Louisiana State University

LSU Scholarly Repository

LSU Doctoral Dissertations | Graduate School

2003

Machine learning techniques for efficient query processing in kowledge base systems

Kevin Paul Grant

Louisiana State University and Agricultural and Mechanical College

Follow this and additional works at: https://repository.lsu.edu/gradschool_dissertations

Part of the Computer Sciences Commons

Recommended Citation

Grant, Kevin Paul, "Machine learning techniques for efficient query processing in kowledge base systems" (2003). *LSU Doctoral Dissertations*. 2952.
https://repository.lsu.edu/gradschool_dissertations/2952

This Dissertation is brought to you for free and open access by the Graduate School at LSU Scholarly Repository. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Scholarly Repository. For more information, please contactgradetd@lsu.edu.

MACHINE LEARNING TECHNIQUES FOR EFFICIENT QUERY PROCESSING IN KNOWLEDGE BASE SYSTEMS

A Dissertation
Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Kevin Paul Grant
B.A., University of New Orleans, 1989
B.S., University of New Orleans, 1990
M.S., University of New Orleans, 1994
December 2003

©Copyright 2003
Kevin Paul Grant
All rights reserved

Acknowledgments

I would like to gratefully acknowledge:

- The long-suffering patience of my major professor, Dr. Jianhua Chen, in helping me to bring this project to a successful conclusion.
- In no particular order, the rest of my committee: Dr. Robert Mathews (my minor professor from psychology), Dr. Evangelos Triantaphyllou (the dean's representative), Dr. Sukhamay Kundu and Dr. Donald Kraft for their help and advice with this project.
- The equally long-suffering efforts of my parents, Newton and Heloise Grant, in helping me to get this far through the educational process.
- My brother and sister-in-law, David and Karen Grant, for some much needed help with transportation.
- Michael Schmauss, without whom I would not have had the computer equipment that I needed, when I needed it.

Table of Contents

Acknowledgments . . . . **iii**

List of Tables . . . . **vi**

List of Figures . . . . **vii**

Abstract . . . . **ix**

1 Introduction **1**
1.1 Background 1

2 The Problem **3**
2.1 A Sample Query 3
2.2 Combinatorial Explosion 4
2.3 Motivation 8
2.4 Some Issues 9
2.5 Our Approach 11
2.6 Related Work 13

3 Our Approach: Probabilistic Heuristic Estimates **18**
3.1 The System Architecture 18
3.2 Operation of the Central Controller 20
3.3 Operation of a Knowledge Base 21
3.4 Probabilistic Selection 21

4 Solution: Choosing the Best Subquery to Expand Next **23**
4.1 Computing the Probabilities 23
4.2 Useful Properties for PHE-based Query Scoring 25
4.3 Choosing Which Subquery to Process (Best-First Search) 27
4.4 How Much Confidence Should We Have in Our Selection (Best-First Selection)? 27
4.4.1 A Correct Selection 28
4.4.2 Best-First Selection 29
4.5 Choosing Which Subqueries to Process (Probabilistically) 29
4.6 How Much Confidence Should We Have in Our Choices (Probabilistic Selection)? 29

4.6.1 The Selection of Subqueries for Inclusion in TEMPSET 29
4.6.2 The Selection of a Subquery from TEMPSET for Processing 39
4.7 Exploration vs. Exploitation 49
4.8 Choosing the Best Conjunct to Process Next 49

5 The Knowledge Representation and Query Language 51
5.1 Part 1: The Attribute Base 51
5.2 Part 2: Predicate Instance Database Entries 51
5.3 Part 3: Relation Instance Database Entries 52
5.4 Part 4: is-a Hierarchy Entries 54
5.5 Part 5: Rulebase Entries 54
5.6 Queries 55
5.7 Answers 55

6 Training the System 56
6.1 Training the System to Choose Subqueries for Further Processing 56
6.2 How Big Should the Set of Training Examples Be? 58
6.3 Case 1 ($+\varepsilon_{final}$): The Math 60
6.4 Case 1 ($+\varepsilon_{final}$): The Resulting Numbers 62
6.5 Case 2 ($-\varepsilon_{final}$): The Math 64
6.6 Case 2 ($-\varepsilon_{final}$): The Resulting Numbers 66
6.7 The Final Result (math and numbers) 68
6.8 Generating the Training Examples 69

7 The Experiment 71
7.1 The Algorithms to be Tested 71
7.2 The Measurements to be Made 72
7.3 Training the System 73
7.4 Selecting the Heuristics to be Used 74
7.5 Details of the Experiment 75
7.6 A Sample Query 76
7.7 The Results 79
7.7.1 Nodes Expanded Averaged Over All Queries Solved 79
7.7.2 Nodes Expanded Averaged Over All Queries Attempted 82
7.7.3 Solution Path Length Averaged Over All Queries Solved 82

8 Summary 86

9 Future Work 87

Bibliography 88

Vita 92

List of Tables

5.1 Table for predicate103 52

5.2 Non-Generic Table for dog 52

5.3 Table for relation103 53

5.4 Non-Generic Table for owns 53

6.1 Case1: Values for the Numerator 63

6.2 Case1: Values for the Denominator 63

6.3 Case 1: ε_{final} as a function of P_{max} and $P_{success}$ 64

6.4 Case2: Values for the Numerator 67

6.5 Case2: Values for the Denominator 67

6.6 Case 2: ε_{final} as a function of P_{max} and $P_{success}$ 68

6.7 Final Values: ε_{final} as a function of P_{max} and $P_{success}$ 69

7.1 Algorithm Performance for Solved Queries (the numbers) 80

7.2 Algorithm Performance for All Queries (the numbers) 83

7.3 Algorithm Scores for All Queries 85

7.4 Path Length for Solved Queries (the numbers) 85

List of Figures

2.1 The Initial Query 5

2.2 The Initial Query As Root 5

2.3 Children of the Initial Query 5

2.4 Processing the Initial Query 6

2.5 An Answer for the Initial Query 7

2.6 Shortest Successful Paths 8

2.7 A BPS Search Space 14

2.8 An Expanded BPS Search Space 15

2.9 A Re-Expanded BPS Search Space 15

3.1 System Architecture 19

3.2 Central Controller Operation 20

4.1 What are the odds that the highlighted node is on a SSP? 23

4.2 Best-First Method for Choosing Which Subquery to Expand Next 27

4.3 Correct Selection 28

4.4 Probabilistic Method for Choosing Which Subqueries to Process Next 30

4.5 The Hybrid Algorithm 31

4.6 Usable Space for Probabilistic Selection When RANDOM = 0.5 32

4.7 Usable Space When RANDOM = 0.0 33

4.8 Usable Space When RANDOM = 0.25 33

4.9 Usable Space When RANDOM = 0.5 34

4.10 Usable Space When RANDOM = 0.75 34

4.11 Usable Space When RANDOM = 1.0 35

4.12 Usable Space for Non-Worst Case When RANDOM = 0.5 36

4.13 Non-Worst Case Usable Space When RANDOM = 0.0 36

4.14 Non-Worst Case Usable Space When RANDOM = 0.25 37

4.15 Non-Worst Case Usable Space When RANDOM = 0.5 37

4.16 Non-Worst Case Usable Space When RANDOM = 0.75 38

4.17 Non-Worst Case Usable Space When RANDOM = 1.0 38

4.18 Percentage of Usable Subspace as a Function of ε 40

4.19 Symmetry of Selection When $P_e = 0.5$ 44

4.20 Asymmetry of Selection When $P_e = 0.25$ 44

4.21 Asymmetry of Selection When $P_e = 0.75$ 45

4.22 Probabilities for Selection of Subqueries for TEMPSET 46

4.23 Probability of a Correct Selection as a Function of ε 48

6.1 Method for Training the System to Choose Subqueries 57

6.2 Approach for Training the System to Choose Subqueries 58

7.1 Progress Through A Sample Query (The Hybrid Algorithm) 77

7.2 Algorithm Performance (for solved queries) 80

7.3 Hybrid vs. Best-First Algorithm Comparison 81

7.4 Algorithm Performance (for all queries) 83

7.5 Nodes In Solution Path 84

Abstract

In this dissertation we propose a new technique for efficient query processing in knowledge base systems. Query processing in knowledge base systems poses strong computational challenges because of the presence of combinatorial explosion. This arises because at any point during query processing there may be too many subqueries available for further exploration. Overcoming this difficulty requires effective mechanisms for choosing from among these subqueries good subqueries for further processing.

Inspired by existing works on stochastic logic programs, compositional modeling and probabilistic heuristic estimates we create a new, nondeterministic method to accomplish the task of subquery selection for query processing. Specifically, we use probabilistic heuristic estimates to make the necessary decisions. This approach combines subquery and knowledge base properties and previous query processing experience with conditional probability theory to derive a probability of success for each subquery. The probabilities of success are used to select the next subquery for further processing. The underlying, property-specific probabilities of success are learned via a machine learning process involving a set of training sample queries.

In this dissertation we present our new methodology and the algorithms used to accomplish both the training and query processing phases of the system. We also present a method for determining the minimum training set size needed to achieve probability estimates with any desired limit on the maximum size of the errors.

Chapter 1

Introduction

1.1 Background

Traditionally, knowledge base systems (KBSs) make use of only one reasoning mechanism applied to a single, monolithic knowledge base (KB). One example of this is an "expert system". In an expert system the KB is a collection of facts and reasoning involves the application of implications to either the KB (forward chaining) or the goal (backward chaining) until a connection between the two can be made. More recently variations on this theme have been used. They include:

1. The use of multiple reasoning mechanisms in a single KBS.

   An expert system, for example, might use both forward and backward chaining together to answer a query.

2. The use of multiple KBs in a single KBS.

   (a) This most often occurs in what are called "hybrid knowledge base systems". This type of system contains multiple, heterogeneous KBs (KBs that store different information using different formalisms) and a reasoning mechanism that makes use of all of them. Typically the output from one stage of the reasoning process is designed to be used as input to the next. Output from the final stage will be the answer to the user's query.

   (b) The use of multiple, loosely coupled, heterogeneous KBs working in concert (their activities directed by a central controller) on the same query. In such a system, each KB works independently. The defining characteristic of such a system is that there exists some mechanism for sharing, among the KBs, the intermediate results of query processing. This allows the processing of queries such that no single KB

has the necessary information (or a powerful enough reasoning mechanism) to process the query.

Systems such as those described in 2b are desirable because:

1. Such systems allow each KB to make optimal use of its reasoning mechanism in query processing, and
2. The loosely coupled nature of the systems make them more easily scalable (i.e. ease the addition of new KBs), and
3. They allow users to store information in the most natural format given the nature of the data, and still have it available for use in query processing in combination with other data stored in different formats. This removes the problem of having to force all of the data that is necessary for use in the knowledge base system into one, uniform format.

Unfortunately, query evaluation in this kind of environment is extremely expensive due to the presence of combinatorial explosion. In this dissertation we present new techniques for dealing with this problem.

Chapter 2

The Problem

The problem that we address in this dissertation is that of how to overcome the difficulties caused by the presence of combinatorial explosion in the processing of queries in complex KBSs. In such systems, initially, a user's query will be passed on to the component KBs for processing, each of which may return any number of partially answered subqueries as its result. Thereafter, there may be many partially processed active subqueries to choose from when making the decision as to what to process next. This quickly leads to a combinatoric explosion of potential paths that the system can take on its way to answering the query.

2.1 A Sample Query

Let us assume a three part system composed of:

1. A rule base with rules:
    - Ancestor(X,Y) ← Parent(X,Z), Ancestor(Z,Y)
    - Ancestor(X,Y) ← GrandParent(X,Z), Ancestor(Z,Y)
    - GrandParent(X,Y) ← Parent(X,Z), Parent(Z,Y)
2. An inheritance hierarchy with contents as follows:
    - Ancestor(X,Y)
        (a) GrandParent(X,Y)
        (b) Parent(X,Y)
            i. Father(X,Y)
            ii. Mother(X,Y)

3. A database containing the following tuples:
   - Father(b,a)
   - Father(c,b)
   - Mother(d,a)
   - Mother(e,d)

To process a query we:

1. Input the initial query (see figure 2.1).
2. Treat it as the root of a search tree (see figure 2.2).
3. Generate new subqueries/nodes by processing the initial query (see figure 2.3).
4. Select active subqueries/nodes in the tree for further expansion, sending them to the KBs for further processing (see figure 2.4).
5. Finish when one of the newly generated subqueries answers the initial query (see figure 2.5).

