University of Alberta

Improvements to and estimating the cost of backtracking algorithms for constraint satisfaction problems

by

Jonathan Sillito ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2000

0-612-59878-0

University of Alberta

Library Release Form

Name of Author: Jonathan Sillito

Title of Thesis: Improvements to and estimating the cost of backtracking algorithms for constraint satisfaction problems

Degree: Master of Science

Year this Degree Granted: 2000

[illegible]

Jonathan Sillito
APT 210 9710 82 Ave
Edmonton, Alberta
Canada, T6E 1Y5

Date: July 28, 2000

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Improvements to and estimating the cost of backtracking algorithms for constraint satisfaction problems** submitted by Jonathan Sillito in partial fulfillment of the requirements for the degree of **Master of Science**.

[illegible]

Peter van Beek

[illegible]

Bruce Cockburn

[illegible]

Joe Culberson

[illegible]

Duane Szafron

Date: 26 June 2000

Abstract

Arc consistency algorithms represent an important class of constraint satisfaction algorithms. General arc consistency 3 (gac3) is the most common algorithm in this class. In 1997, Bessiere and Regin proposed general arc consistency (gac7) as a more sophisticated general arc consistency algorithm. We show how to incorporate Bessiere and Regin's proposal into backtracking search. We also perform the first evaluation of this algorithm.

In this thesis we also present work in the area of estimating the cost of solving constraint satisfaction problems. Some existing work in this area is evaluated. Effective estimation techniques could have applications in areas such as: problem modeling, algorithm selection, and variable ordering. We also develop an estimation technique, based on statistical sampling algorithms by Knuth and Purdom. We demonstrate the usefulness of our estimator as a dynamic variable ordering.

Contents

Chapter 1

Introduction

Constraint programming is a simple, yet powerful paradigm in which problems are represented as constraint satisfaction problems. A constraint satisfaction problem (CSP) is defined by a set of variables, a domain of values for each variable, and a set of constraints over the variables. CSPs are a very natural way to model many interesting problems, including problems in the areas of scheduling and planning.

A constraint satisfaction problem can be solved using a variety of algorithms. The most common complete algorithms used are based on chronological backtracking. Another type of algorithm used in the area of constraint satisfaction is arc consistency algorithms. Often backtracking algorithms incorporate an arc consistency algorithm to enforce arc consistency at each node of the search. Many arc consistency algorithms have been designed specifically for problems with only binary constraints (that is, constraints involving exactly two variables). Arc consistency algorithms that apply, more generally, to problems with constraints of arbitrary arity, are often called *general* arc consistency algorithms. General arc consistency incorporated into backtracking search has been shown to be useful on many problems.

In [18], Mackworth proposed an algorithm, called AC3, for enforcing arc consistency on binary constraint satisfaction problems. In [19], he showed how AC3 could be generalized to handle general CSPs. This, generalized, algorithm is called gac3. More recently, Bessiere and Regin [4] have proposed gac7 as a more sophisticated general arc consistency algorithm. Gac7 has several features that, intuitively, seem to be improvements over the more straight forward gac3 algorithm. In this thesis we show how gac7 can be incorporated into backtracking and we perform the first evaluation of gac7. In particular we compare gac7 with gac3.

In this thesis we also present work in the area of estimating the cost of solving constraint satisfaction problems. In 1975, Knuth [15] developed a sampling algorithm that could take a problem instance, P, and produce an estimate of the cost of solving P using chronological backtracking. In 1983 and 1990, Nadel [24, 22], building on the work of Haralick and Elliott [13], developed, using analytic methods, a series of formulas for predicting the cost of solving a problem instance using backtracking and forward checking. We present an evaluation of these techniques.

In addition to this evaluation, we develop an estimation technique that takes a constraint satisfaction problem instance, P, and an algorithm, A, and returns an estimate of the cost of solving P using A. In particular we apply techniques, similar to those used by Knuth, to backtracking-based algorithms, such as forward checking and arc consistency. We provide a discussion about possible applications of our estimator. We also evaluate the usefulness of our estimator in one of these application areas: dynamic variable orderings.

The thesis is organized as follows. Chapter 2 presents some background information on the topic of constraint satisfaction. Chapter 3 presents our comparison of gac7 with gac3. Chapter 4 presents our work on estimation. Chapter 5 provides suggestions for future work and a summary.

Chapter 2

Background

In this chapter we discuss some necessary background information. In section 2.1 we discuss some basic definitions and introduce the notation we will use. In section 2.2 we introduce four problems that will be used in evaluating the techniques discussed in this thesis. In section 2.3 we introduce some basic techniques and algorithms commonly used in solving constraint satisfaction problems.

2.1 Basic definitions

A *constraint satisfaction problem* P is a triple (X, D, C), where X is a set of n variables $\{x_1, ..., x_n\}$, D is a set of finite domains $\{D(x_1), ..., D(x_n)\}$ on the variables, and C is a set of m constraints over sets of the variables in X. The domain of a variable is a set of possible values that can be assigned to it. A variable, together with a value from its domain, is referred to as a label. A *solution* to P is an assignment of a value $a_i \in D_{x_i}$ to x_i, $1 \leq i \leq n$, that satisfies all of the constraints.

Each constraint $c \in C$ is a constraint over some set of variables in X. This set of variables is known as the scheme of the constraint, and the size of this set is known as the arity of the constraint. If a constraint $c \in C$ is defined on the ordered set of variables $X(c) = (x_1, ..., x_k)$ then c can be defined as a subset of the space $D(x_1) \times ... \times D(x_k)$. This Cartesian product is the space of all possible assignments to the variables in $X(c)$, where each element in the product is a tuple $t = [a_1, ..., a_k]$. The notation $t[x]$ is used to refer to the value corresponding to the variable x in the tuple t. The subset c of this product is referred to as the set of all *satisfying* or *valid* tuples. The ratio

$$\frac{|c|}{|D(x_1) \times ... \times D(x_k)|}$$

is a measure of the tightness (or the tightness ratio) of the constraint. Lower values of this ratio correspond to tighter constraints.

Constraints are sometimes stored explicitly as a set of satisfying tuples, in which case they are said to be stored in *extension* or *extensionally.* More generally, however, constraints can be thought of (and implemented) as a boolean function. This function takes an instantiation of the variables participating in the constraint and returns true if the instantiating variables satisfy the constraint, and false if they do not. Tuples that are not stored in extension are said to be stored *intensionally.*

2.2 Problems

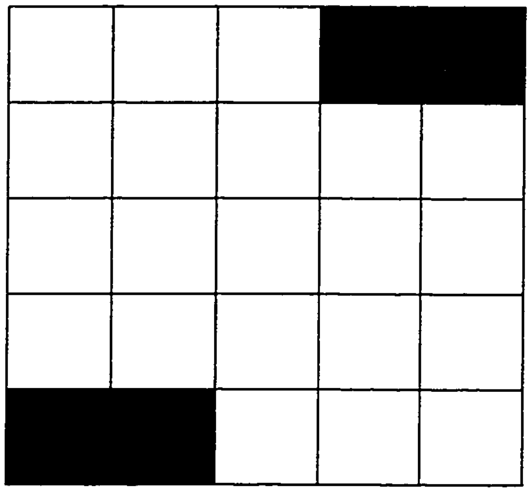
In the constraint programming methodology we cast the problem as a constraint satisfaction problem in terms of variables, values, and constraints. The choice of variables defines the search space and the choice of constraints defines how the search space can be reduced so that it can be effectively searched using backtracking search. Also part of the modeling task is to specify what kind of propagation is desired for each constraint: forward checking or arc consistency checking for example. See section 2.3 for more on this.

Here we discuss several problems that can be cast as constraint satisfaction problems, and that will be useful to us later: the *crossword puzzle* problem (see section 2.2.1), the *Golomb* problem (see section 2.2.2), the *logistics* problem (see section 2.2.3), and *randomly generated* problems (see section 2.2.4).

2.2.1 Crossword puzzle

Input to the crossword puzzle problem is a puzzle and a dictionary. The puzzle is an arrangement of spaces and blanks as in figure 2.1. Given such a puzzle the goal is to find words (from the given dictionary) that *fit* into the adjacent

Figure 2.1: A 5 × 5 crossword puzzle with 4 blanks.



spaces. The solution can be further constrained by requiring that all words in the solution be distinct. We will consider three ways to model this problem as a CSP: the *letter*, the *dual* and the *hidden*. A comparison between these three models can be found in table 2.1.

In the letter model there is a variable for each letter to be filled in and a constraint enforces that a maximally contiguous sequence of letters forms a word that is in the dictionary. In this formulation, the domain of a variable consists of 26 letters, from a to z. The arity of a constraint reflects the length of the word that the constraint represents. For example, a word of 10 letters will result in a 10-ary constraint over those letter variables. The tuples in the constraint are all of the words in the dictionary of the appropriate length.

An alternative representation of the problem, historically called the dual, can be automatically generated from the letter model. In this representation, each word in the puzzle is represented by a dual variable. The domain of a dual variable is all of the words in the given dictionary of the appropriate length. The domain size of a dual variable may be as large as 32865 (depending on the size of the dictionary). Binary constraints between these variables ensure that instantiating words chosen agree on intersecting letters.

In the hidden representation, each of the letters and each of the words in the puzzle are given a variable. This representation is called the hidden representation because the extra variables (corresponding to words in the puzzle) are referred to as hidden variables, because they do not participate in the solution. A hidden constraint enforces an assignment of a letter variable to be compatible with an assignment of a word variable.

Table 2.1: Size of an instance of a model given a dictionary and the grid shown in Figure 2.1, where n is the number of variables, d is the maximum domain size, r is the maximum constraint arity, and m is the number of constraints.

model	dictionary	n	d	r	m
letter	UK	21	26	10	23
dual	UK	10	10,935	2	34
hidden	UK	31	10,935	2	55
letter	words	21	26	10	23
dual	words	10	4,174	2	34
hidden	words	31	4,174	2	55

The set of crossword puzzle instances we used in our experiments include:

- A set of 10 puzzles $\{05.01, ..., 05.10\}$ that represent all legal puzzles of size 5×5. A legal puzzle is one which is symmetric, has no single letter words, and is connected.
- A set of 10 puzzles $\{15.01, ..., 15.10\}$ of size 15×15.
- A set of 10 puzzles $\{19.01, ..., 19.10\}$ of size 19×19.
- A set of 10 puzzles $\{21.01, ..., 21.10\}$ of size 21×21.
- A set of 10 puzzles $\{23.01, ..., 23.10\}$ of size 23×23.

For our experiments we used two dictionaries: one we refer to as the *UK dictionary*, which is available at [1], the other is the standard Linux dictionary, referred to as the *words dictionary*. A puzzle together with a dictionary define an instance of the crossword puzzle problem. Since we have fifty puzzles and two dictionaries this represents 100 instances.

2.2.2 Golomb

The Golomb problem is available at the CSPLib benchmark library (prob006 at http://csplib.cs.strath.ac.uk/). It is also described by Dewdney in [10]. The library describes the problem as:

> "A Golomb ruler may be defined as a set of m integers $0 = a_1 < a_2 < ... < a_m$ such that the $m(m-1)/2$ differences $a_j - a_i, 1 <= i < j <= m$ are distinct. Such a ruler is said to contain m marks and is of length a_m. The objective is to find optimal (minimum length) or near optimal rulers".

To distinguish between the m used in the above description and the m used in our notation for the number of constraints in a CSP, we will use M to refer to the number of marks in a Golomb ruler instance.

Example 1 As an example, a Golomb ruler with $M = 4$ can be constructed by placing mark 1 at distance 0 on the ruler, mark 2 at distance 1, mark 3 at distance 4, and mark 4 at distance 6. This corresponds to a ruler of length 6 which is the optimal Golomb ruler for $M = 4$.

The basic approach to solving an instance of the Golomb problem, like other optimization problems, is an iterative approach. For an instance with M marks the typical initial length to try is the minimum length found for an instance with $M - 1$ marks. So given an M, a ruler length L is selected to try first. Then a CSP instance is generated (more details on how this is done are given below). If a solution can be found to this instance, then L is the optimal length for a Golomb ruler with M marks. If a solution can not be found to this instance, then the process is repeated with a ruler of length $L+1$. This iterative process is continued, with progressively longer rulers, until the optimal length ruler is found.

The Golomb CSP generator takes as input the number of marks M and a ruler length L. The resulting CSP has M variables, $x_1, ..., x_M$, (i.e. $n = M$) each with a finite integer domain of size equal to the length L. This domain corresponds to each of the places the mark can be placed, so an assignment of a value a to a variable x_i corresponds to the i^{th} mark being placed on the ruler at position a.

The model we used involves three types of constraints:

- *binary less than constraints*
  $\forall x_i, x_j \in X$ if $i < j$ then $x_i < x_j$

- *ternary not equal constraints*
  $\forall x_i, x_j, x_k \in X$ if $(i < j < k)$ then ($|x_k - x_i| \neq |x_j - x_i|$ and $|x_k - x_j| \neq |x_j - x_i|$ and $|x_k - x_j| \neq |x_k - x_i|$)

- *quaternary not equal constraints*
  $\forall x_i, x_j, x_k, x_l \in X$ if $(i < j < k < l)$ then ($|x_l - x_i| \neq |x_k - x_j|$ and $|x_l - x_j| \neq |x_k - x_i|$ and $|x_l - x_k| \neq |x_j - x_i|$)

The range of problems that we found to be solvable within a reasonable length of time, given our formulation, are problems with up to 13 marks. If M is 11 and L is 72 (which is the optimal length for a Golomb ruler with 11 marks), the CSP instance generated has:

- 11 variables, each with domains of size of 73,

- 55 binary constraints,

- 165 ternary constraints, and

- 330 quaternary constraints.

2.2.3 Logistics

Logistics is a planning problem. Like the Golomb problem, described above, it is an optimization problem. In the logistics problem, there are packages which need to be moved around between cities and between locations within cities using trucks and planes. In particular an initial state specifies the locations of the packages, trucks and planes. A goal state specifies the desired location for each package. A solution to the problem is a series of transitions from the initial state to the goal state, where the number of transitions is minimized.

Following van Beek and Chen (see [28]) we model each state by a collection of variables and the constraints enforce valid transitions between states. For each state, S_t, we have the following variables: $C_{i,t}$ (representing packages), $T_{j,t}$ (representing trucks), and $P_{k,t}$ (representing planes), where i, j, k range over the number of packages, trucks, and planes, respectively, and t ranges over the number of steps (atomic actions) in the plan. The domains of the package variables are locations, trucks, and planes. Assigning a package variable a location means the package is at that location in that state and assigning a package variable a truck means the package is in that truck in that state. Similarly, the domains of trucks and planes are locations.

Minimally the constraints must enforce how variables can change from state S_t to state S_{t+1} (we call these action constraints) and how variables within a state must be consistent (we call these state constraints). More specifically, the essential constraints are:

- *Action constraints* model the effects of actions. For example, a package variable can only change from being at a location in state S_t to being in a truck or plane in state S_{t+1}. In other words a package can only be loaded onto a truck or plane if it is at the same location as the truck or plane.

- *State constraints* enforce how variables within a state must be consistent.

- *Distance constraints* are upper and lower bounds on how many steps are needed for a variable to change from one value to another. (Actually only the bounds for adjacent states are essential constraints).

When the problem is modeled with only this minimal set of constraints we refer to it as the *basic* model. Additional redundant constraints can be added for efficiency:

- *Symmetric values constraints* are constraints which break symmetries on the values that variables can be assigned. For example, given two package variables, the planes in their domains are often symmetric and if there is a solution (or no solution) with a particular assignment of planes to packages, there is another solution (or no solution) with the planes swapped.

