IMPROVEMENT OF CONSTRAINED SEARCHES

BY

MARKIAN MYRON GOOLEY

B.S., Millikin University, 1981
M.S., University of Illinois at Urbana-Champaign, 1985

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1991

Urbana, Illinois

ABSTRACT

Search is ubiquitous in computer science, but most searches are performed under constraints: time, memory, dependences. We study several cases of constrained search and present methods for improving their performance. Execution of Prolog programs amounts to a search with several types of dependences. We present a method for static reordering of goals in Prolog clauses, characterizing the degrees of equivalence between the original and reordered code, characterizing restrictions on reordering, presenting a heuristic for choosing amongst permissible reorderings, and showing how to restore equivalence that might be lost in the reordering process. Heuristic search in restricted memory is another constrained search. We present a new method called *speculative search* which is competitive with other searches used when memory is limited.

DEDICATION

To my father

Bonus est Dominus sperantibus in eum, animae quaerenti illum.

Bonum est praestolari cum silentio salutare Dei.

Bonum est viro, cum portaverit jugum ab adolescentia sua.

Sedebit solitarius, et tacebit: quia levavit super se.

Ponet in pulvere os suum, si forte sit spes.

ACKNOWLEDGEMENTS

I would like to thank my friends and colleagues who provided help, encouragement, sympathy, advice, and so on. In no particular order, and certainly with many omissions: Joe Rahmeh (now at the University of Texas), Carol Gura (now at IBM Austin), Rob Roy, Jeff Baxter, Randy Brouwer (now at Calvin College), Kent Fuchs, Ken Kubiak, Tom Fruchterman (now at Delfin Systems), Mark Marlett (now at Cypress Semiconductor). Special thanks to my long-suffering office mate Pankaj Mehra, and of course to my family.

My thesis advisor, Prof. Benjamin Wah, introduced me to the fields of research presented here, provided guidance, and supported this work in a multitude of ways.

The National Aeronautics and Space Administration supported this work under a NASA Graduate Fellowship and grants NAG 1-613 and NCC 2-481. The National Science Foundation supported it under grant MIP88-10584.

TABLE OF CONTENTS

CHAPTER 1.

INTRODUCTION

1.1. Motivation and summary

Search is ubiquitous in computer science and computer applications. A multitude of problems can be formulated as searches, and for many of these a search is the method of choice. Artificial intelligence, combinatorial optimizations, databases, and many other disciplines use searches to solve both practical and theoretical problems.

Searches as actually implemented and performed usually suffer from constraints. There may be a limited amount of time in which to complete a search. The computer being used may not have enough memory to support a particular method of search. If the progress of the search is part of the solution sought, the order of the search may be constrained. Dependences between nodes of a search may similarly constrain order.

In this thesis we present our work on improving the behavior of several types of constrained search. In this chapter we present a classification of searches and similar problems, and show how various searches and algorithms fit into it, particularly the subjects of our research. Then we present an overview of our work, and summarize the contributions it makes. In Chapter 2 we characterize restrictions on and demonstrate improvement of a search with dependences, specifically for the logic programming language Prolog [9,49] by means of reordering statically the goals of its clauses. In Chapter 3 we present a new method for heuristic search under limited memory which generalizes and improves on several earlier methods. Finally we present conclusions and ideas for future work.

1.2. A classification of search problems

We view a search as a traversal of a tree or graph, usually generated implicitly by the search process. Such a traversal proceeds according to particular parameters: rules that control it or restric-

tions on the way it can be performed. Different combinations of these parameters yield different types of search, suitable to different problems. We discuss the parameters and then present our classification based on different combinations of their values.

1.2.1. Selection

Consider that graph or tree of search nodes generated by the process of searching. At any given point in the search, what node should we *select* to expand next? We divide strategies for selection into *static* and *dynamic*.

A static strategy is determined before the search is performed. Typically it is some simple rule for choosing the next node, such as the textual (depth-first, top-down, left-to-right) order of expansion of Prolog. Another example is a blind (uninformed) depth-first search that examines child nodes in an arbitrary fixed order and searches to some given fixed depth. An iteration (stage) of depth-first iterative deepening [26] is such a search, except that the choice of depth limit could be considered dynamic.

A dynamic strategy depends on information determined during the search: for example, the value of a lower- and perhaps an upper-bound heuristic for nodes of the search space. The amount of dynamic information used by the search can vary greatly. A best-first search maintains a queue of nodes not yet expanded, ordered by their lower-bound values, so that potentially all untried nodes are candidates for selection; a guided (informed) depth-first search uses such information to decide which of the child-nodes atop its stack should be tried next, and an IDA^* search [26] uses it only to decide how deep it should search in its current iteration.

1.2.2. Guidance

Guidance is the information that controls the search, particularly the selection of nodes. It can be either *exact* or *inexact*.

Exact guidance means that we know exactly what node to expand next to advance towards our goal in the search space (for example, an optimal solution to a problem, an answer that satisfies a set of conditions, etc.). Note that it can be static or dynamic, the latter depending on the data available during the search, but making "ideal" decisions based on that information. Exact guidance usually makes a search degenerate to an exact algorithm for solving the problem in question.

Inexact guidance depends on some sort of heuristic. (We can consider the guidance of a blind search to be a degenerate heuristic that provides no information whatever.) A heuristic applied statically indicates in some fashion the order in which particular groups of nodes are to be searched; the search tree must be explicitly available, at least in part, before the actual search takes place. Any nontrivial static heuristic must label or create or alter some sort of representation, at least a partial one, of the search space: if the search graph does not exist in any form before the search, obviously there is no way to apply a static heuristic. More usually, heuristic guidance of a search is dynamic. We expand nodes of the search, evaluate heuristics for each new node and assign a value to it, and decide what node to expand next based on how good a value nodes have.

1.2.3. Dependences

Sometimes nodes need to be expanded in a particular order. In some way one has a *dependence* on another. Some nodes have side-effects that may not be directly related to the search itself, but need to be expressed in order. In other cases, a node cannot be expanded without information obtained by expanding another node first; a simple form of this is usually true for a child node and its parent, but we are concerned with the problem in pairs of nodes that are not ancestor and descendant.

Many searches have no dependences between nodes except those between ancestor and descendant. Typically for the sort of branch-and-bound heuristic search used to solve problems in NP there are none, but sometimes unusual heuristics or odd circumstances create them. For instance, some methods for solving the asymmetric traveling-salesman problem [1] require, for an N-city problem, an $N \times N$ matrix in each node of the search tree. For large N, the only way that this can possibly fit into the main memory of most machines is to keep a single matrix and modify it when moving from node to node: successive nodes of the search must be contiguous.

1.2.4. A classification

We can classify searches and related problems based on the three parameters just discussed. We can construct a binary tree such as the one in Figure 1.1, the numbered leaves indicating different classes of search problems.

Classes 0, 1, 4, and 5 contain searches with exact guidance. These are typically performed by algorithms meant for specific problems. Sometimes a better algorithm can be devised for a problem, and we can use it instead. Essentially these are degenerate searches, guaranteed to succeed or fail in some number of operations. For example, a fragment of code to add up an array of integers does a class 0 "search" (the lack of dependences would allow several additions to proceed in parallel), a loop to print the values of an array in order a class 1, a sorting algorithm a class 4, and a sorting algorithm that must be stable a class 5. Arguably only optimal algorithms belong in the "exact" classes.

All of the other classes of search have inexact guidance which is prone to improvement. Searches of relational databases fall into class 2, and Prolog execution, rich in side-effects and other types of dependence, into class 3. Improvement of these classes involves determining statically some

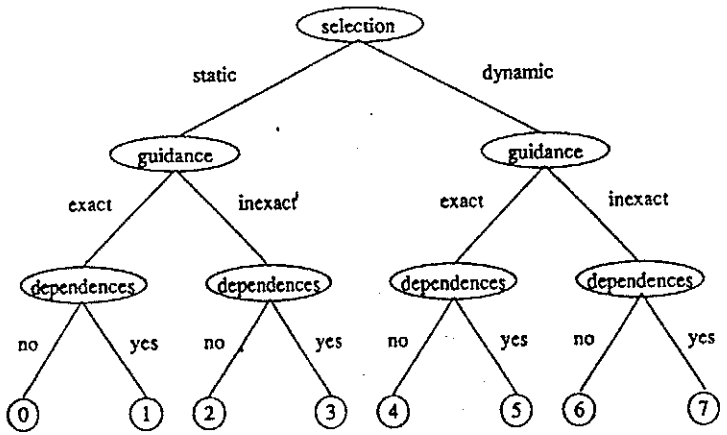

Figure 1.1: Classes of search problems

sort of heuristic information to guide the search, and altering the explicit search tree embodied by the database entries or the Prolog code so that the fixed search strategy works under better guidance. Hence our work on Prolog consists of improvements for class 3 searches.

Heuristic searches, as used in a multitude of applications, fall into classes 6 and 7. As we mentioned above, there can be dependences in this variety of search, but they are atypical: class 6, the searches without dependences, is more usual. (Though arguably some of our work on Prolog, involving run-time tests as it does, can be considered an effort to make Prolog into a class 7 search by adding some dynamic selection.) We can add an extra level to the tree of Figure 1.1 for *constraints* on resources such as time and memory, which are a real concern in solving practical problems. Our work on heuristic search in restricted memory is in this subset of class 6.

1.2.5. Rationale for our work

We seek to improve searches that must be performed under constraints. Prolog is a particularly good subject for this, due to the many dependences between different components of a program, the reliance of control on the default blind search mechanism, and the side-effects produced by so many of its built-in predicates. Prolog is not, however, useful for studying search under restricted memory: its unguided depth-first strategy does not have that problem. One could design a logic language that executes by performing a best-first or breadth-fist search [28], and then try to approximate such strategies in reduced memory, but the control structures of such a language would have to be quite different from Prolog's. Our work on Prolog would probably not be applicable. Instead we study memory constraints on a simpler type of search, avoiding the severe restrictions that a full-featured logic language would impose. We believe that our methods for search in restricted memory would be useful for logic languages that employ search strategies more complex than Prolog's, but we do not address the possibility further in this thesis.

1.3. Reordering of goals in Prolog programs

Prolog is the best known and most widely used implementation of logic programming. It has been studied and used extensively, especially outside of the United States, and it was chosen as the principal language for the Fifth Generation Computer Systems project in Japan. Both Prolog and related parallel logic languages have been somewhat disappointing in practice, principally because programs written in them tend to execute slowly.

One reason for slow execution of Prolog code is inefficiency due to needless work. For ease of implementation and reasonable use of computer memory, Prolog evaluates a program in textual order, left-to-right and depth-first, corresponding to such a traversal of the AND/OR graph [29] determined by the program. Execution may traverse several large subgraphs, each corresponding to a goal, only to fail on a later one. On failure, Prolog backtracks, searching the subgraphs again to find another way to satisfy its conjunction of goals.

We can try to avoid futile work by altering Prolog's mechanism of execution: Prolog is just one possible instance of logic programming based on resolution, and its simple method of search is not sacrosanct. However, anything more complicated can use much more memory, may be more difficult to understand, and is better suited for an entirely new logic language. We believe that the execution of Prolog is best left alone, which leaves us with transformations on programs, specifically reordering of goals as our best hope for improvement.

Experienced Prolog programmers know the importance of order of goals. When they can, they write clauses so that inexpensive goals that are likely to fail, such as tests, are near the beginning [49]: if a test fails early on, the rest of a clause is not evaluated. However, the best order for goals or clauses is often problematic, even for small programs; in some cases it will be different for different calls to a predicate [36]. Also, programmers tend to write code in orders that are easy to understand, especially if the constituent goals or clauses seem to have about the same cost and chance of success. For large, complex predicates, they may well sacrifice efficiency for clarity, giving us another incentive to automate the process.

Here is an example that shows some benefits of reordering goals. Express a family tree in Prolog. Three predicates give the relationships: `wife/2`, `mother/2`, and `female/1`. `wife(john,jane).` means that John and Jane are married. `mother(john,joan).` means that Joan is John's mother, and `female(jan).` means that Jan is female. Assuming no illegitimacy, divorce, or remarriage,

```
female(Woman):- wife(_,Woman).
```

accounts for married women (the underscore is the don't-care *anonymous variable*), and facts of the form `female(jan).` account for single ones.

Assume a large collection of such facts, and these predicates:

```
grandmother(GC,GM):-
      grandparent(GC,GM),
      female(GM).

grandparent(GC,GP):-
      parent(P,GP),
      parent(GC,P).

parent(C,P):-
      mother(C,P).
parent(C,P):-
      mother(C,M),
      wife(P,M).
```

The query

```
:- grandmother(X,Y).
```

finds all grandmother-grandchild pairs; however, it finds a grandparent-grandchild pair first, and rejects it promptly about half the time. Changing the first clause to

```
grandmother(GC,GM):-
      female(GM),
      grandparent(GC,GM).
```

should reduce the cost: `female/1` takes at most two calls, whereas `grandparent/2` can take far more. Note also that `female/1` sets a variable of the call to `grandparent/2`, letting it consider fewer possibilities. Unless only a tiny fraction of the females in the facts are grandmothers, the

reordering pays.

There is some formal justification for choosing these new orders for goals. Consider again a Prolog program as an AND/OR graph, with predicates as OR-nodes and clauses as AND-nodes. Nodes of one type have children of the other. Suppose that for a depth-first search (Prolog execution) the children i of a node (the clauses of a predicate or the goals of a clause), succeed with independent probabilities p_i (and hence fail with probabilities $q_i = 1 - p_i$), and cost c_i to execute. Li and Wah [29] have proven that in such cases, ordering the children of an AND-node (goals) by decreasing $\frac{q_i}{c_i}$, and the children of an OR-node (clauses) by decreasing $\frac{p_i}{c_i}$, minimizes the expected cost of a depth-first search. Of course, the costs and success-probabilities of clauses and goals in Prolog are not independent of each other: in fact, they vary greatly with order. Still, this result shows some added justification for the intuitive ordering.

Though we have just mentioned the possibility of reordering of clauses within predicates, we do not consider it in detail in this thesis because it does not appear to be especially useful. If we need only a single answer to a predicate call, it might pay to put first the clause that we expect will cost the least to evaluate; if we need to examine all answers, as is more usual, we must examine all matching clauses anyhow and reordering profits us nothing.

In practice, reordering clauses of a predicate has several drawbacks. Usually the least-expensive clause handles a special case, such as the base of a recursion or an unusual value of an argument: putting it first may greatly change the meaning of a program (given certain control constructs). Also note that any good implementation of Prolog performs *clause indexing* [56]. That is, when a predicate is called, the Prolog engine examines at least one argument (generally the first) of the calling goal to see which of the clauses it cannot match. Indexing on even one argument usually rules out most clauses of the called predicate, often leaving only one or two that might match.

We present our work on Prolog in Chapter 2. To begin, we define and explain terms and concepts, then proceed to define levels of equivalence for Prolog under reordering: reordered programs

behave somewhat differently from their originals. Next we characterize the restrictions themselves. Most arise from "impure" features of Prolog not related to logic but useful for making Prolog a practical programming language; others are from recursion and the logical variable.

Static analysis of a program can ease some of the restrictions: any extra knowledge of how a program will behave can permit new orders. We review mode inference and similar techniques for finding more about a program before it runs. Then we present an extended version of the Markov-chain reordering method of Li and Wah [29], describe how it can be implemented in practice, and present some results of reorderings.

For some applications, the reordered results produced by reordered code are unacceptable. We have devised *order tracking*, which lets us reorder goals as usual without having them return permuted results.

1.4. Speculative search

Suppose that we have a problem in artificial intelligence, combinatorial optimization or the like, of the sort typically solved by using a branch-and-bound heuristic search, or a best-first specialization such as A^* [2]. Suppose further that we seek an optimal solution, that our lower-bound heuristic is not tight, and our upper-bound heuristic is weak or absent.

In such a case, a search is prone to expand a number of nodes exponential in the size of the problem [39]. Using a best-first search with an admissible heuristic function will minimize this number, but a best-first search must maintain a queue of nodes which can itself grow exponentially large. For problems of more than trivial size, this quickly exhausts even the secondary memory of any computer.

The obvious solution: use a depth-first search instead. These need only a stack of nodes and hence memory linear in the depth of the search. but they can expand many more nodes than a best-first search, particularly if (as we have assumed) the upper-bound function is weak. They can search far deeper in the search tree than where an optimal solution might lie, doing much futile work. Also,

unless the nodes of a search are especially large as for, say, large cases of the asymmetric Traveling Salesman Problem [1] depth-first search uses only a fraction of the primary memory of a computer. *Can we use the extra memory to reduce the number of nodes expanded in a search?*

We present this work in Chapter 3. First we look at several search methods for use in restricted memory. We then describe our *speculative search* method, and examine its behavior under a simple model of search, comparing it with the earlier methods. We present and discuss some problems common to such search methods. We present a version that prunes away nodes when it runs out of memory, and a version for parallel machines. Finally we present some results and conclusions.

1.5. Contributions of our work

To summarize:

(1) We use Prolog as the subject of our study of improving search under constraints of dependence, restrictions on the order of search, and the like. We show how to reorder goals in Prolog programs: show what degrees of equivalence reordering will preserve, characterize fully the restrictions on reordering, show what added information can make more orders possible, present a heuristic for choosing amongst permitted orders, and show how to put into their original order the results produced by a reordered program.

(2) We use heuristic search as the subject of our study of improving search under the constraint of memory. We present a new method of search and show that it is competitive with similar methods for search in restricted memory.

CHAPTER 2.

REORDERING OF GOALS IN PROLOG PROGRAMS

2.1. Introduction

By reordering the goals in the bodies of clauses, we hope to reduce the cost of executing Prolog code. As we mentioned in the introduction to this thesis, some of the inefficiency in Prolog comes from evaluating the expensive goals of a clause early on, only to have the clause fail on inexpensive ones towards its end.

We begin this chapter with an overview of Prolog. In the remainder of this introductory section we define terms and present concepts that we will need later, review Prolog's syntax, and give a sketch of how Prolog programs execute.

Equivalence of programs under reordering is a problem, because reordered code generally does not behave in the same way as the original code. We submit that for some applications, partial equivalence suffices. In Section 2.2 we define a series of levels or degrees of equivalence that we can ensure will be satisfied given the proper restrictions on reordering of code.

Numerous restrictions on reordering are essential to maintaining even partial equivalence. In Section 2.3 we present these and show why they are necessary, describing what happens if we neglect to enforce them and how in some special cases we can ignore them. Then in Section 2.4 we show that although we need little information about a program in order to reorder its clauses safely, we can create many opportunities for reordering by doing some static analysis, particularly mode inference, before we try to reorder it.

Section 2.5 presents our improvements on the Markov-chain heuristic for reordering of goals. We decribe the method, give details of how to implement it, and present results from applying it to various programs.

If reordered results from reordered goals are unacceptable, we can in some cases produce results in the original order, albeit with some overhead, using a method we call *order tracking*, which we present in Section 2.6. We tag the disjunctions descendent from a predicate, and return combined tags with the results of a call. If we collect all the results, sort them by tags into the proper order, and return them one by one to emulate backtracking, the reordered predicate looks just like the original. This can be done by altering the Prolog code or by modifying the compiler and execution engine, and we present details on both approaches. Depending on the improvement due to reordering and the overhead in speed of the added code or engine mechanisms, order tracking can be practical in some cases.

2.1.1. On Prolog

Here we present a brief overview of Prolog, emphasizing features germane to our work. We review essential definitions and ideas, then describe Prolog's syntax and mechanism of execution.

2.1.1.1. Background

Before we can explain the syntax and execution of Prolog programs, we need some definitions and descriptions, perhaps familiar from first-order logic, automated theorem-proving, and other subjects related to Prolog. A *term* is either a *variable*, or a constant symbol (*functor*) associated with a tuple of a particular number (*arity*) of arguments, each of which is another term. A functor with no arguments (an arity of zero) is a *constant*. A variable can be either *uninstantiated* (*free*) — set to no value — or *instantiated* (*bound*) to a non-variable term. A term not itself a variable, but containing free variables in its arguments, is *partially instantiated* (*PI*). Two variables can also be *coupled* together, or *aliased*: neither bound to a non-variable, but rather made identical to each other so that any future binding to either binds both.

A *(finite) substitution* is a finite set of pairs, each representing the binding of a variable to a term, with the variables all distinct; for example, $\{X=f(a,b),Y=b(A),Z=r\}$. If we apply a substitution θ to a term T, usually written $T\theta$, we bind any free variables of T mentioned in the substitution: if T is $t(X,g(Y))$, then $T\theta$ is (rather untidily) $t(f(a,b),g(b(A)))$. A term A is an *instance* of a term B if there is a substitution θ such that $A=B\theta$, and B is *more general* than A if A is an instance of B but not vice-versa. A and B have a *common instance* C if there exist substitutions θ_1 and θ_2 such that $C=A\theta_1=B\theta_2$. Two terms are *(alphabetic) variants* if each is an instance of the other; they are equivalent up to renaming of variables. For example, $f(A,b(c,D))$ and $f(X,b(c,Y))$ are variants.

If there is a substitution that renders two terms identical, they *unify*, and the substitution is their *unifier*. Applying the unifier generates a unique common instance of the terms. For most uses of unification, including theorem-proving systems and Prolog, the *most general unifier*, or *mgu*, is what's wanted. An unifier θ is an mgu if for every unifier σ of a set of terms, there is a substitution γ such that $\sigma=\theta\gamma$.

2.1.1.2. Essentials of the language

Prolog is based on the subset of classical first-order logic in which all the sentences are universally-quantified Horn clauses; to this have been added a particular strategy of execution and various features to provide control and make a useful programming language.

Terms are the building blocks of Prolog syntax. A term that is not a variable is called an *atom*, and a *literal* is either an atom or the negation of an atom (called a *negative literal*). However, Prolog does not allow the programmer to write explicitly negated literals.

A *Horn clause* is the universal closure of a disjunction of literals with at most one positive literal in it. Equivalently, it is a conjunction of premises (positive literals) that implies a single conclusion (another positive literal), with all free variables universally quantified (universal closure of the sentence, indicated by $(\forall^*)$):

$$(\forall^*)(a \leftarrow a_1 \wedge a_2 \wedge \cdots \wedge a_m)$$

For Prolog we assume the presence of the quantifiers and accept the limitations of the usual character set. A corresponding clause in Prolog, the variables shown, might be:

```
a(Xa,Y,Z1) :- a_1(Xa,Z1,_Z2), a_2(Y,_), a_3(_Z2), a_m(Y,Z1).
```

where symbols starting with uppercase letters or the underscore stand for variables.

The conjunction of premises is the *body* of the clause; the single literal of the conclusion is the *head*. An empty body is taken as *true*, and a clause with one is called a *fact*; an empty head, as *false*, the headless clause acting as a *query* or *goal*. (Literals of a clause's body are often called "goals" as well, and we typically use that name instead of literal, and call a headless clause a query.) A query of the form

$$\leftarrow a_1, a_2, \ldots, a_m$$

with free variables X_1 through X_n corresponds to

$$\neg(\exists X_1)\cdots(\exists X_n)(a_1 \wedge a_2 \wedge \cdots \wedge a_m).$$

This is the negation of what we want inferred from the program. An we shall see, for Prolog we in effect state that something is not true and challenge the system to find counterexamples.

Clauses with both head and body are *rules*. Facts and rules are *program clauses* because they form a program in Prolog, while queries presented to a Prolog system initiate execution of code in a program. Clauses with the same name and arity form a *predicate*; we refer to a predicate of name `name` and arity `n` as `name/n`. We write facts in Prolog as literals followed by the "`.`" indicating the end of a clause, and queries with a leading "is implied by" sign; for example:

```
% A fact: [this line is a comment]
a(this,that).

% A query with two literals:
:- a(First,Second), b(Second).
```

The Prolog fact in the example above states that the relation `a/2` holds unconditionally for its arguments. The rule

```
c(X,Y,Z):- a(X,Y), b(Z), d(Y,Z).
```

states that one way of finding whether the relation `c/3` holds for `X`, `Y`, and `Z` is to show that the relations in the clause's body hold.

Prolog has *lists* analogous to those in Lisp; there is a binary functor written as `.` and an empty-list constant written as `[]`. A list that is actually `.(a,.(b,.(c,[])))` can be written `[a,b,c]` instead. Usually a list is an inefficient data structure in Prolog; lists of known or limited size should be eschewed in favor of functors of many arguments [38].

2.1.1.3. Execution of programs

Given a program and a query, a Prolog system tries to refute the combination, seeking counterexamples. Each of these, for there can be many, is an substitution on the variables in the query.

Prolog executes a query to a program by repeatedly applying a restricted form of the linear resolution rule, *SLD-resolution* [49]. Prolog selects the leftmost literal of the query and search the program for a predicate of the same name and arity. It looks at the first clause of that predicate and tries to unify the literal with its head. If the unification succeeds, it replaces the chosen literal with the body of the matching clause, renaming variables as necessary to avoid falsely considering them identical: all variables are local to a clause. Prolog *resolves* the goal with the clause. The altered query consists of the literals from the matching clause (if any), followed by the remaining original literals. It is as if the Prolog system had done a transformation like the following:

```
% A query and a matching clause:
:- a(f(X),Y), b(Y,Z).
a(X,g(Y)):- c(X,Z), d(Y,Z).
% Afterwards, a "new" query (note renamed variables):
:- c(f(X),Z2), d(Y2,Z2), b(g(Y2),Z).
```

Should the leading goal fail to unify, Prolog *backtracks*. First we try, in textual order, any remaining clauses of the matching predicate. If no clause has a head that unifies with the leading goal, then the query as it stands can't be satisfied. Prolog undoes the most recent choice of clause, the one that produced the current form of the query, returning the query to its previous state. Then it tries any remaining clauses of the same predicate.

The process repeats: Prolog selects the first goal of the current query, resolves it with a matching clause if possible, backtrack to the previous form of the query if not. A query can cause infinite recursion, but typically the query either fails or Prolog reduces it to the empty clause, yielding a substitution (a counterexample that refutes the negated sentence presented by the query). After Prolog derives the empty clause, the user can force backtracking and look for an alternative refutation and substitution, repeating this until an eventual failure.

2.1.1.4. Other features of Prolog

Many of the built-in predicates that help make Prolog a useful programming language also cause difficulties in reordering, and we will look at many of these predicates and their associated difficulties presently. Here is a brief overview of the different varieties of built-in; usually we present further details as necessary in later sections.

Input/output predicates convert between substitutions and records in files (we can consider interactive input and output to be files as well). Predicates such as `read/1` and `write/1` read and write terms; others handle character I/O and so on. There are predicates to manipulate files. Many of the I/O predicates vary with dialect.

Various *control* predicates modify program execution. There is an explicit disjunction, written as a semicolon; Prolog treats its arguments as if they were bodies of successive clauses, trying first one and then the next. Sometimes it can save repeated work:

```
% Two clauses with common parts:
a(X,Y):- b(X,Z), q(Z,Y,W), r(W).
a(X,Y):- b(X,Z), c(Y,Z), d(X,Y).

% Perhaps b/2 will be repeated less often:
a(X,Y):- b(X,Z), (q(Z,Y,W), r(W); c(Y,Z), d(X,Y)).
```

The notorious *cut*, written `!`, provides many of the headaches associated with reordering. When Prolog encounters a cut during forward execution, it proceeds, treating it as vacuously true. But backtracking over a cut is forbidden, and when backtracking encounters a cut, the entire predicate call fails. Once we reach a cut in a clause of the predicate, either that clause produces a single solution or the predicate will fail. Prolog's implication predicate `->` behaves like a cut but affects only the clause containing it, and Prolog's version of negation is based on the cut as well.

Assorted *meta-logical* predicates test whether their arguments are of a particular type (uninstantiated variable, atom, number, etc.). Some test whether terms are in *standard order*: a sort of extended lexicographic order in which variables come first, then atoms, then partly-instantiated terms, then fully-instantiated terms. There are also predicates for manipulating structures built of terms, and *all-solutions predicates* for collecting all possible solutions to a goal.

2.1.2. Assumptions

We gloss or ignore some features found in most dialects of Prolog. We avoid details of the many different varieties of input/output predicates, which for our purposes are essentially identical. We assume that predicates that change the program dynamically, such as `asserta/1`, `assertz/1`, and `retract/2`, do not appear in the programs we transform. We also forbid `clause/3`, which reads clauses within a program itself, and both variable goals and the equivalent `call/1` with a variable argument.

2.2. Equivalence under reordering

Reordering a Prolog program can change its behavior. We are forcing Prolog to search a different program graph, mimicking a change in its rigid top-down, left-to-right execution so that it might behave more efficiently, yet clearly we want to keep these changes in behavior small and predictable, so that the transformed program is still useful. If Prolog code has no functional expressions (*i.e.*, Datalog [30]) and no built-in predicates that change control or have side-effects, almost any reordering of goals will cause at most a reordering of the responses to a query, just as when one reorders a query to a relational or a simple deductive database. For full Prolog we must limit the types of new orders we allow. Having fewer restrictions can cause greater changes in program behavior, but allows more possibilities for reordering.

2.2.1. Introduction

We know of no previous discussion of equivalence of Prolog programs under reordering. Papers on other source-to-source transformations for Prolog, such as those on fold/unfold transformations [23, 51] and partial evaluation [5, 25] generally include arguments, of various degrees of formality and for various subsets of Prolog, that their methods preserve program behavior.

We call the version of a piece of code written by its programmer the *original*, and the reordered version a *transform*. "Equivalence" of original and transform, for our purposes, means that given certain inputs they produce outputs that are similar enough for practical use. A program receives input from zero or more files, and produces output on zero or more others. (Files include interactive input and output; for our purposes we consider only a single input and a single output stream.) Note that serious Prolog code does not usually return results via a variable uninstantiated in the top-level query that begins program execution. It uses output predicates to write its results to files.

When a predicate is called, it receives a substitution and, with backtracking, produces some sequence (perhaps empty) of substitutions in return. Note that reordering the goals of the predicate's clauses or their descendants can at least permute the sequence. Also note that it is possible for a transform to respond to inputs in a manner just like its original's, even though some of the transformed predicates within it have changed their own behavior; however, preserving the equivalence of individual predicates of a program (or descendants of a predicate) is a reasonable if perhaps restrictive way to ensure that it will behave as desired.

2.2.2. Levels of equivalence

We submit that for certain applications, a transform need not behave exactly like its original: it has only to preserve a particular *level* or *degree* of partial equivalence. That is, we let a predicate or program produce a changed output in response to a given input; in exchange for this laxity, we can alter a program so that it should execute more quickly and efficiently.

We define five levels of equivalence, in order of decreasing similitude: *exact (reflexive) equivalence*, *set equivalence*, *structure inequivalence*, *tree equivalence*, and *inequivalence*. We base these largely on what happens as one removes various restrictions on reordering: the more restrictions one removes, the less the transform behaves like its original.

2.2.2.1. Exact equivalence

The transform and its original look identical from the outside: a given input produces a corresponding output or set of outputs. Thus if a program and its transform are exactly equivalent, a given query and a given stream of inputs produce a given stream of outputs. If a predicate and its transform are exactly equivalent, a given substitution presented to each elicits (under backtracking, if necessary) the same sequence of substitutions as output. This is the level of equivalence customarily demanded of systems that do source-to-source transformations.

2.2.2.2. Set equivalence

For predicates at this level of equivalence, a transform and its original produce the same *set* and number of output substitutions in response to a given input substitution, but not necessarily the same *sequence*. This is the kind of equivalence often acceptable for database queries: order is not important, only the set of items that match the query. We assume that this level is acceptable for many applications, and the restrictions on reordering that we present will preserve it unless we note otherwise.

At the program level, things are somewhat more complicated. Given an input file, the original program and a set-equivalent transform express all side-effects in the same order. However, the substituions that a built-in predicate is to convert into output might be fed to it in a permuted order, permuting thereby the output of the transform. Note that output independent of substitutions (for example, produced by output predicates called with constant arguments) is unaltered by reordering. All this behavior arises from a particular set of restrictions on reordering, as described later.

2.2.2.3. Structure inequivalence

At this level, the transform can produce output or behavior radically different from the original's. Structure inequivalence differs from set-equivalence only by the removal of one restriction on reordering. We allow certain reorderings not permitted for set-equivalent programs. As a result, we may alter some data structures: partially instantiated (PI) structures, and lists built by some of the all-solutions predicates. The original builds up a structure piece by piece, but the transform adds pieces in a different order, so that when it executes it may build a structure different from that built by the original. If the program tests such structures for particular properties (for example, depth of a tree, order of elements), then tests that once failed may succeed, and vice versa. For certain uses or certain predicates, such as those maintaining "dictionaries" [49] (binary trees for storage and lookup of data), this is unimportant, and a transformed predicate may, to callers, appear set-equivalent in use even though it is not.

2.2.2.4. Tree equivalence

As with the previous level, transforms at this level can behave radically different from their originals. Equivalence of transforms drops to this level if we ignore some restrictions regarding cuts. A cut commits an execution to the first element of the sequence of substitutions that would have been produced by backtracking; permuting of goals might permute a sequence so that a different element comes first and gets chosen by the cut. Thus the search tree defined by the transform has branches in a permuted order, so that different branches are pruned off by the cut than in the original. This level of equivalence is perhaps not widely useful, but if we have a predicate, preferably with no side-effects, from which any single answer will suffice, a tree-equivalent transform may be useful.

2.2.2.5. Inequivalence

We treat any further removal of restrictions as allowing transforms to become so different from their originals that they are useless: their behavior is too unpredictable to make up for any increased efficiency brought on by reordering.

2.3. Restrictions on reordering of goals

We cannot reorder carelessly. In all but the simplest programs written in pure Prolog, unconstrained reordering is risky: we cannot guarantee any sort of equivalence, nor freedom from infinite recursion. Certain built-in predicates have side-effects that cannot be undone by backtracking; others provide control. Neither can be treated lightly.

There is little previous work related to restrictions on reordering Prolog. O'Keefe, in a paper [37] that anticipated some of our work on cuts, also mentioned conditions under which explicit unification can be moved before goals, but the transformation does not appear especially useful. DeGroot [15, 16] independently noted the problems due to predicates with side-effects (especially how a single side-effect can cause difficulties that propagate to a large part of a program), but with regard to restricted AND-parallelism (RAP) [14], not reordering.