Note that processing a subquery consists of selecting one predicate instance from the subquery to work on and replacing it with new knowledge, as dictated by the contents and operation of the KBs.

2.2 Combinatorial Explosion

When answering a query the problem space to be explored by the system is the tree that would exist if, for each node N in the tree (starting with the initial query as the root), every possible child node of N that could be generated was also in the tree, as a child of N. Unfortunately, because of the large branching factor that most real-world queries and KBs are likely to exhibit, this problem space is likely to be huge, and an exhaustive exploration of it, in search of a solution, impractical.

The kind of path through the problem space that we would like to find is a path which leads from the root to any solution in the fewest possible number of hops (and thus a path whose traversal involves as little work as possible on the part of the query processing system). We will call such a path a "Shortest Successful Path" (SSP) and note that it is not necessarily unique (see figure 2.6). Ideally, a tractable mechanical method for finding such a path would be available. In practice, it is not.

$$X : Parent(X,Y) \wedge Ancestor(Y,a)$$

Figure 2.1: The Initial Query

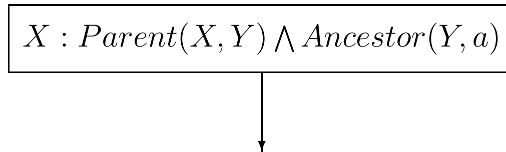

Figure 2.2: The Initial Query As Root

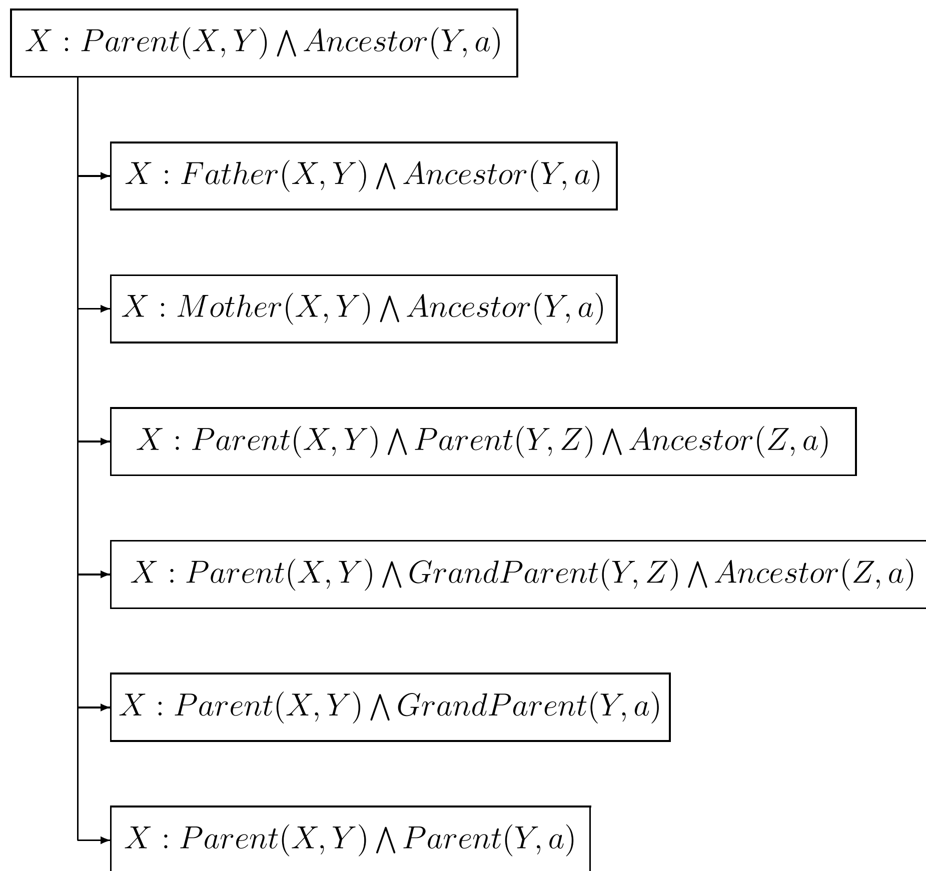

Figure 2.3: Children of the Initial Query

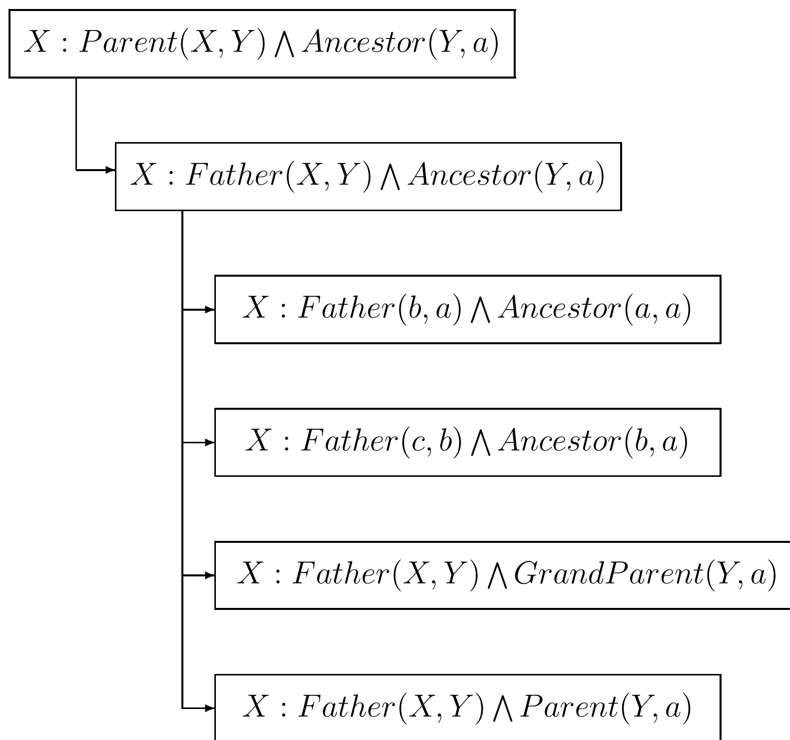

Figure 2.4: Processing the Initial Query

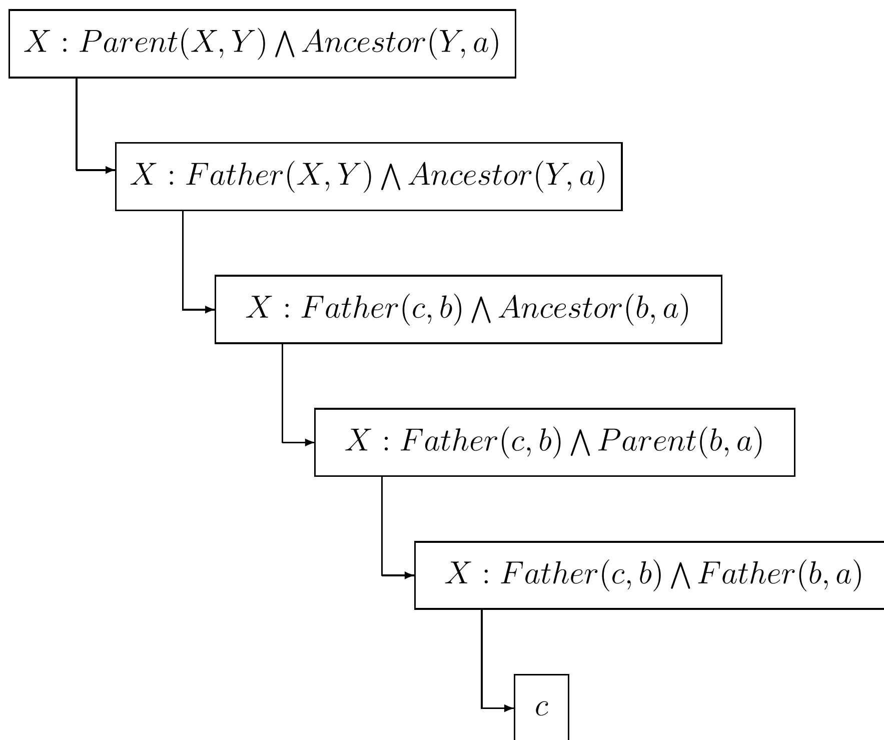

Figure 2.5: An Answer for the Initial Query

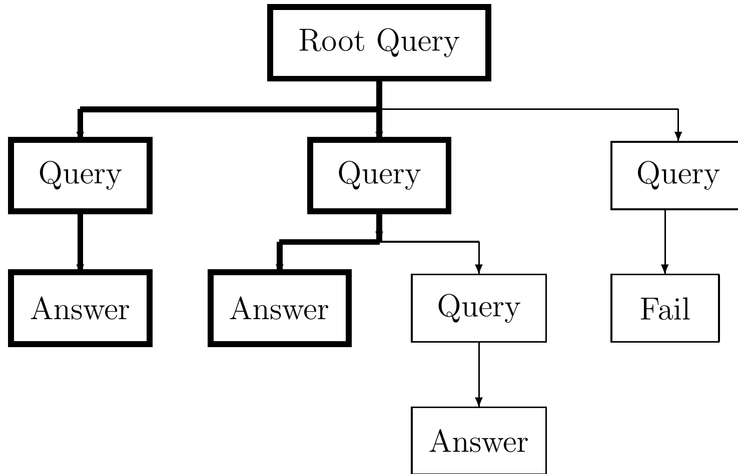

Figure 2.6: Shortest Successful Paths

2.3 Motivation

The problem addressed in this dissertation came to our attention through our reading of [Clar 97]. In this paper the authors present a query answering system built largely around the idea of *compositional modeling*. Compositional modeling involves starting with some small piece of knowledge, selected in reference to the user's query, then adding to it other pieces of knowledge, until the resulting model contains all of the information needed to answer the user's query (for detailed examples of this kind of system see [Falk 91, Rick 97]).

In the system of [Clar 97] the knowledge base initially consists of a large number of relatively simple bits of knowledge, stored as graphs. Query processing begins by taking a single piece of knowledge (as directed by the query) and proceeds via a classify/elaborate cycle, with each step in the cycle directing the system in its addition of a new piece of knowledge to the component already built. Adding a new piece of knowledge involves merging a new graph into the already existing one. The completed graph yields the desired answer, and the finished, composite graph may then be added to the knowledge base as a new atomic component for use in the processing of future queries.

Some of the questions we asked ourselves were:

- In a general purpose (not domain specific) query answering system, will manufactured components be reused often enough to make their construction worthwhile, and how can query processing most profitably proceed if this portion of the effort is left out?
- In this and other KBSs (e.g. [Ali 93]) it is usually assumed that all of the information necessary to answer queries can be reasonably encoded using the single knowledge representation formalism that is native to the authors' system. In the real world

this is not necessarily the case. Can a system be created that can answer queries in a reasonable amount of time if information is stored using more than one representational formalism? Hybrid KBSs that do this, such as [Gold 96] and [Sun 95], have shown good results.

- In the real world there is no reason to believe that any one KB will contain all of the knowledge or reasoning skills necessary to answer a user's query, particularly if the query is complex and involves concepts that span more than one subject domain. In practice the creation of a large, general-purpose, know-everything KB (the quintessential example being CYC [Lena 90, Lena 95]) seems to be much more difficult than creating a number of smaller KBs, each one targeted to handle a particular type of query and/or domain. Can a system be created that successfully and efficiently combines the efforts of many different, special purpose KBs in the answering of a single query?

2.4 Some Issues

Readers may recognize the type of system being considered here as being distantly related to the "blackboard architecture" systems studied during the mid to late 1980's (for the interested a useful starting point might be [Jaga 89]). We did not wish to make use of a blackboard architecture, but rather to have centralized control of query processing so as to avoid the excessively inefficient use of resources that can otherwise ensue.

Such a system as we wanted to build operates in a problem space such that each node in the space represents a query and each arc represents a transition between queries, which transition represents the application of one of many possible reasoning mechanisms upon a node. Answering a query involves starting with that query, modeled as the root of a tree, and traveling through the tree of possible derivative queries until an answer is found. In [Clar 97] a user's query must be entered in the form of an "access path" (see [Craw 91]), which path directs the system in its search of the problem space. Our approach is less amenable to such mechanisms and avoids constraining user queries in this fashion. The cost of this is to lose the user-supplied information about how to search the problem space. Lacking this information we had to develop another method by which the system could determine how best to search the problem space. Some of the issues that come into play when determining how best to carry out such a search were:

- We did not care whether or not the solution that we found utilized the shortest possible path from the root to an answer. Once we have found an answer there is no benefit to be gained in going back and finding another that may involve a shorter path through the search space. For this reason search algorithms like A* (first introduced in [Hart 68] and discussed at some length, along with several of its variations, in [Pear 84]) are of

no value, and the backtracking typically involved in their use would represent a waste of system resources.