- *Action choice constraints* enforce constraints on which actions can be performed in each state. For example, suppose there are two packages at an airport. A plane can either pick up both at once, or pick up one now and another later. All of these will end up being equivalent and a constraint is added which forbids all but one of the action sequences.

- Domain constraints enforce restrictions on the original domains for the variables. For example, if a package is to be delivered to a location within the city that it originates in, it can have its domain restricted to locations and trucks within that city.

When these additional, redundant constraints are added, we refer to this as the *redundant* model.

An iterative approach (similar to that described above for the Golomb problem) is used to find the optimal plan. The procedure employed involves generating a minimum bound on the number of transitions that will be required to reach the goal. This is done by simplifying the problem, by dropping some preconditions on the actions, and then computing the minimum number of steps required to solve this simplified version.

Beginning with this minimum, the algorithm generates a CSP (assuming exactly this many steps will be required) and attempts to solve it. If the instance can not be solved then the minimum bound is incremented and a new CSP is generated. This iterative process continues until a solution (i.e. a plan) is found.

The test suite used has 35 instances of the logistics problem. As an example, here are some statistics from problem number 6 of this test suite: 16 packages, 2 planes, 18 cities, 4 locations per city, and 26 trucks. The minimum bound calculated is 11 steps. Given the basic model, the CSP generated, for the first iteration (corresponding to 11 steps), for problem 6 has: 288 variables with domain sizes ranging from 1 to 23, 660 binary constraints, and 1540 quaternary constraints.

For the first iteration of problem 6, given the redundant model, the CSP generated has: 288 variables with domain sizes ranging from 1 to 23 (same as basic model), 660 binary constraints, 2290 ternary constraints, 5548 quaternary constraints, and 275 5-ary constraints.

2.2.4 Randomly generated problems

Often it is valuable to look at randomly generated problems because it can be helpful to have problems with specific properties and parameters that can be systematically varied. We use a simple approach to generating random problems. Input to the generator is the number of variables n, the domain sizes d of the variables (all variables have the same domain size), the arity of the constraints r (all constraints have the same arity) and the tightness of the constraints t (all constraints have the same tightness).

The generator then creates a CSP with these parameters. Satisfying tuples are stored explicitly and are chosen uniformly from the space of all tuples. With this formulation, the available memory limits the size of problems that can be generated.

2.3 Algorithms for solving constraint satisfaction problems

Once a problem is modeled as a CSP, several algorithms are available for solving it. The most commonly used complete algorithms are based on chronological backtracking. One way in which the various backtracking algorithms differ is in the level of consistency that is enforced. The most common levels of consistency used are *forward checking* and *full arc consistency*. In section 2.3.1 we introduce the basic chronological backtracking algorithm. In section 2.3.2 we introduce forward checking. Arc consistency is discussed, in detail, in chapter 3.

Forward checking and arc consistency can be characterized as look ahead techniques. *Conflict-directed backjumping* is a *backward checking* algorithm that can also be used in conjunction with backtracking. It is discussed in section 2.3.3.

2.3.1 Chronological backtracking

Chronological backtracking, often just called backtracking, is a common problem solving technique. The version of backtracking listed in algorithm 1 is a recursive implementation. Each call to the recursive procedure corresponds to a node in the search tree. At each node the algorithm picks an uninstantiated variable to instantiate next (see line 9 of algorithm 1). Each value in this variable's domain is tried as a possible value for the variable and the function CheckConstraints() is called. For each constraint for which all of the participating variables are instantiated, CheckConstraints() performs a *constraint check* to ensure that it is satisfied. If assigning this value to the variable does not violate any constraints then the search is continued by recursing deeper.

If at any stage, during the search, no satisfying instantiations can be found, the algorithm *backs up* to the previous level in the tree. Any time that the search reaches level n, where n is the number of variables in the problem being solved, if all constraints are satisfied then the current instantiations form a solution to the problem. If this occurs then the algorithm quits. However, if the line "return true" on line 15 is replaced with "return false", then the resulting algorithm continues searching until all solutions are found.

Efficiency gains can be achieved by allowing the selection of the variable to instantiate next, done by the function PickVariable(), to be dynamic. Various heuristics are often used to guide the selection of the next variable to instantiate. These heuristics, often called *dynamic variable orderings*, are used to minimize the size of the search tree.

One of the most popular variable ordering techniques is called *fail first.* This ordering chooses the variable with the fewest (currently) valid values (smallest domain). This reduces the branching factor at the current level. It also attempts to raise the height of failures. We will refer to this ordering

Algorithm 1 Backtrack(level)

```
1: if at leaf then
2:    if CheckConstraints() then
3:       print solution
4:       return true
5:    else
6:       return false
7:    end if
8: end if
9: $var \leftarrow$ PickVariable()
10: for all $d \in D(var)$ do
11:    if $(var, d)$ is a currently valid label then
12:       assign the variable $var$ the value $d$
13:       if CheckConstraints() then
14:          if Backtrack( level+1 ) then
15:             return true
16:          end if
17:          restore( level )
18:       end if
19:    end if
20: end for
21: return false
```

as *ff*. *Degree* is another feature that is often exploited by variable ordering heuristics. A variable's degree is the number of variables that are *connected* with it. Two variables, x and y, are connected if there exists a constraint c such that $x \in X(c)$ and $y \in X(c)$. Selecting the variable that minimizes the ratio

$$\frac{domain\_size}{degree},$$

has been found to be a useful variable ordering heuristic. We will refer to this ordering as *ff/deg*. A third variable ordering, which we will call *ff*+deg, first considers domain size and then breaks ties by selecting the variable with the largest degree. Finally a variable order, which we will call *ff/con*, picks the variable that minimizes the ratio

$$\frac{domain\_size}{con},$$

where *con* is the number of constraints the variable is involved in.

2.3.2 Forward checking

Chronological backtracking is often inefficient and is rarely used on its own in the context of solving constraint satisfaction problems. Combining a *forward checking* algorithm with backtracking can often increase the efficiency (in terms of CPU time) of the search. Forward checking can perform consistency checks on variables that have not been instantiated and potentially remove values from the domains of these variables, thus reducing the branching factor of the search space beneath the current node. This reduction comes at the cost of increasing the cost of processing each node in the tree; however, in many cases the net effect is a reduction in the CPU time required to solve the problem.

To combine a forward checking algorithm with the basic backtracking algorithm (algorithm 1), the function on line 13, CheckConstraints(), is replaced by the function ForwardCheck(). The function ForwardCheck() considers each *forward checkable* constraint, that could have been affected, directly or indirectly, by the assignment in line 12. A constraint is forward checkable if exactly one of the participating variables has not been instantiated. For each such constraint the routine removes inconsistent values from the domain of the uninstantiated variable. If during this process a domain is reduced to the empty set, ForwardCheck() returns false, and the effects of forward checking are undone. Example 2 shows how this is done.

Example 2 Consider the problem $P = (X, D, C)$, where
$X = \{x_0, x_1, x_2\}$,
$D(x_0) = \{a_0, a_1\}$, $D(x_1) = \{a_0, a_1\}$, $D(x_2) = \{a_0, a_1\}$,
$C = \{c_0, c_1\}$, $X(c_0) = \{x_0, x_1\}$, $X(c_1) = \{x_1, x_2\}$,
$c_0 = \{[a_0, a_1], [a_1, a_0], [a_1, a_1]\}$ and
$c_1 = \{[a_0, a_0], [a_0, a_1], [a_1, a_1]\}$.

Assume that x_0 is selected as the first variable to instantiate during backtracking and that it is given value a_0. The constraint c_0 becomes forward checkable, because x_1 is the only uninstantiated variable in $X(c)$. A forward checking algorithm would check each of the values in x_1's domain, a_0 and a_1.

Checking is done by first (temporarily) assigning x_1 the value a_0 and checking if the resulting tuple $[a_0,a_0]$ (that is $x_0 = a_0$ and $x_1 = a_0$) satisfies the constraint c_0. Since it does not, a_0 is removed from x_1's domain. Next x_1 is assigned the value a_1 and the resulting tuple $[a_0,a_1]$ is checked against the constraint c_0. Since this tuple does satisfy the constraint, the value a_1 is not removed from the domain of x_1. Finally x_1 is unassigned.

When the search reaches level n, where n is the number of variables in the problem being solved, it is unnecessary to check that all constraints are satisfied (as the backtracking algorithm does on line 2). This is because the pruning done by the ForwardCheck() function guarantees that all completely instantiated constraints are satisfied. When all variables in a constraint except one are instantiated, forward checking removes all values from the domain of the uninstantiated variable's domain that are inconsistent with the current partial instantiation. So when the last variable is instantiated it is certain to be given a consistent value.

2.3.3 Conflict-directed backjumping

Forward checking can be used to look forward in the search by considering uninstantiated variables. Conflict-directed backjumping can be used to perform, what is sometimes called, backward checking. Every variable has a conflict set that contains the past variables which failed consistency checks with its current instantiation. When consistency checks fail, during backtracking, conflict sets allow the algorithm to keep track of the variables whose instantiations are conflicting. This information allows "multiple backjumps" to be performed; that is, after the initial backjump from a dead-end it can continue backjumping across conflicts, which may potentially result in significant savings.

Conflict-directed backjumping and forward checking are orthogonal and can be beneficially used in conjunction with each other. Algorithms that combine look ahead and backward checking techniques are called *hybrid* algorithms.

Chapter 3

Arc consistency for general constraint satisfaction problems

In this chapter we discuss two general arc consistency algorithms: gac3 and gac7. Gac3 is described in section 3.2. Gac7 is described in section 3.3. Section 3.4 presents an experimental evaluation of these two algorithms. We begin with an introduction and some definitions.

3.1 Introduction

In chapter 2 we introduced the basic backtracking algorithm. We also discussed how a forward checking algorithm could be interleaved with backtracking. The pruning (i.e., the removal of values from domains) performed by forward checking can significantly reduce the number of nodes visited during backtracking.

General arc consistency enforces a stronger level of consistency than forward checking does and, therefore, can in general prune more values from the domains of uninstantiated variables. When an algorithm for maintaining arc consistency is interleaved with a backtracking algorithm the size of the search tree can, in many cases, be significantly reduced. At the same time more work is required at each node in the search tree. For many problems, as in the case of forward checking, the result can be an overall reduction in running time. A backtracking algorithm, combined with a general arc consistency routine, is shown in algorithm 2.

Algorithm 2 Backtrack(level)

```
1: if at leaf then
2:    print solution
3:    return true
4: end if
5: $var \leftarrow$ PickVariable()
6: for all $d \in D(var)$ do
7:    if $(var, d)$ is a currently valid label then
8:       assign the variable $var$ the value $d$
9:       if EnforceConsistency() then
10:         if Backtrack( level+1 ) then
11:            return true
12:         end if
13:         restore( level )
14:      end if
15:   end if
16: end for
17: return false
```

This chapter investigates two different algorithms for enforcing arc consistency on general constraint satisfaction problems (that is, two instantiations of the function EnforceConsistency() on line 9 of algorithm 2): *general arc consistency 3* (gac3) and *general arc consistency 7* (gac7). Gac3 is a well known algorithm that was originally proposed by Mackworth in 1977 (see [19]). Gac7 was proposed by Bessiere and Regin in 1997 (see [4]). Arc consistency can be used in two ways: as a preprocessing algorithm and incorporated into backtracking. It is the latter which is the most useful. Bessiere and Regin show how to enforce arc consistency, but do not describe how to incorporate it into backtracking. In section 3.3 we describe gac7 and describe how it can be incorporated into backtracking. Also, Bessiere and Regin, do not provide a meaningful evaluation of the gac7 algorithm. In section 3.4 we present results from our experimental evaluation of gac7. In particular we compare it to gac3.

We begin this discussion with some definitions. Following Bessiere and Regin [4], let $P = (X, D, C)$ be a constraint satisfaction problem, let $c \in C$ be a constraint, and let $x \in X(c)$ be a variable involved in c. A value $a \in D(x)$ is consistent with c iff $\exists t \in c$, such that $a = t[x]$ and t is still valid (that is all values in the tuple are still valid values). The tuple t can be called a support for the label (x, a) on c. The constraint c is arc consistent iff $\forall x \in X(c), D(x) \neq \emptyset$ and $\forall a \in D(x)$, a is consistent with c. Problem P is arc consistent iff all the constraints in C are arc consistent. Enforcing arc consistency involves removing inconsistent values from domains until all constraints are arc consistent. Note that removing a value from a domain may make some previously valid tuples invalid. As a result some previously consistent values may become inconsistent.

During backtracking, when a variable is instantiated its domain is reduced to one element (namely the value that has just been assigned to it). As a result of this assignment the problem may no longer be arc consistent. The function EnforceConsistency() (line 9 of algorithm 2) is called to reestablish arc consistency; if this is not possible then the function returns false. Example 3 demonstrates that an instantiation (during backtracking) can result in a previously consistent problem becoming inconsistent. It also demonstrates what needs to be done to restore arc consistency.

Example 3 Consider the problem $P = (X, D, C)$, where

- $X = \{x_0, x_1, x_2\}$,
- $D(x_0) = \{a_0, a_1\}$, $D(x_1) = \{a_0, a_1\}$, $D(x_2) = \{a_0, a_1\}$,
- $C = \{c_0, c_1\}$, $X(c_0) = \{x_0, x_1\}$, $X(c_1) = \{x_1, x_2\}$,
- $c_0 = \{[a_0, a_1], [a_1, a_0], [a_1, a_1]\}$ and $c_1 = \{[a_0, a_0], [a_0, a_1], [a_1, a_1]\}$.

P is arc consistent, meaning that each constraint is arc consistent. To see that c_0, for example, is arc consistent notice that each value in the domain of the variables involved in c_0 (x_0 and x_1) is *supported* by (or appears in) at least one of the tuples in c_0. The same observation holds for c_1.

Now let's see what happens during backtracking. Assume that x_0 is selected as the first variable to instantiate and that it is given value a_0. The value a_1 is no longer a valid member of the domain of x_0, which means that the constraint c_0 is (effectively) reduced to $\{[a_0, a_1]\}$ (i.e. only those tuples that do not contain a value of a_1 for x_0). In other words, if x_0 is given the value

a_0, then x_1 can not be assigned the value a_0, without violating c_0. Now notice that P is no longer arc consistent, because not all of the values in $D(x_1)$ (a_0 in particular) are consistent with c_0.

To reestablish arc consistency, a_0 must be removed from $D(x_1)$. However this reduces c_1 to $\{[a_0, a_1], [a_1, a_1]\}$. Because the value a_0 for the variable x_2 is not consistent with c_1, P is still not arc consistent. So a_0 must be removed from $D(x_2)$. Now P is arc consistent.

We now consider two algorithms for enforcing arc consistency: general arc consistency 3 (described in section 3.2) and general arc consistency 7 (described in section 3.3).

3.2 General arc consistency 3 (gac3)

General arc consistency 3, as proposed by Mackworth [18, 19] has become a common method for enforcing arc consistency during backtracking. A brief description of this algorithm follows.

The key data structure used by gac3 is a stack of variables. The presence of a variable in the stack means that its domain has been reduced and the effects of this reduction still need to be propagated. Propagating the effects of this reduction involves determining which other domains now need to be reduced as a consequence.