2.3.1. Predicates with side-effects: fixity

Many special predicates built into Prolog have *side-effects*: they perform actions unrelated to the logic of the program, actions not reversed by backtracking. We are especially concerned with predicates that do input/output (I/O). (Modification of the program as done by such predicates as `asserta/1`, `retract/2`, and `abolish/2`, which we are not considering, is definitely another side-effect.) I/O predicates include `read/1`, `write/1`, `get/1`, `put/1`, and most of the file-handling predicates. It happens that goals calling these predicates cannot usually be moved within a clause, and therefore act as barriers to permutation of other goals.

For an example of why this is necessary, imagine three goals `a`, `b`, and `c`, in that order within the body of a clause. `b` has a side-effect. The variables of the goals are instantiated enough that any permutation will work. Exchange `a` and `b`. Now suppose that when the clause executes, `a` fails. But `b` has succeeded, and its side-effect cannot be undone by backtracking. Originally, the failure of `a` would have kept `b` from being reached and the side-effect from being expressed. Similarly, exchanging `b` and `c` is wrong, because if `c` fails it prevents `b` from expressing its side-effect as in

the original program. Unless `a` or `c` is certain to succeed, we cannot move `b`. We may justly call `b` and the like *fixed* predicates, and other predicates *mobile*.

Note that merely fixing a goal that has a side-effect might not suffice to guarantee set-equivalence for its parent predicate. The goal must receive the same set (not necessarily the same sequence), repetitions included, of substitutions that it received in the original program, at least for those variables in the substitutions that can influence its side-effects. Thus it suffices that the ancestors of a fixed goal, and the (possibly permuted) goals preceding it in its parent clause, preserve set-equivalence.

The restriction on mobility, however, is actually far stronger than we have implied in the example. Any predicate that has a fixed predicate as a descendant must, in general, be fixed itself. For example, if the fixed built-in predicate `write/1` is within a clause of the predicate `w` and prints something, a predicate `x` that calls `w` might print as well. A predicate `y` that calls `x` might also print, and so forth. Predicates are responsible for what their descendants do: in this way, a single fixed goal can contaminate most of a program.

Can we make these restrictions less severe? If we can detect goals that are bound to succeed, we can exchange them with fixed goals, but this will not affect execution time. If we can determine statically that a given call to a predicate, due to its mode or its particular instantiation, will not match a clause that expresses a side-effect, then we can perhaps reorder its ancestors. Possibly we can insert in the code a test, performed during execution, that shows something similar: if it succeeds, we execute the reordered version; if not, the original.

2.3.2. The cut

Use of cuts precludes a great deal of reordering. In general, any goals preceding the cut in the body of a clause cannot be reordered. Furthermore, no descendants of these goals can be reordered, either.

Consider the goals that occur before the cut. The first goal receives a particular substitution and produces the first member of its output-sequence of new substitutions, possibly instantiating some variables. The second does much the same, and so on. When we reach the cut, the cut chooses this initial substitution just produced by the conjunction of preceding goals. Now, if we reorder these goals, we may be reordering the sequence of substitutions they produce. If we reorder descendants of one of these goals, we may be reordering the sequence of substitutions the goal produces, and as a result the cut might choose a different substitution with this transform than with the original program. Such reorderings preserve tree-equivalence but not set-equivalence. Note further that unless the reordering of the descendants of the goals preserves set-equivalence, we have no guarantee of tree-equivalence at this level: a descendant cut might commit to a substitution incompatible with a higher level's, and our conjunction of goals might fail outright when before it would have succeeded.

Clearly there are some cases in which reordering before a cut is safe: for example, suppose that all variables of the goals are instantiated. There is only one substitution, and each goal becomes a test, performing no new instantiations but merely checking whether its argument satisfy particular properties. (The tests may succeed many times, but the presence of the cut takes care of that.) In that case, reordering is safe, and amounts to putting the inexpensive, likely-to-fail tests first. Yet another restriction can arise from the cut, but because it can have other causes, we discuss it in the next section.

2.3.3. Semi-fixity and preservation

Calls to some predicates require that the instantiations of some of their arguments be *preserved*: kept the same as in the original program. This can be either the mode of arguments or, in some cases, a particular term. For example, the built-in `var/1` succeeds only when its argument, is uninstantiated. Consider the following predicate:

```
a(X,Y,Z):- b(X,Y), c(Y,Z),
      var(Z), d(Z).
a(_,_,zed).
```

Suppose that in the execution of the original program, we always call `a/3` with `z` uninstantiated, that `c/2` may instantiate it, and that `d/1` always will. In the original, if `c/2` instantiates `z`, the first clause fails and the second sets `z = zed`. Now there are two pitfalls. First, if we move the call to `var/1` forwards, `c/2` may instantiate `z` yet `d/1` may wrongly succeed; move it back, and `c/2` may not instantiate, `d/1` may, and the clause may wrongly fail. Second, if through reordering `z` is instantiated in the call to `a/3`, but not to `zed`, the call to `a/3` fails. In both cases the meaning of the program changes. We call predicates like `var/1` *semi-fixed*, for reasons that will soon be apparent.

First, we should not permute calls to a semi-fixed predicate with goals that might alter the instantiation of the variable they need preserved (the *culprit variables*). Only if static analysis shows that culprit variables will already be instantiated at an earlier point, can we permute goals, for doing so cannot change instantiations.

Second, if a predicate demands preservation of a culprit, this preservation must be honored by as many of its ancestors as are necessary to ensure it. If a variable appears both in a call to a semi-fixed predicate and in the head of the calling clause, any request to preserve it is propagated upwards, so that in our example above, `a/3` is semi-fixed in its third argument, and that semi-fixity may well be propagated to predicates that call `a/3`.

Note that the latter restriction does not preclude permutation of the instantiations that a culprit had in the original, so long as we preserve set-equivalence for the substitutions that contain them. For example, if a predicate semi-fixed in a variable originally would have received its culprit in a variety of states, say in a mixture of uninstantiated and with various instantiations, then any permutation of those states by a transform would be acceptable. But if the culprit originally would have always been uninstantiated, but with the transform is sometimes instantiated (or any analogous change), that would be unacceptable.

2.3.3.1. Due to the cut

Predicates having a cut in at least one clause can be semi-fixed. If, due to reordering, we call such a predicate in a mode different from that in the original, we get typical semi-fixed behavior. An example:

```
a(X,Y,b):- !.
a(X,Y,Z):- c(X,Y), d(Y,Z).
```

The predicate is meant to be called with the third argument instantiated; the first clause catches a special case. With the third argument uninstantiated, a call matches only the first clause. In other cases, goals preceding the cut may act as tests of mode, so that we may have to preserve either instantiation or uninstantiation. Again, if we preserve the set of instantiations seen by the calls to such a fixed predicate, everything works properly.

2.3.3.2. Due to other built-in predicates

We have already seen how `var/1` can cause semi-fixity. Similar *meta-logical* tests such as `nonvar/1`, `atom/1`, `atomic/1`, and `number/1` also demand that their arguments be preserved, as do comparison predicates that check the position of a variable or term within the standard order. Note that some built-in predicates, notably the arithmetic ones, those that output single characters, and such others as `functor/3`, `arg/3`, and `=../1`, require that some or all of their arguments be fully-instantiated or even of a particular type. Treating them as semi-fixed is a sufficient (if not always optimal) remedy, and we discuss other measures later. Note that a fixed predicate also requires preservation of its argument, but it gets it automatically because its ancestors are also fixed. We consider the all-solutions predicates in a later section.

The *Implication* is Prolog's analog to the *if-then* statements of algorithmic languages. Only if we can satisfy a conjunction of goals `p` do we try to satisfy a conjunction of goals `q`: this is written `p -> q`. It can be the first half of a disjunction, forming an *if-then-else*. Implication behaves as if defined by

```
P -> Q :- P, !, Q.
```

Goals of the implication are just like the goals before a cut. The if-then or if-then-else behaves just like a call to a semi-fixed predicate.

Prolog's explicit negation-as-failure predicate, written `\+/1` (or, in some dialects of Prolog, `not/1`), succeeds if the conjunction of goals in its argument yields no solution. It should always be called with all variables instantiated [38], but programmers do not always heed this. `\+/1` behaves as if defined by

```
\+(X) :- X, !, fail.
\+(X).
```

and so it should be treated as semi-fixed. Any set-equivalent reordering of the goals in its argument (and their descendants) should create a transform that succeeds and fails in the same ways as its original.

2.3.3.3. Due to the logical variable

As we have noted, terms in Prolog can be partially instantiated (PI): that is, contain uninstantiated variables. To ensure set-equivalence, we should preserve both the instantiations of PI structures and the order of operations on them. An example should clarify the problem.

Consider the following predicate from the book by Sterling and Shapiro [49].

```
lookup(Key,d(Key, X, _, _),Value):-
     !, X = Value.
lookup(Key,d(Key1, _, Left, _),Value):-
     Key < Key1, lookup(Key,Left,Value).
lookup(Key, d(Key1, _, _, Right), Value):-
     Key > Key1, lookup(Key,Right,Value).
```

`lookup/3` lets one build and access a binary tree that stores values under keys. Initially we might call `lookup/3` with a key, an uninstantiated variable, and a value (possibly uninstantiated, for we can instantiate it later). The second argument is set to a PI structure.

Now, future operations on the tree will either look up a datum (if the key is in the tree already) or add to the tree. Every operation on the structure can potentially add to it. If we permute operations, on the structure, we build a different structure. Thus the following two clauses

```
example(Dict):- lookup(5,Dict,a), lookup(6,Dict,b).
example(Dict):- lookup(6,Dict,b), lookup(5,Dict,a).
```

build two very different structures:

```
| ?- example(Dict).
Dict = d(5,a,_83,d(6,b,_106,_107)) ? ;
Dict = d(6,b,d(5,a,_106,_107),_84) ? ;
no
```

(The numbers preceded by underscores are internal names for variables. The system's responses to user input are in a lighter typeface.) Considering the usual uses of a predicate like `dict/3`, the different structures hardly matter: we can treat the dictionary as the celebrated "black box" with internal workings that need not concern us, and permute operations on it (though that cannot speed the program's execution in this case). The internal structure may well matter in some programs, however. For example, suppose that we write a predicate that determines the depth of a tree produced by the action of a series of calls to `lookup/3`. If we permute these calls, the tree that the transform produces may be shallower or deeper than the one produced by the original. A test comparing the depth to some constant might then succeed when it should have failed, and the observed behavior of the program could change radically.

Note that the built-in predicate `functor/3` can also build PI structures. If we call it with its first argument uninstantiated and its others set to a functor name and an arity, it returns a term of that name and arity, its arguments all being uninstantiated variables.

2.3.3.4. Due to all-solutions predicates

An *all-solutions predicate*, or *ASP*, namely `setof/3`, `bagof/3`, or `findall/3`, is a built-in predicate that takes a *template* as its first argument and a goal as its second, and produces a list as its third. The template is typically a structure that contains variables appearing in the goal.

The ASP finds all solutions to the goal and creates a list (in some cases several, on backtracking) of copies of the template, its variables instantiated to correspond to the solutions the ASP has found.

To ensure set equivalence, we must preserve the instantiations of variables appearing in the goal within the ASP (its second argument). Otherwise the ASP will deliver a different list, or even a sequence of different lists on backtracking. Reordering subgoals (or their descendants) within the ASP's goal will make it produce a permuted list and make it structure-inequivalent rather than set-equivalent, with one exception: `setof/3` eliminates duplicate solutions and sorts the others into a standard order, so that if the reordered goal preserves set equivalence, the `setof` produces exactly the same list in original and transform. Note that even though we seek all solutions, a permuted goal can still reduce effort by reducing the cost of failures.

2.3.3.5. Due to recursion

Reordering the goals of a recursive predicate is often unsafe. Consider this example [49] written in pure Prolog:

```
select(X, [X|Xs], Xs).
select(X, [Y|Xs], [Y|Ys]) :- select(X,Xs,Ys).

permutation(Xs, [X|Xs]) :-
      select(X, Xs, Zs), permutation(Zs, Ys).
permutation([], []).
```

`permutation/2`, given a list as its first argument, produces (on backtracking) all permutations of the list. Given a variable instead, it goes into an infinite loop. If we swap the two goals of the first clause of `permutation/1`, we will get an infinite loop if the *second* argument is a variable.

A simple solution is to treat all calls to a recursive predicate as semi-fixed in all variables, including, of course, the recursive calls themselves. User annotations indicating "safe" modes are too prone to error, and static analysis cannot in general tell us what modes are safe, let alone what reorderings of recursive clauses. Being conservative and making the calls semi-fixed ensures that recursive predicates always get the same sets of instantiations that they would have had in the original. If the programmer knew what he was doing in writing the program, there should be no problems with infinite recursions.

2.3.4. Other restrictions from built-in predicates

As we mentioned in the introduction to this chapter, sometimes we can make a program more efficient by combining two clauses into a single one that contains an explicit disjunction. Other restrictions permitting, we can undo this, which lets us reorder with other members of the new clauses the goals that were in the halves of the disjunction. Otherwise, we can permute only the goals within a half of the disjunction.

Failure-driven loops are common in code written for Prolog systems without garbage collection. A failing clause would do nothing in pure Prolog because it could instantiate no variables, but in practice it can express side-effects. If we have some code, perhaps complicated, that we need to repeat many times, and we record its result using a side-effect, we can make a loop that fails repeatedly:

```
show_all:- t(X,Y,Z), write((X,Y,Z)), nl, fail.
show_all.
```

This prints out all tuples of `t/3`: it finds one, prints it and a new-line character, and then fails, forcing backtracking to find another; when all are found, the second clause makes the predicate succeed. Goals of a failure-driven loop must remain within it, but we may reorder them, with the usual caveats and restrictions.

2.4. Finding restrictions and exceptions

Detecting restrictions on reordering is usually easy. Even a simple traversal of a program in mimicry of execution will spot some of them, and once we know the restrictions we can perform any reorderings that remain permissible, provided that our reordering heuristic tells us that they might be profitable.

We need not analyze a program further, but more information gives us more chances to reorder. Often no useful reorderings will seem permissible, and then we will want to exploit various properties

of predicates — determinism, guarantees that they instantitate particular arguments, special behavior in particular modes — that often make more orders possible. Given better knowledge of how a predicate behaves, we can find exceptions to restrictions on reordering. Even incomplete information can give us an incentive to include run-time tests to select a reordered version of a predicate (or even of a conjunction of goals) when its use will be safe. The literature on Prolog contains techniques that are useful here, and we discuss how they can be used to gather information about programs. We also mention other properties of predicates that might be inferred or indicated by user annotations.

2.4.1. Detecting restrictions

Fixed and recursive predicates are especially easy to detect: start at a top-level predicate (one called by the user to initiate execution of the code) and traverse the program as if executing it, keeping a list of ancestor predicates. Maintain a dictionary of the program's predicates. On encountering a call to a fixed built-in, mark the dictionary entries of the ancestors as fixed; on encountering a call to an ancestor, mark the entries of the predicates in the recursive loop as being recursive. The dictionary can be passed to other predicates or written to a file for later use.

The immobilizing effects of cuts and if-then are better handled by looking at an entire clause before searching its goals. One can mark predicates as ineligible for reordering, and then traverse them and their descendants, passing on the marking. Several passes over a program might be the best way, each handling the types of analysis for which the particular traversal is best suited.

Semi-fixity is somewhat more difficult to handle. Built-in and cut-bearing predicates make demands for particular instantiations or (more usually) preservation of variables, demands that must sometimes be propagated upwards to their ancestors. As before, traversal of a program suffices to keep track of these things. Compare goals against a list of built-in predicates and their demands, and detect the occurrences of cut and if-then that cause demands. Also check for constructs that build partially-instantiated structures, and keep track of how they are propagated between predicates. After examining a clause and its descendants, simply note which variables that have demands occur in the

clause's head. Combining the demands from the clauses of a predicate gives demands for the predicate itself.

2.4.2. Finding exceptions

The most useful exception to restrictions on reordered code involves showing that a permutation need not be forbidden in order to preserve the instantiation of a variable. For example, suppose that `d/1` in the following clause demands an instantiated argument, and `c/2` always sets `Z` (assume that `Z` will not be partially-instantiated):

```
a(X,Y,Z):-
        b(X,Y),
        c(Y,Z),
        d(Z).
```

Probably `d/1` is a test, and so we would like to put it first: it should cost little to evaluate and might save the trouble of trying the other goals. Of course we can do something like the following:

```
a(X,Y,Z):-
        var(Z) ->
                b(X,Y),
                c(Y,Z),
                d(Z)
        ;
                d(Z),
                c(Y,Z),
                b(X,Y).
```

But unless we are really unsure of the instantiation of `Z`, this increases the size of the program and adds a test needlessly. If `Z` is certainly set, we want the reordered version; if not, the original.

2.4.2.1. Modes

The classic DEC-10 Prolog compiler [41] and some of its descendants allow users to specify the modes in which a predicate will be called; for example, a predicate `foo/3` might have this declaration.

```
:- mode foo(+,-,?).
```

The `+` indicates an instantiated argument, `-` an uninstantiated, and `?` one of either type, an unknown,

or a partially-instantiated one. The compiler then generates object code for those modes only; the result is more compact and faster than a general-purpose version.

This kind of declaration tells us in what modes a predicate might be called during the execution of a program. This is not very useful for reordering. We really want to know two other things: the instantiations a predicate produces in the variables of a call, and the modes in which a predicate can be "safely" called.

A predicate is called in an *input mode* and returns an *output mode*. If the predicate can function given several input modes, it might produce a different output mode in each case. Typically a predicate instantiates all of its arguments at least partly, but we cannot rely on this: an argument may be superfluous, it may control a test, or it may sometimes become instantiated and sometimes not. If we can statically determine output modes for the possible input modes to a predicate, we should be able to do more reorderings, and do with fewer run-time tests before reordered sections of code that make demands.

Preservation of the input mode of a goal obviously suffices to preserve its behavior: if a goal is fed the same instantiations it had in the original code, it should behave as before, even if the rest of the program is altered. For calls to recursive predicates, preservation is probably the best choice: changed modes there can cause infinite recursions. In other cases, preservation may be too restrictive. If we can find statically a set of *legal modes* in which a such a predicate functions correctly, then more orders should be possible; if not, we simply fall back on preservation.

Note further that information about modes need not be exact. The better it is, the more opportunities should arise for reordering, but even declarations in the traditional three-symbol system could be helpful, especially if we retain a few run-time tests to resolve ambiguities of ? declarations. We stress that the meanings of declarations for pairs of input and output modes, or for legal modes, would be quite different from those for conventional modes.

2.4.2.2. Mode Inference

A system of modes can be complex and have the flavor of a system of data types, with symbols for PI structures, lists, aliased variables, and so forth [35, 48]. But then user annotation becomes difficult and prone to errors. Automatic inference of modes can be stultified by an especially complex system [11], but it better than relying on users; it is more accurate if less precise, and we can apply it to many different restructured versions of a predicate if necessary. The recent trend seems to be towards more-complex systems of modes for various types of dataflow analysis [43].

There has been increasing interest over the past few years in *abstract interpretation* or *flow analysis* [3, 8, 11, 31, 33, 34, 58] of Prolog programs. The idea is to execute a program symbolically over an *abstract domain*, usually a complete lattice or complete partial ordering of finite height. This "abstract execution" yields a finite description of the program's behavior during an actual execution [58]. To infer modes, we let the elements of the abstract domain be the mode-symbols, and "execute" the program using them as arguments.

Most researchers consider mode-inference as an application when they present methods for abstract interpretation; Bruynooghe and his colleagues [3, 4], and Debray and his [11, 58], appear to place the most emphasis on modes. Debray's work is particularly interesting because, rather than finding modes by executing a program on an abstract interpreter, he generates a transformed program that, when executed conventionally, yields the mode information. Running the transformed program takes far less time than interpreting its original. It appears to be possible to modify Debray's technique to find legal modes: it even generates output modes internally, so that we need only save them [11].

2.4.2.3. Other Information

Knowing output modes for predicates in general, and legal modes for predicates that make demands, makes many more reorderings possible, but other information about predicates can also be useful. As we noted in the sections on the various restrictions, knowing that a predicate is determinis-

tic, or *functional* — one distinct output for a given input — allows reorderings that would otherwise be forbidden. Often these reorderings are useless (for instance, there is no point in moving before a side-effect goal a goal certain to succeed exactly once), but at times they can put tests or instantiations earlier in a clause and potentially save effort. Debray has presented a method for detecting functional predicates automatically [12].

Various user annotations can be helpful. If a particular level of equivalence is acceptable for a predicate, an annotation to that effect might allow more reorderings. For example, if the internal configuration of a partially-instantiated structure is invisible from the outside, analysis might not be able to detect that and allow structure-inequivalence for the predicate.

2.5. Reordering goals in Prolog: methods and implementation

Once we have decided on a degree of equivalence for a predicate, found the restrictions on reordering the literals in its clauses, and perhaps determined some extra orders that may be shown safe dynamically using run-time tests, we still need to determine which of the permissible orders for a clause body minimizes the cost of its execution. Furthermore, the best order for a clause varies with the mode of the call to its parent predicate [36], and a predicate called in one given mode in the original may be called in another, or even several, in the reordered transform: some calls to it may have been reordered and put thereby into new modes.

We require some sort of heuristic for reordering: the problem is intrinsically difficult (optimally ordering conjuncts of a database query is already in NP [52], and here we have extra difficulties to deal with, such as inference), and our information on the costs of executing goals is imprecise. Also, once we have found a new order, we may have a great deal of bookkeeping to do: definition of new predicates (perhaps one for each of several alternative orders), alteration of calling literals so that they call the new predicates, and so forth.

We need a measure of the cost of evaluating a query. For the static reordering we are doing, the number of predicate calls is convenient. Calls vary widely in the effort needed for their evaluation, so

that this is not, in general, the best measure of cost: number of unifications is better, and ultimately the execution time for a query is what matters. However, unifications also vary greatly in cost; they can require anything from checking a tag and copying an address to a large number of operations [55], and amounts of CPU time are yet harder to estimate statically.

2.5.1. Previous work: Warren's method and extensions

Warren [54] presented a method for reordering goals of conjunctive queries in Prolog. The queries were automated translations of questions in English: a user typed in a question on geography, and a parser generated a query. The order of the words in the question determined the order of the goals in the query — not always an efficient order. Warren gave each mode of each predicate a number: the factor by which the goal multiplies the number of alternatives the system must consider.

For example, suppose that there is a predicate `country/1` with about 150 facts giving names of countries for which the database has information, such as

```
country(canada).
```

A goal `country(C)`, with `C` uninstantiated, matches every fact of `country/1`. By Warren's reasoning, if such a goal appears in a conjunctive query, the number of alternatives the system must consider for that query is multiplied by about 150. With `C` instantiated, however, the goal is a test: because it will often fail, it eliminates possibilities and should have a value less than 1.

Warren estimated the factor for a given mode of a predicate as

$$\frac{|tuples|}{\prod_{instantiated} |domain_i|}$$

$|tuples|$ is the number of tuples in the predicate: for a ground predicate, the number of facts. For every instantiated position of the mode, divide that number by the size of the corresponding domain. For example, a predicate `borders/2` tells that one country borders another. It has about 900 tuples, and because its arguments represent countries, each has a domain size of 150. The estimated factor is therefore 900 for an uninstantiated call, $\frac{900}{150} = 6$ for either type of partly-instantiated call.

and $\frac{900}{150 \times 150} = 0.04$ for a fully instantiated call. Warren found that reordering goals to minimize the product of these factors yielded speedups up to several hundred times.

Warren reordered only top-level conjunctive queries, but one can also reorder clause bodies by his method. We tried this to test the promise of reordering, avoiding the problems of restrictions by working with programs in a subset of Datalog and nearly pure: no constants outside of ground predicates, no recursion, and only a few built-in predicates without side-effects. (In execution of such code, all unifications succeed except in some calls to ground predicates, avoiding another difficulty.) A good example is a program representing a family tree: numerous facts about simple relations (mother-child and husband-wife) and rules for deriving complex ones (mother-in-law, cousins, etc.). For ground predicates the number of tuples is simply the number of facts; for others, executing the predicate itself gives an accurate (though expensive) count of tuples. Speedups were typically 5 to 40, rarely under 1 (slowdowns): somewhat better than Warren's, considering that the database of facts for our programs was about an order of magnitude smaller than his.

Both Warren's method and our extension have drawbacks. Though it seems effective, Warren's function considers only the number of solutions, not their costs. Further, domain size for an argument is problematic even for database programs [54]. Our extended method is expensive: we call each predicate, forcing repeated backtracking, and count the solution-tuples. This is impractical even for "toy" problems like family trees, and would obviously be useless for predicates with unpredictable instantiations or data structures.

2.5.2. Previous work: Markov chains

Li and Wah [29] proposed the use of absorbing Markov chains [24] to model the bodies of Prolog clauses. If we want to estimate the expected cost and probability of success for executing the body of a Prolog clause, given those for the predicates its literals call, we create an appropriate chain and find its properties; from these we get the desired estimates readily. By repeating this for various permutations of the literals, we determine the order expected to be least expensive.

Consider a clause

```
k :- a, b, c, d.
```

(For clarity and conciseness we omit arguments.) Suppose that we know the success probability p_i and cost c_i for each goal i. To find the expected probability and cost of only a single solution to the clause body (e.g., if we called `k/0` interactively and wanted only one solution), we model it as the Markov chain of Figure 2.1. Each goal has a state in the chain. There are two extra absorbing states: F for failure and S for success. Each goal's state has two arcs: one is labeled with its success probability p_i and goes to the following goal's state or to S, and the other is labeled with failure probability $1-p_i$ and goes to the previous goal's state or to F.

Transitions of the chain mimic Prolog execution. For this example, we start in the state of a, proceeding to b if a succeeds, or going to F if it fails. In every state i we move forward (probability p_i), or backtrack (probability $1-p_i$). Eventually we stop in an absorbing state: success or failure.

If we want the cost of finding all solutions to a conjunction of goals, as is more usual, we add an arc of probability 1 from S back to the last goal, as in Figure 2.2. When we find a solution we backtrack, looking for another, until we fail, and the added arc captures this.

Note that this model only approximates how the Prolog engine executes a clause body. It is the basis of a heuristic method: it glosses subtleties of execution that might make analysis impractical. The Prolog engine preserves its state on a stack [55], but a Markov process is memoryless and keeps no history of how it reaches a state. Given a particular instantiation, a goal within a clause body

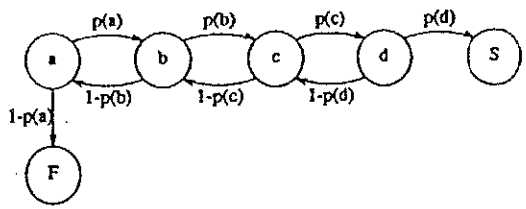

Figure 2.1: A clause body as a Markov chain (single solution).

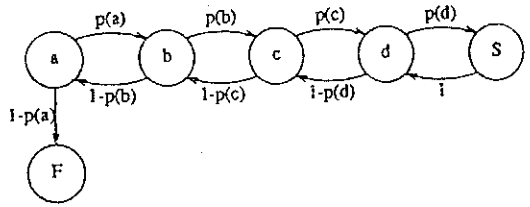

Figure 2.2: A clause body as a Markov chain (all solutions).

succeeds some number of times (perhaps zero), and then fails; when the Markov process is in the corresponding state it has a single fixed probability of proceeding to the next state. Probabilities that goals succeed are not necessarily independent, but probabilities of transitions are. In return for making these approximations, we obtain a practical way to predict the behavior of a clause.

2.5.3. The Markov-chain method: details and extensions

First we present some details, not in the method as it was originally presented, that we have worked out and built on. Then we proceed to variants and extensions.

2.5.3.1. Probability and cost for a clause body

To find the probability p_{body} that a clause body succeeds, we use the single-solution chain of Figure 2.1, even if we are looking at a multiple-solution clause. This is because with that chain we can find the probability that the Markov process ends in state S, and take that as p_{body}, whereas the other chain has only F as an absorbing state and must eventually end there. For the expected cost of a solution we use the appropriate chain. $\sum_{i=1}^{n} c_i v_i$, for the n goals of the body, is the expected cost of all the solutions put together, c_i being the expected cost of goal i and v_i the mean number of visits by the process to state i. Hence the expected cost for solving a single-solution clause is:

$$c_{single} = \sum_{i=1}^{n} c_i v_i,$$

and the expected cost for one solution of a multiple-solution clause is:

$$c_{multiple} = \frac{1}{v_S} \sum_{i=1}^{n} c_i v_i,$$

where v_S is the number of visits to the success-state S.

Calculating p_{body} and all the v_i is textbook [24] mathematics. Begin with the transition matrix P for the single-solution case. An element p_{ij} is the probability of a transition from state i to state j; S has index 1 and F index 2. The chain has r states, s of which are transient. Consider four submatrices of the $r \times r$ matrix P:

$$P = \begin{bmatrix} I_{r-s} & 0 \\ R & Q \end{bmatrix}$$

I_{r-s}, the identity matrix of size $r-s$, for transitions between absorbing states; R, of size $s \times (r-s)$, for transitions into absorbing states; Q, $s \times s$, for transitions between transient states. The zero matrix of size $(r-s) \times s$ shows that nothing leaves an absorbing state. Here is P for the example of the last section:

$$P_k = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1-p_a & 0 & p_a & 0 & 0 \\ 0 & 0 & 1-p_b & 0 & p_b & 0 \\ 0 & 0 & 0 & 1-p_c & 0 & p_c \\ p_d & 0 & 0 & 0 & 1-p_d & 0 \end{bmatrix}$$

The matrix $N = (I-Q)^{-1}$ provides our answers. Because we start in the first state, the first row of N contains the numbers of visits v_i the process makes to the transient states. The product NR gives p_{body}, the probability of success: it is the first element of the column of state S (here, column 1). We need compute only that element, and so we take the dot product of that column of R and the first row of N.

We find N_k numerically; if we have N goals, we invert an $N \times N$ matrix, and do $2N$ extra multiplications and sundry additions for the probability and cost. If the reorderer is written in Prolog, we might call a routine written in a more-suitable language, such as C, to generate and invert the matrix.

For the multiple-solution chain, S is no longer an absorbing state, and for our example the matrix P becomes:

$$P_k = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1-p_a & 0 & p_a & 0 & 0 & 0 \\ 0 & 1-p_b & 0 & p_b & 0 & 0 \\ 0 & 0 & 1-p_c & 0 & p_c & 0 \\ 0 & 0 & 0 & 1-p_d & 0 & p_d \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

It happens that in this case there is a tidy form for the v_i, so that the expected cost of a solution can be written as:

$$c_{multiple} = \frac{1}{v_S} \sum_{i=1}^{n} c_i v_i = \sum_{i=1}^{n} c_i \left(\prod_{i=1}^{n} \frac{p_{i-1}}{1-p_i} \right).$$

2.5.3.2. Applying the results

For each legal calling-mode we find the least-costly reordered version of a predicate. Different modes generally need different orders, and the cost of a goal varies with its mode; essentially we have to consider all permissible permutations, apply the heuristic to each, and see which fares best. For a clause with many goals in which we can reorder the goals as we please, choosing the best permutation, this is very costly: an n-goal clause has $n!$ permutations. In practice, restrictions sharply reduce the number of possible orders: for example, if the third goal of a five-goal clause body is fixed, the number plummets from $5! = 120$ to $2!\,2! = 4$. We can use special Markov chains for clauses containing cuts, explicit disjunctions, or the like, but this is impractical because it is easier to handle the separate conjunctive parts as if they were bodies of short clauses and combine the results.

When many orders are permissible, however, we can adapt an idea presented by Smith and Genesereth [46] for reordering of conjunctive queries whose conjuncts perform no inference: use a best-first search such as A^* [2] to decide the least-expensive order of goals. We do a best-first search on a tree of ordered subsets of the clause body, using as a heuristic the expected cost of finding all solutions to the Markov chain for a subset. Every time we expand a node, we choose the one with the least-expensive sequence. Clearly the all-solutions formula gives an admissible heuristic for A* [2], because the cost the formula gives for a partial sequence of goals is always an underestimate of that for the complete sequence that begins the same way.

2.5.3.3. Initial probabilities and costs

Cost and probability of a clause come from those of its goals; in the subset of Prolog we are considering at this point, these come from costs and probabilities of facts. Facts may have non-variable arguments, so that their heads might fail. Evaluating a fact costs one call; its success probability is problematic unless the call is uninstantiated, when it is unity. We can, like Warren [54], find *domains* for each argument; a domain might be the set of constants in that argument of the predicate, or perhaps the set of all constants in the program. We take the probability as $\prod |domain_i|^{-1}$ for every position i with a constant in both fact and call.

If we allow full pure Prolog with recursion, it becomes hard to tell the likelihood that a call unifies. If an argument-position has a variable in neither goal nor clause head, the probability is no longer unity. For constants, we can use domains, as with facts. For structures, the predicates are usually recursive, with a special case matching empty structures. The size of a structure affects both the probability and cost of a call; the system or user must estimate sizes before run-time. Some predicates may require the user to provide an explicit probability of unification for each argument of each clause head.

2.5.4. Implementation

We assume that we have extensively analyzed the program and know what predicates are fixed, which have demands for preservation or particular instantiations, and so on. We may have done some sort of mode inference, or provided mode declarations (output and legal modes); this is not essential, but it should greatly increase the number of possible reorderings. We should also have unification probabilities for some clause heads, inferred or provided by the user, and probability and cost estimates for recursive predicates.

Note that demands, legal modes, and the like may change for a predicate once some of its clauses have been reordered. We may need to repeat some of the analysis for reordered versions of a predicate. Fortunately, the progress of reordering necessarily proceeds bottom-up, so that predicates at the bottom of the program tree are reordered first, and any repeated analysis will not interfere.

