \neg x_C$, and $\neg x_B \vee \neg x_C$.

Finally, it is possible to encode the original CSP constraints as clauses over the Boolean variables. Suppose that the original CSP contains a second variable y with the domain {A, B, C} and the constraint $\{x = A, y = B\}$ is an illegal assignment. This can be represented in the SAT transformation with a clause that forces any solution to avoid at least one of these variable assignments: $\neg x_A \vee \neg y_B$.

Altogether, this transformation is linear in the size of the constraints of the original CSP and quadratic in the size of its largest variable domain.

2.2 Constraint Optimization Problems

CSPs can be extended to represent not only which combinations of choices are valid, but also which combinations of choices are preferred. The problem is to find a combination of choices that not only satisfies the constraints, but also optimizes the preferences. Such problems are known collectively as COPs.

Definition 2-6. COP: Given a set of variables V, a set of domains D, where each domain $D_i \in D$ is a set of possible values for a variable $v_i \in V$, a set of constraints C over the variables of V, and an ordering $\preccurlyeq$ over assignments, produce an assignment of values to variables that satisfies all of the constraints and is optimal according to the ordering. If A and A' are assignments, $A \preccurlyeq A'$ means that A is

preferred or equivalent to *A'*. An assignment *A* is optimal if $A \preccurlyeq A'$ for all assignments *A'*.

It is common practice to define the preference ordering $\preccurlyeq$ by mapping consistent assignments to a set of objects with a known ordering.

Definition 2-7. Objective function: Given a set *S* of valid solutions to a COP with preference ordering $\preccurlyeq_P$, and given a set of objects *O* with a known ordering $\preccurlyeq_O$, an objective function $f : S \rightarrow O$ maps each solution of *S* to an object of *O* such that for any $s_1, s_2 \in S$, $f(s_1) \preccurlyeq_O f(s_2)$ if and only if $s_1 \preccurlyeq_P s_2$.

Objective functions are often defined as "scoring functions", in which each solution is mapped to a real number and the objective is to either minimize or maximize the score according to the $\leq$ or $\geq$ ordering on the real number line.

Since COP is a generalization of CSP and CSP is NP-complete, COP is generally NP-hard. The exception occurs when restrictions are imposed on the types of variables, constraints, or preferences that can be expressed. These restrictions define special classes of COP.

One of the simplest forms of COP is Max-CSP (Freuder and Wallace 1992).

Definition 2-8. Max-CSP: Given a set of variables *V*, a set of domains *D* where each domain $D_i \in D$ is a set of possible values for a variable $v_i \in V$, and a set of constraints *C* over the variables of *V*, produce an assignment of values to variables that maximizes the number of satisfied constraints.

Max-CSP expresses the basic preference of satisfying more constraints rather than fewer. In other words, Max-CSP is a special case of COP in which the partial ordering is defined such that $A \preccurlyeq A'$ if the number of constraints violated by *A* is less than or equal to the number of constraints violated by *A'*. Max-CSP is approximately (APX)-complete, which means that it is NP-hard and that it can be approximated to within a constant multiple of optimal in polynomial time, but also that there is no polynomial-time approximation scheme unless P = NP.

A Max-CSP representation can be more useful than a CSP representation when the problem is over-constrained, meaning that it is impossible to satisfy all of the constraints simultaneously. A CSP algorithm would simply state that no solution exists. A Max-CSP algorithm would produce an assignment that satisfies as many of the constraints as possible. Barták et al. (2003) propose finding a maximal consistent assignment, which is a consistent assignment of as many of the variables as possible, leaving the remaining variables unassigned; instead of minimizing the number of violated constraints, the problem here is to minimize the number of unassigned variables.

A closely related problem is the Partial Max-CSP.

Definition 2-9. Partial Max-CSP: Given a set of variables V, a set of domains D, where each domain $D_i \in D$ is a set of possible values for a variable $v_i \in V$, a set of constraints C over the variables of V, and a partitioning of C into C_H (the hard constraints) and C_S (the soft constraints), produce an assignment of values to variables that satisfies all of the hard constraints of C_H and maximizes the number of satisfied soft constraints of C_S.

In Partial Max-CSP some of the constraints are hard, meaning that they must be satisfied, and some of the constraints are soft, meaning that they do not necessarily need to be satisfied. Partial Max-CSP is therefore a generalization of both CSP and Max-CSP. CSP can be thought of as a special case of Partial Max-CSP in which all of the constraints are hard, and Max-CSP can be thought of as a special case of Partial Max-CSP in which all of the constraints are soft.

As described in Section 2.1, SAT is a special case of CSP that allows for special-purpose efficiency techniques. In the next section it will be shown that many COP solvers are based on CSP algorithms. To take advantage of the special-purpose efficiency techniques, some of these COP solvers employ a SAT encoding of an optimization problem. Max-SAT and Partial Max-SAT are special cases of Max-CSP and Partial Max-CSP, respectively, in which all of the variables are Boolean variables.

Definition 2-10. Max-SAT: Given a set of Boolean variables V and a set of clauses C over the variables of V, produce an assignment of values to variables that maximizes the number of satisfied clauses.

Definition 2-11. Partial Max-SAT: Given a set of Boolean variables V, a set of clauses C over the variables of V, and a partitioning of C into C_H (the hard clauses) and C_S (the soft clauses), produce an assignment of values to variables that satisfies all of the hard clauses of C_H and maximizes the number of satisfied soft clauses of C_S.

Partial Max-SAT is a generalization of both SAT and Max-SAT. SAT is a special case of Partial Max-SAT in which all of the clauses are hard, and Max-SAT is a special case of Partial Max-SAT in which all of the clauses are soft.

In all of these variations of Max-CSP, it is assumed that all of the soft constraints are equally important. That is, violating one constraint is as bad as violating any other constraint. This assumption does not always hold in the real world. To represent the relative importance of satisfying constraints or the relative cost of violating them, the Weighted CSP (WCSP) assigns each constraint a real-valued

weight. Similarly, Weighted SAT (WSAT) assigns a real-valued weight to each clause. Weighted Partial Max-SAT (WP Max-SAT) (Creignou et al. 2001) divides the clauses into hard and soft clauses and assigns a weight to each soft clause.

Definition 2-12. WCSP: Given a set of variables V, a set of domains D, where each domain is a set of possible values for a variable of V, a set of constraints C over the variables of V, and a weighting function from constraints to real numbers W: $C \rightarrow \mathbb{R}$, produce an assignment of values to variables that minimizes the sum of the weights of the violated constraints.

Definition 2-13. WSAT: Given a set of Boolean variables V, a set of clauses C over the variables of V, and a weighting function from clauses to real numbers W: $C \rightarrow \mathbb{R}$, produce an assignment of values to variables that minimizes the total weight of the violated clauses.

Definition 2-14. WP Max-SAT: Given a set of Boolean variables V, a set of clauses C over the variables of V, a partitioning of C into C_H (the hard clauses) and C_S (the soft clauses), and a weighting function from soft clauses to real numbers W: $C_S \rightarrow \mathbb{R}$, produce an assignment of values to variables that satisfies all of the hard clauses of C_H and minimizes the sum of the weights of the violated soft clauses.

Max-CSP is a special case of WCSP in which each constraint is assigned an equal weight. Partial Max-CSP can also be represented as a special case of WCSP in which all of the hard constraints are assigned infinite weight and all of the soft constraints are assigned the same finite weight. Similarly, WSAT and Partial Max-SAT are special cases of WP Max-SAT.

Other formalisms use real numbers to represent various aspects of COPs but interpret these values differently. Fuzzy CSPs (FCSPs) (Ruttkay 1994), for example, assign a real number to each tuple within a constraint to represent the degree to which the tuple satisfies the constraint.

Definition 2-15. FCSP: Given a set of variables V, a set of domains D, where each domain is a set of possible values for a variable of V, a set of constraints C over the variables of V, and a fuzzy relation function from tuples to real numbers F: $C \rightarrow [0, 1]$, produce an assignment of values to variables that maximizes how well the constraints are satisfied in total. The fuzzy relation function F represents how well each tuple satisfies a constraint.

CSP is a special case of FCSP in which the fuzzy relation F maps every tuple to either 1.0 (meaning that the tuple satisfies the constraint completely) or zero (meaning that the tuple fails to satisfy the constraint whatsoever).

Another representation that assigns real numbers is the Probabilistic CSP (PCSP) (Fargier and Lang 2005). The PCSP assumes that if the actual constraints of a CSP are uncertain, given a set of constraints, the actual CSP will contain some of those constraints but not others. Each constraint of a PCSP is therefore given a probability that it will need to be satisfied in the actual problem.