Algorithm 3 EnforceConsistency() gac3 version

```
1: while $stack \neq \emptyset$ do
2:    remove a variable, $var$, from the $stack$
3:    for all constraints $c$ that $var$ is involved in do
4:       for all uninstantiated variable $v$ (except $var$) participating in $c$ do
5:          if not Revise($v$,$c$) return false
6:       end for
7:    end for
8: end while
9: return true
```

During backtracking, when a variable is instantiated (and therefore its domain is reduced) it is pushed onto the stack. Then EnforceConsistency() is called. The gac3 version of the EnforceConsistency() (shown in algorithm 3) loops until the stack is empty. Each time through the loop a variable, *var*, is popped off the stack, and for each variable, v, that is involved in a constraint with *var*, the function Revise() is called.

Algorithm 4 Revise(v,c)

```
1: $changed \leftarrow false$
2: for all values $a$ in the domain of $v$ do
3:   if not Exists($v$,$a$,$c$) then
4:     remove $a$ from $D(v)$
5:     $changed \leftarrow true$
6:   end if
7: end for
8: if changed then
9:   push $v$ onto the stack
10: end if
11: if $D(v) = \emptyset$ then
12:   return false
13: else
14:   return true
15: end if
```

The function Revise() (see algorithm 4) considers each value in the domain of v. For each of these values the function determines if it is still valid, if it is not it is removed from the appropriate domain and the appropriate variable is pushed on the stack. If at some point in this processing the domain of some variable is reduced to the empty set then false is returned. Otherwise true is returned.

If Revise() returns false, having failed to establish arc consistency on a constraint, EnforceConsistency() also returns false. This means that, given the current partial instantiations, the given problem can not be made arc consistent. This also means that, given the current instantiations, no solution to the problem exists.

It is important to note that while gac3 is characterized as an arc consistency algorithm, it does not need to establish arc consistency before backtracking begins, nor does it, strictly speaking, need to maintain arc consistency during backtracking. In fact, with gac3 it is possible to vary the level of consistency enforced. Just as forward checking (see section 2.3) only considers constraints that have exactly one uninstantiated variable, gac3 can be set to only consider constraints with at most u uninstantiated variables. Then, if u is set to 1, gac3 enforces the same level of consistency as forward checking. If u is set to ∞, full arc consistency is enforced (except that often arc consistency is not established before backtracking begins). This flexibility is of great practical value, because the parameter u can be used to balance the tradeoffs between decreasing the number of nodes visited and reducing the amount of work done at each node.

Using gac3 to enforce arc consistency can be expensive, especially for large values of u. Consider the amount of work that can be performed by Revise(). As mentioned above, Revise() looks at each uninstantiated variable in a constraint and checks if each value in its domain can satisfy the constraint. If u is set to ∞ and d is the maximum domain size, then for a constraint of arity r, the amount of work done to propagate the effects of a deletion on this constraint will be proportional to d^{r-1}. For large problems, or problems where checking constraints is expensive, this can be prohibitively expensive. This difficulty has led to work on improving gac3.

3.3 General arc consistency 7 (gac7)

In [4] Bessiere and Regin propose a general schema for arc consistency on non-binary constraint networks. The schema is based on the AC-7 schema for binary constraint satisfaction problems (see [3]). The basic approach involves maintaining a single supporting tuple for each label. When this is no longer possible (i.e. there is no available support) the label is removed. In addition gac7 also supports *multi-directionality.* That is, if the constraint c involves the variables x_0, x_1 and x_2, and the tuple $t = [a_0, a_1, a_2]$ is currently the support for (x_0, a_0) on c, then the algorithm will infer that t can also serve as a support for the labels (x_1, a_1) and (x_2, a_2).

Two special data structures are needed to store supports and to support multi-directionality: *supports* and *values*.

- The data structure *supports*(c, x, a) is the set of tuples t that are the current support for some value on c and such that $t[x] = a$. This provides a mapping between labels and supporting tuples that depend on them.
- The data structure *values*(c, t) is the set of values (labels) that are currently supported by the tuple t on c. This provides a mapping between tuples and the labels supported by them.

These data structures allow the algorithm to efficiently answer the questions:

- Given a label (x, a), which tuples currently support some labels and contain (x, a)?
- Given a tuple t, which labels are currently supported by t?

This information is used to minimize the work done in propagating the effects of changes to domains.

Where gac3 maintains a stack of variables whose domains have been reduced, gac7 maintains a stack of labels that have been deleted. Recall that when an arc consistency algorithm is interleaved with backtracking, the arc consistency algorithm is called immediately after a variable is instantiated (see algorithm 2).

Assume that a variable x has the domain $\{a_0, a_1, a_2\}$. If x is instantiated with the value a_0, the values a_1 and a_2 are removed from $D(x)$. The labels (x, a_1) and (x, a_2) all become invalid and are pushed onto the stack. Then the function EnforceConsistency() (see algorithm 5) is called.

Algorithm 5 EnforceConsistency() gac7 version

```
1: while $stack \neq \emptyset$ do
2:   remove a label $(x, a)$ from $stack$
3:   for all constraints $c$ that variable $x$ participates in do
4:     if not Propagate(c,x,a) then return false
5:   end for
6: end while
7: return true
```

The gac7 version of EnforceConsistency() is simply a loop that processes each label in the stack. Processing is done by calling Propagate() (see algorithm 6) with a label (x, a) that has been deleted and a constraint c that the variable x participates in. The purpose of Propagate() is to propagate the effects of this deletion; that is, to determine what other labels need to be deleted as a consequence. The first step in this is to determine which tuples, that are currently supporting at least one label on c, become invalid with the deletion of (x, a). This is available by checking the contents of the set $supports(c, x, a)$. Next, for each such tuple, t, the labels that are currently supported by t are retrieved from the set $values(c, t)$. For each of these labels FindSupport() is called.

Algorithm 6 Propagate(c,x,a)

```
1: **for all** $t \in supports(c, x, a)$ **do**
2:    **for all** $(z, b) \in t$ **do** remove $t$ from $supports(c, z, b)$
3:    **for all** $(z, b) \in values(c, t)$ **do**
4:       remove $(z, b)$ from $values(c, t)$
5:       **if** not FindSupport(c,x,a) **then return** false
6:    **end for**
7: **end for**
8: **return** true
```

FindSupport() (see algorithm 7) first calls SeekInferableSupport(). SeekInferableSupport() allows the algorithm to deal with multi-directionality. Assume $t = [a_0, a_1, a_2]$ is a tuple on a constraint c where $X(c) = (x_0, x_1, x_2)$. If t is found as a support for (x_0, a_0), then (x_0, a_0) is added to the set $values(c, t)$; this means that t is currently the support for (x_0, a_0). Also, t is added to the sets $supports(c, x_0, a_0)$, $supports(c, x_1, a_1)$ and $supports(c, x_2, a_2)$. The reason for this is that t could potentially serve as a support for (x_1, a_1) and (x_2, a_2). So SeekInferableSupport() checks for an *already found* support by looking through the appropriate *supports* set. If a tuple t, supporting (x, a) on c is inferred, then (x, a) is added to $values(c, t)$. Recall that $values(c, t)$ is the set of all labels currently supported by t on c.

If support for (x, a) on c can not be inferred then SeekNextSupport() is called. SeekNextSupport() searches for a currently valid tuple (that is, one that satisfies c and contains only valid values) to be a support for (x, a) on c. If no such tuple can be found FindSupport() will return false. This implies that given the current partial assignments the label (x, a) is inconsistent, and is therefore removed and pushed onto the stack.

Algorithm 7 FindSupport(c,x,a)

```
1: $t \leftarrow$ SeekInferableSupport(c,x,a)
2: **if** support was successfully inferred **then**
3:    add $(x, a)$ to $values(c, t)$
4: **else**
5:    $t \leftarrow$ SeekNextSupport(c,x,a)
6:    **if** support could not be found **then**
7:       remove $a$ from $D(x)$
8:       push $(x, a)$ onto the $stack$
9:       **if** $D(x) = \emptyset$ **then return** false
10:   **else**
11:      add $(x, a)$ to $values(c, t)$
12:      **for all** $y \in X(c)$ **do**
13:         add $t$ to $supports(c, y, t[y])$
14:      **end for**
15:   **end if**
16: **end if**
17: **return** true
```

Example 4 A clarifying example may be helpful. We will walk through example 3, given in section 3.1 to show how gac7 works. After the supports have been initialized (which must be done before backtracking begins) the *values* structure could map the following tuples to labels (the mapping is not unique):

$$\begin{array}{ll} values(c_0, [a_0, a_1]) = \{(x_0, a_0), (x_1, a_1)\} & values(c_1, [a_1, a_1]) = \{(x_1, a_1)\} \\ values(c_0, [a_1, a_0]) = \{(x_0, a_1), (x_1, a_0)\} & values(c_1, [a_0, a_1]) = \{(x_2, a_1)\} \\ values(c_1, [a_0, a_0]) = \{(x_1, a_0), (x_2, a_0)\} & \end{array}$$

The first entry can be read as "the labels (x_0, a_0) and (x_1, a_1) are currently supported by the tuple $[a_0, a_1]$ on constraint c_0". The *supports* structure would then map the following labels to tuples:

$$\begin{array}{ll} supports(c_0, x_0, a_0) = \{[a_0, a_1]\} & supports(c_1, x_1, a_0) = \{[a_0, a_0], [a_0, a_1]\} \\ supports(c_0, x_0, a_1) = \{[a_1, a_0]\} & supports(c_1, x_1, a_1) = \{[a_1, a_1]\} \\ supports(c_0, x_1, a_0) = \{[a_1, a_0]\} & supports(c_1, x_2, a_0) = \{[a_0, a_0]\} \\ supports(c_0, x_1, a_1) = \{[a_0, a_1]\} & supports(c_1, x_2, a_1) = \{[a_1, a_1]\} \end{array}$$

The first entry can be read, "the tuple $[a_0, a_1]$ is currently the support for at least one label on constraint c_0 and it depends on the label (x_0, a_0)."

If, during backtracking, the variable x_0 is assigned the value a_0 then the value a_1 is removed from the domain of x_0 and the label (x_0, a_1) is added to the stack. The algorithm next enters the loop in the function Enforce-Consistency(). The label (x_0, a_1) is retrieved from the stack. In order to propagate the effects of removing the label (x_0, a_1), the algorithm looks at the set *supports*(c_0, x_0, a_1) to get the set of supports that depend on (x_0, a_1). This set contains only the tuple $[a_1, a_0]$. Next the set *values*$(c_0, [a_1, a_0])$ is used to retrieve the set of labels that are currently supported by the tuple $[a_1, a_0]$. This set contains two labels (x_0, a_1) and (x_1, a_0), which are both removed from the set *values*$(c_0, [a_1, a_0])$. Note that the label (x_0, a_1) has already been deleted. The algorithm next attempts to find new support for the label (x_1, a_0). No support can be inferrred as the only tuple in the set *supports*(c_0, x_1, a_0) is the tuple $[a_1, a_0]$ which is invalid since the label (x_0, a_1) is invalid. Also, none can be found so the value a_0 is removed from the domain of x_1 and the label (x_1, a_0) is pushed on the stack. Next the algorithm propagates the effects of removing the label (x_1, a_0) in a similar fashion.

In summary, using these data structures allows the algorithm to know exactly which labels need new support in the event of a deletion. This is in an effort to avoid looking at all values in the domains of all variables as gac3 does. However, initializing (before backtracking begins), and maintaining (during backtracking) these data structures imposes some additional overhead.

3.3.1 Implementation notes

Initially the *values* and *supports* structures were implemented as multidimensional arrays (in C++). Given a tuple, t, satisfying a constraint, c, *values*$[c][t]$ is a set of labels for which t is the current support on c. This requires that t be an integer, and therefore that tuples are stored extensionally. Sets are implemented as sorted binary trees (C++ STL set type). For a given problem $P = (X, D, C)$, if t is the maximum number of satisfying tuples then the *values* structure consumes space proportional to $|C| \times t$.

Given a constraint c and a label (x, a), where x participates in c (that is, $x \in X(c)$), *supports*$[c][x][a]$ is a set of tuples that satisfy c and contain the label (x, a). As above tuples are stored as integers and this set is implemented as a sorted binary tree. If r is the maximum arity of a constraint, and d is the maximum domain size, then the *supports* structure consumes space proportional to $|C| \times r \times d$.

For many problems these space requirements are unreasonable. Also for many problems it is infeasible to compute (and store) all satisfying tuples. For these reasons tuples can not always be stored in extension. These difficulties motivated a second implementation of the *values* and *supports* data structures. In our second implementation, tuples were represented as arrays of values (rather that as integers). For each constraint c, *values*$[c]$ and *supports*$[c]$ are map data structures. These maps are implemented as a binary tree of key-value pairs, and are sorted on key (C++ STL multimap type). In the case of *values*, the mapping is from tuples to labels. In the case of *supports*, the mapping is from labels to tuples. The space requirements are proportional to the number of tuples currently supporting some label.

Our initial implementation allows the program to answer questions like "which labels are currently supported by tuple t?" in constant time. Our second implementation sacrifices some time to reduce the memory required. In our second implementation the above question can be answered in logarithmic (in the size of the structure) time.

Another data structure that Bessiere and Regin propose and we found to be of practical value is *lastsupport*$(level, c, (x, a))$. This structure is used by the FindNewSupport(). It stores the last tuple returned as a support for the label (x, a) on the constraint c at the specified level. This data structure can consume a significant amount of space. In implementing this structure, space and time tradeoffs similar to those encountered in implementing the *values* and *supports* data structures are encountered.

To demonstrate the effects of these time versus space tradeoffs, we present some quick comparisons using crossword puzzles. These results were obtained using the letter model of problem 15.01 and the UK dictionary (dvol). Our initial version of gac7 solves this problem in about 23 seconds of CPU time and uses about 48 MB of main memory. If the *lastsupport* data structure is removed, gac7 solves the problem in about 32 seconds of CPU time using 40 MB of main memory. Finally if arrays (for *values* and *supports* data structures) are replaced by maps, as discussed above, gac7 uses 6 MB of main memory but can not solve the problem within 1 hour of CPU time. More extensive experimentation is discussed in the next section.

3.4 Experiments

Gac3 and gac7 were compared on three types of problems: crossword puzzle problems (see section 3.4.1), logistics problems (see section 3.4.2) and some random problems (see section 3.4.3). All of our experiments were run on either a 400 MHz Intel Pentium II or a 450 MHz Intel Pentium III. Each machine had 256 MB of main memory. Results from the Pentium II machine are never compared with results from the Pentium III machine. In all of our implementations, conflict-directed backjumping (as described in section 2.3) is used together with the arc consistency algorithm.

3.4.1 Crossword puzzle problem

The crossword puzzle problem, together with the instances we used, are described in section 2.2.1. Table 3.1 shows the results from some initial experiments done on various models of the crossword puzzle problem. Three algorithms were used: a generic gac3 algorithm, an implementation of gac7 which requires that satisfying tuples be stored extensionally, and a special purpose propagator (called *pac* in the table) for each model that enforces arc consistency. Each of these propagators use knowledge about the model to very efficiently enforce arc consistency. Two dynamic variable orderings were used: *ff+deg* and *ff/deg*. Both of these ordering algorithms are described in section 2.3.1.

The implementation of gac7 used for these experiments could not be used on the dual and hidden models of the crossword puzzle problem because the memory required was well above the memory limit that we set (256 MB). In the letter model of the crossword puzzle problem, the *values*, *supports* and *lastsupport* data structures all fit easily into main memory. In this model the satisfying tuples, which are the words in the dictionary, also easily fit into memory.

