Handbook of Constraint Programming
Edited by F. Rossi, P. van Beek and T. Walsh

Chapter 2

Constraint Satisfaction: An Emerging Paradigm

Eugene C. Freuder and Alan K. Mackworth

This chapter focuses on the emergence of constraint satisfaction, with constraint languages, as a new paradigm within artificial intelligence and computer science during the period from 1965 (when Golomb and Baumert published "Backtrack programming" [34]) to 1985 (when Mackworth and Freuder published "The complexity of some polynomial network consistency algorithms for constraint satisfaction problems" [55]). The rest of this handbook will cover much of the material introduced here in more detail, as well as, of course, continuing on from 1986 into 2006.

2.1 The Early Days

Constraint satisfaction, in its basic form, involves finding a value for each one of a set of problem variables where constraints specify that some subsets of values cannot be used together. As a simple example of constraint satisfaction, consider the task of choosing component parts for the assembly of a bicycle, such as the frame, wheels, brakes, sprockets and chain, that are all mutually compatible.

Constraint satisfaction, like most fields of artificial intelligence, can be separated into (overlapping) concerns with representation and reasoning. The former can be divided into generic and application-specific concerns, the latter into search and inference. While constraint satisfaction has often been pigeon-holed as a form of search, its real importance lies in its broad representational scope: it can be used effectively to model many other forms of reasoning (e.g. temporal reasoning) and applied to many problem domains (e.g. scheduling). For this reason, constraint satisfaction problems are sometimes encountered in application domains that are unaware that an academic community has been studying the subject for years: one reason for the importance of a handbook such as this. Furthermore, while heuristic search methods are a major concern, the distinguishing feature of constraint satisfaction as a branch of artificial intelligence is arguably the emphasis on inference, in the form of constraint propagation, as opposed to search.

Constraint satisfaction problems have been tackled by a dizzying array of methods, from automata theory to ant algorithms, and are a topic of interest in many fields of computer science and beyond. These connections add immeasurably to the richness of the subject, but are largely beyond the scope of this chapter. Here we will focus on the basic methods involved in the establishment of constraint satisfaction as a branch of artificial intelligence. This new branch of artificial intelligence, together with related work on programming languages and systems that we can only touch upon here, laid the groundwork for the flourishing of interest in constraint programming languages after 1985.

Constraint satisfaction of course, predates 1965. The real world problems that we now identify as constraint satisfaction problems, like workforce scheduling, have naturally always been with us. The toy 8-queens problem, which preoccupied so many of the early constraint satisfaction researchers in artificial intelligence, is said to have been proposed in 1848 by the chess player Max Bazzel. Mythology claims that a form of backtrack search, a powerful search paradigm that has become a central tool for constraint satisfaction, was used by Theseus in the labyrinth in Crete. Backtrack search was used in recreational mathematics in the nineteenth century [51], and was an early subject of study as computer science and operations research emerged as academic disciplines after World War II. Bitner and Reingold [2] credit Lehmer with first using the term 'backtrack' in the 1950's [50]. Various forms of constraint satisfaction and propagation appeared in the computer science literature in the 1960's [16, 15, 34, 75].

In artificial intelligence interest in constraint satisfaction developed in two streams. In some sense a common ancestor of both streams is Ivan Sutherland's groundbreaking 1963 MIT Ph.D. thesis, "Sketchpad: A man-machine graphical communication system" [73].

In one stream, the versatility of constraints led to applications in a variety of domains, and associated programming languages and systems. This stream we can call the language stream. In 1964 Wilkes proposed that algebraic equations be allowed as constraint statements in procedural Algol-like programming languages, with relaxation used to satisfy the constraints [80]. Around 1967, Elcock developed a declarative language, Absys, based on the manipulation of equational constraints [22]. Burstall employed a form of constraint manipulation as early as 1969 in a program for solving cryptarithmetic puzzles [9]. In the very first issue of *Artificial Intelligence* in 1970, Fikes described REF-ARF, where the REF language formed part of a general problem-solving system employing constraint satisfaction and propagation as one of its methods [23]. Kowalski used a form of constraint propagation for theorem proving [48]. Sussman and others at MIT applied a form of constraint propagation to analysis, synthesis and fault localization for circuits [6, 17, 18, 67, 71], and Sussman with Steele developed the CONSTRAINTS language [72]. Borning used constraints in his ThingLab simulation laboratory [4, 5], whose kernel was an extension of the Smalltalk language; Lauriere used constraints in Alice, a language for solving combinatorial problems [49]. In the planning domain, Eastman did "constraint structured" space planning with GSP, the General Space Planner [21], Stefik used "constraint posting" in MOLGEN, which planned gene-cloning experiments in molecular genetics [68, 69], and Descotte and Latombe's GARI system, which generated the machining plans of mechanical parts, embedded a planner which made compromises among "antagonistic constraints" [20]. Fox, Allen and Strohm developed ISIS-II [25] a constraint-directed reasoning system for factory job-shop scheduling.