- In a real world system of this nature there will likely be far too much information that might be related to any given query to make data-driven (forward-chaining) query processing practical. This is why goal-driven (backward-chaining) query processing is usually used (both of these methods are discussed in numerous expert system and AI texts, such as [Russ 95]). This is unfortunate, as successful human experts show a marked tendency to use a forward-chaining approach when solving problems, in contrast to both novice and less-successful expert problem solvers who tend to use a backward-chaining approach [Lark 80, Pate 86].

- Similarly, because the system can have no idea (initially) of what the solution to a given query will be, bidirectional searches through the problem space (first introduced to heuristic search in [Pohl 71] with many variations discussed in [Nels 92]) are of no value.

- Because there is little upon which to base guesses as to the nature of derivative queries likely to appear during query processing, search techniques based on knowledge of such intermediate nodes (see [Chak 86, Nels 90]), often called ”islands” in the context of search algorithms, are useless.

- In order to allow for the processing of queries in tractable amounts of time our system must avoid generating any derivative query that is unlikely to lead to a solution in as short a time as is possible. For this reason use of a ”look-ahead” based search algorithm (such as the classical Minimax algorithm first introduced in [Shan 50] and discussed at length, along with many of its variations, in numerous books on computer chess) is inappropriate. The use of such an algorithm would require that we generate, for each candidate query, many of its derivative queries in order to find out what kind of outcome we should expect from such derivative query generation. Unfortunately, in doing this we would have already spent the time that we wish to save in determining whether or not the candidate query is a good risk for having its derivative queries generated.

- Such a search space is far too large to be explored exhaustively. Also the number of potential search topics and the amount, and degree of diversity of types, of information that the system has about those topics is likely to be huge. For these reasons the system is unlikely to have any a-priori, content-specific information that can be used to determine conclusively which portions of the search space will be worth traversing for any specific query. This requires that the search be heuristic in nature.

These considerations left us needing to perform a purely goal-directed, heuristic search for any path leading to any solution. The heuristics used in the search could depend only on information already gained from that portion of the search carried out so far.

At this point it becomes important to be aware of the way that we are going to be using the following two terms:

- *heuristic properties*: the properties of a state that are used in its heuristic analysis (e.g. the number of pawns and knights of a certain color remaining on the board), and
- *heuristics*: which include both the heuristic properties and the mathematics used to generate a numerical value from these properties (e.g. the score for a chess position might be computed as the number of pawns remaining on the board divided by the number of knights, raised to some power).

Unfortunately, despite common perceptions on the subject, humans are not all that good at dealing with heuristics. Indeed, there are many types of common problems that humans (both experts and non-experts) rely on heuristics to solve and for which the heuristics that humans tend to use are chronically misleading. Using them humans presented with sample problems tend to assign, to events, probabilities that are significantly different from the correct probabilities as worked out by probability theory [Kahn 73, Tver 74]. The situation can only get worse when multiple heuristics must be applied, in combination, to the same problem. Additionally, when writing a heuristic evaluation algorithm many programmers tend to tightly integrate the heuristic properties being used into the code, so that if a property turns out to be useless a large block of code must be rewritten.

A better approach would be to bypass the human and let the system generate heuristics on its own. Heuristics that may make use of multiple heuristic properties simultaneously and that isolate them from the algorithm that uses them, so that if any heuristic property is found to be useless it can be replaced without having to rewrite a significant amount of code. Fortunately, the use of conditional probability theory in KBSs has already been extensively examined (for the classical reference see [Pear 88]). Using it we were able to develop an approach such that, when presupplied with a collection of useful heuristic properties, the system can generate a set of heuristics that satisfy these requirements.

2.5 Our Approach

Unfortunately there is no way to mechanically generate relevant heuristic properties from scratch. Fortunately, given such properties and a set of training examples, it is possible to have the system compute the numerical relationship between the heuristic properties and the state that the system is trying to achieve (e.g. checkmate). Moreover, this can be done in such a way as to:

- isolate the heuristic properties from the search code, so as to make the replacement of useless properties relatively inexpensive, and

- allow for the use of a large number of heuristic properties simultaneously, as opposed to using just one, and
- allow the system to determine (and report) the extent to which individual heuristic properties are contributing to the task of achieving the desired end state, thus making it easy for a system maintainer to identify and replace useless heuristic properties. This is particularly important given that we cannot have the system generate good heuristic properties from scratch, but must instead depend on the system maintainer for this. This aspect will not be instantiated in code as part of this dissertation, but the methods used (manually) during heuristic selection will be discussed, as an extension to the dissertation that may be carried out at a later date.

Specifically, if there exists, in the search space, a node v which the system is thinking of exploring, then we need to know how likely exploration of v is to lead to a solution as quickly as possible. Lacking a conclusive way to determine this from information available at the moment the system can only make an "educated guess" by apply a heuristic property (h) to v. This application yields a value x which can be used to make a judgment about the fitness of v as a candidate for exploration. Mathematically:

$$h(v) = x$$

Unfortunately, because of the uncertainty inherent in the use of heuristics, we have no way of knowing with certainty the relationship between x and the node's fitness. Specifically, what we need to know is: "given a specific value for x, what is the chance that exploration of v will yield the desired outcome?" Mathematically, we desire to know the value of y when:

$$y = P(success|h(v) = x)$$

and "success" refers to the desired outcome. Of course we can use more than one heuristic property at a time, seeking the value of y when:

$$y = P(success|h_1(v) = x_1, h_2(v) = x_2, ..., h_n(v) = x_n)$$

It should be noted that the heuristic properties that we use in this dissertation are designed to point us *towards* a *successful* path as opposed to pointing us *away from* an *unsuccessful* path (the same properties that are useful in spotting good paths may not be as useful in spotting bad ones). While this second option will not be explored in this dissertation, it is conceivable that a significant improvement in system performance could be generated by the use of additional "negative" heuristic properties to aid in the avoidance of bad decision making as we move through the search space. Specifically, we could add something like:

$$y = P(failure|g_1(v) = x_1, g_2(v) = x_2, ..., g_n(v) = x_n)$$

to the system and have it use both probabilities in determining how to proceed through the problem space. This kind of "bad move avoidance" approach has been used in classical heuristic search in [Kwa 89] where it led to significant improvements in performance.

Our algorithmic approach to searching the problem space is neither purely exploratory nor purely exploitative in nature. Alternative "next steps" are assigned a probability of success and the decision to take or not take the step is at least partially determined (depending on which algorithm is used) by comparison of this probability against a random number. This allows multiple paths through the search space to be explored simultaneously and means that while alternatives with very high probabilities of success are very likely to be taken it is not certain that they will. Likewise alternatives with low probabilities of success might be taken in spite of their low probabilities. This allows the system to learn more and more about the problem space as more and more user queries are processed, continually improving performance. A very similar approach to managing exploration vs. exploitation was used in [Bres 96] and a not-so-similar (non-probabilistic) approach in [Smir 96], both to significant success.

2.6 Related Work

This approach has much in common with that used with *stochastic logic programs* [Mugg 01, Mugg 00]. A stochastic logic program is like a standard logic program except that each clause is tagged with a probability value, and when it comes time to select a specific clause for use a clause is selected probabilistically from among all of the possible alternatives.

In order to be useful in a real-life environment our approach must provide some useful measure of how trustworthy the probabilities that result are. In the context of our software system our approach must tell us how many "training samples" we train the system with to ensure that a useful minimum level of trustworthiness is achieved. For this it turns out that a combination of:

- a fairly straightforward application of basic statistics (which can be found in [Freu 97] and is available in most basic statistics texts), which tells us how many samples we will need to use to be able to expect that our probabilities will be correct to within a certain predefined margin of error, and
- an extension of a form of mathematical analysis applied to the simple Bayesian classifier in [Domi 97] which gives us the probability that our system will give us the right answer given a certain difference between our derived probability and an ideal one

is sufficient to develop the theory that we need.

The probabilistic portion of this approach turned out to be closely related to a small existing body of work regarding "probabilistic heuristic estimates" (PHEs). In [Hans 89, Hans 90a]

the authors introduce the concept of PHEs and a method for their use in heuristic search, which method they implement in their system "Bayesian Problem Solver" (BPS).

Assume that we are given a partially explored search space that looks like that of figure 2.7.

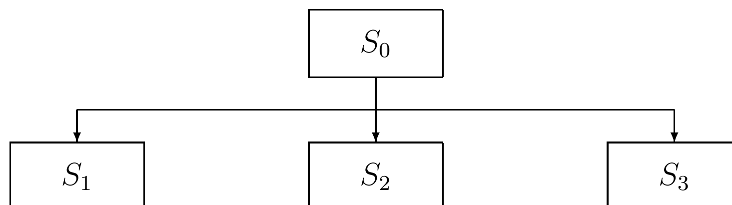

Figure 2.7: A BPS Search Space

S_0 is the root node. The system must determine if it should "commit" to a move to node S_2 (just as in a game of chess a player might consider the advantages of many moves and then commit to one of them). Using classical PHEs the system would use the results of a probability statement that looks like:

$$P(O_2|h(S_0), ..., h(S_n))$$

where:

- O_2 is a vector representing the possible outcomes of committing to S_2 (the function would have to be solved separately for each possible outcome), and
- $h(S_0), ..., h(S_n)$ is the set of heuristic values for all relevant nodes of the tree (and it is a non-trivial problem to determine which nodes are relevant in this context) for which such values have already been computed.

along with the results obtained by solving the same function for O_1 and O_3, to determine which node of S_1, S_2 and S_3 should be committed to. As part of this process BPS would expand (in look-ahead fashion) the nodes under consideration as well as several nodes chosen from among their descendents, whose heuristic values would also be used in the function above. For example, in its attempt to determine the advantages of committing to node S_2, BPS might choose to expand and evaluate the search space far enough to allow it to include the heuristic values for nodes S_{21} and S_{22} in its analysis (see figure 2.8).

If, for any reason, there is still insufficient reason to commit to one node over another, the results could be extended by generating more nodes in the tree and recomputing the probabilities. For example, if we desired more accuracy than the function above could give us, where $h(S_0), ..., h(S_n)$ ranges over all 6 nodes in the tree above, the system could generate the children of S_{21} (say S_{211} and S_{212}) and then recompute the needed probabilities with

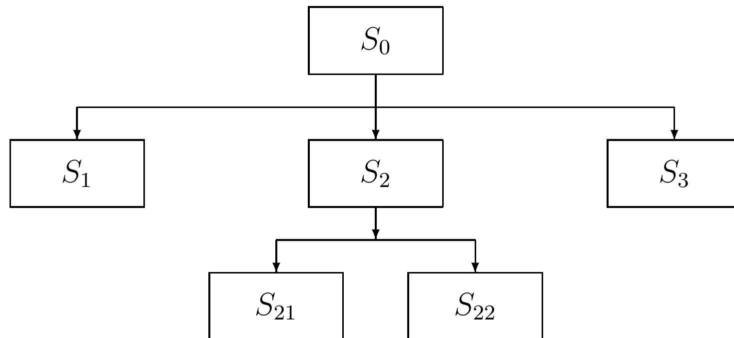

Figure 2.8: An Expanded BPS Search Space

$h(S_0), ..., h(S_n)$ ranging over all 8 heuristic values in the re-expanded tree (see figure 2.9). Of course the system monitors itself to prevent the expansion from going on indefinitely if no progress is being made.

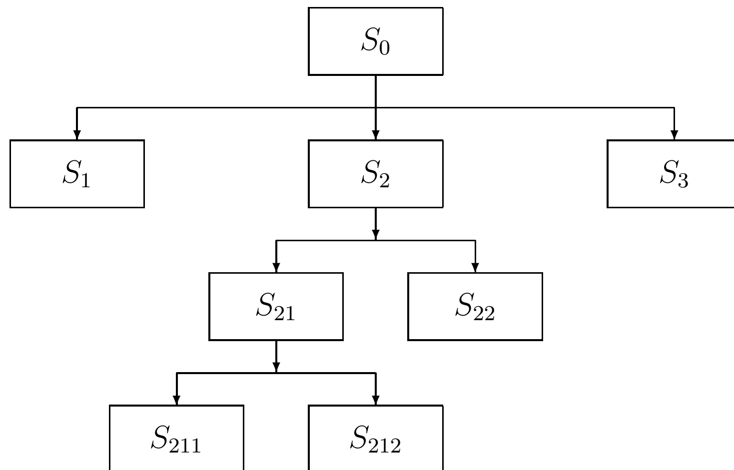

Figure 2.9: A Re-Expanded BPS Search Space

Later the authors propose an extension to BPS (proposed in [Hans 90b] and discussed further in [Hank 90]) that would allow it to design plans by making use of abstraction and specialization of states in the problem space. In [Maye 94] Mayer discusses the underlying theory of BPS in some detail, and applies it to the problem of solving the classical "eight puzzle". In this application puzzle states are associated with nodes of the graph and movements of the tiles with the arcs separating them. It is shown that BPS solves eight puzzles in far fewer moves, and with the look-ahead examination of far fewer nodes, than other classic search algorithms. In [Hans 98] Hansson applies BPS (also with significant success)

to the SAT problem, using it to determine which truth value should be assigned to a variable pre-selected (by some other mechanism than BPS) from a proposition.

