A Multipurpose Backtracking Algorithm

H.A. Priestley
HAP@@maths.ox.ac.uk
Mathematical Institute
24/29, St. Giles
Oxford OX1 3LB

M.P. Ward
Martin.Ward@@durham.ac.uk
Computer Science Department
Science Laboratories
South Rd
Durham DH1 3LE

January 17, 2003

Abstract

A backtracking algorithm with element order selection is presented, and its efficiency discussed in relation both to standard examples and to examples concerning relation-preserving maps which the algorithm was derived to solve.

1 Introduction

Backtracking has long been used as a strategy for solving combinatorial problems and has been extensively studied (Gerhart & Yelowitz (1976), Roever (1978), Walker (1960), Wells (1971)). In worst case situations it may be highly inefficient, and a systematic analysis of efficiency is very difficult. Thus backtracking has sometimes been regarded as a method of last resort. Nevertheless, backtracking algorithms are widely used, especially on NP-complete problems. In order to make these algorithms computationally feasible on a range of large problems, they are usually tailored to particular applications (see, for example, Butler and Lam's approach to isomorphism-testing in Butler & Lam (1985) and Knuth and Szwarcfiter's approach to topological sorting (that is, extending partial orders to linear orders) Knuth & Szwarcfiter (1974)).

Our approach to backtracking is based on Ward's work on program transformations Ward (1989), Ward (1992), Ward (1994), Ward (1993). We derive (and simultaneously prove correct) a 'universal' simple backtracking algorithm. Even in this rudimentary form our algorithm proved remarkably effective for the type of problem for which it was devised. These problems can all be cast as problems requiring the counting, listing, or otherwise processing, of the relation-preserving maps from a finite relational structure to another relational structure of the same type. In particular isomorphism-testing would come under this umbrella. Priestley was concerned specifically with problems arising in connection with Stone type dualities for varieties of algebras whose members were distributive lattices with additional structure. It turned out that, in these applications, the running time of the algorithm depended critically on the order in which the data elements were listed (by a factor of several thousand). The same techniques that yielded the simple backtracking algorithm were then employed to derive a version of the algorithm which incorporates a mechanism for permuting elements. By exploiting this in various ways enormous improvements in efficiency were obtained which enabled us to complete various calculations which would otherwise have been totally impractical. Tables 3 and 4 in Section 6.2 strikingly illustrate the effect of judicious element order selection in one particular case.

Our paper is aimed at two groups of readers with (probably) small intersection. The first group consists of those interested in backtracking *per se*. The second group contains mathematicians who need to solve problems in, for example, algebra or graph theory, to which our methods can be applied. For the benefit of such readers we have included some discussion of aspects of programming

folklore which would not have been required in a paper directed solely at computer scientists.

The paper is organised as follows. Section 2, which uses the well-known eight queens puzzle as an illustration, serves two purposes. It provides a brief introduction to backtracking for those unfamiliar with it, and also allows us to draw attention to the factors affecting efficiency which we address later. Section 3 presents the fragment of Ward's Wide Spectrum Language (WSL) which we use, and Section 4 contains the theory from Ward's work on program transformations on which our algorithm derivations are based. The simple backtracking algorithm is given in Section 5. The next section discusses the applications of this algorithm out of which the paper has arisen. It provides the mathematical background to a range of examples on which we have tested our methods. While Section 6 is reasonably self-contained it is aimed primarily at mathematicians with appropriate interests. Section 7 discusses various heuristics for element order selection, with illustrations. We conclude with some brief comments relevant to further developments: we discuss the state of the art concerning complete automation of the process of algorithm development, from abstract specification to implementation in a suitable programming language.

We stress that an understanding of the machinery in Sections 4–5 is not needed by users of the end product. The theory guarantees that the algorithm meets its specification. Because of its universal character, the algorithm (with or without element order selection) can very easily be adapted to a variety of situations without further recourse to the theory. Implementation is straightforward. We discuss implementation issues in a special case in Section 6. We also include an Appendix which gives a C implementation of the simple backtracking algorithm. The source code for all the algorithms and sample data files can be obtained from the authors.

2 The Eight Queens Puzzle

It is convenient to introduce the concept of backtracking by means of a simple puzzle, the *eight queens* problem:

> How many ways are there to place eight queens on a chessboard in such a way that no queen is attacking any other?

Two queens are attacking each other if they lie on the same row, column or diagonal. Figure 1 illustrates one of the solutions.

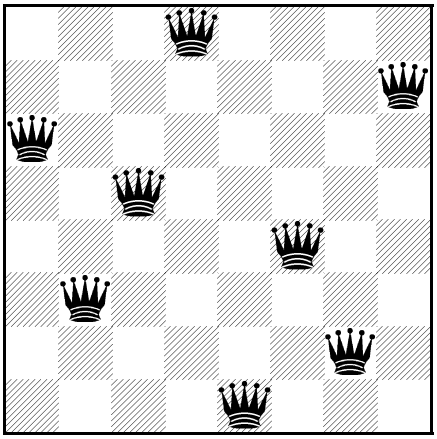

Figure 1: One solution to the eight queens puzzle.

The *brute force* solution for any combinatorial problem is to enumerate all the possible solutions, testing each in turn and rejecting those which fail to meet the required conditions. In this case, the "most brutish" method tests every possible arrangement of eight queens on a chessboard. There are 64 places for the first queen, for each of these there are 63 places for the second queen, and so on, for

a total of $64 \times 63 \times \cdots \times 57 = 178{,}462{,}987{,}637{,}760$ cases. This number can be reduced substantially by the observation that any valid solution must contain exactly one queen in each column. So we only need to consider the $8^8 = 16{,}777{,}216$ ways of placing eight queens, one per column, into eight columns. Any such arrangement can be represented as a sequence of eight numbers from 1 to 8, for example the situation in Figure 1 is represented as $\langle 3, 6, 4, 1, 8, 5, 7, 2\rangle$.

2.1 Backtracking

A simple way to reduce the number of cases still further now suggests itself. Consider the situation where the first two queens have been placed in positions 1 and 1, or 1 and 2 in the first two columns. Since these two are attacking each other, we need not consider any of the $2^6 = 262{,}144$ ways of placing the remaining six queens. Similarly, after placing the first four queens in Figure 1, there are only two valid positions for the fifth queen. Such a sequential placement can be represented as a tree structure, as shown in Figure 2 for the four queens puzzle. The four nodes below the root

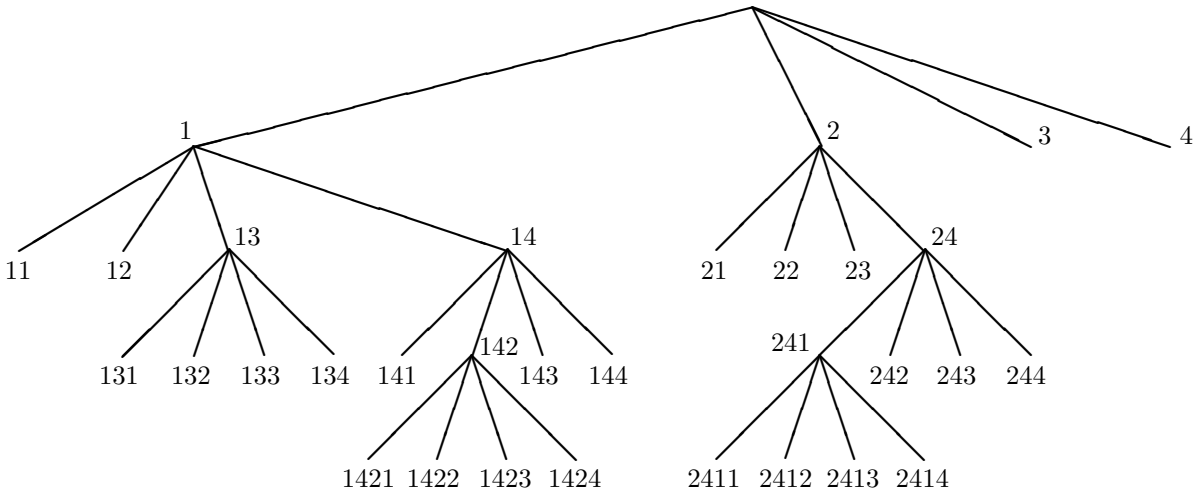

Figure 2: The Four Queens Search Tree

(top) node of the tree represent the four positions for the first queen. Below each valid node are further nodes representing the positions for the next queen to be placed. Note that branches 3 and 4 of the tree are mirror images of branches 2 and 1 respectively, and are omitted for brevity. The solutions are $\langle 2, 4, 1, 3\rangle$ and its mirror image $\langle 3, 1, 4, 2\rangle$.

This procedure cuts the number of cases examined (for the eight queens puzzle) to a total of 15,720. To enumerate systematically all these cases we start at the root and move down the tree, taking a leftmost branch at each junction, but if it is impossible to move down we "backtrack" by considering the next junction at the previous level. This may lead to further backtracking if all the junctions at the previous level have now been covered. The first computerised formulation of this method was by Walker in 1958 (Walker (1960)).

Assuming we have a predicate *valid*(p) which tests if the sequence of integers p is a valid arrangement of queens with no queen attacking any other, then the following recursive procedure will solve the problem. (The notation $p +\!\!+ \langle t\rangle$ denotes the sequence p with the singleton sequence $\langle t\rangle$ appended. See Section 3 for a description of the other notation).

```
begin
  count := 0;
  Queens(⟨⟩)
where
proc Queens(p) ≡
  if ℓ(p) = 8 then count := count + 1
```

```
        else for t := 1 to 8 do
                if valid(p ++ ⟨t⟩) then Queens(p ++ ⟨t⟩) fi od.
end
```

This is a special case of the algorithm we will derive in Section 5. Also in Section 5 we transform this recursive algorithm into an equivalent iterative algorithm:

```
count := 0;
var p := ⟨⟩, t := 1:
  while p ≠ ⟨⟩ ∨ t ⩽ 8 do
        if t > 8 then t ←(last) p; t := t + 1
        elsif valid(p ++ ⟨t⟩) ∧ ℓ(p) = 7 then count := count + 1; t := t + 1
        elsif valid(p ++ ⟨t⟩) ∧ ℓ(p) < 7 then p := p ++ ⟨t⟩; t := 1
                                          else t := t + 1 fi od end
```

The recursive program emphasises downward movement in the tree. Backtracking (upward movement) occurs as a matter of course when each position in the column has been considered. The iterative program emphasises backtracking by explicitly searching up and down the tree, working from left to right. The four cases in the loop deal with:

1. Moving up, i.e. backtracking;
2. Moving right when a solution has been found;
3. Moving down to the leftmost branch of the current node, and;
4. Moving right when the current arrangement is invalid.

2.2 Element Order Selection

So far we have assumed that the queens will be placed in their columns from left to right, but this is by no means essential. The placement of any group of queens can be in any order without affecting the final result; however, a different order may result in fewer cases needing to be analysed. Consider the situation in Figure 3 where the first three queens have been placed in the first three columns. Here, we have three places for the next queen in columns 4 and 5 (marked with ♕s), but

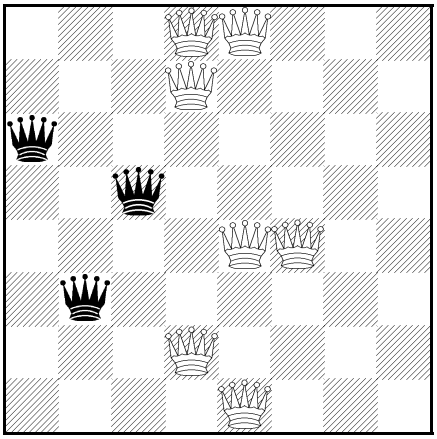

Figure 3: A partial solution

only one place in column 6. Placing the next queen in column 6 (rather than column 4) will reduce the total number of cases to be considered, without affecting the result. This is the basis for the various "element order selection" heuristics discussed below.

It should be pointed out that for this particular problem (and the more general N queens problem), the heuristics do not provide all that much benefit. This is because:

1. The size of the total search tree only increases by a factor of around 2 or 3 when a random element order is chosen rather than the optimal order (so a large reduction in the number of trials is not possible);
2. Using the method in Wirth (1971), a trial solution can be tested very efficiently (so there is not much to be gained from a small reduction in the number of trials);
3. The "naïve" solution of placing the queens in left to right order turns out to be the optimal order (if the element order is fixed throughout the calculation).

For the problems we were interested in solving, a suitable element order is critical in producing a result within a feasible amount of time. Even with this problem though, by starting with a random permutation each time we were able to produce some improvements by using the heuristics discussed in Section 7. See Figure 1 for some sample results, each of which is averaged over ten different random initial permutations. The "pre-analysis" method analyses the search tree to select an initial permutation. The "bush pruning" method dynamically updates the permutation as the search proceeds. The "hybrid" method is a combination of a small amount of pre-analysis, followed by bush pruning. It is the most efficient in terms of the number of trials, but imposes a higher overhead than simple pre-analysis—which is the most efficient in terms of CPU time.

	No. of trials		CPU time	
Method used	13 queens	14 queens	13 queens	14 queens
none	130,150,618	899,139,237	138.43	942.14
pre-analysis	100,515,902	654,151,660	110.61	696.29
hybrid	89,088,384	569,929,575	140.00	878.52

Table 1: Sample results from the N queens problem

3 The Language WSL

In this section we give a brief introduction to the language WSL (Bull (1990), Ward (1989), Ward (1994)) the "Wide Spectrum Language", used in Ward's program transformation work, which includes low-level programming constructs and high-level abstract specifications within a single language. By working within a single formal language we are able to prove that a program correctly implements a specification, or that a specification correctly captures the behaviour of a program, by means of formal transformations in the language. We don't have to develop transformations between the "programming" and "specification" languages. An added advantage is that different parts of the program can be expressed at different levels of abstraction, if required.

A *program transformation* is an operation which modifies a program into a different form which has the same external behaviour (it is equivalent under a precisely defined denotational semantics). Since both programs and specifications are part of the same language, transformations can be used to demonstrate that a given program is a correct implementation of a given specification. In Ward (1990), Ward (1996) program transformations are used to derive a variety of efficient algorithms from abstract specifications. In this paper we use transformations to derive various efficient backtracking algorithms from a formal specification.

3.1 Syntax of Expressions

Expressions include variable names, numbers, strings of the form "`text...`", the constants $\mathbb{N}$, $\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, and the following operators and functions. Note that since WSL is a wide spectrum language it must not be restricted to finite values and computable operations. In the following e_1, e_2, etc., represent any valid expressions:

Numeric operators: $e_1 + e_2$, $e_1 - e_2$, $e_1 * e_2$, e_1/e_2, $e_1^{e_2}$ and so on, with the usual meanings.

Sequences: $s = \langle a_1, a_2, \ldots, a_n \rangle$ is a sequence, the ith element a_i is denoted $s[i]$, $s[i\,..\,j]$ is the subsequence $\langle s[i], s[i+1], \ldots, s[j] \rangle$, where $s[i\,..\,j] = \langle\rangle$ (the empty sequence) if $i > j$. The length of sequence s is denoted $\ell(s)$, so $s[\ell(s)]$ is the last element of s.