Definition 2-16. PCSP: Given a set of variables V, a set of domains D, where each domain is a set of possible values for a variable of V, a set of constraints C over the variables of V, and a probability function from constraints to real numbers P: $C \rightarrow [0, 1]$, produce an assignment of values to variables that minimizes the overall probability of a constraint violation. The probability function P represents the likelihood that the constraint will need to be satisfied in the actual problem.

CSP is a special case of PCSP in which the probability function P maps every constraint to either 1.0 (meaning that the constraint will certainly appear in the actual problem) or zero (meaning that the constraint will certainly not appear in the actual problem). Even though FCSPs and PCSPs have a very similar form, they have very different interpretations. Bistarelli et al. (1996) point out that an FCSP can be cast as a PCSP and vice versa.

With so many alternative interpretations and representations of COPs, it can be difficult to understand the relationships between them. To describe the theoretical relationships between many of these representations, Bistarelli et al. (1996) present the Valued CSP (VCSP), based on ordered monoids, and Semiring-based CSP (SCSP), based on c-semirings. Ordered monoids and c-semirings are abstract algebraic constructs that describe various properties that impose some notion of an ordering among the elements of a set. When applied to a CSP, these properties define how to combine constraints and compare assignments against one another.

Definition 2-17. VCSP: This is a CSP in which each constraint is given a value, an ordering operator defines a total ordering over the values, and a combination operator describes how values on constraints must be combined. The values and operators of a VCSP must follow the structure of an ordered monoid. The value of a constraint corresponds to its importance; informally, it is the cost of violating the constraint. The set of possible values are not necessarily numbers on a number line, so the ordering operator defines which values are larger and smaller. An assignment of values to variables (whether partial or complete) might violate a set of constraints, so the combination operator defines how the values of those violated constraints can be combined into a value for the assignment.

Special cases of VCSP include WCSP, FCSP, and PCSP. The authors also define new type of COP that they call the Lexicographic CSP, which combines aspects of WCSP and PCSP.

Definition 2-18. SCSP: This is a CSP in which each tuple is given a value, an ordering operator defines a partial ordering over the values, and a combination operator describes how values on tuples must be combined. The values and operators of an SCSP must follow the structure of a c-semiring. The value of a tuple corresponds to its importance; informally, it is the amount gained by including the tuple in an assignment as a subset of the assignment's variable bindings. The set of possible values are not necessarily numbers on a number line, so the ordering operator defines which values are larger and smaller. An assignment of values to variables (whether partial or complete) might contain several subtuples within it, so the combination operator defines how the values of those subtuples can be combined into a value for the assignment.

As with VCSP, special cases of SCSP include WCSP, FCSP, and PCSP. The authors also define new type of COP that they call the Egalitarianism and Utilitarianism CSP, which combines aspects of WCSP and FCSP.

The VCSP and SCSP are closely related but not equivalent. Bistarelli et al. (1996) prove that VCSP and SCSP are only equivalent if the ordering operators of both impose a total ordering.

Another complication of optimization problems is introduced when multiple objectives are competing against one another. Such a problem can be expressed as a Multiobjective Optimization Problem (MOP) (Sawaragi et al. 1985).

Definition 2-19. MOP (also called Multi-Criteria or Multi-Attribute Optimization Problem): Given a set of variables *V*, a set of domains *D*, where each domain is a set of possible values for a variable of *V*, a set of constraints *C* over the variables of *V*, and a set of objective functions *O*, produce an assignment of values to variables that simultaneously optimizes each of the objective functions.

The major hurdle to overcome when dealing with MOPs is the fact that changing an assignment to make it more optimal according to one objective function often makes it less optimal according to other objective functions. One approach to this issue is to combine all of the objective functions into a single aggregate objective function (AOF). A common AOF assigns a weight to each individual objective function and optimizes the weighted sum. Each weight expresses the relative importance of the objective function, similar to the way weights within a WCSP express the relative importance of constraints. The difficulty with the AOF approach lies in identifying a meaningful way to combine the individual objective

functions. After the objective functions are combined, the problem can be expressed using one of the other COP formalisms, such as VCSP or SCSP, and solved using standard techniques.

An alternative approach is to produce an assignment that does not optimize for all objective functions but is instead Pareto optimal (Pareto 1906).

Definition 2-20. Pareto optimal (also called Pareto efficient): An assignment is Pareto optimal relative to a set of objective functions *O* if it is impossible to change the assignment in a way that increases its optimality according to one objective function without simultaneously decreasing its optimality according to another objective function. The Pareto frontier is the set of assignments that are Pareto optimal.

By seeking a solution that is Pareto optimal, it is not necessary to devise a scheme that combines all of the objective functions, as with AOF. The issue with Pareto optimality, however, is that the set of assignments that are Pareto optimal can often be quite large. In practice, preferences often exist between assignments that are Pareto optimal but are not captured in the problem representation.

3. Constraint Optimization Algorithms

3.1 Constraint Satisfaction Algorithms

3.1.1 Brute-Force search

Several methods exist for solving an instance of CSP. The most straightforward method is a brute-force search, which simply tests every possible complete assignment of values to variables until either an assignment is found that satisfies all of the constraints or the set of all possibilities is exhausted. For finite-domain CSPs, brute-force search is easy to implement, sound (meaning that the result is guaranteed to be correct), complete (meaning that it is guaranteed to eventually terminate and give a result), and systematic (meaning that it will not test the same assignment more than once).

In practice, however, brute-force search can be very inefficient. The number of complete assignments is exponential in the number of variables, and if a CSP has no solution, then brute-force search must test every single complete assignment before terminating. Brute-force search should generally only be considered when the number of variables is very low or when the number of solutions is very high.

3.1.2 Constraint Propagation

A second approach to solving a CSP is constraint propagation, the process of inferring additional constraints based on the set of constraints given in the problem description. The constraints that are given in the problem description are known as explicit constraints; implicit constraints are the constraints that are implied by the explicit constraints but not listed explicitly. Constraint propagation is therefore the process of deducing the implicit constraints based on the explicit constraints.

If a CSP contains n variables, it is possible to identify 2^n unique subsets of variables. A constraint (whether explicit or implicit) exists over every subset of variables, so it is possible to define an exponential number of constraints. Of these constraints, n are unary constraints, each of which is defined over a single variable. One of these constraints is the global constraint, which is defined over the complete set of variables.

Many forms of propagation exist, but all of them operate by iteratively tightening the set of constraints. A constraint is tightened by removing a tuple from the set of tuples that satisfy the constraint. A tuple can safely be removed from the satisfying set if it can be proved that the tuple does not have support.

Definition 3-1. Support: A tuple t_1 of constraint c_1 is supported by a tuple t_2 of constraint c_2 if either the scope of c_2 is a subset of the scope of c_1 and the individual variable-value pairs of t_2 are a subset of the variable-value pairs of t_1, or if the scope of c_2 is a superset of the scope of c_1 and the individual variable-value pairs of t_2 are a superset of the variable-value pairs of t_1.

Support is reciprocal: Tuple t_1 supports tuple t_2 if and only if t_2 supports t_1.

A CSP constraint propagation algorithm repeatedly removes tuples from constraints if they have no support. Many propagation algorithms are both sound and complete. If the propagation process is sound and complete, when it is unable to remove any more tuples, it has proved that at least one solution exists, and when it has removed all of the tuples from any constraint, it has proved that no solution exists. The remainder of this section presents constraint propagation techniques for finite-domain CSPs. Constraint propagation techniques also exist for CSPs with continuous variable domains.

The most basic form of constraint propagation is node consistency (Mackworth 1977), which simply ensures that all of the unary constraints are satisfied. In practice, the constraints of a CSP might be imposed for different reasons or might come from different sources. Node consistency boils down to the process of collecting the unary constraints from different sources into a single representation to ensure that there exists at least one value that might be assigned to each variable.

With n variables and a maximum domain size of k, node consistency can be enforced in $O(nk)$ time.

Arc consistency (AC) (Mackworth 1977) extends this idea to pairs of variables by removing any tuple from a binary constraint (constraints over 2 variables) that does not have support from either of the 2 unary constraints that could support it. AC-1 (Mackworth 1977), the basic algorithm for enforcing AC, requires $O(n^3k^3)$ time, where n is the number of variables and k is the maximum domain size. AC-3 (Mackworth 1977) uses a queue to reduce this to $O(n^2k^3)$ time. AC-4 (Mohr and Henderson 1986) uses a complex counting scheme and is a provably optimal algorithm in the worst case, requiring $O(n^2k^2)$ time, but AC-3 is often faster in practice.