Reordering starts from an entry predicate (typically the predicate called to start execution of a program), and the reorderer traverses the program depth-first until it finds a predicate whose clauses have goals with known cost and success probability. It works on the clauses one by one. If a clause contains control predicates, the reorderer divides it into blocks of goals, deciding which require the one-solution chain and which the multiple-solution. Choosing a legal mode of the predicate, it propagates to the goals the instantiations produced by that mode.

Operation of the reorderer-program is fairly obvious. The reorderer looks, in order, at the one or more blocks of goals from the clause. If a block is immobile, it uses the one-solution Markov chain, calling an external predicate (written in C) to construct and invert the matrix; this gives probability and cost for the block. It permutes other blocks exhaustively and computes their cost, saving the least expensive order; or, if too many permutations are possible, it reorders them using best-first search. It finds the success-probability by, for the best order, evaluating a one-solution chain. It discards illegal orders.

Combining the probabilities and costs for the blocks of goals yields values for the clause body; combining results for clause bodies with match probabilities for heads yields an estimated probability and expected cost for a call to the reordered predicate in that legal mode. The reorderer stores the reordered version under a new name, along with mode, probability, and cost. It repeats this for every legal mode of the predicate. Working upwards, the reorderer handles every user predicate in the program, changing goal names as necessary to correspond to the new predicate names.

2.5.5. Performance of reordered code

Reordering methods seems most useful for predicates with these properties:

(1) **Mobility:** We need something to reorder: at least several permitted orders for the goals in a clause. Restrictions and short clauses prevent this.

(2) **Nondeterminism:** Reordering is less helpful for predicates that do not use Prolog's simulated nondeterminism. Moving tests forward can save effort by making an essentially deterministic clause fail sooner, but reordering cannot generate an improved deterministic algorithm.

(3) **Diverse probabilities and costs:** If probabilities and costs of goals are too similar for different goals, they cannot guide the heuristic towards a better order.

(4) **Unbalanced search graph:** A balanced search graph exacerbates the lack of the previous property: again, goals appear too similar.

Prolog databases, some search programs, and nondeterministic programs in general gain the most. Standard benchmark programs violate at least the first two of our criteria: they are largely deterministic, and there is little in them to reorder.

2.5.6. Reordered programs

Table 2.1 shows some results of reordering a family-tree program written in nearly pure Prolog.

Table 2.1: Results of reordering a family tree program.

Predicate	Mode	Number of calls: original	Markov	optimal	ratio of improvement
`aunt`	(−,−)	1452	985		1.47
	(−,+)	61872	1409		43.91
	(+,−)	2035	2035		1.00
	(+,+)	93354	67238		1.39
`brother`	(−,−)	792	792	792	1.00
	(−,+)	4555	539	530	8.45
	(+,−)	1004	1004	1004	1.00
	(+,+)	16003	21449	16003	0.75
`cousins`	(−,−)	62987	1477		42.65
	(−,+)	106606	2034		52.41
	(+,−)	63570	2203		28.86
	(+,+)	138088	151062		0.91
`grandmother`	(−,−)	230	200	200	1.15
	(−,+)	11254	367	367	30.66
	(+,−)	548	548	548	1.00
	(+,+)	28426	18705	18705	1.52

55 constants in the program represent people. We called each predicate in each mode, with one call for each possible instantiation. Therefore, testing mode (-,-) requires one call, modes (-,+) and (+,-) require 55 apiece, and modes (+,+) require 3025. The programs given in Appendix A did the reordering; the Prolog code requires a version of C-Prolog 1.5 [40] that we modified heavily to be able to call the routine written in C. The C code in turn calls some LINPACK [17] routines (not included in the appendix, but widely available) to invert the matrix. Results were generated by another version of C-Prolog 1.5, instrumented to count calls. We give figures for the original program, the reordered version, and for the best reordering possible (found by exhaustive enumeration when practical).

Figure 2.3 gives pertinent predicates of the original program. There are also 10 facts for `girl/1`, 19 for `wife/2`, and 34 for `mother/2`.

Figure 2.4 gives reordered versions of the above predicates. (Clearly, predicates with clauses of one goal have nothing to reorder.) This is essentially raw output from the reorderer, with variables represented as numbers preceded by underscores.

```
female(X) :- girl(X).
female(X) :- wife(_,X).
male(X) :- not(female(X)).
father(X,Y) :- mother(X,Z), wife(Y,Z).
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).
married(X,Y) :- wife(X,Y).
married(X,Y) :- wife(Y,X).
siblings(X,Y) :- mother(X,Z), mother(Y,Z), unequal(X,Y).
unequal(X,Y) :- X==Y.
sister(X,Y) :- siblings(X,Y), female(Y).
brother(X,Y) :- siblings(X,Y), male(Y).
grandmother(X,Y) :- parent(X,Z), mother(Z,Y).
cousins(X,Y) :- parent(X,Z), parent(Y,W), siblings(W,Z).
cousins(X,Y) :- parent(X,Z), parent(Y,W), siblings(W,V), married(V,Z).
aunt(X,Y) :- parent(X,Z), sister(Z,Y).
aunt(X,Y) :- parent(X,Z), brother(Z,W), wife(W,Y).
```

Figure 2.3: Predicates from the family-tree program.


```
father_uu(_5804,_5805):-wife(_5805,_5806),mother(_5804,_5806).
father_ui(_5804,_5805):-wife(_5805,_5806),mother(_5804,_5806).
father_iu(_5804,_5805):-mother(_5804,_5806),wife(_5805,_5806).
father_ii(_5804,_5805):-wife(_5805,_5806),mother(_5804,_5806).
siblings_uu(_7234,_7235):-mother(_7234,_7236),mother(_7235,_7236),
      unequal(_7234,_7235).
siblings_ui(_7234,_7235):-mother(_7235,_7236),mother(_7234,_7236),
      unequal(_7234,_7235).
siblings_iu(_7234,_7235):-mother(_7234,_7236),mother(_7235,_7236),
      unequal(_7234,_7235).
siblings_ii(_7234,_7235):-unequal(_7234,_7235),mother(_7234,_7236),
      mother(_7235,_7236).
sister_uu(_7434,_7435):-female(_7435),siblings_ui(_7434,_7435).
sister_ui(_7434,_7435):-female(_7435),siblings_ui(_7434,_7435).
sister_iu(_7434,_7435):-siblings_iu(_7434,_7435),female(_7435).
sister_ii(_7434,_7435):-siblings_ii(_7434,_7435),female(_7435).
brother_uu(_8054,_8055):-siblings_uu(_8054,_8055),male(_8055).
brother_ui(_8054,_8055):-male(_8055),siblings_ui(_8054,_8055).
brother_iu(_8054,_8055):-siblings_iu(_8054,_8055),male(_8055).
brother_ii(_8054,_8055):-siblings_ii(_8054,_8055),male(_8055).
grandmother_uu(_9346,_9347):-parent_uu(_9346,_9348),mother(_9348,_9347).
grandmother_ui(_9346,_9347):-mother(_9348,_9347),parent_ui(_9346,_9348).
grandmother_iu(_9346,_9347):-parent_iu(_9346,_9348),mother(_9348,_9347).
grandmother_ii(_9346,_9347):-mother(_9348,_9347),parent_ii(_9346,_9348).
cousins_uu(_8342,_8343):-married_uu(_8344,_8345),parent_ui(_8342,_8345),
      siblings_ui(_8346,_8344),parent_ui(_8343,_8346).
cousins_uu(_8332,_8333):-parent_uu(_8332,_8334),siblings_ui(_8335,_8334),
      parent_ui(_8333,_8335).
cousins_ui(_8332,_8333):-parent_iu(_8353,_8334),siblings_iu(_8334,_8335),
      parent_ui(_8332,_8335).
cousins_ui(_8342,_8343):-parent_iu(_8343,_8344),siblings_iu(_8344,_8345),
      married_iu(_8345,_8346),parent_ui(_8342,_8346).
cousins_iu(_8342,_8343):-parent_iu(_8342,_8344),married_ui(_8345,_8344),
      siblings_ui(_8346,_8345),parent_ui(_8343,_8346).
cousins_iu(_8332,_8333):-parent_iu(_8332,_8334),siblings_ui(_8335,_8334),
      parent_ui(_8333,_8335).
cousins_ii(_8332,_8333):-parent_iu(_8332,_8334),parent_iu(_8333,_8335),
      siblings_ii(_8335,_8334).
cousins_ii(_8342,_8343):-parent_iu(_8342,_8344),married_ui(_8345,_8344),
      parent_iu(_8343,_8346),siblings_ii(_8346,_8345).
aunt_uu(_11311,_11312):-wife(_11313,_11312),brother_ui(_11314,_11313),
      parent_ui(_11311,_11314).
aunt_uu(_11302,_11303):-sister_uu(_11304,_11303),parent_ui(_11302,_11304).
aunt_ui(_11311,_11312):-wife(_11313,_11312),brother_ui(_11314,_11313),
      parent_ui(_11311,_11314).
aunt_ui(_11302,_11303):-sister_ui(_11304,_11303),parent_ui(_11302,_11304).
aunt_iu(_11302,_11303):-parent_iu(_11302,_11304),sister_iu(_11304,_11303).
aunt_iu(_11311,_11312):-parent_iu(_11311,_11313),brother_iu(_11313,_11314),
      wife(_11314,_11312).
aunt_ii(_11311,_11312):-wife(_11313,_11312),brother_ui(_11314,_11313),
      parent_ii(_11311,_11314).
aunt_ii(_11302,_11303):-parent_iu(_11302,_11304),sister_ii(_11304,_11303).
```

Figure 2.4: Reordered predicates from the family-tree program.

Note the new names for the versions of predicates that are tuned to a particular mode: the

terminal letters are for uninstantiated (-) and instantiated (+). We can add a dummy predicate for anything that a user might call interactively, for example:

```
aunt(X,Y):-
      var(X) ->
            ( var(Y) ->
                  aunt_uu(X,Y)
            ;
                  aunt_ui(X,Y)
            )
      ;
            ( var(Y) ->
                  aunt_iu(X,Y)
            ;
                  aunt_ii(X,Y)
            )
      ).
```

As with other such run-time tests, the tests are very inexpensive (checking tag bits), so that this is bulky but fast. In many cases, the reorderer produces only one or two distinct versions of a predicate, allowing fewer clauses and tests.

Gains are most impressive for the half-instantiated modes. Apparently, partial instantiation is exploited by a good goal order, so that large search trees are pruned early. This does not happen so readily for mode (-,-): the first goal is called uninstantiated, and so must be called in its most expensive form. Similarly, for mode (+,+); enough variables are already instantiated that goal order is not crucial; sometimes reordered clauses are more expensive in this mode.

We also reordered (static analysis using programs, but we did the actual reordering manually) some rules from a simple corporate database (over 100 employees) written in Prolog; Table 2.2 summarizes the results. (These results and those in Table 2.3 were generated using SB-Prolog 2.3 [13]. The facts in this database are indexed on an employee identification number; once that is instantiated, many goals of the rules become trivial. Reordering essentially becomes a way to make the rules find, as quickly and inexpensively as possible, the smallest superset of these numbers whose owners satisfy the rule.

Table 2.2: Results of reordering a corporate database program.

Name and mode	Original version	Reordered version	Ratio of improvement
benefits(-,-)	1288	551	2.34
pay(-,-,-)	1112	1112	1.00
pay(-,jane,-)	55	55	1.00
maternity(-,-)	294	142	2.07
maternity(-,jane)	20	20	1.00
average_pay(-,-)	611	611	1.00
fms(-,-)	284	243	1.17
fms(-,jane)	29	29	1.00

Table 2.3 shows results for several other programs; most of them query small databases. (Again, we used static analysis programs, but manual reordering was more practical.) p58 is Problem 58 from "How to solve it in Prolog" [10]; meal plans meals and team generates project teams. kmbench is a substantial program: a theorem-prover running a set of benchmark problems. Only a single clause of p58, meal, and kmbench can be reordered; only four clauses of team on two levels: the gains in performance are less impressive than with our other programs.

Table 2.3: Results of reordering several programs.

Name and mode	Original version	Reordered version	Ratio of improvement
p58(-,-)	121	78	1.55
meal(-,-,-)	17626	1669	1.06
meal(+,+,-)	8011	7554	1.06
team(-,-)	34682	9990	3.47
team(-,+)	33292	8600	3.87
kmbench	161816	142559	1.14

2.6. Order tracking

We have seen that reordering goals reorders results. A programmer, however, may demand exact equivalence for the transformed program, and refuse to accept one that permutes results. In some applications, exact equivalence for a predicate or an entire program may be essential. We need to be able to reorder predicates yet have the transform produce results in the same order as would the original; we have devised a method that can do this in many cases, which we call *order tracking*.

2.6.1. Basic Idea

Suppose that a predicate is non-deterministic. It or its descendants must contain disjunctions, either as multiple clauses or written explicitly. We can represent the predicate as an AND/OR graph (or an AND/OR tree, if the predicate is not recursive), with an AND node for each clause and an OR node for each predicate or explicit disjunction. The children of each AND node are goals; the children of each OR node are clauses or the alternatives of explicit disjunctions. Now we label the arcs from an OR node to its children with numbers corresponding to the order in which they are written: the order in which Prolog will try them during execution. Hence OR nodes with a single child need no labels themselves, although their descendants may. (Such labeling schemes have been proposed independently for many different purposes; for example, by Hermenegildo [19] to prevent problems with memory management on a machine for parallel Prolog.)

Labels can be strings of bits: for a disjunction of n alternatives we need $\lceil \log_2 n \rceil$ bits. Figure 2.5 shows a labeled tree for the following contrived program:

```
a:- b, c, d.
a:- e, f.
a:- g, h, i.
a:- j, k.

b:- l.          b:- m.          b:- n.          c:- o.
d:- p.          d:- q.          e:- r.          f:- s.
f:- t.          g:- u.          g:- v.          h:- w.
h:- x.          i:- y.          j:- z.          j:- a1.
j:- b1.         k:- c1.
```

Now if we traverse the tree as Prolog would execute the program, appending labels to a bit-string as we try alternatives and removing them when we backtrack, we get a unique string for each solution. The strings are in order: a solution's string is always larger than that of its predecessor, if we treat strings as binary fractions (give the leading bit of two strings the same significance even if it is a leading zero, whether or not the strings have the same length). Thus the strings tell us the "natural" order of the answers to a predicate.

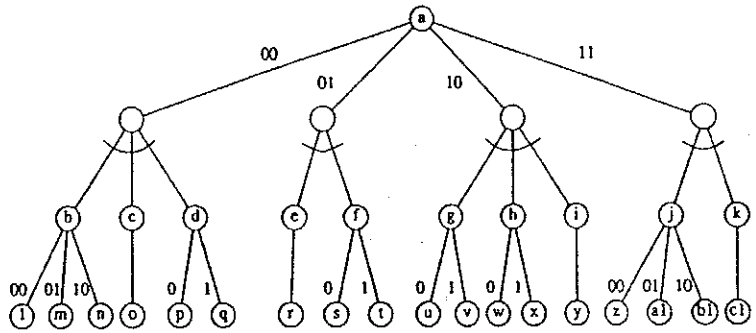

Figure 2.5: A labeled AND/OR graph

Consider a simple implementation of this, done by altering the program source (we present details of a similar, practical version later). Suppose that we make predicates return labels to the goals that call them, and then the parent clause collects and combines them:

```
% Original predicate:
foo(X,Y):- bar0(X,Z), bar1(Z,Y), bar2(W,Z).
foo(X,Y):- bar3(X,Z), bar4(Y,Z).

% A transform that handles labels:
foo(X,Y,L):- bar0(X,Z,L0), bar1(Z,Y,L1), bar2(W,Z,L2),
     combine([0,L0,L1,L2],L).
foo(X,Y,L):- bar3(X,Z,L0), bar4(Y,Z,L1), combine([1,L0,L1],L).
```

`combine/2` simply merges the labels into a single one, in order; note that the first member of the input lists in the example is the label of the clause itself. As in the more-abstract example, the new version of the predicate returns with every finite substitution a combination of labels, or *signature*, that uniquely identifies it; the signatures are in ascending order.

Continuing the example, suppose that we now permute goals in the clauses of the transform `foo/3`, but we do not change the way we combine the labels:

```
% A transform that handles labels, but reordered
foo(X,Y,L):- bar2(W,Z,L2), bar1(Z,Y,L1), bar0(X,Z,L0),
     combine([0,L0,L1,L2],L).
foo(X,Y,L):- bar4(Y,Z,L1), bar3(X,Z,L0), combine([1,L0,L1],L).
```

If the reorderings are set-equivalent, they will produce the same finite substitutions, only in a new order — but each will have the same signature as in the original. Therefore, to make a reordered predicate return answers in the same order as would the original, we need only collect substitutions and their signatures, sort them into order, and rig a mechanism to return the first answer on the initial call, and successive answers on backtracking.

2.6.2. Details

When we do order tracking over predicates descendent from a goal or from a conjunction of goals, we have to find all solutions. Unless the effort saved by reordering is especially large, this is impractical if we need only one solution. Therefore we cannot in general use order tracking let us reorder goals (or their descendants) preceding a cut.

Whether it is practical to reorder goals and then use order-tracking to restore lost reflexive equivalence will depend greatly on the amount of effort that the reordering saves, and on the overhead of order-tracking. If the Markov-chain or other heuristics indicate that reordering is very beneficial, yet set-equivalence will not suffice, order-tracking may be useful. The overhead will vary greatly with the implementation of Prolog, and the chosen implementation of order-tracking itself.

2.6.2.1. Implementation in software

The last example shows the basic idea of how to do order tracking in software: tag the branches of disjunctions with labels, add code to combine the labels into signatures in the proper order, collect the results of a call to a predicate, sort them into order by signatures, and mimic the original code by returning them one by one on backtracking. Here we present a way to implement order-tracking in software.

The top level of any section of code on which we might want to do reordering and order tracking will have either a predicate or a conjunction of goals (perhaps themselves reordered). In the latter case, for the sake of convenience, we replace the goals with a single goal calling a new predicate; for example:


```
% Before: the first two goals are going to be swapped and tagged
foo(X,Y):- bar0(X,Z), bar1(Y,Z), bar2(X,Y,Z,W).

% First step: permute the goals and define a new predicate foo2/3
foo(X,Y):- foo2(X,Y,Z), bar2(X,Y,Z,W).

foo2(X,Y,Z):- bar1(Y,Z), bar0(X,Z).
```

Now we always have a single goal calling a predicate. Call this predicate the *tracked predicate*.

We alter clauses of order-tracked predicates as follows:

(1) The head and (most) goals of a clause each gain two integer arguments: a *tag* and its *size* (its length). The tag is a complete or partial signature, typically implemented as an integer.

(2) Each clause that is non-deterministic (hence no ground clause) gains an extra goal at its end: a call to a predicate `combine/4`. This takes two structures, built explicitly as arguments to the call, one containing the tags in the original order and the other their lengths, and combines them into a new tag and length.

(3) Ground clauses get a tag and length. Deterministic predicates do not change, but deterministic clauses that aren't mutually exclusive need to return a tag and length as well. This can all be done explicitly in the clauses' heads.

We replace the tracked predicate with code resembling the following:

```
foo(X,Y):-
      bagof(t(Tag,Mask,X,Y),foo(X,Y,Tag,Mask),List0),
      longest(List0,Length),
      pad_sigs(List0,Length,List1),
      keysort(List1,List2),
      dole(List2,t(X,Y)).
```

First we find a list of all solutions to the tagged version `foo/4`, using `bagof/3`. If we cannot place an upper bound on the length of the signatures (`foo/4` or some descendant is recursive, non-deterministic, and handles lists or other structures with sizes we can't know in advance), we must find the longest signature (as with `longest/2`) and pad shorter ones to its length (`pad_sigs/3`). Finally we sort the list of answers into order and have them returned one by one with backtracking:

```
dole([_-H|_],H).
dole([_|T],X):- dole(T,X).
```

The length of signatures can cause problems on many Prolog systems. For some implementations on machines with a 32-bit word, integers can be at most 29 bits; on other systems [6] there is, for practical purposes, no limit on size. If descendent predicates have many alternatives (as with large collections of facts) or build recursive data structures nondeterministically, integers may not hold enough bits. Tags can be data structures rather than strings of bits, but manipulating those is much slower.

An attractive but incorrect solution to problems due to long tags is to collect substitutions at lower levels, assign shorter tags to them, and return them with these tags. A low-level predicate may produce tens of substitutions that sport tags near 32 bits, when 5-bit tags would apparently suffice. The problem is that unless we know that a low-level call elicits all possible substitutions from a predicate, such a relabeling is unreliable. Consider this:

```
foo(X,Y,Tag,Long):-
      goal2(W,Z,T2,L2),
      goal0(X,Z,T0,L0),
      goal1(Y,W,T1,L1),
      collect(t(T0,T1,T2),l(L0,L1,L2),Tag,Long).
```

Suppose that the tags returned by `goal0/4` approach an inconvenient length. If `goal2/4` instantiates `Z`, then typically the substitutions elicited from `goal0/4` given this instantiation of `Z` are not its complete "repertoire." They are generally some subset, not necessarily contiguous, of the substitutions it can return. Suppose that we sort these and relabel them. Now on backtracking `goal2/4` instantiates `Z` again, and we sort and relabel the new substitutions produced by `goal0/4`. The sets of tags from `goal0/4` are inconsistent, and because the tags of `goal0/4` are the leading bits of the signature for `foo/4`, order-tracking breaks down entirely.

2.6.2.2. Implementation in the Prolog engine

We can augment the usual Prolog execution engines to support order tracking. Because these are usually implemented in software or firmware rather than hardware, the extra features are not difficult to add. We describe in brief the most popular engine, the Warren Abstract Machine (WAM) [55], and the changes required for it to support order tracking.

The WAM is Warren's extension of the execution model described in his doctoral thesis [56]. It has become the standard model for serial Prolog. Figure 2.6 shows the data areas of the WAM.

The *code* area holds the program, compiled into Warren-machine object code; there are also three stacks. The *heap* holds data structures and global variables, which are created, altered, and discarded (on backtracking) as necessary. Warren used structure-sharing [56] to represent structures in his original machine, but the WAM uses structure-copying [32], pushing copies of altered structures onto the heap instead.

The *stack* contains *environments* and *choice points*. The WAM pushes an environment onto the stack on entering a clause (unless the clause has no permanent variables [55]). An environment has value cells for variables local to the clause, and pointers that show where within its parent clause execution will continue when the clause completes. Environments can be discarded when they are no longer needed (*last call optimization* [57]), or can have variable cells removed (*environment trimming* [55]).

The WAM pushes a choice point (*CP*) onto the stack upon entering a predicate. A CP contains the current state of the machine, and a pointer to the next untried clause of the predicate. If the current clause fails, the WAM checks the last CP (pointed to by register B), reloads the saved state into its registers, and tries the alternative clause. Restoring the old values of the registers reclaims the stack and heap space used by the failed clause, but it cannot undo instantiations made to variables deeper in the data areas. The *trail* records instantiations which will have to be undone on backtracking. The WAM also has *argument registers* used to pass arguments when a predicate is called, and a "push-down list" (actually a conventional *stack*, but that word is overused already) for unification.

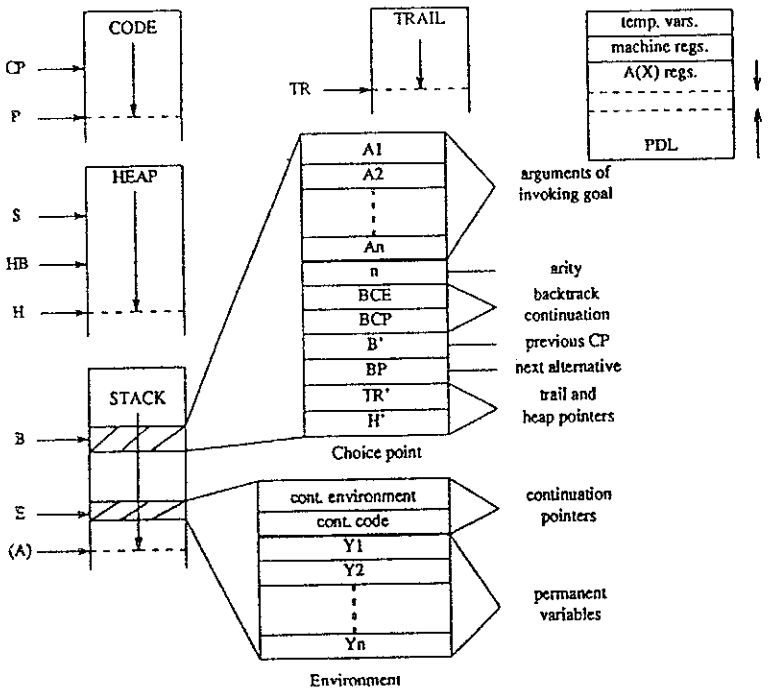

Figure 2.6: Data areas of the Warren abstract machine (WAM).

Assume as before that we have a top-level tracked predicate. When we call it we create a *reordering frame* on the stack; this is a choice point with the BP (next alternative) pointer replaced by a pointer (RP') to a new area called the *reordering heap*, and a new field to count the number of solutions. The reordering heap holds solutions as long as they are needed, and has an extra machine register (RP) pointing to its top. Figure 2.7 shows the features added to the WAM.

The machine copies its (new) RP register into the RP' field of the reordering frame, and makes a *reordering environment* on the stack for the reordered goals. This is an environment with fields for

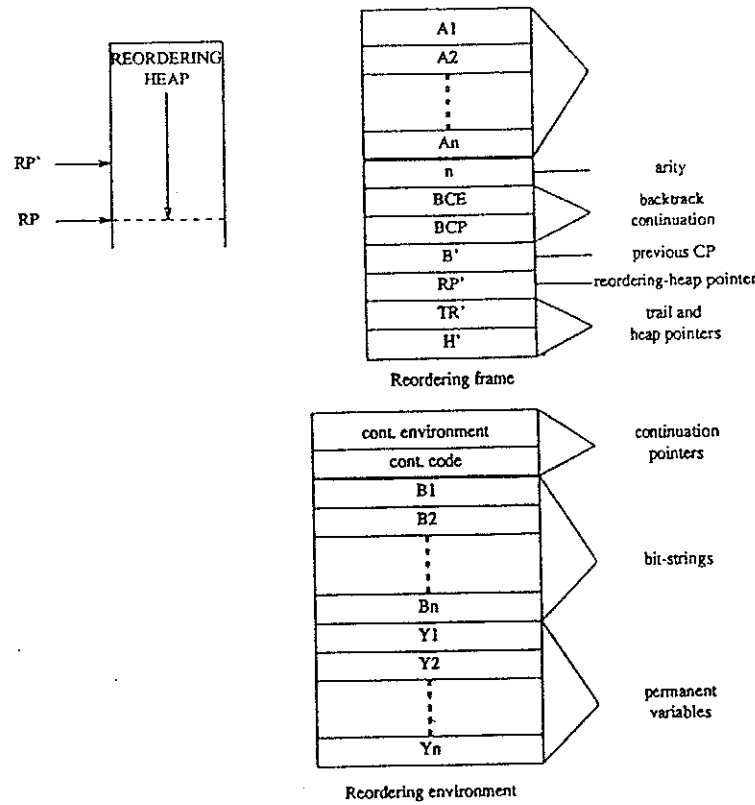

Figure 2.7: Additions to the WAM to support order tracking.

the bit-strings produced by each goal. When the conjunction produces a solution, the machine creates a signature from the bit-strings, combines it with the solutions to form a *solution frame* that it pushes onto the reordering heap, and increments the solution counter of the reordering frame. We force backtracking and find all solutions, discarding the environment as usual.

Now the RP' pointer of the reordering frame points to the first solution frame on the reordering heap. Some extra hardware sorts the solution frames by signature; there are several possible ways to do this. We might create a sorted linked list of solution frames, which would not have to be moved; instead, we sort them such that the first solution is at the top of the reordering heap and the last towards the bottom, allowing us to reclaim some space as solutions are used. When the sorting is done, the reordering frame returns the first solution and decrements its solution counter. On backtracking it fetches the next solution from the reordering heap, decrements its solution counter, and updates RP and RP' as needed. When no solutions remain, the machine backtracks to the previous choice point, as usual.

2.7. Conclusions

We have presented a a collection of methods for reordering goals within Prolog clauses for more efficient execution. After defining levels of equivalence between original and transformed programs, we proceeded to characterize the restrictions on reordering that are necessary to retain a useful degree of equivalence. We showed how additional information, inferred statically or provided by users, is not essential but can provide additional opportunities for reordering. We have extended the Markov-chain method of heuristic reordering to cover full Prolog, described how it can be implemented, and presented some results of applying it to programs. We have also shown how the reordering of results caused by reordering of goals can in some cases be undone, though at some added overhead.

There is scope for additional work. Combining our methods with static analysis techniques developed by other researchers could create further opportunities for reordering. Other techniques for restructuring might act synergistically with reordering, as we discuss in the section in Chapter 4 on future work. Finally, we should integrate our techniques into one system, so that we can provide a program as input and, with slight intervention by the user, receive a reordered, improved program as output.

CHAPTER 3.

SPECULATIVE SEARCH

3.1. Introduction

Assume that we want an optimal solution to the sort of problem typically solved using a branch-and-bound search. As we noted in the introduction to this thesis, if the lower-bound heuristic is not tight, and the upper-bound heuristic is weak or absent, a best-first search will not fit into a machine's memory, and a depth-first search does not make use of all available memory and is prone to go needlessly deep into the search tree. We want to be able to put to good use all memory at our disposal, hoping to reduce the number of nodes a search needs to expand, and avoid the wasted effort of fruitless deep search.

We review various search methods that ameliorate the problem of a weak upper-bound function; most of them also find a use for the extra memory. They are variants on the *IDA** search of Korf [26]. Then we present our *speculative search* method, first informally and then formally, and a variant called *Speculative IDA**, or *SIDA**, that is a generalization of *IDA**. We analyze their behavior and show that it is similar to that of the *MA** variant [7] of Chakrabarti *et al.* Then we show how to add pruning to *SIDA**, and how to run speculative searches on parallel machines. Finally we present some results on the performance of speculative searches.

3.2. Previous work

Various researchers have proposed search methods that combine aspects of depth-first and best-first searches. Pearl [39] describes such methods in his book, particularly noting Ibaraki's *depth-m search* [21]. Our method is related more closely to those we describe next.

3.2.1. Iterative-deepening A^*

Iterative-deepening A^*, or *IDA** [26] simulates an A^* search using a series of depth-first searches. Like A^*, it requires an admissible lower-bound function. It is a variant of *depth-first iterative deepening* (*DFID*). DFID mimics breadth-first search by performing a series of distinct depth-first searches to progressively greater depths. As originally described, *IDA** initially sets an *incumbent* to the (lower-bound) value of the search's root node s, and searches depth-first from s, backtracking when it reaches nodes whose values exceed the incumbent. Such a depth-first search is a *stage* or *iteration*. If a stage finds a solution of value equal to the incumbent, that solution is optimal; if not, *IDA** alters the incumbent, setting it to the smallest value borne by any leaf of the stage (its *periphery*). Then it does the next stage: a new depth-first search from the root, having discarded all results of the previous stage apart from the new incumbent. Figure 3.1 shows three iterations of *IDA** on a search tree: edges to previously-encountered nodes are shown as thicker lines.

The first solution that A^* finds is optimal, assuming that the implementation does not test whether a node is a solution until it tries to expand it. Delaying the test causes nodes to be enqueued that might have been discarded, but it does not increase the number of nodes tried. Similarly, the first solution *IDA** finds is optimal if we do not consider leaves of a stage (with their higher-than-incumbent values) as potential solutions. Nodes first seen in a given stage of *IDA**, apart from the leaves, all necessarily have values no greater than that of the incumbent; thus the search examines new nodes in nondecreasing order of cost — just as does A^*.

Each stage re-expands all nodes expanded in previous stages, keeping none from before. Asymptotically *IDA** expands the same (optimally small) number of nodes as A^* [26]; in practice it is clearly discarding information and wasting effort. Like a depth-first search, *IDA** uses memory linear in the size of the problem, being unable to put any additional memory to good use.

We can choose a larger incumbent than the smallest leaf of the stage just finished. This reduces the fraction of nodes in the new stage that are not themselves new: the search becomes less repetitive. If the values on the peripheries range widely, we may be forced to do this, lest each new stage

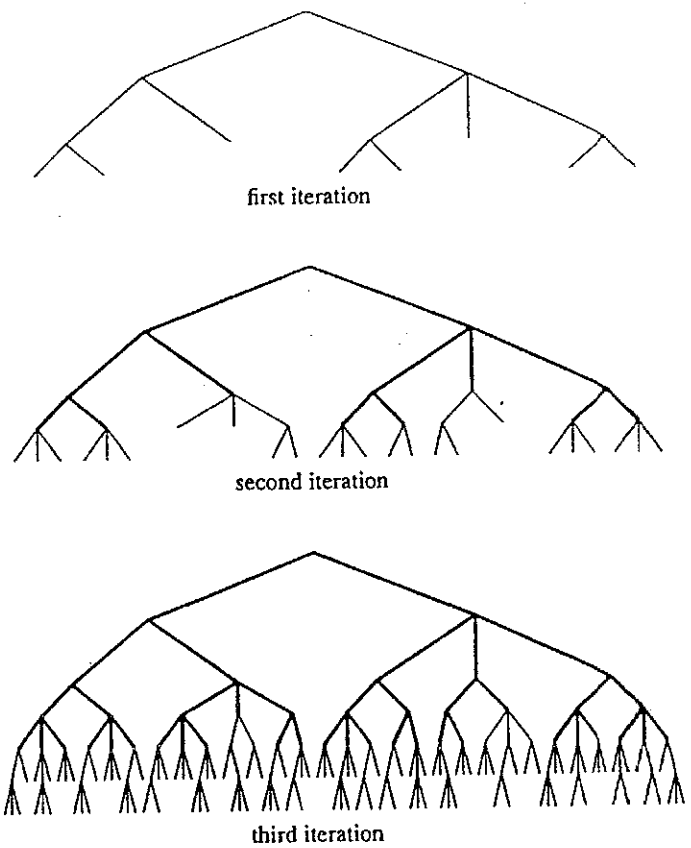

Figure 3.1: *IDA**

have almost no new nodes. The first solution we find is no longer necessarily optimal; for optimality we may have to search all of the current stage once we find a solution, possibly examining a much larger search tree than would have A^*. Korf [27] claims that using a larger incumbent makes *IDA** useful for such problems as traveling salesman (TSP) [18]; we discuss this in a later section of this

chapter.