Table 3.1: Number of problems, out of 100, that could be solved given a limit of 256 MB of memory and 10 hours of CPU time per problem.

	model		
algorithm	letter	dual	hidden
gac3, ff+deg	20	50	83
gac3, ff/deg	20	50	81
gac7, ff+deg	89	0	0
gac7, ff/deg	92	0	0
pac, ff+deg	88	80	84
pac, ff/deg	91	85	84

Gac7 performed very well on this set of problems. Out of the 100 problems described in section 2.2.1, given 10 hours of CPU time, gac7 can solve 92. The next best result was obtained by the special purpose propagator on the letter model. For a general purpose algorithm (gac7) to perform as well or nearly as well as a special purpose propagator is significant.

Gac3 performed less well on this problem set. Out of the 100 problems, given 10 hours of CPU time, gac3 can solve only 20 instances of the letter model. However, gac3 has the benefit of being applicable to all of the models of the problem that we considered. On the dual version of the problem, gac3 was able to solve 50 problems and on the hidden model gac3 was able to solve 83 of the 100 problems.

The gac3 implementation used in the above experiments, did not take advantage of the fact that the tuples were stored extensionally, while the gac7 implementation did. The constraints in the letter model of this problem are extremely tight, so making use of the explicitly stored tuples gives gac7 a significant advantage over gac3. So, to compare more closely the relative performance of gac3 and gac7 we consider here some results from a slightly simplified version of the crossword puzzle. This simplification comes by dropping the constraint that words in the solution must be unique. This is done to allow for a more direct comparison of the propagation done by the two algorithms. A second implementation of gac3 was developed to take advantage of the tuples being in extension. Recall from the discussion in section 3.2 that gac3 can enforce varying levels of consistency. This level is controlled by the parameter u. In our experiments we consider two values of u, 2 and ∞.

Table 3.2: Number of seconds spent finding a solution to ten crossword puzzle problems. The number in brackets, beside gac3, is the value of u.

	extensional			not extensional		
puzzle	gac7	gac3 (2)	gac3 (∞)	gac7	gac3 (2)	gac3 (∞)
05.01	1.06	57.22	3.65	68.24	4.25	136.72
05.02	0.68	36.33	1.90	41.18	3.38	71.77
05.03	0.37	7.63	1.15	23.21	0.78	34.89
05.04	0.28	0.43	0.66	10.80	0.14	5.95
05.05	0.22	0.18	0.49	10.88	0.16	4.72
05.06	0.52	277.96	1.44	39.05	23.90	68.18
05.07	0.34	121.34	1.13	29.83	11.40	36.47
05.08	0.42	19.85	1.23	32.92	1.97	56.42
05.09	0.27	6.87	0.56	12.08	0.60	5.26
05.10	0.21	0.19	0.43	10.54	0.12	4.33
total	4.37	528.00	12.64	278.73	46.70	424.71
median	0.36	13.74	1.14	26.52	1.38	35.68

Results were obtained from experiments on the letter model of the crossword puzzle problem. In particular 5×5 (see table 3.2) and 15×15 (see table 3.3) instances were used. In all cases the *words* dictionary was used.

These results allow several interesting comparisons. First, notice that on both the 5×5 and 15×15 puzzles, gac7 with tuples being stored extensionally performs the best. On puzzles of size 15×15, for example, gac7 reduced the median time by a factor of 4.51, and the total time by a factor of 4.06 over gac3 with u set to ∞. Gac3 with u set to 2, and tuples stored extensionally, could not solve any of the 15×15 puzzles with in the given the one hour CPU

Table 3.3: Number of seconds spent finding a solution to ten crossword puzzle problems. The number in brackets, beside gac3, is the value of u. A '-' means that the algorithm could not solve the particular problem within 1 hour of CPU time.

	extensional			not extensional		
puzzle	gac7	gac3 (2)	gac3 (∞)	gac7	gac3 (2)	gac3 (∞)
15.01	23.33	-	75.86	-	-	-
15.02	24.68	-	188.24	-	-	-
15.03	22.51	-	72.61	-	966.2	-
15.04	19.16	-	130.92	-	-	-
15.05	14.98	-	43.60	-	-	-
15.06	44.79	-	234.67	-	-	-
15.07	83.66	-	220.55	-	-	-
15.08	17.95	-	45.27	-	-	-
15.09	16.61	-	46.77	-	-	-
15.10	38.87	-	187.65	-	-	-
total	306.54	-	1246.14	-	-	-
median	22.92	-	103.39	-	-	-

time limit.

When tuples are not stored extensionally, gac3 with u set to 2 performs best. For 5×5 problems, using gac3 with u set to 2 reduced the median time by a factor of 25.9 over gac3 with u set to ∞, and by a factor of 19.2 over gac7. It seems that when tuples are not stored extensionally (and therefore enforcing consistency becomes more expensive) the extra work done to enforce a stronger level of consistency outweighs the savings in nodes searched.

In both the extensional and intensional cases gac7 improves performance over gac3 when u is set to ∞, though in the intensional case the improvement is only moderate. This shows that on this problem gac7 is a superior algorithm for enforcing full arc consistency.

As a last comparison it is worth noting that with u set to 2, gac3 performs better when tuples are not stored extensionally. The reason for this is that by the time the algorithm is looking for a support (see algorithm 4) for a particular value, part of the constraint is already instantiated. As an example, consider a four letter word in the crossword puzzle. Before any propagating is done on this constraint (if u is set to 2) two letters have been filled in. If the first two letters are filled in it may look like "t,h,_,_". During propagation the algorithm may ask the question "is the letter *e* a valid value for the third letter in this word"? This is equivalent to asking the question "is there a four letter word in the dictionary that begins with *the*?" If tuples are stored extensionally the algorithm will look at each four letter word until one beginning with *the* is found. If tuples are not stored extensionally the algorithm will consider each of the (at most) 26 possible terminations to the word (i.e. *thea*, *theb*, *thec*, ...) until one is found that is in the dictionary. Of course the second approach will be more efficient in this case.

On the other hand, if u is set to ∞ then the algorithm will begin propagating the effects of domain reductions even if no variables have been instantiated. In the case of a four letter word the algorithm will ask the question "is there a four letter word in the dictionary beginning with *e*?" If satisying tuples are not stored explicitly, the algorithm will try all 26^3 possible completions of e,_,_,_. If satisying tuples are stored explicitly, the algorithm will look at each four letter word in the dictionary until one is found that begins with *e*. This second approach will be more efficient in this case.

From these experiments we conclude that gac7 is superior to gac3 on the letter model of the crossword puzzle problems. However, we have also exposed some weaknesses of our gac7 implementation. Gac7 could not be used on the dual and hidden models of the crossword puzzle problem due to excessive space consumption. Also, with our implementation of gac7, it is not possible to vary the level of consistency enforced during backtracking. This flexibility proved valuable to gac3 in our experiments.

3.4.2 Logistics

Next we compared gac3 and gac7 on the logistics problem. The logistics problem, together with the instances we used, are described in section 2.2.3. The experiments reported here are based on the basic model of the logistics problem. The major constraints in this problem are action constraints. These constraints specify when a package can be loaded or unloaded from a plane or truck. In one set of experiments gac3, with u set to ∞, is used to propagate this constraint. In another set of experiments gac7 is used to propagate this constraint. Other constraints (distance constraints) are handled by a forward checking routine. So the results of our experiments (see table 3.4) are a comparison of how effectively gac3 and gac7 can propagate the quaternary action constraint.

Table 3.4: Performance of gac7 and gac3 on instances of the logistics problem, where P is the problem number, *time* is the CPU time, in seconds, spent solving the problem, and *visits* is the number of nodes visited during the search. A '-' means that the algorithm could not solve the problem given 1 hour of CPU time. Only problems solved by at least one of the algorithms are included in the table.

	gac7		gac3	
P	time	visits	time	visits
1	0.24	106	0.02	106
2	0.4	105	0.04	123
3	1.94	210	0.11	226
4	2.77	234	0.15	255
5	80.63	13198	5.55	13200
7	1.11	164	0.08	164
8	2.42	166	0.11	177
11	0.72	202	0.04	202
12	1.74	170	0.11	170
13	7.96	275	0.48	292
14	11.46	457	0.56	457
15	-	-	448.63	264777
16	2.89	276	0.16	304
17	1.62	330	0.09	330
20	1978.70	18235	114.71	18430
21	12.65	441	0.59	472
24	1.41	187	0.08	200
31	0.03	45	0.01	45
32	0.06	71	0.01	71
33	0.33	138	0.01	136
34	1.63	284	0.10	284
35	-	-	2478.71	5519278

On instances of the logistics problem, gac3 was found to consistently perform better than gac7 by a significant margin. The most difficult problem that can be solved by both algorithms is number 20. This problem is solved 17.2 (1978.70/114.74) times faster by gac3 than by gac7.

As mentioned in section 3.3, gac7 establishes arc consistency before backtracking begins and therefore enforces a slightly stricter form of arc consistency than does gac3. This can be seen as a strength or as a weakness of the algorithm. This extra work can have the effect of significantly reducing the number of nodes visited during backtracking. On the other hand, it may be the case that the extra work is done without a significant reduction in the search space. Consider the results in table 3.4. For each problem the number of visits performed by gac7 is very close (and in some cases identical) to the number of visits performed by gac3. On problem 20, for example, a reduction of less than 2% in nodes visited is achieved by the stricter consistency (18235 versus 18430). The extra work does not seem to pay off in the case of this problem. It seems likely that the reason for this is the looseness of the action constraints. This is explored in the next section, using random problems.

3.4.3 Random problems

The experiments reported in section 3.4.1 showed that gac7 performs much better that gac3 on some problems. The experiments reported in section 3.4.2 showed that gac7 performs much worse on other problems. The purpose of the experiments presented here is to explore where one algorithm performs better than the other.

To begin we compare the characteristics of the crossword puzzle problem with those of the logistics problem. Recall, from section 2.1, that the tightness of a constraint is defined as the number of tuples that satisfy the constraint divided by the size of the space of all tuples. The letter model of the crossword puzzle problem has very tight constraints. For example, in the case of the *words* dictionary, a constraint on four variables corresponds with a set of 2237 satisfying tuples (because these are 2237 four letter words in the dictionary). The total space of all tuples is 26^4 or 456976. This corresponds to a tightness ratio of $2237/456976 = 0.005$. On the other hand the major constraint in the logistics problem is very loose, as an example the tightness ratio, for one of the action constraints for problem 6, is 0.922 (138060/149688).

To investigate the correlation between tightness and the performance of the two algorithms we performed a series of experiments on random problems. The data is presented in tables 3.5 to 3.10. Each data point represents the median value from 11 runs. In each of these experiments, except where otherwise specified, the parameter u is set to ∞.

Table 3.5: A comparison of gac7's performance and gac3's performance on random problems with 20 variables each with domain size 25, and 40 constraints each of arity 3. The first column in the table (t) is the tightness of the constraints. A '-' means that the algorithm could not solve the particular problem within 2 hours of CPU time.

	gac7		gac3	
t	time	visits	time	visits
0.1	550.00	623	-	-
0.2	-	-	-	-
0.3	16.43	28	1257.83	1690
0.4	10.64	21	8.43	24
0.5	11.41	20	5.15	21
0.6	13.52	20	6.49	20
0.7	13.64	20	6.82	20
0.8	12.89	20	6.23	20
0.9	11.02	20	6.65	20

These experiments demonstrate areas where gac7 performs better than gac3. In particular, where the constraints are tight gac7 seems to perform well relative to gac3. To take just one example, consider table 3.6. On instances where the tightness ratio is set to 0.1, gac7 has a median time of 1941.87 seconds and gac3 has a median time of more than 7200 seconds (which was the CPU time limit we set).

At the very loose end of the spectrum, gac3 performs better than gac7. As a representative example, again consider table 3.6. On instances where the tightness ratio is set to 0.7, gac7 has a median time of 10.18 seconds and gac3 has a median time of 3.50. However, at this end the problems also become relatively easy (for both algorithms) to solve. In fact, we were not able to generate (hard) problems on which gac3 performs significantly better than gac7.

In the case of the logistics problem, even though the constraint that gac3

Table 3.6: A comparison of gac7's performance and gac3's performance on random problems with 20 variables each with domain size 25, and 30 constraints each of arity 3. The first column in the table (t) is the tightness of the constraints. A '-' means that the algorithm could not solve the particular problem within 2 hours of CPU time.

	gac7		gac3	
t	time	visits	time	visits
0.1	1941.87	3580	-	-
0.2	10.57	33	348.70	1052
0.3	6.86	21	5.59	27
0.4	7.68	20	3.70	22
0.5	8.78	20	3.11	20
0.6	10.21	20	3.43	20
0.7	10.18	20	3.50	20
0.8	9.36	20	3.48	20
0.9	8.04	20	4.15	20

and gac7 are being compared on is very loose, the problems are not easy. The reason for this seems to be the interaction with the other constraints that are being forward checked. The result is that the extra work done by gac7 (maintaining the various data structures on this very loose constraint) does not pay off, neither in a significant reduction of the number of nodes visits, nor in a reduction in the amount of work processing each node.

3.5 Conclusions

In this chapter we have experimentally compared two arc consistency algorithms: gac3 and gac7. This comparison showed that gac7 is an effective arc consistency enforcing algorithm on some types of constraints. In particular, gac7 performed better than gac3 on the letter model of the crossword puzzle problem as well as on some random problems.

We found that gac3 performed better than gac7 on all instances of the logistics problem. We also showed that gac3 is a more flexible algorithm in that the level of consistency enforced can be varied, and that it consumes less memory.