Path consistency (Mackworth 1977) continues in this manner by considering sets of 3 variables. Assume there are 3 variables, x_1, x_2, and x_3, and 3 binary constraints between them with scopes $\{x_1, x_2\}$, $\{x_2, x_3\}$, and $\{x_1, x_3\}$. Assume that the partial assignment $\{x_1 = v_1, x_2 = v_2\}$ satisfies the constraint over $\{x_1, x_2\}$. If there is no value v_3 in the domain of x_3 such that the tuples $\{x_1 = v_1, x_3 = v_3\}$ and $\{x_2 = v_2, x_3 = v_3\}$ both satisfy their respective constraints, it is impossible to extend the partial assignment $\{x_1 = v_1, x_2 = v_2\}$, so path consistency will remove the tuple $\{x_1 = v_1, x_2 = v_2\}$ from the constraint over $\{x_1, x_2\}$. In other words, path consistency removes any tuple from a binary constraint that would not have support from a constraint over a superset of 3 variables. With n variables and a maximum domain size of k, path consistency can be enforced in $O(n^5k^5)$ time using the basic algorithm (Dechter 2003). Analogously to AC, it is possible to use a queue like AC-3 to speed up this process to $O(n^3k^5)$ time (Dechter 2003) or to use a counting scheme like AC-4 to speed it up to $O(n^3k^3)$ time (Mohr and Henderson 1986). The counting-based algorithm for path consistency is provably optimal in the worst case but the queue-based algorithm is often faster in practice.

Path consistency can be generalized to include more variables in i-consistency (also called k-consistency) (Freuder 1978). Path consistency ensures that a consistent assignment of 2 variables can be extended to a consistent assignment of 3 variables; i-consistency ensures that a consistent assignment of i-1 variables can be extended to a consistent assignment of i variables. If not, i-consistency removes the tuple that represents the partial assignment of i-1 variables from its constraint. In other words, i-consistency removes any tuple from a constraint c_1, where c_1 is defined over i-1 variables, that does not have support from a constraint c_2, where the scope of c_2 extends the scope of c_1 to an ith variable, and support from all constraints c_3, where the scope of c_3 is a subset of c_2 and contains the ith variable that is not in the scope of c_1. If the problem has n variables and a maximum domain size of k, i-consistency

can be enforced using a basic algorithm in $O((nk)^{2i}2^i)$ time (Dechter 2003). The provably optimal algorithm for *i*-consistency uses a counting scheme like AC-4 and runs in $O((nk)^i)$ time in the worst case (Dechter 2003).

Although very similar, 3-consistency (that is, *i*-consistency where $i = 3$) is not identical to path consistency in all cases. These 2 forms of consistency are only identical when there are no tertiary constraints (constraints over 3 variables), as in a binary CSP, which only contains unary and binary constraints. If the CSP contains any tertiary constraints, then a 3-consistency algorithm might remove tuples from a binary constraint because of an inconsistency in the tertiary constraint, but a path consistency algorithm would not. This means that 3-consistency is stronger than path consistency when the CSP contains tertiary constraints.

The concept of *i*-consistency can be extended to global consistency (Freuder 1982), which can determine whether or not a CSP is satisfiable. A set of constraints is strong *i*-consistent if it is *j*-consistent for all $j \leq i$. A set of constraints is globally consistent if it is strong *i*-consistent and *i* is greater than the size of the scope of the largest explicit constraint. For example, because 3-consistency and path consistency are equivalent in binary CSPs, a binary CSP that is path consistent is also globally consistent. Once the constraints of a CSP have been tightened to the point of global consistency, it is possible to extract a solution from it in polynomial time (Dechter 1992). Even an optimal global consistency algorithm, however, requires an exponential amount of time in general (Cooper 1990).

Although global consistency is a sound and complete constraint propagation technique, several other inference mechanisms have also been developed. For example, generalized AC (Mohr and Masini 1988) is the inverse of *i*-consistency; it removes any individual variable binding if it does not have support in a constraint over *i* variables. As another example, a set of *m* constraints is relational (*i*,*m*)-consistent (Dechter and van Beek 1995) if every consistent assignment to a subset of *i* of their variables can be extended to an assignment to all of their variables that satisfies all *m* constraints. If *m* is the maximum domain size of any variable in a given CSP, the consistency of the CSP can be tested by imposing strong relational (1,*m*) consistency.

For any of these given forms of consistency, it is possible to define a form of directional consistency (Dechter and Pearl 1987). Directional consistency is useful in algorithms that assign values to variables in sequence, such as depth-first search (see Section 3.1.3). Given an ordering over the variables of a CSP, and given a consistent partial assignment over the first *k* variables in this ordering, directional consistency only enforces the form of consistency on the constraints with one or more unassigned variables in their scopes. Directional consistency can improve

efficiency by ignoring constraints over the variables that are already assigned values.

Generally speaking, solving a CSP through constraint propagation alone can be very inefficient. The number of constraints (whether explicit or implicit) is exponential in the number of variables in the CSP, and the number of tuples in a constraint can be exponential in the number of variables in its scope. This means that constraint propagation can potentially be even less efficient than brute-force search. Many special classes of CSP have been identified for which special-purpose constraint propagation techniques can be applied very effectively.

Unfortunately, many instances of CSP do not fall into any of these special classes. Constraint propagation is most frequently used to improve the efficiency of the methods described in the next few sections. For a survey of constraint propagation techniques, see Bessière (2006).

3.1.3 Depth-First Search

Depth-first search (also called backtracking search) is a trial-and-error process by which variables are assigned values one at a time. Given a finite-domain CSP, the basic depth-first search algorithm operates by choosing an unassigned variable, assigning a value to it, and checking the constraints for any inconsistencies. If the current partial assignment violates any constraints, the algorithm backtracks by unassigning the most recent variable binding and reassigning the variable to a different value. If all of the constraints are satisfied, then another variable is chosen for assignment, and the process is repeated until either all of the variables have been assigned and a solution is produced, or all of the values of the first variable have been tested and failed, proving that no solution exists.

Depth-first search organizes the set of partial assignments of a CSP into a tree structure. Each vertex of this search tree represents a partial assignment, and each branch represents the binding of one more value to a variable. The root of this search tree represents the empty partial assignment, and each vertex below it represents a partial assignment in which the number of variables that have been assigned is equal to the length of the path from that vertex back to the root. The leaves of this search tree represent complete assignments, and some of these leaves represent solutions.

Like brute-force search, depth-first search is sound, complete, and systematic for finite-domain CSPs. In fact, brute-force search could be described as a search technique that bypasses the upper levels of the depth-first search tree and scans through each of the leaves directly. Depth-first search makes use of the rest of the search tree to help organize this process.

Like any search algorithm, depth-first search must deal with a fundamental tradeoff. The term "search" is employed to convey the fact that these algorithms must guess where it should go to find a solution. Good guesses reduce the amount of searching necessary to find a solution but more computation time is generally required to make a good guess. The fundamental tradeoff of search is to balance the processing time per guess against the total number of guesses that must be made to minimize the overall search time. Each of the efficiency techniques described in the following presents an option for dealing with this tradeoff.

As a depth-first search algorithm progresses, it is necessary to decide which variable should be assigned a value next. A variable ordering heuristic is any rule or method for making this decision. The most widely used variable ordering heuristic is the most-constrained variable heuristic (also called the fail first heuristic) (Haralick and Elliott 1980). This heuristic chooses the variable with the fewest values to be tested remaining its domain. If several variables are tied by this measure, the variable that appears in the most constraints with unassigned variables is chosen. If the current partial assignment is a dead end (meaning it cannot be extended to a solution), it will be necessary to test all of the values for at least one variable, so choosing the variable with the fewest values will help discover this fact with the least computation. If the current partial assignment is not a dead end, it will be necessary to assign a value to every variable, so choosing the variable with the tightest constraints will remove the most values from the domains of other unassigned variables through constraint propagation (discussed in the following). Smith and Grant (1998) have shown that the most-constrained variable heuristic can lead to poor performance in some situations, so Beck et al. (2003) have introduced the notion of promise to describe variable heuristics that attempt to maximize the likelihood of success. For a comparison of variable ordering heuristics see Gent et al. (1996).

After choosing a variable, it is necessary to choose a value to assign to that variable. The most-common value-ordering heuristic is the least-constraining value heuristic (Geelen 1992). This heuristic chooses the value that will remove the fewest tuples from the constraints of unassigned variables. If the current partial assignment is a dead end, it will be necessary to test all of the values to prove this, so the value ordering will have no effect. If the current partial assignment is not a dead end, at least one value must lead to a solution, so choosing the value that is least likely to cause a constraint violation reduces the chance that it will be necessary to backtrack and test the other values later. Because the value ordering only affects efficiency when the current partial assignment is not a dead end, it generally has less of an impact on efficiency than the variable ordering. For more recent work on value ordering heuristics see Lecoutre et al. (2007).