3.2.2. MA^*

MA^* [7] tries to use as much of a computer's main memory as the user makes available, yet guarantees by a pruning mechanism that it does not try to use more. It resembles an IDA^* performed using an informed depth-first search for each stage, but one that saves some of the nodes it expands.

Consider the tree that IDA^* examines during a stage i given an incumbent inc_i. Call it $tree_i$: it is the set of nodes in the search space that have value inc_i or less, along with their children, which have value greater than inc_i. Given inc_i, MA^* also examines the nodes of $tree_i$ before it proceeds to others, but in a way different than that of IDA^*.

At any point in the search, MA^* has saved on a priority queue (its *open list*) the nodes of a tree rooted at s: a subset of the nodes of $tree_{i-1}$ (initially just s). Each node of the saved tree bears the value of the smallest leaf of $tree_{i-1}$ that is its descendant: this value is inc_i or greater. It also contains this "best-descendant" information for each of its children.

Suppose that MA^* is at an early stage of a search, so that it will do no pruning for some time. We remove from the front of *open* a node n of value inc_i, and generate n's best child m. (If n has no more untried children, it goes onto another priority-queue, the *closed* list.) We expand m's children, back up their values to m, and choose the best child of m to expand next. This repeats until we have expanded a path of nodes from m to the parent of a leaf of $tree_i$, and all of their children. However, we save only the nodes on the path itself. (Note that the saved nodes of the path all had the value inc_i before we backed up the values of their children.) We expand all such paths until $tree_i$ has been examined completely.

Figure 3.2 shows a simple case that does not involve pruning: first the tree of an initial stage of an IDA^* search (the values of the leaves exceed those of interior nodes), then several paths (dashed lines show a new path, dotted ones lead to siblings), and finally the portion of the stage (in this case, all but its leaves) saved by MA^*.

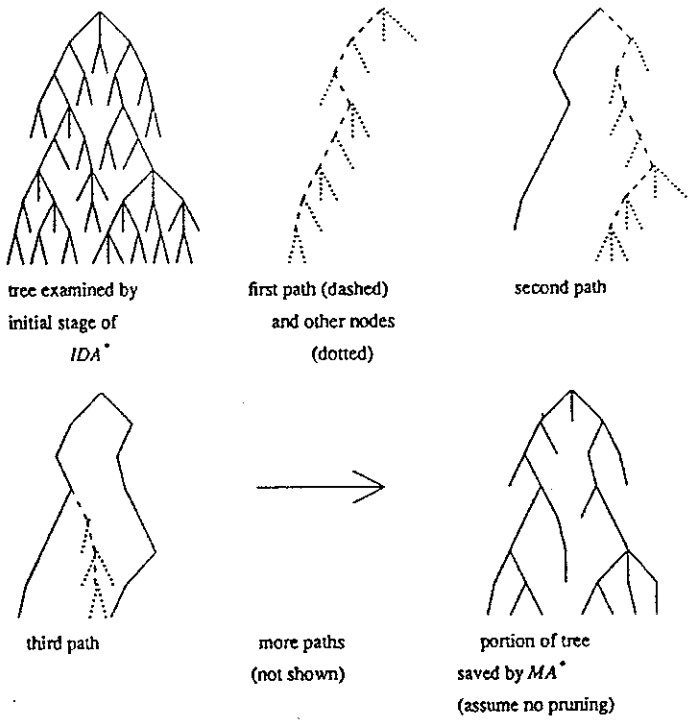

Figure 3.2: MA^* expands a stage of IDA^*, given enough memory.

However, MA^* usually lacks the memory to save all nodes of all paths. It has a fixed limit of *MAX* nodes, set as slightly fewer than the number that can fit in the available memory. When it has saved more than *MAX* nodes, it starts to prune, even though the nodes being pruned may yet lead to optimal solutions. Nodes being pruned have the largest values, and they must have no saved descendants; before pruning a node, we may have to propagate its value back to its ancestors. Pruning continues until all nodes with backed-up values greater than that of the root are deleted: this seems reasonable if arbitrary, and the inventors give no rationale for it.

The backing-up ensures that we keep some information from the nodes we prune. Typically we avoid re-expanding parts of $tree_i$ that we have seen earlier, either by keeping nodes in memory explicitly, or by saving backed-up information that shows expansions from certain nodes to be futile (at least while we are looking at $tree_i$).

Note a detail glossed in the principal paper on MA^* [7]: before we enqueue a node of a path, we must first get the heuristic values of its children. For that we must expand the children in some way and evaluate the lower-bound function for each. For some problems, such as the Fifteen Puzzle [20], a "partial" expansion takes little effort, and perhaps its cost can be ignored, as it appears to have been in the paper. For other problems, expanding a child "far enough" to find its lower bound may be almost as hard as expanding it entirely. For such problems we should expand the children completely, and store but not enqueue all siblings of each path. For a branching-factor b, storing siblings makes a stage use up to b times as much memory as before, but if we do not do this, MA^* expands many nodes twice that IDA^* expands but once.

3.2.3. MREC

MREC [44] is essentially an IDA^* that uses extra memory to save nodes near the root. So long as memory remains, it extends the stored search tree (or graph — it has some features that make it useful for searching graphs) by doing a series of depth-first stages from the root to nodes whose values just exceed an incumbent. The only differences from IDA^* are that it traverses but doesn't re-expand nodes that are already in storage, and that it stores new nodes that aren't. (It stores the saved nodes in a representation of a graph, not in a priority queue.) When memory is exhausted, it searches as before, but adds no more nodes to storage.

With IDA^*, once a node is expanded in any stage but the last, it is re-expanded in every stage thereafter. *MREC* prevents this re-expansion for some nodes near the root. It has the pleasing property of not re-expanding any nodes until it runs out of memory [44]. We conjecture (and the results presented by its inventors seem to confirm this) that it will show little improvement over IDA^* on the types of search at which IDA^* excels, but will fare better in other cases; later in this chapter we discuss reasons for this.

3.3. The algorithm

First we describe our algorithm informally, providing an intuitive feel for how it works. We then proceed to details and an a description in pseudo-code.

3.3.1. An informal description

To begin a best-first search we put an initial node on a priority queue. Then, until we find a solution, we repeat a series of steps: remove the "best" node (that of lowest lower-bound) from the front of the priority queue, expand all of its children, discard those whose values exceed the best known upper bound, insert the rest in the queue, and, if we have just found a tighter upper bound on the solution, prune the queue of any nodes with values exceeding this upper bound.

Our basic technique, speculative branch-and-bound (*SB&B*), also uses a priority queue, and removes and expands the node at its front. This node, or one of its children (depending on circumstances we describe presently), is the *rootlet*. The search does not expand simply the selected node; rather, it uses the rootlet as the root of a depth-first search to some given depth or value of node: a *speculation*.

We speculate from the rootlet using the usual last-in-first-out stack of depth-first search. We keep the *minimum leaf* (or simply *minimum* for short) of the search, and a corresponding *minimum path* from the rootlet to this leaf. At the start of a speculation there is no minimum: the first leaf of the speculation becomes the initial one, and the state of the stack at that point becomes the minimum path. Thereafter the minimum becomes the leaf of smallest value yet seen in the speculation (if there are duplicates we use the first to save effort), and the minimum path the corresponding state of the stack. When the speculation is done, the minimum path leads to the minimum leaf of the speculation.

During the speculation, we propagate the values of the leaves back to the minimum path. By the end of a speculation, each node of the path has been labeled with the value of the best leaf (apart from

the minimum itself) descending from it; thus the nodes of the path bear the best values that we can expect of their descendants should we re-expand them later. (As with MA^*, we also back up to a node the best leaf-values reachable via each child.) We enqueue the nodes of the incumbent path, omitting those of cost greater than the upper bound on the solution. We prune the queue as necessary, remove the best node, and perform another speculation. Figure 3.3 shows a few operations of *SB&B*, emphasizing the points made above.

3.3.2. Details

For *SB&B* we store some extra information in each node we enqueue. Namely, a node has a flag that is set if it is a *virgin*; *i.e.*, none of its children has been generated. The starting node and all minima are initially virgins. A node also has a flag for each child, telling whether or not the child may be expanded as a rootlet when the node is selected from the queue. Finally, for each child eligible for expansion, the node has the backed-up value of its best descendant leaf. (See Figure 3.4.)

If the node removed from the queue is a virgin, it is the rootlet of a speculation. Otherwise, we generate its most-promising eligible child, and speculate from that, returning the parent to the queue, its flag for the chosen child set to ineligibility and the value of the parent suitably altered. (See Figure 3.5.)

We can back up leaf-values during the depth-first search without much difficulty. We alter the stack used for speculation, adding extra slots to its frames; these hold values backed up from each child of the node. We also keep track of how large a section of the current minimum path and the stack is identical, using a counter called *common*.

When a speculation reaches its first minimum, we make the minimum path a copy of the current state of the stack. Because stack and path are identical, we set *common* to the depth of the leaf. Now, when we backtrack a level, we pop the stack and decrement *common*; we do not alter *common* when we generate new nodes on the same level or deeper.

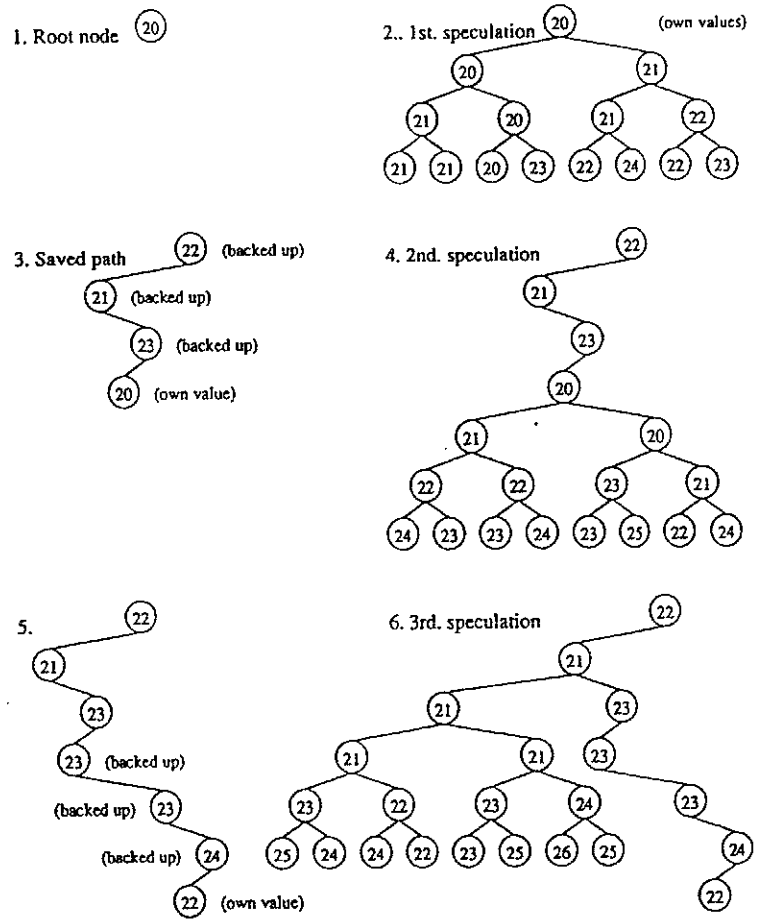

Figure 3.3: Speculative branch-and-bound ($d=3$)

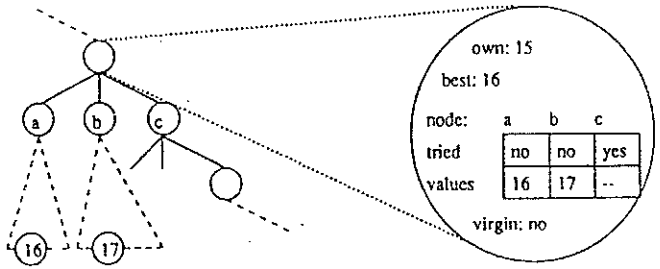

Figure 3.4: Contents of a node of an *SB&B* search.

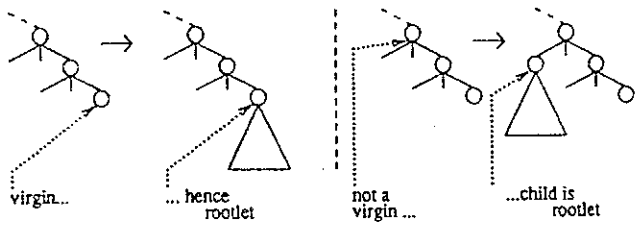

Figure 3.5: A rootlet must be a virgin.

When do we propagate the values of leaves upwards to their ancestors? The rule of thumb is simple: *propagate the value of a node upwards only when you have searched all the node's descendants in the speculation.* Suppose that we speculate to a fixed depth d (see Figure 3.6). We immedi-

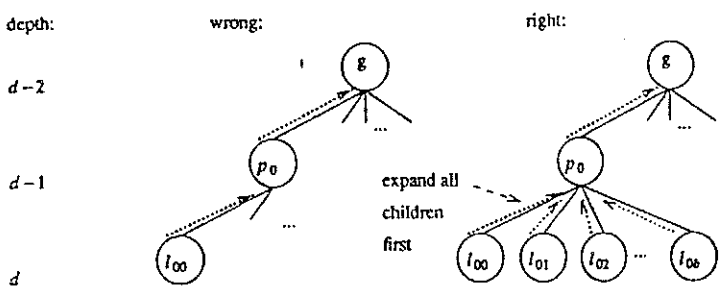

Figure 3.6: When to propagate values upward.

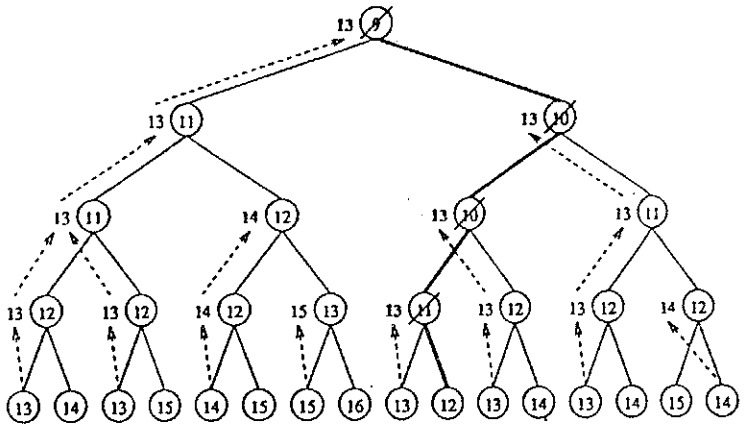

Figure 3.7: Values of leaves are propagated back to the path.

ately propagate the value of leaves l_{i0}, l_{i1}, and so on to their parent p_i at depth $d-1$; however, we needn't propagate the value of p_i to the grandparent g at depth $d-2$ until we've generated all the children l_{ij} of p_i. Only when we are certain of the best value of the children of p_i, and hence the final value p_i takes in this speculation, should we propagate it up to g. The same principle holds for the upper levels of the speculation. Figure 3.7 shows the effects of propagation. Values trickle up from the leaves, each node of the path acquiring the value of its least-costly child (not including children on the path).

Often we will not generate an entire speculation-tree: some nodes may have values exceeding the upper bound, or we may find a solution (not necessarily optimal) shallower than the depth of speculation. In such cases we can simply propagate an "infinite" value back to the parent of the node, showing that this child is not available for further expansion and therefore has no value to propagate upwards. If all leaves or their ancestors have values exceeding the upper bound, the speculation fails.

```
Procedure s_b&b:

(1) [Initialize]
 Queue contains the root node s.
 Set global upper bound u = ∞; best_sol = nil.

(2) [Termination test]
 If Queue is empty, stop; best_sol is the solution.

(3) [Selection]
 Dequeue the node n of lowest cost.

(4) [Speculation]
 u = minimum of u and upper-bound value of n.
 Set solution = speculate(n, Queue, u).
 If solution is nil, go to (2).

(5) [Comparison]
 If cost(solution) < cost(best_sol), best_sol = solution.

(6) [Pruning]
 prune(Queue, u).
 Go to (2).
```

Figure 3.8: Main procedure of *SB&B* algorithm.

3.3.3. A more-formal description

Figures 3.8 and 3.9 present the algorithm for *SB&B* in a more-precise form. We have a priority-queue *Queue*, a global upper-bound u, and assorted procedures for pruning the queue, enqueueing and dequeueing nodes, and so forth.

We speculate by calling a function *speculate* (Figure 3.9 that speculates from a node n or its child, putting the minimum path onto *Queue* and altering u as needed. It translates the rootlet into a single record *node* and an initial frame that it pushes onto a stack. It then uses the stack for a conventional depth-first search (the speculation), altering *node* with each change of frame. Whenever a new minimum appears, it copies the stack to the minimum-path; it calls a procedure *propagate* to pass leaf-values up the stack and the portion of the saved minimum-path that coincides with it. At the end of the speculation, it constructs nodes by combining information from *node* and the saved minimum-path, and puts them on *Queue*.


```
node Function speculate(n, Queue, u)
(1) [Determine the rootlet]
 If n is a virgin,
  Set rootlet = n.
 else
  Set rootlet = best eligible child of n.
  Mark rootlet's record in n as ineligible.
  If n has eligible children,
   Give n the value of its best eligible child;
   Return n to Queue.
 If rootlet is a solution, return rootlet.
(2) [Further initialization]
 Set depth = common = 0; minimum = lb = ∞; solution = nil.
 Translate rootlet into node and an initial frame frame.
 Push frame onto the stack.
(3) [Make a new frame]
 Examine the frame atop the stack.
 Make a frame frame for its first untried child; alter node if necessary.
 If no such frame can be made, go to (7).
 Push frame onto the stack.
 Increment depth; set lb = lower-bound value of node.
(4) [Test for too-large node]
 If lb > u go to (6).
(5) [Test for solution, leaves]
 If node is a solution,
  Set u = lb − 1; set solution = node.
  Propagate values up stack & min_path (using common).
 else if depth = d
  Propagate values up stack & min_path (using common).
  If lb < minimum
   Set minimum = lb.
   Set u = min. of u and upper_bound(node).
   Copy stack to min_path.
   Set common = d.
  Undo last frame & changes to node.
 Pop the stack.
 Decrement common and depth.
 Go to (3).
(6) [New node too large]
 Propagate values up stack & min_path (using common).
 Undo last frame and changes to node.
 Go to (3).
(7) [Cannot make a child]
 Pop the stack.
 If the stack is not empty, go to (3).
(8) [Cleaning up]
 Create nodes from node and min_path.
 Put those of value ≤ u on Queue.
 Return solution [usually nil].
End of speculate.
```

Figure 3.9: Speculation procedure for *SB&B* algorithm

3.3.4. Variants

SB&B speculating to a fixed depth d is analogous to depth-first iterative deepening. We can also make speculations mimic the iterations of IDA^*. To speculate, we search depth-first search from the rootlet in the style of IDA^*, with the initial value of the rootlet as the incumbent. We backtrack on reaching nodes of value higher than the incumbent — just as with IDA^*. In this way we get an analog to IDA^* which we call *Speculative* IDA^*, or $SIDA^*$. We will show later that $SIDA^*$, like IDA^*, is optimal given an admissible heuristic, and typically expands fewer nodes than IDA^*. As with MA^*, the savings is at least linear in the size of the extra memory used: if we search by speculations a space equivalent to a stage, we avoid re-expanding at least all the nodes of that space that are already in storage. $SIDA^*$ can, like the other methods, use an outsized incumbent, at the cost of not finding an optimal solution initially. Figure 3.10 shows a stylized view of a few iterations of $SIDA^*$ on a search tree; it should also provide a good feel for how speculative searches examine a search space. Note the "skeletal" tree formed by the saved paths (thicker lines).

We can also use a local A^* speculation, expanding just enough new nodes each time to limit repetition. This requires that we keep track of how many descendants of each node we have already expanded. One problem: successive speculations tend to grow larger and larger, with more nodes each time, in order to ensure good progress; eventually they may grow too large to fit in a machine's memory.

Varying d from speculation to speculation is also possible, but if we do that we must keep d generally increasing as the search progresses, or many speculations will be entirely subsumed by their ancestors. This may prove useful for parallel versions of *SB&B*, as we shall see.

3.4. Behavior

How does SB&B behave? Namely, how many nodes can we expect it to expand, compared to a best-first search, or to IDA^* or MA^*, and how much memory can we expect it to use? What problems might *SB&B* have in common with the other searches?

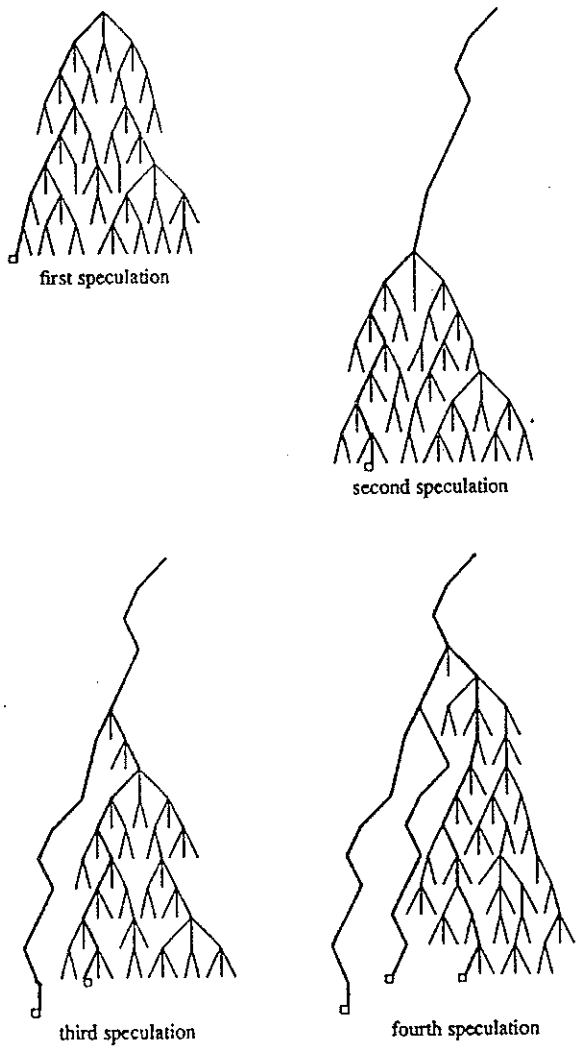

Figure 3.10: A few speculations of $SIDA^*$.

3.4.1. Node-expansions

Consider a simplified view of search, such as Korf used in analyzing the performance of *IDA** [26]. Suppose that our search tree has a uniform branching-factor b (each interior node has b children), and all speculations are to a fixed depth d. Every node seen in the initial speculation is new. But what of the speculations starting from the nodes of the saved path? The worst case is for the $b-1$ speculations spawned when s is re-selected from the priority queue, and the rootlet is one of its children. With these, only the deepest level contains nodes not seen before. Figure 3.11 shows the situation: only the solid lines of the picture lead to new nodes; the dashed lines show repetitions.

Consider the rootlet of a speculation as level 0. A speculation to depth d, given the uniform b, expands $\frac{b}{b-1}(b^d-1)$ nodes [50]. For the worst-case speculations mentioned above, the first $d-1$ levels, or $\frac{b}{b-1}(b^{d-1}-1)$ of these, are old, or almost exactly $\frac{1}{b}$ of the total. Hence, even for the worst-case speculations, at least $\frac{b-1}{b}$ of the nodes are new.

A similar result holds for *IDA** under such a model [26]. This is unrealistic for some real-life searches, and the result is therefore questionable, but we will see that *IDA**, *MA**, and both *SB&B* and *SIDA** all encounter similar problems when the search tree is not so "well-behaved."

It should be obvious that in examining an entire stage, *SIDA** never examines more nodes than does *IDA**. Both methods look at the same number of nodes, but *SIDA** already has the saved nodes

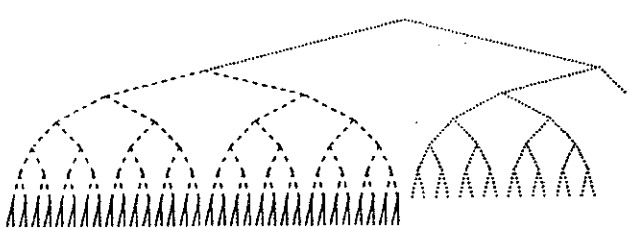

Figure 3.11: Worst-case repetition for $b=2$, $d=6$

stored in memory, and so it does not re-expand them or their ancestors. Neither method typically examines the entire final stage, but *SIDA** looks at it using many small speculations, performed best-rootlet-first, whereas *IDA** uses a single search. Typically *SIDA** will find a solution sooner, just as an informed depth-first search typically does better than a blind one, but this is not certain. If there are many optimal solutions, the purely depth-first search of *IDA** on the final stage may find the "leftmost" one very quickly.

3.4.2. Space

In the worst case, *A** stores a number of nodes exponential in the size of the problem. *SB&B* generates at least $\frac{b}{b-1}(b^d-1)-\frac{b}{b-1}(b^{d-1}-1)=b^d$ new nodes with every speculation, but stores only d of them. Therefore, under this simplified model, *SB&B* stores $\frac{d}{b^d}$ of the nodes stored by *A**. Hence the worst-case use of memory for *SB&B* is still exponential, but with a very small constant. A judicious choice of d (and b, if the nature of the problem makes that practical) can make an *SB&B* search fit in a primary memory that could not have held the nodes of an *A** search. Table 3.1 shows the ratio for various values of b and d. If this does not control the use of memory tightly enough, we can add to *SB&B* (or to *SIDA** specifically) a pruning mechanism similar to that of *MA**, which we

Table 3.1: Effects of b and d on saving of nodes.

b	d	$\frac{\text{nodes expanded}}{\text{nodes stored}}$
2	2	2
	4	4
	8	32
	16	2048
3	2	4.5
	4	20
	8	820
	16	2690420
4	2	8
	4	64
	8	8192
	16	268435456

describe in a later section. Such pruning keeps use of memory under a fixed amount, though at some increase in overhead.

3.4.3. A family of search methods

We can consider *IDA**, *MA** and *SB&B* (*SIDA** specifically, for the purposes of the following discussion) as members of a family of search methods. *MA** is essentially a generalization of *IDA** [7], and *SIDA** with pruning is a generalization of *MA**. The problem is that all of these methods depend for their effectiveness, to some degree, on this assumption: *The number of nodes in the search tree increases exponentially with the depth of the search tree or the lower-bound value of the nodes.*

In other words, we can look at the tree generated in a search, even though the upper-bound function causes pruning and the representation of the problem lowers the branching-factor as we go deeper, and there is still some "*de facto*" branching-factor $\bar{b}$, $\bar{b} > 1$, such that level $i+1$ of the tree has at least $\bar{b}$ times as many nodes as has level i, or such that about $\bar{b}$ times as many nodes have values in the range $(h, h+\delta]$ as in the range $(h-\delta, h]$, for some $\delta>0$ (presumably $\delta \ll h$). If there is this dependence on depth, we can do *SB&B* with speculation to a fixed depth, or do *DFID*, and know that most of the effort of a speculation or a stage is spent on finding new nodes rather than re-expanding ones that we have seen earlier. Similarly, the dependence on value lets us do *IDA**, *MA**, or *SIDA**, and know that most nodes of a stage, path, or speculation are new.

For what problems does the number of nodes increase exponentially with depth or value? Note that the papers on *IDA** [26, 27] and *MA** [7] concentrate on the famous sliding-tile puzzle called the Fifteen Puzzle [20]. The usual lower-bound heuristic based on Manhattan distance behaves very well: values of a parent and its child differ by either zero or two. Hence the search tree has "slices," at least one level thick at their thinnest, containing nodes of equal value. Figure 3.12 is a stylized picture of this. Even a slice one level thick typically has as many nodes as all higher levels of the tree combined, because the average branching-factor for the Puzzle is 2. An upper-bound function could

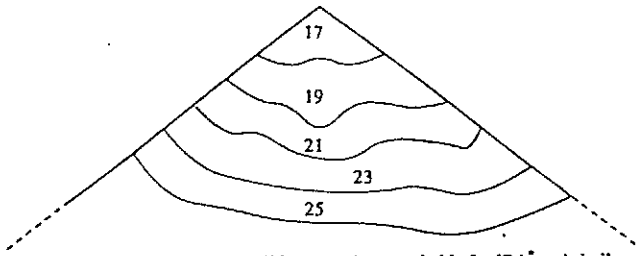

Figure 3.12: "Layers" in a search tree suitable for *IDA** and similar searches.

effectively reduce the branching factor (make $\bar{b}$ smaller), especially deeper in the tree, but apparently no such function exists.

The three methods all do well with the Fifteen Puzzle because the amount of work deep in the search tree overwhelms even the oft-repeated work near its top. (Actually it is a search *graph*, because certain states are reachable in many ways, but the in-degree of nodes is much smaller than the out-degree [26], and researchers using it as a benchmark treat it as a tree, doing some duplicate work.) We have counted the numbers of nodes of each value for tens of instances of the Fifteen Puzzle, and have never found the ratio between the numbers of nodes of value $|h+2|$ and of value $|h|$ to be less than 6.

Korf [27] notes that *IDA** as presented originally [26] performs poorly on the traveling-salesman problem, but claims that by using thresholds sufficiently exceeding the value of the minimum leaf of the previous stage, we can ensure that the next stage has overwhelmingly many new nodes. Analogous claims can be made for the other methods. For *SB&B* or *DFID* we can increase d, for *SIDA** or *MA** we can increase the threshold. We are trying to avoid the following: If there are many discrete values of node, each borne by relatively few nodes, setting the threshold to successive values means that only a small fraction of the nodes explored are new. (Note that all nodes of lower value are re-expanded at each stage.) Therefore the search covers the same ground repeatedly. New speculations of *SB&B* or *DFID*, or new paths of *MA**, will be similarly repetitive.

Smith [47] has studied *relaxation-guided* branch-and-bound searches, particularly for the asymmetric TSP. (A relaxation of a problem is a version of the problem with some constraint removed; a solution to the original problem also solves the relaxation. For instance, the usual relaxation for the asymmetric TSP is the assignment problem (AP) [1], which is the TSP without the constraint that the tour be a single one — solutions to the AP are sets of cycles.) Frequently such methods give excellent lower bounds on the cost of an optimal solution. Smith, using random trees, found that for such searches, the size of a depth-first search tree is *linear* in the depth of the first leaf found (the length of the path to the leftmost solution). This implies that when heuristics are strong, the number of nodes might not increase exponentially with depth or cost. The exponential increase that IDA^* and related searches require would demand an exponential increase of the threshold. In fact, in such cases an informed depth-first search will typically expand about the same number of nodes as a best-first search [53], so that it is clearly the method of choice.

Korf [26] quotes results summarized by Pearl [39] to justify IDA^* (and hence related methods, including $SIDA^*$): if a heuristic used by A^* exhibits constant relative error, the number of nodes generated by A^* (and hence in the space that a method must search) grows exponentially with depth. A heuristic must have logarithmic relative error for the growth to be polynomial, and such heuristics are rare. However, this result assumes that the branching factor b is constant. Hence the model is valid for the Fifteen Puzzle problem, but does not necessarily apply when, as with relaxation methods, b approaches 1 with depth.

Another case when the IDA^* family may fare poorly: suppose that we know exactly how deep in a search tree the solutions will lie. For example, a problem in *NP* might be solved by assembly of prospective solutions one element at a time: a child has one more element of a partial solution than its parent, and solutions all have the same size. Again, an informed depth-first search, in this case to exactly the right depth, is clearly the method of choice.

What happens if the incumbent for IDA^*, the d for *SB&B*, or the change in node-values for $SIDA^*$ is too large? As we mentioned earlier, one might deliberately add a number to such a

parameter, hoping that if it is outsized, most nodes of the next stage or speculation will be new. The problem: though most of the nodes are indeed new, they are perhaps superfluous as well; A^* would not have expanded them. For example, suppose that an optimal solution has value 34: A^* would dequeues only nodes of value 34 or less, and at worst expand all of their children. IDA^* (say), with an outsize incumbent of 38 on its final stage, might well expand most nodes of value 38 or less, and some children of greater value. In such cases, especially if we chose the overestimate to guarantee an exponential increase in the number of nodes, we might find that the "overshoot" alone contains more nodes than A^* would have expanded for the search. Note then that there are two potential problems: repetition of nodes and expansion of superfluous new ones, and that solutions to one problem appear to exacerbate the other.

A good upper-bound function alone for a problem also renders members of the IDA^* family less useful. If speculations or stages go too deep, they are cut off not by their own limits (d or incumbent), but by the upper-bound function. In that case, an informed depth-first search would search about the same space, but without repetitions, and therefore fare better.

3.4.4. Optimality of $SIDA^*$

The argument of Lemma 6.3 of Korf's paper [26], that IDA^* finds a solution of least cost if one exists, holds for $SIDA^*$ as well. $SIDA^*$ also expands all nodes of a given cost before proceeding to those of higher cost, given an admissible heuristic function. The optimality argument then follows the one for IDA^* (Korf's Theorem 6.4).

3.5. Adding pruning

Although *SB&B* uses substantially less memory than does A^*, it can still use an exponentially large amount. Sometimes we cannot adjust b and d to ensure that the search fits in a machine's primary memory: for example, some problems cannot easily be recast to make b large, and b tends to shrink with depth. Too large a d, and the search degenerates into depth-first. Also, it may be hard to estimate precisely the number of distinct nodes we will examine in a search; we may have to bound it

loosely and choose a large d to ensure that the search fits in memory.

SIDA* is the most important case: increasing the depth of speculation destroys optimality, so that we cannot save memory in that way. Therefore we present a version of *SIDA** with pruning, and then discuss its behavior. Pruning is possible with other forms of *SB&B*, but it requires more bookkeeping (such as keeping count of descendants).

