Backtracking algorithms for constraint satisfaction problems*

Rina Dechter and Daniel Frost
Department of Information and Computer Science
University of California, Irvine
Irvine, California, USA 92697-3425
{dechter,frost}@ics.uci.edu

September 17, 1999

Abstract

Over the past twenty five years many backtracking algorithms have been developed for constraint satisfaction problems. This survey describes the basic backtrack search within the search space framework and then presents a number of improvements developed in the past two decades, including look-back methods such as backjumping, constraint recording, backmarking, and look-ahead methods such as forward checking and dynamic variable ordering.

1 Introduction

Constraint networks have proven successful in modeling mundane cognitive tasks such as vision, language comprehension, default reasoning, and abduction, as well as specialized reasoning tasks including diagnosis, design, and temporal and spatial reasoning. The constraint paradigm is a generalization of propositional logic, in that variables may be assigned values from a set with any number of elements, instead of only TRUE and FALSE. Flexibility in the number of values can improve the ease and naturalness with which interesting problems are modeled.

The contribution of this paper is a survey of several approaches to solving constraint satisfaction problems, focusing on the backtracking algorithm and its variants, which form the basis for many constraint solution procedures. We provide on a careful exposition of each algorithm, its theoretical underpinnings, and its relationship to similar algorithms. Worst-case bounds on time and space usage are developed for each algorithm. In addition to the survey,

*This work was partially supported by NSF grant IRI-9157636, Air Force Office of Scientific Research grant AFOSR F49620-96-1-0224, Rockwell MICRO grants ACM-20775 and 95-043.

the paper makes several original contributions in formulation and analysis of the algorithms. In particular the look-back backjumping schemes are given a fresh exposition through comparison of the three primary variants, Gashnig's, graph-based and conflict-directed. We show that each of these backjumping algorithms is optimal relative to its information gathering process. The complexity of several schemes as a function of parameters of the constraint-graph are explicated. Those include backjumping complexity as a function of the depth of the DFS traversal of the constraint graph, learning algorithms as a function of the induced-width, and look-ahead methods such as partial-lookahead as a function of the size of the cycle-cutset of the constraint graph.

The remainder of the paper is organized as follows. Section 2 defines the constraint framework and provides an overview of the basic algorithms for solving constraint satisfaction problems. In Section 3 we present the backtracking algorithm. Sections 4 and 5 survey and analyze look-back methods such as backjumping and learning schemes while Section 6 surveys look-ahead methods. Finally, in Section 7 we present a brief historical review of the field. Previous surveys on constraint processing as well as on backtracking algorithms can be found in [Dec92, Mac92, Kum92, Tsa93, KvB97].

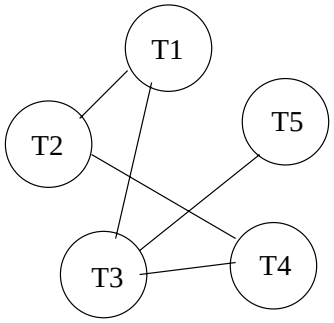
2 The constraint framework

2.1 Definitions

A *constraint network* or *constraint satisfaction problem* (CSP) is a set of n variables $X = \{x_1, \ldots, x_n\}$, a set of value domains D_i for each variable x_i, and a set of constraints or relations. Each value domain is a finite set of values, one of which must be assigned to the corresponding variable. A *constraint* R_S over $S \subseteq X$ is a subset of the cartesian product of the domains of the variables in S. If $S = \{x_{i_1}, \ldots, x_{i_r}\}$, then $R_S \subseteq D_{i_1} \times \cdots \times D_{i_r}$. S is called the scope of R_S. A *nogood* is a particular assignment of values to a subset of variables which is not permitted. In a *binary* constraint network all constraints are defined over pairs of variables. A *constraint graph* associates each variable with a node and connects any two nodes whose variables appear in the same scope.