Once the chosen value has been assigned to the chosen variable, it is possible to apply a form of directional consistency (see Section 3.1.2) to the resulting constraint network to propagate the effects of that variable assignment. This often results in the removal of values from the domains of unassigned variables that would lead to constraint violations. Any value that is removed through constraint propagation does not need to be tested during search, reducing the size of the search tree. A constraint propagation technique can impose any type of consistency—stronger forms of consistency require more processing time but can also prune more values. If the form of consistency is strong enough, it is sometimes possible to remove all of the values from the domain of an unassigned variable, proving that the current partial assignment cannot be extended to a solution. The most common forms of constraint propagation are forward checking (Haralick and Elliott 1980; Dent and Mercer 1994) and maintaining arc consistency (Sabin and Freuder 1994). See Section 7 of Bessière (2006) for a survey of constraint propagation techniques in depth-first search.

Even the best variable and value ordering heuristics can be fooled into making bad decisions for some problem instances. Random restarts can help to avoid this situation by imposing a limit on the number of times the algorithm can backtrack. When this limit is reached, the algorithm restarts and the limit is increased. If the heuristics rely on random choices (to break ties, for example) or information that was accumulated during the previous search attempt, a new sequence of variables and values will be chosen. If the sequence is effective for the problem instance, it will lead to a solution before the backtracking limit is reached. If not, the search will be restarted, and the new sequence will hopefully be more effective. See Gomes et al. (2000) for a discussion of random restarts.

When a constraint is violated or a dead end is identified, the basic depth-first search algorithm backtracks over the most recent variable assignment to test a different value. This form of backtracking is called chronological backtracking. Other forms of backtracking are able to safely jump back over several variables without sacrificing completeness. Graph-based backjumping (Dechter 1989) backtracks to the last variable that shares a constraint with the conflict variable, skipping over any variables that are unrelated. Conflict-directed backjumping (Prosser 1993) keeps track of the variables that interfere with each other and is able to backtrack all the way to the most recent variable that was definitely involved in the conflict. Dynamic backtracking (Ginsberg and McAllester 1993) keeps track of even more information, allowing it to backtrack as far as conflict-directed backjumping and then reassign any variables that were unassigned during backtracking but were not involved in the conflict. See Kondrak and van Beek (1997) for a comparison of backtracking techniques.

It is not uncommon for a depth-first search algorithm to encounter a conflict, backtrack to test a different combination, and then encounter the exact same conflict later on during search. In some instances, a considerable amount of search is required to discover a conflict, so rediscovering the same conflict repeatedly can waste a significant amount of time. Nogood recording (Schiex and Verfaillie 1994) helps to avoid this waste of time by recording and conflicts as nogoods.

Definition 3-2. Nogood: This is a partial assignment that cannot be extended to a solution. In other words, a nogood is a tuple that violates either an explicit or implicit constraint.

In practice, it is generally necessary to record each nogood along with a justification.

Definition 3-3. Justification: Given a nogood assignment *A*, some subset *C'* of the explicit constraints *C* makes it impossible to extend *A* to a solution. The justification of nogood *A* is the subset of variables over which the constraints of *C'* are defined.

Each time a depth-first search algorithm backtracks, the set of variable assignments involved in the conflict is recorded as a nogood. As search continues, each new assignment is compared with the set of nogoods. If a nogood is found that matches the current partial assignment, the algorithm backtracks immediately without continuing to search that branch of the tree. In other words, nogood recording is the process of identifying implicit constraints and listing them explicitly. In practice, the number of nogoods identified can quickly exceed the space available to store them, so the largest ones are discarded because they require the most storage space and match the fewest partial assignments.

Depth-first search is often the best approach for problem domains in which it is unclear whether or not a solution exists. The algorithm is sound and complete, so it is guaranteed to prove whether or not the problem has a solution. Additionally, the variety of techniques available to enhance the algorithm can make it much more efficient than either brute-force search or constraint propagation. Unfortunately, it is generally difficult to identify the combination of techniques that will lead to the best efficiency of depth-first search for any specific problem domain. Practitioners often implement several different efficiency techniques and compare the run times of each combination over a representative set of test problems. For example, Bessière and Régin (1996) found that maintaining AC combined with conflict-directed backjumping does not significantly outperform maintaining AC alone. On the other hand, Narendra et al. (2000) found that maintaining AC can be improved when combined with dynamic backtracking, and Chen and van Beek (2001) found

that conflict-directed backjumping can be improved when combined with generalized AC.

As with the general form of a finite-domain CSP, an instance of SAT can be solved with depth-first search and the efficiency techniques described in the previous section. The special form of the SAT representation, however, allows for several other techniques that can improve performance even further. For example, after assigning a value to a variable, Boolean propagation operates by removing all of the clauses that contain a literal that is equivalent to the assignment (because these clauses are satisfied) and all of the literals that are the negation of this assignment from any clauses that contain them (because these literals cannot satisfy their respective clauses). If all of the literals have been removed from a clause, it is impossible to satisfy the clause given the current partial assignment, so the algorithm must backtrack. Unit propagation (Zhang and Stickel 1996) improves the efficiency of Boolean propagation by checking the number of literals that remain in each clause every time a literal is removed. If only one literal remains, the assignment that matches this literal is the only way to satisfy the clause, so unit propagation makes this assignment and adds it to a queue to be processed before the search continues. The Davis-Putnam-Logemann-Loveland (DPLL, Davis et al. 1962) algorithm applied unit propagation to great effect and serves as the basis of many modern SAT solvers.

Recent advances in SAT have increased efficiency significantly. Notably, a SAT solver called Chaff (Moskewicz et al. 2001), an extension of DPLL, introduced 2 new techniques. First, the 2 watched literals scheme speeds up unit propagation by identifying unit clauses very quickly. Second, the Variable State Independent Decaying Sums variable-ordering heuristic employs a clever counting scheme to select variables that appear in the most nogood clauses (this is another example of the most-constrained variable heuristic). Een and Sörensson (2003) combined state-of-the-art techniques such as these into a single, compact, highly efficient, and freely available SAT solver called MiniSAT. Even though MiniSAT is implemented in only 600 lines of C++, it is competitive with any of the most efficient SAT solvers, many of which are tens of thousands of lines of code. Because of the efficiency improvements provided by such systems, it is not uncommon to solve a finite-domain CSP by converting the problem to an instance of SAT, solving it using these specialized techniques, and then converting the solution back to the original CSP representation. See Dixon et al. (2004) for a survey of SAT techniques.

3.1.4 Local Search

Each of the aforementioned algorithms solves the CSP through some form of systematic search. Local search departs from this approach, opting instead for a nonsystematic search through the space of complete assignments. A local search algorithm begins by generating one or more complete (but possibly inconsistent) assignments. Each assignment is compared with constraints, and if any constraints are violated the assignment is modified. This process repeats until a solution is found, the available time is exceeded, or a prespecified number of assignments have been tested.

As a local search algorithm progresses, it is possible that it will consider the same complete assignment more than once, which means that local search is not systematic. If a solution exists, but a local search algorithm cannot find it before the limit on time or the number of assignments tested is exceeded, the algorithm will not produce the correct result, which means that (when such limits are imposed) local search is not sound. Alternatively, if no limits are imposed, a local search algorithm could search the same subset of complete assignments repeatedly without ever terminating, so (without any limits imposed) local search is not complete. Because a local search algorithm cannot be both sound and complete, it is impossible for local search to prove that no solution exists for a given instance of CSP. The effectiveness of local search therefore increases with the number solutions to the given problem.

It is interesting to compare the characteristics of local search to those of brute-force search and depth-first search. Local search and brute-force search both explore the space of complete assignments while depth-first search explores the space of partial assignments. Depth-first search and local search must both make heuristic guesses to decide where to explore next while brute-force search does not. Both brute-force and depth-first search are sound, complete, and systematic while local search is not systematic and either not sound or not complete for general CSPs.

The central idea behind local search is to attempt to "repair" an assignment by changing the value of one or more variables. By repeatedly repairing more and more of the constraint violations, the algorithm might eventually be able to find a complete assignment that does not violate any constraints. Each local-search algorithm must therefore define the set of possible changes that could be made to an assignment along with a method of deciding which change to make. Depth-first search imposes tight restrictions on the set of partial assignments to consider next, retaining soundness and completeness at the possible expense of efficiency. Local search does not impose such tight restrictions, so soundness or completeness is sacrificed in the hopes of improving efficiency.

Like depth-first search, local search must consider the same fundamental tradeoff of any search algorithm: Making a better guess as to where to search next requires more time per decision but possibly reduces the number of decisions that must be made. To discuss this tradeoff from a theoretical perspective, it is helpful to describe the search space of a local-search algorithm as a graph. In this local-search graph, each vertex represents a complete assignment and each edge represents a change from one assignment to another. Given a complete assignment that is represented by a vertex *v*, the neighborhood of that assignment is the set of complete assignments that are represented by vertices that are adjacent to *v*. In this graph, a local-search algorithm moves from one vertex to another until it finds one that represents a solution.

