TECHNICAL REPORT
R-121-AI

ELSEVIER

Artificial Intelligence 68 (1994) 211–241

Artificial Intelligence

Experimental evaluation of preprocessing algorithms for constraint satisfaction problems

Rina Dechter[a,*], Itay Meiri[b]

[a]Information and Computer Science Department, University of California, Irvine, CA 29717–3425, USA

[b]Cognitive Systems Laboratory, Computer Science Department, University of California, Los Angeles, CA 90024, USA

Received February 1992; revised July 1993

Abstract

This paper presents an experimental evaluation of two orthogonal schemes for preprocessing constraint satisfaction problems (CSPs). The first of these schemes involves a class of local consistency techniques that includes *directional arc consistency*, *directional path consistency*, and *adaptive consistency*. The other scheme concerns the prearrangement of variables in a linear order to facilitate an efficient search. In the first series of experiments, we evaluated the effect of each of the local consistency techniques on *backtracking* and *backjumping*. Surprisingly, although *adaptive consistency* has the best worst-case complexity bounds, we have found that it exhibits the worst performance, unless the constraint graph was very sparse. *Directional arc consistency* (followed by either *backjumping* or *backtracking*) and *backjumping* (without any preprocessing) outperformed all other techniques: moreover, the former dominated the latter in computationally intensive situations. The second series of experiments suggests that *maximum cardinality* and *minimum width* are the best preordering (i.e., static ordering) strategies, while *dynamic search rearrangement* is superior to all the preorderings studied.

1. Introduction

Constraint satisfaction tasks belong to the class of NP-complete problems and, as such, normally lack realistic measures of performance. Worst-case analysis, because it depends on extreme cases, may yield an erroneous view of typical performance of algorithms used in practice. Average-case analysis, on the other

* Corresponding author. E-mail: dechter@ics.uci.edu.

hand, is extremely difficult and is highly sensitive to simplifying theoretical assumptions. Thus, theoretical analysis must be supplemented by experimental studies.

The most thorough experimental studies reported so far include Gaschnig's comparisons of *backjumping*, *backmarking* and constraint propagation [12], Haralick and Elliot's study of look-ahead strategies [14], Brown and Purdom's experiments with dynamic variable orderings [21, 22], and, more recently, Dechter's experiments with structure-based techniques [3], and Prosser's hybrid tests with *backjumping* and *forward-checking* strategies [20]. Additional studies were reported in [6, 13, 23, 26, 27].

Experimental studies are most informative when conducted on a "representative" set of problems from one's own domain of application. However, this is very difficult to effect. Real-life problems are often too large or too ill-defined to suit a laboratory manipulation. A common compromise is to use either randomly generated problems or canonical examples (e.g., *n*-queens, crossword puzzles, and graph-coloring problems). Clearly, conclusions drawn from such experiments reflect only on problem domains that resemble the experimental conditions and caution must be exercised when generalizing to real-life problems. Such experiments do reveal the crucial parameters of a problem domain, and so help establish the relative usefulness of various algorithms.

Our focus in this paper is on algorithms whose performance, as revealed by worst-case analysis, is dependent on the topological structure of the problem. Our aim is to uncover whether the same dependency is observed empirically and to investigate the extent to which worst-case bounds predict actual performance. Our primary concern is with preprocessing algorithms and their effect on *backtracking*'s performance. Since our preprocessing algorithms are dependent on a static ordering of the variables they invite different heuristics for variable ordering. We tested the effect of such orderings on the preprocessing algorithms as well as on regular *backtracking* and *backjumping*.

We organized our experimental results into two classes. The first class concerns consistency enforcing algorithms, which transform a given constraint network into a more explicit representation. On this more explicit representation, any *backtracking* algorithm is guaranteed to encounter fewer deadends [16]. Since these algorithms are polynomial while *backtracking* is exponential, and since they always improve search, one may hastily conclude that they should always be exercised. Our aim was to test this hypothesis. The three consistency enforcing algorithms tested are *directional arc consistency* (*DAC*), *directional path consistency* (*DPC*), and *adaptive consistency* (*ADAPT*) [6]. These algorithms represent increasing levels of preprocessing effort as well as an increasing improvement in subsequent search. Although *DAC* and *DPC*, whose complexities are quadratic and cubic, respectively, can still be followed by exponential search (in the worst case), *ADAPT* is guaranteed to yield a solution in time bounded by $O(\exp(W^*))$, where W^* is a parameter reflecting the sparseness of the network.