In the other stream, an interest in constraint solving algorithms grew out of the machine vision community; we cite some of the early work here. We refer to this stream as

the algorithm stream. The landmark 'Waltz filtering' (arc consistency) constraint propagation algorithm appeared in a Ph.D. thesis on scene labeling [79], building upon work of Huffman [41] and Clowes [10]. Montanari developed path consistency and established a general framework for representing and reasoning about constraints in a seminal paper entitled "Networks of constraints: fundamental properties and applications to picture processing" [60]. Mackworth exploited constraints for machine vision [52], before providing a general framework for "Consistency in networks of relations" and new algorithms for arc and path consistency [53]. Freuder generalized arc and path consistency to k-consistency [26] shortly after completing a Ph.D. thesis on "active vision". Barrow and Tenenbaum, with MSYS [1] and IGS [74], were also early users of constraints for image interpretation. Rosenfeld, Hummel and Zucker, in "Scene labeling by relaxation operations", explored the "continuous labeling problem", where constraints are not 'hard', specifying that values can or cannot be used together, but 'soft' specifying degrees of compatibility [65]. Haralick, Davis, Rosenfeld and Milgram discussed "Reduction operations for constraint satisfaction" [38], and Haralick and Shapiro generalized those results in a two-part paper on "The consistent labeling problem" [36, 37]. Together with J. R. Ullman, they even discussed special hardware for constraint propagation and parallel search computation in [76].

The language and algorithm streams diverged, and both became more detached from specific application domains. While applications and commercial exploitation did proliferate, the academic communities focused more on general methods. While the generality and scientific rigor of constraint programming is one of its strengths, we face a continuing challenge to reconnect these streams more firmly with their semantic problem-solving roots.

The language stream became heavily influenced by logic programming, in the form of constraint logic programming, and focused on the development of programming languages and libraries. Hewitt's Planner language [40] and its partial implementation as Micro-Planner [70] can be seen as an early logic programming language [3]. The major early milestone, though, was the development of Prolog by Colmerauer and others around 1972 [14] and the logic as a programming language movement [39, 47]. Prolog can be framed as an early constraint programming language, solving equality constraints over terms (including variables) using the unification algorithm as the constraint solver. Colmerauer pushed this view much further in his introduction of Prolog II in 1982 [13, 12]. The integration of constraint propagation algorithms into interpreters for Planner-like languages was proposed by Mackworth [53]. Van Hentenryck developed and implemented CHIP (Constraint Handling in Prolog) as a fully-fledged constraint logic programming language [77]. In a parallel development Jaffar *et al.* developed the CLP(X) family of constraint logic programming languages [42] including CLP($\mathcal{R}$) [44]. For more on these developments in the language stream see the surveys in [11, 43] and other chapters in this handbook.

The algorithm stream, influenced by the paradigm of artificial intelligence as search, as exemplified in Nilsson's early textbook [61], and by the development of the science of algorithms, as exemplified by Knuth's *The Art of Computer Programming* [45], focused on algorithms and heuristics. The second stream remained more firmly within artificial intelligence, developing as one of the artificial intelligence communities built around reasoning paradigms: constraint-based reasoning [29], case-based reasoning, and the like. It also focused increasingly on the simple, but powerful and general, *constraint satisfaction problem* (CSP) formulation and its variants. We shall focus primarily on this stream, and the development of the CSP paradigm, in this chapter.

The challenge then became to reintegrate the language and algorithm streams, along with related disciplines, such as mathematical programming and constraint databases, into a single constraint programming community. This process began in earnest in the 1990's when Paris Kanellákis, Jean-Louis Lassez, and Vijay Saraswat chaired a workshop that soon led to the formation of an annual International Conference on Principles and Practice of Constraint Programming, and, at the instigation of Zsófia Ruttkay, Gene Freuder established the *Constraints* journal, which "provides a common forum for the many disciplines interested in constraint programming and constraint satisfaction and optimization, and the many application domains in which constraint technology is employed".

2.2 The Constraint Satisfaction Problem: Representation and Reasoning

Here we consider the representation of constraint satisfaction problems, the varieties of reasoning used by algorithms to solve them and the analysis of those solution methods.

2.2.1 Representation

The classic definition of a Constraint Satisfaction Problem (CSP) is as follows. A CSP $\mathcal{P}$ is a triple $\mathcal{P} = \langle X, D, C \rangle$ where X is an n-tuple of variables $X = \langle x_1, x_2, \ldots, x_n \rangle$, D is a corresponding n-tuple of domains $D = \langle D_1, D_2, \ldots, D_n \rangle$ such that $x_i \in D_i$, C is a t-tuple of constraints $C = \langle C_1, C_2, \ldots, C_t \rangle$. A constraint C_j is a pair $\langle R_{S_j}, S_j \rangle$ where R_{S_j} is a relation on the variables in $S_i = scope(C_i)$. In other words, R_i is a subset of the Cartesian product of the domains of the variables in S_i.[1]

A solution to the CSP $\mathcal{P}$ is an n-tuple $A = \langle a_1, a_2, \ldots, a_n \rangle$ where $a_i \in D_i$ and each C_j is satisfied in that R_{S_j} holds on the projection of A onto the scope S_j. In a given task one may be required to find the set of all solutions, $sol(\mathcal{P})$, to determine if that set is non-empty or just to find any solution, if one exists. If the set of solutions is empty the CSP is unsatisfiable. This simple but powerful framework captures a wide range of significant applications in fields as diverse as artificial intelligence, operations research, scheduling, supply chain management, graph algorithms, computer vision and computational linguistics, to name but a few.