Sequence concatenation: $s_1 +\!\!+ s_2 = \langle s_1[1], \ldots, s_1[\ell(s_1)], s_2[1], \ldots, s_2[\ell(s_2)] \rangle$.

Stacks: Sequences are also used to implement stacks, for this purpose we have the following notation: For a sequence s and variable x: $x \xleftarrow{\text{pop}} s$ means $x := s[1];\ s := s[2\,..\,\ell(s)]$ which pops an element off the stack into variable x. To push the value of the expression e onto stack s we use: $s \xleftarrow{\text{push}} e$ which represents: $s := \langle e \rangle +\!\!+ s$.

Queues: The statement $x \xleftarrow{\text{last}} s$ removes the last element of s and stores its value in the variable x. It is equivalent to $x := s[\ell(s)];\ s := s[1\,..\,\ell(s) - 1]$.

Sets: We have the usual set operations $\cup$ (union), $\cap$ (intersection) and $\setminus$ (set difference), $\subseteq$ (subset), $\in$ (element), $\wp$ (powerset). $\{\, x \in A \mid P(x) \,\}$ is the set of all elements in A which satisfy predicate P. For the sequence s, $set(s)$ is the set of elements of the sequence, i.e. $set(s) = \{\, s[i] \mid 1 \leqslant i \leqslant \ell(s) \,\}$. The expression $\#A$ denotes the size of the set A.

3.2 Syntax of Formulae

In the following $\mathbf{Q}$, $\mathbf{Q}_1$, $\mathbf{Q}_2$ etc., represent arbitrary formulae and e_1, e_2, etc., arbitrary expressions:

Relations: $e_1 = e_2$, $e_1 \neq e_2$, $e_1 < e_2$, $e_1 \leqslant e_2$, $e_1 > e_2$, $e_1 \geqslant e_2$;

Logical operators: $\neg\mathbf{Q}$, $\mathbf{Q}_1 \vee \mathbf{Q}_2$, $\mathbf{Q}_1 \wedge \mathbf{Q}_2$;

Quantifiers: $\forall \mathbf{v}.\, \mathbf{Q}$, $\exists \mathbf{v}.\, \mathbf{Q}$.

3.3 Syntax of Statements

In the following, $\mathbf{S}_1$, $\mathbf{S}_2$ etc., are statements, $\mathbf{Q}$, $\mathbf{B}$ etc., are formulae, x_1, x_2 etc., are variables and e_1, e_2 etc. are expressions.

Sequential composition: $\mathbf{S}_1;\ \mathbf{S}_2;\ \mathbf{S}_3;\ \ldots;\ \mathbf{S}_n$

Assertion: $\{\mathbf{B}\}$. An assertion is a partial **skip** statement, it aborts if the condition is false but does nothing if the condition is true.

Assignment: $\langle x_1, \ldots, x_n \rangle := \langle x'_1, \ldots, x'_n \rangle.\mathbf{Q}$. This assigns new values to the variables $x_1, \ldots, x_n$. In the formula $\mathbf{Q}$, x_i represent the old valies and x'_i represent the new values. The new values are chosen so that $\mathbf{Q}$ will be true, then they are assigned to the variables. If there are several sets of values which satisfy $\mathbf{Q}$ then one set is chosen nondeterministically. If there are no values which satisfy $\mathbf{Q}$ then the statement does not terminate. For example, the assignment $\langle x \rangle := \langle x' \rangle.(x = 2.x')$ halves x if it is even and aborts if x is odd. If the sequence contains one variable then the sequence brackets may be omitted, for example: $x := x'.(x = 2.x')$. The assignment $x := x'.(y = 0)$ assigns an arbitrary value to x if $y = 0$ initially, and aborts if $y \neq 0$ initially: it does not change the value of y.

Simple assignment: $\langle x_1, \ldots, x_n \rangle := \langle e_1, \ldots, e_n \rangle$. This assigns the values of the expressions e_i to the variables x_i. The assignments are carried out simultaneously, so for example $\langle x, y \rangle := \langle y, x \rangle$ swaps the values of x and y. The single assignment $\langle x \rangle := \langle e \rangle$ can be abbreviated to $x := e$.

Deterministic choice: **if** $\mathbf{B}$ **then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi**. The choice of which statement to execute is determined by the condition $\mathbf{B}$.

Nondeterministic choice: The "guarded command" of Dijkstra (1976):

if $\mathbf{B}_1 \to \mathbf{S}_1$
$\Box\ \mathbf{B}_2 \to \mathbf{S}_2$
...
$\Box\ \mathbf{B}_n \to \mathbf{S}_n$ **fi**

Each of the "guards" $\mathbf{B}_1$, $\mathbf{B}_2$, ..., $\mathbf{B}_n$ is evaluated, one of the true ones is selected and the corresponding statement executed. If no guard is true then the statement aborts. If several guards are true, then one of the corresponding statements is chosen nondeterministically.

Deterministic iteration: **while** **B** **do** **S** **od** The condition **B** is tested and **S** is executed repeatedly until **B** becomes false.

Uninitialised local variables: **var** **x**: **S** **end** Here **x** is a local variable which only exists within the statement **S**. It must be initialised in **S** before it is first accessed.

Initialised local variables: **var** **x** := **t**: **S** **end** This is an abbreviation for **var** **x**: **x** := **t**; **S** **end**. The local variable is initialised to the value **t**. We can combine initialised and uninitialised variables in one block, for example: **var** **x** := **t**, **y**: **S** **end** where **x** is initialised and **y** is uninitialised.

Counted iteration: **for** $i := b$ **to** f **step** s **do** **S** **od** is equivalent to:

var $i := b$:
  while $i \leqslant f$ **do**
    S; $i := i + s$ **od** **end**

Unbounded loops and exits: Statements of the form **do** **S** **od**, where **S** is a statement, are "infinite" or "unbounded" loops which can only be terminated by the execution of a statement of the form **exit**(n) (where n is an integer, *not* a variable or expression) which causes the program to exit the n enclosing loops. To simplify the language we disallow **exit**s which leave a block or a loop other than an unbounded loop. This type of structure is described in Knuth (1974) and more recently in Taylor (1984).

3.4 Action Systems

This subsection will introduce the concept of an *action system* as a set of parameterless mutually recursive procedures. A program written using labels and jumps translates directly into an action system. Note however that if the end of the body of an action is reached, then control is returned to the calling action, or to the statement following the action system if there was no calling action, rather than "falling through" to the next label. The exception to this is a special action called the terminating action, usually denoted Z, which when called results in the immediate termination of the whole action system.

An *action* is a parameterless procedure acting on global variables (cf Arsac (1982a), Arsac (1982b)). It is written in the form $A \equiv$ **S.** where A is a statement variable (the name of the action) and **S** is a statement (the action body). A set of (mutually recursive) actions is called an *action system*. There may sometimes be a special action Z, execution of which causes termination of the whole action system even if there are unfinished recursive calls. An occurrence of a statement **call** X within the action body refers to a call of another action.

An action system is written as follows, with the first action to be executed (A_1 below) named at the beginning:

actions A_1:
$A_1 \equiv \mathbf{S}_1$**.**
$A_2 \equiv \mathbf{S}_2$**.**
...
$A_n \equiv \mathbf{S}_n$**. endactions**

For example, this action system is equivalent to the while loop **while** **B** **do** **S** **od**:

actions A:
$A \equiv$ **if** ¬**B** **then** **call** Z **fi**;
  S; **call** A**. endactions**

With this action system, each action call must lead to another action call, so the system can only

terminate by calling the Z action (which causes immediate termination). Such action systems are called *regular*.

3.5 Procedures and Functions with Parameters

We use the following notation for procedures with parameters:

begin $\mathbf{S}_1$
where
proc $F(x, y) \equiv \mathbf{S}_2$**.**
end

where $\mathbf{S}_1$ is a program containing calls to the procedure F which has parameters x and y. The body $\mathbf{S}_2$ of the procedure may contain recursive procedure calls. We use a similar notation (with **funct** instead of **proc**) for function calls.

4 Program Refinement and Transformation

The WSL language includes both specification constructs, such as the general assignment, and programming constructs. One aim of our program transformation work is to develop programs by refining a specification, expressed in first order logic and set theory, into an efficient algorithm. This is similar to the "refinement calculus" approach of Hoare et al. (1987), Morgan (1994), however, our wide spectrum language has been extended to include general action systems and loops with multiple exits. These extensions are essential for our second, and equally important aim, which is to use program transformations for reverse engineering from programs to specifications. In Ward (1993) we describe our method for formal reverse engineering using transformations.

Refinement is defined in terms of the denotational semantics of the language: the semantics of a program $\mathbf{S}$ is a function which maps from an initial state to a final set of states. The set of final states represents all the possible output states of the program for the given input state. Using a set of states enables us to model nondeterministic programs and partially defined (or incomplete) specifications. For programs $\mathbf{S}_1$ and $\mathbf{S}_2$ we say $\mathbf{S}_1$ is refined by $\mathbf{S}_2$ (or $\mathbf{S}_2$ is a refinement of $\mathbf{S}_1$), and write $\mathbf{S}_1 \leq \mathbf{S}_2$, if $\mathbf{S}_2$ is more defined and more deterministic than $\mathbf{S}_1$. If $\mathbf{S}_1 \leq \mathbf{S}_2$ and $\mathbf{S}_2 \leq \mathbf{S}_1$ then we say $\mathbf{S}_1$ is equivalent to $\mathbf{S}_2$ and write $\mathbf{S}_1 \approx \mathbf{S}_2$. Equivalence is thus defined in terms of the external "black box" behaviour of the program. A transformation is an operation which maps any program satisfying the applicability conditions of the transformation to an equivalent program. See Ward (1989) and Ward (1991a) for a description of the semantics of WSL and the methods used for proving the correctness of refinements and transformations.

The rest of this section describes the transformations we will use later in the derivation of the backtracking algorithm.

4.1 Expand IF statement

The **if** statement:

$$\textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi}; \mathbf{S}$$

can be expanded over the following statement to give:

$$\textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}_1; \mathbf{S} \textbf{ else } \mathbf{S}_2; \mathbf{S} \textbf{ fi}$$

4.2 Refine Assignment

If $\mathbf{Q}' \Rightarrow \mathbf{Q}$ and $\exists x.\, \mathbf{Q} \Rightarrow \exists x.\, \mathbf{Q}'$ then we can refine the assignment $x := x'.\mathbf{Q}$ to $x := x'.\mathbf{Q}'$. For example, if $\varnothing \neq I' \subseteq I$ then $x := x'.x' \in I \leq x := x'.x' \in I'$.

4.3 Split Block

If the statement $\mathbf{S}_2$ assigns a new value to x before it accesses it then the block: **var** x: $\mathbf{S}_1$; $\mathbf{S}_2$ **end** can be split into two blocks: **var** x: $\mathbf{S}_1$ **end**; **var** x: $\mathbf{S}_2$ **end**

4.4 Loop Inversion

If the statement $\mathbf{S}_1$ contains no **exit**s which can cause termination of an enclosing loop (i.e. in the notation of Ward (1989) it is a *proper sequence*) then the loop:

$$\textbf{do } \mathbf{S}_1;\ \mathbf{S}_2 \textbf{ od}$$

can be inverted to:

$$\mathbf{S}_1;\ \textbf{do } \mathbf{S}_2;\ \mathbf{S}_1 \textbf{ od}$$

This transformation may be used in the forwards direction to move the termination test of a loop to the beginning, prior to transforming it into a **while** loop, or it may be used in the reverse direction to merge two copies of the statement $\mathbf{S}_1$.

4.5 Loop Unrolling

The next three transformations concern various forms of loop unrolling. They play an important rôle in the proofs of other transformations as well as being generally useful.

Lemma 4.1 *Loop Unrolling:*

$$\textbf{while } \mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od} \quad \approx \quad \textbf{if } \mathbf{B} \textbf{ then } \mathbf{S};\ \textbf{while } \mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od fi}$$

Lemma 4.2 *Selective unrolling of* **while** *loops:* For any condition $\mathbf{Q}$ we have:

$$\textbf{while } \mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od} \quad \approx \quad \textbf{while } \mathbf{B} \textbf{ do } \mathbf{S};\ \textbf{if } \mathbf{B} \wedge \mathbf{Q} \textbf{ then } \mathbf{S} \textbf{ fi od}$$

Lemma 4.3 *Entire Loop Unrolling:* if $\mathbf{B}' \Rightarrow \mathbf{B}$ then for any condition $\mathbf{Q}$:

$$\textbf{while } \mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od} \quad \approx \quad \textbf{while } \mathbf{B} \textbf{ do } \mathbf{S};\ \textbf{if } \mathbf{Q} \textbf{ then while } \mathbf{B}' \textbf{ do } \mathbf{S} \textbf{ od fi od}$$

For each of these transformations there is a generalisation in which, instead of inserting the "unrolled" part after $\mathbf{S}$, it is copied into an arbitrary selection of the terminal positions in $\mathbf{S}$.

The converse transformations are, naturally, called loop rolling and entire loop rolling.

4.6 Introducing Recursion

This section introduces an important theorem on the recursive implementation of statements. It shows how a general statement can be transformed into an equivalent recursive statement. The transformations involved can be used to implement recursive specifications as recursive procedures, to introduce recursion into an abstract program to get a "more concrete" program (i.e. closer to a programming language implementation), and to transform a given recursive procedure into a different form. The theorem is used in the algorithm derivations of Ward (1996) and Ward (1989), we use it below in Section 5.

Suppose we have a statement $\mathbf{S}'$ which we wish to transform into the recursive procedure **proc** $F \equiv \mathbf{S}$. This is possible whenever:

1. The statement $\mathbf{S}'$ is refined by $\mathbf{S}[\mathbf{S}'/F]$ (which denotes $\mathbf{S}$ with all occurrences of F replaced by $\mathbf{S}'$). In other words, if we replace recursive calls in $\mathbf{S}$ by copies of $\mathbf{S}'$ then we get a refinement of $\mathbf{S}'$;
2. We can find an expression $\mathbf{t}$ (called the *variant function*) whose value is reduced before each occurrence of $\mathbf{S}'$ in $\mathbf{S}[\mathbf{S}'/F]$.