Once the neighborhood is known, the local-search algorithm must choose which of its neighbors to move to next. The problem lies in figuring out which neighbor will bring the search one step closer to a solution that might be many steps away. Local-search algorithms generally favor neighbors that improve the complete assignment the most in hopes that these are the most likely to lead the search toward the closest solution. This is accomplished by defining an evaluation function that ranks each assignment. Maximizing evaluation functions assign a score to each assignment that reflects how closely the assignment matches a solution, so a higher value is better. Minimizing evaluation functions assign a distance to each assignment that reflects the amount of difference between the assignment and a solution, so a lower value is better. Whether the evaluation function is a maximizing or minimizing function, the same set of local-search techniques can be applied.

Based on the local-search graph and the evaluation function, search algorithms are often discussed in terms of a landscape metaphor, in which each complete assignment lies on the surface of the land. The graph of neighbors defines where each assignment is located relative to the others, similar to coordinates of latitude and longitude. Carrying this metaphor forward, the evaluation function then defines the altitude of each assignment. For example, if a minimizing evaluation function is used, assignments that violate many constraints would be found at high altitudes, assignments that violate fewer constraints would be found at lower altitudes, and solutions would be found at the lowest altitude possible.

The landscape metaphor leads to terminology that is found throughout the literature on local search. A local optimum is a region of the search space that is surrounded by assignments with values that are worse according to the evaluation function. A local optimum can be problematic because it can cause a local-search algorithm to repeatedly test the same few assignments without moving away to find a solution. A plateau is a region with little variance between nearby assignments according to

the evaluation function. When a local-search algorithm encounters a plateau, it must decide which direction to move based on sparse information, which can lead to aimless wandering over the search space.

As will be discussed in the following, the landscape metaphor carries over to descriptions of how different local-search algorithms operate with terms such as hill climbing and random walk. Local-search algorithms can generally be grouped into 3 categories: greedy, random, and genetic.

A greedy local-search algorithm (Selman and Kautz 1993) attempts to exploit local information to the fullest by always choosing a neighbor with the best evaluation. The reasoning behind this approach is that all of the constraint violations must be repaired to find a solution, so the fastest way to find a solution is to repair as many conflicts as possible at each step. Greedy local search is also known as hill-climbing when a maximizing evaluation function is used and as steepest descent when a minimizing evaluation function is used. The Min-Conflicts heuristic (Minton et al. 1992) is a well-known type of greedy local search that has been successfully applied to a variety of domains. The Min-Conflicts heuristic randomly chooses a variable that is involved in a conflict and then reassigns that variable to the value that minimizes the number of constraint violations. In other words, Min-Conflicts restricts its set of neighbors to those of a randomly chosen variable and then chooses a neighbor greedily from that set. GSAT (Selman et al. 1992, Selman and Kautz 1993) is a greedy search algorithm that solves an instance of SAT by flipping the value of the variable that will increase the number of satisfied clauses the most.

A GSAT can have difficulty when it encounters various features of the search space landscape. Plateau search (Hampson and Kibler 1993) allows the algorithm to move to assignments of equal valuation when none of the neighbors has a worse evaluation. This allows the algorithm to cross a plateau instead of stopping. Several techniques exist to escape local optima. Random restarts can be applied to start the search from a randomly chosen new assignment, similar to their use in depth-first search (Schöning 1999). Tabu search (Glover and Laguna 1993) keeps a list of recent variable-value bindings and disallows them to prevent backwards moves to assignments that have already been visited. Constraint weighting (Thronton 2000; Selman and Kautz 1993) calculates the distance of each assignment as a weighted sum of constraint violations and increases the weight of each constraint each time it is violated. If the same constraint is violated repeatedly, its weight will increase and the algorithm will move to an assignment that satisfies it.

A random local-search algorithm employs random processes to increase the likelihood that it will find a solution. These algorithms are also called random-walk algorithms because they mimic a person walking randomly over the search space

landscape. The basic form of a random local-search algorithm is called ε-greedy, in which a neighbor is chosen greedily with probability ε and randomly with probability 1 – ε. When ε = 1, ε-greedy is equivalent to greedy search, and when ε = 0, ε-greedy is completely random. By choosing ε somewhere between 0 and 1, the algorithm can take advantage of greedy moves to guide the search toward a solution and random moves to help keep it out of local optima. A variant of ε-greedy called simulated annealing (Kirkpatrick et al. 1983) slowly increases ε to eventually converge on a solution. Hao and Pannier (1998) present a comparison of simulated annealing and tabu search in which they find that tabu search is superior on a set of randomly generated problems.

Random local-search algorithms can also be applied to solve instances of SAT. For example, WalkSAT (Selman et al. 1993) is a combination of ε-greedy and GSAT. WalkSAT applies the logic that a solution must satisfy all of the clauses but each clause only needs to be satisfied by a single literal. To choose the variable whose value will be flipped, WalkSAT begins by considering the set of variables that appear in unsatisfied clauses. WalkSAT then attempts to retain as many satisfied clauses as possible by restricting the set of possible variables to those that, when flipped, will not cause any new clauses to become unsatisfied. If more than one variable matches these criteria, WalkSAT greedily chooses the variable from this set that will repair the largest number of unsatisfied clauses. If no variable matches these criteria, then, with probability ε, WalkSAT greedily chooses the variable that will increase the number of satisfied clauses the most (like GSAT), and with probability 1 – ε, WalkSAT chooses randomly.

A genetic algorithm attempts to mimic biological processes to cause a solution to evolve out of a population. A genetic algorithm begins by generating an initial population. When solving a CSP, each individual of the population represents a complete assignment. Each individual is evaluated according to a fitness function, which is simply another name for an evaluation function, and given a fitness score. Based on these fitness scores, a subset of individuals is chosen to survive and reproduce the next generation through the process of selection; Blickle and Thiele (1996) compare several selection schemes. Over time, the traits of the most successful individuals are more likely to be passed on and combined in future generations, eventually leading to a solution.

To reproduce, 2 individuals are chosen randomly to be parents and the assignments they represent are copied. Next, a random crossover point is chosen to partition both of these copied assignments. For example, if the CSP contains 10 variables and the random crossover point is 3, each complete assignment will be split into 2 partial assignments, one containing the first 3 variable assignments and the other

containing the last 7 variable assignments. Two offspring individuals can then be produced through the process of recombination, which combines the first partial assignment copied from one parent with the second partial assignment copied from the second parent. After recombination, random mutation randomly reassigns a subset of the variables of each offspring.

Like other forms of local search, genetic algorithms must incorporate methods to escape regions that are locally optimal. As a genetic algorithm progresses, the fittest individuals survive and reproduce. Over time, the traits of a small number of exceptionally fit individuals can be spread throughout the population. As this happens, the population becomes more homogeneous and the algorithm can get trapped at a local optimum. Random mutation can only do so much to avoid this situation, so other methods have been developed. Random immigrants (Tinós and Yang 2007) are new random assignments that can be incorporated into the population to increase genetic diversity. Co-evolving species (Morrison 1998) must compete with each other but cannot reproduce with each other. Vrajitoru (2002) divides the population into genders, preventing each individual from reproducing with the half of the population and slowing the spread of genetic traits. See Michalewicz (1995) or Deb (1997) for a survey of genetic algorithms.

Local search is most effective either when a large portion of the complete assignments are solutions, offering many opportunities for the search process to encounter one, or when local features of the search space give an accurate indication of the correct direction to move to find a solution. Local search can also be successfully combined with other techniques to solve CSPs. For example, Jussien and Lhomme (2002) apply constraint propagation to local search to improve efficiency. Chatzikokolakis et al. (2004) describe an algorithm that begins with depth-first search, but when a conflict is encountered, it repairs the conflict with local search instead of backtracking. The main weakness of local search is its incompleteness; if a solution exists, there is no guarantee of finding it, and if no solution exists, there is no way to prove it. The main strength of local search is its application to constraint optimization, as discussed in the next section.

3.2 Constraint Optimization Algorithms

Before jumping into constraint optimization techniques, it is worth discussing a metric commonly used to compare COP algorithms: anytime performance. An anytime algorithm can provide a response at any point in time and are useful when the time at which a solution will be requested is unknown. Many COP algorithms can act as anytime algorithms because they can produce a suboptimal solution

extremely quickly and then improve on it as time permits. The anytime performance of an anytime algorithm is the way in which solution quality improves over time.

Many of the algorithms and techniques used for constraint optimization are variants of those used for constraint satisfaction. Here these techniques are categorized as constraint propagation, depth-first search, and local search.