Our results show, contrary to predictions based on worst-case analysis, that the average complexity of *backtracking* on our randomly generated problems is far from exponential. Indeed the preprocessing performed by the most aggressive scheme, *ADAPT*, did not pay off unless the graph was very sparse, in spite of its theoretical superiority to *backtracking*. On the other hand, the least aggressive scheme, *DAC*, came out as a winner in computationally intensive cases. Apparently, *DAC* performs just the desired amount of preprocessing. Additionally, while *ADAPT* showed that its average complexity is exponentially dependent on W^*, the dependence of all other schemes on W^* seems to be quite weak or even non-existent.

In the second class we report the effect of various static ordering strategies on *backtracking* and *backjumping* without preprocessing. Static orderings, in contrast to dynamic orderings, are appealing in that they do not require any overhead during search. We tested four static heuristic orderings, *minimum width* (*MIN*), *maximum degree* (*DEG*), *maximum cardinality* (*CARD*), and *depth-first search* (*DFS*). Those orderings are advised when analyzing their effect on the preprocessing algorithms *ADAPT* and even *DPC* as they yield a low W^*. Although no worst-case complexity ties *backtracking* or *backjumping* to W^*, we nevertheless wanted to discover whether a correlation exists, and which of these static orderings yields a better average search. Lastly, in order to relate our experiments with other experiments reported in the literature, we compared our static ordering with one dynamic ordering, *dynamic search rearrangement* (*DSR*) [21]. We tested two implementation styles of *DSR*, presenting a tradeoff between space and time overhead.

Our results show that *minimum width* and *maximum cardinality* clearly dominated the *maximum degree* and *depth-first search* orderings. However, the exact relationship between the first two is still unclear. While *dynamic ordering* was only second or third best when implemented in a brute-force way it outperformed all static orderings when a more careful implementation that restricted its time overhead was introduced.

The remainder of the paper is organized as follows: we review the constraint network model and general background in Section 2, present the tested algorithms in Section 3, describe the experimental design in Section 4, discuss the results in Section 5, and provide a summary and concluding remarks in Section 6.

2. Constraint processing techniques

A **constraint network** (CN) consists of a set of **variables** $X = \{X_1, \ldots, X_n\}$, each associated with a **domain** of discrete values $D_1, \ldots, D_n$, and a set of **constraints** $\{C_1, \ldots, C_t\}$. Each constraint is a relation defined on a subset of variables. The tuples of this relation are all the simultaneous value assignments to this variable subset which, as far as this constraint alone is concerned, are legal.[1]

[1] This does not mean that the actual representation of any constraint is necessarily in the form of its defining relation, rather the relation can in principle be generated using the constraint's specification without the need to consult other constraints in the network.

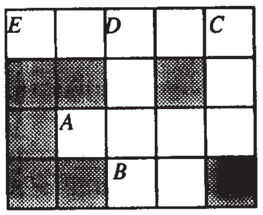
Formally, constraint C_i has two parts: a subset of variables $S_i = \{X_{i_1}, \ldots, X_{i_{j(i)}}\}$ on which it is defined, called a **constraint-subset**, and a **relation** rel_i defined over S_i: $rel_i \subseteq D_{i_1} \times \cdots \times D_{i_{j(i)}}$. The **scheme** of a CN is the set of its constraint subsets, namely, $scheme(CN) = \{S_1, S_2, \ldots, S_t\}$, $S_i \subseteq X$. An assignment of a unique domain value to each member of some subset of variables is an **instantiation**. An instantiation is a **solution** only if it satisfies *all* the constraints. The **set of all solutions** is a relation ρ defined on the set of all variables. Formally,

$$\rho = \{(X_1 = x_1, \ldots, X_n = x_n) \mid \forall S_i \in scheme,\ \Pi_{S_i}\rho \subseteq rel_i\}\ , \tag{1}$$

where $\Pi_X\rho$ is the projection of relation ρ over a subset of its variables X, namely it is the set of all subtuples over X that can be extended to a full tuple in ρ.

A CN may be associated with a constraint graph in which nodes represent variables and arcs connect variables that appear in the same constraint. For example, the CN depicted in Fig. 1(a) represents a crossword puzzle. The variables are E, D, C, A, and B. The scheme is $\{ED, EC, CA, AD, DB\}$. For instance, the pair DE is in the scheme since the word associated with D and the word associated with E share a letter. The constraint graph is given in Fig. 1(b).