The classic CSP paradigm can be both specialized and generalized in a variety of important ways. One important specialization considers the extensionality/intensionality of the domains and constraints. If all the domains in D are finite sets, with extensional representations, then they, and the constraint relations, may be represented and manipulated extensionally. However, even if the domains and the relations are intensionally represented, many of the techniques described in this chapter and elsewhere in the handbook still apply. If the size of the scope of each constraint is limited to 1 or 2 then the constraints are unary and binary and the CSP can be directly represented as a constraint graph with variables as vertices and constraints as edges. If the arity of constraints is not so limited then a hypergraph is required with a hyperedge for each p-ary constraint ($p > 2$) connecting the p vertices involved. The satisfiability of propositional formulae, SAT, is another

[1] This is the conventional definition, which we will adhere to here. A more parsimonious definition of a CSP would dispense with D entirely leaving the role of D_i to be played by a unary constraint C_j with $scope(C_j) = \langle x_i \rangle$.

specialization of CSP, where the domains are restricted to be $\{T, F\}$ and the constraints are clauses. 3-SAT, the archetypal NP-complete decision problem, is a further restriction where the scope of each constraint (clause) is 3 or fewer variables.

The classic view of CSPs was initially developed by Montanari [60] and Mackworth [53]. It has strong roots in, and links with, SAT [16, 15, 54], relational algebra and database theory [58], computer vision [10, 41, 79] and graphics [73].

Various generalizations of the classic CSP model have been developed subsequently. One of the most significant is the Constraint Optimization Problem (COP) for which there are several significantly different formulations, and the nomenclature is not always consistent [19]. Perhaps the simplest COP formulation retains the CSP limitation of allowing only 'hard' Boolean-valued constraints but adds a cost function over the variables, that must be minimized. This arises often, for example, in scheduling applications.

2.2.2 Reasoning: Inference and Search

We will consider the algorithms for solving CSPs under two broad categories: inference and search, and various combinations of those two approaches. If the domains D_i are all finite then the finite search space for putative solutions is $\Omega = \bowtie_i D_i$ (where $\bowtie$ is the join operator of relational algebra [58]). Ω can, in theory, be enumerated and each n-tuple tested to determine if it is a solution. This blind enumeration technique can be improved upon using two distinct orthogonal strategies: inference and search. In inference techniques, local constraint propagation can eliminate large subspaces from Ω on the grounds that they must be devoid of solutions. Search systematically explores Ω, often eliminating subspaces with a single failure. The success of both strategies hinges on the simple fact that a CSP is conjunctive: to solve it, all of the constraints must be satisfied so that a local failure on a subset of variables rules out all putative solutions with the same projection onto those variables. These two basic strategies are usually combined in most applications.

2.2.3 Inference: Constraint Propagation Using Network Consistency

The major development in inference techniques for CSPs was the discovery and development, in the 1970's, of network consistency algorithms for constraint propagation. Here we will give an overview of that development.

Analysis of using backtracking to solve CSPs shows that it almost always displays pathological *thrashing* behaviors [3]. Thrashing is the repeated exploration of failing subtrees of the backtrack search tree that are essentially identical–differing only in assignments to variables irrelevant to the failure of the subtree. Because there is typically an exponential number of such irrelevant assignments, thrashing is often the most significant factor in the running time of backtracking.

The first key insight behind all the consistency algorithms is that much thrashing behavior can be identified and eliminated, once and for all, by tightening the constraints, making implicit constraints explicit, using tractable, efficient polynomial-time algorithms. The second insight is that the level, or scope, of consistency, the size of the set of variables involved in the local context, can be adjusted as a parameter from 1 up to n, each increase in level requiring correspondingly more work.

For simplicity, we will initially describe the development of the consistency algorithms for CSPs with finite domains and unary and binary constraints only, though neither restric-

tion is necessary, as we shall see. We assume the reader is familiar with the basic elements of graph theory, set theory and relational algebra.

Consider a CSP $\mathcal{P} = \langle X, D, C \rangle$ as defined above. The unary constraints are $C_i = \langle R_{\langle x_i \rangle}, \langle x_i \rangle \rangle$. We use the shorthand notation R_i to stand for $R_{\langle x_i \rangle}$. Similarly, the binary constraints are of the form $C_s = \langle R_{\langle x_i, x_j \rangle}, \langle x_i, x_j \rangle \rangle$ where $i \neq j$. We use R_{ij} to stand for $R_{\langle x_i, x_j \rangle}$.

Node consistency is the simplest consistency algorithm. Node i comprised of vertex i representing variable x_i with domain D_i is node consistent iff $D_i \subseteq R_i$. If node i is not node consistent it can be made so by computing:

$$D_i' = D_i \bigcap R_i$$
$$D_i \leftarrow D_i'$$

A single pass through the nodes makes the network node consistent. The resulting CSP is $\mathcal{P}' = \langle X, D', C \rangle$ where $D' = \langle D_1', D_2', \ldots, D_n' \rangle$. We say $\mathcal{P}' = NC(\mathcal{P})$. Clearly $sol(\mathcal{P}) = sol(\mathcal{P}')$. Let $\Omega' = \bowtie_i D_i'$ then $|\Omega'| \leq |\Omega|$.