The expression $\mathbf{t}$ need not be an integer: any set Γ which has a well-founded order $\preccurlyeq$ is suitable. To prove that the value of $\mathbf{t}$ is reduced it is sufficient to prove that if $\mathbf{t} \preccurlyeq t_0$ initially, then the assertion $\{\mathbf{t} \prec t_0\}$ can be inserted before each occurrence of $\mathbf{S}'$ in $\mathbf{S}[\mathbf{S}'/F]$. The theorem combines these two requirements into a single condition:

Theorem 4.4 *If $\preccurlyeq$ is a well-founded partial order on some set Γ and $\mathbf{t}$ is an expression giving values in Γ and t_0 is a variable which does not occur in $\mathbf{S}$ then if for some premiss $\mathbf{P}$*

$$\forall t_0.\,((\mathbf{P} \wedge \mathbf{t} \preccurlyeq t_0) \Rightarrow \mathbf{S}' \;\leq\; \mathbf{S}[\{\mathbf{P} \wedge \mathbf{t} \prec t_0\};\; \mathbf{S}'/F])$$

then

$$\mathbf{P} \Rightarrow (\mathbf{S}' \;\leq\; \underline{\mathbf{proc}}\; F \;\equiv\; \mathbf{S}.)$$

It is frequently possible to *derive* a suitable procedure body $\mathbf{S}$ from the statement $\mathbf{S}'$ by applying transformations to $\mathbf{S}'$, splitting it into cases etc., until we get the statement $\mathbf{S}[\mathbf{S}'/F]$ which is still defined in terms of $\mathbf{S}'$. If we can find a suitable variant function for $\mathbf{S}[\mathbf{S}'/F]$ then we can apply the theorem and refine $\mathbf{S}[\mathbf{S}'/F]$ to $\underline{\mathbf{proc}}\; F \;\equiv\; \mathbf{S}.$ which is no longer defined in terms of $\mathbf{S}'$.

As an example we will consider the familiar factorial function. Let $\mathbf{S}' = r := n!$. We can transform this (by appealing to the definition of factorial) to get:

$$\mathbf{S}' \;\approx\; \underline{\mathbf{if}}\; n = 0\; \underline{\mathbf{then}}\; r := 1\; \underline{\mathbf{else}}\; r := n.(n-1)!\; \underline{\mathbf{fi}}$$

Separate the assignment:

$$\mathbf{S}' \;\approx\; \underline{\mathbf{if}}\; n = 0\; \underline{\mathbf{then}}\; r := 1\; \underline{\mathbf{else}}\; n := n-1;\; r := n!;\; n := n+1;\; r := n.r\; \underline{\mathbf{fi}}$$

So we have:

$$\mathbf{S}' \;\approx\; \underline{\mathbf{if}}\; n = 0\; \underline{\mathbf{then}}\; r := 1\; \underline{\mathbf{else}}\; n := n-1;\; \mathbf{S}';\; n := n+1;\; r := n.r\; \underline{\mathbf{fi}}$$

The positive integer n is decreased before the copy of $\mathbf{S}'$, so if we set $\mathbf{t}$ to be n, Γ to be $\mathbb{N}$ and $\preccurlyeq$ to be $\leqslant$ (the usual order on natural numbers), and $\mathbf{P}$ to be **true** then we can prove:

$$n \leqslant t_0 \Rightarrow \mathbf{S}' \;\leq\; \underline{\mathbf{if}}\; n = 0\; \underline{\mathbf{then}}\; r := 1\; \underline{\mathbf{else}}\; n := n-1;\; \{n < t_0\};\; \mathbf{S}';\; n := n+1;\; r := n.r\; \underline{\mathbf{fi}}$$

So we can apply Theorem 4.4 to get:

$$\mathbf{S}' \;\leq\; \underline{\mathbf{proc}}\; X \;\equiv\; \underline{\mathbf{if}}\; n = 0\; \underline{\mathbf{then}}\; r := 1\; \underline{\mathbf{else}}\; n := n-1;\; X;\; n := n+1;\; r := n.r\; \underline{\mathbf{fi}}.$$

and we have derived a recursive implementation of factorial.

4.7 Transforming Recursion to Iteration

The following general purpose recursion removal transformation was presented in Ward (1992). The proof may be found in Ward (1991b).

Suppose we have a recursive procedure whose body is a regular action system in the following form:

$\underline{\mathbf{proc}}\; F(x) \;\equiv$
  $\underline{\mathbf{actions}}\; A_1:$
  $A_1 \;\equiv \mathbf{S}_1.$
  $\ldots A_i \;\equiv \mathbf{S}_i.$
  $\ldots B_j \;\equiv \mathbf{S}_{j0};\; F(g_{j1}(x));\; \mathbf{S}_{j1};\; F(g_{j2}(x));\; \ldots;\; F(g_{jn_j}(x));\; \mathbf{S}_{jn_j}.$
  $\ldots$ $\underline{\mathbf{endactions}}.$

where $\mathbf{S}_{j1}, \ldots, \mathbf{S}_{jn_j}$ preserve the value of x and no $\mathbf{S}$ contains a call to F (i.e. all the calls to F are listed explicitly in the B_j actions) and the statements $\mathbf{S}_{j0}, \mathbf{S}_{j1} \ldots, \mathbf{S}_{jn_j-1}$ contain no action calls.

There are $M+N$ actions in total: $A_1, \ldots, A_M, B_1, \ldots, B_N$. Note that the since the action system is regular, it can only be terminated by executing **call** Z which will terminate the current invocation of the procedure.

The aim is to remove the recursion by introducing a local stack L which records "postponed" operations: When a recursive call is required we "postpone" it by pushing the pair $\langle 0, e\rangle$ onto L (where e is the parameter required for the recursive call). Execution of the statements $\mathbf{S}_{jk}$ also has to be postponed (since they occur between recursive calls), we record the postponement of $\mathbf{S}_{jk}$ by pushing $\langle\langle j, k\rangle, x\rangle$ onto L. Where the procedure body would normally terminate (by calling Z) we instead call a new action $\hat{F}$ which pops the top item off L and carries out the postponed operation. If we call $\hat{F}$ with the stack empty then all postponed operations have been completed and the procedure terminates by calling Z.

Theorem 4.5 *A recursive procedure in the form:*

proc $F(x)$ $\equiv$
  actions A_1:
  $A_1 \equiv \mathbf{S}_1$.
  $\ldots A_i \equiv \mathbf{S}_i$.
  $\ldots B_j \equiv \mathbf{S}_{j0};\ F(g_{j1}(x));\ \mathbf{S}_{j1};\ F(g_{j2}(x));\ \ldots;\ F(g_{jn_j}(x));\ \mathbf{S}_{jn_j}$.
  $\ldots$ **endactions**.

where $\mathbf{S}_{j1}, \ldots, \mathbf{S}_{jn_j}$ *are as above, is equivalent to the following iterative procedure which uses a new local stack* L *and a new local variable* m*:*

proc $F'(x)$ $\equiv$
  var $L := \langle\rangle, m$:
    actions A_1:
    $A_1 \equiv \mathbf{S}_1[$**call** $\hat{F}/$**call** $Z]$.
    $\ldots A_i \equiv \mathbf{S}_i[$**call** $\hat{F}/$**call** $Z]$.
    $\ldots B_j \equiv \mathbf{S}_{j0};\ L := \langle\langle 0, g_{j1}(x)\rangle, \langle\langle j, 1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle, \ldots, \langle 0, g_{jn_j}(x)\rangle, \langle\langle j, n_j\rangle, x\rangle\rangle +\!\!+ L;$
      call $\hat{F}$.
    $\ldots \hat{F} \equiv$ **if** $L = \langle\rangle$ **then call** Z
      else $\langle m, x\rangle \xleftarrow{\text{pop}} L;$
        if $m = 0 \rightarrow$ **call** A_1
        $\square \ldots \square\ m = \langle j, k\rangle \rightarrow \mathbf{S}_{jk};$ **call** $\hat{F}$
        $\ldots$ **fi fi. endactions end**.

Note that any procedure $F(x)$ can be restructured into the required form; in fact there may be several different ways of structuring $F(x)$ which meet the required criteria.

Consider the recursive factorial program we derived above (Section 4.6):

proc X $\equiv$ **if** $n = 0$ **then** $r := 1$ **else** $n := n - 1;\ X;\ n := n + 1;\ r := n.r$ **fi.**

We can restructure this as:

proc X $\equiv$
  actions A:
  $A \equiv$ **if** $n = 0$ **then** $r := 1;$ **call** Z **else call** B **fi.**
  $B \equiv n := n - 1;\ X;\ n := n + 1;\ r := n.r;$ **call** Z.
  endactions.

This is in the right form to apply Theorem 4.5. This gives:

proc X $\equiv$
  var $L := \langle\rangle, m$:
    actions A:
    $A \equiv$ **if** $n = 0$ **then** $r := 1;$ **call** $\hat{F}$ **else call** B **fi.**

$B \equiv n := n - 1;\ L := \langle 0, 1\rangle +\!\!+ L;\ \underline{\textbf{call}}\ \hat{F}$**.**
$\hat{F} \equiv \underline{\textbf{if}}\ L = \langle\rangle\ \underline{\textbf{then}}\ \underline{\textbf{call}}\ Z$
$\underline{\textbf{else}}\ \langle m\rangle \xleftarrow{\text{pop}} L;$
$\underline{\textbf{if}}\ m = 0 \to \underline{\textbf{call}}\ A$
$\Box\ m = 1 \to n := n + 1;\ r := n.r;\ \underline{\textbf{call}}\ \hat{F}\ \underline{\textbf{fi}}\ \underline{\textbf{fi}}$**.**
$\underline{\textbf{endactions}}\ \underline{\textbf{end}}$**.**

Notice that B pushes 0 onto L, then calls $\hat{F}$ which immediately pops off the 0 and calls A. So we could call A directly:

$\underline{\textbf{proc}}\ X \equiv$
$\underline{\textbf{var}}\ L := \langle\rangle, m:$
$\underline{\textbf{actions}}\ A:$
$A \equiv \underline{\textbf{if}}\ n = 0\ \underline{\textbf{then}}\ r := 1;\ \underline{\textbf{call}}\ \hat{F}\ \underline{\textbf{else}}\ \underline{\textbf{call}}\ B\ \underline{\textbf{fi}}$**.**
$B \equiv n := n - 1;\ L := \langle 1\rangle +\!\!+ L;\ \underline{\textbf{call}}\ A$**.**
$\hat{F} \equiv \underline{\textbf{if}}\ L = \langle\rangle\ \underline{\textbf{then}}\ \underline{\textbf{call}}\ Z$
$\underline{\textbf{else}}\ \langle m\rangle \xleftarrow{\text{pop}} L;$
$\underline{\textbf{if}}\ m = 0 \to \underline{\textbf{call}}\ A$
$\Box\ m = 1 \to n := n + 1;\ r := n.r;\ \underline{\textbf{call}}\ \hat{F}\ \underline{\textbf{fi}}\ \underline{\textbf{fi}}$**.**
$\underline{\textbf{endactions}}\ \underline{\textbf{end}}$**.**

Now that we only ever push ones onto L, all we need to know is its length[1]. So convert L to an integer variable and remove the redundant local variable m:

$\underline{\textbf{proc}}\ X \equiv$
$\underline{\textbf{var}}\ L := 0:$
$\underline{\textbf{actions}}\ A:$
$A \equiv \underline{\textbf{if}}\ n = 0\ \underline{\textbf{then}}\ r := 1;\ \underline{\textbf{call}}\ \hat{F}\ \underline{\textbf{else}}\ \underline{\textbf{call}}\ B\ \underline{\textbf{fi}}$**.**
$B \equiv n := n - 1;\ L := L + 1;\ \underline{\textbf{call}}\ A$**.**
$\hat{F} \equiv \underline{\textbf{if}}\ L = 0\ \underline{\textbf{then}}\ \underline{\textbf{call}}\ Z$
$\underline{\textbf{else}}\ L := L - 1;\ n := n + 1;\ r := n.r;\ \underline{\textbf{call}}\ \hat{F}\ \underline{\textbf{fi}}$**.**
$\underline{\textbf{endactions}}\ \underline{\textbf{end}}$**.**

Now the A and B actions just copy n into L, set r to 1, set n to 0, and call $\hat{F}$. The $\hat{F}$ action can be expressed as a **while** loop, so we have:

$\underline{\textbf{proc}}\ X \equiv$
$\underline{\textbf{var}}\ L := n:$
$r := 1;\ n := 0;$
$\underline{\textbf{while}}\ L \neq 0\ \underline{\textbf{do}}\ L := L - 1;\ n := n + 1;\ r := n.r\ \underline{\textbf{od}}\ \underline{\textbf{end}}$**.**

Note that L reaches zero when n reaches its original value, so we can write the **while** loop as a **for** loop:

$\underline{\textbf{proc}}\ X \equiv\ r := 1;\ \underline{\textbf{for}}\ i := 1\ \underline{\textbf{to}}\ n\ \underline{\textbf{do}}\ r := i.r\ \underline{\textbf{od}}$**.**

This is an efficient factorial algorithm, derived from the specification given in Section 4.6

4.8 Nondeterministic Iteration

We introduce the following notation for nondeterministic iteration over the elements of a finite set:

$$\underline{\textbf{for}}\ i \in I\ \underline{\textbf{do}}\ \mathbf{S}\ \underline{\textbf{od}} =_{\text{DF}} \underline{\textbf{var}}\ i, I' := I:\ \underline{\textbf{while}}\ I' \neq \varnothing\ \underline{\textbf{do}}\ i := i'.(i' \in I');\ I' := I' \setminus \{i\};\ \mathbf{S}\ \underline{\textbf{od}}\ \underline{\textbf{end}}$$

[1] Technically, we introduce a new variable, l say, which records the length of L. Then we replace references to L by corresponding references to l (this is possible since we only refer to the length of L). Then L becomes redundant and can be removed. Finally we rename l to L

This picks elements from the (finite) set I in an arbitrary order and executes **S** once for each element.

Lemma 4.6 *If I_1 and I_2 partition I (i.e. $I = I_1 \cup I_2$ and $I_1 \cap I_2 = \varnothing$) then the* **for** *loop refines to the pair of loops:*

$$\textbf{for } i \in I \textbf{ do } \mathbf{S} \textbf{ od} \quad \leq \quad \textbf{for } i \in I_1 \textbf{ do } \mathbf{S} \textbf{ od}; \textbf{ for } i \in I_2 \textbf{ do } \mathbf{S} \textbf{ od}$$

Proof: The proof is by induction on the size of the (finite) set I using transformations 4.2, 4.3 and 4.5.

By induction on this lemma we get the more general result:

Lemma 4.7 *Suppose the finite set I is partitioned as $\bigcup_{j \in J} I_j$ where the sets I_j are disjoint. Then the* **for** *above refines to the double nested loop:*

$$\begin{array}{lll} \textbf{for } i \in I \textbf{ do } \mathbf{S} \textbf{ od} & \leq & \textbf{for } j \in J \textbf{ do} \\ & & \quad \textbf{for } i \in I_j \textbf{ do } \mathbf{S} \textbf{ od } \textbf{od} \end{array}$$

Proof: By induction on the size of J, using the previous lemma.

5 Simple Backtracking: Algorithm Derivation

The type of algorithms we are considering are those where we want to count, or otherwise process, the set of solutions to a problem. These solutions are represented as sequences of elements from some domain $\mathcal{D}$ which satisfy two properties: "completeness" and "validity". We use $\mathcal{D}^*$ to denote the set of all finite sequences of elements of $\mathcal{D}$. Each of the solutions will be constructed by successively extending an incomplete but valid solution until it is either complete or we have some simple way of determining that there are no valid extensions. Our specification may be expressed as:

$$SPEC \ =_{\text{DF}} \ \textbf{for } p \in \{\, x \in \mathcal{D}^* \mid V(x) \ \wedge \ C(x) \,\} \ \textbf{do } process(p) \ \textbf{od}$$

where *process* is the procedure we will execute for each complete and valid sequence p. $V(p)$ is true for each valid p and $C(p)$ is true for each complete p.