3.5.1. Informal description

The original *SIDA** simply enqueues the nodes generated by speculations. Now it must also manage *saved paths*, not just their nodes, in another priority queue. This *Path* queue holds a *header* for each path: which contains information about its path, and the head of a doubly-linked list through the path's nodes. Nodes now contain pointers to their header and their neighbors in their path.

A speculation generates a path. Enqueue its nodes as usual, but also put their header on the *Path* queue. (Initially a header bears the value of its speculation's minimum leaf, but that changes when we prune child paths.) The header keeps a count *Children* of child paths, and three pointers: *Parent* to the parent of the rootlet, *Rootlet* to the rootlet, and *Minimum* to the minimum leaf. After a successful speculation from a node (that is, one that reaches at least one leaf, and therefore returns a path), increment *Children* in the header of the path containing the node.

Disjoint paths keep the algorithm tidy: treat the minimum leaf as distinct from the path leading to it. Enqueue it as usual; it generally becomes the rootlet of its own speculation, and hence the first node of another path. (The pointer from the associated header lets us find it if we must delete it before then.)

As with *MA**, there is a limit *MAX* on usable memory. Start to delete paths when use of memory exceeds *MAX*, The headers on the *Path* queue are in order of suitability for pruning: worst-first and newest-first; the first path the *Path* queue has no child paths. (Because the lower-bound function is admissible, children of a path have values no less than that of the path itself, and they are newer — hence they appear earlier in the *Path* queue than their parent.) Dequeue the first header,

and, following the links down its path, delete its elements from the main queue. (Implement the queue so that it allows arbitrary elements to be deleted: a splay tree [45] with added parent pointers [22] suffices.) If that minimum leaf associated with a path has not, in this case, yet become the rootlet of a later speculation, delete it as well.

To finish deleting a path, alter the corresponding child-value stored in its parent node, and, if necessary, the parent's own value. Decrement the count of active child paths in the header of the path containing the parent, using the pointer from the parent to its path's header. If the path being deleted is the last child of the parent's path, update the value of the parent path, scanning through its nodes and copying the best value to the header. If this changes the value of the path, remove it from the Path queue and re-insert it before all nodes of equal value: because it no longer has child paths, it is at least as eligible for deletion as any other path of its value. Paths generated later are nearer the front of the Path queue: they are newer, their values are no lower, and therefore they are prone to be deleted sooner.

Note the following:

(1) Alter a path's value only after all of its child paths are deleted. Before then, the value may be much lower than those of the child paths. This doesn't matter because the parent cannot be pruned at that point.

(2) There are now two types of pruning. One is irrevocable: removal of nodes from the main queue because they exceed the global upper bound, just as in a best-first search. The other is not: the pruning of paths as just described, To do the first type, check the Path queue first and prune irrevocably any paths whose headers have costs exceeding the global upper bound, *before* removing any nodes from the main queue. Then remove any nodes in the main queue that cost more than the global upper bound. Using doubly-linked lists for paths facilitates this pruning: when pruning a node, simply revise its path to skip it.

(3) When all the children of a node have been expanded, do not return it to the main queue. Do not, however, delete it unless it exceeds the global upper bound: pruning might delete some of its

child paths and let it be expanded again. Leave it in the linked list of its path, awaiting possible re-expansion: MA^* must also store such "dead" nodes [7].

(4) Suppose that we speculate from a node n of a path that was earlier stripped of child-paths. Must we propagate this path's value forward to the newly re-expanded path? No. In fact, for $SIDA^*$, we can speculate using the value backed-up to n. This generates a single (though longer) saved path to replace the ones pruned earlier.

3.5.2. A more-formal description

Figure 3.13 describes more formally $SIDA^*$ with pruning. The *speculate* function is much like that for ordinary *SB&B* or $SIDA^*$, except that it creates a path header and sets the added pointers in the nodes to make the doubly-linked list representing the path.

3.5.3. Behavior

It should now be obvious that $SIDA^*$ with pruning is a generalization of MA^*. Instead of searching only one level deep, $SIDA^*$ goes all the way to the bottom of an IDA^*-stage. In earlier stages, MA^* may expand fewer nodes than would $SIDA^*$, because MA^* quickly fills the allotted memory with nodes, whereas $SIDA^*$ takes longer. Recall that the savings in searching a stage with $SIDA^*$ or MA^* instead of IDA^* is proportional to the number of nodes in storage. On the other hand, $SIDA^*$ saves the best path of a speculation, which gives it an element of an informed search.

3.6. Parallel *SB&B*

A parallel version of *SB&B* is almost obvious. Assume a machine of many processors that communicate by passing messages over a network. Have one processor, the *server*, maintain the priority queue and send rootlets to other processors, the *clients*, which speculate from them and return paths to the server, which enqueues them. Clients also send new values of the global upper bound u back to the server, which periodically sends (broadcasts, if the network can do that) the best u it knows to the clients. A client returns an empty path if a speculation fails, and a prospective solution

Procedure pruning_s_b&b(Min, Max):

(1) [Initialize]
Queue contains the root node *s*.
PathQueue is empty.
Set global upper bound $u = \infty$; *best_sol* = *nil*.

(2) [Termination test]
If *Queue* is empty, stop; *best_sol* is the solution.

(3) [Selection]
Remove from *Queue* the node *n* of lowest cost.

(4) [Speculation]
Set u = minimum of u and upper-bound value of *n*.
Set *solution* = speculate(*n*, *Queue*, u, *header*).
If *header* is not *nil*,
  Increment count of *Children* in header of *n*'s path.
  Set *header.Children* = 0.
  Enqueue header on *PathQueue*.
If less than *Max* memory is in use, go to (8).

(5) [Pruning to conserve memory]
Get header *h* at the head of *PathQueue*.
Delete from Queue all nodes of *h*'s path.
Propagate value of *h* to its parent *p*; adjust *p* as needed.
Decrement *Children* in header of *p*'s path *P*.
If *P.Children* > 0,go to (7).

(6) [Updating parent path]
Start from *P* and scan its path for the smallest cost *c*.
If $c > P.cost$
  $P.cost = c$.
  Remove *P* from *PathQueue*; restore with its new value.

(7) [Test to stop pruning]
If more than *Min* memory is in use, go to (5).

(8) [Loop]
If *solution* is *nil*, go to (2).

(9) [Comparison]
If cost(*solution*) < cost(*best_sol*), set *best_sol* = *solution*.

(10) [Pruning to eliminate costly nodes]
Remove headers *h*, *h.value* $\geq u$, and path [as in (5), (6)]
For every node *n* of value $\geq u$:
  Remove *n* from *Queue*.
  Adjust pointers of neighbors of *n* on path, to skip *n*.
Go to (2).

Figure 3.13: $SIDA^*$ with pruning.

when a speculation finds one. The server handles the queue much as in the serial version, adding the nodes sent by the clients, pruning nodes based on the information they send, keeping a best-known

solution. If a single server is not sufficient, we can use several, each sending rootlets to a subset of processors and receiving paths from other processors to ensure a roughly equal load on all servers.

We have implemented parallel *SB&B* (without pruning) on a 16-node Intel IPSC/2 hypercube, using the hypercube's host processor as the server. The best speedups were near-linear (up to 15-16 times serial on 16 nodes plus the host), but speedup seemed to depend a great deal on *d*; more-typical speedups were about half that. We have yet to compare our implementation with other such searches, such as the parallel *IDA** of Rao *et al.* [42], so we give no further results.

3.7. Results

Our results show that if we use the same heuristic and the same amount of "overshoot," *SIDA** expands fewer nodes that *IDA**. It should be obvious that when both examine all the nodes of an *IDA** stage, *SIDA** uses its stored nodes to save effort and always expands fewer. *IDA** sometimes solves a problem more quickly, however, because its overhead is lower and because it is possible that it will stumble on a solution early in its final stage. Table 3.2 shows some results for sets of 100 randomly-generated problems, with given overshoot in the value of the incumbent (written "o = 20," for example) or fixed depth of speculation ("d = 12"). We give average number of nodes and standard deviation for each set. Instances of the jobshop scheduling problem are for 4 jobs, each having 4 tasks, scheduled on 4 machines (hence "4/4/4"); they were generated by adding tasks of random length (in this case, nonzero but less than 64) to a job, checking to ensure that no contiguous tasks

Table 3.2: Comparison of methods for 100 random problems.

Problem	Search	|*nodes*|	σ
4/4/4 jobshop	*IDA**	148859139	296580397
	*IDA**, o = 20	23536998	34971060
	*IDA**, o = 30	20381512	29519058
	*SIDA**, o = 20	20685240	30757773
	*SIDA**, o = 30	20034856	30950953
	blind DFS	15932349	19437159
	SB&B, d = 12	36242055	44740843
	SB&B, d = 13	40707373	44376572
8-puzzle	*IDA**	5864	9218
	*SIDA**	4465	8812

demanded the same processor. They were solved using a naive heuristic, showing strengths and weaknesses peculiar to various methods. Using a weak heuristic makes the number of nodes increase exponentially with value; however, all solutions are at the same depth. Conventional *IDA** fares poorly because of the many different values that nodes can take; adding an "overshoot" to the incumbent value reduces the number of nodes, but *SIDA** still fares better under the same advantage. Note that *SB&B* with fixed *d* does poorly, because the branching-factor drops sharply (from 4 near the root of the tree to 1 just before the leaves). For this sort of problem, even a blind depth-first search fares better than any method of the *IDA** family, because it repeats no nodes and cannot search past the level where the solutions lie. In Appendix B we give the code for solving jobshop scheduling using *SIDA**; the Fifteen Puzzle code in Appendix C can be used for Eight Puzzle as well.

Table 3.3 shows some results for instances of the Fifteen Puzzle presented by Korf [26] and also used as examples in the paper on *MA** [7]. (Code for our *SIDA** version is in Appendix C.) Usually *SIDA** performs better than *IDA** and not as well as *MA** with no additional memory; however, the *MA** paper does not appear to count its examinations of child nodes as node-expansion. Also, because we do not at this writing have the pruning version of *SIDA** working, to ensure that the search would fit in machine memory, we had to collect these data using *SIDA** with an overshoot of 10 added to the threshold: speculations are expanding needless nodes deep in the tree. Furthermore, for the results to be comparable to those for *MA**, *SIDA** should use informed depth-first search in its speculations.

Table 3.3: Instances of the 15 Puzzle from Korf's paper.

No.	*IDA**	*MA**(0)	*SIDA**, overshoot = 10
15	543598067	270016222	298196963
30	2196593	898573	1682282
40	63276188	34010298	54167881
45	6158733	2396522	8309799
50	63036422	17989651	62453094
55	927212	454994	391449
85	2725456	1426580	2154883
100	67880056	19221994	70243385

3.8. Conclusions

We have presented a new algorithm, *speculative branch and bound* (*SB&B*), for performing heuristic search in restricted memory. It is analogous to depth-first iterative deepening (*DFID*). We have also presented a variant, *speculative IDA** (*SIDA**), analogous to *IDA**. Both of these use substantially less memory than conventional branch-and-bound or A^* methods, but because both can still generate a number of nodes exponential in the size of the problem, we have also devised a pruning mechanism, analogous to that of *MA**, to restrict use of memory. Both *SB&B* and the *IDA** variant can be run on loosely-connected parallel processors.

SIDA* expands fewer nodes than *IDA**. It appears to be competitive with *MA**, but we need to run comparisons for a variety of problems, and do a theoretical analysis that is more rigorous.

All of these methods (*DFID*, *IDA**, *MA**, *SB&B*, *SIDA**) perform best on search spaces for which the number of nodes increases exponentially with depth (*DFID* and *SB&B*) or heuristic value (the others). This requires a heuristic with constant relative error (which is usual), and a representation of the problem that ensures that the branching factor of the search tree is roughly constant with depth.

CHAPTER 4.

CONCLUDING REMARKS AND FUTURE WORK

4.1. Reordering goals in Prolog programs

We have shown that if we observe certain restrictions and especially if we are willing to accept some changes in the behavior of a Prolog program, we can reorder it in the hopes of improving its performance. Improvements in performance can be several orders of magnitude for simple programs with execution that resembles that of a query to a database, but are usually much less with "impure" Prolog code.

We have examined programs from diverse sources. Most of these are not toy programs or "benchmarks" in the usual sense of being contrived to test particular abilities of a system, but rather production code from large software projects. Typically they offer few opportunities for reordering. It seems that programmers take extraordinary measures to ensure determinism in their code, so that few clauses can be reordered without breaking restrictions imposed by the cut, implication, or various built-in predicates. usually even a predicate with many arguments will have a single legal mode. Reordering, except for database-style programs, is probably not to be relied on as the chief form of optimization for a program, and for many programs any reordering will be impossible.

It may be possible to create opportunities for reordering by applying other forms of restructuring to Prolog code. *Folding* and *unfolding* [51] look promising for this. Unfolding a clause is replacement of a calling goal with an instance of the clause it calls:

```
% Clause containing calling goal:
foo(X,Y):- bar0(X,Z), bar1(Y,Z), bar2(Y,X).

% A clause of the predicate being called:
bar1(A,B):- a(A), b(B,C), c(C,A).

% Unfolded version:
foo(X,Y):- bar0(X,Z), a(Y), b(Z,C), c(C,Y), bar2(Y,X).
```

To unfold a call to a predicate of several clauses, we may need to replace the original clause with a set of new ones, one for each unfolding. Folding is the reverse process; it is often combined with defining a new predicate, perhaps one of a single clause:

```
% Original clause:
a(T,U,V):- b(U,W), c(T,U,X), d(W), e(X), f(U,V).

% Define a new predicate of one clause:
a2(A,B):- b(B,C), c(A,B,D), d(C), e(D).

% Folded version:
a(T,U,V):- a2(T,U), f(U,V).
```

By unfolding we can create longer clauses which may have more opportunities for reordering. The process could produce an explosion in the size of code if allowed to proceed unchecked, however, and controlling it might be difficult.

We might use successive unfoldings and foldings to minimize the effects of fixed predicates. First we fold into its caller the clause(s) of a predicate that bears a side-effect:

```
% Original clause:
a(X,Y):- b(X,Y), c(X), d(Y).

% c/1 has a side-effect:
c(A):- e(A,_).
c(A):- f(A,B,C), write(B), g(C,A).

% Replacements:
a(X,Y):- b(X,Y), e(X,_), d(Y).
a(X,Y):- b(X,Y), f(X,Z,W), write(Z), g(W,X), d(Y).
```

Then we can define new predicates free of the stigma of fixity:

```
% Definitions:
a2(A,B,C,D):- b(B,C), f(B,D,A).

a3(A,B,C):- g(A,B), d(C)

% a/2 after folding of second clause:
a(X,Y):- b(X,Y), e(X,_), d(Y).
a(X,Y):- a2(W,X,Y,Z), write(Z), a3(W,X,Y).
```

By repeating this process we can force side-effects to the top of the program, where they will cause as little fixity as possible. We do not yet know how useful this is.

4.2. Speculative search

We have developed new methods for heuristic search in restricted memory: speculative branch-and-bound, which is analogous to depth-first iterative deepening, and *SIDA**, analogous to *IDA**. *SIDA** appears to be about as useful as another search, *MA** that is also related to *IDA**. Both *MA** and *SIDA** can make use of all of a computer's available main memory, and save a number of node expansions over *IDA** that is proportional to the amount of memory available. Speculative searches can also be run on loosely-connected parallel processors.

All the methods of this family of searches — *IDA**, *MA**, and *SIDA** — seem to be best suited for searches for which the lower-bound heuristic is not especially tight, the upper-bound heuristic weak or absent, and the heuristic values of the nodes in the search form a fairly small set. It seems possible that searches of this type have a niche to fill.

Future work on speculative search should help establish the utility of such search methods. Better comparisons with *MA** are essential, which will require reproducing the results of the *MA** paper, which are ambiguous. We need a way to better control the grain size for parallel implementations of speculative search, particularly for *SIDA**. We need to compare parallel *SIDA** with other parallel searches such as the parallel *IDA** of Rao *et al.* [42]. Also some tighter theoretical bounds on expansion of nodes and use of memory should be possible.

REFERENCES

[1] E. Balas and P. Toth, "Branch and Bound Methods," pp. 361-402 in *The Traveling Salesman Problem*, ed. D. B. Shmoys, Wiley, New York, NY, 1985.

[2] A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence*, vol. 1, 2, and 3, William Kaufmann, Los Altos, CA, 1981, 1982.

[3] M. Bruynooghe, *et al.*, "Abstract Interpretation: Towards the Global Optimization of Prolog Programs," *1987 Symposium on Logic Programming*, pp. 192-204, IEEE, Sept. 1987.

[4] M. Bruynooghe and G. Janssens, "An Instance of Abstract Interpretation Integrating Type and Mode Inferencing," *Fifth International Conference and Symposium on Logic Programming*, pp. 669-683, Aug. 1988.

[5] M. Bugliesi and F Russo, "Partial Evaluation in Prolog: Some Improvements About Cut," *1989 North American Conference on Logic Programming*, pp. 645-660, BuR89.

[6] M. Carlsson and J. Widen, SICStus Prolog User's Manual, SICS Research Report R 88007, Swedish Institute of Computer Science, Kista, Sweden, Feb. 1988.

[7] P. P. Chakrabarti, S. Ghose, A. Acharya, and S. C. de Sarkar, "Heuristic Search in Restricted Memory," *Artificial Intelligence*, no. 41, pp. 197-221, 1989.

[8] J. H. Chang, A. M. Despain, and D. DeGroot, "AND-Parallelism of Logic Programs Based on A Static Data Dependency Analysis," *Proc. COMPCON Spring*, pp. 218-225, IEEE, 1985.

[9] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, 1981.

[10] H. Coelho, J. C. Cotta, and L. M. Pereira, How to Solve It in Prolog, Laboratorio Nacional de Engenhara Civil, Lisbon, Portugal, July 1982.

[11] S. K. Debray and D. S. Warren, "Automatic Mode Inference for Prolog Programs," *1986 Symposium on Logic Programming*, pp. 78-88, IEEE, Sept. 1986.

[12] S. K. Debray and D. S. Warren, "Detection and Optimization of Functional Computations in Prolog," *Third International Conference on Logic Programming*, pp. 490-504, July 1986.

[13] S. K. Debray, The SB-Prolog System, Version 2.3.2: A User Manual, Tech. Rep. 87-15, Department of Computer Science, University of Arizona, Tucson, Arizona, Mar. 1988.

[14] D. DeGroot, "Restricted AND-Parallelism," *Proc. Int'l Conf. on Fifth Generation Computers*, pp. 471-478, ICOT and North-Holland, Nov. 1984.

[15] D. DeGroot, "Restricted AND-Parallelism and Side-Effects," *1987 Symposium on Logic Programming*, pp. 80-89, IEEE, Sept. 1987.

[16] D. DeGroot, "Restricted AND-Parallelism and Side-Effects in Logic Programming," in *Supercomputers and AI Machines*, ed. K. Hwang and D. DeGroot, McGraw-Hill, 1988.

[17] J. J. Dongarra, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

[18] M. R. Garey and D. S. Johnson, *Computers and Intractability*, Freeman, San Francisco, CA, 1979.

[19] M. V. Hermenegildo, "Relating Goal Scheduling, Precedence, and Memory Management in AND-parallel Execution of Logic Programs," *Fourth International Conference on Logic Programming*, pp. 556-575, May 1987.

[20] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, MD, 1978.

[21] T. Ibaraki, "*m*-depth search in branch-and-bound algorithms," *Int'l. J. of Computer and Information Sciences*, vol. 7, no. 4, pp. 315-373, Plenum Press, 1978.

[22] D. W. Jones, "An Empirical Comparison of Priority-Queue and Event-Set Implementations," *CACM*, vol. 29, no. 4, pp. 300-311, April 1986.

[23] T. Kanamori and K. Horiuchi, "Construction of Logic Programs Based on Generalized Unfold/Fold Rules," *Fourth International Conference on Logic Programming*, pp. 744-768, May 1987.

[24] J. G. Kemeny and J. L. Snell, *Finite Markov Chains*, D. Van Nostrand Company, Inc., New York, NY, 1965.

[25] H. J. Komorowski, "Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the case of Prolog," *Ninth Annual ACM Symposium on the Principles of Programming Languages*, pp. 255-268, ACM, 1982.

[26] R. E. Korf, "Depth-First Iterative Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*, vol. 27, pp. 97-109, North-Holland, 1985.

[27] R. E. Korf, "Optimal Path Finding Algorithms," pp. 223-267 in *Search in Artificial Intelligence*, ed. V. Kumar, Springer-Verlag, New York, 1988.

[28] R. Kowalski, *Logic for Problem Solving*, North-Holland, 1979.

[29] G. J. Li and B. W. Wah, "How Good are Parallel and Ordered Depth-First Searches?," *Proc. Int'l Conf. on Parallel Processing*, pp. 992-999, IEEE, Aug. 1986.

[30] D. Maier and D. S. Warren, *Computing with Logic*, Benjamin/Cummings Publishing, Menlo Park, CA, 1988.

[31] H. Mannila and E. Ukkonen, "Flow Analysis of Prolog Programs," *1987 Symposium on Logic Programming*, pp. 205-214, IEEE, Sept. 1987.

[32] C. S. Mellish, "An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter," pp. 99-106 in *Logic Programming*, ed. K. Clark and S. A. Tarnlund, Academic Press, New York, NY, 1982.

[33] C. S. Mellish, "Some Global Optimizations for a Prolog Compiler," *Journal of Logic Programming*, vol. 1, pp. 43-66, 1985.

[34] C. S. Mellish, "Abstract Interpretation of Logic Programs," *Third International Conference on Logic Programming*, pp. 463-474, July 1986.

[35] A. Mycroft and R. A. O'Keefe, "A Polymorphic Type System for Prolog," *Artificial Intelligence*, vol. 23, pp. 295-307, 1985.

[36] L. Naish, *Negation and Control in Prolog*, Springer-Verlag, Berlin, 1986.

[37] R. A. O'Keefe, "On the Treatment of Cuts in Prolog Source-level Tools," *1985 Symposium on Logic Programming*, pp. 68-72, IEEE, July 1985.

[38] R. A. O'Keefe, *The Craft of Prolog*, MIT Press, Cambridge, MA, 1990.

[39] J. Pearl, *Heuristics--Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading, MA, 1984.

[40] F. Pereira, C-Prolog User's Manual, Version 1.5, Edinburgh Computer Aided Architectural Design, Edinburgh, Scotland, Sept. 1986.

[41] L. M. Pereira, F. C. N. Pereira, and D. H. D. Warren, User's Guide to DECsystem-10 Prolog, Sept. 1978.

[42] V. N. Rao, V. Kumar, and K. Ramesh, "A Parallel Implementation of Iterative-Deepening-A*," *Proc. 6th. National Conf. on Artificial Intelligence*, pp. 178-182, Morgan Kaufman, Los Altos, CA, July 1989.

[43] P. Van Roy and A. M. Despain, "The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler," *1990 North American Conference on Logic Programming*, pp. 501-515, Oct. 1990.

[44] A. K. Sen and A. Bagchi, "Fast Recursive Formulations for Best-Fist Search That Allow Controlled Use of Memory," *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 297-302, IJCAI, Inc., Detroit, MI, Aug. 1989.

[45] D. D. Sleator and R. E. Tarjan, "Self-Adjusting Binary Search Trees," *JACM*, vol. 32, no. 3, pp. 652-686, 1985.

[46] D. E. Smith and M. R. Genesereth, "Ordering Conjunctive Queries," *Artificial Intelligence*, vol. 26, pp. 171-215, 1985.

[47] D. R. Smith, "Random Trees and the Analysis of Branch-and-Bound Procedures," *J. of the ACM*, vol. 31, no. 1, pp. 163-188, 1984.

[48] Z. Somogyi, "A System of Precise Modes for Logic Programs," *Fourth International Conference on Logic Programming*, pp. 769-787, May 1987.

[49] L. Sterling and E. Y. Shapiro, *The Art of Prolog*, MIT Press, Cambridge, MA, 1986.

[50] M. E. Stickel and W. M. Tyson, "An Analysis of Consecutively Bounded Depth-First Search with Applications in Automated Deduction," *9th IJCAI*, pp. 1073-5, 1985.

[51] H. Tamaki and T. Sato, "Unfold/Fold Transformation of Logic Programs," *Second International Conference on Logic Programming*, pp. 127-138, 1984.

[52] J. D. Ullman, *Principals of Database Systems, 2nd Edition*, Computer Science Press, 1988.

[53] B. W. Wah and C. F. Yu, "Stochastic Modeling of Branch-and-Bound Algorithms with Best-First Search," *Trans. on Software Engineering*, vol. SE-11, no. 9, pp. 922-934, IEEE, Sept. 1985.

[54] D. H. D. Warren, "Efficient Processing of Interactive Relational Database Queries Expressed in Logic," *Proc. 7th Int'l Conf. on Very Large Data Bases*, pp. 272-281, M. Kaufman Inc., 1981.

[55] D. H. D. Warren, An Abstract Prolog Instruction Set, Technical Note 309, SRI International, Menlo Park, Calif., Oct. 1983.

[56] D. H. D. Warren, Applied Logic -- Its Use and Implementation as a Programming Tool, Technical Note 290, SRI International, Menlo Park, Calif., June 1983.

[57] D. H. D. Warren, "Optimizing Tail Recursion in Prolog," pp. 77-90 in *Logic Programming and its Applications*, ed. D. H. D. Warren, Ablex, 1986.

[58] R. Warren, M. Hermenegildo, and S. K. Debray, "On the Practicality of Global Flow Analysis of Logic Programs," *Fifth International Conference and Symposium on Logic Programming*, pp. 684-699, Aug. 1988.

APPENDIX A.

RESTRUCTURER FOR SIMPLE PROLOG CODE

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% transform.pl
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%     Program-restructurer for nearly "pure" Datalog-style Prolog
%
%          Main file: execution causes others to be read in.
%     This uses a hacked version of C-Prolog 1.5 that can call
%     object code from routines written in C or Fortran.
%

transform(Input_file,Output_file):-
      initconst,                        % set up the constant-counter
      get_other_predicates,                   % read other files of program
      scan_file(Input_file),                  % to get constants, pred. names
      get_domains,                            % #solns, etc.
      update_dependences,                     % remove dep. on ground terms
      tell(Output_file),                      % output goes to Output_file
      !,                                % shouldn't need this
      write('% Restructured by Markov-chain method:'),
      nl,
      [markov],                         % read it in only if necessary
      load([['lib.o']]),                % load the C program
      procedure(matrix(float_vector,float_vector,int,float_ptr)),
      markov,
      told(Output_file).                      % close Output_file

update_dependances:-
      assertz(ground(last)),                  % a hack
      repeat,
            ground(Ground),                   % get ground term
            retractall(needs(_,Ground)),      % remove all related "needs"
            Ground = last,
      retract(ground(last)).                  % probably could have left it
update_dependences:-
      !.

get_domains:-
      assertz(data(last)),                    % a hack
      repeat,
            data(Name),                 % get a predicate
            get_for_one(Name,Method),         % do it for one predicate
            Name=last,                  % out of data-predicates now
      retract(data(last)).                    % get rid of end-marker

get_for_one(last,_):-!.
get_for_one(Name,Method):-
      current_predicate(Name,F),              % find the form of its functor
      bagof(A,F^A^                            % do it the slow way:
            ( call(F),                  % find all answer-tuples for
```



```
            F=..[_|A]),                 %     the predicate F &...
            L ),                              % & put them in a list of lists
            countup1(L,_,_,0,An,_,D),         % # of answers, domain sizes
            assertz(g(Name,An,D)),            % assert "g" fact
            find_prob(F,An).                  % assert probabilities of each

find_prob(Functor,Succ):-
      functor(Functor,Name,Arity),            % get arity
      find_prob2(Arity,Mode,Succ,Prob,1,_),         % construct modes, find probs.
      F=..[Name|Mode],
      assertz(prob(F,Prob)),                  % assert it
      fail.                             % force backtracking
find_prob(_,_).                               % force success

find_prob2(0,[],Succ,Prob,Denom,inst):-       % Fully-inst. (last arg. unset)
      Prob is Succ / Denom,             % no "correction" needed
      !.                                % red cut to prevent dupl. ans.
find_prob2(0,[],Succ,Prob,Denom,un):-         % set last arg. means partial
      Prob is Succ / (Succ + Denom),          % appropriate formula
      !.
find_prob2(Count,[u|Args],Succ,Prob,Denom,_):-  % argument uninstantiated
      Count2 is Count - 1,                    % count down: denom. unaffected
      find_prob2(Count2,Args,Succ,Prob,Denom,un).
find_prob2(Count,[i|Args],Succ,Prob,Denom,F):-  % arg. inst.; propagate flag
      Count2 is Count - 1,
      constants(K),                           % get # of constants
      Denom2 is Denom * K,                    % increase denominator
      find_prob2(Count2,Args,Succ,Prob,Denom2,F).

countup1([],L,L,L0,L0,T,T):-!.
countup1([H|T],ThereL,BackL,L0,Lb,Tuple,Btuple):-
      L2 is L0 + 1,
      countup2(H,ThereL,NewL,Tuple,Ntuple),
      countup1(T,NewL,BackL,L2,Lb,Ntuple,Btuple).

countup2([],[],[],[],[],[]):-!.
countup2([H|T],[HH|TT],[NH|NT],[Tl|Tr],[Tn|Tb]):-
      intersect(H,HH,NH,T1,Tn),
      countup2(T,TT,NT,Tr,Tb).

intersect(S,[],[S],T1,Tn):-
      (var(T1) -> Tn is 1;
      Tn is T1 + 1),
      !.
intersect(S,[S|T],[S|T],T1,T1):- !.
intersect(S,[U|T],[U|V],T1,Tn):-
      intersect(S,T,V,T1,Tn).

scan_file(Input_file):-
      see(Input_file),                  % make it the input stream
      repeat,                                 % until end_of_file
            read(Clause),                     % get next clause of file
            get_info(Clause),           % extract its information
            Clause = end_of_file,             % stopping condition
      seen.

get_info(end_of_file):-!.                     % pass it but don't assert it
get_info((Head:-Body)):-                      % full clauses
      assertz((Head:-Body)),                  % put it in the database
```

```
    !,
        Head=..[Name|Args],                 % get name & list of arguments
        (data(Name) -> true ;               % if it hasn't been done...
            assertz(data(Name))),           % ...assert that it's data
        get_constants(Args),                % extract the constants
        get_goals(Name,Body).               % get goals & extract constants
get_info(Head):-                        % facts
        assertz(Head),                      % put it in the database
        Head=..[Name|Args],                 % get name & list of arguments
        (data(Name) -> true ;               % if it hasn't been done...
            assertz(ground(Name)),          % ...assert that it's ground
            assertz(data(Name))),           % ...assert that it's data
        get_constants(Args).                % extract the constants

get_goals(Name,(Conj1,Conj2)):-  .              % conjunctive
        !,
        get_goals(Name,Conj1),              % front end
        get_goals(Name,Conj2).              % rear end
get_goals(Name,(Disj1;Disj2)):-                 % disjunctive: treat same way
        !,
        get_goals(Name,Disj1),              % front end
        get_goals(Name,Disj2).              % arse end
get_goals(Name,Lone):-                  % single goal
        ( Lone=..[Other|Args] ->        % head can be put to good use
            assert_need(Name,Other),        % assert dependence
            get_constants(Args) ;           % extract the constants
            true).                          % didn't work, so ignore it

get_constants([]).
get_constants([First|Rest]):-
        ( not(constant(First)), atom(First) ->
            incrconst(constants),           % update count of constants
            assertz(constant(First)) ;      % not numbers
            true),
        get_constants(Rest).

assert_need(Name,Other):-                   % Name depends on Other
        (needs(Name,Other),not(ground(Other)) ->
            true ;                          % avoid duplicate assertions
            assertz(needs(Name,Other))).    % do it

get_other_predicates.                       % read other files of this prog
                                        % -none yet; 'markov' on demand

initconst:-                             % start a counter (ack!)
        asserta(constants(0)).

incrconst(Counter):-                        % increment a counter (pthft!)
        retract(constants(N)),
        NewN is N + 1,
        asserta(constants(NewN)).

retractall(Fact):-                          % facts only, unlike p.179 C&M
        retract(Fact),                      % remove matching fact
        fail.                           % force backtracking: get all
retractall(_).                              % force success
```



```
% markov.pl
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%     Program-restructuter for "pure" Datalog-style Prolog
%
%           Restructuring using full Markov-chain method.
%
%           The main program has done all the dirty work by the time that
%     this gets called.  We have the names of the constants, the number of
%     constants, the answer-set and domain sizes (yeah, it's wasteful to
%     recompute them each time, but at least this package should be self-
%     contained and easy to understand later).
%
%           We have a dependence graph for the predicates ready-built,
%     in the form of facts: "needs(<calling predicate>,<called predicate>)."
%     This should make it easy to determine if a predicate is ready to
%     transform.

markov:-
        setof(Name,data(Name),Name_list),   % get list of data-predicates
        scan_data(Name_list).               % scan it repeatedly until []

scan_data([]):-!.                       % bottom of recursion
scan_data(Name_list):-
        scan_data_2(Name_list,New_list),    % single sweep down the list
        scan_data(New_list).                % retry with shorter version

scan_data_2([],[]):-!.                      % bottom of recursion
scan_data_2([First|Rest],New_Rest):-        % no need to process grounds
        ground(First),
        !,                              % can skip other cases (red)
        scan_data_2(Rest,New_Rest).
scan_data_2([First|Rest],[First|New_Rest]):-     % can't process this one yet...
        needs(First,_),
        !,                              % can skip other cases (red)
        scan_data_2(Rest,New_Rest).
scan_data_2([First|Rest],New_Rest):-        % ready to process
        analyze(First),                     % do the dirty work
        retractall(needs(_,First)),         % free those that needed First
        scan_data_2(Rest,New_Rest).