Table 3.7: A comparison of gac7's performance and gac3's performance on random problems with 15 variables each with domain size 15, and 30 constraints each of arity 4. The first column in the table (t) is the tightness of the constraints. A '-' means that the algorithm could not solve the particular problem within 2 hours of CPU time.

	gac7		gac3	
t	time	visits	time	visits
0.2	-	-	-	-
0.3	-	-	-	-
0.4	127.88	86	869.93	690
0.5	24.00	18	27.20	54
0.6	28.37	16	8.68	16
0.7	24.34	15	5.44	15
0.8	23.11	15	6.34	15
0.9	19.14	15	4.41	15

Table 3.8: A comparison of gac7's performance and gac3's performance on random problems with 30 variables each with domain size 25, and 20 constraints each of arity 4. The first column in the table is the number of satisfying constraints (out of a possible 390625). The number in brackets beside gac3 is the value of u. A '-' means that the algorithm could not solve the particular problem within 2 hours of CPU time.

	gac7		gac3 (2)		gac3 (∞)	
tuples	time	visits	time	visits	time	visits
1000	160.78	346	-	-	4857.40	7723
2000	706.96	747	-	-	-	-
2500	-	-	-	-	-	-
5000	-	-	-	-	-	-
7500	363.12	234	-	-	-	-
10000	97.54	70	-	-	-	-
12500	24.61	36	-	-	-	-
15000	40.53	42	-	-	-	-
25000	43.03	33	2930.23	7420	63.25	48
35000	57.07	30	31.55	90	56.94	37
45000	64.70	30	47.40	64	50.02	32
55000	73.90	30	67.64	92	65.12	37

Table 3.9: A comparison of gac7's performance and gac3's performance on random problems with 30 variables each with domain size 25, and 40 constraints each of arity 4. The first column in the table is the number of satisfying constraints (out of a possible 390625). The number in brackets beside gac3 is the value of u. A '-' means that the algorithm could not solve the particular problem within 2 hours of CPU time.

	gac7			gac3 (2)			gac3 (∞)		
tuples	time	visits	solns	time	visits	solns	time	visits	solns
1000	33	25	11	-	-	2	509	528	11
2000	758	518	11	-	-	0	6298	1855	6
2500	1724	717	11	-	-	0	-	-	0
5000	-	-	2	-	-	0	-	-	0
7500	-	-	2	-	-	0	-	-	0
10000	-	-	0	-	-	0	-	-	0
12500	-	-	0	-	-	0	-	-	0
15000	-	-	0	-	-	0	-	-	0
25000	-	-	0	-	-	0	-	-	0
35000	-	-	0	-	-	0	-	-	0
45000	-	-	0	-	-	0	-	-	0
55000	-	-	0	-	-	0	-	-	0

Table 3.10: A comparison of gac7's performance and gac3's performance on random problems with 50 variables each with domain size 25, and 20 constraints each of arity 4. The first column in the table is the number of satisfying constraints (out of a possible 390625). The number in brackets beside gac3 is the value of u. A '-' means that the algorithm could not solve the particular problem within 2 hours of CPU time.

	gac7		gac3 (2)		gac3 (∞)	
Tuples	time	visits	time	visits	time	visits
1000	22.03	136	-	-	-	-
2000	3.72	51	-	-	283.47	246
2500	5.71	52	-	-	170.96	328
5000	8.65	50	3283.29	79061	15.50	66
7500	12.24	50	2096.91	10170	12.78	51
10000	16.53	50	1229.42	12741	19.53	53
12500	20.14	50	750.88	2172	19.91	59
15000	22.75	50	84.37	452	20.23	52
25000	38.82	50	107.38	883	30.28	52
35000	50.67	50	22.39	75	29.34	52
45000	62.97	50	14.56	52	27.14	50
55000	68.22	50	16.18	51	29.17	50

Chapter 4

Estimating the cost of solving constraint satisfaction problems

4.1 Introduction

Experience has shown that the cost of solving constraint satisfaction problems can vary wildly, even for slightly different instances of the same problem. This gives rise to the question, *is it possible to accurately estimate the cost of solving a constraint satisfaction problem*?

The cost of solving a constraint satisfaction problem can be measured in several ways. One approach is to measure the CPU time required to solve the problem. It is also often convenient to consider the number of nodes visited during the search or the number of consistency checks performed. Also, the meaning of *solving* the problem can vary. The goal may be to find *one* solution (i.e. the first solution) to a problem, or the goal may be to find *all* solutions to a problem. Finding all solutions is essentially a complete search of the space. Solving optimization problems, such as the Golomb problem, often involves showing that there is no solution to some formulations of the problem and finding one solution to the formulation corresponding to the optimal. Solving the 11 mark Golomb problem, for example, involves proving that there is no solution for formulations corresponding to ruler lengths of less than 72 and that there is at least one solution to the formulation corresponding to a ruler of length 72.

In this chapter we explore techniques for estimating or predicting the *cost* of solving a constraint satisfaction problem. We discuss both solving for one solution and solving for all solutions, though our focus is on the all solutions case. In particular, we are interested in a tool that could be given a problem instance and an algorithm, and would return an estimate of the size of the tree (or some other measure of the cost) that would be searched by the algorithm in solving the given instance. This is different from related efforts to explore techniques for estimating the average cost of finding solutions over an entire problem class.

Our major contributions in the area of estimation are listed here.

- We identify and discuss a comprehensive list of possible applications for an estimator.
- We evaluate several analytic techniques that have been proposed as estimation techniques.
- We develop an estimator for backtracking-based algorithms. Our estimator is based on Purdom's statistical sampling technique, which in turn is based on a similar technique due to Knuth. We also provide an evaluation of our estimator.
- We develop and evaluate a new domain independent dynamic variable ordering algorithm based on our estimator.

4.1.1 Applications of an estimator

Here we discuss several possible applications for an estimator. This discussion will motivate the rest of this chapter. It surveys previous work done in this area.

- **Algorithm selection** Given a formulation of a problem as a CSP, several algorithms are available for solving it. For sufficiently difficult problems, algorithm selection is important. Selecting an algorithm involves selecting a variable ordering heuristic, selecting a value ordering heuristic, and determining what level level of consistency should be enforced.

In [17], Lobjois and Lemaitre propose a method called *selection by performance prediction* (SPP), which is based on Knuth's sampling method (described in section 4.5.1). Lobjois and Lemaitre's work is in the area of *maximal constraint satisfaction problems*, which can be solved by one of several branch and bound algorithms. The four branch and bound algorithms discussed in [17] differ in variable orderings, value orderings and in the type of consistency algorithm used. Given a problem, P, SPP uses an estimate of the time each of these algorithms will take to solve P, to select which algorithm to use. Lobjois and Lemaitre found that SPP works very well in practice. Their results demonstrate that it is better to use the algorithm selected by SPP than to use the best algorithm on average for all instances.

Other competing approaches to selecting the best algorithm include Tsang's work on constructing maps [27], Minton's work on configuring constraint satisfaction programs [21], and Frost's work on finding the best algorithm [12] for binary problems. The general approach in these papers involves mapping an instance to some class of problems based on one or more parameters, then the best algorithm for the class is selected as the algorithm to use on the instance.

- **Model selection** For a given problem, there is generally more than one way to model it as a CSP. The task of modeling involves choosing variables and constraints. In the crossword puzzle problem, described in section 2.2.1, we identified three possible ways to model the problem as a CSP. In the letter model the variables represent letters in the puzzle and constraints enforce that adjacent letters form words. In the dual model the variables represent words in the puzzle and constraints enforce that words agree on intersection letters.

  Selecting between alternative problem models is often difficult, and could be a useful application of an estimator. Also the process of tuning a model (adding redundant constraints for example) could benefit from the use of an estimator.

Some work was done in this area by Borrett [5]. He uses Nadel's estimator (see section 4.3.2 for a description of this estimator) to determine which redundant constraints should be added to a problem.

- **Dynamic variable ordering** It is possible that an estimator could be used as a variable ordering. That is, it could be used during backtracking to determine which variable to instantiate next. For more on this see section 4.6.
- **Value ordering heuristic** A value ordering heuristic can be used during backtracking to determine which order the values should be tried in. An estimator could be used to estimate the size of the subtrees beneath each value for the current variable. Bart Selman, in a personal communication, suggested that asymmetries in the size of these trees may provide a useful heuristic to guide value ordering.
- **Predicting** An estimator may be useful in predicting whether a given problem instance has a solution. It may also be useful in predicting whether an instance would be solved by the backtracking algorithm in a reasonable amount of time. For some work in this area see [16].
- **Thrashing detection** Bart Selman, in a personal communication, suggested that it would be valuable to explore the usefulness of an estimator in a restart algorithm. In particular it may be useful to use an estimator to detect *thrashing.* This would involve answering the question *how much progress is the algorithm making*?

Barbara Smith, in a personal communication, points out that for many of the applications mentioned above it is not necessary for an estimator to be an accurate predictor of the cost of solving a problem. Rather it is important that the estimates are *relatively* correct. That is, given instances P_1 and P_2, if the actual cost of solving P_1 is less than the actual cost of solving P_2, then for a relatively correct estimator, the estimated cost of solving P_1 would be less than the estimated cost of solving P_2. This would mean that decisions made, based on these estimates, such as which model to use, would be correct.

A significant amount of work has been published on estimating the cost of finding *all* solutions to a problem. Three basic approaches are used: *hard problem* identification, *statistical models*, and *stochastic sampling*. We discuss each of these approaches below. Less work has been published on the more difficult task of estimating the cost of finding *one* solution. In section 4.4 we briefly discuss some work on the one solution case.

4.2 Hard problems

In [7], Cheesman, Kanefsky and Taylor discuss hard problem generation and identification. They find that hard instances of NP-complete problems typically occur at critical values of some "order parameter". For example instances of the 3-colorability problem can be ordered based on average connectivity, which is the average number of edges per node. Cheesman et al. find that 3-colorability is hard on average in a certain range of values for average connectivity. Similar *hard* regions exist for other NP-complete problems as well. Cheesman et al. conjectured that:

> "because it is possible to locate a region where hard problems occur, it is possible to predict whether a particular problem is likely to be easy to solve."

In other words, knowing one parameter for an instance would very coarsely give an estimate for the cost of solving the problem by predicting whether it would be "easy" or not.

More recent work takes a different view on the usefulness of these order parameters. For example, in [14] Hogg reports that:

> "A readily computed measure of problem structure predicts the difficulty of solving the problem, on average. However, this prediction is associated with a large variance and depends on the somewhat arbitrary choice of the problem ensemble. Thus these results are of limited direct use for individual instances."

It is generally agreed today, that a single order parameter can not accurately predict the difficulty of an instance of an NP-complete problem. For more on this topic see [9], a survey by Cook and Mitchell.

4.3 Analytic or statistical models for finding all solutions

Statistical modeling is another approach to estimating that has been discussed in the literature. This approach involves developing a mathematical model of the algorithm. Building such a model typically involves making simplifying assumptions about the problem or the search space. Once the model has been selected, formulas can be derived for the expected cost (total number of nodes visited, for example) of finding all solutions to instances of the problem for a given algorithm.

Though a significant amount of work has been published in this area, we will discuss only a representative sample of this work. We will discuss, in detail, Haralick and Elliot's work (see [13]), and Nadel's work (see [22]) since these deal specifically with CSP's. First however we will mention briefly some other significant contributions. This will illustrate the types of assumptions that are made.

In [26], Stone and Sipala, analyze backtracking search (they refer to this as depth-first search). They assume:

1. uniform branching factor of 2,
2. p, the probability of a node being expanded, is fixed and greater than 0.5, and
3. the probability of an arbitrary leaf node being successful is

   $$p^{NLEFT}(1-p)^{NRIGHT},$$

   where $NLEFT$ is the number of left hand branches on the path from the root to this leaf and NRIGHT is the number of right hand branches on this path.

The assumption that the probability of a node being expanded is independent of depth is a very strong assumption. In [23] Nicol observes that larger partial solutions are more difficult to expand than smaller partial solutions. So he assigns an *extension probability, p,* such that a node expands if $p < \frac{1}{n}$, where n is the current depth of the search. Nicol's analysis also allows for a random number of children rather than assuming a uniform branching factor of 2 as Stone and Sipala do.

Other significant contributions include, Franco and Paull's analysis of the Davis Putnam procedure (DPP) (see [11]) and Bender and Wilf's analysis of the graph coloring problem (see [2]). Franco and Paull describe two different distributions of the SAT problem, which they call G and F. Each distribution represents different assumptions about the problem space. The analysis shows that for distribution G, DPP's expected complexity is *polynomial,* while for distribution F, DPP's expected complexity is *exponential.*

Bender and Wilf analyzed backtracking on graph coloring decision problems. They describe a parameterized distribution of the problem space, $X_n(p)$. Their analysis shows that when $p = \frac{1}{2}$ the expected complexity is *constant.* However over all graphs the expected complexity is *exponential.*

4.3.1 Haralick and Elliott's approach

In [13], Haralick and Elliott develop a model for binary CSP's. In this model it is assumed that all domains are the same size M. They also assume that a pair of labels is consistent with probability p. Using this model Haralick and Elliott analyze the *backtracking* and the *forward checking* algorithms. Here we present a summary of their analysis.

Backtracking

Haralick and Elliott observe that backtracking visits a node iff its parent is consistent. The probability that a node at level k has a consistent parent is given by,

$$p^{(k-1)(k-2)/2}$$

since consistent nodes must satisfy consistency checks between every pair of variables. The expected number of nodes at level k is given by,

$$N(k) = m^k p^{(k-1)(k-2)/2}$$

since there are m^k possible nodes at level k. At each node a value must be tested for consistency with the values given to the previous $k-1$ variables. With probability $1-p$ the first consistency check fails and only one consistency check is performed. With probability $p(1-p)$ the first consistency check succeeds and the second consistency check fails and so only two consistency checks are performed. And so on. Hence the expected number of consistency checks performed at each node is,

$$\sum_{i=1}^{k-1} ip^{i-1}(1-p) + (k-1)p^{k-1},$$

which can be simplified to

$$\frac{1-p^{k-1}}{1-p}.$$

Thus the expected number of consistency checks at level k is given by

$$C(k) = N(k)\frac{1-p^{k-1}}{1-p}.$$

The expected number of solutions is just the number of nodes at the deepest level in the search tree, namely

$$m^n p^{n(n-1)/2}.$$

Forward Checking

Haralick and Elliott observe that the forward checking algorithm visits a node iff it is consistent and its parent is consistent with all future variables. The probability that a node is consistent is given by

$$p^{k(k-1)/2}$$

since consistent nodes must satisfy $\frac{1}{2}k(k-1)$ consistency checks. The probability that a node is consistent and its parent is consistent with all future variables is given by

$$p^{k(k-1)/2}[1-(1-p^{k-1})^m]^{n-k}.$$

Thus the expected number of nodes at level k is given by

$$N(k) = m^k p^{k(k-1)/2}[1 - (1 - p^{k-1})^m]^{n-k}.$$

Thus the expected number of consistency checks at level k is given by

$$C(k) = N(k)\frac{mp^{k-1}}{1 - (1 - p^{k-1})^m} \times \frac{1 - [1 - (1 - p^k)^m]^{n-k}}{(1 - p^k)^m}.$$

The assumptions made by Haralick and Elliott in their analysis are quite strong. In particular the uniform consistent probability assumption is a serious weakness of this model. The model used by Nadel allows this assumption to be dropped.

4.3.2 Nadel's approach

Nadel's work, which builds on the work by Haralick and Elliott described above, provides a more sophisticated analysis of binary CSP's. In this model a given pair of variables x_i and x_j with a given pair of values is consistent with probability p_{ij}. The domain sizes of each variable are given by M_i. Nadel used this model to analyze the backtracking [22] and forward checking algorithms [24].

Backtracking

The probability that a node at level k has a consistent parent is given by

$$\prod_{1 \leq i < j \leq k-1} p_{ij}$$

since consistent nodes must satisfy consistency checks between every pair of variables. Thus, the expected number of nodes at level k is given by

$$N(k) = \prod_{i=1}^{k} m_i \prod_{1 \leq i < j \leq k-1} p_{ij}.$$

At each node a value must be tested for consistency with the values given to the previous $k - 1$ variables. With probability $1 - p_{1k}$ the first consistency check fails and only one consistency check is performed. With probability $p_{1k}(1 - p_{2k})$ the first consistency check succeeds and the second consistency check fails and only two consistency checks are performed. And so on. Hence the expected number of consistency checks performed at each node is

$$\sum_{i=1}^{k-1}\left(i(1-p_{ik})\prod_{j=1}^{i-1}p_{jk}\right)+(k-1)\prod_{i=1}^{k-1}p_{ik}.$$

Thus the expected number of consistency checks at level k is given by

$$C(k)=N(k)\left[\sum_{i=1}^{k-1}\left(i(1-p_{ik})\prod_{j=1}^{i-1}p_{jk}\right)+(k-1)\prod_{j=1}^{k-1}p_{jk}\right].$$

The expected number of solutions is given by

$$\prod_{i=1}^{n}m_i\prod_{1\leq i<j\leq n}p_{ij}.$$

Nadel provides experimental results to demonstrate the accuracy of these formulas. This is done by experimenting with various formulations of the N-Queens problem for 3, 4 and 5 queens. We reproduced some of his experiments using up to 12 queens. Figure 4.1 shows the relative error for consistency checks performed and nodes visited. The relative error is given by

$$\frac{|actual-expected|}{actual}.$$

In the region of the graph reported in [22] the relative error is small, meaning that the expected values are close to the actual values. However the relative error increase significantly as the problem size (the number of queens) grows.