An *initial segment* of a sequence p' is a sequence p such that $\exists q \in \mathcal{D}^*. p' = p +\!\!+ q$. A *proper initial segment* p of p' is an initial segment such that $p \neq p'$. For all proper initial segments p of p' we assume that:

- A complete value cannot be further extended, i.e. $C(p) \Rightarrow (\neg C(p') \ \wedge \ \neg V(p'))$; and
- Any proper initial segment of a valid value is also valid, i.e. $V(p') \Rightarrow V(p)$.

Since we are interested in extensions of valid solutions we extend the specification to process all the valid and complete extensions of a given sequence in an arbitrary order:

$$SPEC(p) \ =_{\text{DF}} \ \textbf{for } q \in \{\, x \in \mathcal{D}^* \mid p \sqsubseteq x \ \wedge \ V(x) \ \wedge \ C(x) \,\} \ \textbf{do } process(q) \ \textbf{od}$$

where $p \sqsubseteq x \ =_{\text{DF}} \ \exists p' \in \mathcal{D}^*. x = p +\!\!+ p'$.

The derivation of an algorithm from this specification follows three stages:

1. Introduce recursion;
2. Transform recursion to iteration;
3. Optimisation.

5.1 Introducing Recursion

The first step is to transform the specification into a suitable form for translation to a recursive procedure using Theorem 4.4. We want to transform $SPEC(p)$ into a statement containing copies

of $SPEC(p)$ where the value of p is "smaller" according to some well-founded ordering. First we introduce an **if** statement to take out special cases. We may assume that $SPEC(p)$ is only called when $V(p)$ is true since an invalid p can have no valid extensions. We know that $SPEC(p) \approx process(p)$ if $C(p)$ is true, so we introduce an **if** statement which tests $C(P)$. If $C(p)$ is false then all the values we want to process must be strictly greater than p, so we have:

$$SPEC(p) \approx \textbf{if } C(p) \textbf{ then } process(p) \textbf{ else for } q \in \{\, p +\!\!+ q \mid q \in \mathcal{D}^* \wedge V(p +\!\!+ q) \wedge C(p +\!\!+ q) \,\} \textbf{ do } process(q) \textbf{ od}$$

where we know that each element of the set we loop over will be longer than p. So we can write this set as a union of disjoint subsets:

$$\bigcup_{t \in \mathcal{D}} \{\, p +\!\!+ \langle t \rangle +\!\!+ q \in \mathcal{D}^* \mid V(p +\!\!+ \langle t \rangle +\!\!+ q) \wedge C(p +\!\!+ \langle t \rangle +\!\!+ q) \,\}.$$

This means we can refine the loop to a double loop (by Lemma 4.7):

$$\begin{array}{l} SPEC(p) \leq \\ \quad \textbf{if } C(p) \textbf{ then } process(p) \\ \qquad \textbf{else for } t \in \mathcal{D} \textbf{ do} \\ \qquad\quad \textbf{for } q \in \{\, p +\!\!+ \langle t \rangle +\!\!+ q \in \mathcal{D}^* \mid V(p +\!\!+ \langle t \rangle +\!\!+ q) \wedge C(p +\!\!+ \langle t \rangle +\!\!+ q) \,\} \textbf{ do} \\ \qquad\qquad process(q) \textbf{ od} \end{array}$$

If $\neg V(p +\!\!+ \langle t \rangle)$ then $\{\, p +\!\!+ \langle t \rangle +\!\!+ q \in \mathcal{D}^* \mid V(p +\!\!+ \langle t \rangle +\!\!+ q) \wedge C(p +\!\!+ \langle t \rangle +\!\!+ q) \,\} = \varnothing$ and the **for** loop refines to **skip**:

$$\begin{array}{l} SPEC(p) \leq \\ \quad \textbf{if } C(p) \textbf{ then } process(p) \\ \qquad \textbf{else for } t \in \mathcal{D} \textbf{ do} \\ \qquad\quad \textbf{if } V(p +\!\!+ \langle t \rangle) \\ \qquad\qquad \textbf{then for } q \in \{\, p +\!\!+ \langle t \rangle +\!\!+ q \in \mathcal{D}^* \mid V(p +\!\!+ \langle t \rangle +\!\!+ q) \wedge C(p +\!\!+ \langle t \rangle +\!\!+ q) \,\} \textbf{ do} \\ \qquad\qquad\quad process(q) \textbf{ od} \end{array}$$

So we have:

$$SPEC(p) \leq \textbf{if } C(p) \textbf{ then } process(p) \textbf{ else for } t \in \mathcal{D} \textbf{ do if } V(p +\!\!+ \langle t \rangle) \textbf{ then } SPEC(p +\!\!+ \langle t \rangle) \textbf{ fi od}$$

We know that the set $\{\, p +\!\!+ q \mid q \in \mathcal{D}^* \wedge V(p +\!\!+ q) \wedge C(p +\!\!+ q) \,\}$ is finite, so there is an upper limit to the length of valid sequences, say L. So we can use $L - \ell(p)$ as a variant function and introduce recursion using Theorem 4.4:

$$\begin{array}{l} SPEC(p) \leq \textbf{proc } processall(p) \equiv \\ \quad \textbf{if } C(p) \textbf{ then } process(p) \\ \qquad \textbf{else for } t \in \mathcal{D} \textbf{ do} \\ \qquad\quad \textbf{if } V(p +\!\!+ \langle t \rangle) \textbf{ then } processall(p +\!\!+ \langle t \rangle) \textbf{ fi od.} \end{array}$$

We have renamed the recursive procedure F, provided by Theorem 4.4, to *processall* and made p a parameter of this procedure.

5.2 Recursion Removal

Having introduced recursion, the next step is to transform the recursive procedure to an iterative equivalent, using Theorem 4.5. First, note that we can replace the parameter p by a global variable because we can restore its value and the value of t after a recursive call by doing $t \xleftarrow{\text{last}} p$. The nondeterministic **for** loop introduces another local variable $\mathcal{D}'$ (as well as t) which records the remaining elements of $\mathcal{D}$ which have yet to be processed. If we assume (without loss of generality) that $\mathcal{D}$ is the set of integers from 1 to D (the size of $\mathcal{D}$), i.e. $\mathcal{D} = \{\, i \in \mathbb{N} \mid 1 \leqslant i \leqslant D \,\}$, then we can refine the nondeterministic **for** loop into a deterministic loop which processes the elements of

$\mathcal{D}$ in order. This is because the value of t tells us which elements of $\mathcal{D}$ have yet to be processed, in fact $\mathcal{D}' = \{\, i \in \mathbb{N} \mid t < i \leqslant D \,\}$:

$SPEC \leq$
  var $p := \langle\rangle$:
    processall **end**
  where
  proc *processall* $\equiv$
    if $C(p)$ **then** *process*(p)
      else $t := 1$;
        while $t \leqslant D$ **do**
          if $V(p +\!\!+ \langle t \rangle)$ **then** $p := p +\!\!+ \langle t \rangle$; *processall*; $t \xleftarrow{\text{last}} p$ **fi**;
          $t := t + 1$ **od.**

The procedure *processall* processes all valid extensions of (the global variable) p in a particular order. Our specification is "incomplete" in the sense that it doesn't specify the order in which the valid and complete elements are to be processed. An implementation of the specification is thus free to choose the most convenient order.

Restructure the procedure body as an action system:

proc *processall* $\equiv$
  actions A :
  $A \equiv$ **if** $C(p)$ **then** *process*(p); **call** Z
    else $t := 1$; **call** A_1 **fi.**
  $A_1 \equiv$ **if** $t \leqslant D$ **then if** $V(p +\!\!+ \langle t \rangle)$ **then call** B_1
    else call A_2 **fi**
    else call Z **fi.**
  $A_2 \equiv t := t + 1$; **call** A_1**.**
  $B_1 \equiv p := p +\!\!+ \langle t \rangle$; *processall*; $t \xleftarrow{\text{last}} p$; **call** A_2**. endactions.**

This is now in the right form for applying the recursion removal transformation (Theorem 4.5). There is one "B-type" action (B_1) which contains one recursive call. So $\mathbf{S}_{10} = p := p +\!\!+ \langle t \rangle$ and $\mathbf{S}_{11} = t \xleftarrow{\text{last}} p$; **call** A_2. Removing the recursion we get:

proc *processall* $\equiv$
  var $L := \langle\rangle, m$:
    actions A :
    $A \equiv$ **if** $C(p)$ **then** *process*(p); **call** $\hat{F}$
      else $t := 1$; **call** A_1 **fi.**
    $A_1 \equiv$ **if** $t \leqslant D$ **then if** $V(p +\!\!+ \langle t \rangle)$ **then call** B_1
      else call A_2 **fi**
      else call $\hat{F}$ **fi.**
    $A_2 \equiv t := t + 1$; **call** A_1**.**
    $B_1 \equiv p := p +\!\!+ \langle t \rangle$; $L := \langle 0, \langle 1, 1 \rangle\rangle +\!\!+ L$; **call** $\hat{F}$**.**
    $\hat{F} \equiv$ **if** $L = \langle\rangle$ **then call** Z
      else $m \xleftarrow{\text{pop}} L$;
        if $m = 0 \rightarrow$ **call** A
        $\square$ $m = \langle 1, 1 \rangle \rightarrow t \xleftarrow{\text{last}} p$; **call** A_2 **fi fi. endactions end.**

5.3 Optimisation

As with the factorial algorithm (Section 4.7) we push 0 onto L and immediately pop it and call A. So we can avoid the push and call A directly. As before, we now have a stack of identical elements which could be implemented as an integer. In this case however, we can do even better, since the length of L is the same as the length of p, so we can test p instead of L and remove L altogether:

$\underline{\textbf{proc}}$ *processall* $\equiv$
  $\underline{\textbf{actions}}$ A :
  $A \equiv \underline{\textbf{if}}\ C(p)\ \underline{\textbf{then}}\ process(p);\ \underline{\textbf{call}}\ \hat{F}$
    $\underline{\textbf{else}}\ t := 1;\ \underline{\textbf{call}}\ A_1\ \underline{\textbf{fi}}$.
  $A_1 \equiv \underline{\textbf{if}}\ t \leqslant D\ \underline{\textbf{then}}\ \underline{\textbf{if}}\ V(p +\!\!+ \langle t \rangle)\ \underline{\textbf{then}}\ \underline{\textbf{call}}\ B_1$
      $\underline{\textbf{else}}\ \underline{\textbf{call}}\ A_2\ \underline{\textbf{fi}}$
    $\underline{\textbf{else}}\ \underline{\textbf{call}}\ \hat{F}\ \underline{\textbf{fi}}$.
  $A_2 \equiv t := t + 1;\ \underline{\textbf{call}}\ A_1$.
  $B_1 \equiv p := p +\!\!+ \langle t \rangle;\ \underline{\textbf{call}}\ A$.
  $\hat{F} \equiv \underline{\textbf{if}}\ p = \langle\rangle\ \underline{\textbf{then}}\ \underline{\textbf{call}}\ Z$
    $\underline{\textbf{else}}\ t \xleftarrow{\text{last}} p;\ \underline{\textbf{call}}\ A_2\ \underline{\textbf{fi}}$. $\underline{\textbf{endactions}}$.

Remove the action system and restructure:

SPEC $\approx$
  $\underline{\textbf{var}}\ p := \langle\rangle, t$:
    $\underline{\textbf{do}}\ \underline{\textbf{if}}\ C(p)\ \underline{\textbf{then}}\ process(p);$
      $\underline{\textbf{if}}\ p = \langle\rangle\ \underline{\textbf{then}}\ \underline{\textbf{exit}}\ \underline{\textbf{fi}};$
      $t \xleftarrow{\text{last}} p;\ t := t + 1$
    $\underline{\textbf{else}}\ t := 1\ \underline{\textbf{fi}};$
    $\underline{\textbf{do}}\ \underline{\textbf{if}}\ V(p +\!\!+ \langle t \rangle) \wedge t \leqslant D\ \underline{\textbf{then}}\ \underline{\textbf{exit}}$
      $\underline{\textbf{elsif}}\ p = \langle\rangle \wedge t > D\ \underline{\textbf{then}}\ \underline{\textbf{exit}}(2)$
      $\underline{\textbf{elsif}}\ t > D\ \underline{\textbf{then}}\ t \xleftarrow{\text{last}} p;\ t := t + 1$
      $\underline{\textbf{else}}\ t := t + 1\ \underline{\textbf{fi}}\ \underline{\textbf{od}};$
    $p := p +\!\!+ \langle t \rangle\ \underline{\textbf{od}}\ \underline{\textbf{end}}$

Take the first statement out of the loop and convert to a single loop. We will assume $C(\langle\rangle)$ is false (since otherwise no other sequences can be valid) and define $C'(t,p) =_{\text{DF}} C(p +\!\!+ \langle t \rangle)$, $V'(t,p) =_{\text{DF}} V(p +\!\!+ \langle t \rangle)$, $process'(t,p) =_{\text{DF}} process(p +\!\!+ \langle t \rangle)$.

$\underline{\textbf{var}}\ p := \langle\rangle, t := 1$:
  $\underline{\textbf{do}}\ \underline{\textbf{if}}\ V'(t,p) \wedge t \leqslant D\ \underline{\textbf{then}}\ p := p +\!\!+ \langle t \rangle;$
      $\underline{\textbf{if}}\ C(p)\ \underline{\textbf{then}}\ process(p);\ t \xleftarrow{\text{last}} p;\ t := t + 1$
      $\underline{\textbf{else}}\ t := 1\ \underline{\textbf{fi}};$
    $\underline{\textbf{elsif}}\ p = \langle\rangle \wedge t > D\ \underline{\textbf{then}}\ \underline{\textbf{exit}}$
    $\underline{\textbf{elsif}}\ t > D\ \underline{\textbf{then}}\ t \xleftarrow{\text{last}} p;\ t := t + 1$
      $\underline{\textbf{else}}\ t := t + 1\ \underline{\textbf{fi}}\ \underline{\textbf{od}}\ \underline{\textbf{end}}$

Push the statement $p := p +\!\!+ \langle t \rangle$ into the inner $\underline{\textbf{if}}$ statement:

$\underline{\textbf{var}}\ p := \langle\rangle, t := 1$:
  $\underline{\textbf{do}}\ \underline{\textbf{if}}\ V'(t,p) \wedge t \leqslant D\ \underline{\textbf{then}}\ \underline{\textbf{if}}\ C'(t,p)\ \underline{\textbf{then}}\ process'(t,p);\ t := t + 1$
      $\underline{\textbf{else}}\ p := p +\!\!+ \langle t \rangle;\ t := 1\ \underline{\textbf{fi}};$
    $\underline{\textbf{elsif}}\ p = \langle\rangle \wedge t > D\ \underline{\textbf{then}}\ \underline{\textbf{exit}}$
    $\underline{\textbf{elsif}}\ t > D\ \underline{\textbf{then}}\ t \xleftarrow{\text{last}} p;\ t := t + 1$
      $\underline{\textbf{else}}\ t := t + 1\ \underline{\textbf{fi}}\ \underline{\textbf{od}}\ \underline{\textbf{end}}$

Finally, re-arrange the tests to make a $\underline{\textbf{while}}$ loop:

$\underline{\textbf{var}}\ p := \langle\rangle, t := 1$:
  $\underline{\textbf{while}}\ p \neq \langle\rangle \vee t \leqslant D\ \underline{\textbf{do}}$
      $\underline{\textbf{if}}\ t > D\ \underline{\textbf{then}}\ t \xleftarrow{\text{last}} p;\ t := t + 1$
      $\underline{\textbf{elsif}}\ V'(t,p)\ \underline{\textbf{then}}\ \underline{\textbf{if}}\ C'(t,p)\ \underline{\textbf{then}}\ process'(t,p);\ t := t + 1$
        $\underline{\textbf{else}}\ p := p +\!\!+ \langle t \rangle;\ t := 1\ \underline{\textbf{fi}}$
    $\underline{\textbf{else}}\ t := t + 1\ \underline{\textbf{fi}}\ \underline{\textbf{od}}\ \underline{\textbf{end}}$

This is our basic backtracking algorithm. The derivation used only transformations which have been proved to preserve the semantics (Ward (1989), Ward (1991a), Ward (1992), Ward (1994)) so we can guarantee that this algorithm correctly implements the specification *SPEC*.