Arc consistency is a technique for further tightening the domains using the binary constraints. Consider node i with domain D_i. Suppose there is a non-trivial relation R_{ij} between variables x_i and x_j. We consider the arcs $\langle i, j \rangle$ and $\langle j, i \rangle$ separately. Arc $\langle i, j \rangle$ is arc consistent iff:

$$D_i \subset \pi_i(R_{ij} \bowtie D_j)$$

where π is the projection operator. That is, for every member of D_i, there is a corresponding element in D_j that satisfies R_{ij}. Arc $\langle i, j \rangle$ can be tested for arc consistency and made consistent, if it is not so, by computing:

$$D_i' = D_i \bigcap \pi_i(R_{ij} \bowtie D_j)$$
$$D_i \leftarrow D_i'$$

(This is a semijoin [58]). In other words, delete all elements of D_i that have no corresponding element in D_j satisfying R_{ij}. A network is arc consistent iff all its arcs are consistent. If all the arcs are already consistent a single pass through them is all that is needed to verify this. If, however, at least one arc has to be made consistent (i.e. $D_i' \neq D_i$ – there is a deletion from D_i) then one must recheck some number of arcs. The basic arc consistency algorithm simply checks all the arcs repeatedly until a fixed point of no further domain reductions is reached. This algorithm is known as AC-1 [53].

Waltz [79] realized that a more intelligent arc consistency bookkeeping scheme would only recheck those arcs that could have become inconsistent as a direct result of deletions from D_i. Waltz's algorithm, now known as AC-2 [53], propagates the revisions of the domains through the arcs until, again, a fixed point is reached. AC-3, presented by Mackworth [53], is a generalization and simplification of AC-2. AC-3 is still the most widely used and effective consistency algorithm. For each of these algorithms let $\mathcal{P}' = AC(\mathcal{P})$ be the result of enforcing arc consistency on $\mathcal{P}$. Then clearly $sol(\mathcal{P}) = sol(\mathcal{P}')$ and $|\Omega'| \leq |\Omega|$.

The best framework for understanding all the network consistency algorithms is to see them as removing local inconsistencies from the network which can never be part of any global solution. When those inconsistencies are removed they may propagate to cause

inconsistencies in neighboring arcs that were previously consistent. Those inconsistencies are in turn removed so the algorithm eventually arrives, monotonically, at a fixed point consistent network and halts. An inconsistent network has the same set of solutions as the consistent network that results from applying a consistency algorithm to it; however, if one subsequently applies, say, a backtrack search to the consistent network the resultant thrashing behavior can be no worse and almost always is much better, assuming the same variable and value ordering.

Path consistency [60] is the next level of consistency to consider. In arc consistency we tighten the unary constraints using local binary constraints. In path consistency we analogously tighten the binary constraints using the implicit induced constraints on triples of variables.

A path of length two from node i through node m to node j, $\langle i, m, j \rangle$, is path consistent iff:

$$R_{ij} \subset \pi_{ij}(R_{im} \bowtie D_m \bowtie R_{mj})$$

That is, for every pair of values $\langle a, b \rangle$ allowed by the explicit relation R_{ij} there is a value c for x_m such that $\langle a, c \rangle$ is allowed by R_{im} and $\langle c, b \rangle$ is allowed by R_{mj}.

Path $\langle i, m, j \rangle$ can be tested for path consistency and made consistent, if it is not, by computing:

$$R'_{ij} = R_{ij} \bigcap \pi_{ij}(R_{im} \bowtie D_m \bowtie R_{mj})$$
$$R_{ij} \leftarrow R'_{ij}$$

If the binary relations are represented as Boolean bit matrices then the combination of the join and projection operations (which is relational composition) becomes Boolean matrix multiplication and the $\bigcap$ operation becomes simply pairwise bit $\wedge$ operations. In other words, for all the values $\langle a, b \rangle$ allowed by R_{ij} if there is no value c for x_m allowed by R_{im} and R_{mj} the path is made consistent by changing that bit value in R_{ij} from 1 to 0. The way to think of this is that the implicit constraint on $\langle i, j \rangle$ imposed by node $\langle m \rangle$ through the relational composition $R_{im} \circ R_{mj}$ is made explicit in the new constraint R'_{ij} when path $\langle i, m, j \rangle$ is made consistent.

As with arc consistency the simplest algorithm for enforcing path consistency for the entire network is to check and ensure path consistency for each length 2 path $\langle i, m, j \rangle$. If any path has to be made consistent then the entire pass through the paths is repeated again. This is algorithm PC-1 [53, 60].

The algorithm PC-2 [53] determines, when any path is made consistent, the set of other paths could have become inconsistent because they use the arc between that pair of vertices and queues those paths, if necessary, for further checking. PC-2 realizes substantial savings over PC-1 just as AC-3 is more efficient than AC-1 [55].

Typically, after path consistency is established, there are non-trivial binary constraints between all pairs of nodes. As shown by Montanari [60], if all paths of length 2 are consistent then all paths of any length are consistent, so longer paths need not be considered. Once path consistency is established, there is a chain of values along any path satisfying the relations between any pair of values allowed at the start and the end of the path. This does *not* mean that there is necessarily a solution to the CSP. If a path traverses the entire network with a chain of compatible values, if that path self-intersects at a node the two

values on the path at that node may be different. Indeed, it is a property of both arc consistency and path consistency that consistency may be established with non-empty domains and relations even though there may be no global solution. Low-level consistency, with no empty domains, is a necessary but not sufficient condition for the existence of a solution. So, if consistency does empty any domain or relation there is no global solution.

Parenthetically, we note that our abstract descriptions of these algorithms, in terms of relational algebra, are specifications not implementations. Implementations can often achieve efficiency savings by, for example, exploiting the semantics of a constraint such as the all different global constraint, *alldiff*, that requires each variable in its scope to assume a different value.