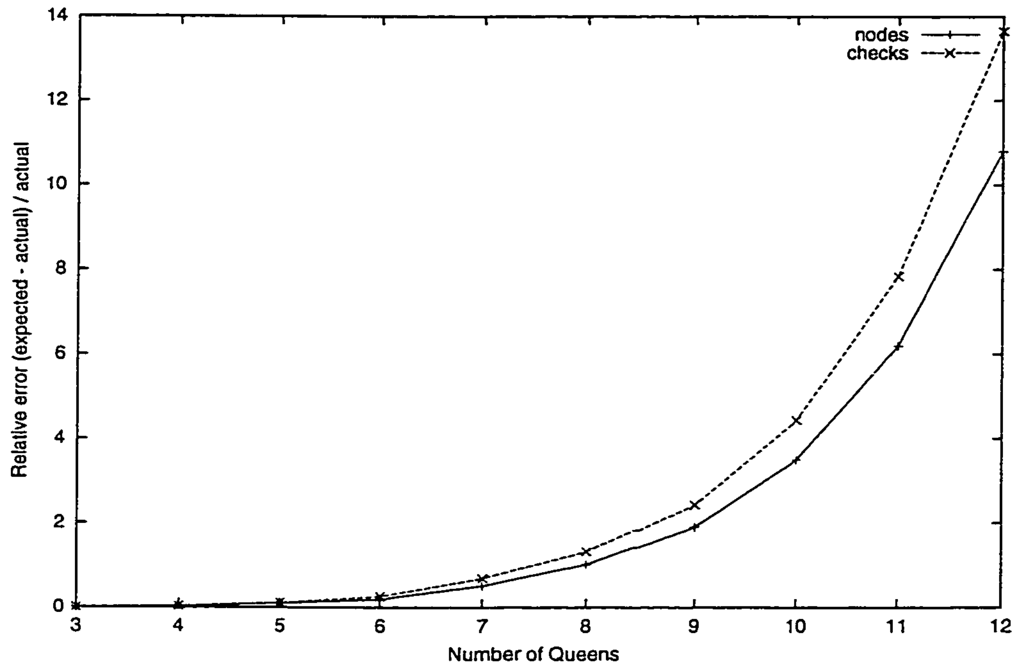
Forward Checking

In [24], Nadel describes his aims in analyzing the forward checking algorithm (in this quote clp refers to the Consistent-Labeling Problem, which is another name for CSP's):

> "What we ideally want is the complexity for solving a *specific* clp (by some given algorithm). Since this appears difficult to obtain we might aim for the expected complexity of solving clps in an equivalence class to which our problem belongs."

The parameters used to partition the space into equivalence classes are the number of variables (n) the domain sizes of the variables (M_i is the domain

Figure 4.1: Relative error of Nadel's estimator, for backtracking on various instances of the N-Queens problem.



size of variable i) and the compatibility count matrix I. The number of satisfying tuples for the constraint between variables i and j is I_{ij}. R_{ij} is the compatibility ratio ($R_{ij} = I_{ij}/(M_i M_j)$). We have previously defined this as the tightness ratio of a constraint.

Nadel's analysis allows for a static but specifiable variable ordering, A_0. A_k is the first k variables in the list A_0. F_k is the list of $n-k$ variables not yet assigned at level k. The analysis also allows for a static but specifiable consistency checking order. The list G_0 is a list of variables in which order future variables will be checked. The order in which future variables are considered for forward checking at level k is $G_k = G_0 - A_k$.

Then the expected number of nodes visited is at level k is given by

$$N_k = (\prod_{i \in A_k} M_i)(\prod_{i<j \in A_k} R_{ij})(\prod_{f \in F_k} S_f^{(k-1)}),$$

where $S_f^{(k)}$ is the probability that f has at least one of its M_f possible labels consistent with the instantiations up to level k. Since there are no forward checkable constraints at level n, the expected number of nodes visited at level

n (which is also the expected number of solutions) is given by

$$S = (\prod_i M_i)(\prod_{i<j} R_{ij}).$$

The expected number of consistency checks performed at level k is given by

$$C_k = N_k c_k,$$

where c_k is the expected number of consistency checks performed at each node at level k. This is given by

$$c_k = [\sum_{i=1}^{|G_k|} (M_k^{g_{ik}} \prod_{j=1}^{i-1} S_{g_{ik}}^{(k)} / S_{g_{ik}}^{(k-1)})].$$

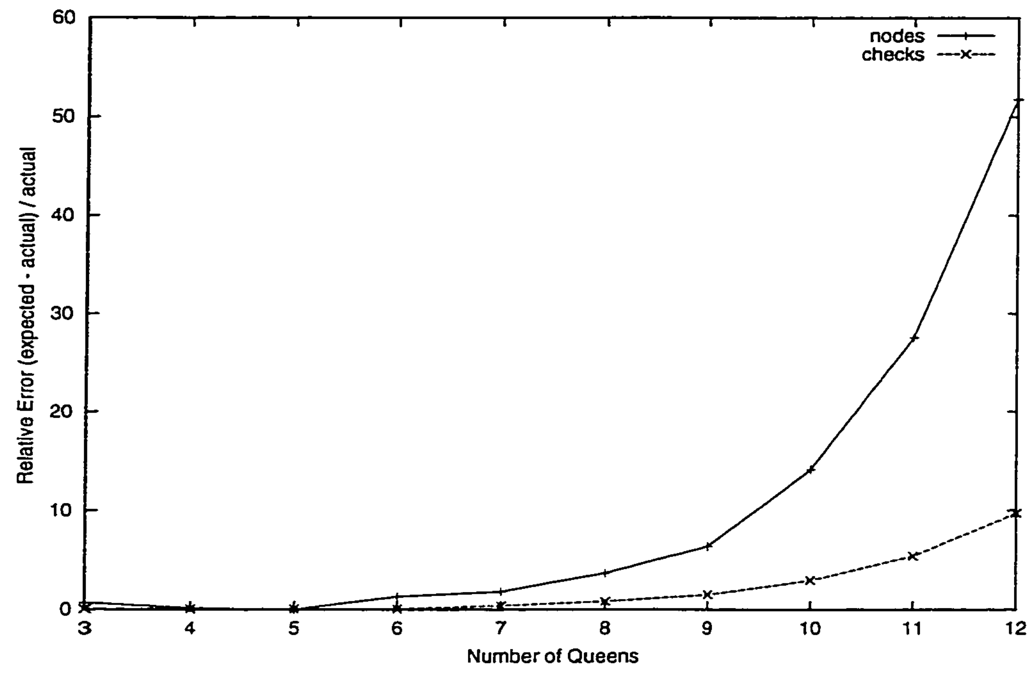
Nadel [24] reports on some experimentation used to explore the usefulness of his analysis in predicting the cost of solving CSP instances. The 5-Queens problem is used in these experiments (as well as some random problems). To further evaluate the accuracy of the formulas, we experimented with the N-Queens problem using from 3 up to 12 queens. Figure 4.2 shows the relative error. As in the case of backtracking, the relative error grows as the problem size grows. It is worth noting here that the 12-Queens problem is a trivial problem.

4.3.3 Discussion

The accuracy of the analysis is very sensitive to changes in the assumptions about the problem. This is demonstrated by Franco and Paull in [11]. They showed that DPP's expected complexity can vary between polynomial and exponential depending on assumptions about the search space.

This type of analysis has not been done for newer algorithms that enforce stronger levels of consistency, such as arc consistency or arc consistency with conflict-directed backjumping. This analysis has also not been done for non-binary constraint satisfaction problems. While doing this analysis may have some value, it does not seem to be an effective tool for predicting the cost of solving CSP instances. Experimental results are not encouraging, even for the relatively simple algorithms, backtrack and forward checking, and quite simple problems, small instances of the N-Queens problem, for example.

Figure 4.2: Relative error of Nadel's estimator for forward checking on various instances of the N-Queens problem.



In [6] Brown and Purdom carry out an analysis of backtracking. They suggest that to obtain an accurate estimate for the running time of a given backtrack program, Knuth's method should be used. Knuth's method falls into the category of *simulation or statistical sampling* discussed in section 4.5.

4.4 Analytic or statistical models for finding one solution

Estimating the cost of finding one solution to a given problem has proved to be a difficult problem. Van Liempd and van den Herik's have published some work on this topic. Bacchus also did some work in this area, though it was not published. Bacchus found that the analytic method he was developing did not provide a useful estimate of the cost of finding the first solution to a problem.

In [29] and [30], van Liempd and van den Herik build on Haralick and Elliott's work discussed in section 4.3.1. It is based on the assumption that a pair of labels is consistent with some uniform probability p, independent of the variables or values involved and independent of any past processing. Van Liempd and van den Herik use this model to derive estimates for the cost of finding one solution to a CSP using *backtracking* and *min forward checking.*

Van Liempd and van den Herik report on some experiments to test the accuracy of their estimates. They found that for random CSP's the formulas accurately predicted the cost for both of the algorithms considered. For the N-Queens problem the predicted cost was (some what) close to the actual cost for the min forward checking algorithm. However for the backtracking algorithm the predicted cost was not close to the actual cost. Van Liempd and van den Herik attribute this difficulty to the assumptions mentioned above.

Van Liempd and van den Herik conclude that these formulas can be used to select which search algorithm (i.e. backtracking or min forward checking) should be used for a given instance of a CSP. This conclusion is based on experimentation with random problems. However in the case of N-Queens it was shown that the estimates were not as accurate. Results from [29] are reproduced in table 4.1. It is clear that using the predicted costs to guide algorithm selection would be misleading. For example, the predicted cost for solving the 20-Queens problem using backtracking is less than the predicted cost for solving the same problem using forward checking. However the measured costs are much less for forward checking than for backtracking.

Table 4.1: Predicted and measured costs for solving various sizes of the N-Queens problem using standard backtracking (bt) and min forward checking (fc). This data is taken from table 1 in [29].

		backtracking		forward checking	
n	p	predicted	measured	predicted	measured
10	0.77	1.8×10^2	3.2×10^3	3.2×10^2	5.7×10^2
20	0.89	5.6×10^2	2.5×10^7	2.1×10^3	4.1×10^3
30	0.92	1.2×10^3	1.6×10^{10}	7.1×10^3	9.1×10^3
40	0.94	2.1×10^3	–	1.7×10^4	1.9×10^4

This work exposes some difficulties in estimating the cost of finding one solution. Van Liempd and van den Herik, like Nadel, take an analytic approach. To do this they make several assumptions about the search space. When random problems are generated, based on these assumptions about the search space, the estimates are reasonably accurate. However on arbitrary problems these assumptions may not hold as well, and so the estimates can be expected to be much less accurate.

4.5 Simulation or statistical sampling techniques

Simulation or statistical sampling has been suggested as a method for estimating the cost of tree searches. The foundation for this work was laid by Knuth. Follow up work was done by Purdom and by Chen.

4.5.1 Knuth's approach

In [15] Knuth explains his motivation for pursuing his study of estimators:

> "Sometimes a backtrack program will run to completion in less than a second, while other applications seem to go on forever... These great discrepancies in execution time are characteristic of backtrack programs, yet it is usually not obvious what will happen until the algorithm has been coded and run on a machine."

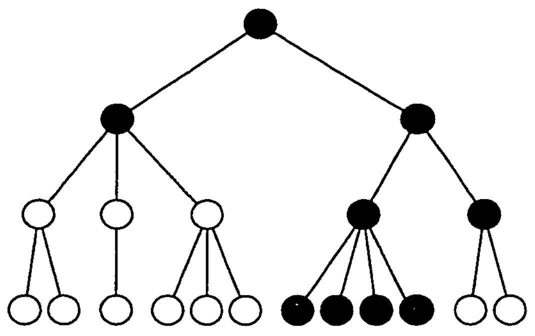
The basic approach, used by Knuth, involves taking samples of the search tree. A sample is taken by exploring one path from the root of the tree to a leaf of the tree. The path is chosen by selecting a successor at random. During the exploration statistics are gathered about the size of the tree. The statistics of interest to Knuth are: the cost of processing a node, and the number of children. If the number of children at level i is d_i then

$$(1) + (d_1) + (d_1 \times d_2) + ... + (d_1 \times ... \times d_n)$$

gives an estimate of the number of nodes in the tree.

Figure 4.3 shows one sample of a search tree. The black nodes represent a random path chosen as a sample of the tree. The grey nodes are the nodes that the sampling algorithm *knows* about (by checking the number of children that each of the black nodes has) but does not explore. Given this sample of the tree

Figure 4.3: A sample of a search tree.



Knuth's algorithm would estimate the size of the tree as $1+2+2\times2+2\times2\times4$, which is 23. The actual number of nodes in this search tree is 20. An estimate of the size of the tree (or an estimate of the cost of searching the entire tree) can be obtained by averaging over many such samples.

Now, to make this more precise we next introduce some of Knuth's notation.

- P is the predicate that a solution must satisfy.
- P_k is the predicate that a partial solution must satisfy at level k.
- T is the search tree explored by the search procedure. That is, all partial solutions that satisfy the appropriate P_k, which is the set $\{(x_1, ..., x_k) | k \geq 0 \wedge P_k(x_1, ..., x_k)\}$.
- The value $c(t)$ is the cost of processing node $t \in T$. Various measures of the cost are possible including consistency checks and CPU time. The value $c()$ is the cost of processing the root node.
- The value $cost(T)$ is the cost of searching T, or in other words, the cost of processing each node in T, which is just $\sum_{t \in T} c(t)$.
- S_k is the set of all successors to the node at level k.
- The value d_k is the number of children at level k.

Then, given a tree T, the goal is to estimate *cost*(T). Knuth's algorithm to produce this estimate is listed in algorithm 8. When the algorithm completes, the value of variable C is returned as an estimate for *cost*(T) (see line 8 of the algorithm). If $c(t)$ is set to 1 for all $t \in T$, then *cost*(T) is an estimate of the size of the tree. If $c(t)$ is the number of consistency checks performed, while processing t, then *cost*(T) is an estimate of the number of consistency checks that would be performed during the search. Similarly setting $c(t)$ to be the amount of CPU time taken to process the node t, produces an estimate of the CPU time that would be required to solve the problem.

Algorithm 8 Knuth's Algorithm

```
1:  $k \leftarrow 0$
2:  $D \leftarrow 1$
3:  $C \leftarrow c()$
4:  **while** true **do**
5:      $S_k \leftarrow \{x_{k+1} | P_{k+1}(x_i, ..., x_{k+1})\}$
6:      $d_k \leftarrow |S_k|$
7:      **if** $d_k = 0$ **then**
8:          return C
9:      **end if**
10:     $x_{k+1} \leftarrow$ random element of $S_k$
11:     $D \leftarrow d_k D$
12:     $C \leftarrow C + c(x_1, ..., x_{k+1})D$
13:     $k \leftarrow k + 1$
14: **end while**
```

4.5.2 Purdom's enhancements

In [25], Purdom builds on Knuth's work described above. Purdom finds that for trees that have many nodes with no successors at each level, Knuth's algorithm has difficulty learning about the deeper levels of the tree. To overcome this difficulty Purdom proposes a sampling algorithm that he calls *partial backtracking.* While Knuth's algorithm never backtracks and explores exactly one path from the root to a leaf, Purdom's algorithm allows the exploration of some specifiable number (w) of children of each visited node. When $w = 1$ then Purdom's algorithm is equivalent to Knuth's. Purdom showed that for some types of trees his algorithm produces significantly better estimates of the tree size.