A variable is called *instantiated* when it is assigned a value from its domain; otherwise it is *uninstantiated*. By $x_i = a$ or by (x_i, a) we denote that variable x_i is instantiated with value a from its domain. A *partial instantiation* or *partial assignment* of a subset of X is a tuple of ordered pairs $((x_1, a_1), ..., (x_i, a_i))$, frequently abbreviated to $(a_1, ..., a_i)$ or $\vec{a_i}$ when the order of the variables is known.

Let Y and S be sets of variables, and let $\vec{y}$ be an instantiation of the variables in Y. We denote by $\vec{y_S}$ the tuple consisting of only the components of $\vec{y}$ that correspond to the variables in S. A partial instantiation $\vec{y}$ satisfies a constraint R_S iff $\vec{y_S} \in R_S$. *[Rina, the next sentence is new.]* $\vec{y}$ is *consistent* if $\vec{y}$ satisfies all constraints $R_T, T \subseteq Y$. A consistent partial instantiation is also called a *partial solution*. A *solution* is an instantiation of all the variables that is consistent.



Unary constraint
T4 $\neq$ 2:00
Binary constraints
T1, T2: {(1:00,2:00), (1:00,3:00), (2:00,1:00) (2:00,3:00), (3:00,1:00), (3:00,2:00)}
T1, T3: {(2:00,1:00), (3:00,1:00), (3:00,2:00)}
T2, T4: {(1:00,2:00), (1:00,3:00), (2:00,1:00) (2:00,3:00), (3:00,1:00), (3:00,2:00)}
T3, T4: {(1:00,2:00), (1:00,3:00), (2:00,3:00)}
T3, T5: {(2:00,1:00), (3:00,1:00), (3:00,2:00)}

Figure 1: The constraint graph and constraint relations of the scheduling problem in Example 1.

Example 1. The constraint framework is useful for expressing and solving scheduling problems. Consider the problem of scheduling 5 tasks T1, T2, T3, T4, T5, each of which takes 1 hour to complete. The tasks may start at 1:00, 2:00, or 3:00. Any number of tasks can be executed simultaneously, subject to the restrictions that T1 must start after T3, T3 must start before T4 and after T5, T2 cannot execute at the same time as T1 or T4, and T4 cannot start at 2:00.

With five tasks and three time slots, we can model the scheduling problem by creating five variables, one for each task, and giving each variable the domain {1:00, 2:00, 3:00}. Another equally valid approach is to create three variables, one for each starting time, and to give each of these variables a domain which is the powerset of {T1, T2, T3, T4, T5}. Adopting the first approach, the problem's constraint graph is shown in Figure 1. The constraint relations are shown on the right of the figure.

2.2 Constraint algorithms

Once a problem of interest has been formulated as a constraint satisfaction problem, it can be attacked with a general purpose constraint algorithm. Many CSP algorithms are based on the principles of search and deduction; more sophisticated algorithms often combine both principles. In this section we briefly survey the field of CSP algorithms.

2.2.1 Search - backtracking

The term *search* is used to represent a large category of algorithms which solve problems by guessing an operation to perform or an action to take, possibly with the aid of a heuristic [Nil80, Pea84]. A good guess results in a new state that is nearer to a goal. If the operation does not result in progress towards the goal (which may not be apparent until later in the search), then the operation can be retracted and another guess made.

For CSPs, search is exemplified by the backtracking algorithm. Backtracking search uses the operation of assigning a value to a variable, so that the current partial solution is extended. When no acceptable value can be found, the previous assignment is retracted, which is called a *backtrack*. In the worst case the backtracking algorithm requires exponential time in the number of variables, but only linear space. The algorithm was first described more than a century ago, and since then has been reintroduced several times [BR75].

2.2.2 Deduction - constraint propagation

To solve a problem by deduction is to apply reasoning to transform the problem into an equivalent but more explicit form. In the CSP framework the most frequently used type of deduction is known as constraint propagation or consistency enforcing [Mon74, Mac77, Fre82]. These procedures transform a given constraint network into an equivalent yet more explicit one by deducing new constraints, tightening existing constraints, and removing values from variable domains. In general, a consistency enforcing algorithm will make any partial solution of a subnetwork extendable to some surrounding network by guaranteeing a certain degree of local consistency, defined as follows.