analyze(Name):-                             % find "optimal" reorderings
        current_predicate(Name,Functor),    % get the functor
        functor(Functor,_,Arity),           % get the arity
        bagof(V,                        % find bodies of the clauses
              (clause(Functor,P),smash(P,V)),     %     of the predicate...
              Versions),                % & put them in a list of lists
        !,                              % probably unnecessary
        analyze2(Functor,Arity,Versions,0), % one level lower, so to speak
        update(Functor,Arity).              % convert results and reassert

update(Functor,Arity):-                     % correct and assert results
        full_mode(Arity,Mode),              % generate a "ufi" mode
        bagof( (Cost,(F,B,Mode,MB,I)),
              temp(Cost,F,B,Mode,MB,I),
              List ),
        sort_of(List,Sorted),
        embody(_,Mode,Sorted),
        fail.                           % force backtracking
```

```
update(_,_):-                          % force success
      abolish(temp,6).                 % remove temporary grot

embody(Functor,Mode,[]):-
      mode_ify(Functor,Mode,NF),
      find_cost(Functor,NF,Mode).      % see how expensive it is
embody(_,Mode,[(_,(F,Body,_,Mode_Body,_))|T]):-  % assert altered clauses
      mode_ify(F,Mode,NF),
      mode_ify_list(Body,Mode_Body,NewB),
      embody2(NewB,NewerB),
      Clause = (NF:-NewerB),           % change into clause
      assertz(Clause),                 % put new clause into database
      write_clause(Clause),            % write it as well for the user
      embody(F,Mode,T).

embody2([Lone],Lone):-!.
embody2([First|Rest],(First,Rest2)):-
      embody2(Rest,Rest2).

mode_ify_list([],[],[]).
mode_ify_list([B|Body],[M|Mode_Body],[N|New_Body]):-
      M=..[_|Mode],                    % fetch mode
      mode_ify(B,Mode,N),              % rename term, based on mode
      mode_ify_list(Body,Mode_Body,New_Body).

mode_ify(F,Mode,NF):-
      F=..[Name|Args],                 % fillet the functor F
      ( ground(Name) ->                % if it's a ground term
            NF = F                     % don't alter it
      ;                                % else alter it
            mode_num(Mode,Nums),       % convert mode to ASCII-numbers
            name(Name,Name_List),      % ASCII-list the name
            append(Name_List,[95|Nums],New_List),   % append "_<mode>*"
            name(New_Name,New_List),   % reconvert to name
            NF=..[New_Name|Args] ).    % construct new term from bits

mode_num([],[]).
mode_num([u|T],[117|U]):-              % ASCII for 'u'
      mode_num(T,U).
mode_num([i|T],[105|U]):-              % ASCII for 'i'
      mode_num(T,U).

find_cost(OldF,Functor,Mode):-
      OldF=..[Name|_],                 % store under original name
      Functor=..[_|Args],              % store under original name
      start_cost(Name,Mode),           % start cost-counter
      !,
      (alamode(Args,Mode),             % stuff with constants
      find_cost2(Functor,Cost),
      update_cost(Name,Mode,Cost),fail;true),   % force backtrack & success
      g(Name,S,_),                     % get # of successes
      NS is -S,
      update_cost(Name,Mode,NS),       % ... & subtract it to correct
      fixcost(Name,Mode).

fixcost(Name,Mode):-                   % convert total cost to average
      F=..[Name|Mode],                 % build the necessary term
      retract(cost(F,OldCost)),        % retract & get old cost
      g(Name,S,_),                     % get number of successes
```



```
      constants(C),                    % get number of constants
      fixcost2(Name,Mode,1,Denom,S,C,_),   % find denominator
      Cost is OldCost/Denom,           % average the cost
      assertz(cost(F,Cost)),           % assert it
      !.                               % is this cut necessary?

fixcost2(_,[],Denom,Denom,_,_,1):-     % no need to fudge denominator
      !.                               %     if fully-instantiated
fixcost2(_,[],TDenom,Denom,S,_,_):-    % partly (or un-)instantiated
      Denom is TDenom + S.             %     add # of successes
fixcost2(Name,[u|M],TDenom,Denom,S,C,u):-  % part-uninst.: set flag
      fixcost2(Name,M,TDenom,Denom,S,C,u).
fixcost2(Name,[i|M],TDenom,Denom,S,C,X):-  % inst.: propagate flag
      TDenom2 is TDenom*C,             % C times larger for every "i"
      fixcost2(Name,M,TDenom2,Denom,S,C,X).


find_cost2(Head,Cost):-                % dirty work of counting costs
      R0 is recalls,
      C0 is calls,
      (Head,fail;true),                % success causes a bogus call
      C1 is calls,
      R1 is recalls,
      Cost is C1-C0+R1-R0-2.           % or so I think

update_cost(Name,Mode,Cost):-          % increment counter
      F=..[Name|Mode],
      retract(cost(F,C0)),
      C1 is C0 + Cost,
      assertz(cost(F,C1)),
      !.

alamode([],[]).                        % stuff with constants
alamode([_|T],[u|R]):-                 % uninstantiated
      alamode(T,R).
alamode([X|T],[i|R]):-                 % instantiated
      constant(X),
      alamode(T,R).

start_cost(Name,Mode):-
      F=..[Name|Mode],                 % form the mode-bearing functor
      assertz(cost(F,0)),
      !.

write_clause(Clause):-                 % write a clause to output
      write(Clause),                   % clause proper
      write(.),                        % end-of-clause period
      nl.                              % new line

analyze2(_,_,[],_).                    % bottom of recursion
analyze2(Functor,Arity,[First|Others],ID):-
      ID2 is ID + 1,
      analyze3(Functor,Arity,First,ID2),   % down another level
      analyze2(Functor,Arity,Others,ID2).  % recursion

sort_of([H|T],S):-                     % insertion sort
      sort_of(T,L),
      insert(H,L,S).                   % tailored to this application
sort_of([],[]).
```

```
insert((XK,XR),[(HK,HR)|T],[(HK,HR)|L]):-
      HK < XK,                          % UNALTERED
      !,
      insert((XK,XR),T,L).
insert(X,L,[X|L]).

full_mode(0,[]):-
      !.                                % bottom of recursion
full_mode(Arity,[u|Rest]):-             % uninstantiated
      A is Arity-1,
      full_mode(A,Rest).
full_mode(Arity,[i|Rest]):-             % instantiated
      A is Arity-1,
      full_mode(A,Rest).

analyze3(F,Arity,G,ID):-
      length(G,Size),
      get_mode(Arity,M),
      perm(G,P),                        % get a permutation
      new((F,P,M),(Functor,Permed,Mode)), % make copy to foul
      Functor=..[_|Mode],               % propagate instantiation
      propagate(Permed,New),            % propagate everything else
      get_costs(New,Costs),             % replace goals w/costs
      get_probs(New,Probs),             % replace goals w/probabilities
      c_call(matrix(Costs,Probs,Size,Answer)), %call external routine!
      stuff2(M,Quasi),                  % hack: provide u&i form
      compare(Answer,F,P,Quasi,New,ID), % compare with incumbent
      fail.                             % force backtracking
analyze3(_,_,_,_).                      % force success

% temp(<best>,<uninst. fnct.>,<uninst. body-list>,<mode>,<inst. body-list>,ID).

compare(Answer,Functor,Body,Mode,BodyM,ID):-
      ( temp(Best,_,OldBody,Mode,_,ID)-> % if there is an incumbent
            (Answer < Best,
                  retract(temp(Best,_,_,Mode,_,ID)),  % replace.
                  assertz(temp(Answer,Functor,Body,Mode,BodyM,ID))
            ;
                  true )
      ;
            assertz(temp(Answer,Functor,Body,Mode,BodyM,ID)) ).


get_costs([],[]).
get_costs([Goal1|Goals],[Cost1|Costs]):-   % get cost of each goal
      cost(Goal1,Cost1),
      get_costs(Goals,Costs).

get_probs([],[]).
get_probs([Goal1|Goals],[Prob1|Probs]):-   % get probability of each goal
      prob(Goal1,Prob1),
      get_probs(Goals,Probs).

propagate([],[]).
propagate([First|Rest],[NewFirst|Rest2]):- % will require great subtlety
      stuff(First,NewFirst),              % make a copy w/ u replacing _
      instantiate(First),                 % set all variables to i
      propagate(Rest,Rest2).              % recursion
```



```
stuff(Partial,Full):-                   % fully-instantiate all vars.
      Partial=..[Name|Args],            % separate name from args.
      stuff2(Args,New),                 % vars->u, i->i
      Full=..[Name|New].                % construct simulacrum term

stuff2([],[]).                          % bottom of recursion
stuff2([A1|A],[u|N]):-                  % uninstantiated-variable case
      var(A1),                          % variables only!
      !,                                % red cut to prevent duplicates
      stuff2(A,N).                      % recursive call
stuff2([i|A],[i|N]):-                   % red cut, order makes it work
      stuff2(A,N).                      % recursive call

instantiate(Goal):-                     % set all variables to i
      Goal=..[_|Args],
      instantiate2(Args).

instantiate2([]).                       % trivial
instantiate2([i|Rest]):-
      instantiate2(Rest).

get_mode(0,[]):-!.                      % bottom of recursion
get_mode(Arity,[_|Rest]):-              % uninstantiated
      A is Arity-1,
      get_mode(A,Rest).
get_mode(Arity,[i|Rest]):-              % instantiated
      A is Arity-1,
      get_mode(A,Rest).

smash((Dis1;Dis2),Result):-             % disjunctive case
      !,                                % prevent match w/last clause
      (smash(Dis1,Result);smash(Dis2,Result)). %generates all by backtracking
smash((Head,Rest),Result):-             % conjunctive case
      !,                                % prevent match w/last clause
      smash(Rest,Tail),
      smash(Head,New),
      append(New,Tail,Result).          % join pieces together
smash(Last,[Last]).                     % bottom of recursion


perm([],[]).                            % standard list-permuter
perm([X|Y],[U|V]):-
      delete(U,[X|Y],W),
      perm(W,V).

delete(X,[X|Y],Y).                      % standard
delete(U,[X|Y],[X|V]):-
      delete(U,Y,V).

cost(Goal,1):-                          % non-ground predicates must
      functor(Goal,Name,_),             %     be found by experiment
      not(beast(Name)), %hack
      ground(Name),                     %     and then asserted
      !.

append([], L, L).                       % standard
append([H|T], L, [H|V]):-
      append(T, L, V).
```

```
new(X,Y) :-  % Y is a fresh copy of X (with new variables) [from Shapiro's MIS]
      abolish('gross hack',1),
      assert('gross hack'(X)),
      retract('gross hack'(Y)).


/*****************************************************************************
 matrix.c
 *****************************************************************************/

/*
 * Cost function for re-ordering Prolog goals.
 * Invert a matrix to find the cost; use calls to LINPACK routines.
 *
 */

#include <stdio.h>
#define LDA       10                    /* size of entire matrix */
#define JOB 1           /* compute inverse only, not determinant */


matrix(cost,prob,size,answer)
float cost[];
float prob[];
long int size;
float *answer;
{
      long int info,spl,tmp,job;
      long int ipvt[LDA];                     /* pivot vector for LINPACK  */
      float det[2];                           /* not actually used         */
      float sigma;
      float a[LDA][LDA];
      float work[LDA];              /* work vector for LINPACK   */
      short int i,j;
/*
 * This is a positive-definite band matrix, and therefore I could save
 * by using more-specialized LINPACK routines.  Well, I don't want to
 * rewrite the code much if I switch to a method that uses a more-general
 * matrix.
 */
      spl = size + 1;                         /* used very often           */
      for (i=0; i<spl; i++)
            for (j=0; j<spl; j++)
                  a[i][j] = 0.0;              /* clear the matrix */
      a[0][0] = 1.0;                          /* special cases */
      a[1][0] = -prob[0];
      a[size-1][size] = (-1.0);
      a[size][size] = 1.0;
      for (i=1; i<size; i++)  {               /* note that we construct the*/
            a[i-1][i] = prob[i] - 1.0;        /* transpose because we're   */
            a[i][i] = 1.0;                    /* going into Fortran        */
            a[i+1][i] = - prob[i];
      }
      tmp = LDA;                    /* Fortran uses call-by-ref.;*/
      job = JOB;                    /* we can't pass constants.  */
      sgefa_(a,&tmp,&spl,ipvt,&info);           /* factor the beast          */
      if (info != 0)
            printf("Matrix is singular.0);
      else
```



```
      {
            sgedi_(a,&tmp,&spl,ipvt,det,work,&job);   /* find the inverse  */
      }
/*
 * Now we have to compute the expected cost, which isn't too hard:
 */
      sigma = 0.0;                            /* clear sum-variable        */
      for (i=0; i<size; i++)                  /* success state has no cost */
            sigma += a[i][0] * cost[i];
      *answer = sigma;                /* pass answer to caller     */
}
```

APPENDIX B.

*SIDA** FOR JOBSHOP SCHEDULING

```
/*****************************************************************************
  global.h: global constants, etc.
 *****************************************************************************/

#include <sys/types.h>
#include <sys/times.h>
#include <stdio.h>

#include "splay.h"                      /* includes prob-specific */
#include "jobshop.h"                    /* def'ns specific to prob. */

#define Falsch 0
#define Treu 1

#define StackSize 32                    /* max. depth of spec. stack */
#define Infinity 0x7FFFFFFF             /* nice large 31-bit number */
                                      /* default values */
#define NumProbs 1
#define Size 3
#define Depth 8
#define Seedy 0x0FADFADE                /* arbitrary value for seed */

typedef struct params {
  int problems;                         /* number of problems */
  int size;                           /* size of a problem */
  int depth;                            /* depth of speculation */
  FILE *filep;                          /* file-pointer */
} Params;                             /* probably more to come */


/*****************************************************************************
  jobshop.h: constants, structures specific to the problem
 *****************************************************************************/

#define Procs 4                         /* number of processors */
#define Jobs 4                          /* number of jobs */
#define MaxTasks 6                      /* max. no. of tasks/job */
                                      /* maximum length of a task */
#define Length 64

typedef struct task
{
  int proc;
  int length;
} Task;

typedef struct sked
{
  int proc;
  int job;
} Sked;
```



```
typedef struct oot
{
  int job;
  int length;
} Oot;

typedef struct state
{
  int key;                              /* remaining length/job */
  int L[Jobs];                        /* earliest a job can run */
  int when[Jobs];                       /* time used on each proc'r */
  int used[Procs];                      /* left to be put on each */
  int demand[Procs];                    /* [index] available tasks */
  int tasks[Jobs];                      /* partial schedules */
  Sked done[Jobs*MaxTasks];             /* do not try these next */
  int no[Jobs];
  int kids[Jobs];                       /* how far in schedule */
  int where;                          /* next child to spawn */
  int next_child;                       /* node is a mother */
  int mother;
} State;

typedef struct frame
{
  int move;                           /* move that this node reps. */
  int delay;                            /* doing task caused delay */
  int smaller;
  int value;                            /* backed-up value */
  int which;                            /* where _value_ came from */
  int current_kid;                      /* child we're working on */
  int kids[Jobs];                     /* l-bounds for kids */
  int no[Jobs];                         /* children that're useless */
} Frame;


/*****************************************************************************
  main.c: main program
 *****************************************************************************/

/*  Speculative B&B program.

    Jobshop scheduling using naive heuristic.  The program generates and
    solves the problems.  To run:

    jobshop <problems> <size> <depth> <seed> <outfile>

    <problems>: number of problems to generate and solve (default 1)
    <size>:     number of tasks/job (default 3)
    <depth>:    depth of speculation (default 8)
    <seed>:     seed for random number generator (default 0x0FADFADE)
    <outfile>:  name of output file

    Give 0 as an argument to get a default value.

    4 jobs on 4 machines: this is set in "jobshop.h" & can easily be altered.
*/

#include "global.h"
```

```
  /* Global variables: */

struct tms buffer;                          /* for _times_ */
long time_is, time_was;

main(argc, argv)
int argc;
char *argv[];
{
  struct params p;
  register int j;
  register int num_probs;

  prelude(argc, argv, &p);                  /* fiddle args, etc. */
  num_probs = p.problems;

  for (j = 0; j < num_probs; j++)           /* for all instances */
    bb(&p);

  postlude(&p);
}


/*****************************************************************************
  bb.c: branch-and-bound-style framework for SIDA*
 *****************************************************************************/

#include "global.h"                         /* global definitions */

extern struct tms buffer;
extern long time_is, time_was;
extern int expanded, extant, guessed;

bb(ps)
struct params *ps;                          /* parameters for control */
{
  Queue *q;                               /* priority-queue */
  State *s, *i, i_state;
  int u, inc;                               /* incumbent */

  generate(ps);                             /* make instance; pass prams */
  u = Infinity;                             /* initialize the incumbent */
  inc = Infinity;
  expanded = extant = guessed = 0;          /* 0 the accounting-counters */
  q = splay_init();                         /* initialize the queue */
  initialize(q);                          /* start the search */
  times(&buffer);                         /* find time taken by search */
  time_was = buffer.tms_utime;

  while ( (s = (State *) splay_dequeue(q)) ) {  /* main loop: go until empty */
    u_bound(s, &u);                           /* find upper bound for p */
    i = (State *) branch(s, q, &u);
    if (i) {
      if (i->key < inc) {
      copy_state(i, &i_state);
      i_state.key = inc = i->key;
      }
      free(i);
    }
```



```
    splay_prune(u, q);
  }
  print_solution(&i_state);
  free(q);
}


/*****************************************************************************
  indep.c: get input parameters, handle files, etc.
 *****************************************************************************/

#include "global.h"

void prelude(argc, argv, P)
int argc;
char *argv[];
Params *p;
{
  int depth, problems, seedy, size;
  char outfile[15];                         /* name of output file */
  FILE *filep;
                                          /* get params; if 0, default */
  sscanf(argv[1], "%d", &problems);
  p->problems = problems ? problems : NumProbs;
  sscanf(argv[2], "%d", &size);
  p->size = size ? size : Size;
  sscanf(argv[3], "%d", &depth);
  p->depth = depth ? depth : Depth;
  sscanf(argv[4], "%d", &seedy);
  seedy = seedy ? seedy : Seedy;
  srand48(seedy);                           /* seed for random-#s */
  sscanf(argv[5], "%s", outfile);
  p->filep = filep = fopen(outfile, "w");
  setbuf(filep, (char *) 0);
}

void initialize(q)
Queue *q;
{
  int k;
  State *s;

  s = (State *) root_node();
  k = l_bound(s);
  s->key = k;
  splay_enqueue(k, (char *) s, q);
}

void postlude(p)
Params *p;                                  /* does essentially nothing */
{
  FILE *filep;

  filep = p->filep;
  fclose(filep);
}

/*****************************************************************************
```

```
 jobshop.c: routines specific to the problem
 ***************************************************************************/

/* Routines specific to the job-shop scheduling problem:

   Split on jobs (hence trees of degree Jobs)

   SIDA* version.
 */

#include "global.h"

/* Externals: */

extern struct tms buffer;
extern long time_is, time_was;
int expanded, extant, guessed;

/* Globals: */

static Task jobs[Jobs][MaxTasks];       /* jobs */
static int size;                        /* number of tasks / job */
static int depth;                       /* depth of speculation */
static FILE *filep;                     /* file pointer */

/* end Globals */

void generate(para)
Params *para;
{
  register int j, k;                    /* initialize globals */

  size = para->size;
  depth = para->depth;
  filep = para->filep;

  for (j = 0; j < Jobs; j++) {
    register int last_proc;

    last_proc = -1;
    for (k = 0; k < size; k++) {
      register int length, proc;

      proc = kruger(Procs);
      while (proc == last_proc)             /* contiguous tasks are not */
      proc = kruger(Procs);                 /* to demand the same job! */
      last_proc = proc;
      length = kruger(Length - 2) + 1;      /* 0 < length < Length */
      jobs[j][k].proc = proc;
      jobs[j][k].length = length;
    }
  }
}

State *root_node()
{
  State *s;
  register int j, k;
  register int *demand, *L, *used, *when;
```



```
  register int *tasks;

  s = (State *) malloc(sizeof(State));
  L = s->L;                             /* fast access */
  when = s->when;
  used = s->used;
  demand = s->demand;
  tasks = s->tasks;

  s->mother = 0;
  s->where = 0;

  for (j = 0; j < Procs; j++) {                 /* zero things */
    used[j] = 0;
    demand[j] = 0;
  }
  for (j = 0; j < Jobs * MaxTasks; j++) { /* probably not essential */
    s->done[j].proc = 0;
    s->done[j].job = 0;
  }
  for (j = 0; j < Jobs; j++) {
    register int i, l;

    when[j] = 0;
    tasks[j] = 0;
    s->kids[j] = 0;
    s->no[j] = 0;
    i = 0;
    for (k = 0; k < MaxTasks; k++) {
      register Task *t;

      t = &jobs[j][k];
      l = t->length;
      i += l;
      demand[t->proc] += l;
    }
    L[j] = i;
  }
  ++expanded;
  ++extant;
  return(s);
}

int solution(s)                         /* return 1 if s >solution */
State *s;
{
  register int i, yes, *L;

  yes = True;
  L = s->L;
  for (i = 0; i < Jobs; i++)
    if (L[i] != 0) {
      yes = False;
      break;
    }
  return(yes);
}

int l_bound(s)
```

```
State *s;                                   /* lower bound on solution */
{
  register int *used, *demand;
  register int i, inc;

  used = s->used;
  demand = s->demand;
  inc = 0;                                  /* want maximum value */
  for (i = 0; i < Procs; i++) {
    register int j;

    j = demand[i];
    j += used[i];                           /* lb for use of proc. i*/
    inc = (j > inc) ? j : inc;                      /* inc become maximum */
  }
  return(inc);
}

void u_bound(s, u)                                  /* return 1 if bound is new */
State *s;
int *u;
{
  register int i, inc, incval;
  int finished, u_val;
  int l[Jobs], us[Procs], w[Jobs];
  int t[Jobs];
  int length, j;
  struct task chore;

  for (i = 0; i < Procs; i++)               /* foulable copies made */
    us[i] = s->used[i];

  for (i = 0; i < Jobs; i++) {
    l[i] = s->L[i];
    w[i] = s->when[i];
    t[i] = s->tasks[i];
  }

  finished = 0;                             /* set to 0 unless done */
  do {                                      /* standard heuristic: job */
/*  with longest leftover */
    incval = 0;                             /* want _biggest_ leftover */
    for (i = 0; i < Jobs; i++) {
      j = l[i];                             /* on-the-fly in _if_ fails? */
      if (j > incval ) {                    /* =0 -> none left */
      inc = i;
      incval = j;
      }
    }

/* At this point, inc is the job whose task we schedule next; hence t[inc]
   points to that task, jobs[inc][t[inc]] is that task, .proc is the processor
   and .length the task's length. */

    if (! incval)                           /* only if we're done */
      finished = 1;
    else {
      int whe, use, pro;
```



```
      chore = jobs[inc][t[inc]++];          /* must incr. t[inc] */
      length = chore.length;
      pro = chore.proc;
      l[inc] -= length;
      if ( (whe = w[inc]) > (use = us[chore.proc])) {
      w[inc] = whe += length;
      us[pro] = whe;
      }
      else {
      us[pro] = use += length;
      w[inc] = use;
      }
    }
  } while (! finished);                     /* find last proc. to finish */

  incval = 0;
  for (i = 0; i < Procs; i++) {
    register int j2;

    if ( (j2 = us[i]) > incval)
      incval = j2;
  }
  u_val = *u;
  if (incval < u_val)
    *u = incval;
}

void copy_state(s, t)                       /* source, target */
State *s, *t;                               /* copy source-state... */
{                                           /* probably not enough regs. */
  register int i;
  register int *s_i, *t_i, *s_i2, *t_i2, k;
  register int *s_u, *t_u, *t_u2;

  s_i = s->L;                               /* fast access */
  t_i = t->L;
  s_i2 = s->when;
  t_i2 = t->when;
  s_u = s->tasks;
  t_u = t->tasks;
  t_u2 = t->no;
  for (i = 0; i < Jobs; i++) {                      /* copy */
    t_i[i] = s_i[i];
    t_i2[i] = s_i2[i];
    t_u[i] = s_u[i];
    t_u2[i] = 0;
    t->kids[i] = 0;
  }

  s_i = s->used;                            /* more fast access */
  t_i = t->used;
  s_i2 = s->demand;
  t_i2 = t->demand;
  t->where = k = s->where;

  for (i = 0; i <= k; i++) {
    t->done[i].proc = s->done[i].proc;
    t->done[i].job = s->done[i].job;
  }
```

```
  for (i = 0; i < Procs; i++) {                    /* copy */
    t_i[i] = s_i[i];
    t_i2[i] = s_i2[i];
  }
  t->next_child = 0;
  t->mother = 0;
}

void dump_prob()                                   /* 4 */
{                                                  /* print original problem */
  register int i, j;

  for (i = 0; i < Jobs; i++) {
    register int m;

    fprintf(filep, "Job %d:0, i);
    m = 0;
    for (j = 0; j < size; j++) {
      register int k, l;

      k = jobs[i][j].proc;
      l = jobs[i][j].length;
      fprintf(filep, " %d /", k);
      if (l < 10)                                  /* kludge: could use format */
      fprintf(filep, " ");
      if (l < 100)
      fprintf(filep," ");
      fprintf(filep," %d ", l);
      if (! (++m % 18))
      fprintf(filep,"0);
    }
    fprintf(filep,"0);                             /* separator */
  }
}

/* 'Strewth, what a bloody mess! */

void print_solution(s)
register State *s;
{
  int pp[Procs], jp[Jobs];                         /* place-markers for out */
  int used[Procs], when[Jobs];
  Oot out[Procs][Jobs*MaxTasks];                   /* buffer to hold output */
  register int i, j;
  int end, inc;
  int timely;

  times(&buffer);
  time_is = buffer.tms_utime;
  timely = time_is - time_was;
  fprintf(filep,"finding a schedule took %d0, timely);
  fprintf(filep, "%d nodes expanded, %d remain.  %d guesses.0,
        expanded, extant, guessed);

  inc = 0;
  for (i = 0; i < Procs; i++) {
    j = s->used[i];
    inc = (j > inc) ? j : inc;
    pp[i] = 0;                                     /* zero it: unrelated */
```



```
    used[i] = 0;                                   /* ditto */
  }
  fprintf(filep, "Schedule requires %d0, inc);
  dump_prob();                                     /* print out original prob. */

  end = s->where;
  for (i = 0; i < Jobs; i++) {
    jp[i] = 0;
    when[i] = 0;
  }

/* Now jp[job] points to the task being considered in jobs[job]. pp[proc]
   corresponds to the job or delay being considered in out[proc].
*/

  for (i = 0; i < end; i++) {
    register int delay, length, job, proc;

    proc = s->done[i].proc;
    job = s->done[i].job;
    length = jobs[job][jp[job]++].length;
    delay = when[job] - used[proc];
    if (delay > 0) {                               /* must delay */
      when[job] += length;
      used[proc] = when[job];
      out[proc][pp[proc]].job = -1;                /* -1 means "delay" */
      out[proc][pp[proc]++].length = delay;        /* length of delay */
    }
    else {
      used[proc] += length;
      when[job] = used[proc];
    }
    out[proc][pp[proc]].job = job;
    out[proc][pp[proc]++].length = length;
  }
  for (i = 0; i < Procs; i++) {                    /* for every processor */
    fprintf(filep, "Processor %d:0, i);
    for (j = 0; j < pp[i]; j++)                    /* names of jobs */
      if (out[i][j].job < 0)                       /* a delay */
      fprintf(filep, "   d");
      else
      fprintf(filep, "   %1d", out[i][j].job);
    fprintf(filep, "0);
    for (j = 0; j < pp[i]; j++)                    /* lengths of jobs */
      fprintf(filep, " %3d", out[i][j].length);
    fprintf(filep, "0);
  }
}

State *make_state(s, job)
State *s;
int job;                                           /* job to schedule next */
{
  register int length, proc, used, where, which, when;
  register State *t;

  which = s->tasks[job];                           /* caller must assure safe! */
  proc = jobs[job][which].proc;                    /* access */
  length = jobs[job][which].length;                /* access */
```

```
  t = (State *) malloc(sizeof(State));

  copy_state(s,t);                          /* copy _only_ the goodies */

/* Making the new state:  The used[proc] tell how early the desired processor
   will be available.  The when[job] tell how early a particular task can
   be scheduled.  If when is larger, the proc. idles until then; it then
   does the job.  If used is larger, the job must wait until the proc.
   is available.  In either case, when[job] = used[proc] = max(prev. values
   + length).  Simple as that.  The L-value of the job drops, the remaining
   demand on the processor drops, and the schedule gets another element.
 */

  t->tasks[job]++;                          /* next task. assume no oob */
  t->L[job] -= length;                      /* length left / job: reduce */
  if ( (when = t->when[job]) > (used = t->used[proc]) ) { /* delay */
    t->when[job] = when += length;          /* earliest for next task */
    t->used[proc] = when;                   /* earliest for next job */
  }
  else {                                 /* no delay: start at once */
    t->used[proc] = used += length;         /* earliest for next job */
    t->when[job] = used;                    /* earliest for next task */
  }

  t->demand[proc] -= length;                /* time left / proc: reduce */
  where = t->where;
  t->done[where].proc = proc;               /* add entry to schedule */
  t->done[where++].job = job;
  t->where = where;
  ++expanded;
  ++extant;
  return(t);
}

State *expand(s)                         /* front-end for make_state */
State *s;
{
  int child, i;
  State *t;

  child = s->next_child;

  for (i = child; i < Jobs; i++)            /* try all jobs if necessary */
    if (s->L[i] > 0)                        /* not done; s->no unused */
      break;
  if (i < Jobs) {                        /* can spawn a brat */
    s->next_child = i + 1;
    t = make_state(s, i);
  }
  else
    t = (State *) 0;
  return(t);
}

/* Make a frame to be the root of the _spekulationsbaum_.  This won't be
   based on any other frame, and it needs only the forbidden children from
   the initial state of the speculation.  Basically just a lot of
   initializing & clearing.
 */
```



```
void root_frame(fp, s)
Frame *fp;
State *s;
{
  int i;

  fp->current_kid = 0;
  fp->move = -1;
  fp->delay = 0;
  fp->smaller = 0;
  fp->value = Infinity;
  fp->which = -1;
  for (i = 0; i < Jobs; i++) {
    fp->no[i] = s->no[i];
    s->no[i] = 0;                       /* won't rely on this! */
    fp->kids[i] = 0;
  }
}

/* Make a virgin (sort of) child from the selected node.  _expand_ already
   checks L[i] > 0, so that we can omit that.  Basically we want the cheapest
   kid that's not forbidden by _no_ & whose job has at least one task left.
   We check the kid to ensure that it's not itself of value > u, or a solution,
   then check mama for ability to spawn further brats.
 */
State *make_virgin(s, q, u)
State *s;
Queue *q;
int u;
{
  int best, i, inc, kid_count;
  State *t;

  if (! s->mother)                          /* leaf or initial node */
    return(s);                              /* don't bother to make new */

  inc = Infinity;                        /* set incumbent to Infinity */
  kid_count = 0;
  for (i = 0; i< Jobs; i++) {
    if ((! s->no[i]) && (s->tasks[i] < size)) { /* hasn't existed but can */
      int b;

      b = s->kids[i];
      if (b <= u) {
      if (b < inc) {
        inc = b;
        best = i;
      }
      }
      else {
      s->no[i] = True;
      ++kid_count;
      }
    }
    else {
      s->no[i] = True;
      ++kid_count;
    }
  }
```

```
  if (kid_count == Jobs) {              /* can't make a kid */
    --extant;
    free(s);
    return((State *) 0);
  }
                                    /* spawn it */
  t = make_state(s, best);              /* make_state keeps count */
  for (i = 0; i < Jobs; i++)
    if (t->L[i] == 0)
      t->no[i] = Treu;

  if (kid_count == Jobs - 1) {                  /* this is mum's last */
    --extant;
    free(s);
  }
  else {                                /* life in the old girl yet */
    int b, key;

    s->no[best] = Treu;
    key = Infinity;
    for (i = 0; i < Jobs; i++)
      if ((! s->no[i]) && ((b = s->kids[i]) < key))
        key = b;
    s->mother = Treu;                   /* paranoia */
    s->key = key;
    splay_enqueue(key, (char *) s, q);
  }
  return(t);
}

/* On propagation:

   Two main cases: the kid that isn't allowed (solution, too large, or
   can't be made because its move isn't legal) and the leaf.

   The basic idea is to propagate only when something is the last thing of
   its level.
*/

int propagandalet(kid, dad, forbid_p)
Frame *kid, *dad;
int *forbid_p;                          /* return 1 -> next forbid. */
{
  int i, m, v;
  int blocked;

  m = kid->move;
  if (*forbid_p)                    /* this child is forbidden */
    dad->no[m] = Treu;                  /* set it so in the dad */
  else {                            /* usual case: permitted */
    dad->kids[m] = v = kid->value;
    if (v < dad->value) {
      dad->value = v;
      dad->which = m;
    }
  }
  if (m == Jobs - 1) {                  /* propagate */
    blocked = Treu;
    for (i = 0; i < Jobs; i++)
```



```
      if (! (dad->no[i]) ) {
      blocked = Falsch;
      break;
      }
    *forbid_p = blocked;
    return(Treu);                   /* go on */
  }
  else
    return(Falsch);                         /* stop. */
}

void propaganda(stack, incumbent, where, common, forbid)
Frame *stack, *incumbent;
int where, common, forbid;
{
  int dad, propagate;

  --where;
  --common;
  dad = where - 1;

  do {
    propagate = propagandalet(&stack[where], &stack[dad], &forbid);
    if (common >= dad)
      propagate = propagandalet(&stack[where], &incumbent[dad], &forbid);
    --where;
    --dad;
  } while (propagate && (dad >= 0));
}

/* Make a frame as new...could easily inline this, but what the heck...
 */
void new_frame(p)
Frame *p;
{
  int i;

  for (i = 0; i < Jobs; i++) {
    p->kids[i] = 0;
    p->no[i] = 0;
  }
  p->current_kid = 0;
  p->value = Infinity;
  p->which = -1;
  ++guessed;
}

/* Usual way to make a frame, based both on a local state and the parent frame.
   The new frame appears on the stack in the position after the parent's.
   Treat as a Boolean: 0 if the new frame cannot be made, 1 if it can.
   Check for legality, but test no bounds.  This is mostly a copy of
   make_state, but without the allocation.