Briefly, let us establish that consistency algorithms do not require the finite domain or binary constraint restrictions on the CSP model. As long as we can perform $\bowtie$, π and $\bigcap$ operations on the domain and relational representations these algorithms are perfectly adequate.

Consider, for example, the trivial CSP $\mathcal{P} = \langle\langle x_1, x_2\rangle, \langle[0,3],[2,5]\rangle, \langle =, \langle x_1, x_2\rangle\rangle\rangle$ where x_1 and x_2 are reals. That is, $x_1 \in D_1 = [0,3], x_2 \in D_2 = [2,5]$. Arc consistency on arc $\langle 1,2\rangle$ reduces D_1 to $[2,3]$ and arc consistency on arc $\langle 2,1\rangle$ reduces D_2 to $[2,3]$.

If some of the constraints are p-ary ($p > 2$) we can generalize arc consistency. In this case we can represent each p-ary constraint $C = \langle R_{S_j}, S_j\rangle$ as a hyperedge connecting the vertices representing the variables in S_j. Consider a vertex $x_i \in S_j$. We say we make the directional hyperarc $\langle x_i, S_j - \langle x_i\rangle\rangle$ generalized arc consistent by computing:

$$D_i' = D_i \bigcap \pi_i(R_{S_j} \bowtie (\bowtie_{m \in S_j - \langle x_i\rangle} D_m))$$
$$D_i \leftarrow D_i'$$

In other words the hyperarc is made generalized arc consistent, if necessary, by deleting from D_i any element that is not compatible with some tuple of its neighbors under the relation R_s. As with AC-3 any changes in D_i may propagate to any other hyperarcs directed at node i. This is the generalized arc consistency algorithm GAC [53]. One can also specialize arc consistency: Mackworth, Mulder and Havens exploited the properties of tree-structured variable domains in a hierarchical arc consistency algorithm HAC [57].

While there is no immediately obvious graph theoretic concept analogous to nodes, arcs and paths to motivate a higher form of consistency, the fact that consideration of paths of length two is, in fact, sufficient for path consistency, provides a natural motivation for the concept of *k-consistency* introduced by Freuder in 1978 [26]. k-consistency requires that given consistent values for any $k-1$ variables, there exists a value for any kth variable, such that all k values are consistent (i.e. the k values form a solution to the subproblem induced by the k variables). Thus 2-consistency is equivalent to arc consistency, and 3-consistency to path consistency. Freuder provided a synthesis algorithm for finding all the solutions to a CSP without search by achieving higher and higher levels of consistency.

Freuder went on in 1985 to generalize further to (i,j)-*consistency* [28]. A constraint network is (i,j)-consistent if, given consistent values for any i variables, there exist values for any other j variables, such that all $i+j$ values together are consistent. k-consistency is $(k-1,1)$-consistency. Special attention was paid to $(1,j)$-consistency, which is a generalization of what would now be termed 'singleton consistency'.

2.2.4 Search: Backtracking

Backtrack is the fundamental 'complete' search method for constraint satisfaction problems, in the sense that one is guaranteed to find a solution if one exists. Even in 1965, Golomb and Baumert, in a *JACM* paper simply entitled "Backtrack programming" [34], were able to observe that the method had already been independently 'discovered' many times. Golomb and Baumert believed their paper to be "the first attempt to formulate the scope and methods of backtrack programming in its full generality", while acknowledging the "fairly general exposition" given five years earlier by Walker [78].

Indeed, Golomb and Baumert's formulation is almost too general for our purposes here in that it is presented as an optimization problem, with the objective to maximize a function of the variables. Arguably Golomb and Baumert are presenting 'branch and bound programming', where upper and lower bounds on what is possible or desirable at any point in the search can provide additional pruning of the search. What we would now call a classic CSP, the 8-queens problem, they formulate by specifying a function whose value is 0 when the queens do not attack each other, and 1 otherwise. It is worth noting also that in this optimization context, again even in 1965, Golomb and Baumert acknowledge the existence of "learning programs and hill climbing programs" that converge on relative maxima. They observe dryly that while "the backtrack algorithm lacks such glamorous qualities as learning and progress, it has the more prosaic virtue of being exhaustive".

Basic backtrack search builds up a partial solution by choosing values for variables until it reaches a dead end, where the partial solution cannot be consistently extended. When it reaches a dead end it undoes the last choice it made and tries another. This is done in a systematic manner that guarantees that all possibilities will be tried. It improves on simply enumerating and testing of all candidate solutions by brute force in that it checks to see if the constraints are satisfied each time it makes a new choice, rather than waiting until a complete solution candidate containing values for all variables is generated. The backtrack search process is often represented as a search tree, where each node (below the root) represents a choice of a value for a variable, and each branch represents a candidate partial solution. Discovering that a partial solution cannot be extended then corresponds to pruning a subtree from consideration. Other noteworthy early papers on backtracking include Bitner and Reingold's "Backtrack programming techniques" [2] and Fillmore and Williamson's "On backtracking: a combinatorial description of the algorithm" [24], which used group theory to address symmetry issues.

Heuristic search methods to support general purpose problem solving paradigms were studied intensely from the early days of artificial intelligence, and backtracking played a role in the form of depth-first search of state spaces, problem reduction graphs, and game trees [61]. In the 1970's as constraint satisfaction emerged as a paradigm of its own, backtrack in the full sense we use the term here, for search involving constraint networks, gained prominence in the artificial intelligence literature, leading to the publication in the *Artificial Intelligence* journal at the beginning of the 1980's of Haralick and Elliott's "Increasing Tree Search Efficiency for Constraint Satisfaction Problems" [35]. This much-cited paper provided what was, for the time, an especially thorough statistical and experimental evaluation of the predominant approaches to refining backtrack search.