Algorithm 9 PartialBacktracking(k)

```
1: if at leaf node then return
2: $v \leftarrow PickVariable()$
3: $Q \leftarrow$ all values in $D(v)$
4: $a \leftarrow |Q|$
5: $m \leftarrow min(a, w)$
6: $d_k \leftarrow d_{k-1}a/m$
7: $S \leftarrow m$ randomly chosen elements of $Q$
8: for all $b$ in $S$ do
9:     assign $v$ the value $b$
10:    if CheckConstraints() then
11:        PartialBacktracking(k+1)
12:    end if
13:    unassign $v$
14: end for
```

The efficiency and accuracy of Purdom's approach both depend on the choice of w. As w grows partial backtracking approaches complete backtracking, and the accuracy, of course, increases while the efficiency decreases. An adaptation of Purdom's algorithm is listed algorithm in 9.

4.5.3 Chen's enhancements

In [8] Chen's goal is to provide techniques to determine the feasibility of a particular search program. In particular the goal is to estimate the sum

$$\varphi = \sum_{s \epsilon S} f(s),$$

where S is the set of states or nodes in the tree. If $f(s)$ is the cost of processing a node s then φ is the cost of traversing the tree.

To this end Chen generalizes Knuth's sampling technique. He adopts a *stratified sampling* (or *heuristic sampling*) approach based on a "heuristic function" called a *stratifier.* Basically the stratifier's role is to categorize the nodes of the tree. More specifically, it is a heuristic function $h : S \to P$ that maps each state s to a stratum $h(s)$ in a partially ordered set P. The function h should be strictly decreasing along each edge of the tree. Then, given a tree that is stratified under h,XS a heuristic sampling algorithm will sample each stratum α for a representative node s_α and simultaneously to obtain an estimate $w_\alpha = w(s_\alpha)$ for the number of nodes in that stratum. Then an unbiased estimate of φ is

$$\phi = \sum_\alpha w_\alpha f(s_\alpha).$$

Chen finds that a good stratifier can significantly reduce the variance relative to Knuth's algorithm. The heuristic sampling algorithm is listed in algorithm 4. When $h(s) = -depth(s)$, which is the one obvious domain independent stratifier, then heuristic sampling is equivalent to Knuth's method.

Algorithm 10 Heuristic Sampling

```
1: $Q \leftarrow (root, 1)$
2: while $Q$ not empty do
3:    $(s, w) \leftarrow pop(Q)$
4:    for all children $t$ of $s$ do
5:       $\alpha \leftarrow h(t)$
6:       if $Q$ contains an element $(s_\alpha, w_\alpha)$ in stratum $\alpha$ then
7:          $w_\alpha \leftarrow w_\alpha + w$
8:          with probability $w/w_\alpha$ do $s_\alpha \leftarrow t$
9:       else
10:         insert a new element $(t, w)$ into $Q$
11:      end if
12:   end for
13: end while
```

4.5.4 Discussion

We have discussed three categories of estimation techniques: hard problem identification, analytical models, and statistical sampling. Of these three, statistical sampling seems to be the most promising. The next section will focus on further developing these techniques.

4.5.5 Our implementation

Knuth's sampling algorithm, as well as Purdom's algorithm, was designed to estimate the cost of finding all solutions to an instance using basic backtracking. Our goal was to apply similar techniques to more general backtracking *based* algorithms, such as forward checking and general arc consistency. While

Knuth's algorithm assumes a fixed variable ordering, it can be simply extended to allow for dynamic variable orderings.

The sampling algorithm that we implemented is heavily based on Purdom's sampling algorithm, described above. One difference is that our sampling algorithm allows for a complete search to some depth in the tree. The number of nodes at deeper levels is then estimated by applying a sampling algorithm on the subtrees below each of the nodes at that level. This technique is used in an effort to allow the algorithm to more frequently sample nodes deeper in the tree, and to avoid the repeated sampling of the most shallow levels. We use the notation, $nodes_i$, to represent the number of nodes in the tree at level i.

The behavior of our algorithm is controlled by four parameters.

1. *sample_time*

   The amount of time to spend sampling the tree. Alternatively, the parameter *samples* can be set which results in a fixed number of samples being taken.

2. *sample_depth*

   Depth at which to begin sampling. Above this level complete search is used. Then for all $i < sample\_depth$, the number of nodes at level i, $nodes_i$, is known exactly. For all $i \geq sample\_depth$, $nodes_i$ is an estimate of the actual number of nodes at level i. When $sample\_depth = 1$, our implementation samples repeatedly, beginning at the root, as Purdom's algorithm does.

3. *bf_ratio*

   A ratio specifying the number of children to sample at each node. The children to sample are chosen randomly.

4. *bf_max*

   The maximum number of children to sample at each node, branching factor notwithstanding. If a given node, visited during sampling, has c children, then the number of children visited by the algorithm will be the minimum of, $bf\_ratio \times c$ and bf_max. If the value of $bf\_ratio \times c$ is less than 1, then one child will be sampled.

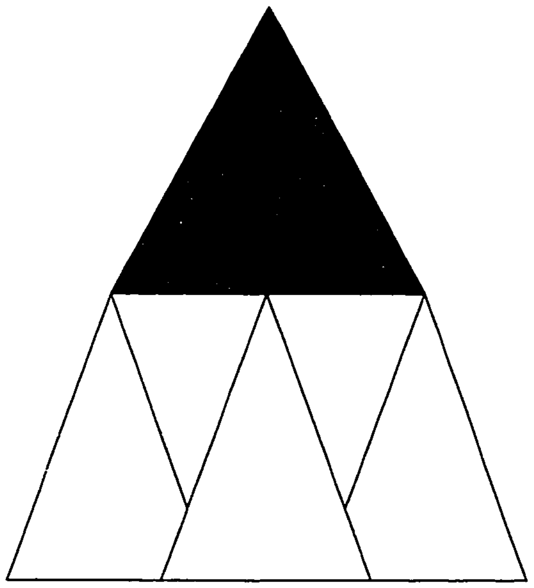
Algorithm 11 Estimate(*level*)

```
1: if $level = search\_depth$ then
2:    Sample( $level$ )
3:    return
4: end if
5: if $level > N$ then
6:    return
7: end if
8: $v$ = PickVariable()
9: for all $a$ in D($v$) do
10:    $nodes_{level} \leftarrow nodes_{level} + 1$
11:    assign $v$ the value $a$
12:    if EnforceConsistency() then
13:       Sample($level + 1$)
14:    end if
15:    unassign $v$
16: end for
```

Our sampling algorithm uses two main functions to produce an estimate: Estimate() (see algorithm 11) and Sample(). The portion of the function Estimate(), from lines five to eighteen, is nearly identical to the generic backtracking algorithm. One difference is that, on line ten the *nodes* data structure is updated. The PickVariable() function (on line 8) can be any variable ordering routine. The EnforceConsistency() function (on line 13) can be any consistency enforcing algorithm. In this way our estimator can be use to estimate the cost of solving an instance using an arbitrary backtracking-based algorithm.

Estimating begins by calling the function Estimate() with $level = 1$, and proceeds just like backtracking when $level < sample\_depth$. When $level = sample\_depth$, Estimate() calls the function Sample(), to sample the subtree below the current node. Sample() is essentially Purdom's partial backtracking algorithm (see algorithm 9) adapted to work on subtrees. This function produces an estimate of the number of nodes at each level in the subtree below the current node. Figure 4.4 shows a tree sampled using our technique. The

Figure 4.4: Regions of a search tree.



grey section of the tree is the portion above *sample_depth*. The number of nodes in this section is known exactly. The white sections of the tree in figure 4.4 represent the subtree's below each of the nodes at level *sample_depth*. An estimate for the size of the entire tree (below *sample_depth*) can be derived by summing the estimated size of each of these subtrees, by level. Then, when the function Estimate() completes,

$$ts = \sum_{i=1}^{n} nodes[i],$$

where n is the number of levels in the tree, is an estimate of the total size of the search tree.

The algorithm can be run on the instance for a short period of time to produce an estimate for the rate, r, at which nodes are visited. Then $ts \times r$ is an estimate of the time that would be required by the algorithm to solve the instance. It turns out that, in practice, this technique (which was used in [17]) is more accurate than maintaining statistics during sampling about the time to process each node.

4.5.6 Experiments

We performed some experiments to explore the accuracy of our estimator, as well as the trade offs between various values of the parameters that need to be calibrated. Random problems, Golomb problems, and logistics problems were used. In our experiments on random problems we are solving for all solutions in each instance. In the case of the Golomb and logistics problems used in these experiments, we are proving that no solution exists.

The accuracy of our estimates is measured in terms of *relative error*, which is calculated using the formula

$$\frac{|actual - estimate|}{actual}.$$

As an example, if the actual size of a tree is 1.60×10^5, and the estimated size is 8.0×10^4, then the relative error is

$$\frac{|1.60 \times 10^5 - 8.0 \times 10^4|}{1.60 \times 10^5}$$

which is 0.5.

Random problems

In general, the trees built to solve random problems are easy to sample. Sampling assumes that the size of the entire tree can be inferred from a small sample of the tree. Random problems tend to generate quite uniform trees and so this assumption holds. To demonstrate the algorithm's accuracy with random problems, we report on a few experiments with these problems.

To choose values for the sampling parameters (described above) to use in our experiments we calibrated using three small problems. Because these problems were small, it was easy to get the actual sizes of the corresponding search trees. This was done by trying various values for *sample_depth* (1, 2, and 3), *bf_ratio* (0.25, 0.5, 0.75, and 1.0), and *bf_max* (1, 2, 3, and 4), and choosing the values that produced the most accurate estimates. The values chosen for these experiments were:

- *sample_depth* = 2,

- *bf_ratio* = 1.0, and
- *bf_max* = 2.

With these values, sampling will begin at depth 2 and will visit 2 children of each node that is visited (unless there are fewer than 2 children). Using these settings we obtained the results in table 4.2.

Table 4.2: Tree size estimates versus actual size for random problems with 31, 32, 33 and 35 variables. In all cases domains of size 40 were used. Sampling was limited to 20 or 40 seconds. Solving is for all solutions.

time	vars	actual	estimated	error
20	31	2.037×10^8	2.017×10^8	0.0098
20	32	2.560×10^8	2.560×10^8	0.0000
20	33	8.135×10^8	9.661×10^8	0.1876
20	35	1.728×10^9	1.751×10^9	0.0133
40	31	2.037×10^8	2.013×10^8	0.0118
40	32	2.560×10^8	2.617×10^8	0.0223
40	33	8.135×10^8	8.382×10^8	0.0304
40	35	1.728×10^9	1.621×10^9	0.0619

On the random problems our sampling algorithm very accurately predicts the size of the search tree. For the largest tree considered in our experiments (see the last line of table 4.2), with just 40 seconds of sampling, an estimate within 0.5% of the actual is obtained. However as mentioned above this is to be expected because the trees tend to be quite uniform. For this reason these results are of limited value.

The Golomb problem

The Golomb problem is described in detail in section 2.2.2. Since the goal is to find the shortest ruler that can accommodate a given number of marks M, an iterative approach is used in solving instances. This is done by beginning with some initial length L for the ruler and attempting to solve the problem. If no solution can be found the process is repeated with $L + 1$. For each "failed" attempt the algorithm searches the complete tree. In fact a significant portion of the work to solve an instance of the Golomb ruler is in proving optimality (i.e. in these complete searches). Each of the trees searched (corresponding to various ruler lengths) are considered separately.

Three small trees (11 marks with length 67, length 68 and length 69) were used in experiments to calibrate the sampling algorithm; that is, to select values for the three parameters. As described above we ran our estimator using various values for *sample_depth* (1, 2, and 3), *bf_ratio* (0.25, 0.5, 0.75, and 1.0), and *bf_max* (1, 2, 3, and 4), and chose the values that produced the most accurate estimates. The values chosen for these experiments were:

- *sample_depth* = 2,
- *bf_ratio* = 0.5, and
- *bf_max* = 2.

As an aside, a *sample_depth* of 3, *bf_ratio* of 1, and *bf_max* of 4 was found to be the worst (produced the least accurate estimates of the small trees). Table 4.3 shows some of our results.

Table 4.3: Tree size estimates versus actual size for Golomb problems. Sampling was limited to 20 or 40 seconds.

time	M	L	actual	estimated	error
20	11	71	4.842×10^7	4.346×10^7	0.1024
20	11	72	5.446×10^7	5.670×10^7	0.0411
20	12	84	3.432×10^8	3.696×10^8	0.0769
20	12	85	4.019×10^8	5.002×10^8	0.2446
40	11	71	4.842×10^7	4.793×10^7	0.0101
40	11	72	5.446×10^7	5.059×10^7	0.0712
40	12	84	3.432×10^8	3.380×10^8	0.0152
40	12	85	4.019×10^8	4.433×10^8	0.1030

Our estimator is quite accurate on the trees searched while solving instances of the Golomb problem. For the largest tree considered in these experiments (see table 4.3) a size estimate within 10% of the actual tree size was obtained. In further experiments we found that if 80 seconds of sampling time is allowed, an estimate within 1% of the tree size was obtained. To put these results into perspective, these results mean that 80 seconds of sampling can produce an accurate estimate of the size of a tree that takes 6.9×10^4 seconds to search.

Logistics problem

The logistics problem is described in detail in section 2.2.3. As in the Golomb problem, solving an instance of the logistics problem involves several searches. For a given instance a lower bound, S, on the number of steps is required to achieve the final state. Then the problem is formulated and search is begun. If no solution can be found the process is repeated using $S + 1$ steps. Much of the total search effort is spent searching trees that have no solution. This search is by its nature a complete search of the tree. Table 4.4 demonstrates the effectiveness of our estimator on several trees. The largest of these trees (which corresponds to problem 27 with 13 steps) has 1.659×10^6 nodes and requires 12468 seconds to completely search.

Table 4.4: Actual tree sizes versus estimated tree sizes, for several instances of the logistics problem, where t is the number of seconds spent sampling, P is the problem number, and S is the number of steps.

time	P	S	actual	estimated	error
40	27	12	5.804×10^3	3.583×10^3	0.1464
40	20	13	8.129×10^5	4.126×10^5	0.3890
40	27	13	1.659×10^6	1.175×10^6	0.4740
80	27	12	5.804×10^3	3.983×10^3	0.0984
80	20	13	8.129×10^5	4.403×10^5	0.3085
80	27	13	1.659×10^6	8.343×10^5	0.5557

Our estimator (as described in section 4.5.5) proved to be not very accurate on the trees encountered when solving instances of the logistics problem. As an example, consider the last row of table 4.4. Given 80 seconds of sampling, the estimated tree size is only about half of the actual tree size. We also found that increasing the amount of time sampling did not reliably increase the accuracy. On the other hand, the estimates are relatively correct, for example it accurately predicted that the tree corresponding to problem 20, with S set to 13 is smaller than the tree corresponding to problem 27 with S set to 13.

On closer inspection, we were able to determine why our estimator was inaccurate on these trees. The reason is related to the shape of the trees. Every node has 0, 1, or 2 children, but the trees are very large because they are hundreds or thousands of levels deep. If c is the number of children of a node, our sampling algorithm chooses the number of successors to sample by choosing the minimum of these two values:

- $bf\_ratio \times c$, and
- bf_max.