6 Some Applications

This section outlines the problems which gave rise to the algorithms presented in this paper. These problems concern the concrete representation, by functions or by sets, of algebraic structures. Stone duality for Boolean algebras provides a prototype for such representations. We outline the mathematical background shortly, but begin by describing the form of the backtracking algorithm which we need.

6.1 Relation-preserving Maps

We are given two finite relational structures of the same type. That is, we have two finite sets X and Y and two finite sets of relations R_X on X and R_Y of Y such that for each relation ρ in R_X there is a corresponding relation ρ' in R_Y of the same arity, and vice versa. For simplicity in this subsection we shall assume that all relations are binary. The aim is to find all the maps $\Psi : X \ \rightarrow \ Y$ which preserve all relations, i.e. for all relations $\rho \in R_X$, all pairs of elements in X which are related by ρ, map to elements in Y which are related by ρ'. More formally:

Definition 6.1 A function $\Psi : X \ \rightarrow \ Y$ is *relation preserving* iff:

$$\forall \rho \in R_X . \forall x, y \in X . (x \ \rho \ y) \Rightarrow (\Psi(x) \ \rho' \ \Psi(y))$$

Thus we want to find the size of the set:

$$\left\{ \ \Psi : X \ \rightarrow \ Y \mid \forall \rho \in R_X . \forall x, y \in X . (x \ \rho \ y) \Rightarrow (\Psi(x) \ \rho' \ \Psi(y)) \ \right\}$$

Without loss of generality we may take X and Y to be sets of integers: $X = \{1, 2, \ldots, \#X\}$ and $Y = \{1, 2, \ldots, \#Y\}$. We can represent a partial map $\Psi : \{1, 2, \ldots, n\} \ \rightarrow \ Y$ (where $n \leqslant \#X$) as a sequence p of length n where $p[i] = \Psi(i)$. A complete sequence is one of length $\#X$ and a valid sequence is a relation preserving one. So we have the definitions:

$$\begin{aligned} C(p) \ &=_{\text{DF}} \ \ell(p) = \#X \\ V(p) \ &=_{\text{DF}} \ \forall \rho \in R_X . \forall x, y, 1 \leqslant x, y \leqslant \ell(p) . (x \ \rho \ y) \Rightarrow (p[x] \ \rho' \ p[y])) \end{aligned}$$

Any subset of a (partial or total) relation preserving map is also relation preserving, so these definitions clearly satisfy the conditions for the backtracking algorithm.

The iterative algorithm actually uses $V'(t, p)$ (defined as $V'(t, p) = V(p +\!\!+ \langle t \rangle)$ which is only evaluated when $V(p)$ is true. This means that we only need to check the pairs (x, y) where one or both of x or y is equal to t. So we can use the definitions:

$$\begin{aligned} V'(t, p) \ &=_{\text{DF}} \ \forall \rho \in R_X . \forall x, 1 \leqslant x < n . (n \ \rho \ x \Rightarrow t \ \rho' \ p[x]) \ \wedge \ (x \ \rho \ n \Rightarrow p[x] \ \rho' \ t) \ \wedge \ (n \ \rho \ n \Rightarrow t \ \rho' \ t) \\ C'(t, p) \ &=_{\text{DF}} \ n = \#X \end{aligned}$$

where $n = \ell(p) + 1$ (so $V'(t, p)$ is testing if t is a valid image for n in the extension of p from $\{1, 2, \ldots, n-1\}$ to $\{1, 2, \ldots, n\}$).

For the implementation we only need to record the sizes of X and Y (in variables SX and SY). We represent the two sets of relations using two three-dimensional integer arrays RX and RY (we could use boolean arrays but integer arrays are probably slightly faster to access and memory is not at a premium). The integer *rho* represents the relation ρ where:

$$RX[rho, x, y] = \begin{cases} 1 & \text{if } x \ \rho \ y \\ 0 & \text{otherwise} \end{cases} \qquad \text{and} \qquad RY[rho, x, y] = \begin{cases} 1 & \text{if } x \ \rho' \ y \\ 0 & \text{otherwise} \end{cases}$$

The relation-preserving test is implemented as a double-nested **while** loop which stores the result in Boolean variable *rp*. The loops terminate as soon as *rp* becomes false, to avoid unnecessary testing. The variable R records the number of relations. So we have the following testing procedure which sets *rp* to **true** iff $V'(t,p)$ is true, provided $n = \ell(p)+1$ and $V(p)$ is true:

proc $DO(t,p,n)$ $\equiv$
  var $np := 0$:
    $rp :=$ **true**; $rho := 1$;
    while $rho \leqslant R \wedge rp$ **do**
        if $RX[rho,n,n] = 1 \wedge RY[rho,t,t] = 0$
          then $rp :=$ **false**
          else $np := 1$;
              while $np < n \wedge rp$ **do**
                  if $(RX[rho,n,np] = 1 \wedge RY[rho,t,p[np]] = 0)$
                      $\vee$ $(RX[rho,np,n] = 1 \wedge RY[rho,p[np],t] = 0)$
                    then $rp :=$ **false** **fi** **od** **fi** **od** **end.**

Note that this version will repeatedly search for the set of elements related to a particular element in X for each relation. If the X relations are fairly sparse (not many pairs of elements related) then it will be more efficient to represent the X relations using two integer arrays *rel_to_X* and *rel_X_to* which record the following information:

$$\begin{aligned}
rel_to_X[rho,n,0] &= \text{the number of elements } x \text{ such that } n\ \rho\ x\\
rel_to_X[rho,n,i] &= \text{the } i\text{th element } x_i \text{ such that } n\ \rho\ x_i\\
rel_X_to[rho,n,0] &= \text{the number of elements } x \text{ such that } x\ \rho\ n\\
rel_X_to[rho,n,i] &= \text{the } i\text{th element } x_i \text{ such that } x_i\ \rho\ n
\end{aligned}$$

Then our improved version of *DO* is:

proc $DO(p,t,n)$ $\equiv$
  var $np := 0, numrels := 0, i := 0$:
    $rp :=$ **true**; $rho := 1$;
    while $rho \leqslant R \wedge rp$ **do**
        $numrels := rel\_to\_X[rho,n,0]$;
        $i := 1$;
        while $i \leqslant numrels \wedge rp$ **do**
            $np := rel\_to\_X[rho,n,i]$;
            if $np < n$ **then** **if** $RY[rho,t,p[np]] = 0$ **then** $rp :=$ **false** **fi**
            elsif $np = n$ **then** **if** $RY[rho,t,t] = 0$ **then** $rp :=$ **false** **fi** **fi**;
            $i := i+1$ **od**;
        if rp
          then $numrels := rel\_X\_to[rho,n,0]$;
              $i := 1$;
              while $i \leqslant numrels \wedge rp$ **do**
                  $np := rel\_X\_to[rho,n,i]$;
                  if $np < n \wedge RY[rho,p[np],t] = 0$ **then** $rp :=$ **false** **fi**;
                  $i := i+1$ **od** **fi** **od** **end.**

6.2 Problems in Duality Theory

Suppose we are given a class $\mathscr{A}$ of algebras of a fixed type. Assume further that $\mathscr{A}$ is generated from a given finite algebra $\underline{P}$ by forming products, subalgebras and isomorphic copies: in symbols, $\mathscr{A} = \mathbb{ISP}(\underline{P})$. Such structures arise frequently as algebraic models for classical and non-classical logics, for example. In this situation, $\underline{P}$ plays the role of "truth value algebra". For example:

1. Take $\underline{P}$ to be the 2-element distributive lattice $\underline{\mathbf{2}} = (\{0,1\}; \vee, \wedge, 0, 1)$ (so 0 and 1 are treated as nullary operations). Then $\mathscr{A}$ is the class $\mathbf{D}$ of $\{0,1\}$-distributive lattices.
2. Take $\underline{P}$ to be the 2-element Boolean algebra $(\{0,1\}; \vee, \wedge, ', 0, 1)$. Then $\mathscr{A}$ is the class $\mathbf{B}$ of Boolean algebras.
3. By taking $\underline{P}$ to be $(\{0, a, 1\}; \vee, \wedge, \sim, 0, 1)$, where $(\{0, a, 1\}; \vee, \wedge, 0, 1)$ is the $\{0,1\}$-distributive lattice with $0 < a < 1$ and the negation operator $\sim$ satisfies $\sim 0 = 1$, $\sim 1 = 0$ and $\sim a = a$, we obtain the class $\mathbf{K}$ of Kleene algebras.

In these and in other logic-based examples $\underline{P}$ has an underlying lattice structure, with the operations $\vee$ and $\wedge$ modelling disjunction and conjunction. We shall henceforth always assume that $\underline{P}$ has a lattice reduct, since this simplifies the theory on which we rely (though we note that interesting work lies ahead on classes of algebras where this restriction is not satisfied). In many cases it happens that $\mathscr{A} = \mathbb{ISP}(\underline{P})$ coincides with the variety $\mathbb{HSP}(\underline{P})$, where $\mathbb{H}$ denotes the formation of homomorphic images. Then, by a famous theorem of G. Birkhoff, $\mathscr{A}$ can be specified by a set of identities. This occurs for each of $\mathbf{D}$, $\mathbf{B}$ and $\mathbf{K}$ above. (Where the quasivariety $\mathbb{ISP}(\underline{P}$ is strictly smaller than the variety $\mathbb{HSP}(\underline{P})$, all is not lost. However a more complicated representation theory, using multi-sorted structures, is then required (see Davey & Priestley (1987)).)

Given $\mathscr{A} = \mathbb{ISP}(\underline{P})$, we may seek a concrete representation for the algebras in $\mathscr{A}$. Another important question to address is the determination of the free algebra $F\mathscr{A}(n)$ on n generators (note that the free algebras in $\mathbb{HSP}(\underline{P})$ lie in $\mathbb{ISP}(\underline{P})$ always, so that for this problem it is sufficient to consider classes of the latter form). A major systematic study of the representation of algebras, in a manner which represents the free algebras in a very natural way, was undertaken by B.A. Davey and H. Werner in Davey & Werner (1983). Their theory encompasses many well-known dualities (including Stone duality for $\mathbf{B}$ and Priestley duality for $\mathbf{D}$) within a common framework. In the present paper it is finite structures that concern us. Accordingly we shall restrict to the finite algebras in $\mathscr{A}$. This spares us having to introduce the topological machinery involved in representing arbitrary algebras. The representation we require relies on an appropriate choice of a relational structure $\underset{\sim}{P} = (P; R)$ on the underlying set P of $\underline{P}$. Given any set R of relations on P we extend each $\rho \in R$ pointwise to powers of P. For each finite $A \in \mathscr{A}$ define the *dual* of A to be $D(A)$, where $D(A)$ is the set $\mathscr{A}(A, \underline{P})$ of $\mathscr{A}$-homomorphisms from A into $\underline{P}$, with relational structure inherited from $\underset{\sim}{P}^A$. Then, from $X = D(A)$ we form the algebra $E(X)$, defined to be the set of R-preserving maps from X into $\underset{\sim}{P}$, with algebraic structure inherited from $\underline{P}^X$. Then the theory in Davey & Werner (1983) (specifically Theorem 1.18) implies that we have the following theorem.

Theorem 6.2 *Assume that $\mathscr{A} = \mathbb{ISP}(\underline{P})$ is a class of algebras such that $\underline{P}$ is a finite algebra with an underlying lattice structure. Then it is possible to choose R so that*

1. *$A \cong ED(A)$ for each finite $A \in \mathscr{A}$, and*
2. *$D(F\mathscr{A}(n)) = \underset{\sim}{P}^n$ (so that $F\mathscr{A}(n)$ is the algebra of all R-preserving maps from $\underset{\sim}{P}^n$ to $\underset{\sim}{P}$ $(1 \leqslant n < \infty)$).*

Further, R above can be chosen to consist of binary relations, each of which is a subalgebra of $\underline{P}^2$.

If (1) in Theorem 6.2 holds we say that R *yields a duality on (the finite algebras in)* $\mathscr{A}$. For Boolean algebras we take $R = \varnothing$, while for $\mathbf{D}$ we obtain Priestley duality by choosing R on $\{0,1\}$ to contain the single relation $\leqslant$, the inequality relation in which $0 < 1$ (so $\underset{\sim}{\mathbf{2}}$ is the 2-element chain, *qua* ordered set). In these examples a suitable set R was recognised with hindsight, the dualities being known before the Davey–Werner theory was developed. Since the publication of Davey & Werner (1983) various techniques (notably Davey and Werner's piggyback method) have been devised which make it quite easy to identify a set R which will yield a duality on $\mathscr{A}$. However such a set may often be too large and complex to lead to a workable duality. This is strikingly illustrated by subvarieties of the variety $\mathbf{B}_\omega$ of distributive p-algebras. These varieties were first discussed by K.B. Lee Lee (1970). He showed that the proper non-trivial subvarieties of $\mathbf{B}_\omega$ form a chain

$\mathbf{B}_0 \subset \mathbf{B}_1 \subset \ldots$ (in which $\mathbf{B}_0 = \mathbf{B}$ and $\mathbf{B}_1$ is the class known as Stone algebras). These varieties may be defined equationally. Alternatively, Birkhoff's Subdirect Product Theorem implies that they are equivalently given by $\mathbf{B}_n = \mathbb{ISP}(\underline{P}_n)$, where $\underline{P}_n = (\mathbf{2}^n \oplus \mathbf{1}; \vee, \wedge, ^*, 0, 1)$ denotes the n-atom Boolean lattice with a new top adjoined, and with an operation * of pseudocomplementation given by

$$a^* = \max\{\, c \mid a \wedge c = 0 \,\}.$$

To avoid degenerate cases we henceforth assume $n \geqslant 3$. As shown in Davey & Priestley (1993a), a duality is obtained for $\mathbf{B}_n$ by taking $\underset{\sim}{P}_n = (P_n; R_n)$, where the set R_n of relations consists of

(i) the graphs of 3 endomorphisms, e, f, g of $\underline{P}_n$, and

(ii) a set T_n of subalgebras of $\underline{P}_n^2$ indexed by the partitions of the integer n.