There are two major themes in the early work on improving backtracking: controlling search and interleaving inference (constraint propagation) with search. Both of these themes are again evident even in Golomb and Baumert. They observe that "all other things

being equal, it is more efficient to make the next choice from the set [domain] with fewest elements", an instance of what Haralick and Elliott dubbed the "fail first principle", and they discuss "preclusion", where a choice for one variable rules out inconsistent choices for other variables, a form of what Haralick and Elliott called "lookahead" that they presented as "forward checking". Of course, preclusion and the smallest domain heuristic nicely complement one another.

In general, one can look for efficient ways to manage search both going 'forward' and 'backward'. When we move forward, extending partial solutions, we make choices about the order in which we consider variables, values and constraints. This order can make an enormous difference in the amount of work we have to do. When we move backwards after hitting a dead end, we do not have to do this chronologically by simply undoing the last choice we made. We can be smarter about it. In general, constraint propagation, most commonly in the form of partial or complete arc consistency, can be carried out before, and/or during, search, in an attempt to prune the search space.

Haralick and Elliott compared several forms of lookahead, carrying out different degrees of partial arc consistency propagation after choosing a value. Oddly their "full lookahead" still did not maintain full arc consistency. However, restoring full arc consistency after choosing values had been proposed as early as 1974 by Gaschnig [31], and McGregor had even experimented with interleaving path consistency with search [59]. Mackworth observed that one could generalize to the alternation of constraint manipulation and case analysis, and proposed an algorithm that decomposed problems by splitting a variable domain in half and then restoring arc consistency on the subproblems [53].

Basic backtrack search backtracks chronologically to undo the last choice and try something else. This can result in silly behavior, where the algorithm tries alternatives for choices that clearly had no bearing on the failure that induced the backtracking. Stallman and Sussman, in the context of circuit analysis, with "dependency-directed backtracking" [67], Gaschnig with "backjumping" [33], and Bruynooghe with "intelligent backtracking" [8] all addressed this problem. These methods in some sense remember the reasons for failure in order to backtrack over legitimate 'culprits'. Stallman and Sussman went further by "learning" new constraints ("nogoods") from failure, which could be used to prune further search. Gaschnig used another form of memory in his "backmarking" algorithm to avoid redundant checking for consistency when backtracking [32].

2.2.5 Analysis

While it was recognized early on that solving CSPs was in general NP-hard, a variety of analytical techniques were brought to bear to evaluate, predict or compare algorithm performance and relate problem complexity to problem structure. In particular, there are tradeoffs to evaluate between the effort required to avoid search, e.g. by exercising more intelligent control or carrying out more inference, and the reduction in search effort obtained.

Knuth [46] and Purdom [63] used probing techniques to estimate the efficiency of backtrack programs. Haralick and Elliott carried out a statistical analysis [35], which was refined by Nudel [62] to compute "expected complexities" for classes of problems defined by basic problem parameters. Brown and Purdom investigated average time behavior [7, 64]. Mackworth and Freuder carried out algorithmic complexity analyses of worst case behavior for various tractable propagation algorithms [55]. They showed the time com-

plexity for arc consistency to be linear in the number of constraints, settling an unresolved issue. This result turned out to be important for constraint programming languages that used arc consistency as a primitive operation [56]. Of course, experimental evaluation was common, though in the early days there was perhaps too much reliance on the n-queens problem, and too little understanding of the potential pitfalls of experiments with random problems.

Problem complexity can be related to problem structure. Seidel [66] developed a dynamic programming synthesis algorithm, using a decomposition technique based on graph cutsets, that related problem complexity to a problem parameter that he called "front length". Freuder [27] proved that problems with tree-structured constraint graphs were tractable by introducing the structural concept of the "width" of a constraint graph, and demonstrating a connection between width and consistency level that ensured that tree-structured problems could be solved with backtrack-free search after arc consistency preprocessing. He subsequently related complexity to problem structure in terms of maximal biconnected components [28] and stable sets [30].

2.3 Conclusions

This chapter has not been a complete history, and certainly not an exhaustive survey. We have focused on the major themes of the early period, but it is worth noting that many very modern sounding topics were also already appearing at this early stage. For example, even in 1965 Golomb and Baumert were making allusions to symmetry and problem reformulation.

Golomb and Baumert concluded in 1965 [34]:

> Thus the success or failure of backtrack often depends on the skill and ingenuity of the programmer in his ability to adapt the basic methods to the problem at hand and in his ability to reformulate the problem so as to exploit the characteristics of his own computing device. That is, backtrack programming (as many other types of programming) is somewhat of an art.

As the rest of this handbook will demonstrate, much progress has been made in making even more powerful methods available to the constraint programmer. However, constraint programming is still "somewhat of an art". The challenge going forward will be to make constraint programming more of an engineering activity and constraint technology more transparently accessible to the non-programmer.

Acknowledgements

We are grateful to Peter van Beek for all his editorial comments, help and support during the preparation of this chapter. This material is based upon works supported by the Science Foundation Ireland under Grant No. Grant 00/PI.1/C075 and by the Natural Sciences and Engineering Research Council of Canada. Alan Mackworth is supported by a Canada Research Chair in Artificial Intelligence.