If bf_ratio is set to 1.0 and bf_max is set to 2, then the algorithm will attempt to sample all children of each node. This, of course, is a complete search and can not produce a good estimate in the few seconds that are allowed for sampling. On the other hand if bf_max is set to 1, the algorithm will behave like Knuth's, in that exactly one child will be sampled (regardless of the value of bf_ratio). Purdom observed that for tall, thin trees Knuth's method does not tend to produce good estimates. This, in fact, was Purdom's motivation, for introducing his partial backtracking method.

In an attempt to remedy this difficulty we changed the way the number of children to explore is chosen. A new parameter was introduced, p, and the value of bf_max was allowed to vary. At each node encountered during sampling, bf_max is recalculated in such a way that $bf\_max = \infty$ with probability p and $bf\_max = 1.0$ with probability $1 - p$. This change allowed for more sampling to be done deeper in the tree and this seemed to improved the accuracy of the estimator.

To demonstrate this improvement we present some results from our experiments. We calibrated our estimator using a small tree (5804 nodes) from the logistics problem domain. This was done by trying various values for *sample_depth* (1,69,70,71,72, and 73), and p ($\frac{1}{3}$, $\frac{1}{4}$, $\frac{1}{5}$, and $\frac{1}{6}$), and choosing the values that produced the most accurate estimates. The reason that some of values for *sample_depth* were so high is because the initial (and goal) states have variables with domain sizes of 1. So at the top of the tree search all nodes have exactly one successor. The values for the parameters that produced the best estimates, on the small tree, were:

- $sample\_depth = 73$, and

- $p = 0.25$

Using these parameters we performed experiments on the tree corresponding to logistics problem 27 with 13 steps. The actual size of this tree is 1.659×10^6. Our results are presented in table 4.5. These results demonstrate that this revised estimator can produce (some what) better estimates of the search tree. We found that the accuracy is sensitive to changes in the parameters chosen. It may be that for some applications the accuracy achieved is sufficient.

Table 4.5: Performance of our revised estimator on logistics problem 27 with 13 steps, where *sample_depth* is set to 73, p is set to 0.25 and *bf_ratio* is set to 1.0. In the table, *time* is the number of seconds spent sampling to produce the estimate. The actual size of this tree is 1.659×10^6.

time	estimate	error
20	1.0212×10^6	0.3844
40	1.2628×10^6	0.2388
60	1.6098×10^6	0.0297
80	9.3558×10^5	0.4361
100	2.5452×10^6	0.5342
120	1.5532×10^6	0.0638

4.6 Variable ordering algorithm

We are now going to explore one possible application of our estimator.

A dynamic variable ordering algorithm (dvo) is used during backtracking at each level, to determine which variable to instantiate next. An effective dvo can significantly improve the efficiency of a backtracking algorithm. As discussed in section 2.3.1, *ff+deg* and *ff/deg* are popular domain independent dvo algorithms. In this section we present a new domain independent dvo algorithm based on the estimator described above.

The idea behind this dvo algorithm is simple. First, for each possible choice as the next variable to instantiate, estimate the size of the subtree that would result if this variable were instantiated next. Then, return the variable associated with the smallest estimate. If this estimate were exact then the procedure described above would pick exactly the *right* variable to instantiate next. In this context the *right* variable means the choice that would produce the smallest search tree, given the assumption that we are finding all solutions to a problem or showing that no solutions exist.

In general, it is impractical to perform this estimate for all variables. So instead we generate a list of *candidate* variables that seem promising. We make this selection based on the current domain sizes of the variables. Near the root of the search tree, a variable ordering can make the most difference on the overall size of the search tree, and at the root there is less information for more standard variable orderings (such as *ff+deg*) to make decisions on, since less propagation has occurred. Also at deeper levels, time spent sampling is less likely to be rewarded by a sufficiently reduced search tree size. So we use sampling above some maximum depth, and a more standard variable ordering method below (see line 1 and 2 of algorithm 12).

In addition to the parameters used by the sampling algorithm, the dvo algorithm uses a few others.

- *max_depth*

  The maximum depth at which sampling will be used to select the next variable to instantiate.

- *standard_dvo*

  The variable ordering to use at depths greater than *max_depth*.

- *close*

  This is used in generating a list of candidate variables. If d is the minimum domain size, then all variables with domain size less than or equal to $d \times close$ will be considered a candidate.

The complete algorithm is listed in algorithm 12. The first step is to come up with a set of possible candidates for the choice of next variable. Then the function Sample() is used to estimate the size of the subtree that would be search if v is selected. While the function Sample() is estimating the size of the subtree it uses a static variable ordering. The variable choice that corresponds to the smallest (estimated) tree size is returned as the variable to

Algorithm 12 Dvos(level)

```
1: if $level \geq max\_depth$ then
2:     return a result from some other dvo algorithm
3: end if
4: $min\_domain \leftarrow$ FindMinimumDomainSize()
5: $min\_size \leftarrow \infty$
6: $min\_var \leftarrow 0$
7: for all $v$ such that $D(v) \leq (min\_domain \times close)$ do
8:     $estimate \leftarrow Sample()$
9:     if $estimate < min\_size$ then
10:        $min\_size \leftarrow estimate$
11:        $min\_var \leftarrow v$
12:    end if
13: end for
14: return $min\_var$
```

be instantiated next. In some sense our algorithm can be thought of as using the Sample() algorithm as a tie breaking strategy for fail first. We call our algorithm *dvos*.

4.6.1 Experiments

We applied our variable ordering algorithm to instances of the Golomb and crossword puzzle problems.

Golomb

The Golomb problem is described in section 2.2.2. Table 4.6 presents some of our experimental results. Our ordering algorithm, dvos, is compared with *ff/con* on trees from the Golomb problem. On these problems *ff/con*, proved more effective than *ff+deg* or *ff/deg*. In all cases, we are solving for one solution, however for many of the trees no solution exists. In particular, solutions exist for $M = 11$ when $L = 72$, and for $M = 12$ when $L = 85$. However the number of solutions in these instances is very small. In these experiments dvos used the following (arbitrarily chosen) sampling parameters:

- $samples = 15$,
- $bf\_max = 1$, and

- *max_depth* = 4.

Note that when *bf_max* is set to 1, sampling will always visit exactly one successor of each node independent of the value of *bf_ratio*.

Table 4.6: A comparison of two variable orderings, *ff/con* and *dvos*, combined with forward checking on instances of the Golomb problem. The numbers in the table show the time (in seconds) required to solve the problem. A '-' means that the problem could not be solved within the 144000 second time limit.

M	L	ff/con	dvos
11	72	130	216
12	60	1601	311
12	70	9405	3830
12	80	48432	12587
12	81	57213	17961
12	82	66853	34568
12	83	77925	21882
12	84	90843	22245
12	85	12899	1967
13	90	-	78394

Our experiments suggest that *dvos* is an effective dynamic variable ordering on instances of the Golomb problem. For simple problems, *ff/con* performs better than *dvos*. However on more difficult problems, *dvos* performs better than *ff/con*. As an example, when *ff/con* is used on the tree corresponding to $M = 12$ and $L = 84$, 90843 seconds are required to solve the problem. When *dvos* is used on the same tree, 22245 seconds are required. This represents a 75% performance gain.

Crossword puzzle

The crossword puzzle problem is described in section 2.2.1. Table 4.7 presents some of our experimental results. Our ordering algorithm, dvos, is compared with *ff+deg* on instances of the crossword puzzle problem.

In all cases, we are solving for one solution using an arc consistency algorithm (gac3 in particular). In these experiments dvos used the following sampling parameters:

- $samples = 5$,
- $bf\_max = 1$, and
- $max\_depth = 3$.

We initially tried running experiments with $samples = 15$ and $max\_depth = 4$ (as in the Golomb experiments) but quickly found that too much time was spent sampling.

Table 4.7: A comparison of two variable orderings, *ff+deg* and *dvos*, combined with arc consistency on (15 × 15) crossword puzzle problems. The numbers in the table show the time (in seconds) required to solve the problem. A '-' means that the problem could not be solved within the 7200 second time limit.

p	ff+deg	dvos
15.01	20.14	49.34
15.02	248.93	248.36
15.03	10.55	37.82
15.04	-	3618
15.05	7.84	12.44
15.06	2526.8	2560.8
15.07	24.09	64.86
15.08	9.14	37.85
15.09	11.48	47.09
15.10	-	827.7

On easy problems the *dvos* algorithm performs worse than the *ff+deg* algorithm. The reason for this is that the time that the *dvos* algorithm spends sampling dominates any savings from an improved variable ordering. On the most difficult problems *dvos* performed better than *ff+deg*. When *ff+deg* is used on puzzle 15.04, for example, more than 7200 seconds are required to solve the problem. When *dvos* is used on puzzle 15.04, the problem can be solved in 3618 seconds. When *ff+deg* is used on puzzle 15.10 more than 7200 seconds are required to solve the problem. When *dvos* is used on puzzle 15.10, the problem can be solved in 827 seconds.

In choosing the values for the various sampling parameters, somewhat arbitrary values were chosen. It is possible that *dvos* would perform better with other values.

These results from our experiments on both the Golomb problem and the crossword puzzle problem should be characterized as preliminary. However they indicate that this dynamic variable ordering is promising. It is particularly interesting to note that an estimate of the size of the tree associated with finding *all* solutions to the problem can provide a valuable variable ordering heuristic for a search for *one* solution.

Chapter 5

Conclusion

In this chapter we provide some suggestions for future work and a summary.

5.1 Future work

1. Further work could be done to characterize the types of problems on which gac7 is superior to gac3. A technique for selecting which algorithm to use could be useful.

2. Possibly, features of gac7, such as the *lastsupport* structure, could be added to gac3. The goal would be to produce an algorithm with some of the efficiency gains found in gac7 while retaining gac3's flexibility.

3. Our estimator has several parameters that control the way sampling is done. Currently, tuning these parameters is a manual process. An automated method for tuning parameters would be useful.

4. In the introduction to chapter 4 we motivated the topic of estimation by discussing some possible applications for an estimator. We applied our estimator to only one of these applications: a dynamic variable ordering. Future work could involve determining if our estimator can be used in others of these applications.

5. The estimator we developed works well on two traditional backtracking-based algorithms: forward checking and arc consistency. A sampler for

other algorithms such as conflict-directed backjumping, while challenging, could be equally useful.

6. In [20], Maron and Moore discuss *Hoeffding Races*, which is a technique for selecting between sets of options. This technique can be applicable when sampling is done to produce estimates of how good a particular option is. The general approach involves iteratively performing sampling for each option and discarding options as they are shown to be inferior. Russ Greiner, in a personal communication, suggested that it would be valuable to use this approach in selecting between variables in our dynamic variable ordering algorithm, *dvos*.

5.2 Summary

In chapter 3 we described two general arc consistency algorithms: gac3 and gac7. We described how gac7 could be interleaved with a backtracking algorithm. We also performed an experimental comparison of these two algorithms. Crossword puzzle problems, logistics problems, and some simple random problems, were used in this comparison. Gac7 performed better than gac3 on the crossword puzzle problems and the random problems. Gac3 performed better than gac7 on the logistics problem. Gac3 was also found to be easier to implement, more flexible (in the level of consistency that is enforced) and to consume less memory.

In chapter 4 we discussed techniques for estimating the cost of solving an instance of a constraint satisfaction problem. We surveyed work that has been done in this area including hard problems, analytical techniques, and statistical sampling techniques. We discussed previous work that has shown that the hard problems approach is not an effective technique for estimating the cost of solving a problem. Our evaluation of existing analytical techniques showed that they can not accurately estimate this cost of solving constraint satisfaction problems. We implemented a statistical sampling algorithm, which builds on the work done by Knuth and Purdom in this area. We evaluated our algorithm on random problems, Golomb problems, and logistics problems.

We found that our estimator can accurately predict the cost of solving for all solutions of these problems.

We discussed applications for estimators and evaluated our estimator on one such application: a dynamic variable ordering which we called *dvos*. The limited experiments we did to evaluate our ordering algorithm were encouraging. In particular we found that, on Golomb problems, *dvos* outperformed *ff/con*, and on (sufficiently difficult) crossword puzzle problems, *dvos* outperformed *ff+deg*.

Bibliography

[1] UK advanced cryptics dictionary v1.5. http://www.bryson.demon.co.uk/wordlist.html.

[2] E. A. Bender and H. S. Wilf. A theoretical analysis of backtracking in the graph coloring problem. *Journal of Algorithms*, 6:275–282, 1985.

[3] C. Bessière, E. C. Freuder, and J.-C. Régin. Using inference to reduce arc consistency computation. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 592–598, Montreal, Quebec, 1995.

[4] C. Bessiere and J.C. Regin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 398–404, Nagoya, Japan, 1997.

[5] J. E. Borrett. *Formulation Selection for Constraint Satisfaction Problems: A Heuristic Approach.* PhD thesis, University of Essex, United Kingdom, 1998.

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

[7] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 331–337, Sydney, Australia, 1991.

[8] P. C. Chen. Heuristic sampling: A method for predicting the performance of tree searching programs. *SIAM J. Comput.*, 21:295–315, 1992.

[9] S. A. Cook and D. G. Mitchell. Finding hard instances of the satisfiability problem: A survey. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, to appear, 1997.

[10] A. K. Dewdney. Computer recreations. *Scientific American*, pages 16–26, December 1985.

[11] J. Franco and M. Paull. Probabilistic analysis of the davis putnam procedure for solving the satisfiability problem. *American Mathematical Monthly*, 5:77–87, 1983.

[12] D. Frost and R. Dechter. In search of the best search: An empirical evaluation. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 294–300, Seattle, Wash., 1994.

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

[14] T. Hogg. Applications of statistical mechanics to combinatorial search problems. In *World Scientific Series on Computational Physics, Vol. 2.* 1995.

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

[16] C.W.H. Lam. The search for a finite projective plane of order 10. *American Mathematical Monthly*, 98:306, 1991.

[17] L. Lobjois and M. Lemaitre. Brand and bound algorithm selection by performance prediction. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 353–358, Madison, Wisconsin, 1998.

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

[19] A. K. Mackworth. On reading sketch maps. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 598–606, Cambridge, Mass., 1977.

[20] O. Maron and A. W. Moore. Hoeffding races: Accelerating model selection search for classification and function approximation. In *Advances in Neural Information Processing Systems 6.* Morgan Kaufmann, 1993.

[21] S. Minton. Automatically configuring constraint satisfaction programs: A case study. *CONSTRAINTS*, 1:7–44, 1996.

[22] B. A. Nadel. Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert*, 5:16–23, 1990.

[23] D. M. Nicol. Expected performance of m-solution backtracking. *SIAM J. Comput.*, 17:114–127, 1988.

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

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

[26] H. S. Stone and P. Sipala. The average complexity of depth-first search with backtracking and cutoff. *IBM J. Res. and Develop.*, 30:242–258, 1986.

[27] E. P. K. Tsang, J. E. Borrett, and A. C. M. Kwan. An attempt to map the performance of a range of algorithm and heuristic combinations. In *Proceedings of the AI and Simulated Behaviour Conference*, pages 203–216, 1995.

[28] P. van Beek and X. Chen. Cplan: A constraint programming approach to planning. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, Orlando Florida, 1999.

[29] G. van Liempd and H. J. van den Herik. Predicting the behavior of search algorithms on csp problems. Department of Computer Science Technical Report 94-01, University of Limburg, Limburg, The Netherlands, 1994.

[30] G. van Liempd and H. J. van den Herik. Predicting the behavior of search algorithms on csp problems (cont'd). Department of Computer Science Technical Report 95-04, University of Limburg, Limburg, The Netherlands, 1995.