Unlike this use of PHEs our system:

- Cannot, for the reason mentioned above, make use of look-ahead of any sort. In fact, in [Maye 94] the author goes into some detail about the need for BPS to analyze at least some descendent nodes before making a commitment, and about the various means for deciding which and how many such nodes to analyze.
- Does not, when doing a heuristic analysis of one node in the graph, use the heuristic analyses of other nodes (even non-look-ahead nodes) as input. When attempting to determine if a node (say S_2) should be expanded, our system only analyzes the properties of node S_2. Given the size of the search space that we are dealing with and the nature of the task involved there is no tractable way to include heuristic values from all other nodes existing in the tree at that point. Also, there is no reason to expect that any particular sibling or cousin node(s) may be of significant importance, as individuals, to the resulting probability (the existence of such dependencies among nodes in the eight puzzle tree is discussed in [Maye 94]). At most, some information about the computation that has led directly to the generation of the node being analyzed, such as the number of steps from the root that the node falls, might be recorded within the node itself, and thus available for use cheaply as a heuristic property.
- Makes use of several heuristic properties per node, which possibility is mentioned, but not done, in any of the BPS works that we have reviewed until the most recent [Hans 98]. In this work 2 heuristic properties for each look-ahead node are used. Previous to this, heuristic properties were taken one each from each of the look-ahead nodes being used in analysis of the current node.

Like this use of PHEs our system:

- Does not use PHEs for selecting which child-generating operations to carry out on each node. The idea of using operator selection (even of basing the selection on the previous performance of the operator when applied to a node having similar properties) to increase search efficiency is not new, having been proposed at least as far back as 1967 (see [Mich67]).

The semantics of both our query language and the contents of our KBs are based on the semantics of first order logic [Lloy 87] and frames [Haye 81]. Because our KBs share this language as a built-in feature we are spared the usual difficulty of having to construct *translators* (also called "wrappers") to enable KBs to understand each other's output (for a discussion of translators see [Papa 95]). Of course there is no reason that a KB not built to our specifications could not participate, via the use of a translator.

All of the KBs within our system use a single, shared *ontology*. In simple terms this means that all of the KBs use the same words in the same way, to mean the same thing (see [Grub 93, van 97] for a detailed discussion of this and [Grub 96] for detailed excerpts from an existing ontology). Obviously if the word "dog" in one KB meant something different from the word "dog" in another KB then having the two KBs cooperate to answer a query involving dogs, without grave problems arising from this difference, would be difficult if not impossible. The issue of ontology sharing in this context is a real one that must be addressed, but it is beyond the scope of this dissertation.

Chapter 3

Our Approach: Probabilistic Heuristic Estimates

In order to accomplish our goals we will make use of a variation on probabilistic heuristic estimates to allow the system to make intelligent guesses (after a lengthy training process) concerning which subquery to process next. The system will be attempting to choose that subquery which has the best chance of being on a SSP. Our procedure will be to:

1. Randomly generate a set of KBs.
2. Train the system using a number of randomly generated sample queries.
3. Select, from a large list, a small collection of heuristics for use in query processing, based on their performance with our smart algorithms and a small number of sample queries.
4. Process a large number of test queries, using PHEs to determine at each point which subquery is likely to be the best one to process next.
5. Collect results for analysis.

A more detailed discussion follows.

3.1 The System Architecture

As part of this dissertation we have built a working KBS and used it to demonstrate the techniques described herein. The system is composed of 4 modules (see figure 3.1).

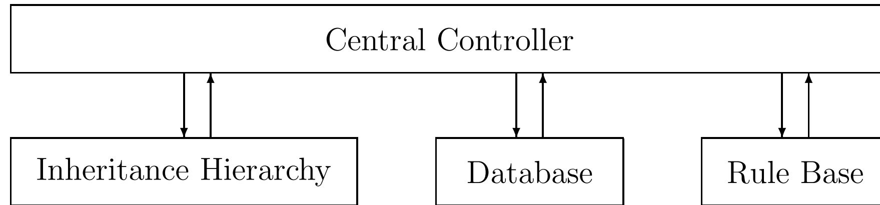

Figure 3.1: System Architecture

- The Central Controller: This module receives queries from the users and makes and implements all decisions regarding how they should be processed.
- The Inheritance Hierarchy: Given a predicate, in a subquery, to process, the Inheritance Hierarchy generates new subqueries by replacing the assigned predicate with its immediate descendents in the hierarchy, as appropriate. Each such substitution will generate a new subquery. For example, given a subquery that contains the predicate $Parent(X, Y)$ it might return a new subquery in which the original predicate has been replaced by one of its descendants in the hierarchy, $Father(X, Y)$.
- The Database: This knowledge base binds variables to ground items. Given a predicate, in a subquery, to process it generates new subqueries by replacing variables in the assigned predicate with ground items taken from matching tuples. For example, given a subquery that contains the predicate instance $Father(X, Joe)$ it might return a new subquery in which the original variables have been replaced by ground items such that the predicate instance $Father(John, Joe)$ results.
- The Rule Base: This knowledge base substitutes instances of one predicate for instances of other predicate(s), generating a new subquery in the process. For example, given a subquery that contains the predicate instance $Ancestor(X, Joe)$ it might return a new subquery in which the original predicate instance has been replaced by $Parent(X, Y) \bigwedge Ancestor(Y, Joe)$.

All message traffic travels between the central controller and the individual knowledge bases (again see figure 3.1). Each message consists of a subquery to be processed by the receiving knowledge base (if the message is headed away from the Central Controller) or a subquery created by a knowledge base, as the result of processing (if the message is headed toward the Central Controller). Individual knowledge bases do not talk to each other directly, nor do they talk indirectly by passing messages intended for each other through the Central Controller.

3.2 Operation of the Central Controller

The job of the Central Controller is to take queries from the users, make and implement all decisions regarding how they should be processed, and return the results of processing to the user. This takes place in the following steps(see figure 3.2).

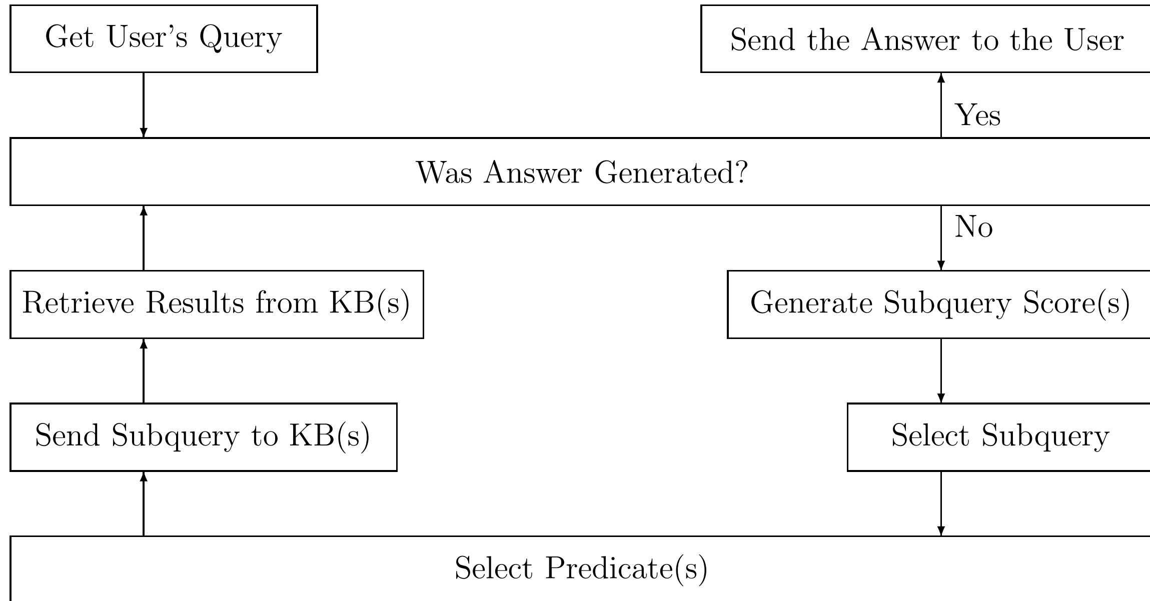

Figure 3.2: Central Controller Operation

1. Get a query from the user.

2. See if an answer has been generated, and if so return the answer to the user and quit.

3. Generate a numerical score for each subquery currently under consideration. These scores will be used later to determine for each subquery whether or not the subquery merits further processing in the attempt to answer the user's query. See chapter 4 for more about this process.

4. Select the subquery to be processed next. See chapter 4 for more about this process.

5. For the subquery selected for further processing, select a specific predicate instance appearing within the subquery as the specific predicate instance to be processed next. Remember that subquery processing by a knowledge base consists of substituting one predicate instance for another such that the new subquery generated will be closer to a solution. See section 4.8 for more about this process.

6. Send, to each of the knowledge bases, the subquery that has been selected by the system to be processed next.
7. Retrieve the new subqueries generated by the knowledge bases.
8. Return to step 2.

3.3 Operation of a Knowledge Base

A knowledge base is sent queries by the Central Controller, processes them, and returns the results of this processing to the Central Controller. Processing consists of replacing the pre-selected predicate instance occurring within the received query with another predicate instance, such that the new subquery thus created will be closer to the goal than the original was. This operation proceeds as follows:

1. Get a subquery from the Central Controller.
2. Select those rules that can be applied to the given subquery.
3. Perform all possible rule applications.
4. Return to the Central Controller all new subqueries generated.

3.4 Probabilistic Selection

Lacking a tractable, mechanical method for determining, for example, how good a choice a given subquery is for processing, we may still achieve our objective by *estimating* how good the subquery is, provided that our estimation is good enough. What makes this approach possible is that we do have a tractable method for making such estimates. This method involves the use of a variation on "probabilistic heuristic estimates" (PHEs). PHEs are derived via an application of the rules of conditional probability to the practice of heuristically estimating the value of a state in a state space.

When searching a state space like our problem space we wish to know what the probability of the desired outcome of our search would be if we "committed" to the state currently being evaluated. Using "success" as a generic term to refer to whatever that desired outcome is, a PHE of the probability of success associated with the state currently being evaluated would be generated by the following formula:

$$P(success|property_1, ..., property_n) \tag{3.1}$$

where each $property_x$ is some property of a state that is relevant to the determination of the probability of success if we commit to that state. A system making such use of PHEs must first be trained by being run on many sample searches, so that the probabilities associated with the various combinations of properties can be derived. Thereafter it can be applied to real-world searches to estimate the probability of success for a state, provided only that the properties chosen for use in the formula are sufficiently relevant and comprehensive (more than just a small percentage of the number of relevant properties).

Chapter 4

Solution: Choosing the Best Subquery to Expand Next

Given a set of subgoals for processing, a KBS can select any subgoal from the set to process next. The processing of a subgoal uses resources, so we would like to process only those subgoals whose processing will lead quickly to answers. We choose these subgoals via the application of a variation on PHEs.

4.1 Computing the Probabilities

What we need to know, in order to know which subgoal to process next, is the probability that a specific subgoal (g) with certain properties (p) will be on a SSP (see figure 4.1).

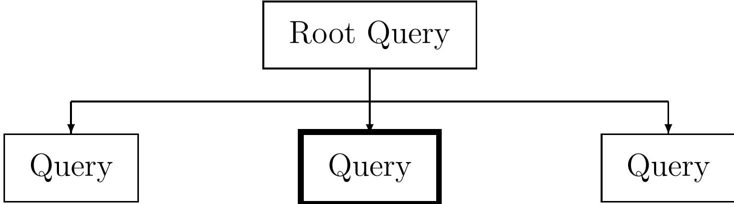

Figure 4.1: What are the odds that the highlighted node is on a SSP?