3.2.1 Constraint Propagation

Some of the earliest use of constraint propagation in optimization was by Rosenfeld et al. (1976), who used a relaxed form of AC when solving FCSP. The form of AC that they used is relaxed because constraint propagation is generally harder for optimization problems than for satisfaction problems. Schiex et al. (1995) prove that AC is NP-complete for Max-CSP and that AC works for VCSP only when the combination operator is idempotent. A relation (or operator) $\otimes$ is idempotent if, given a set S and any element $e \in S$, $e \otimes e = e$. Bistarelli et al. (1996) provide algorithms for *i*-consistency of VCSP and SCSP but these assume that combination is idempotent. Schiex (2000) defines a form of AC for VCSP that allows for nonidempotent combination operators, terminates, and produces equivalent problems; this method only gives up the uniqueness property of AC. Because constraint propagation is so difficult when applied to optimization, it is rarely considered an option for solving a COP alone. Instead, constraint propagation is usually applied to enhance the efficiency of optimization algorithms based on search.

3.2.2 Depth-First Search

Depth-first search is one of the common approaches to solving optimization problems. The application of depth-first search to COP is very similar to its application to CSP. Consider an instance of COP that uses the partial ordering $\preccurlyeq_O$. Ignoring the soft constraints, a depth-first search algorithm can solve the hard constraints as an instance of CSP. Let A be an assignment identified by this depth-first search that satisfies all of the hard constraints. A is recorded as the best solution found so far. At this point, there is no guarantee that A is an optimal solution, so the search continues by backtracking from A and building on the same search tree. Let A' be the next solution found that satisfies all of the hard constraints. If $A' \preccurlyeq_O A$, meaning that A' is preferred to A, then A' replaces A as the best solution found so far. The process continues until the search space is exhausted. Once the search space is exhausted, the best solution found is guaranteed to be optimal.

The efficiency of depth-first search can be improved using several techniques. One such technique is branch-and-bound. Branch-and-bound (e.g., Black 2005) is a

process of pruning branches from the depth-first search tree based on estimated upper and lower bounds around the preference value of the optimal solution. As depth-first search progresses, branch-and-bound uses a heuristic to estimate the best possible preference value that can be achieved by extending the current partial assignment. An admissible heuristic is one that is guaranteed never to provide an estimated preference value that is worse than what is really possible.

For example, when solving an instance of Partial Max-CSP, a partial assignment *A* might have already satisfied 4 soft constraints out of 10 but violated one. Assume that, in reality, it is possible to extend *A* to a complete assignment that satisfies 3 more soft constraints, so this solution would satisfy a total of 7 soft constraints and violate 3. Since it is possible to extend *A* and only violate 3 soft constraints, an admissible heuristic cannot return a preference value greater than 3. A simple admissible heuristic might assume that it is possible to extend *A* to a complete assignment without violating any additional soft constraints, so it would estimate a preference value of 1 (because 1 soft constraint is already violated). A more-sophisticated heuristic could perform additional reasoning such as constraint propagation and might estimate a preference value of 2.

When minimizing the preference value (as in the previous example), an admissible heuristic produces a lower bound on a partial assignment. A lower bound of a partial assignment *A* is a preference value *lb*(*A*) such that the preference value of every complete extension of *A* is greater than (meaning less preferred) or equal to *lb*(*A*). It is also possible to estimate an upper bound on the preference value of an optimal solution. An upper bound is a preference value *ub* such that the preference value of an optimal solution is less than or equal to *ub*. As with lower bounds, heuristics exist for estimating the upper bound of a problem, and an admissible heuristic is guaranteed to always err on the side of safety. One basic admissible upper-bound heuristic is to set the upper bound equal to the preference value of the best solution found so far.

The branch-and-bound algorithm uses upper and lower bounds to speed up depth-first search optimization as follows. First, the upper-bound heuristic estimates an upper bound for the problem. Depth-first search then begins by assigning a value to a variable and creating a partial assignment. The lower-bound heuristic then estimates a lower bound for the partial assignment. The combination of the upper- and lower-bound estimates gives the branch-and-bound algorithm its power: If the lower bound of a partial assignment is greater than the upper bound of the problem, it is impossible to extend that partial assignment to an optimal solution, so depth-first search can backtrack immediately. As better solutions are found, the upper

bound can continue to decrease, allowing for earlier pruning and improved efficiency as the search progresses.

For example, assume the branch-and-bound algorithm is given a Partial Max-CSP with 50 soft constraints. An admissible upper-bound heuristic could start with ub = 51 because it is impossible to violate that many soft constraints. The depth-first search process finds a solution that violates 27 soft constraints, so ub is reduced to 27. As search continues, it produces a partial assignment that violates only 25 soft constraints, but the admissible lower-bound heuristic is powerful enough to estimate that any complete extension will violate at least 30 soft constraints. Since it is impossible to extend this partial assignment to a solution that is better than the one already found, the algorithm can prune this branch and search elsewhere in the tree.

This example demonstrates why the heuristics are so important to branch-and-bound search. Branch-and-bound can backtrack whenever the upper and lower bounds cross. This means that decreasing the upper bound and increasing the lower bound will make them cross sooner, allowing for earlier pruning and improved efficiency. It is important to maintain admissibility, however, because the use of an inadmissible heuristic might cause branch-and-bound to backtrack too early and miss an optimal solution. Branch-and-bound is complete only if the upper- and lower-bound heuristics are both admissible.

Because of the importance of heuristics in branch-and-bound search, a significant amount of COP research focuses on identifying heuristics that are both powerful and admissible. Much of this research attempts to improve lower-bound heuristics by incorporating constraint propagation techniques into branch-and-bound algorithms. Bistarelli et al. (1996) discuss incorporating soft consistency checks into branch-and-bound for binary VCSP. Larrosa and Schiex (2004) present efficient algorithms for maintaining AC in WCSP and WSAT. Cooper et al. (2007) present a new method for soft AC in WCSP that improves on existing lower bound heuristics. Dago and Verfaillie (1996) define valued nogoods, which extend the definition of nogoods used for CSP and make it possible to improve the quality of lower-bound estimates during the course of branch-and-bound search.

Other research focuses on combining techniques into larger COP solvers that incorporate branch-and-bound search. Fu and Malik (2006) present 2 complete Partial Max-SAT algorithms based on Chaff. Heras et al. (2007) extend the MiniSAT solver to handle Max-SAT optimization in a solver they call MiniMaxSat. Bouveret et al. (2006) review a set of techniques of Max-CSP solvers submitted to the 2006 Max-CSP competition.

Assuming admissible heuristics, branch-and-bound search is sound and complete. Because it keeps finding better solutions and can be combined with efficiency techniques, it offers fairly good anytime performance for many applications. It is especially well-suited for partial constraint optimizations in which it is possible that the constraints cannot all be satisfied simultaneously.

3.2.3 Local Search

Local search is a very popular approach to COP. Recall that when using local search to solve a CSP, an evaluation function ranks each assignment based on how closely it matches a solution. To use local search to solve a COP, one need only design the evaluation function to implement the preference operator. In other words, when solving a CSP, the evaluation function of a local-search algorithm measures how close the assignment is to satisfying all of the constraints. When solving a COP, the evaluation function of a local-search algorithm measures how preferred the assignment is. The semantics of the evaluation function change but the local-search algorithm does not. Local search is an appealing option because the anytime performance is generally very good, but because it is incomplete, local search is not guaranteed to produce a globally optimal solution.

Greedy- and random-search algorithms have been applied to many types of optimization problems. Talbi et al. (1997) use tabu search to solve large optimization problems in a parallel environment. Daum and Menzel (2002) perform natural language parsing by casting the problem as WCSP and solving it with local search. Kautz et al. (1997) present MaxWalkSAT, an extension of the WalkSAT algorithm (Selman et al. 1993), which is designed to solve instances of Max-SAT.

Genetic algorithms are generally the method of choice when solving instances of MOP. Because the objectives of an instance of MOP can conflict with one another, a variety of Pareto optimal solutions is often preferable to a single solution. Greedy- and random-local-search algorithms only move from one assignment to a neighbor, so they focus on one small part of the search space at a time. Genetic algorithms, on the other hand, search a variety of assignments simultaneously across the population, making them ideal for quickly approximating the Pareto frontier. Van Veldhuizen and Lamont (2000) present a theoretical review of methods of evolutionary algorithms applied to MOP, while Zitzler et al. (2000) present an empirical comparison. Some research has combined depth-first search and local search into hybrid algorithms for constraint optimization. Sosič and Wilby (1994) demonstrate that a hybrid approach produces better anytime performance than either depth-first search or local search alone in instances of large traveling salesman optimization problems. Sachenbacher and Williams (2005) present a method that exploits plateaus in the search space by reformulating VCSPs into a

combination of hard and soft constraints, separating the search into distinct satisfaction and optimization phases. Hang et al. (2007) discuss the use of local search to improve approximate Max-CSP algorithms.

Finally, several methods have been explored for improving efficiency when solving one type of COP by converting the problem to another type. For example, de Givry et al. (2003) describe methods for solving Max-SAT by converting it to WCSP. On the other hand, Ansótegui et al. (2007) describe a method for solving WCSP by converting it to Max-SAT. Argelich et al. (2008) present several efficient methods for encoding Max-CSP as Partial Max-SAT.