Typical tasks defined on a CN are determining whether a solution exists, finding one solution or the set of all solutions, and establishing whether an instantiation of a subset of variables is part of a global solution. Collectively, these tasks are known as **constraint satisfaction problems** (CSPs).



$D_E = \{\texttt{hoses,laser,sheet,snail,steer}\}$
$D_A = D_D = \{\texttt{hike,aron,keet,earn,same}\}$
$D_C = \{\texttt{run,sun,let,yes,eat,ten}\}$
$D_B = \{\texttt{no,be,us,it}\}$

$C_{AB} = \{(\texttt{hoses,same}),(\texttt{laser,same}),(\texttt{sheet,earn}),$
$(\texttt{snail,aron}),(\texttt{steer,earn})\}$

(a)

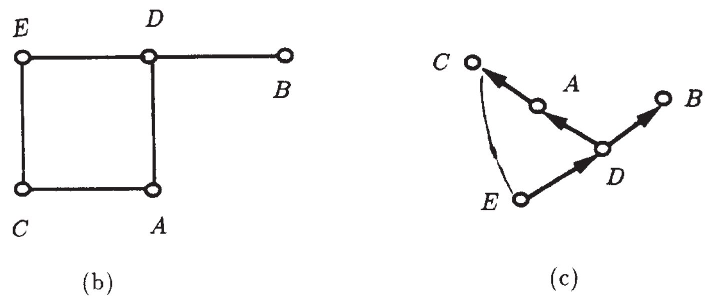


Fig. 1. (a) A crossword puzzle (D denotes domains of variables, C_{AB} is the constraint between variables A and B), (b) its CN representation, and (c) a depth-first search preordering.

Techniques used in processing constraint networks can be classified into three categories: (1) **search algorithms**, for systematic exploration of the space of all solutions, which all have backtracking as their basis; (2) **consistency enforcing algorithms**, that enforce consistency on small parts of the network, and (3) **structure-driven algorithms**, which exploit the topological features of the network to guide the search. Hybrids of these techniques are also available. For a detailed survey of constraint processing techniques, see [4, 15].

Backtracking traverses the search space in a depth-first fashion. The algorithm typically considers the variables in some order. It systematically assigns values to variables until either a solution is found or the algorithm reaches a **deadend**, where a variable has no value consistent with previous assignments. In this case the algorithm backtracks to the most recent instantiation, changes the assigned value, and continues. It is well known that the worst-case running time of *backtracking* is exponential.

Improving the efficiency of *backtracking* amounts to reducing the size of the search space it expands. Two types of procedures were developed: preprocessing algorithms that are employed prior to performing the search, and dynamic algorithms that are used during the search.

The preprocessing algorithms include a variety of **consistency enforcing algorithms** [9, 16, 18]. These algorithms transform a given CN into an equivalent, yet more explicit form, by deducing new constraints to be added to the network. Essentially, a consistency enforcing algorithm makes a small subnetwork consistent relative to its surrounding constraints. For example, the most basic consistency algorithm, called arc consistency or 2-consistency (also known as constraint propagation or constraint relaxation), ensures that any legal value in the domain of a single variable has a legal match in the domain of any other variable. Path consistency (or 3-consistency) ensures that any consistent solution to a two-variable subnetwork is extensible to any third variable, and, in general, i-consistency algorithms guarantee that any locally consistent instantiation of $i-1$ variables is extensible to any ith variable. The algorithms, *DAC*, *DPC*, and *ADAPT* are all restricted (because they take into account the direction in which *backtracking* instantiates the variables) versions of these consistency enforcing algorithms.

The preprocessing algorithms also include algorithms for ordering the variables prior to search. Several heuristics for **static orderings** have been proposed [7, 10]. The heuristics used in this paper—*minimum width*, *maximum cardinality*, *maximum degree*, and *depth-first search*—follow the intuition that tightly constrained variables should be instantiated first.

Strategies that dynamically improve the pruning power of backtracking can be classified as either **look-ahead schemes** or **look-back schemes**. Look-ahead schemes are invoked whenever the algorithm is about to assign a value to the next variable. Some schemes, such as *forward-checking*, use constraint propagation [14, 28] to predict the way in which the current instantiation restricts future assignments of values to variables. An example of a look-ahead scheme is *dynamic search rearrangement*, which decides what variable to instantiate next

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.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase® Smarter legal research.