Bibliography

[1] H. G. Barrow and J. M. Tenenbaum. MSYS: A system for reasoning about scenes. In *SRI AICenter*, 1975.

[2] J. R. Bitner and E. M. Reingold. Backtrack programming techniques. *Comm. ACM*, 18:651–656, 1975.

[3] D. G. Bobrow and B. Raphael. New programming languages for artificial intelligence research. *ACM Computing Surveys*, 6(3):153–174, Sept. 1974.

[4] A. Borning. ThingLab – an object-oriented system for building simulations using constraints. In R. Reddy, editor, *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 497–498, Cambridge, MA, Aug. 1977. William Kaufmann. ISBN 0-86576-057-8.

[5] A. Borning. Thinglab: A constraint-oriented simulation laboratory. Report CS-79-746, Computer Science Dept., Stanford University, CA, 1979.

[6] A. Brown. Qualitative knowledge, casual reasoning and the localization of failures. Technical Report AITR-362, MIT Artificial Intelligence Laboratory, Nov. 6 1976. URL `http://dspace.mit.edu/handle/1721.1/6921`.

[7] C. A. Brown and P. W. Purdom Jr. An average time analysis of backtracking. *SIAM J. Comput.*, 10:583–593, 1981.

[8] M. Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters*, 12:36–39, 1981.

[9] R. M. Burstall. A program for solving word sum puzzles. *Computer Journal*, 12(1): 48–51, Feb. 1969.

[10] M. B. Clowes. On seeing things. *Artificial Intelligence*, 2:79–116, 1971.

[11] J. Cohen. Constraint logic programming languages. *CACM*, 33(7):52–68, July 1990. ISSN 0001-0782. URL `http://www.acm.org/pubs/toc/Abstracts/0001-0782/79209.html`.

[12] A. Colmerauer. Prolog II reference manual and theoretical model. Technical report, Groupe d'Intelligence Arificielle, Univeristé d'Aix-Marseille II, Luminy, Oct. 1982.

[13] A. Colmerauer. Prolog and infinite trees. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 231–251. Academic Press, 1982.

[14] A. Colmerauer and P. Roussel. The birth of Prolog. In R. L. Wexelblat, editor, *Proceedings of the Conference on History of Programming Languages*, volume 28(3) of *ACM Sigplan Notices*, pages 37–52, New York, NY, USA, Apr. 1993. ACM Press. ISBN 0-89791-570-4.

[15] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, 1960.

[16] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Comm. ACM*, 5:394–397, 1962.

[17] J. de Kleer. Local methods for localizing faults in electronic circuits. Technical Report AIM-394, MIT Artificial Intelligence Laboratory, Nov. 6 1976. URL `http://dspace.mit.edu/handle/1721.1/6921`.

[18] J. de Kleer and G. J. Sussman. Propagation of constraints applied to circuit synthesis. Technical Report AIM-485, MIT Artificial Intelligence Laboratory, Sept. 6 1978. URL `http://hdl.handle.net/1721.1/5745`.

[19] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[20] Y. Descotte and J.-C. Latombe. GARI : A problem solver that plans how to machine

mechanical parts. In *International Joint Conference on Artificial Intelligence (IJCAI '81)*, pages 766–772, 1981.

[21] C. M. Eastman. Automated space planning. *Artificial Intelligence*, 4(1):41–64, 1973.

[22] E. W. Elcock. Absys: the first logic programming language - A retrospective and a commentary. *Journal of Logic Programming*, 9(1):1–17, July 1990.

[23] R. E. Fikes. REF-ARF: A system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.

[24] J. P. Fillmore and S. G. Williamson. On backtracking: A combinatorial description of the algorithm. *SIAM Journal on Computing*, 3(1):41–55, Mar. 1974.

[25] M. S. Fox, B. P. Allen, and G. Strohm. Job-shop scheduling: An investigation in constraint-directed reasoning. In *AAAI82, Proceedings*, pages 155–158, 1982.

[26] E. C. Freuder. Synthesizing constraint expressions. *Comm. ACM*, 21:958–966, 1978.

[27] E. C. Freuder. A sufficient condition for backtrack-free search. *J. ACM*, 29:24–32, 1982.

[28] E. C. Freuder. A sufficient condition for backtrack-bounded search. *J. ACM*, 32: 755–761, 1985.

[29] E. C. Freuder and A. K. Mackworth. Introduction to the special volume on constraint-based reasoning. *Artificial Intelligence*, 58:1–2, 1992.

[30] E. C. Freuder and M. J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 1076–1078, Los Angeles, 1985.

[31] J. Gaschnig. A constraint satisfaction method for inference making. In *Proc. 12th Annual Allerton Conf. on Circuit System Theory*, pages 866–874, U. Illinois, 1974.

[32] J. Gaschnig. A general backtracking algorithm that eliminates most redundant tests. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, page 457, Cambridge, Mass., 1977.

[33] J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Second Canadian Conference on Artificial Intelligence*, pages 268–277, Toronto, 1978.

[34] S. Golomb and L. Baumert. Backtrack programming. *J. ACM*, 12:516–524, 1965.

[35] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[36] R. M. Haralick and L. G. Shapiro. The consistent labeling problem: Part I. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 1(2):173–184, Apr. 1979.

[37] R. M. Haralick and L. G. Shapiro. The consistent labeling problem: Part II. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 2(3):193–203, May 1980.