Ideally we could test every possible subgoal with properties p with every possible initial query and find out how often it is on a SSP. Then, for any given subgoal g with properties p, we could compute the probability that g would be a member of such a path to be:

$$\frac{|G_p^*|}{|G_p|}$$

where G_p is the set of all possible occurrences of all subgoals g with properties p in complete search trees for all possible initial queries and G_p^* is the subset of those such that g fell on a SSP. We can also express this in terms of a PHE as:

$$P(success|property_1, ..., property_n)$$

where success occurs when a subgoal is on a SSP and $property_1, ..., property_n$ are the properties p. Note that:

1. even if we could generate the set of all possible initial queries and subgoals to train the system with (which we cannot, of course, see chapter 6 for discussion of the minimum necessary size of the training set), and
2. even if we could generate a complete search tree when processing a query (which we cannot, of course, see chapter 6 for discussion of how this affects the training process), then

this formula would still present us with tractability problems. This occurs because the size of the cross product of the properties can grow too large. For example, if we use 5 properties and each one can take 10 different values then the number of different property-value combinations that will need to be worked with will be 10^5 or 100,000. In order to compensate for this we must work with the formula to get it into a form that does not have this problem.

In deriving the new form of this formula that we will use we assume that the properties are both independent of each other and conditionally independent given *success*. The derivation of the new formula then proceeds as follows:

$$\frac{P(property_1, ..., property_n|success)P(success)}{P(property_1, ..., property_n)} =$$

$$\frac{\prod_{i=1}^{n}[P(property_i|success)]P(success)}{\prod_{i=1}^{n}[P(property_i)]} =$$

$$\frac{\prod_{i=1}^{n}[\frac{P(success|property_i)P(property_i)}{P(success)}]P(success)}{\prod_{i=1}^{n}[P(property_i)]} =$$

$$\frac{\frac{\prod_{i=1}^{n}[P(success|property_i)]\prod_{i=1}^{n}[P(property_i)]}{P(success)^n}P(success)}{\prod_{i=1}^{n}[P(property_i)]} =$$

$$\frac{\prod_{i=1}^{n}[P(success|property_i)]\prod_{i=1}^{n}[P(property_i)]}{P(success)^{n-1}\prod_{i=1}^{n}[P(property_i)]} =$$

resulting in:

$$\frac{\prod_{i=1}^{n}[P(success|property_i)]}{P(success)^{n-1}} \tag{4.1}$$

Thus, when a specific subgoal must be selected for processing we can use this formula to generate, in a computationally much less expensive fashion, the probability of success for that subgoal.

Of course any single $property_i$ appearing in formula 4.1 above can, in fact, be a joint combination of a number of other properties. For instance, given a situation where n in formula 4.1 is equal to 2, it may be the case that for some other properties 3...7:

- $Property_1 = Property_3 \times Property_4$
- $Property_2 = Property_5 \times Property_6 \times Property_7$

4.2 Useful Properties for PHE-based Query Scoring

The heuristic properties of queries (and knowledge bases) that are available for use by our system are:

1. number of conjuncts
2. number of conjuncts that are predicate instances
3. number of conjuncts that are relation instances
4. number of non-negated conjuncts
5. number of negated conjuncts
6. number of arguments, totaled over all conjuncts that are predicate instances
7. average number of arguments appearing in conjuncts that are predicate instances
8. maximum number of arguments appearing in a conjunct that is a predicate instance
9. number of variables, totaled over all conjuncts
10. number of variables, totaled over all conjuncts that are predicate instances

11. number of variables, that appear in a conjunct that is a predicate instance, that also appear in another conjunct (in the same query) that is a predicate instance
12. number of variables, that appear in a conjunct that is a predicate instance, that do not appear in another conjunct (in the same query) that is a predicate instance
13. number of variables, totaled over all conjuncts that are relation instances
14. number of non-variables, totaled over all conjuncts that are predicate instances
15. average number of non-variables appearing in conjuncts that are predicate instances
16. maximum number of non-variables appearing in a conjunct that is a predicate instance
17. number of non-variables, totaled over all conjuncts that are relation instances
18. maximum number of non-variables appearing in a conjunct that is a relation instance
19. minimum number of DB tuples relating to a predicate appearing in the query
20. average number of DB tuples relating to a predicate appearing in the query
21. average number of hops between the root of the inheritance hierarchy and a predicate appearing in the query
22. maximum number of hops between the root of the inheritance hierarchy and a predicate appearing in the query
23. number of rules (total) relating to all predicates and relations appearing in the query
24. number of rules (total) relating to all predicates appearing in the query
25. average number of rules relating to a predicate appearing in the query
26. minimum number of rules relating to a predicate appearing in the query
27. number of rules (total) relating to all relations appearing in the query
28. average number of rules relating to a relation appearing in the query
29. minimum number of rules relating to a relation appearing in the query

Once the system has been trained, a subset of these heuristics will be selected for use in subquery selection. For example, a subquery with only 1 conjunct (property 1) remaining, and only 1 variable (property 9) left to instantiate, and regarding whose conjunct the database has thousands of tuples (property 19) in storage is probably a good risk for immediate expansion. Such an expansion effort has a good chance of turning up a matching tuple for the single remaining conjunct, and providing a ground item for assignment to the one remaining variable, thus solving the query immediately.

4.3 Choosing Which Subquery to Process (Best-First Search)

In deciding on whether or not to spend resources and actually send a subquery off for processing, the algorithm in figure 4.2 will be used when best-first selection of a subquery is desired.

QUERY: the subquery currently under evaluation
QUERYSET: the set of all subqueries that can be expanded, sorted by score

QUERY := the highest scoring item from QUERYSET
process QUERY;
remove QUERY from QUERYSET;
add to QUERYSET all new subqueries that resulted from the processing of QUERY;

Figure 4.2: Best-First Method for Choosing Which Subquery to Expand Next

4.4 How Much Confidence Should We Have in Our Selection (Best-First Selection)?

Because formula 4.1 was derived with the use of independence assumptions and because the size of our training set is limited, our use of this formula only approximates the results that we desire. In order to have confidence in our estimates we must have some idea of how good the estimates are likely to be.

The training method that we use (see chapter 6) will yield for each subgoal's "score" a value of the form $P \pm \varepsilon$, where P is the estimated probability (from 0 to 1) of success for the subgoal and ε is a combined error term equal to $\varepsilon_{sampleSize} + \varepsilon_{iAssumptions}$ where:

- $\varepsilon_{sampleSize}$ is equal to the maximum, worst case, absolute error that P will exhibit with a certainty of X% (the source of this error is the limited size of the training set). Given a value of 95 for X, and .01 for $\varepsilon_{sampleSize}$ we would say, if were to put it in plain English, that "we are 95% certain that the maximum, worst case, absolute error that P will exhibit, as a result of the finite size of the training set, is .01". For a discussion of the kinds of values that we expect to be able to achieve for $\varepsilon_{sampleSize}$ see chapter 6.
- $\varepsilon_{iAssumptions}$ comes from our use of the independence assumptions in deriving formula 4.1. If the properties that we choose to use in applying this formula are not completely independent of each other then the lack of independence will manifest as an additional source of error in P. While there is no direct way to quantify this source of error we

believe that we can choose properties that will prevent the value of $\varepsilon_{iAssumptions}$ from rising beyond a workable minimum.

4.4.1 A Correct Selection

A correct selection is a selection which is the best that can possibly be done with the information at hand. For example, let us imagine two subqueries, S1 and S2. S1 has the highest estimated probability of success ($P_{e,s1}$), while S2 has the second ($P_{e,s2}$). Choosing S1, we may be certain that we are choosing a subquery whose ideal probability of success (P_i, the probability the program would assign if it were omniscient) has a probability of X^2 of being within 2ε of the of the highest P_i assigned to any subquery in the list of subqueries under consideration. This is because the chance of each P_i being within a distance of ε of its associated P_e is X, and so the chance that both of them will fall within this range is X^2. Given that both of them fall within this range, the greatest discrepancy that can occur when we pick S1, because it is the subquery with the highest P_e, occurs when:

1. $P_{e,s2}$ is as close as possible to $P_{e,s1}$, and
2. $P_{i,s2}$ is as much higher as possible than $P_{e,s2}$, and
3. $P_{i,s1}$ is as much lower as possible than $P_{e,s1}$

See figure 4.3. In effect, even though S1 has the higher P_e, S2 has the higher P_i, by as much as possible.

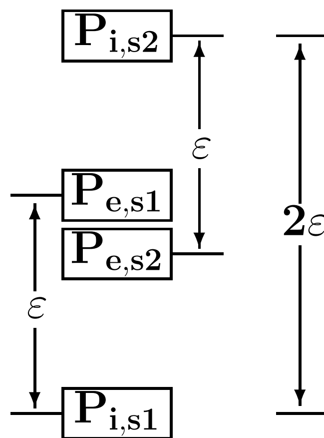

Figure 4.3: Correct Selection

When this occurs, the greatest possible distance between $P_{i,s1}$ and $P_{i,s2}$ is 2ε. Since there is no way to tell, in an an actual example, if $P_{i,s1} > P_{i,s2}$ or vice-versa, the best we can say is

that we may be certain that we are choosing a subquery whose P_i has a probability of X^2 of being within 2ε of the of the highest P_i assigned to any subquery in the list of subqueries under consideration. Given our lack of knowledge of the actual values of the various P_i's this is the best that can be done under ideal circumstances, and a selection of which this can be said will be called a correct selection.

4.4.2 Best-First Selection

When subgoal processing proceeds in best first fashion all subqueries will be available for selection, and we can be certain that the one with the highest P_e will be selected. Thus we can be 100% certain that the subquery chosen will be a correct selection.

4.5 Choosing Which Subqueries to Process (Probabilistically)

In deciding on whether or not to spend resources and actually send a query off for processing, the "Hybrid" algorithm of figures 4.4 and 4.5 will be used when probabilistic selection of queries is desired.

4.6 How Much Confidence Should We Have in Our Choices (Probabilistic Selection)?

If formula 4.1 were derived without use of the independence assumption and trained using a training set containing all possible initial queries we could be sure that its use would generate correct results. Because this is not the case our use of the formula will only approximate the results that we desire. In order to have confidence in our estimates we must have some idea of how good our results are likely to be when the estimates are used in this fashion.

4.6.1 The Selection of Subqueries for Inclusion in TEMPSET

First, we will examine how likely it is that the error in our estimate will cause a subquery to be placed into TEMPSET when it should not be, or not to be placed into TEMPSET when it should be. Given the 3 variables below:

QUERY: the subquery currently under evaluation
QUERYSET: the set of all subqueries that can be expanded
TEMPSET: temporary holding space for a collection of subqueries
P: the probability of success for a subquery
RANDOM: A random, floating point number between 0 and 1 inclusive

```
while (QUERYSET is not empty and answer is not found)
    for (each member QUERY of QUERYSET)
        RANDOM := new random number;
        P := the probability that expanding QUERY will lead to success (see formula 4.1)
        if (RANDOM <= P)
            copy QUERY into TEMPSET;
        end if;
    end for;
    if (TEMPSET is not empty)
        process TEMPSET query with highest P;
        add the results of the query processing to QUERYSET;
        remove processed query from QUERYSET;
        empty TEMPSET;
    end if;
end while;
```

Figure 4.4: Probabilistic Method for Choosing Which Subqueries to Process Next

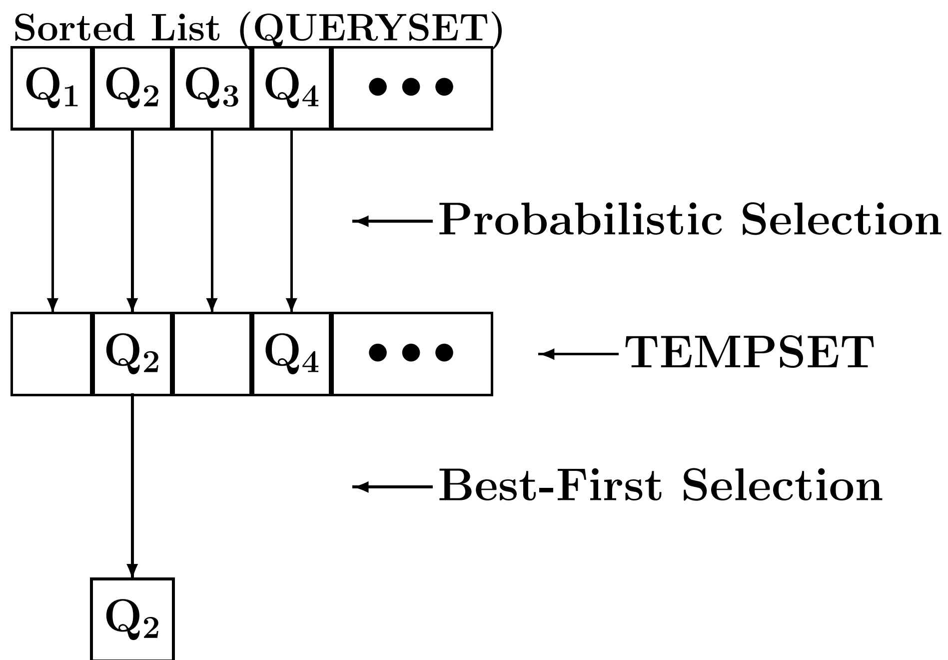

Figure 4.5: The Hybrid Algorithm

- P_i: the ideal (correct) probability of success for a subquery
- P_e: our estimated probability of success for a subquery
- RANDOM: the value of RANDOM used by the algorithm above to determine if a subquery should be selected

We may analyze the situation as follows:

The Worst Case:

Assuming that there is no useful relationship between P_e and P_i our algorithm will still correctly place a subquery into TEMPSET (or not) if the following condition is true:

$$(RANDOM < P_i \land RANDOM < P_e) \lor (RANDOM > P_i \land RANDOM > P_e) \quad (4.2)$$

Given that both P_i and P_e range between 0 and 1 inclusive, we can assign RANDOM the value of 0.5 and generate the graph of figure 4.6, wherein the shaded areas represent those values for P_e and P_i such that P_e and P_i are both simultaneously less than (green area) or both simultaneously greater than (red area) RANDOM.