4. Conclusions

This report defines several different types of constraint satisfaction and optimization problems, explains the relationships between them, and details solution approaches and techniques. It cites numerous references for the interested reader to find even more detailed information about the topics discussed here.

5. References

Ansótegui C, Bonet ML, Levy J, Manyà F. The logic behind weighted CSP. In: Sangal R, Mehta H, Bagga RK, editors. IJCAI'07. Proceedings of the 20th International Joint Conference on Artificial Intelligence; 2007 Jan.; Hyderabad, India. San Francisco (CA): Morgan Kaufmann Publishers Inc.; c2007. p. 32–37.

Argelich J, Cabiscol A, Lynce I, Manyà F. Encoding Max-CSP into partial Max-SAT. Presented at the 38th International Symposium on Multiple Valued Logic; 2008 May 22–23; Dallas, TX.

Barták R, Tomáš M, Hana R. A new approach to modeling and solving minimal perturbation problems. In: Apt KR, Fages F, Rossi F, Szeredi P, Vancza J, editors. Recent advances in constraints: lecture notes in computer science 3010. Berlin (Germany): Springer; 2003. p. 233–249.

Beck CJ, Prosser P, Wallace RJ. Toward understanding variable ordering heuristics for constraint satisfaction problems. Presented at the 14th Irish Artificial Intelligence and Cognitive Science Conference; 2003; Dublin, Ireland.

Bessière C. Constraint propagation. Montpellier (France): Centre d'Ecologie Fonctonnelle & Evolutive (CNRS), University of Montpellier, France; 2006. Technical Report No.: LIRMM 06020.

Bessière C, Régin J-C. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. CP'96. In: Freuder EG, editor. Proceedings of the International Conference on Principles and Practice of Constraint Programming; 1996 Aug 19–22; Cambridge, MA. Berlin (Germany): Springer; c1996. p. 61–75.

Bistarelli S, Fargier H, Montanari U, Rossi F, Schiex T, Verfaillie G. Semiring-based CSPs and valued CSPs: basic properties and comparison. In: Freuder E, Maher M, Jampel M, editors. Over-constrained systems. Paris (France): INRA Editions; 1996. p. 111–150.

Black PE. Branch-and-bound. US national institute of standards and technology dictionary of algorithms and data structures; 2005 [accessed 2008 Oct 16]. http://www.nist.gov/dads /HTML/branchNbound.html.

Blickle T, Thiele, L. A comparison of selection schemes used in evolutionary algorithms. Evolutionary Computation. 1996;4(4):361–394.

Bouveret S, de Givry S, Heras F, Larrosa J, Rollon E, Sanchez M, Ytnicki M. Max-CSP competition 2006: toolbar/toolbar2 solver brief description. Presented at the 3rd International Workshop on Constraint Propagation and Implementation; 2006; Nantes, France.

Chatzikokolakis K, Boukeas G, Stamatopoulos P. Construction and repair: a hybrid approach to search in CSPs. Presented at the 3rd Hellenic Conference on Artificial Intelligence; 2004 May 5–8; Samos, Greece.

Chen X, van Beek P. Conflict-directed backjumping. Journal of Artificial Intelligence Research. 2001;14:53–81.

Cooper MC. An optimal k-consistency algorithm. Artificial Intelligence. 1990;41(1):89–95.

Cooper MC, de Givry S, Schiex T. Optimal soft arc consistency. IJCAI'07. Proceedings of the 20th International Joint Conference on Artificial Intelligence; 2007 Jan 6–12; Hyderabad, India. San Francisco (CA): Morgan Kaufmann Publishers; c2007. p. 68–73.

Creignou N, Khanna S, Sudan M. Complexity classifications of Boolean constraint satisfaction problems. Philadelphia (PA): SIAM Press; 2001.

Dago P, Verfaillie G. Nogood recording for valued constraint satisfaction problems. ICTAI'96. Proceedings of the 8th International Conference on Tools with Artificial Intelligence; 1996 Nov; Toulouse, France. Washington (DC): IEEE Computer Society; c1996. p. 132–139.

Daum M, Menzel W. Parsing natural language using guided local search. ECAI'02. Proceedings of the 15th European Conference on Artificial Intelligence; 2002; Lyon, France. Amsterdam (The Netherlands): IOS Press; c2002. p. 435–439.

Davis M, Logemann G, Loveland D. A machine program for theorem proving. Communications of the ACM. 1962;5(7):394–397.

de Givry S, Larrosa J, Meseguer P, Schiex T. Solving Max-SAT as weighted CSP. Lecture Notes in Computer Science. 2003;2833:363–376.

Deb K. Genetic algorithm in search and optimization: the technique and applications. Presented at the International Workshop on Intelligent Systems and Soft Computing; 1997; Aachen, Germany. p. 58–87.

Dechter R. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. Artificial Inteligence. 1989;41:273–312.

Dechter R. From local to global consistency. Artificial Intelligence. 1992;55,87–107.

Dechter R. Constraint processing. San Francisco (CA): Morgan Kaufmann Publishers; 2003.

Dechter R, Pearl J. Network-based heuristics for constraint-satisfaction problems. Artificial Intelligence. 1987;34(1):1–38.

Dechter R, van Beek P. Local and global relational consistency. Theoretical Computer Science. 1995;173(1):283–308.

Dent MJ, Mercer RE. Minimal forward checking. Proceedings of the 6th IEEE International Conference on Tools with Artificial Intelligence; 1994 Nov [accessed 2015 Nov 17]. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=1012. p. 432–438.

Dixon HE, Ginsberg ML, Parkes AJ. Generalizing Boolean satisfiability: background and survey of existing work. Journal of Artificial Intelligence Research. 2004;24:193–243.

Een N, Sörensson N. An extensible SAT-solver. Proceedings of theory and applications of satisfiability testing: lecture notes in computer science. Berlin (Germany): Springer; 2003.

Fargier H, Lang J. Uncertainty in constraint satisfaction problems: a probabilistic approach. In: Symbolic and quantitative approaches to reasoning and uncertainty: lecture notes in computer science. Berlin (Germany): Springer; 2005. Vol. 747; p. 97–104.

Freuder EC. Synthesizing constraint expressions. Communications of the ACM. 1978;21:958–966.

Freuder EC. A sufficient condition for backtrack-free search. Journal of the ACM. 1982;29:24–32.

Freuder EC, Wallace RJ. Partial constraint satisfaction. Artificial Intelligence. 1992;58(1–3):21–70.

Fu Z, Malik S. On solving the partial MAX-SAT problem. Proceedings of theory and applications of satisfiability testing: lecture notes in computer science. Vol. 4121. Berlin (Germany): Springer; 2006. p. 252–265.

Geelen PA. Dual viewpoint heuristics for binary constraint satisfaction problems. ECAI ’92. In: Kautz H, Selman B, editors. Proceedings of the 10th European Conference on Artificial Intelligence; 1992 Aug 3–7; Vienna, Austria. New York (NY): John Wiley and Sons; c1992. p. 31–35.

Gent IP, MacIntyre E, Prosser P, Smith BM, Walsh T. An empirical study of dynamic variable ordering heuristics for constraint satisfaction problems. Proceedings of CP96. Berlin (Germany): Springer; 1996. p. 179–193.

Ginsberg ML, McAllester DA. Dynamic backtracking. Journal of Artificial Intelligence Research. 1993;1:25–46.

Glover F, Laguna M. Tabu search. In: Reeves C, editor. Modern heuristic techniques for combinatorial problems. Oxford (England): Blackwell Scientific Publishing; 1993. p. 70–141.

Gomes CP, Selman B, Crato N, Kautz H. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. Journal of Automated Reasoning. 2000;24(1–2):67–100.

Hampson S, Kibler D. Large plateaus and plateau search in Boolean satisfiability problems: when to give up searching and start again. In: DIMACS series in discrete mathematics and theoretical computer science, Vol. 26. Providence (RI): American Mathematical Society; 1993 .p. 437–455.

Hang CD, Abdelmeged A, Rinehart D, Lieberherr KJ. The promise of polynomial-based local search to boost Boolean MAX-CSP solvers. Presented at the 4th International Workshop on Local Search Techniques in Constraint Satisfaction; 2007 Sep 23; Providence, RI.

Hao J-K, Pannier J. Simulated annealing and tabu search for constraint solving. Presented at the 5th International Symposium on Artificial Intelligence and Mathematics; 1998 Jan; Ft. Lauderdale, FL.

Haralick RM, Elliott GL. Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence. 1980;14:263–313.