 */
int make_frame(fp, s)
Frame *fp;
State *s;
{
  Frame *sp2;
```

```
  int i;
  int job, length, proc, used, when, where, which;

  job = fp->current_kid;
  if (job >= Jobs)
    return(-1);                          /* indicate no more alt'ives */

  sp2 = fp + 1;
  if (fp->no[job] || (s->L[job] == 0)) {  /* cannot expand */
    sp2->move = job;                     /* kludge for _propaganda_ */
    return(Falsch);                      /* indicate alt'ive invalid */
  }

  new_frame(sp2);
  sp2->move = job;

  which = s->tasks[job];
  proc = jobs[job][which].proc;
  length = jobs[job][which].length;

  ++(s->tasks[job]);
  s->L[job] -= length;                   /* reduce length left/job */
  if ( (when = s->when[job]) > (used = s->used[proc]) ) {
    sp2->delay = 1;                      /* delay: info. for b'track */
    sp2->smaller = used;                 /* cache value for b'track */
    s->when[job] = when += length;       /* earliest for next task */
    s->used[proc] = when;                /* earliest for next job */
  }
  else {
    sp2->delay = 0;                      /* no delay */
    sp2->smaller = when;                 /* cache value for b'track */
    s->used[proc] = used += length;      /* earliest for next job */
    s->when[job] = used;                 /* earliest for next task */
  }

  s->demand[proc] -= length;             /* reduce time left/proc */
  where = s->where;
  s->done[where].proc = proc;
  s->done[where++].job = job;
  s->where = where;

  for (i = 0; i < Jobs; i++)
    if (s->L[i] == 0)                    /* DO NOT set s->no[i] ! */
      sp2->no[i] = Treu;                 /* essential */
  ++guessed;
  return(Treu);
}

/* Undo the frame just made, returning State _s_ to its previous condition.
   Advance current_kid (crucial!!), etc. Void because we are discarding,
   not creating.

 */
void unmake_frame(fp, s)
Frame *fp;
State *s;
{
  Frame *sp2;
  int job, length, proc, used, which;
```



```
  sp2 = fp + 1;
  job = sp2->move;
  which = --s->tasks[job];               /* make move look untried */
  proc = jobs[job][which].proc;          /* plate des boilers */
  length = jobs[job][which].length;

  s->L[job] += length;                   /* length left/job: restore */
  if (sp2->delay) {                      /* doing task caused delay */
    s->when[job] -= length;
    s->used[proc] = sp2->smaller;
  }
  else {                                 /* there was no delay */
    s->used[proc] -= length;
    s->when[job] = sp2->smaller;
  }
  ++(fp->current_kid);                   /* important! */
  s->demand[proc] += length;
  --s->where;
}

/* Make a frame + corresponding state into a new state to be enqueued.
   This handles all cases except the initial one, in which we don't have
   to build a new state but just vet an old one.
   Be certain to set _done_ and _mother_.

   Note that nodes are being generated in this way: first the root + its
   stack-entry is used to make the chosen son of the root, then son + root
   the grandson, and so on.  Note then that at worst fp2 points to the
   stack-position of the chosen leaf.
 */
State *reinstate(fp, s, u, valid)
Frame *fp;
State *s;
int u;
int *valid;
{
  Frame *fp2, *fp3;
  State *t;
  int i;
  int done, l, which;

  fp2 = fp + 1;
  which = fp2->move;
  t = make_state(s, which);

  if (fp2->which < 0) {                  /* leaf */
    t->mother = Falsch;                  /* not already a mother */
    done = Treu;
    if ( (t->key = fp2->value) <= u)
      for (i = 0; i < Jobs; i++)
      if (! (t->no[i] = fp2->no[i]))
        done = Falsch;
  }
  else {                                 /* not a leaf */
    t->mother = Treu;                    /* hence already a mother */
    fp3 = fp2 + 1;
    which = fp3->move;
    fp2->no[which] = Treu;               /* forbid child == move */
    done = Treu;
```

```
    l = Infinity;
    for (i = 0; i < Jobs; i++)
      if (! (t->no[i] = fp2->no[i])) {
      int b;

      t->kids[i] = b = fp2->kids[i];
      if (b <= u) {
        done = Falsch;
        if (b < l)
          l = b;
      }
      else
        t->no[i] = Treu;
      }
    t->key = l;
  }
  *valid = done;
  return(t);
}

/* Call reinstate repeatedly to convert a path into a series of states
   enqueued one by one.
 */
void enqueue_path(fp, s, q, u)
Frame *fp;
State *s;
Queue *q;
int u;
{
  State *t;                          /* new states: use this */
  State *trash[StackSize], **tp;          /* trash-states */
  Frame *fp2;
  int i;
  int done, l, which;

  tp = trash;
                                     /* special case for initial */
  fp2 = fp + 1;
  which = fp2->move;
  fp->no[which] = Treu;                   /* forbid move == child's */
  done = Treu;
  l = Infinity;
  for (i = 0; i < Jobs; i++)
    if (! (s->no[i] = fp->no[i])) {       /* a child is yet possible */
      int b;

      s->kids[i] = b = fp->kids[i];       /* load only for valid kids */
      if (b <= u) {
      done = Falsch;
      if (b < l)
        l = b;
      }
      else
      s->no[i] = Treu;
    }

  if (! done) {                           /* put it back */
    s->key = l;
    s->mother = Treu;                     /* just to be certain */
```



```
    splay_enqueue(l, (char *) s, q);
  }
  else
    *(tp++) = s;

  for (i = 1; i < depth; i++) {                    /* other cases */
    done = Falsch;
    t = reinstate(fp, s, u, &done);
    ++fp;
    if (!done)
      splay_enqueue(t->key, (char *) t, q);
    else
      *(tp++) = t;
    s = t;
  }

  --tp;
  while (tp >= trash) {
    free(*tp);
    --tp;
    --extant;
  }
}

/* Copy the contents of the source frame to the sink frame.  Zero anything
   that is useless within the incumbent.
 */
void copy_frame(source, sink)
Frame *source, *sink;
{
  int i;

  sink->move = source->move;
  sink->delay = 0;
  sink->smaller = 0;
  sink->value = source->value;
  sink->which = source->which;
  sink->current_kid = 0;
  for (i = 0; i < Jobs; i++) {
    sink->kids[i] = source->kids[i];
    sink->no[i] = source->no[i];
  }
}

/* Copy the contents of the stack from the source stack to the sink stack,
   using copy_frame to do it frame by frame.
   Examples: for a new incumbent when the last was a brother, source and
   sink point to the last entry of their stacks, and we do exactly one copy.
   For the first incumbent, we start at the bases of the stacks and do
   depth - 0 = depth copies.
 */
void copy_stack(source, sink, entries)
Frame *source, *sink;
int entries;
{
  int i;

  for (i = 0; i < entries; i++)
    copy_frame(&source[i], &sink[i]);
```

```
}

/* Speculation: */

State *branch(s, q, u)
State *s;
Queue *q;
int *u;
{
  Frame stack[StackSize], incumbent[StackSize], *sp;
  State *t, *inc_state;
  int common, current, depthm1, inc, l, can;

  t = make_virgin(s, q, *u);
  if (! t)                              /* can't make a child */
    return((State *) 0);                /* quit */
  if (solution(t)) {                    /* catch virgins */
    l = l_bound(t);
    if (l <= *u) {
      *u = l - 1;
      inc_state = (State *) malloc(sizeof(State));
      copy_state(t, inc_state);
      inc_state->key = l;
      return(inc_state);
    }
    else
      return((State *) 0);
  }

  common = 0;                           /* only initially; not again */
  current = 1;                          /* to keep upper limit excl. */
  depthm1 = depth - 1;
  inc = Infinity;
  sp = stack;
  root_frame(sp, t);
  inc_state = (State *) 0;

  do {                                  /* main loop */
    if ( (can = make_frame(sp, t)) > 0) { /* begin CAN MAKE */
      l = l_bound(t);
      if (l <= *u) {                    /* begin KEEP THIS STATE */
      if (solution(t)) {                /* state is a solution */
        *u = l - 1;                     /* tighten u further */
        if (! inc_state)                /* allocate incumbent-state */
          inc_state = (State *) malloc(sizeof(State));
        copy_state(t, inc_state);
        inc_state->key = l;

        propaganda(stack, incumbent, current + 1, common, Treu);
        unmake_frame(sp, t);
      }
      else {                            /* begin USUAL CASE */
        ++sp;
        ++current;
        if (! (current < depthm1)) {    /* begin LEAVES */
          sp->value = l;                /* for propagation */
          propaganda(stack, incumbent, current, common, Falsch);
          if (l < inc) {                /* new incumbent */
            u_bound(s, u);
```



```
            inc = l;
            copy_stack(&stack[common], &incumbent[common], depth - common);
            common = depthm1;           /* exclusive limit */
          }
          --current;                    /* can go no deeper: b'track */
          --sp;
          unmake_frame(sp, t);
        }                               /* end LEAVES */
      }                                 /* end USUAL CASE */
      }                                 /* end KEEP THIS STATE */
      else {                            /* l-bound is too large */
      propaganda(stack, incumbent, current + 1, common, Treu);
      unmake_frame(sp, t);
      }
    }                                   /* end CAN MAKE */
    else if (can < 0) {                 /* can't make; no others */
      --sp;                             /* backtrack */
      --current;
      if (current < common)
      common = current;
      if (sp >= stack)                  /* kludge */
      unmake_frame(sp, t);
    }
    else {                              /* exactly == 0 */
      propaganda(stack, incumbent, current + 1, common, Treu);
      ++(sp->current_kid);
    }
  } while (sp >= stack);
  if (inc < Infinity)                   /* made it to the leaves */
    enqueue_path(incumbent, t, q, *u);
  return(inc_state);
}


/*****************************************************************************
  random.c: call random number generator & return random integers
 *****************************************************************************/

#define MaxRandLog 31                   /* lg(random) < 31 */
#define MaxLog 12                     /* maximum value of lg: log2 */

static int log2(n)                      /* floor(lg(n)), kludge */
int n;                                  /* assume > 0 always */
{
  register int i,j, k;

  i = n;
  j = 1 << MaxLog;                      /* start from top bit */
  k = MaxLog;                           /* important: adjust MaxLog! */
  do {
    if (i & j)                          /* bit is set */
      break;                            /* k must be floor of log */
    else {
      k--;                            /* try next bit */
      j >>= 1;
    }
  } while (j);
  return(k);
}
```

```
int kruger(range)                       /* RANDom #, 0 <= # < range */
int range;
{
  int random, shift, which;

  shift = log2(range) + 1;              /* how far to shift random */
  random = lrand48();
  random >>= shift;
  which = (random * range) >> (MaxRandLog - shift);
  return(which);
}


/*****************************************************************************
  splay.h: definitions for splay-tree package
 *****************************************************************************/

#define MaxStack 16384

typedef struct node {
  struct node *left;
  struct node *right;
  struct nodelet *in;                   /* nodes of identical value */
  int key;
  char *data;                           /* cast to suitable type */
} Node;

typedef struct nodelet {                /* for identical-key nodes */
  struct nodelet *in;
  char *data;                           /* cast to suitable type */
} Nodelet;

typedef struct queue {
  Node *root;                           /* root of splay tree */
  Node *head;                           /* head of priority queue */
} Queue;

extern struct queue *splay_init();
extern void splay_enqueue();
extern splay_prune();
extern char *splay_dequeue();


/*****************************************************************************
  splay.c: priority-queue package using splay trees
 *****************************************************************************/

/* Splay trees for priority queues.  Code, after Sleator & Tarjan, JACM 32(3),
   p. 668-9.
 */

#include "splay.h"

Queue *splay_init()
{
  Queue *q;

  q = (Queue *) malloc(sizeof(Queue));
  q->root = (Node *) 0;
```



```
  return(q);
}

void splay_enqueue(i, s, q)
register int i;
char *s;
Queue *q;
{
  register Node *t;

  if ( (t = q->root) ) {                /* queue is not empty */
    Node Null;
    register int k;
    register Node *l, *r;

    l = r = &Null;
    l->left = (Node *) 0;
    l->right = (Node *) 0;

    do {
      k = t->key;
      if (i < k) {                      /* go left */
      register Node *left;

      if ( (left = t->left) ) {
        register int lk;

        lk = left->key;
        if (i < lk) {                   /* zig-zig: rotate r, link r */
          t->left = left->right;
          left->right = t;
          t = left->left;
          left->left = (Node *) 0;
          r->left = left;
          r = left;
        }
        else if (i > lk) {              /* zig-zag: link r, link l */
          r->left = t;
          r = t;
          l->right = left;
          t->left = (Node *) 0;
          t = left->right;
          left->right = (Node *) 0;
          l = left;
        }
        else {                          /* match: zig: link right */
          r->left = t;
          r = t;
          r->left = (Node *) 0;
          t = left;
          break;
        }
      }
      else {                            /* empty: zig: link right */
        r->left = t;
        r = t;
        r->left = (Node *) 0;
        t = left;
        break;
```

```
}
  }
  else if (i > k) {                    /* go right */
  register Node *right;

  if ( (right = t->right) ) {
    register int rk;

    rk = right->key;
    if (i < rk) {                      /* zig-zag: link l, link r */
      l->right = t;
      l = t;
      r->left = right;
      t->right = (Node *) 0;
      t = right->left;
      right->left = (Node *) 0;
      r = right;
    }
    else if (i > rk) {                 /* zig-zig: rotate l, link l */
      t->right = right->left;
      right->left = t;
      t = right->right;
      right->right = (Node *) 0;
      l->right = right;
      l = right;
    }
    else {                           /* match: zig: link left */
      l->right = t;
      l = t;
      l->right = (Node *) 0;
      t = right;
      break;
    }
  }
  else {                             /* empty: zig: link left */
    l->right = t;
    l = t;
    l->right = (Node *) 0;
    t = right;
    break;
  }
  }
  else                             /* match, straight off */
  break;
} while (t);
if (t) {                           /* node already exists */
  Nodelet *n;

  n = (Nodelet *) malloc(sizeof(Nodelet));
  n->in = t->in;                   /* often 0, but what matter? */
  t->in = n;
  n->data = t->data;
  l->right = t->left;
  r->left = t->right;
}
else {                             /* create a new node */
  t = (Node *) malloc(sizeof(Node));
  t->in = (Nodelet *) 0;
  t->key = i;
```



```
  }
  t->data = s;
  t->right = Null.left;
  t->left = Null.right;
  q->root = t;
 }
 else {                              /* queue is empty: init. */
  t = (Node *) malloc(sizeof(Node));
  t->left = (Node *) 0;
  t->right = (Node *) 0;
  t->in = (Nodelet *) 0;
  t->data = s;
  t->key = i;
  q->root = t;
 }
}

/* Prune all nodes with values strictly greater than _i_.  Splay the last node
   less than or equal to i to the root, which creates a right tree that can be
   discarded.
 */

splay_prune(i, q)
int i;
Queue *q;
{
  register Node *t;

  if ( (t = q->root) ) {                    /* queue is not empty */
    register int k;
    Node Null;
    register Node *l, *r;

    l = r = &Null;
    l->left = (Node *) 0;
    l->right = (Node *) 0;
    l->in = (Nodelet *) 0;

    do {
      k = t->key;
      if (i > k) {                          /* go right */
      register Node *right;

      if ( (right = t->right) ) {
        register int rk;

        rk = right->key;
        if (i > rk) {                       /* zig-zig: rotate l, link l */
          t->right = right->left;
          right->left = t;
          t = right->right;
          right->right = (Node *) 0;
          l->right = right;
          l = right;
        }
        else if (i < rk) {                  /* zig-zag: link l, link r */
          l->right = t;
          l = t;
          r->left = right;
```

```
t->right = (Node *) 0;
    t = right->left;
    right->left = (Node *) 0;
    r = right;
  }
  else {
    l->right = t;
    l = t;
    l->right = (Node *) 0;          /* new hack! */
    t = right;
    break;
  }
}
else {                              /* zig: link left */
  l->right = t;
  l = t;
  l->right = (Node *) 0;            /* new hack! */
  t = right;
  break;
}
}
else if (i < k) {                   /* go left */
register Node *left;

if ( (left = t->left) ) {
  register int lk;

  lk = left->key;
  if (i > lk) {                     /* zig-zag: link r, link l */
    r->left = t;
    r = t;
    l->right = left;
    t->left = (Node *) 0;
    t = left->right;
    left->right = (Node *) 0;
    l = left;
  }
  else if (i < lk) {                /* zig-zig: rotate r, link r */
    t->left = left->right;
    left->right = t;
    t = left->left;
    left->left = (Node *) 0;
    r->left = left;
    r = left;
  }
  else {
    r->left = t;
    r = t;
    r->left = (Node *) 0;           /* new hack! */
    t = left;
    break;
  }
}
else {                              /* zig: link right */
  r->left = t;
  r = t;
  r->left = (Node *) 0;             /* new hack! */
  t = left;
  break;
```



```
}
  }
  else                              /* exact match */
  break;
} while(t);

if (t) {                            /* a node exists of value i */
  register Node *x;

  l->right = t->left;
  t->left = Null.right;
  q->root = t;

  if ( (x = t->right) ) {           /* FREE rh tree of t */
  if (! (x->left && x->right)) {          /* only one value */
    register Nodelet *in;

    if ( (in = x->in) ) {
      register Nodelet *in2;

      while (in) {
        in2 = in;
        in = in->in;
        free(in2->data);
        free(in2);
      }
    }
    free(x->data);
    free(x);
  }
  else {
    Node *stack[MaxStack];
    register Node *l, *r, **sp;

    sp = stack;                     /* point to bottom of stack */
    while (sp >= stack) {           /* stack op every iteration */
      if (x) {                      /* can operate on current x */
        register Nodelet *in;

        if ( (in = x->in) ) {
        register Nodelet *in2;

        while (in) {
          in2 = in;
          in = in->in;
          free(in2->data);
          free(in2);
        }
        }
        free(x->data);
        l = x->left;
        r = x->right;
        free(x);
        x = l;
        if (x)
        *(sp++) = r;
      }
      else                          /* current x is nil */
        x = *(--sp);
```

```
}
 }
 t->right = (Node *) 0;                    /* saver */
 }

 if ( (x = Null.left) )                    /* FREE anything in R tree */
 if (! (x->left && x->right)) {                  /* only one node */
   register Nodelet *in;

   if ( (in = x->in) ) {
     register Nodelet *in2;

     while (in) {
       in2 = in;
       in = in->in;
       free(in2->data);
       free(in2);
     }
   }
   free(x->data);
   free(x);
 }
 else {
   Node *stack[MaxStack];
   register Node *l, *r, **sp;

   sp = stack;                             /* point to bottom of stack */
   while (sp >= stack) {                   /* stack op every iteration */
     if (x) {                              /* can operate on current x */
       register Nodelet *in;

       if ( (in = x->in) ) {
       register Nodelet *in2;

       while (in) {
         in2 = in;
         in = in->in;
         free(in2->data);
         free(in2);
       }
       }
       free(x->data);
       l = x->left;
       r = x->right;
       free(x);
       x = l;
       if (r)
       *(sp++) = r;
     }
     else                                  /* current x is nil */
       x = *(--sp);
   }
  }
}
else {                                     /* no node exists of value i */
  register Node *x;

  q->root = Null.right;
```



```
 if ( (x = Null.left) ) {                  /* FREE anything in R tree */
 if (! (x->left && x->right)) {                  /* only one node */
   register Nodelet *in;

   if ( (in = x->in) ) {
     register Nodelet *in2;

     while (in) {
       in2 = in;
       in = in->in;
       free(in2->data);
       free(in2);
     }
   }
   free(x->data);
   free(x);
 }
 else {
   Node *stack[MaxStack];
   register Node *l, *r, **sp;

   sp = stack;                             /* point to bottom of stack */
   while (sp >= stack) {                   /* stack op every iteration */
     if (x) {                              /* can operate on current x */
       register Nodelet *in;

       if ( (in = x->in) ) {
       register Nodelet *in2;

       while (in) {
         in2 = in;
         in = in->in;
         free(in2->data);
         free(in2);
       }
       }
       free(x->data);
       l = x->left;
       r = x->right;
       free(x);
       x = l;
       if (r)
       *(sp++) = r;
     }
     else                                  /* current x is nil */
       x = *(--sp);
   }
  }
  }
  return(1);
 }
}
else {                                     /* queue is empty: */
  return(0);
 }
}

/* Return head of priority queue.  Splay the tree on the value 0, which puts
   everything in the tree R.  Note that we don't need to know how to do cases
```

```
   other than lefthanded zig and zig-zig, which saves tests & registers.
   Note that this _does_not_ free the head: that has to be done by the caller,
   or somebody else.

   Revised: now it returns only the state, not the head proper.  The head node
   it deletes...
 */

char *splay_dequeue(q)
Queue *q;
{
  register Node *t;

  if ( (t = q->root) ) {                    /* queue is not empty */
    Node Null;
    register Node *r, *dad;

    r = &Null;
    r->left = (Node *) 0;
    r->right = (Node *) 0;

    do {
      register Node *left;

      dad = r;                              /* save last value */
      if ( (left = t->left) ) {
      t->left = left->right;                /* zig-zig */
      left->right = t;
      t = left->left;
      left->left = (Node *) 0;
      r->left = left;
      r = left;
      }
      else {                                /* zig: link right */
      r->left = t;
      r = t;
      r->left = (Node *) 0;
      t = left;
      break;
      }
    } while(t);

/* t points to nothing.  L is empty (we don't even have code that looks at it).
   R contains everything else.  r points to the leftmost node of R, which is
   the node we want. dad points to its father.  r->left necessarily is nil,
   but r->right may be a subtree.  We know that this subtree has items of
   key less than dad's, so we set dad->left to it.  We delete r (but not its
   state -- that's the duty of the caller).

   Of course, if r is a portmanteau node, we simply return its state, promote
   the state of the first member of its string, & delete not r but that 1st
   member.
 */
    {
      register Nodelet *in;
      char *s;

      if ( (in = r->in) ) {                 /* a portmanteau node */
      r->in = in->in;
```



```
      s = r->data;
      r->data = in->data;
      free(in);
      }
      else {
      dad->left = r->right;                 /* replace son with grandson */
      s = r->data;
      free(r);
      }
      q->root = Null.left;
      return(s);
    }
  }
  else {                                    /* queue is empty: init. */
    return((char *) 0);
  }
}
```

APPENDIX C.

*SIDA** FOR EIGHT- AND FIFTEEN-PUZZLES

```
/*****************************************************************************
  global.h: global declarations
 *****************************************************************************/

#include <sys/types.h>
#include <sys/times.h>
#include <stdio.h>

#include "splay.h"                      /* includes prob-specific */
#include "nbyn.h"                     /* def'ns specific to prob. */

#define Falsch 0
#define Treu 1

#define StackSize 128                   /* max. depth of spec. stack */
#define Infinity 0x7FFFFFFF             /* nice large 31-bit number */
                                      /* default values */

#define Size 3
#define Depth 8

typedef struct params {
  int size;                           /* size of a problem */
  int depth;                            /* _Spekulationsniederung_ */
  FILE *filep;                          /* file-pointer */
} Params;


/*****************************************************************************
  nbyn.h: constants, structures, etc. specific to the problem
 *****************************************************************************/

#define B 4                           /* maximum branching-factor */
#define MaxSize 4                     /* 24-puzzle intractable? */
#define MaxMoves 128                        /* max. # of moves/solution */

#define None -1
#define D 0                           /* mimic order used by Korf */
#define R 1                           /* (no rhyme or reason to it)*/
#define L 2
#define U 3

typedef struct state
{
  int key;
  int move;
  int board[MaxSize*MaxSize];
  int blank;
  int depth;
  int h;
  char done[MaxMoves];                      /* record of moves */
```



```
                                    /* place to stow next move */
  int where;
  int no[B];
  int kids[B];
  int next_child;
  int mother;
} State;

typedef struct frame
{
  int move;
  int value;
  int which;
  int depth;
  int current_kid;
  int kids[B];
  int no[B];
} Frame;


/*****************************************************************************
  main.c: main program
 *****************************************************************************/

/* 8-Puzzle or 15-Puzzle by SIDA*.  This version accepts a single problem and
   solves it.  To run:

   nbyn size depth output_file < input_file

   size: 3 for 8-Puzzle, 4 for 15-Puzzle
   depth: intentional overshoot added to upper bound of each stage.
   output_file: name of output file

   The input file is a permutation of the numbers 0 through 8 (8-Puzzle)
   or 0 through 15 (15-Puzzle) separated by white space.  0 is the blank,
   and a solved puzzle is represented as
      0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
*/

#include "global.h"

  /* Global variables: */

struct tms buffer;                          /* for _times_ */
long time_is, time_was;                     /* going whole hog for Bacon */

main(argc, argv)                    /* 0 */
int argc;
char *argv[];
{
  struct params P;
  register int j;

  prelude(argc, argv, &P);                  /* fiddle args, etc. */
  bb(&P);
  postlude(&P);
}

/*****************************************************************************
```

```
 bb.c: branch-and-bound-style framework for SIDA*
 ********************************************************************************/


#include "global.h"                         /* global definitions */

extern struct tms buffer;
extern long time_is, time_was;
extern int expanded, extant, guessed;

bb(ps)
struct params *ps;                          /* parameters for control */
{
  Queue *q;                               /* priority-queue */
  State *s, *i, i_state;
  int u, inc;
  int permutation[MaxSize*MaxSize];

  generate(ps, permutation);                /* make instance; pass prams */
  u = Infinity;                             /* initialize the incumbent */
  inc = Infinity;
  expanded = extant = guessed = 0;          /* 0 the accounting-counters */
  q = splay_init();                         /* initialize the queue */
  initialize(q, permutation);               /* start the search */
  times(&buffer);                         /* find time taken by search */
  time_was = buffer.tms_utime;

  while ( (s = (State *) splay_dequeue(q)) ) {  /* main loop: go until empty */
    i = (State *) branch(s, q, &u);
    if (i) {
      if (i->key < inc) {
      copy_state(i, &i_state);
      i_state.key = inc = i->key;
      }
      free(i);
    }
    splay_prune(u, q);
  }
  print_solution(&i_state);
  splay_prune(0, q);
  free(q);
}

/********************************************************************************
  indep.c: get arguments, open & close files, initialize queues, etc.
********************************************************************************/

#include "global.h"

void prelude(argc, argv, p)
int argc;
char *argv[];
Params *p;
{
  int depth, size;
  char outfile[15];                         /* name of output file */
  FILE *filep;
                                          /* get params; if 0, default */
```



```
  sscanf(argv[1], "%d", &size);
  p->size = size ? size : Size;
  sscanf(argv[2], "%d", &depth);
  p->depth = depth ? depth : Depth;
  sscanf(argv[2], "%s", outfile);
  p->filep = filep = fopen(outfile, "w");
}

void initialize(q, permutation)
Queue *q;
int permutation[];
{
  int k;
  State *s;

  s = (State *) root_node(permutation);
  k = l_bound(s);
  s->key = k;
  splay_enqueue(k, 0, (char *) s, q);
}

void postlude(p)
Params *p;                                /* does essentially nothing */
{
  FILE *filep;

  filep = p->filep;
  fclose(filep);
}


/********************************************************************************
  nbyn.c: procedures specific to problem, including speculation routine
 ********************************************************************************/

#include "global.h"

/* Externals: */

extern struct tms buffer;
extern long time_is, time_was;
int expanded, extant, guessed;

/* Globals: */

static int size, sizesize, sizem1;        /* size of puzzle; size*size */
static int depth;
static FILE *filep;                       /* file pointer */

int reverse[8] = { U, L, R, D };
char symbol[8] = { 'd', 'r', 'l', 'u' };

/* end Globals */

void dump_prob(permutation)
int permutation[];
{
  int i;
```

```
  fprintf(filep,"Original configuration is:0);
  for (i = 0; i < sizesize; i++)
    fprintf(filep," %d", permutation[i]);
  fprintf(filep,"0);
}

void generate(para, permutation)
Params *para;
int permutation[];
{
                                    /* initialize globals */
  int i;

  size = para->size;
  depth = para->depth;
  filep = para->filep;
  sizem1 = size - 1;
  sizesize = size * size;
  i = 0;
  while (scanf("%d", &permutation[i++]) != EOF)
    ;
  dump_prob(permutation);
}

void forbid(blank, verboten)
int blank, verboten[];
{
  int x, y;

  y = blank / size;
  x = blank % size;

  if (y == 0)
    verboten[D] = Treu;
  else if (y == sizem1)
    verboten[U] = Treu;
  if (x == 0)
    verboten[R] = Treu;
  else if (x == sizem1)
    verboten[L] = Treu;
}

State *root_node(permutation)
int permutation[];
{
  State *s;
  register int i, j;

  s = (State *) malloc(sizeof(State));

  s->depth = 0;
  s->h = 0;
  s->where = 0;
  s->next_child = 0;
  s->mother = 0;

  for (j = 0; j < B; j++) {
    s->no[j] = 0;
    s->kids[j] = 0;
```



```
  }

  for (i = 0; i < sizesize; i++) {
    int t;

    s->board[i] = t = permutation[i];
    if (t == 0) {                    /* location of the blank */
      s->blank = i;
      forbid(i, s->no);
    }
  }
  s->key = l_bound(s);
  ++expanded;
  ++extant;

  return(s);
}

int solution(s)                      /* return 1 if a solution */
State *s;
{
  register int i, yes, *board;

  yes = Treu;
  board = s->board;
  for (i = 0; i < sizesize; i++)
    if (board[i] != i) {
      yes = Falsch;
      break;
    }
  return(yes);
}

/* Return full cost, not just h (for compatibility -- really should return h
   instead).
 */

int l_bound(s)
State *s;
                                     /* lower bound on solution */
{
  register int h, i, j, *board;

  board = s->board;
  h = 0;
  for (i=0; i < sizesize; i++)
    if ((j = board[i]))              /* nonzero entries */
      h += abs(i / size - j / size) + abs(i % size - j % size);
  h += s->depth;
  return(h);
}

void u_bound(s, u)                   /* null for now: I need one */
State *s;                          /*  for these puzzles but I */
int *u;                              /*  know of nothing decent */
{
}

void print_solution(s)
register State *s;
```

```
{
  int i, j, timely;

  times(&buffer);
  time_is = buffer.tms_utime;
  timely = time_is - time_was;
  fprintf(filep, "Finding a schedule took %d0, timely);
  fprintf(filep, "%d nodes expanded.0, guessed);
  fprintf(filep, "Solution requires %d moves.0, s->where);

  i = 0;
  for (j = 0; j < s->where; j++) {
    putc(s->done[j], filep);
    if ((++i % 36) == 0)
      fprintf(filep, "0);
    else
      putc(' ', filep);
  }
  fprintf(filep, "0);
}

void copy_state(s, t)                          /* & */
State *s, *t;                                  /* source, target */
{                                            /* copy source-state... */
  int i, where;
  int *sboard, *tboard;
  char *sdone, *tdone;
  int *tno, *tkids;

  t->key = s->key;
  t->blank = s->blank;
  t->depth = s->depth;
  t->h = s->h;
  t->where = where = s->where;
  t->next_child = 0;
  t-> mother = 0;

  sboard = s->board;
  tboard = t->board;
  for (i = 0; i < sizesize; i++)
    tboard[i] = sboard[i];

  sdone = s->done;
  tdone = t->done;
  for (i = 0; i < where; i++)
    tdone[i] = sdone[i];

  tno = t->no;
  tkids = t->kids;
  for (i = 0; i < B; i++) {
    tno[i] = 0;
    tkids[i] = 0;
  }
}

State *make_state(s, move)
State *s;
int move;                                    /* job to schedule next */
{
```



```
register int blank, *board, last, new_blank, where;
register State *t;

t = (State *) malloc(sizeof(State));
copy_state(s, t);                            /* copy _only_ the goodies */

t->move = move;
++(t->depth);
board = t->board;
blank = t->blank;
switch(move) {
case L:
  new_blank = blank + 1;
  break;
case R:
  new_blank = blank - 1;
  break;
case D:
  new_blank = blank - size;
  break;
case U:
  new_blank = blank + size;
  break;
default:
  printf("attempted illegal move [make_state]0);
  break;
}
board[blank] = board[new_blank];
board[new_blank] = 0;
t->blank = new_blank;

where = t->where;
t->done[where++] = symbol[move];
t->where = where;

forbid(new_blank, t->no);
last = reverse[move];
t->no[last] = Treu;
++expanded;
++extant;
return(t);
}

/* Make a frame to be the root of the speculation tree.  This won't be
   based on any other frame, and it needs only the forbidden children from
   the initial state of the speculation.  Basically just a lot of
   initializing & clearing.
 */
void root_frame(fp, s)
Frame *fp;
State *s;
{
  int i;

  fp->move = -1;
  fp->value = Infinity;
  fp->which = -1;
  fp->depth = 0;
  fp->current_kid = 0;
```

```
  for (i = 0; i < B; i++) {
    fp->no[i] = s->no[i];
    s->no[i] = 0;
    fp->kids[i] = 0;
  }
}

State *make_virgin(s, q, u)
State *s;
Queue *q;
int u;
{
  int best, i, inc, kid_count;
  State *t;

  if (! s->mother)                          /* leaf or initial node */
    return(s);                              /* don't bother to make new */

  inc = Infinity;                         /* set incumbent to Infinity */
  kid_count = 0;
  for (i = 0; i< B; i++) {
    if (! s->no[i]) {
      int b;

      b = s->kids[i];
      if (b <= u) {
      if (b < inc) {
        inc = b;
        best = i;
      }
      }
      else
      ++kid_count;
    }
  }
  if (kid_count == B) {                     /* can't make a kid */
    --extant;
    free(s);
    return((State *) 0);
  }
                                          /* spawn it */
  t = make_state(s, best);                  /* make_state keeps count */
  if (kid_count == B - 1) {                 /* this is mum's last */
    --extant;
    free(s);
  }
  else {                                  /* life in the old girl yet */
    int b, key;

    s->no[best] = Treu;                     /* forbid the kid */
    key = Infinity;
    for (i = 0; i < B; i++)
      if ((! s->no[i]) && ((b = s->kids[i]) < key))
      key = b;
    s->mother = Treu;
    s->key = key;
    splay_enqueue(key, s->depth, (char *) s, q);
  }
  return(t);
```



```
}

int propagandalet(kid, dad, forbid_p)
Frame *kid, *dad;
int *forbid_p;                              /* return 1 -> next forbid. */
{
  int i, m, v;
  int blocked;

  m = kid->move;
  if (*forbid_p)                          /* this child is forbidden */
    dad->no[m] = Treu;                      /* set it so in the dad */
  else {                                  /* usual case: permitted */
    dad->kids[m] = v = kid->value;
    if (v < dad->value) {
      dad->value = v;
      dad->which = m;
    }
  }
  if (m == B - 1) {                         /* propagate */
    blocked = Treu;
    for (i = 0; i < B; i++)
      if (! (dad->no[i]) ) {
      blocked = Falsch;
      break;
      }
    *forbid_p = blocked;
    return(Treu);                         /* go on */
  }
  else
    return(Falsch);                         /* stop. */
}

void propaganda(stack, incumbent, where, common, forbid)
Frame *stack, *incumbent;
int where, common, forbid;
{
  int dad, propagate;

  --where;
  --common;
  dad = where - 1;

  do {
    propagate = propagandalet(&stack[where], &stack[dad], &forbid);
    if (common >= dad)
      propagate = propagandalet(&stack[where], &incumbent[dad], &forbid);
    --where;
    --dad;
  } while (propagate && (dad >= 0));
}

/* Make a frame as new...could easily inline this, but what the heck...
 */
void new_frame(p)
Frame *p;
{
  int i;
```

```
  for (i = 0; i < B; i++) {
    p->kids[i] = 0;
    p->no[i] = 0;
  }
  p->value = Infinity;
  p->which = -1;
  p->current_kid = 0;
  ++guessed;
}

/* Usual way to make a frame, based both on a local state and the parent frame.
   The new frame appears on the stack in the position after the parent's.
   Treat as a Boolean: 0 if the new frame cannot be made, 1 if it can.
   Check for legality, but test no bounds.  This is mostly a copy of
   make_state, but without the allocation.
 */

int make_frame(fp, s)
Frame *fp;
State *s;
{
  Frame *sp2;
  int blank, last, length, move, new_blank, where;
  int *board;

  move = fp->current_kid;
  if (move >= B)
    return(-1);                          /* indicate no more alt'ives */

  sp2 = fp + 1;
  if (fp->no[move]) {                    /* cannot expand */
    sp2->move = move;                    /* kludge for _propaganda_ */
    return(Falsch);                      /* indicate alt'ive invalid */
  }

  new_frame(sp2);
  sp2->move = move;
  sp2->depth = ++s->depth;

  board = s->board;
  blank = s->blank;
  switch(move) {
  case L:
    new_blank = blank + 1;
    break;
  case R:
    new_blank = blank - 1;
    break;
  case D:
    new_blank = blank - size;
    break;
  case U:
    new_blank = blank + size;
    break;
  default:
    printf("attempted illegal move0);
    break;
  }
  board[blank] = board[new_blank];
```



```
  board[new_blank] = 0;
  s->blank = new_blank;

  where = s->where;
  s->done[where++] = symbol[move];
  s->where = where;

  forbid(new_blank, sp2->no);
  last = reverse[move];
  sp2->no[last] = Treu;

  return(Treu);
}

/* Undo the frame just made, returning State _s_ to its previous condition.
   Advance current_kid (crucial!), etc. Void because we are discarding,
   not creating.