In (i), f and g are automorphisms and are determined by the permutations they induce on the atoms of $\underline{P}_n$, *viz.* the cycles $(1\ 2\ \ldots\ n)$ and $(1\ 2)$. In (ii) $|T_n|$ grows exponentially with n, and it is natural to ask whether any proper subset of R_n still serves to yield a duality. A partition of n into k parts is a k-tuple $(\lambda_1, \ldots, \lambda_k)$ of natural numbers where $\lambda_1 \geqslant \cdots \geqslant \lambda_k$ and $\sum_{i=1}^{k} \lambda_i = n$. Two of the partition-induced relations in T_n are isomorphic as algebras precisely when the associated partitions have the same number of parts. An optimistic but reasonable conjecture was that a duality would be obtained by reducing T_n by selecting just one k-part partition for each k (giving $n+3$ relations in total).

Given a set R of subalgebras of $\underline{P}^2$ that is known to yield a duality for a class $\mathscr{A} = \mathbb{ISP}(\underline{P})$, how might we test whether a proper subset $R' = R \setminus \{\rho\}$ still yields a duality? Certainly if the reduced set R' fails to give $A \cong ED(A)$ for just one $A \in \mathscr{A}$ then the relation ρ cannot be discarded. Dropping a relation cannot decrease the size of $ED(A)$, so the point at issue is whether the number of R'-preserving maps from $D(A)$ to $(P; R')$ is greater than the size of the *test algebra* A. We say R' *yields a duality on* A if no extra maps become allowable.

When trying to decide whether $\rho \in R$ can be discarded a natural choice for a test algebra A is $\underline{\rho}$, by which we mean the relation ρ regarded as an algebra (remember that each of our relations is a subalgebra of $\underline{P}^2$). Here (at last!) we have a computational problem: compare the size of A with the size of the set $ED(A)$ of R'-preserving maps from $D(A)$ into $(P; R')$. The very earliest version of our backtracking algorithm (an implementation in VAX BASIC) successfully demonstrated that none of the partition-induced relations could be discarded from the duality for $\mathbf{B}_3$ (as expected, since each partition of 3 has a different number of parts). The critical test case for our conjecture came with $n = 4$ and the relations ρ_1 and ρ_2 associated with the 2-part partitions $(2, 2)$ and $(3, 1)$. Here $D(\underline{\rho}_1) = D(\underline{\rho}_2)$ has 42 elements, $|P_4| = 17$, and $|R_4| = 8$. We sought to calculate the number of maps $\Psi : D(\underline{\rho}_i) \to \underset{\sim}{P}_4$ preserving $R_4 \setminus \{\rho_i\}$ $(i = 1, 2)$. These calculations were successfully carried through on a PC, but only after a judicious choice of ordering of the elements of the domain had been made. Before indicating how element order affects the calculations we conclude the history of the $\mathbf{B}_n$ problem.

It turned out that dropping either ρ_1 or ρ_2 did not destroy the duality on the test algebra $\underline{\rho}_1 = \underline{\rho}_2$ (though dropping both did). This negative result was consistent with the conjecture that only one of the relations was needed, but did not prove it. At this point, examination of the computer output provided sufficient insight to enable the conjecture to be confirmed mathematically for $n = 4$, and subsequently for general n. Much more significantly it led Davey and Priestley (Davey & Priestley (1993b)) to prove a theorem of which the following is a special case.

Theorem 6.3 *Assume $\mathscr{A}$ is as in Theorem 6.2, that R yields a duality on $\mathscr{A}$, and let $R' = R \setminus \{\rho\}$ $(\rho \in R)$. Then R' yields a duality on $\mathscr{A}$ if and only if R' yields a duality on the algebra $\underline{\rho}$.*

Thus testing for redundancy of any given relation in a duality is reduced to a finite problem, solvable by application of the backtracking algorithm (subject of course to computational feasibility).

Language	Machine	Time (simple)	Time (improved)
VAX BASIC	VAX	not available	not available
GW BASIC	286 PC	2 hrs 24 mins	not available
GW BASIC	386 PC	49 mins	not available
C (gcc compiler)	Sun 3/50	3.9 seconds	0.84 seconds
Turbo PASCAL	286 PC	7 mins 30 secs	5.1 secs
Turbo PASCAL	386 PC	1 min 52 secs	1.2 secs
C (gcc compiler)	Sparc 2	0.43 seconds	0.08 seconds

Table 2: Timings for different implementations solving the $\mathbf{B}_4$ problem

We now return to computational aspects of the $\mathbf{B}_4$ problem. The elements of the domain set fall into disjoint orbits under the action of the automorphisms f and g. Also, if Ψ is relation-preserving, then if i is mapped to $\Psi(i)$ then $f(i)$ must be mapped to $f(\Psi(i))$ and $g(i)$ to $g(\Psi(i))$). These observations lead us to order the domain in the following way. We start from an arbitrary element, denoted 1, and take $2 = f(1)$, $3 = g(1)$ (unless $f(1) = g(1)$). Thereafter we pick as the next element in the order the f- or g-image of the first listed element whose images have not already been included until the orbit of 1 is exhausted. This process is repeated for the remaining orbits. We thereby get a highly economical search tree, of 17,391 nodes. Note that the element ordering heuristics of Section 7 are very good at finding such chains of relationships automatically—even starting from a random permutation, they have so far always managed to find a better permutation than the best "hand crafted" efforts! For example, a typical calculation for the $\mathbf{B}_4$ problem yielded a search tree with 10,336 nodes.

Some comparative timings for the $\mathbf{B}_4$ problem for different implementations are shown in Table 2.

The VAX BASIC version was the first to be implemented, it was never actually used for this problem which is why the times are not available. The first implementation on a Sun 3/50 was written in perl which is an interpreted language more suited to string processing than numerical processing. This gave timings roughly similar to the GW BASIC version. Switching to a compiled language, i.e. C on the Sun 3/50, produced a dramatic improvement in speed which encouraged us to tackle some much larger examples.

Tables 3 and 4 illustrate the importance of a "good" permutation for this problem, Table 3 shows the effect of making small changes to a good permutation. The permutation for each entry in the table is produced by composing the previous permutation with a permutation of the form $(i\ i+1\ \ldots\ j)$ where $i < j$. For example composing (1 2 3 4 5 6 7 8 9) with (4 5 6) yields (1 2 3 5 6 4 7 8 9). We call this operation an *insertion*. Table 3 shows the effect of a sequence of

Insertions	Result	Trials Required	Insertions	Result	Trials Required
1	21	56,083	6	21	261,647
2	21	196,163	7	21	527,663
3	21	211,191	8	21	1,104,847
4	21	228,327	9	21	4,495,633
5	21	211,735	10	21	6,289,235

Table 3: The effect of random insertions on a "good" permutation for the $\mathbf{B}_4$ problem

random insertions starting with the permutation used above. Table 4 shows what happens to the same problem when a random permutation is chosen (computing this table required over 1 week of CPU time on a Sparc 2). Each pair of results is for a different random permutation with the same problem as above. The "estimated" values for the search tree were calculated using Knuth's

Estimated	Actual	Estimated	Actual
138,568,972,267	$> 10,000,000,000$	637,837,147,299	$> 10,000,000,000$
190,500,933	190,764,514	16,911,522,238	$> 10,000,000,000$
174,773,383	174,780,145	530,343,262,662	$> 10,000,000,000$
55,741,733,482,643	$> 10,000,000,000$	740,322,532	724,756,716
868,614,966	867,753,321	259,217,324	257,759,780
17,217,321,614	$> 10,000,000,000$	32,654,546,016	$> 10,000,000,000$
122,313,962	121,979,114	28,751,130	28,464,800
557,074,692	554,723,651	441,680,890	441,982,864
275,179,396,064	$> 10,000,000,000$	27,659,921,478	$> 10,000,000,000$
31,348,549,817	$> 10,000,000,000$	16,474,273	16,314,730
3,574,075,557	3,568,600,866	106,754,057,688	$> 10,000,000,000$
2,342,038,092	2,347,470,755	6,362,152,171	6,417,860,672
454,900,453	454,932,869	14,821,585,522	$> 10,000,000,000$

Table 4: Using random permutations on the $\mathbf{B}_4$ problem

backtracking estimation method (Knuth (1975)) with 1,000,000 probes, see Section 7.4 for details. Using Knuth's estimation method on 487 random permutations with 1,000,000 probes each yielded an average search tree size of 3,145,968,416,638 nodes. This corresponds to an execution time (on a Sparc 2) of about 3 years, while the hybrid method requires an average of 16 seconds and examines a total of about 800,000 nodes.

For a larger problem of the same type (testing a duality for optimality) with a 153 element domain, a 33 element range and 7 relations, the average search tree size for a random permutation is around 5×10^{35} which indicates an execution time of 5×10^{23} years (about 30 million million times the age of the visible universe). The hybrid method reduces this to 37,09,801 nodes and 763 seconds.

6.3 Other Applications

Priestley duality tells us that a finite distributive lattice L is concretely represented as the set of order-preserving maps from its dual $D(L) = \mathbf{D}(L, \underset{\sim}{\mathbf{2}})$, ordered pointwise, into the 2-element chain $\underset{\sim}{\mathbf{2}}$. Further, given finite distributive lattices L and M, there is a bijection between the $\mathbf{D}$-homomorphisms from L to M and order-preserving maps from $D(M)$ to $D(L)$. See Chapter 8 of Davey & Priestley (1990) for a textbook account of this theory, which allows problems about finite distributive lattices to be translated into problems about finite ordered sets. The map $L \mapsto D(L)$ acts like a "logarithm": in general $|L|$ grows exponentially with $|D(L)|$. Computationally this is highly significant: problems which are intractable in their lattice form become accessible once translated into ordered set terms. The relation-preserving maps algorithm can obviously be used to calculate the order-preserving maps from one finite ordered set $(P; \leqslant)$ to another $(Q; \leqslant)$. In particular we can find the order-preserving maps from $(P; \leqslant)$ into $\underset{\sim}{\mathbf{2}}$. We note that various specialised algorithms have been derived which will handle this problem. Our algorithm has the merit that it very easily adapts to a wide diversity of other situations. We do not claim that even when enhanced by the heuristics in Section 8 it will necessarily out-perform algorithms tailored for specific problems. Consider, for example, the determination of the cardinality of the free $\{0,1\}$-distributive lattice on n generators. The elements of this lattice are well-known to be the order-preserving maps from $\underset{\sim}{\mathbf{2}}^n$ to $\underset{\sim}{\mathbf{2}}$; one proof is given by Theorem 6.3, applied in the case of Priestley duality. The values of $|F\mathbf{D}(n)|$ are known only for $n \leqslant 8$ (Weidemann (1991)recently computed $|F\mathbf{D}(8)|$). Our algorithm calculates these values easily for $n \leqslant 6$ but is defeated by the case $n = 7$, for which a more sophisticated mathematical approach seems essential. Empirical evidence suggests that the determination of the number of order-preserving maps from an m-element ordered set into

$\underset{\sim}{\mathbf{2}}$ is always viable for $m \leqslant 2^6$, and is viable in many instances for $m \leqslant 2^7$, but with worst case behaviour which renders our algorithm impractical in those cases.

Now assume, as in 6.2, that a set R of relations on P gives a duality for a class of algebras $\mathscr{A} = \mathbb{ISP}(\underline{P})$. By Theorem 6.3, the free algebra $F\mathscr{A}(n)$ is given by the R-preserving maps from $\underset{\sim}{P}^n$ to $\underset{\sim}{P}$), where $\underset{\sim}{P} = (P; R)$. Both theory and experience tell us that for algebras arising in algebraic logic (the classes $\mathbf{B}_n$, various classes of Heyting algebras, etc.) the norm is that these free algebras grow exceedingly rapidly with n, the more so if $|P| > 2$. For example, $|F\mathbf{B}(n)| = 2^{2^n}$, and $|F\mathbf{K}(3)| = 43,918$ while $|F\mathbf{K}(4)| = 160,297,985,276$ (Berman & Mukaidono (1984)). Nevertheless we have successfully used our algorithm on some problems of this sort: see for example Priestley (1992).

The relation-preserving maps algorithm was devised to enable a given duality on a class of algebras $\mathscr{A}$ to be tested for optimality. Initially the input data files were set up by laborious hand calculation. In more recent applications of the technique (Davey & Priestley (1992), Priestley (1992)) the backtracking algorithm has been used to generate these data files. To see why this might be possible, recall that a domain set $D(A)$ (as in 6.2) is itself a set of maps, namely the $\mathscr{A}$-homomorphisms from A into $\underline{P}$. Such maps are just those which preserve the relations (not in general binary) which are the graphs of the operations. Of course, the procedures described in Section 6.1 easily adapt to relations of different arities. In the examples so far analysed, the algebras in $\mathscr{A}$ have always had an underlying distributive lattice structure, so that the full machinery of Priestley duality has been at our disposal. This has allowed us to work not with homomorphisms but with their dual equivalents, which are certain order-preserving maps; this is done throughout Davey & Priestley (1992), Davey & Priestley (1993a), Davey & Priestley (1993b) and Priestley (1992). We thus gain the benefit of the "logarithmic" feature of the duality. Further, duality has often allowed us to identify explicitly relations from a theoretical algebraic description (this was done, for example, for $\mathbf{B}_n$ in Davey & Priestley (1993a)). These calculations, once again, are done by a suitable application of the algorithm given in Section 6.1. The programs used here form part of a package which is an invaluable toolkit for anyone investigating algebras with an underlying distributive lattice structure. This kit includes, in particular, facilities for finding (in many classes of algebras) homomorphisms, congruences, subalgebras and retracts, and for isomorphism-testing.

Duality theory for distributive lattices has been very extensively used, partly because, being pictorial, it is exceedingly easy to work with. Representations do exist for arbitrary lattices which generalise that provided by Priestley duality. That given by A. Urquhart (Urquhart (1977)) replaces ordered sets by structures with two quasi-order relations, while G. Hartung's theory (Hartung (1992)) employs the formalism of concept analysis (introduced by R. Wille in Rival (1982), pp. 445–470. See also Ganter, Wille & Wolff (1987) or Chapter 11 of Davey & Priestley (1990)). The backtracking algorithm is ideally suited to making these representations into a practical tool. With the aid of this package, it is possible completely to automate the generation of optimal natural dualities in a large number of cases, for example, for certain Heyting algebra varieties (see Davey & Priestley (1993a)). Each such calculation requires many different subroutines, each of which employs backtracking in a different way.

7 Backtracking with Element Order Selection

In Section 5 we were searching for sequences of elements p which satisfy the predicates $V(p)$ and $C(p)$. The backtracking algorithm tries each possible extension of a partial sequence in turn. Since we will eventually have to try all the possible extensions, it makes no difference what order we choose to try them in: they will all have to be tried eventually. If we were searching for *one* element then we could use heuristics to try the "most likely" extensions first. Now consider the case where p is an array whose elements may be filled up in any order. The eight queens problem can be expressed in this form, as discussed in Section 2.2. With this type of problem the order in which the elements of the array are filled can have a dramatic effect on the execution time, as

illustrated in tables 3 and 4. In this section we will discuss various heuristics which we have used to select the element order: these have frequently enabled us to complete calculations which would otherwise be totally unfeasible.