[38] R. M. Haralick, L. S. Davis, A. Rosenfeld, and D. L. Milgram. Reduction operations for constraint satisfaction. *Inf. Sci*, 14(3):199–219, 1978. URL `http://dx.doi.org/10.1016/0020-0255(78)90043-9`.

[39] P. J. Hayes. Computation and deduction. In *Proc. 2nd International Symposium on Mathematical Foundations of Computer Science*, pages 105–118. Czechoslovakian Academy of Sciences, 1973.

[40] C. Hewitt. PLANNER: A language for proving theorems in robots. In *Proceedings of the First International Joint Conference on Artificial Intelligence*, pages 295–301, Bedford, MA., 1969. Mitre Corporation.

[41] D. A. Huffman. Impossible objects as nonsense sentences. In B. Meltzer and D. Michie, editors, *Machine Intelligence 6*, pages 295–323. Edinburgh Univ. Press,

1971.

[42] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 111–119, München, 1987.

[43] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19(20):503–581, 1994.

[44] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) language and system. *TOPLAS*, 14(3):339–395, July 1992. ISSN 0164-0925. URL `http://www.acm.org/pubs/toc/Abstracts/0164-0925/129398.html`.

[45] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[46] D. E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29:121–136, 1975.

[47] R. A. Kowalski. Predicate logic as a programming language. *Proc. IFIP '74*, pages 569–574, 1974.

[48] R. A. Kowalski. A proof procedure using connection graphs. *J. ACM*, 22(4):572–595, 1975.

[49] J.-L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.

[50] D. H. Lehmer. Combinatorial problems with digital computers. In *Proc. of the Fourth Canadian Math. Congress*, pages 160–173, 1957.

[51] E. Lucas. *Récréations Mathématiques*. Gauthier-Villars, Paris, 1891.

[52] A. K. Mackworth. Interpreting pictures of polyhedral scenes. *Artificial Intelligence*, 4:121–137, 1973.

[53] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8: 99–118, 1977.

[54] A. K. Mackworth. The logic of constraint satisfaction. *Artificial Intelligence*, 58: 3–20, 1992.

[55] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 247, 1985.

[56] A. K. Mackworth and E. C. Freuder. The complexity of constraint satisfaction revisited. *Artificial Intelligence*, 59:57–62, 1993.

[57] A. K. Mackworth, J. A. Mulder, and W. S. Havens. Hierarchical arc consistency: Exploiting structured domains in constraint satisfaction problems. *Computational Intelligence*, 1:118–126, 1985.

[58] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[59] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Inform. Sci.*, 19:229–250, 1979.

[60] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inform. Sci.*, 7:95–132, 1974.

[61] N. J. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, New York, 1971.

[62] B. Nudel. Consistent-labeling problems and their algorithms: Expected-complexities and theory-based heuristics. *Artificial Intelligence*, 21:135–178, 1983.

[63] P. W. Purdom Jr. Tree size by partial backtracking. *SIAM J. Comput.*, 7:481–491, 1978.

[64] P. W. Purdom Jr. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21:117–133, 1983.

[65] A. Rosenfeld, R. A. Hummel, and S. W. Zucker. Scene labelling by relaxation operations. *IEEE Trans. on Systems, Man, and Cybernetics*, SMC-6:420–433, 1976.

[66] R. Seidel. A new method for solving constraint satisfaction problems. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 338–342, Vancouver, 1981.

[67] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[68] M. Stefik. Planning with constraints (MOLGEN: Part 1). *Artificial Intelligence*, 16: 111–140, 1981.

[69] M. J. Stefik. Planning and meta-planning (MOLGEN: Part 2). *Artificial Intelligence*, 16:141–169, 1981.

[70] G. Sussman and T. Winograd. Micro-planner reference manual. Technical Report AIM-203, MIT Artificial Intelligence Laboratory, July 1 1970. URL `ftp://publications.ai.mit.edu/ai-publications/0-499/AIM-203.ps;ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-203.pdf`.

[71] G. J. Sussman and R. M. Stallman. Heuristic techniques in computer-aided circuit analysis. *IEEE Trans. on Circuits and Systems*, CAS-22(11), 1975.

[72] G. J. Sussman and G. L. Steele. CONSTRAINTS: a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14, 1980.

[73] I. E. Sutherland. SKETCHPAD: A man-machine graphical communications system. Technical Report 296, MIT, Lincoln Laboratory, Jan. 1963.

[74] J. M. Tenenbaum and H. G. Barrow. Experiments in interpretation-guided segmentation. *Artif. Intell*, 8(3):241–274, 1977.

[75] J. R. Ullmann. Associating parts of patterns. *Information and Control*, 9(6):583–601, 1966.

[76] J. R. Ullmann, R. M. Haralick, and L. G. Shapiro. Computer architecture for solving consistent labelling problems. *Comput. J*, 28(2):105–111, 1985.

[77] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[78] R. L. Walker. An enumerative technique for a class of combinatorial problems. In *Combinatorial Analysis, Proceedings of Symposium in Applied Mathematics, Vol X, Amer. Math. Soc., Providence, RI, USA*, pages 91–94, 1960.

[79] D. Waltz. Understanding line drawings of scenes with shadows. In P. H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.

[80] M. V. Wilkes. Constraint-type statements in programming languages. *CACM*, 7(10): 587–588, 1964. URL `http://doi.acm.org/10.1145/364888.364967`.