 */
void unmake_frame(fp, s)
Frame *fp;
State *s;
{
  Frame *sp2;
  int blank, last, length, move, new_blank, where;
  int *board;

  sp2 = fp + 1;
  move = sp2->move;

  board = s->board;
  blank = s->blank;
  switch(move) {
  case L:
    new_blank = blank - 1;
    break;
  case R:
    new_blank = blank + 1;
    break;
  case D:
    new_blank = blank + size;
    break;
  case U:
    new_blank = blank - size;
    break;
  default:
    printf("attempted illegal undoing of move0);
    break;
  }
  board[blank] = board[new_blank];
  board[new_blank] = 0;
  s->blank = new_blank;

  ++(fp->current_kid);
  --s->where;
  --s->depth;
}

State *reinstate(fp, s, u, valid)
```

```
Frame *fp;
State *s;
int u;
int *valid;
{
  Frame *fp2, *fp3;
  State *t;
  int i;
  int done, l, which;

  fp2 = fp + 1;
  which = fp2->move;
  t = make_state(s, which);

  if (fp2->which < 0) {                    /* leaf */
    t->mother = Falsch;                    /* not already a mother */
    done = Treu;
    if ( (t->key = fp2->value) <= u)
      for (i = 0; i < B; i++)
      if (! (t->no[i] = fp2->no[i]))
        done = Falsch;
  }
  else {                                   /* not a leaf */
    t->mother = Treu;                      /* hence already a mother */
    fp3 = fp2 + 1;
    which = fp3->move;
    fp2->no[which] = Treu;                 /* forbid child == move */
    done = Treu;
    l = Infinity;
    for (i = 0; i < B; i++)
      if (! (t->no[i] = fp2->no[i])) {
      int b;

      t->kids[i] = b = fp2->kids[i];
      if (b <= u) {
        done = Falsch;
        if (b < l)
          l = b;
      }
      else
        t->no[i] = Treu;
      }
    t->key = l;
  }
  *valid = done;
  return(t);
}

void enqueue_path(fp, s, q, u, inc_depth)
Frame *fp;
State *s;
Queue *q;
int u, inc_depth;
{
  State *t;                                /* new states: use this */
  State *trash[StackSize], **tp;           /* trash-states */
  Frame *fp2;
  int i;
  int done, l, which;
```



```
  tp = trash;
                                           /* special case for initial */
  fp2 = fp + 1;
  which = fp2->move;
  fp->no[which] = Treu;                    /* forbid move == child's */
  done = Treu;
  l = Infinity;
  for (i = 0; i < B; i++)
    if (! (s->no[i] = fp->no[i])) {        /* a child is yet possible */
      int b;

      s->kids[i] = b = fp->kids[i];        /* load only for valid kids */
      if (b <= u) {
      done = Falsch;
      if (b < l)
        l = b;
      }
      else
      s->no[i] = Treu;
    }

  if (! done) {                            /* put it back */
    s->key = l;
    s->mother = Treu;                      /* just to be certain */
    splay_enqueue(l, s->depth, (char *) s, q);
  }
  else
    *(tp++) = s;                           /* liber scriptus proferetur */

  for (i = 1; i < inc_depth; i++) {        /* other cases */
    done = Falsch;
    t = reinstate(fp, s, u, &done);
    ++fp;
    if (! done)
      splay_enqueue(t->key, t->depth, (char *) t, q);
    else
      *(tp++) = t;
    s = t;
  }

  --tp;
  while (tp >= trash) {
    free(*tp);
    --tp;
    --extant;
  }
}

/* Copy the contents of the source frame to the sink frame.  Zero anything
   that is useless within the incumbent.
 */
void copy_frame(source, sink)
Frame *source, *sink;
{
  int i;

  sink->move = source->move;
  sink->value = source->value;
  sink->which = source->which;
```

```
  sink->depth = source->depth;
  sink->current_kid = 0;
  for (i = 0; i < B; i++) {
    sink->kids[i] = source->kids[i];          /* let's see if the compiler */
    sink->no[i] = source->no[i];              /*  is any good, eh? */
  }
}

/* Copy the contents of the stack from the source stack to the sink stack,
   using copy_frame to do it frame by frame.
   Examples: for a new incumbent when the last was a brother, source and
   sink point to the last entry of their stacks, and we do exactly one copy.
   For the first incumbent, we start at the bases of the stacks and do
   depth - 0 = depth copies.
 */
void copy_stack(source, sink, entries)
Frame *source, *sink;
int entries;
{
  int i;

  for (i = 0; i < entries; i++)
    copy_frame(&source[i], &sink[i]);
}

State *branch(s, q, u)
State *s;                               /*  solution */
Queue *q;
int *u;
{
  Frame stack[StackSize], incumbent[StackSize], *sp;
  State *t, *inc_state;
  int common, current, id_inc, inc, inc_depth, l, can;

  if (! s->mother)
    id_inc = s->key;
  else
    id_inc = s->key + depth;
  t = make_virgin(s, q, *u);
  if (! t)                              /* can't make a child */
    return (State *) 0;
  if (solution(t)) {                          /* catch virgins */
    l = l_bound(t);
    if (l <= *u) {
      *u = l - 1;
      inc_state = (State *) malloc(sizeof(State));
      copy_state(t, inc_state);
      inc_state->key = l;
      return inc_state;
    }
    else
      return (State *) 0;
  }

  common = 0;                                 /* only initially; not again */
  current = 1;                                /* to keep upper limit excl. */
  inc = Infinity;
  sp = stack;
  root_frame(sp, t);
```



```
inc_state = (State *) 0;

do {                                          /* main loop */
  if ( (can = make_frame(sp, t)) > 0) { /* begin CAN MAKE */
    l = l_bound(t);
    if (l <= *u) {
    if (solution(t)) {
      *u = l - 1;
      if (! inc_state)
        inc_state = (State *) malloc(sizeof(State));
      copy_state(t, inc_state);
      inc_state->key = l;

      propaganda(stack, incumbent, current + 1, common, Treu);
      unmake_frame(sp, t);
    }
    else {                                    /* begin USUAL CASE */
      ++sp;
      ++current;
      if (l > id_inc) {                       /* begin LEAVES */
        sp->value = l;                        /* for propagation */
        propaganda(stack, incumbent, current, common, Falsch);
        if (l < inc) {                        /* new incumbent */
          u_bound(s, u);
          inc = l;
          copy_stack(&stack[common], &incumbent[common], current - common);
          common = current - l;               /* exclusive limit */
          inc_depth = current;                /* depth of this leaf */
        }
        --current;                            /* can go no deeper: b'track */
        --sp;
        unmake_frame(sp, t);
      }                                       /* end LEAVES */
    }                                         /* end USUAL CASE */
    }                                         /* end KEEP THIS STATE */
    else {                                    /* l-bound is too large */
    propaganda(stack, incumbent, current + 1, common, Treu);
    unmake_frame(sp, t);
    }
  }                                           /* end CAN MAKE */
  else if (can < 0) {                         /* can't make; no others */
    --sp;                                     /* backtrack */
    --current;
    if (current < common)
    common = current;
    if (sp >= stack)                          /* kludge */
    unmake_frame(sp, t);
  }
  else {                                      /* exactly == 0 */
    propaganda(stack, incumbent, current + 1, common, Treu);
    ++(sp->current_kid);
  }
} while (sp >= stack);
if (inc < Infinity)                           /* made it to the leaves */
  enqueue_path(incumbent, t, q, *u, inc_depth);
return(inc_state);
}
```

```
/*****************************************************************************
  splay.h: definitions for splay-tree package
 *****************************************************************************/

/* Splay trees for priority queues.  Definitions.
   Version for queue-of-queues implementation
 */

#define MaxStack 16384

#define Front 0                         /* front of Queuelet */
#define Back 1                          /* back of Queuelet: depths */

#define Big 0x7fffffff                  /* very big number */

typedef struct node {                   /* different values */
  struct node *left;
  struct node *right;
  struct queuelet *in;                  /* nodes of identical value */
  int key;
} Node;

typedef struct nodelet {                /* different depths */
  struct nodelet *left;
  struct nodelet *right;
  struct nodeletlet *in;
  int key;
  char *data;
} Nodelet;

typedef struct nodeletlet {             /* both keys identical */
  struct nodeletlet *in;
  char *data;                           /* cast to suitable type */
} Nodeletlet;

typedef struct queue {                  /* hysterical raisins */
  Node *root;                           /* root of splay tree */
} Queue;

typedef struct queuelet {
  Nodelet *root;
} Queuelet;

extern struct queue *splay_init();
extern void splay_enqueue();
extern splay_prune();
extern char *splay_dequeue();


/*****************************************************************************
  splay.c: routines for priority queues (continued in "splaylet.c")
 *****************************************************************************/

/* Splay trees for priority queues.  Code, after Sleator & Tarjan, JACM 32(3),
   p. 668-9.

   Version for "queues of queues" on double keys.  Each member of the queue is
   a queue itself, handled by the routines in the "splaylet.c" file.
```



```
 */

extern struct queuelet *splaylet_init();
extern void splaylet_enqueue();
extern splaylet_prune();
extern char *splaylet_dequeue();
extern char *splaylet_dequeue2();

#include "splay.h"

Queue *splay_init()
{
  Queue *q;

  q = (Queue *) malloc(sizeof(Queue));
  q->root = (Node *) 0;
  return(q);
}

void splay_enqueue(i, j, s, q)
register int i;                         /* primary key */
int j;                                  /* secondary key */
char *s;
Queue *q;
{
  register Node *t;

  if ( (t = q->root) ) {                /* queue is not empty */
    Node Null;
    register int k;
    register Node *l, *r;

    l = r = &Null;
    l->left = (Node *) 0;
    l->right = (Node *) 0;

    do {
      k = t->key;
      if (i < k) {                      /* go left */
      register Node *left;

      if ( (left = t->left) ) {
        register int lk;

        lk = left->key;
        if (i < lk) {                   /* zig-zig: rotate r, link r */
          t->left = left->right;
          left->right = t;
          t = left->left;
          left->left = (Node *) 0;
          r->left = left;
          r = left;
        }
        else if (i > lk) {              /* zig-zag: link r, link l */
          r->left = t;
          r = t;
          l->right = left;
          t->left = (Node *) 0;
          t = left->right;
```

```
left->right = (Node *) 0;
    l = left;
  }
  else {                          /* match: zig: link right */
    r->left = t;
    r = t;
    r->left = (Node *) 0;
    t = left;
    break;
  }
}
else {                                  /* empty: zig: link right */
  r->left = t;
  r = t;
  r->left = (Node *) 0;
  t = left;
  break;
}
}
else if (i > k) {                       /* go right */
register Node *right;

if ( (right = t->right) ) {
  register int rk;

  rk = right->key;
  if (i < rk) {                         /* zig-zag: link l, link r */
    l->right = t;
    l = t;
    r->left = right;
    t->right = (Node *) 0;
    t = right->left;
    right->left = (Node *) 0;
    r = right;
  }
  else if (i > rk) {                    /* zig-zig: rotate l, link l */
    t->right = right->left;
    right->left = t;
    t = right->right;
    right->right = (Node *) 0;
    l->right = right;
    l = right;
  }
  else {                          /* match: zig: link left */
    l->right = t;
    l = t;
    l->right = (Node *) 0;
    t = right;
    break;
  }
}
else {                                  /* empty: zig: link left */
  l->right = t;
  l = t;
  l->right = (Node *) 0;
  t = right;
  break;
}
}
```



```
  else                            /* match, straight off */
    break;
  } while (t);
  if (t) {                                  /* node already exists */
    splaylet_enqueue(j, s, t->in);
    l->right = t->left;
    r->left = t->right;
  }
  else {                                    /* create a new node */
    t = (Node *) malloc(sizeof(Node));
    t->in = splaylet_init();
    t->key = i;
    splaylet_enqueue(j, s, t->in);
  }
  t->right = Null.left;
  t->left = Null.right;
  q->root = t;
 }
 else {                                     /* queue is empty: init. */
  t = (Node *) malloc(sizeof(Node));
  t->left = (Node *) 0;
  t->right = (Node *) 0;
  t->in = splaylet_init();
  t->key = i;
  splaylet_enqueue(j, s, t->in);
  q->root = t;
 }
}

/* Prune all nodes with values strictly greater than _i_.  Splay the last node
   less than or equal to i to the root, which creates a right tree that can be
   discarded.
 */

splay_prune(i, q)
int i;
Queue *q;
{
  register Node *t;

  if ( (t = q->root) ) {                    /* queue is not empty */
    register int k;
    Node Null;
    register Node *l, *r;

    l = r = &Null;
    l->left = (Node *) 0;
    l->right = (Node *) 0;
    l->in = (Queuelet *) 0;                 /* beware... */

    do {
      k = t->key;
      if (i > k) {                          /* go right */
      register Node *right;

      if ( (right = t->right) ) {
        register int rk;

        rk = right->key;
```

```
if (i > rk) {                   /* zig-zig: rotate l, link l */
      t->right = right->left;
      right->left = t;
      t = right->right;
      right->right = (Node *) 0;
      l->right = right;
      l = right;
    }
    else if (i < rk) {                  /* zig-zag: link l, link r */
      l->right = t;
      l = t;
      r->left = right;
      t->right = (Node *) 0;
      t = right->left;
      right->left = (Node *) 0;
      r = right;
    }
    else {
      l->right = t;
      l = t;
      l->right = (Node *) 0;            /* new hack! */
      t = right;
      break;
    }
  }
  else {                                /* zig: link left */
    l->right = t;
    l = t;
    l->right = (Node *) 0;              /* new hack! */
    t = right;
    break;
  }
  }
  else if (i < k) {                     /* go left */
  register Node *left;

  if ( (left = t->left) ) {
    register int lk;

    lk = left->key;
    if (i > lk) {                       /* zig-zag: link r, link l */
      r->left = t;
      r = t;
      l->right = left;
      t->left = (Node *) 0;
      t = left->right;
      left->right = (Node *) 0;
      l = left;
    }
    else if (i < lk) {                  /* zig-zig: rotate r, link r */
      t->left = left->right;
      left->right = t;
      t = left->left;
      left->left = (Node *) 0;
      r->left = left;
      r = left;
    }
    else {
      r->left = t;
```



```
      r = t;
      r->left = (Node *) 0;             /* new hack! */
      t = left;
      break;
    }
  }
  else {                                /* zig: link right */
    r->left = t;
    r = t;
    r->left = (Node *) 0;               /* new hack! */
    t = left;
    break;
  }
  }
  else                              /* exact match */
  break;
} while(t);

if (t) {                                /* a node exists of value i */
  register Node *x;

  l->right = t->left;
  t->left = Null.right;
  q->root = t;

  if ( (x = t->right) ) {               /* FREE rh tree of t */
  if (! (x->left && x->right)) {            /* only one value */
    splaylet_prune(-1, x->in);          /* remove whole queuelet */
    free(x->in);                        /* remove queuelet-header */
    free(x);
  }
  else {
    Node *stack[MaxStack];
    register Node *l, *r, **sp;

    sp = stack;                         /* point to bottom of stack */
    while (sp >= stack) {               /* stack op every iteration */
      if (x) {                          /* can operate on current x */
        splaylet_prune(-1, x->in);
        free(x->in);
        l = x->left;
        r = x->right;
        free(x);
        x = l;
        if (r)
        *(sp++) = r;
      }
      else                          /* current x is nil */
        x = *(--sp);
    }
  }
  t->right = (Node *) 0;                /* sever */
  }

  if ( (x = Null.left) )                /* FREE anything in R tree */
  if (! (x->left && x->right)) {            /* only one node */
    splaylet_prune(-1, x->in);
    free(x->in);
    free(x);
```

```
}
   else {
     Node *stack[MaxStack];
     register Node *l, *r, **sp;

     sp = stack;                          /* point to bottom of stack */
     while (sp >= stack) {                /* stack op every iteration */
       if (x) {                           /* can operate on current x */
         splaylet_prune(-1, x->in);
         free(x->in);
         l = x->left;
         r = x->right;
         free(x);
         x = l;
         if (r)
         *(sp++) = r;
       }
       else                         /* current x is nil */
         x = *(--sp);
     }
   }
 }
 else {                             /* no node exists of value i */
   register Node *x;

   q->root = Null.right;

   if ( (x = Null.left) ) {                  /* FREE anything in R tree */
   if (! (x->left && x->right)) {            /* only one node */
     splaylet_prune(-1, x->in);
     free(x->in);
     free(x);
   }
   else {
     Node *stack[MaxStack];
     register Node *l, *r, **sp;

     sp = stack;                          /* point to bottom of stack */
     while (sp >= stack) {                /* stack op every iteration */
       if (x) {                           /* can operate on current x */
         splaylet_prune(-1, x->in);
         free(x->in);
         l = x->left;
         r = x->right;
         free(x);
         x = l;
         if (r)
         *(sp++) = r;
       }
       else                         /* current x is nil */
         x = *(--sp);
     }
   }
   }
   return(1);
 }
}
else {                              /* queue is empty: */
  return(0);
```



```
  }
}

/* Return head of priority queue.  Splay the tree on the value 0, which puts
   everything in the tree R.  Note that we don't need to know how to do cases
   other than lefthanded zig and zig-zig, which saves tests & registers.

   Revised: now it returns only the state, not the head proper.  The head node
   it deletes...
 */

char *splay_dequeue(q)
Queue *q;
{
  register Node *t;

  if ( (t = q->root) ) {                    /* queue is not empty */
    Node Null;
    register Node *r, *dad;

    r = &Null;
    r->left = (Node *) 0;
    r->right = (Node *) 0;

    do {
      register Node *left;

      dad = r;                              /* save last value */
      if ( (left = t->left) ) {
      t->left = left->right;                /* zig-zig */
      left->right = t;
      t = left->left;
      left->left = (Node *) 0;
      r->left = left;
      r = left;
      }
      else {                                /* zig: link right */
      r->left = t;
      r = t;
      r->left = (Node *) 0;
      t = left;
      break;
      }
    } while(t);

/* t points to nothing.  L is empty (we don't even have code that looks at it).
   R contains everything else.  r points to the leftmost node of R, which is
   the node we want. dad points to its father.  r->left necessarily is nil,
   but r->right may be a subtree.  We know that this subtree has items of
   key less than dad's, so we set dad->left to it.  We delete x (but not its
   state -- that's the duty of the caller).

   Of course, if r is a portmanteau node, we simply return its state, promote
   the state of the first member of its string, & delete not r but that 1st
   member.
 */
    {
      register Queuelet *in;
      char *s;
```

```
    in = r->in;
    s = splaylet_dequeue2(in);
    if (! in->root) {                   /* all out of nodelets */
    dad->left = r->right;
    free(in);                           /* remove queuelet */
    free(r);                            /* remove nodelet */
    }
    q->root = Null.left;
    return(s);
   }
  }
  else
   return((char *) 0);
}


/*****************************************************************************
  splaylet.c: remainder of splay tree package
 *****************************************************************************/

/* Splay trees for priority queues.  Code, after Sleator & Tarjan, JACM 32(3),
   p. 668-9.

   Routines for handling little lists on a second key.  This package makes
   a "top level" priority queue, each node of which contains another
   priority queue on values of a second key.  Procedures in this file handle
   these "lower level" queues.
 */

#include "splay.h"

Queuelet *splaylet_init()
{
  Queuelet *q;

  q = (Queuelet *) malloc(sizeof(Queuelet));
  q->root = (Nodelet *) 0;
  return(q);
}

void splaylet_enqueue(i, s, q)
register int i;
char *s;
Queuelet *q;
{
  register Nodelet *t;

  if ( (t = q->root) ) {                /* queue is not empty */
    Nodelet Null;
    register int k;
    register Nodelet *l, *r;

    l = r = &Null;
    l->left = (Nodelet *) 0;
    l->right = (Nodelet *) 0;

    do {
      k = t->key;
      if (i < k) {                      /* go left */
```



```
register Nodelet *left;

    if ( (left = t->left) ) {
      register int lk;

      lk = left->key;
      if (i < lk) {                     /* zig-zig: rotate r, link r */
        t->left = left->right;
        left->right = t;
        t = left->left;
        left->left = (Nodelet *) 0;
        r->left = left;
        r = left;
      }
      else if (i > lk) {                /* zig-zag: link r, link l */
        r->left = t;
        r = t;
        l->right = left;
        t->left = (Nodelet *) 0;
        t = left->right;
        left->right = (Nodelet *) 0;
        l = left;
      }
      else {                        /* match: zig: link right */
        r->left = t;
        r = t;
        r->left = (Nodelet *) 0;
        t = left;
        break;
      }
    }
    else {                              /* empty: zig: link right */
      r->left = t;
      r = t;
      r->left = (Nodelet *) 0;
      t = left;
      break;
    }
    }
    else if (i > k) {                   /* go right */
    register Nodelet *right;

    if ( (right = t->right) ) {
      register int rk;

      rk = right->key;
      if (i < rk) {                     /* zig-zag: link l, link r */
        l->right = t;
        l = t;
        r->left = right;
        t->right = (Nodelet *) 0;
        t = right->left;
        right->left = (Nodelet *) 0;
        r = right;
      }
      else if (i > rk) {                /* zig-zig: rotate l, link l */
        t->right = right->left;
        right->left = t;
        t = right->right;
```

```
    right->right = (Nodelet *) 0;
        l->right = right;
        l = right;
      }
      else {                          /* match: zig: link left */
        l->right = t;
        l = t;
        l->right = (Nodelet *) 0;
        t = right;
        break;
      }
    }
    else {                                /* empty: zig: link left */
      l->right = t;
      l = t;
      l->right = (Nodelet *) 0;
      t = right;
      break;
    }
    }
    else                              /* match, straight off */
    break;
  } while (t);
  if (t) {                                /* node already exists */
    Nodeletlet *n;

    n = (Nodeletlet *) malloc(sizeof(Nodeletlet));
    n->in = t->in;                        /* often 0, but what matter? */
    t->in = n;
    n->data = t->data;
    l->right = t->left;
    r->left = t->right;
  }
  else {                              /* create a new node */
    t = (Nodelet *) malloc(sizeof(Nodelet));
    t->in = (Nodeletlet *) 0;
    t->key = i;
  }
  t->data = s;
  t->right = Null.left;
  t->left = Null.right;
  q->root = t;
 }
 else {                               /* queue is empty: init. */
   t = (Nodelet *) malloc(sizeof(Nodelet));
   t->left = (Nodelet *) 0;
   t->right = (Nodelet *) 0;
   t->in = (Nodeletlet *) 0;
   t->data = s;
   t->key = i;
   q->root = t;
 }
}

/* Prune all nodes with values strictly greater than __i__.  Splay the last node
   less than or equal to i to the root, which creates a right tree that can be
   discarded.
 */
```



```
void splaylet_prune(i, q)
int i;
Queuelet *q;
{
  register Nodelet *t;

  if ( (t = q->root) ) {                  /* queue is not empty */
    register int k;
    Nodelet Null;
    register Nodelet *l, *r;

    l = r = &Null;
    l->left = (Nodelet *) 0;
    l->right = (Nodelet *) 0;
    l->in = (Nodeletlet *) 0;

    do {
      k = t->key;
      if (i > k) {                        /* go right */
      register Nodelet *right;

      if ( (right = t->right) ) {
        register int rk;

        rk = right->key;
        if (i > rk) {                     /* zig-zig: rotate l, link l */
          t->right = right->left;
          right->left = t;
          t = right->right;
          right->right = (Nodelet *) 0;
          l->right = right;
          l = right;
        }
        else if (i < rk) {                /* zig-zag: link l, link r */
          l->right = t;
          l = t;
          r->left = right;
          t->right = (Nodelet *) 0;
          t = right->left;
          right->left = (Nodelet *) 0;
          r = right;
        }
        else {
          l->right = t;
          l = t;
          l->right = (Nodelet *) 0;       /* new hack! */
          t = right;
          break;
        }
      }
      else {                              /* zig: link left */
        l->right = t;
        l = t;
        l->right = (Nodelet *) 0;         /* new hack! */
        t = right;
        break;
      }
      }
      else if (i < k) {                   /* go left */
```

```
register Nodelet *left;

    if ( (left = t->left) ) {
      register int lk;

      lk = left->key;
      if (i > lk) {                     /* zig-zag: link r, link l */
        r->left = t;
        r = t;
        l->right = left;
        t->left = (Nodelet *) 0;
        t = left->right;
        left->right = (Nodelet *) 0;
        l = left;
      }
      else if (i < lk) {                /* zig-zig: rotate r, link r */
        t->left = left->right;
        left->right = t;
        t = left->left;
        left->left = (Nodelet *) 0;
        r->left = left;
        r = left;
      }
      else {
        r->left = t;
        r = t;
        r->left = (Nodelet *) 0;        /* new hack! */
        t = left;
        break;
      }
    }
    else {                              /* zig: link right */
      r->left = t;
      r = t;
      r->left = (Nodelet *) 0;              /* new hack! */
      t = left;
      break;
    }
    }
    else                            /* exact match */
    break;
  } while(t);

  if (t) {                              /* a node exists of value i */
    register Nodelet *x;

    l->right = t->left;
    t->left = Null.right;
    q->root = t;

    if ( (x = t->right) ) {             /* FREE rh tree of t */
    if (! (x->left && x->right)) {          /* only one value */
      register Nodeletlet *in;

      if ( (in = x->in) ) {
        register Nodeletlet *in2;

        while (in) {
          in2 = in;
```



```
          in = in->in;
          free(in2->data);
          free(in2);
        }
      }
      free(x->data);
      free(x);
    }
    else {
      Nodelet *stack[MaxStack];
      register Nodelet *l, *r, **sp;

      sp = stack;                       /* point to bottom of stack */
      while (sp >= stack) {             /* stack op every iteration */
        if (x) {                        /* can operate on current x */
          register Nodeletlet *in;

          if ( (in = x->in) ) {
          register Nodeletlet *in2;

          while (in) {
            in2 = in;
            in = in->in;
            free(in2->data);
            free(in2);
          }
          }
          free(x->data);
          l = x->left;
          r = x->right;
          free(x);
          x = l;
          if (r)
          *(sp++) = r;
        }
        else                        /* current x is nil */
          x = *(--sp);
      }
    }
    t->right = (Nodelet *) 0;               /* saver */
    }

    if ( (x = Null.left) )              /* FREE anything in R tree */
    if (! (x->left && x->right)) {          /* only one node */
      register Nodeletlet *in;

      if ( (in = x->in) ) {
        register Nodeletlet *in2;

        while (in) {
          in2 = in;
          in = in->in;
          free(in2->data);
          free(in2);
        }
      }
      free(x->data);
      free(x);
    }
```

```
else {
      Nodelet *stack[MaxStack];
      register Nodelet *l, *r, **sp;

      sp = stack;                          /* point to bottom of stack */
      while (sp >= stack) {                /* stack op every iteration */
        if (x) {                           /* can operate on current x */
          register Nodeletlet *in;

          if ( (in = x->in) ) {
          register Nodeletlet *in2;

          while (in) {
            in2 = in;
            in = in->in;
            free(in2->data);
            free(in2);
          }
          }
          free(x->data);
          l = x->left;
          r = x->right;
          free(x);
          x = l;
          if (r)
          *(sp++) = r;
        }
        else                           /* current x is nil */
          x = *(--sp);
      }
    }
  }
  else {                               /* no node exists of value i */
    register Nodelet *x;

    q->root = Null.right;

    if ( (x = Null.left) ) {                    /* FREE anything in R tree */
    if (! (x->left && x->right)) {              /* only one node */
      register Nodeletlet *in;

      if ( (in = x->in) ) {
        register Nodeletlet *in2;

        while (in) {
          in2 = in;
          in = in->in;
          free(in2->data);
          free(in2);
        }
      }
      free(x->data);
      free(x);
    }
    else {
      Nodelet *stack[MaxStack];
      register Nodelet *l, *r, **sp;

      sp = stack;                          /* point to bottom of stack */
```



```
      while (sp >= stack) {                /* stack op every iteration */
        if (x) {                           /* can operate on current x */
          register Nodeletlet *in;

          if ( (in = x->in) ) {
          register Nodeletlet *in2;

          while (in) {
            in2 = in;
            in = in->in;
            free(in2->data);
            free(in2);
          }
          }
          free(x->data);
          l = x->left;
          r = x->right;
          free(x);
          x = l;
          if (r)
          *(sp++) = r;
        }
        else                           /* current x is nil */
          x = *(--sp);
      }
    }
    }
  }
}

/* Mirror-image version for getting tail of queuelet.  Left is right, etc.
 */
char *splaylet_dequeue2(q)
Queuelet *q;
{
  register Nodelet *t;

  if ( (t = q->root) ) {                    /* queue is not empty */
    Nodelet Null;
    register Nodelet *l, *dad;

    l = &Null;
    l->left = (Nodelet *) 0;
    l->right = (Nodelet *) 0;

    do {
      register Nodelet *right;

      dad = l;                              /* save last value */
      if ( (right = t->right) ) {
      t->right = right->left;               /* zig-zig */
      right->left = t;
      t = right->right;
      right->right = (Nodelet *) 0;
      l->right = right;
      l = right;
      }
      else {                                /* zig: link left */
```

```
 l->right = t;
     l = t;
     l->right = (Nodelet *) 0;
     t = right;
     break;
     }
   } while(t);

/* Extended description of current state has been omitted...should just be
   mirror image, anyway, of other case.
*/
   {
     register Nodeletlet *in;
     char *s;

     if ( (in = l->in) ) {                    /* a portmanteau node */
     l->in = in->in;
     s = l->data;
     l->data = in->data;
     free(in);
     }
     else {
     dad->right = l->left;                    /* replace son with grandson */
     s = l->data;
     free(l);
     }
     q->root = Null.right;
     return(s);
   }
 }
 else
   return((char *) 0);
}
```


VITA

Markian M. Gooley was born on July 26, 1961 in Decatur, Illinois. He received his Bachelor of Science in Mathematics *summa cum laude* in August of 1981 from Millikin University, and his Master of Science in Computer Science from the University of Illinois at Urbana-Champaign in January of 1985. At the University of Illinois, he was employed as a teaching assistant in 1982 and as a research assistant from 1983 to 1989. He spent the summer of 1989 at NASA Ames Research Center at Moffet Field, California, then continued study at the University of Illinois, holding a NASA Graduate Fellowship from August 1989 through August 1990.

His research interests include logic programming, heuristic search, combinatorial optimization, and parallel processing.