Heras F, Larrosa J, Oliveras A. MiniMaxSat: A new weighted Max-SAT solver. SAT’07. Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing; 2007 May 28–31; Lisbon, Portugal. Berlin (Germany): Springer; c2007. p. 41–55.

Jussien N, Lhomme O. Local search with constraint propagation and conflict-based heuristics. Artificial Intelligence. 2002;139:21–45.

Kautz H, Selman B, Jiang Y. A general stochastic approach to solving problems with hard and soft constraints. In: Dingzhu G, Du J, Pardalos P, editors. The satisfiability problem: theory and applications. DIMACS series in discrete mathematics and theoretical computer science. 1997;35:573–586.

Kirkpatrick S, Gelatt CD, Vecchi MP. Optimization by simulated annealing. Science. 1983;13:671–680.

Kondrak G, van Beek P. A theoretical evaluation of selected backtracking algorithms. Artificial Intelligence. 1997;89:541–547.

Larrosa J, Schiex T. Solving weighted CSP by maintaining arc consistency. Artificial Intelligence. 2004;159(1–2):1–26.

Lecoutre C, Saïs L, Vion J. Using SAT encodings to derive CSP value ordering heuristics. Journal on Satisfiability, Boolean Modeling and Computation. 2007;1:169–186.

Mackworth AK. Consistency in networks of relations. Artificial Intelligence. 1977;8(1):99–118.

Michalewicz Z. A survey of constraint handling techniques in evolutionary computation methods. In: McDonnell J, Reynolds R, Fogel, D, editors. Evolutionary programming: proceedings of the 4th Annual Conference on Evolutionary Programming; 1995. Cambridge (MA): MIT Press; c1995. p. 135–155.

Minton S, Johnston MD, Philips AB, Laird P. Minimizing conflicts: a heuristic repair method for constraint-satisfaction and scheduling problems. Artificial Intelligence.
1992;58(1–3):161–205.

Mohr R, Henderson TC. Arc and path consistency revised. Artificial Intelligence. 1986;28:225–233.

Mohr R, Masini G. Good old discrete relaxation. ECAI-88. Proceedings of the 8th European Conference on Artificial Intelligence. London (UK): Pitman; 1988. p. 651–656.

Montanari U. Networks of constraints: fundamental properties and applications to picture processing. Information Science. 1974;7(66), 95–132.

Morrison J. Co-evolution and genetic algorithms [thesis]. [Ottawa (Canada)]: Carleton University; 1988.

Moskewicz MW, Madigan CF, Zhao Y, Zhang L, Malik S. Chaff: engineering an efficient SAT solver. DAC'01. In: Rabaey J, editor. Proceedings of the 38th Design Automation Conference; 2001 June 18–22; Las Vegas, NV. New York (NY): ACM; c2001.

Narendra J, Debruyne R, Boizumault P. Maintaining arc-consistency within dynamic backtracking. In: Principles and practice of constraint programming: lecture notes in computer programming. Vol. 1894. Berlin (Germany): Springer; 2000. p. 249–261.

Pareto V. Manuale d'economia politica. Milan (Italy): Societa Editrice Libaria; 1906.

Prosser P. Hybrid algorithms for the constraint satisfaction problem. Computational Intelligence. 1993;9:268–299.

Rosenfeld A, Hummel R, Zucker S. Scene labeling by relaxation operations. IEEE Transactions on Systems, Man, and Cybernetics. 1976;6:173–184.

Ruttkay Z. Fuzzy constraint satisfaction. Proceedings of the 3rd international conference on fuzzy systems. Piscataway (NJ): IEEE Press; 1994. p. 1263–1268.

Sabin D, Freuder E. Contradicting conventional wisdom in constraint satisfaction. CP'94. In: Borning A, editor. Principles and practice of constraint programming: lecture notes in computer science. Vol. 874. Berlin (Germany): Springer; 1994.

Sachenbacher M, Williams BC. Solving soft constraints by separating optimization and satisfiability. CP'05. In: van Beek P, editor. Principles and practice of constraint programming: CP 2005. Berlin (Germany): Springer; 2005. p. 119–132.

Sawaragi Y, Nakayama H, Tanino T. Theory of multiobjective optimization. Mathematics in Science and Engineering. Orlando (FL): Academic Press Inc.; 1985.

Schiex T. Arc consistency for soft constraints. Artificial Intelligence. 2000:411–424.

Schiex T, Verfaillie G. Nogood recording for static and dynamic constraint satisfaction. International Journal of Artificial Intelligence Tools. 1994;3:48–55.

Schiex T, Fargier H, Verfaillie G. Valued constraint satisfaction problems: hard and easy problems. In: IJCAI95: Proceedings of the 14th International Joint Conference on Artificial Intelligence. Vol. 1. San Francisco (CA): Morgan Kaufmann Publishers, Inc.; 1995. p. 631–637.

Schöning U. A probabilistic algorithm for k-SAT and constraint satisfaction problems. FOCS ’99. Proceedings of the 40th Symposium on the Foundations of Computer Science. Washington (DC): IEEE Computer Society; 1999. p. 410–414.

Selman B, Kautz H. An empirical study of greedy local search for satisfiability testing. AAAI-93. Proceedings of the 11th National Conference on Artificial Intelligence. Palo Alto (CA): AAAI; 1993. p. 46–51.

Selman B, Kautz H. Domain-independent extensions of GSAT: solving large structured satisfiability problems [accessed 2015 Nov 17]. http://ijcai.org/Past%20Proceedings/IJCAI-93-VOL1/PDF/041.pdf.

Selman B, Kautz H, Cohen B. Local search strategies for satisfiability testing. Cliques, coloring, and satisfiability: second DIMACS implementation challenge. In: Johnson DS, Trick MA, editors. DIMACS series in discrete mathematics and theoretical computer science. Vol. 26. Providence (RI): American Mathematical Society, 1996.

Selman B, Levesque H, Mitchell D. A new method for solving hard satisfiability problems. AAAI-92. Proceedings of the 10th National Conference on Artificial Intelligence. Palo Alto (CA): AAAI; 1992. p. 440–446.

Smith BM, Grant SA. Trying harder to fail first. ECAI’98. In: Prade H, editor. Proceedings of the 13th European Conference on Artificial Intelligence; London (UK): Wiley; 1998. p. 249–253.

Sosič R, Wilby GD. Using the quality-time tradeoff in local optimization. Proceedings of the 2nd Australia New Zealand Intelligent Information Systems (ANZIIS) Conference; 1994. Piscataway (NJ): Institute of Electrical and Electronics Engineers; c1994. p. 253–257.

Talbi EG, Hafidi Z, Geib J-M. Parallel adaptive tabu search for large optimization problems. In: Vob S, Martello S, Osman IH, Roucairol C, editors. Meta-heuristics: advances and trends in local search paradigms for optimization. Berlin (Germany): Springer; 1999. p. 345–358.

Thronton JR. Constraint weighting local search for constraint satisfaction [thesis]. [Nathan (Australia)]: Griffith University; 2000.

Tinós R, Yang S. A self-organizing random immigrants genetic algorithm for dynamic optimization problems. Genetic Programming and Evolvable Machines. 2007;8(3):255–286.

Van Veldhuizen DA, Lamont GB. Multiobjective evolutionary algorithms: analyzing the state-of-the-art. Evolutionary Computation. 2000;8(2):125–147.

Vrajitoru D. Simulating gender separation with genetic algorithms. GECCO'02. Proceedings of the Genetic and Evolutionary Computation Conference; 2002 July 9–13; New York, NY. Burlington (MA): Morgan Kaufmann Publishers; c2002.

Wallace M. Practical applications of constraint programming. Constraints: An International Journal. 1996;1:139–168.

Zhang H, Stickel M. An efficient algorithm for unit-propagation. Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics; 1996 Jan 3–5; Ft. Lauderdale, FL.

Zitzler E, Deb K, Thiele L. Comparison of multiobjective evolutionary algorithms: empirical results. Evolutionary Computation. 2000;8:173–195.

List of Symbols, Abbreviations, and Acronyms

AC	arc consistency
AOF	aggregate objective function
APX	approximately
COP	Constraint Optimization Problem
CSP	Constraint Satisfaction Problem
DPLL	Davis-Putnam-Logemann-Loveland
FCSP	Fuzzy CSP
GSAT	Greedy SAT
MOP	Multiobjective Optimization Problem
NP	nondeterministic polynomial
PCSP	Probabilistic CSP
SAT	satisfiability
SCSP	Semiring-based CSP
VCSP	Valued CSP
WCSP	Weighted CSP
WP Max-SAT	Weighted Partial Max-SAT
WSAT	Weighted SAT

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIRECTOR US ARMY RESEARCH LAB RDRL CIO LL IMAL HRA MAIL & RECORDS MGMT
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
1 (PDF)	DIR USARL RDRL CIH S P JUNGWIRTH

INTENTIONALLY LEFT BLANK.