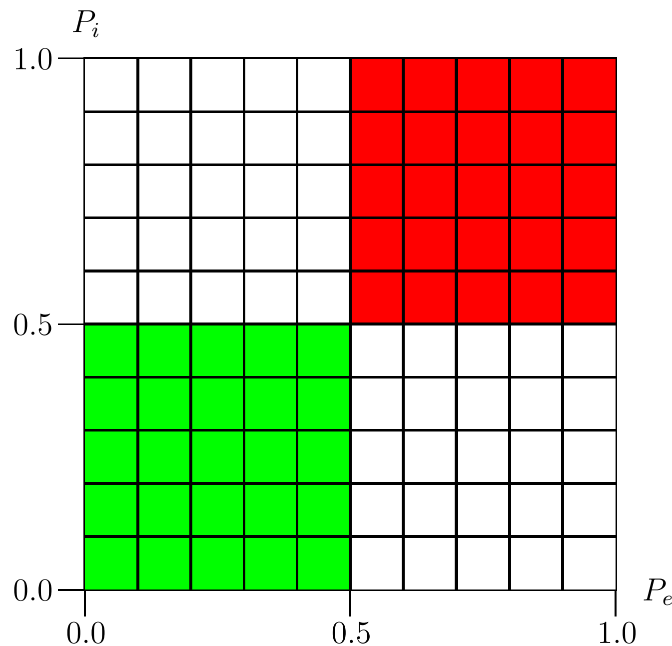

Figure 4.6: Usable Space for Probabilistic Selection When RANDOM = 0.5

Using figure 4.6 it can be seen that there is a 50% chance of our algorithm yielding the correct answer, even if P_e and P_i range between 0 and 1 completely independently of each other. Specifically, this is because for 50% of the possible values of P_e and P_i (the shaded areas) the values will both be either simultaneously less than RANDOM or simultaneously greater than RANDOM, so that our algorithm will yield the same answer for P_e as it would for P_i, regardless of the difference between them. In fact, the case for $RANDOM = 0.5$ is the worst case. Figures 4.7 through 4.11 show what the graph looks like for some other values of RANDOM.

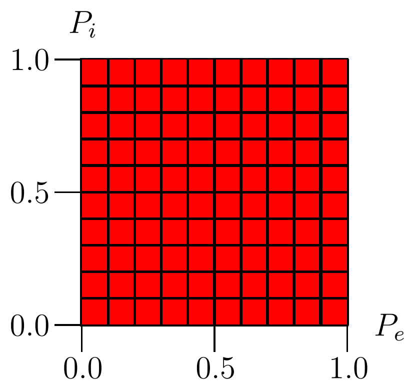

Figure 4.7: Usable Space When RANDOM = 0.0

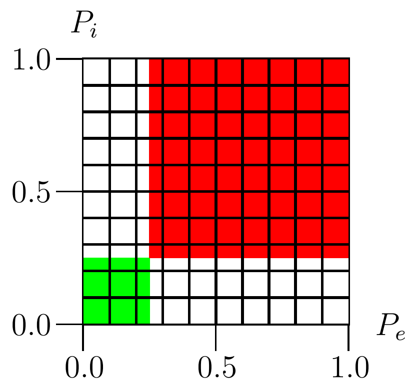

Figure 4.8: Usable Space When RANDOM = 0.25

From figures 4.7 through 4.11 we can see that, for any selected value of RANDOM, the shaded area of the graph is equal to $RANDOM^2 + (1 - RANDOM)^2$. RANDOM varies from 0 to 1, giving us a 3 dimensional space, whose total volume is 1, of all possible combinations of values for RANDOM, P_e and P_i. Assigning V to be that percentage of this space which

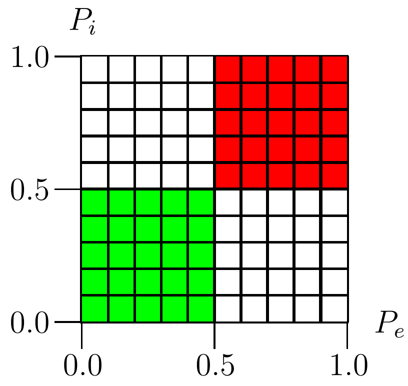

Figure 4.9: Usable Space When RANDOM = 0.5

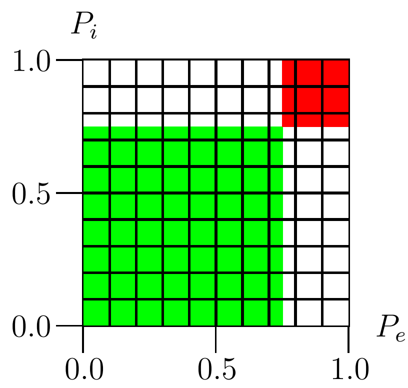

Figure 4.10: Usable Space When RANDOM = 0.75

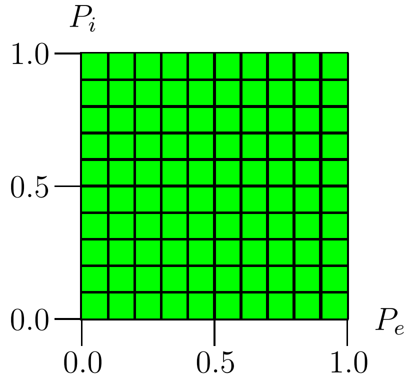

Figure 4.11: Usable Space When RANDOM = 1.0

is made up of points whose coordinates are values of RANDOM, P_e, and P_i for which our algorithm will give correct results, regardless of the difference between P_e and P_i, we get the equation:

$$V_{usableSpace} = \int_{RANDOM=0}^{1} [RANDOM^2 + (1 - RANDOM)^2] = 2/3 \tag{4.3}$$

This tells us that even in the worst possible case, that P_e takes its value completely independently of P_i, the probability is still 2/3 that our algorithm will handle the transition of a subquery into TEMPSET correctly. This is significantly better than the case for the naive 0-1 classifier, for which the same type of analysis gives only a 50% chance of yielding correct results when P_e varies independently of P_i, regardless of the fact that the naive 0-1 classifier has proven extremely useful for all sorts of tasks.

Not The Worst Case:

Of course we need not take such a pessimistic view. While our choice of properties for use in the heuristics might not be so perfect that complete independence between properties is achieved, it is likely that we can come fairly close. Also, we can use a training set of sufficient size to reduce the amount of error from that source to a manageable minimum. This being the case, it is likely that P_e will fall fairly close to P_i, and thus the percentage of usable space will be significantly higher than the 2/3 discussed above.

Let us, for example, suppose that the value of P_e falls within 0.2 of the value of P_i (we will call this distance ε). This causes it to be the case that the set of all possible values for P_e and P_i, when RANDOM = 0.5, is the set of values that falls between the two diagonal lines appearing in figure 4.12.

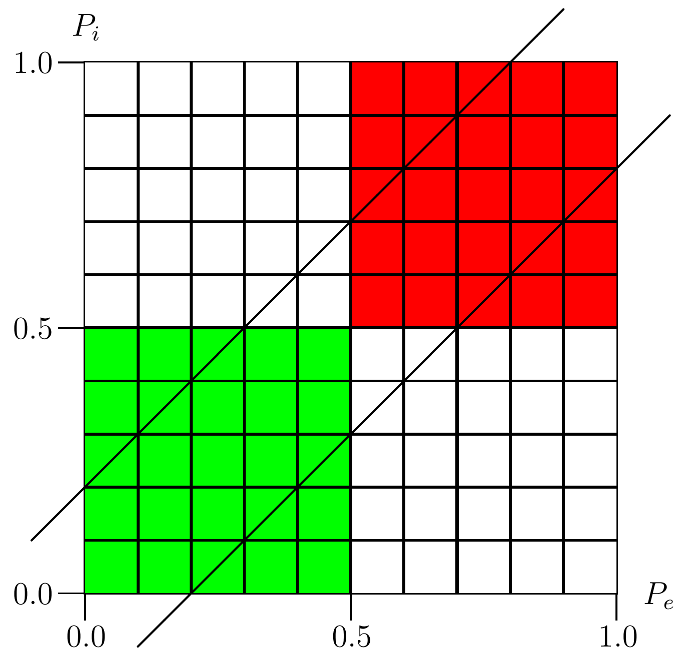

Figure 4.12: Usable Space for Non-Worst Case When RANDOM = 0.5

The volume of this subspace is significantly less than the volume of the entire space (which is 1), and most of the subspace (far more than 2/3 of it) is shaded (usable). Looking at figures 4.13 through 4.17 we can see that this remains the case as the value of RANDOM varies between 0 and 1.

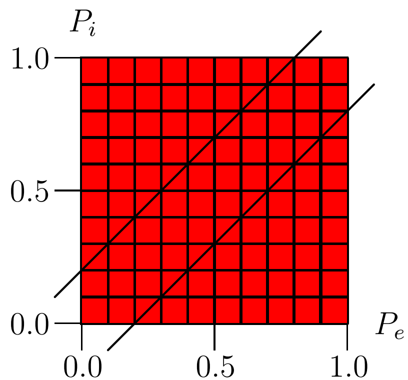

Figure 4.13: Non-Worst Case Usable Space When RANDOM = 0.0

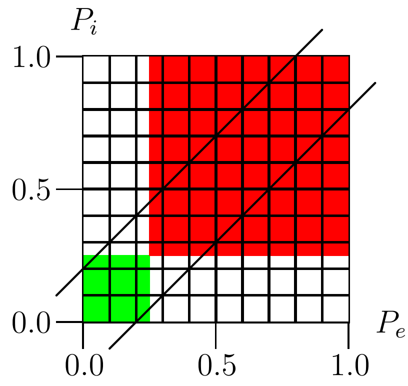

Figure 4.14: Non-Worst Case Usable Space When RANDOM = 0.25

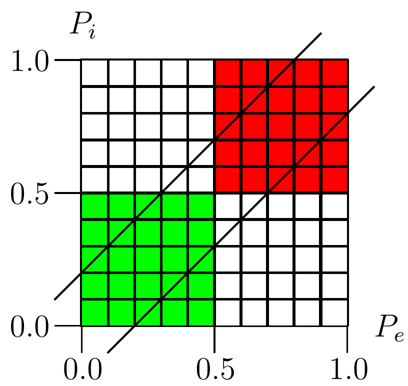

Figure 4.15: Non-Worst Case Usable Space When RANDOM = 0.5

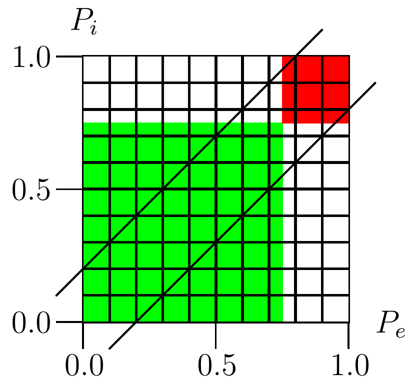

Figure 4.16: Non-Worst Case Usable Space When RANDOM = 0.75

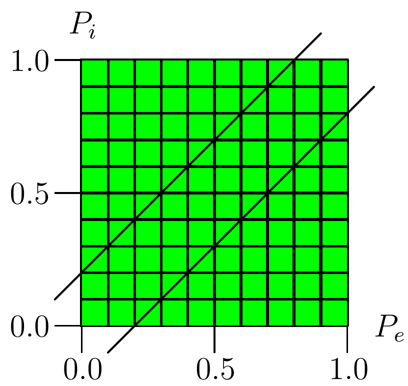

Figure 4.17: Non-Worst Case Usable Space When RANDOM = 1.0

The entire volume of the subspace contained between the two diagonals, derived as a function of ε, is given by the equation:

$$V_{subspace} = \int_{RANDOM=0}^{1} [1 - (1-\varepsilon)^2] = \varepsilon(2-\varepsilon)$$