We consider the same specification as in Section 5 except that all sequences are the same length, N, and a valid sequence is one which has no "unfilled" positions, i.e. $C(p) =_{\text{DF}} \forall i, 1 \leqslant i \leqslant N.p[i] \neq \perp$ where $\perp$ is a new element (not in $\mathcal{D}$) which is used to represend an unfilled position.

$$SPEC =_{\text{DF}} \underline{\textbf{for}}\ p \in \{\ p \in \mathcal{D}^* \mid V(p) \wedge C(p) \wedge \ell(p) = N\ \}\ \underline{\textbf{do}}\ process(p)\ \underline{\textbf{od}}$$

Instead of extending this to process all valid extensions of p, we need to process all valid *completions* of p where an incomplete sequence is a "partially filled array". Let $p \in (\mathcal{D} \cup \{\perp\})^*$ be of length N. Then we define:

$$SPEC(p) =_{\text{DF}} \underline{\textbf{for}}\ q \in \{\ q \in \mathcal{D}^* \mid p \sqsubseteq q \wedge V(q) \wedge C(q) \wedge \ell(q) = N\ \}\ \underline{\textbf{do}}\ process(q)\ \underline{\textbf{od}}$$

where $p \sqsubseteq q$ means that q is p with some of its unfilled elements filled in, i.e. $p \sqsubseteq q =_{\text{DF}} \forall i.\, 1 \leqslant i \leqslant N.(p[i] = \perp \vee p[i] = q[i])$.

By using a permutation $\pi : \{1, 2, \ldots, N\} \rightarrow \{1, 2, \ldots, N\}$ and a variable n to record how many elements of p are filled we can avoid the need for the extra element $\perp$. The permutation also records in which order the array is to be filled. The n elements $p[\pi[1]], \ldots, p[\pi[n]]$ of p are filled, and the $N - n$ elements $p[\pi[n+1]], \ldots, p[\pi[N]]$ of p are currently unfilled. A derivation similar to that in Section 5 yields the following algorithm:

```
var n := 0, t := 1:
  while n > 0 ∨ t ⩽ D do
        if t > D then t := p[π[n]]; n := n − 1; t := t + 1
        elsif V′(t, p, π, n) then if n = N then process′(t, p); t := t + 1
                                               else n := n + 1; p[π[n]] := t; t := 1 fi
                                  else t := t + 1 fi od end
```

This gives the same result as *SPEC* for any permutation $\pi : \{1, 2, \ldots, N\} \rightarrow \{1, 2, \ldots, N\}$. In fact we can permute the elements of $\pi[d+1]$, to $\pi[N]$ at any time during the execution of the program.

The selection of suitable values for π is crucial: we have used two basic heuristics to achieve this, which may be combined to form a third hybrid method. These are called "pre-analysis" and "bush pruning".

In the algorithms below we fill in the array p in the order given by π. However, in the C implementation we update the arrays *rel_X_to* and *rel_to_X* whenever π changes (in effect, changing the map between integers and elements of X). This improves efficiency by eliminating most of the accesses to π.

7.1 The "Pre-Analysis" Heuristic

The pre-analysis heuristic is a method for selecting a suitable permutation π. It involves iteratively extending a "good" partial permutation by adding an element and searching for elements which can be shifted to improve the permutation. "Shifting" an element in a permutation means composing π with a permutation $(i\ i{+}1\ \ldots\ j)$ where $i < j$ or composing with a permutation $(i\ i{-}1\ \ldots\ j)$ where $i > j$. This seems to have a better chance of improving the permutation than simply swapping two adjacent elements (composing with $(i\ i+1)$), or swapping two random elements (composing with $(i\ j)$).

We use two different definitions of a "better" partial permutation:

1. Pick the permutation which yields a smaller backtracking search tree (i.e. the number of elements examined by the *processall* procedure). If the two search trees are the same, then pick the permutation whose search tree has the smaller number of "leaf" elements (i.e. the number of full-size elements examined by *processall*);

2. Pick the permutation whose search tree has the smaller number of leaf elements; if they have the same number of leaf elements, pick the permutation whose total search tree is smaller.

We use a version of the basic backtracking algorithm to count the size of the search tree and number of leaf elements for the partial permutation $\pi[mindepth\,..\,maxdepth]$, where the elements $\pi[1\,..\,mindepth-1]$ have already been "frozen" (see below). This terminates immediately if the number of trials required for the calculation exceeds *cutoff* (each evaluation of $V'(t,p,\pi,n)$ is one "trial" since these evaluations dominate the whole computation):

```
proc calculate(mindepth, maxdepth, cutoff) ≡
  count := 0; trials := 0;
  var n := mindepth; t := 1:
    while (n > 0 ∨ t ⩽ D) ∧ trials ⩽ cutoff do
          if t > D then t := p[π[n]]; n := n − 1; t := t + 1
                   else trials := trials + 1;
                        if V'(t, p, π, n) then if n = maxdepth
                                                 then count := count + 1; t := t + 1
                                                  else d := d + 1; p[π[d]] := t; t := 1 fi
                                          else t := t + 1 fi od end.
```

We repeatedly test random "shifts" (where at least one of i or j must be within the partial permutation) to see if the partial permutation can be improved. After a certain number of failed attempts we assume that this is the best we can do for this size of partial permutation, so we increase the size by adding one element, and then attempt to improve this larger permutation. Note that in general, the "best" permutation of size $n+1$ is not a simple extension of the "best" permutation of size n. We have a "budget" which limits the number of times we want to evaluate the V' function (since this is the most expensive part of the algorithm). Once this budget has been used up we "freeze" the current partial permutation, and start building a new partial permutation with the remaining elements. Once all the elements have been used up we put together the "frozen" partial permutations to get a complete permutation which is used to attempt a full calculation. We have another budget for the full calculation and if this is exhausted before the calculation finishes then we halt the full calculation, double both analysis and calculation budgets and start again from scratch. The program prints messages as it proceeds (which may be captured in a log file) so that the user can monitor its progress.

Thus the pre-analysis routine works by increasing the size of the current partial permutation, stored in $\pi[mindepth\,..\,depth]$, by incrementing *depth*, and then adjusting the permutation to minimise the search tree. The subroutine *find_good_depth_element* tries inserting each of the elements $\pi[depth+1]$ to $\pi[SX]$ in position *depth* to find the best one. It updates global variables *count* and *trials* with the number of leaf nodes in the search tree for $\pi[mindepth\,..\,depth]$ and the total number of nodes in the tree. *find_good_insert* repeatedly picks a random pair of elements in π (at least one of which must be within $\pi[mindepth\,..\,depth]$), inserts one in place of the other, and tests if this improves the permutation. It terminates when it runs out of budget (the total number of trials allowed), or it has tried *maxgoes* insertions without improving the partial permutation. It sets the global variable *inserts_done* to the number of good insertions found.

```
mindepth := 1;
for depth := 1 to SX − 1 step 1 do
    do find_good_depth_element(mindepth, depth, maxtrials/10);
       best_result := count; best_trials := trials;
       if best_trials ⩾ maxtrials/10
         then print("Best next element exceeded:", maxtrials/10,
                    " trials at depth:", depth, mindepth);
              mindepth := depth;
              for i := 1 to mindepth step 1 do p[i] := 0 od
```

```
        else exit fi od;
    budget := maxtrials;
    printinfo(mindepth, depth, best_trials, best_result);
    if depth > mindepth
      then retries := 1
           do budget := maxtrials − best_trials;
              find_good_insert(mindepth, depth, best_trials, best_result);
              best_result := count; best_trials := trials;
              printinfo(mindepth, depth, best_trials, best_result);
              retries := retries + 1;
              if retries > maxretries ∨ inserts_done = 0 then exit fi od od
```

The assignments to $p[i]$ when *mindepth* is increased are to indicate to the *calculate* routine that relations involving these elements do not have to be preserved. The element "0" can be thought of as a new element, added to the set B, which is related to itself and everything else in every relation in R_Y.

The procedure *printinfo* prints a status report on the progress of the calculation, this includes the total CPU time used, and the CPU time used since the last status report.

Once this routine terminates (when $depth = SX - 1$) we use the calculate routine with a budget of *total_trials* (the total number of trials used by the pre-analysis). If this fails by exceeding its budget then we double *maxtrials* and *maxgoes* and run the pre-analysis again with this larger budget. This will hopefully result in a better permutation for the next full calculation, which in any case will have a larger budget to use. Thus our time will be divided roughly equally between pre-analysis and attempted calculations.

Note that this heuristic may take up to four times longer than necessary if an attempted calculation runs out of budget "just before" it would have completed. Also, it is not always easy to see from the status reports how much more time will be required to finish the calculation. A useful by-product of this method is a printout of the best permutation found.

7.2 The "Bush Pruning" Heuristic

The pre-analysis method does some initial analysis (for a given amount of time) to try and find a suitable permutation. Then it attempts a full calculation with that permutation. If the calculation fails (by running out of time before it has completed the search) then we double the analysis and calculation budgets and start again.

In contrast, the method described in this section starts the full calculation immediately, with an initial permutation which is updated "on the fly" as the calculation proceeds. This depends on the fact, noted above, that the elements $\pi[n+1]$ to $\pi[N]$ can be permuted at any time without affecting the final result.

The method is called bush pruning because it relies on minimising the size of the small tree[2] formed by adding a few elements to the current partial map. The simplest application of the method (which achieved some success) involves picking the element with the fewest valid images, each time an element is added to the current map (unless the current map is almost complete). Naturally, if an element is found which has no valid images, then the current partial sequence can be abandoned (the current bush has been pruned completely away!).

In the general case, the method divides its time between pushing forwards with the search and pruning the bush at the current position. Since it is best to prune equally frequently at all levels of the tree (apart from the last 1/4 of the tree levels which are not worth pruning) we use an array *next_bush*$[1\,..\,N]$ to record the "time" when each depth is next due to be pruned. This "time" is

[2] A bush is a small tree!

measured in terms of the total number of trials, i.e. evaluations of $V'(t, p, \pi, n)$.

The algorithm is based on the *calculate* algorithm with the bush pruning code added. The procedure *find_good_element*(n) picks the element to put in $\pi[n]$ which has the smallest number of relation-preserving images in Y. *find_bush_size*(n) sets *bush* to the size of a "suitable" bush for pruning: in other words, adding *bush* elements to the current partial permutation results in a search tree containing about *bush_budget*/*bush_goes* nodes. *prune_bush*$(n, bush)$ then uses up *bush_budget* trials in attempting to improve the part of the permutation between n and $n+bush$.

```
bush_trials := 0; count := 0;
n := 0; t := 1;
do if t ⩽ D
      then total_trials := total_trials + 1;
           if V'(t, p, π, n)
             then if n = SX then count := count + 1; t := t + 1;
                             else p[π[n]] := t; n := n + 1; t := 1;
                                  if n < 3N/4 ∧ total_trials > next_bush[n]
                                    then last_bush := bush_trials;
                                         find_bush_size(n);
                                         prune_bush(n, bush);
                                         update_next_bush() fi fi
                  else t := t + 1 fi
      else n := n − 1;
           if n = 0 then exit fi;
           t := p[π[n]] + 1 fi od
where
proc update_next_bush() ≡
  for i := n to n + bush step 1 do
      next_bush[i] :=total_trials + 5(bush_trials − last_bush) od.
```

Note that if *find_bush_size* extends the bush to the rest of the set then the current p element has been completely fathomed (all its valid extensions have been discovered). Also if *find_bush_size* or *prune_bush* ever reach a bush with no leaves (no valid extensions up to $n+bush$) then there can be no complete and valid extensions of the current p. In either case we can jump immediately to the step $n := n - 1$, and this is what the C implementation does. The C implementation also prints a regular status report (after each *bush_pr_step* trials).

7.3 A Hybrid Method

The bush pruning heuristic has to start with *some* permutation, if only a random one. The hybrid method starts by doing an initial pre-analysis to provide this initial permutation, and then switching to the bush pruning method. The "first part" of this permutation, up to the point where the pre-analysis first increased *mindepth*, is preserved from modification by bush pruning. This is achieved by setting *next_bush*$[i]$ for these elements to some suitably large value. This part of the domain will have a certain number of partial maps, we are to determine all the valid extensions of these maps. We therefore have a crude measure of progress through the calculation by looking at how many of these partial maps have been processed so far. This is only a crude measure since some of the partial maps may have many more valid extensions than others, but it can give some indication of feasibility: for example if the program has been left running over the weekend and has processed less than 1% of the set of partial maps, then it is likely to take many more days to run to completion.

7.4 Heuristics based on Knuth's Estimation Algorithm

In Knuth (1975), Knuth presents a method for estimating the size of the search tree of a simple backtracking algorithm. The method is based on making a number of random "probes" into the tree, picking a random path at each stage, and computing the weighted total of the cost of the calculation carried out at each node:

$$C = c() + d_0 c(x_1) + d_0 d_1 c(x_1, x_2) + d_0 d_1 d_2 c(x_1, x_2, x_3) + \dots$$

Here $c(x_1, \dots)$ is the cost of the computation at the node $p = \langle x_1, \dots \rangle$ (in our case these costs are all the same so we set them all to 1), d_0 is the number of initial elements x such that the sequence $\langle x \rangle$ is valid. One of these elements, x_1, is chosen at random. For each valid sequence $p = \langle x_1, \dots, x_i \rangle$, d_{i+1} is the number of elements x which can be added to p to get a valid sequence. One of these elements, x_{i+1}, is chosen at random. The procedure terminates when d_i is zero. C is the cost estimate from this probe.

Knuth provides two "proofs" (*'at least one of which should be convincing'*) that the expected value of C is the cost of the complete backtracking search. He says that the method has been tested on dozens of applications and has *'consistently performed amazingly well, even on problems which were intended to serve as bad examples. In virtually every case the right order of magnitude for the tree size was found after ten trials.'* He discusses one experiment in detail (a "knight's tour" problem) where averaging over 1,000 random walks produced an estimate within 0.5% of the actual answer of 3,137,317,290.

This algorithm appears to provide an ideal method for determining which of two permutations is better (and hence for finding a good permutation), which, unlike our previous methods, takes into account the whole permutation. Unfortunately, for most of our relation-preserving maps problems, the estimations appeared to be less accurate than we hoped: even averaging over 100,000 probes, and taking several minutes of CPU time on a Sparc 2, the estimates would vary by a factor of two or more, with some problems giving wildly inaccurate estimates.

Despite these discouraging results, we implemented a permutation-selection algorithm based on Knuth's estimation method. The algorithm tests various potential insertions, using Knuth's method to see if the permutation has improved. As soon as the permutation appears to provide a feasible search tree, the algorithm attempts a calculation. If this fails (by taking more than twice the estimated number of trials) we assume that by averaging over more random walks we would get a better estimate. We therefore increase the number of random walks (by say 20% to 50% at a time) until the estimate is greater than twice the old estimate (we know that the actual search tree size is at least this big). We define a "feasible" search tree to be one which is estimated to take less than a quarter of the total number of trials we have carried out in the analysis so far: this means that as the search for a good permutation takes more and more time, we will be steadily relaxing the feasibility requirements.