A constraint network is 1-consistent if the values in the domain of each variable satisfy the network's unary constraints (that is, constraints which pertain to a single variable). A network is *k-consistent*, $k \geq 2$, iff given any consistent partial instantiation of any $k-1$ distinct variables, there exists a consistent instantiation of any kth additional variable [Fre78]. The terms *node-*, *arc-*, and *path-consistency* [Mac77] correspond to 1-, 2-, and 3-consistency, respectively. Given an ordering of the variables, the network is *directional* k-consistent iff any subset of $k-1$ variables is k-consistent relative to variables that succeed the $k-1$ variables in the ordering [DP87]. A problem that is k-consistent for all k is called *globally* consistent.

A variety of algorithms have been developed for enforcing local consistency [MF85, MH86, Coo90, VHDT92, DP87]. For example, arc-consistency algorithms delete certain values from the domains of certain variables, to ensure that each value in the domain of each variable is consistent with at least one value in the domain of each other variable. Path-consistency is achieved by introducting new constraints or nogoods which disallow certain pairs of values.

Constraint propagation can be used as a CSP solution procedure, although doing so is usually not practical. If global consistency can be enforced, then one or more solutions can easily be found in the transformed problem, without backtracking. However, enforcing k-consistency requires in general exponential time and exponential space in k [Coo90], and so in practice only local consistency, with $k \leq 3$, is used.

In Example 1, enforcing 1-consistency on the network will result in the value 2:00 being removed from the domain of T4, since that value is incompatable with a unary constraint. Enforcing 2-consistency will cause several other domain values to be removed. For instance, the constraint between T1 and T3 means that if T1 is scheduled for 1:00, there is no possible time for T3, since it must

occur before T1. Therefore, an arc-consistency algorithm will, among other actions, remove 1:00 from the domain of T1.

Algorithms that enforce local consistency can be performed as a preprocessing step in advance of a search algorithm. In most cases, backtracking will work more efficiently on representations that are as explicit as possible, that is, those having a high level of local consistency. The value of the tradeoff between the effort spent on pre-processing and the reduced effort spent on search has to be assessed experimentally, and is dependent on the character of the problem instance being solved [DM94]. Varying levels of consistency-enforcing can also be interleaved with the search process, and doing so is the primary way consistency enforcing techniques are incorporated into constraint programming languages [JL94].

2.2.3 Other constraint algorithms

In addition to backtracking search and constraint propagation, other approaches to solving constraint problems include *stochastic local search* and *structure-driven* algorithms. Stochastic methods typically move in a hill-climbing manner augmented with random steps in the space of complete instantiations [MJPL90]. In the CSP community interest in stochastic approaches was sparked by the success of the GSAT algorithm [SLM92]. Structure-driven algorithms, which employ both search and consistency-enforcing components, emerge from an attempt to characterize the topology of constraint problems that are tractable. *Tractable classes* of constraint networks are generally recognized by realizing that for some problems enforcing low-level consistency (in polynomial time) guarantees global consistency. The basic graph structure that supports tractability is a tree [MF85]. In particular, enforcing 2-consistency on a tree-structured binary CSP network ensures global consistency along some ordering of the variables.

3 Backtracking

A simple algorithm for solving constraint satisfaction problems is *backtracking*, which traverses the search graph in a depth-first manner. The order of the variables can be fixed in advance or determined at run time. The backtracking algorithm maintains a partial solution that denotes a state in the algorithm's search space. Backtracking has three phases: a forward phase in which the next variable in the ordering is selected; a phase in which the current partial solution is extended by assigning a consistent value, if one exists, to the next variable; and a backward phase in which, when no consistent value exists for the current variable, focus returns to the variable prior to the current variable.

Figure 2 describes a basic backtracking algorithm. As presented in Figure 2, the backtracking algorithm returns at most a single solution, but it can easily be modified to return all solutions, or a desired number. The algorithm employs a series of mutable value domains D'_i such that each $D'_i \subseteq D_i$. D'_i holds the subset

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.