The shaded volume of the subspace between the two diagonals, as a function of ε, is given by the following equation, where the first two integrals refer to the shading contributed by the square in the lower left hand corner of the diagrams and the second two by the square in the upper right:

$$\begin{aligned}
&V_{usable} = \\
&\quad \int_{RANDOM=0}^{\varepsilon} [RANDOM^2] + \\
&\quad \int_{RANDOM=\varepsilon}^{1} [RANDOM^2 - (RANDOM - \varepsilon)^2] + \\
&\quad \int_{RANDOM=0}^{1-\varepsilon} [(1 - RANDOM)^2 - ((1 - RANDOM) - \varepsilon)^2] + \\
&\quad \int_{RANDOM=1-\varepsilon}^{1} [(1 - RANDOM)^2] = \\
&\quad 2\varepsilon(1 - \varepsilon + \frac{\varepsilon^2}{3})
\end{aligned}$$

Dividing the second result by the first, we get an equation that gives us, as a function of ε, the amount of usable subspace, as a percentage of the whole:

$$V_{usableSubspace} = \frac{6 - 6\varepsilon + 2\varepsilon^2}{3(2-\varepsilon)} \tag{4.4}$$

A graph of which relationship appears in figure 4.18.

From figure 4.18 it can be seen that even if a significant degree of error is present it is still quite likely that probabilistic selection will correctly handle the decision about wether or not to copy a subquery into TEMPSET, regardless.

4.6.2 The Selection of a Subquery from TEMPSET for Processing

The previous section provides a figure that tells us how likely it is that an individual subquery will be *correctly* included (or not) in TEMPSET. From this we must derive a number telling us how likely it is that the subquery selected from TEMPSET, in the next step, will be the correct one.

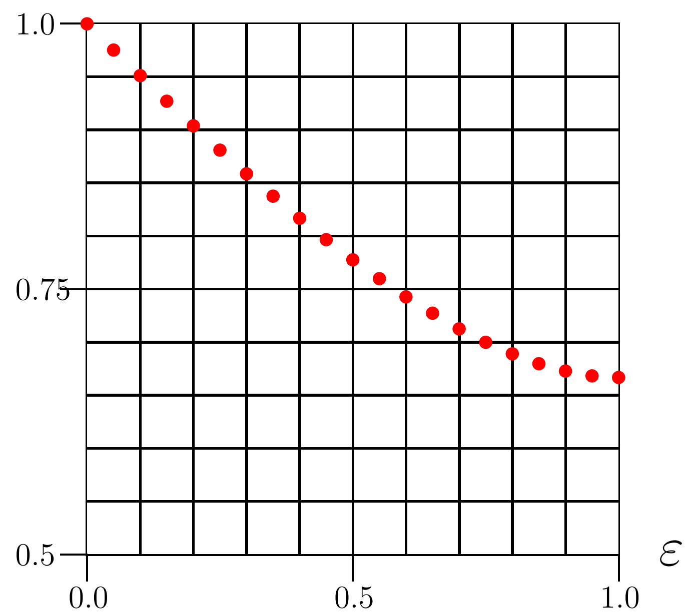

Figure 4.18: Percentage of Usable Subspace as a Function of ε

Of course we will get that member of TEMPSET which has the highest estimated probability of success, but if there was some subquery in QUERYSET with an even higher estimated probability of success, that should have been included in TEMPSET but was not, the final selection will not be the correct one. This might happen if, for example, there were a subquery in QUERYSET that had an estimated probability of success of .6, an ideal one of .7, and the random number generated by the algorithm for comparison was .65. In this case our intent is that the subquery be copied into TEMPSET but the error in the estimate prevents this from happening. If this subquery was the member of QUERYSET with the highest estimated probability of success then it would have been the one picked for expansion in the next step, if it had been correctly included in TEMPSET, which it was not. What we need to know is, what are the odds of the algorithm selecting the correct subquery for expansion next, given that the error in P_e might cause the correct subquery not to be copied into TEMPSET?

Starting with that subquery S, in QUERYSET, which has the highest estimated probability of success, there are exactly 4 (disjoint) possibilities:

1. S is selected for inclusion in TEMPSET, and the error (the difference between the estimated probability of success and the ideal one) is such that if the estimated probability had been the ideal one, the result would have been the same. In effect, the presence of the error in the estimate made no difference in the final outcome of the decision about the inclusion of S in TEMPSET. In this case S will, correctly, be selected to be expanded next. This possibility occurs when RANDOM is less than both P_e and P_i simultaneously.

2. S is not selected for inclusion in TEMPSET though it should have been (the problem being caused by the aforementioned error). In this case S will, incorrectly, not be selected to be expanded next. This possibility occurs when RANDOM is greater than P_e and less than P_i.

3. S is selected for inclusion though it should not have been. In this case S will, incorrectly, be selected to be expanded next. This possibility occurs when RANDOM is less than P_e and greater than P_i.

4. S is, correctly, not selected for inclusion. This case is indeterminate. The results may or may not be correct depending on the correctness of the selection process for the subquery in QUERYSET with the next highest estimated probability of success. This possibility occurs when RANDOM is greater than both P_e and P_i simultaneously. Regarding the selection of the next subquery in line, there are again 4 disjoint possibilities:

   (a) if the subquery with the next highest estimated probability should correctly be included in TEMPSET and is, the result will be a correct final selection.

(b) if the subquery should be included and is not then the result will be an incorrect final selection.

(c) if the subquery should not be included but is then the result will be an incorrect final selection.

(d) if it should not be included and is not, then the case is indeterminate. The results may or may not be correct depending on the correctness of the selection process for the subquery in QUERYSET with the next highest estimated probability of success, and so on, recursively.

The algorithm will select the correct subquery for expansion next when either possibility 1 occurs, or when possibility 4a occurs. If possibility 4d occurs the algorithm may generate a correct result, depending on which one of *its* sub-possibilities is chosen, and so on ad infinitum. We need to derive a formula for the probability that a correct result will occur.

We cannot know beforehand what the value of P_e or P_i will be for any given subquery (i.e. where it will fall in the 3-dimensional graph of which the graphs shown in figures 4.13 through 4.17 are 2-dimensional slices). Thus the most we can say is that the probability that any single query in QUERYSET will be correctly included (or not) in TEMPSET will be, on the average, equal to amount usable space (u) generated given the size of the error in our estimates. Thus, for any given subquery, the probability that it will be correctly included (or not) in TEMPSET is u.

Now let us imagine a subquery such that $P_e = 0.5$ and $\varepsilon = 0.1$. Our algorithm will correctly handle the subquery selection if $RANDOM < 0.4$ or $RANDOM > 0.6$ (the difference between P_e and P_i will only make a difference in the result when the value of RANDOM falls between the values of P_e and P_i). For this case it is true that the chance of RANDOM being less than both P_e and P_i is the same as the chance of RANDOM being greater than both P_e and P_i (remember that the value of RANDOM is generated from a flat distribution between 0 and 1). Obviously it is also true for any other value of ε, as long as $P_e = 0.5$ (see figure 4.19, noting that the areas of the yellow spaces on either side of the two broken, blue, vertical lines are the same. The red and yellow areas represent the usable space when $P_e = 0.5$).

Now let us take the case where $P_e = 0.25$ and $\varepsilon = 0.1$. Here our algorithm will correctly handle the subquery selection if $RANDOM < 0.15$ or $RANDOM > 0.35$. This case lacks the symmetry of the case where $P_e = 0.5$ (see figure 4.20). But keeping in mind that:

1. For each case where $P_e = 0.5 - x$ there is another non-symmetric case where $P_e = 0.5 + x$ which complements it. For example, when $P_e = 0.75$ and $\varepsilon = 0.1$ our algorithm will correctly handle the subquery selection if $RANDOM < 0.65$ or $RANDOM > 0.85$ (see figure 4.21, noting that the areas of the two yellow spaces are the mirror image of those of the yellow spaces in figure 4.20).

2. We have no way of knowing in advance where, in the range 0..1, a given subquery's estimated probability of success (P_e) will fall.

We may say that, on the average, for any given subquery, we expect that the probability of RANDOM being less than both P_e and P_i is the same as the probability of RANDOM being greater than both P_e and P_i.

Given that:

1. for any given subquery the probability that either possibility 1 or possibility 4 will occur is, on the average, u, and
2. for any given subquery we expect that, on the average, the probability of RANDOM being less than both P_e and P_i (possibility 1) is the same as the probability of RANDOM being greater than both P_e and P_i (possibility 4).

We conclude that:

1. the probability of possibility 1 occurring is, on the average, $u/2$, and
2. the probability of possibility 4 occurring is, on the average, $u/2$

Thus, if we consider the queries in QUERYSET in order from highest estimated probability of success to lowest, the probability that the first will be correctly handled (i.e. that possibility 1 or possibility 4 will occur) is:

$$\frac{u}{2} + \frac{u}{2}$$

Of course if possibility 4 occurs then the correct selection of a subquery will then depend on whether or not the subquery with the second highest estimated probability of success is also handled correctly. The odds of possibility 4 occurring for the first subquery were $\frac{u}{2}$ and, like the first subquery, the odds that the second subquery in line is handled correctly are $\frac{u}{2} + \frac{u}{2}$ (possibility 1 or possibility 4). So the odds that both the first and second subqueries in line are handled correctly (as regards subquery selection by the algorithm) are:

$$\frac{u}{2} + \frac{u}{2}\left(\frac{u}{2} + \frac{u}{2}\right) = \frac{u}{2} + \left(\frac{u}{2}\right)^2 + \left(\frac{u}{2}\right)^2$$

Of course if possibility 4 occurs for the second subquery in line we have the same problem that was present if it occurred for the first, causing us to go yet another level deeper, taking the first 3 subqueries into account. This gives us:

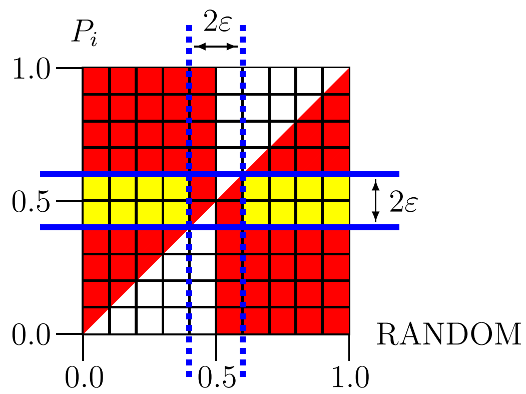

Figure 4.19: Symmetry of Selection When $P_e = 0.5$

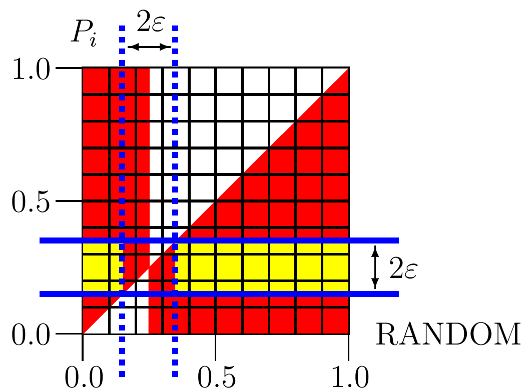

Figure 4.20: Asymmetry of Selection When $P_e = 0.25$

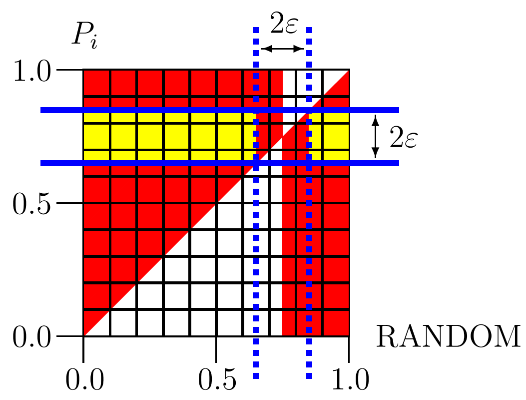

Figure 4.21: Asymmetry of Selection When $P_e = 0.75$

$$\frac{u}{2} + \left(\frac{u}{2}\right)^2 + \left(\frac{u}{2}\right)^3 + \left(\frac{u}{2}\right)^3$$

and so on (see figure 4.22).