Unfortunately, this heuristic method was a dismal failure! The main problems are:

1. Occasional false underestimates (even with a large number of probes)—this causes it to think a particular random insertion is better when it is probably worse. So it takes a step *away* from optimality;
2. The large number of probes required means that only a small number of insertions can be tested. So it only takes a few steps *towards* optimality.
3. On the larger problems, rather than finding a "good" permutation, the algorithm merely finds permutations for which Knuth's method consistently underestimates the result. For example, every estimate might be around 10^8 to 10^9 while the actual tree is orders of magnitude greater than 10^9 nodes. The effect is that it keeps doing trial calculations which fail and cause the number of probes to be increased to such a degree that the program effectively "grinds to a halt". In one case, increasing the sample size from 5,000 to 7,500 caused the estimate to

change from a quite feasible 10^7 trials to a totally impractical 10^{20} trials. The algorithm then "improved" this permutation by finding a new one for which Knuth's method underestimates the tree size.

These problems are still present even when averaging over a very large number of probes, for example with 100,000 samples which takes several minutes of CPU time on a Sparc 2 to test one insertion. As a result, this method has been abandoned, though the code is available from the authors. Table 5 compares the bush pruning method with the method based on Knuth's algorithm for some of our smallest examples.

	No. of trials		CPU time	
Problem Size	Knuth	Hybrid	Knuth	Hybrid
42×17	28,550,038	789,604	533	16
64×2	67,119,212	37,248,521	2,722	1,048
65×2	49,438,402	23,438,477	2,223	599
64×8	2,898,059	231,928	102	94
153×33	unknown[3]	1,422,829	$> 44,000$	763

Table 5: Bush Pruning compared to the method based on Knuth's estimation algorithm. ("Problem size" refers to the size of the domain and range).

7.5 Practical Limits of the Methods

After testing the different methods on many relation-preserving maps examples of different types, our conclusion is that the hybrid method, with a budget of 10,000 for pre-analysis and a bush pruning budget of 100,000, is the best overall. The only practical limitation of the method is a necessary consequence of its generality: the algorithm is designed to process each relation-preserving map individually, and is therefore limited by the number of maps to be found, even when the "process" is a simple count. All our examples with up to 10^9 maps succeeded, our only failures were those problems known to have many more solutions (10^{11} or more maps). Such problems can only be solved by using much deeper knowledge of the particular lattice structures involved, in order to count the solutions in large "clumps" rather than one at a time. For example, Berman and Mukaidono (Berman & Mukaidono (1984)) used symmetry and clumping to calculate $|F\mathbf{K}(n)|$, the number of free Kleene algebras on n generators, for the case $n = 4$. The result of 160,297,985,276 is just outside the feasible range for our general-purpose algorithm. Our algorithm deals with the $n = 3$ case in just 11 seconds and 500,000 trials.

The source code for an optimised C implementation of all the algorithms, together with sample data files and parameter files, is available from the authors. M. Ward would be interested to hear from anyone who has other backtracking applications which could benefit from the element ordering heuristics.

8 Automating the Transformation Process

The program transformation theory used in this paper forms the foundation of the "Maintainer's Assistant" project (Bull (1990), Ward & Bennett (1993), Ward & Bennett (1995), Ward, Calliss & Munro (1989)) at Durham University and the Centre for Software Maintenance Ltd. which aims to produce tools to assist a maintenance programmer in understanding and reverse-engineering large software systems. The Maintainer's Assistant consists of an interactive structure editor and pretty-printer, implemented under X Windows, and a transformation engine, implemented in LISP and WSL. The transformation engine includes a library of over six hundred proven transformations,

[3]The program had not terminated after over 12 hours of CPU time.

including most of the ones used in the program derivations above. Once the remaining transformations have been implemented it will be possible to carry out the derivation interactively: starting from the formal specification and invoking a sequence of proven transformations and refinements, with the system checking all correctness conditions at each stage, finally translating the resulting (executable) WSL code into a suitable programming language, such as C. One interesting result we have noticed from our experiences with manual transformation is that the kinds of (clerical and logical) errors made in deriving an algorithm tend to be the sort of errors (for example writing $<$ instead of $>$) which are uncovered by the first few test cases. Once these errors have been corrected, the programs invariably pass all test cases with flying colours. This contrasts with typical programming bugs which tend to be subtle and extremely difficult to track down.

References

Arsac, J. (1982a): Transformation of Recursive Procedures. In: Neel, D. (ed.) Tools and Notations for Program Construction. Cambridge University Press, Cambridge, pp. 211–265

Arsac, J. (1982b): Syntactic Source to Source Program Transformations and Program Manipulation. Comm. ACM **22**, 1, pp. 43–54

Berman, J. & Mukaidono, M. (1984): Enumerating fuzzy switching functions and free Kleene algebras. Comput. Math. Appl. **10**, pp. 25–35

Bull, T. (1990): An Introduction to the WSL Program Transformer. Conference on Software Maintenance 26th–29th November 1990, San Diego

Butler, G. & Lam, C. W. H. (1985): A General Backtrack Algorithm for the Isomorphism Problem of Combinatorial Objects. J. Symb. Comput.

Davey, B. A. & Priestley, H. A. (1987): Generalised piggyback dualities, with applcations to Ockham algebras. Houston J. Math. **13**, pp. 151–198

Davey, B. A. & Priestley, H. A. (1990): Introduction to Lattices and Order. Cambridge University Press, Cambridge

Davey, B. A. & Priestley, H. A. (1992): Optimal dualities for varieties of Heyting algebras. preprint

Davey, B. A. & Priestley, H. A. (1993a): Partition-induced natural dualities for varieties of pseudocomplemented distributive lattices. Discrete Math. **113**, pp. 41–58

Davey, B. A. & Priestley, H. A. (1993b): Optimal natural dualities. Trans. Amer. Math. Soc. **338**, pp. 655–677

Davey, B. A. & Werner, H. (1983): Dualities and equivalences for varieties of algebras. In: Huhn, A. P. & Schmidt, E. T. (eds.) Contributions to lattice theory (Szeged, 1980). (Colloq. Math. Soc. János Bolyai no. 33.) North-Holland, Amsterdam, pp. 101–275

Dijkstra, E. W. (1976): A Discipline of Programming. Prentice-Hall, Englewood Cliffs, NJ

Ganter, B., Wille, R. & Wolff, K. (eds.) (1987): Beiträge zur Begriffsanalyse. B.I. Wissenschaftsverlag, Mannheim, Zürich

Gerhart, S. L. & Yelowitz, L. (1976): Control Structure Abstractions of the Backtracking Programming Technique. IEEE Trans. Software Eng. **SE 2**, 4, pp. 285–292

Hartung, G. (1992): A topological representation of lattices. Algebra Universalis **29**, pp. 273–299

Hoare, C. A. R., Hayes, I. J., Jifeng, H. E., Morgan, C. C., Roscoe, A. W., Sanders, J. W., Sørensen, I. H., Spivey, J. M. & Sufrin, B. A. (1987): Laws of Programming. Comm. ACM **30**, 8, pp. 672–686

Knuth, D. E. (1974): Structured Programming with the GOTO Statement. Comput. Surveys **6**, 4, pp. 261–301

Knuth, D. E. (1975): Estimating the Efficiency of Backtracking Algorithms. Math. of Comput. **29**, 129, pp. 121–136

Knuth, D. E. & Szwarcfiter, J. L. (1974): A Structured Program to Generate All Topological Sorting Arrangements. Inform. Process. Lett. **2**, pp. 153–157

Lee, K. B. (1970): Equational classes of distributive pseudo-complemented lattices. Canad. J. Math. **22**, pp. 881–891

Morgan, C. C. (1994): Programming from Specifications. Prentice-Hall, Englewood Cliffs, NJ. Second Edition

Priestley, H. A. (1992): Natural dualities for varieties of distributive lattices with a quantifier. Proceedings of the 38th Banach Centre Semester on Algebraic Logic and Computer Science Applications to appear

Rival, I., (ed.) (1982): Ordered Sets, Reidel, Dordrecht

Roever, W. P. de (1978): On Backtracking and Greatest Fixpoints. In: Neuhold, E. J. (ed.) Formal Description of Programming Constructs. North-Holland, Amsterdam, pp. 621–636

Stallman, R. M. (1989): Using and Porting GNU CC. Free Software Foundation, Inc.

Taylor, D. (1984): An Alternative to Current Looping Syntax. SIGPLAN Notices **19**, 12, pp. 48–53

Urquhart, A. (1977): A topological representation theory for lattices. Algebra Universalis **8**, pp. 45–58

Walker, R. J. (1960): An Enumerative Technique for a class of Combinatorial Problems. In: Bellman, R. E. & Hall Jr., M. (eds.) Proceedings of Symposia in Applied Mathematics 10: Combinatorial Analysis. Am. Math. Soc., Providence R.I.

Ward, M. (1989): Proving Program Refinements and Transformations. Oxford University, DPhil Thesis

Ward, M. (1990): Derivation of a Sorting Algorithm. Durham University, Technical Report. ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/sorting-t.ps.gz⟩

Ward, M. (1991a): Specifications and Programs in a Wide Spectrum Language. Submitted to J. Assoc. Comput. Mach.

Ward, M. (1991b): A Recursion Removal Theorem—Proof and Applications. Durham University, Technical Report. ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/rec-proof-t.ps.gz⟩

Ward, M. (1992): A Recursion Removal Theorem. Springer, New York Berlin Heidelberg. Proceedings of the 5th Refinement Workshop, London, 8th–11th January. ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/ref-ws-5.ps.gz⟩

Ward, M. (1994): Foundations for a Practical Theory of Program Refinement and Transformation. Durham University, Technical Report. ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/foundation2-t.ps.gz⟩

Ward, M. (1993): Abstracting a Specification from Code. J. Software Maintenance: Research and Practice **5**, 2, John Wiley & Sons, pp. 101–122. ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/prog-spec.ps.gz⟩

Ward, M. (1996): Derivation of Data Intensive Algorithms by Formal Transformation. IEEE Trans. Software Eng. **22**, 9, pp. 665–686. ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/sw-alg.ps.gz⟩

Ward, M. & Bennett, K. H. (1993): A Practical Program Transformation System For Reverse Engineering. Working Conference on Reverse Engineering, May 21–23, 1993, Baltimore MA. ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/icse.ps.gz⟩

Ward, M. & Bennett, K. H. (1995): Formal Methods for Legacy Systems. J. Software Maintenance: Research and Practice **7**, 3, John Wiley & Sons, pp. 203–219. ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/legacy-t.ps.gz⟩

Ward, M., Calliss, F. W. & Munro, M. (1989): The Maintainer's Assistant. Conference on Software Maintenance 16th–19th October 1989, Miami Florida. ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/MA-89.ps.gz⟩

Weidemann, D. A. (1991): A computation of the 8th Dedekind number. Order **8**, pp. 5–6

Wells, M. B. (1971): Elements of Combinatorial Computing. Pergamon Press, New York

Appendix: C Implementation of the Simple Backtracking Algorithm

This Appendix gives a C implementation of the simple backtracking algorithm (without element order selection). The source code and sample data files for all the algorithms is available from the authors.

Note that the *DO* "subroutine" (called *relpres* below) is copied in place: this is to avoid the subroutine call overhead for those compilers which (unlike the GNU C compiler gcc for example Stallman (1989)) cannot deal with "inline" subroutines. This version uses `ii` instead of i and uses an array `psi` for the sequence p. As noted above (Section 7) we update *rel_to_X* and *rel_X_to* whenever π is changed, this means that $p[\pi[i]]$ can be stored in `psi[i]`.

```
/*********************************************************************
 *
 * Calculate and return no. of relation-preserving maps
 * on the part of the domain between mindepth and maxdepth inclusive.
 * Assumes that the arrays rel_to_X and rel_X_to are set up.
 * Returns the no. of maps found.
 * Sets global variable trials to no. of calls of relpres required.
 * Terminate immediately if no. of trials exceeds cutoff.
 *
 *********************************************************************/

double
calculate (mindepth, maxdepth, cutoff)
     int mindepth, maxdepth;
     double cutoff;
{
  /* Calculate and return no. of relation-preserving maps:
   * use mindepth to maxdepth elements of domain,
   * terminate as soon as no. of trials exceeds cutoff
   */
  double count;
  int rho;
  register short rp, n, t, np;
  register int ii, numrels;

  trials = 0;
  count = 0;
  n = mindepth;
  t = 1;
  for (;;) {                    /* loop calc_do1: */
    if (t <= SY) {
      trials = trials + 1;
      if (trials > cutoff) {
        goto calc_od1;
      }
      /* Start of relpres subroutine:
       * if relpres(psi,t,n) then rp=1
       * relpres(psi,t,n) is eqt to psi[n]=t; relpres(psi,n)
       * t is the test value for psi[n], relpres(psi,n-1) is true */
      rp = 1;
      for (rho = 0; rho < R; rho++) {
        /* test relation rho: */
        /* Test elements relating to X */
        numrels = rel_to_X[rho][n][0];
        for (ii = 1; ii <= numrels; ii++) {
          np = rel_to_X[rho][n][ii];    /* np is the iith elt related to n */
```

```
          if (np < n) {         /* psi[np] is defined: */
            if (RY[rho][t][psi[np]] == 0) {
              rp = 0;
              goto calc_end_relpres;
            }
          } else {
            if (np == n) {      /* n is related to itself */
              if (RY[rho][t][t] == 0) { /* check t is related to itself */
                rp = 0;
                goto calc_end_relpres;
              } else {          /* np > n, ie no more elts related to n */
                goto calc_end_inner_1;
              }
            }                   /* fi (np == n) */
          }                     /* fi np < n */
        }                       /* End of inner for loop */
      calc_end_inner_1:
        /* Test elements X relates to */
        numrels = rel_X_to[rho][n][0];
        for (ii = 1; ii <= numrels; ii++) {
          np = rel_X_to[rho][n][ii];    /* np is the iith elt n relates to */
          if (np < n) {         /* psi[np] is defined: */
            if (RY[rho][psi[np]][t] == 0) {
              rp = 0;
              goto calc_end_relpres;
            }
          } else {              /* already checked n relates to n case */
            goto calc_end_inner_2;
          }                     /* fi np < n */
        }                       /* End of inner for loop */
      calc_end_inner_2:;
      }                         /* End outer for loop, next relation */
      /* End of relpres subroutine */
    calc_end_relpres:
      if (rp == 1) {
        if (n == maxdepth) {
          count++;
          t++;
        } else {
          psi[n] = t;
          n++;
          t = 1;
        }
      } else {
        t++;
      }
    } else {                    /* from (t <= SY) */
      n--;
      if (n < mindepth) {
        goto calc_od1;
      }
      t = psi[n] + 1;
    }                           /* fi from (t <= SY) */
  }
calc_od1:
  return (count);
}                               /* End of calculate(mindepth, maxdepth, cutoff) */
```