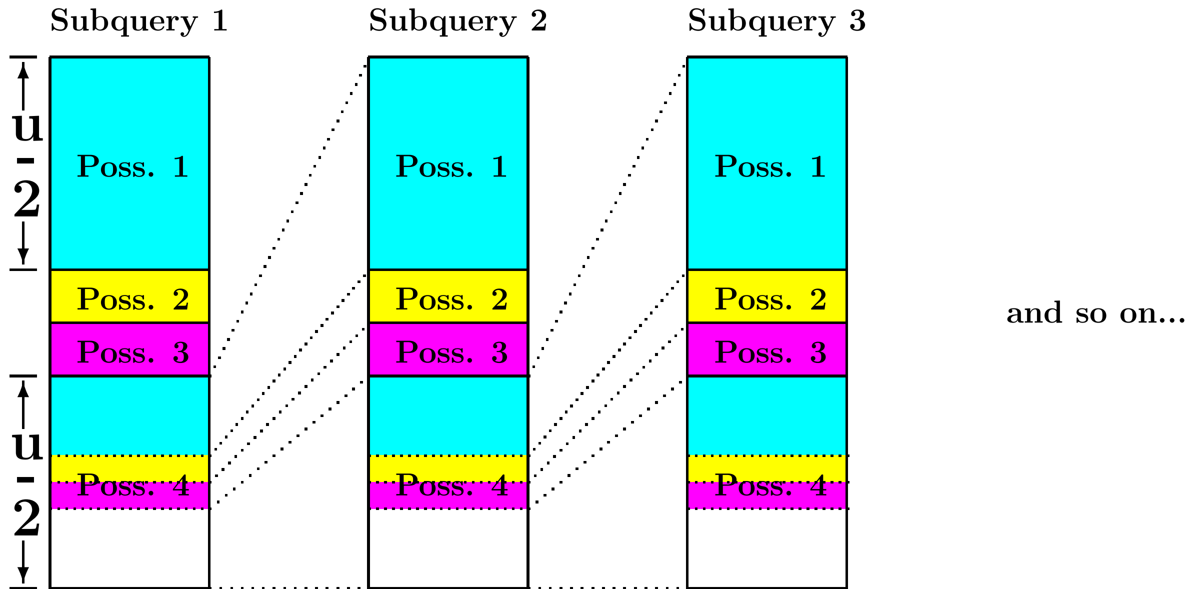

Figure 4.22: Probabilities for Selection of Subqueries for TEMPSET

Note that if the algorithm goes through the entirety of QUERYSET without selecting any query for inclusion in TEMPSET it starts over again with QUERYSET until it does. This makes the progression effectively infinite and leads us to the final formula:

$$\sum_{k=1}^{\infty} \left(\frac{u}{2}\right)^k =$$

$$\sum_{k=0}^{\infty} \left(\frac{u}{2}\right)^k - 1$$

The first part of this formula (everything except the "-1") is an infinite geometric series. Given that $\frac{u}{2}$ must fall within the range $0..\frac{1}{2}$ we know that this series converges [1] to:

[1] In an infinite geometric series, when the value of the term to be summed (r) falls between 1 and -1 (exclusive on both sides) the sum converges to $\frac{1}{1-r}$. See almost any basic calculus text for more details.

$$\frac{1}{1-\frac{u}{2}}$$

Adding the "−1" back in, the entire formula converges to [2]:

$$\frac{1}{1-\frac{u}{2}}-1 \tag{4.5}$$

Thus, a system can be trained to have an epsilon of no more than .03 (see table 6.7). This yields a usable space of about 0.98 (see figure 4.18). With this much usable space the probability that the algorithm would make the correct selection of which query to expand next would be:

$$\frac{1}{1-\frac{.98}{2}}-1$$

Or about 96%, where we use the term "correct selection" to mean that the final selection is the same selection that would have been made, given the values of RANDOM generated, had it been the case that $\varepsilon = 0$. i.e. that the presence of an error in our probability estimates made no difference in the final selection made by the algorithm.

The Error Function:

Taking the formula for usable subspace (formula 4.4) and plugging it into the formula for convergence (formula 4.5) we can create a combined formula giving us, for our probabilistic algorithm, the probability that the next node selected will be a correct one (see section 4.4.1), as a function of ε. This formula is:

$$\frac{1}{1-\frac{\frac{6-6\varepsilon+2\varepsilon^2}{3(2-\varepsilon)}}{2}}-1=$$

[2]Actually this formula very slightly underestimates the "goodness" of the algorithm, because of the presence of one special case. For example, let us assume that we are given a small QUERYSET with 10 items, such that the subquery with the highest probability of success is incorrectly not chosen to be copied into TEMPSET (possibility 2). There is still a small chance that the correct choice will be made for possibility 2 if, for every subquery following the first, that subquery is also not selected (correctly or incorrectly) to be copied into TEMPSET. Then the algorithm, faced with an empty TEMPSET, starts over again with QUERYSET, this time correctly selecting the previously-missed item. The probability of a randomly selected subquery (which could have any value between 0 and 1 for P_e) not being selected are $\frac{1}{2}$, making the probability that all other items in a 10-item QUERYSET are not selected, thus giving the desired subquery a second chance, to be $\frac{1}{2^9}$.

$$\frac{3 - 3\varepsilon + \varepsilon^2}{3 - \varepsilon^2} \tag{4.6}$$

and which is shown in figure 4.23.

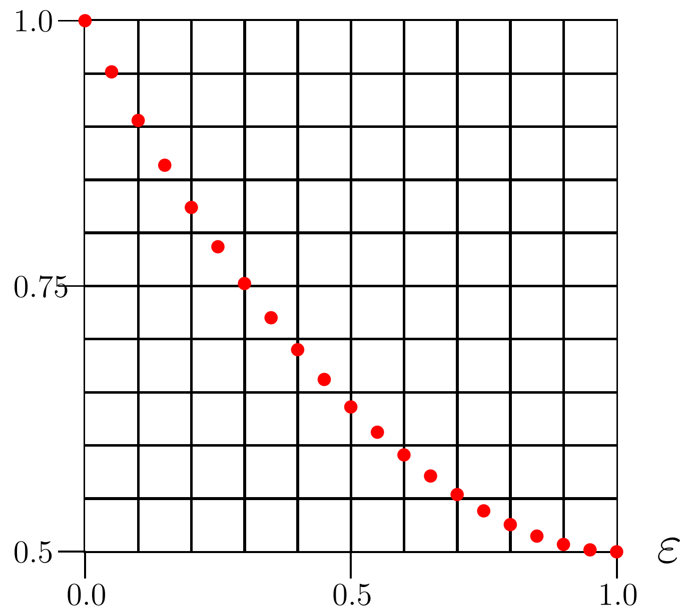

Figure 4.23: Probability of a Correct Selection as a Function of ε

This function actually provides a lower bound on the probability of making a correct selection. This is because of both the matter discussed in footnote 2 on the previous page (which we expect to be so small a factor in practice as to make no practical difference) and the fact that even possibilities 2 and 3 may, with sufficient luck, render a correct result. Let us assume, for example, that:

1. the subquery with the highest P_e (S1) is (incorrectly) not chosen for copying into TEMPSET (which is possibility 2), and

2. the subquery with the second highest P_e (S2) is (correctly) chosen, and

3. by chance, S2 happens to have a P_i whose value is within 2ε of the value of P_i for S1.

In this case the algorithm will return a correct selection, even though possibility 2 was involved (it is also possible for possibility 3 to yield a correct selection, for similar reasons).

Unfortunately for possibility 2 or 3 to yield a correct result relies on a chance distribution of the values of P_i for the nodes in TEMPSET (in our example, that the value of P_i for S2 was so close to that of P_i for S1), and cannot be predicted beforehand. For some problems to which this algorithm may be applied this closeness of values may rarely occur. For others it may occur often. This is why this function provides a lower bound, because it measures those conditions (possibilities 1 and part of 4) under which we can be certain that the algorithm will produce a correct result. Essentially this amounts to providing the probability that the node in TEMPSET with the highest value for its P_e, and all other nodes from QUERYSET with even higher values for their P_e's that might have been selected for copying into TEMPSET, but were not, were selected for copying (or not) correctly.

We say that function 4.6 provides a bound on the probability that the probabilistic algorithm will make a correct selection, in terms of the size of the error in our estimates, and we call it the "error function" for this algorithm. We believe that error functions may be derived for other probabilistic selection algorithms and that they may be used as another method for analyzing the performance of such algorithms, along with measurements of time and space complexity.

4.7 Exploration vs. Exploitation

- Exploration: Occurs when an algorithm makes use of problem space traversal to learn more useful things about the problem space.
- Exploitation: Occurs when an algorithm makes use of knowledge about the problem space to reach goals as efficiently as possible.

Algorithms used to traverse problem spaces differ in the extent to which they are explorative and/or exploitative. Often these two properties are in conflict with each other, heavily exploitative techniques doing little exploration and vice-versa. It is noteworthy that our probabilistic approach to node selection is both explorative and exploitative in its operation. It is exploitative in that nodes more likely to lead to quick solutions are expanded preferentially, and explorative in that nodes with lower probabilities will still be expanded occasionally, allowing for the slow accumulation of additional knowledge about the problem space. This knowledge may lead to quicker solutions for future queries.

4.8 Choosing the Best Conjunct to Process Next

Given a specific query to be processed the final decision that the system must make before processing can begin is, if the query contains more than one conjunct, which conjunct to

process next (remember that our system handles queries one conjunct at a time). Our system selects conjuncts in the following order:

1. Predicate instances, in order from most to least ground items in the argument list:
   Predicate instances with large numbers of ground items in the argument list are the predicate instances most likely to prevent the generation of child nodes. This is because the more ground items in the argument list the less likely there is to be a match among the database tuples or among the heads of the rules in the rulebase. Thus, choosing these conjuncts first will identify dead ends quickly, before resources are wasted processing other conjuncts from the same query (conjuncts that might not lead immediately to an inevitable dead end).

2. Relation instances, in order from first to last, skipping any negated ones.

3. The negated relation instance, if any:
   Saving the negated relation instance (if there is one) for last will ensure that all of the arguments of the negated relation instance are ground items, preventing the inadvertent formation of a ”floundering” subquery.

Chapter 5

The Knowledge Representation and Query Language

The knowledge representation and query language is a function-free subset of FOL, composed of the formats for expressing the contents of the knowledge base and the queries. The KB can be broken down into the following 5 parts:

5.1 Part 1: The Attribute Base

An attribute base entry is an attribute followed by the list of values that can be assigned to it. An example of this would be:

attribute102:
    value118, value119, value120, value121, value122, value123,
    value124, value125, value126, value127, value128

where in real life we might have an attribute like "color" to which could be assigned values like "red", "blue", etc...

5.2 Part 2: Predicate Instance Database Entries

This KB consists of a collection of tables for predicates, each describing a collection of ground facts which are instances of that predicate. A predicate instance database entry is a row in a table, such as that in the one row table of table 5.1, where "attribute138" is an attribute of the predicate and takes the value "value501". The ID "item551" is a key used by the system to identify the tuple, and can be treated as if it were an attribute value for an attribute named "ID", which attribute is shared by all predicates in the system.

The table entry implies the truth of the predicate instance "predicate103(item551, value501, value290, value1023)". Such a predicate instance is a ground atom. Items like item551 that appear in the predicate instance database comprise the set of items about which we can make statements in this language. Table 5.2 is the previous table as it might appear if the entries were not generic.

Table 5.1: Table for predicate103

ID:	attribute138	attribute118	attribute184
item551	value501	value290	value1023
*	*	*	*
*	*	*	*
*	*	*	*

Table 5.2: Non-Generic Table for dog

ID:	color	sex	vaccinated
item551	brown	male	yes
*	*	*	*
*	*	*	*
*	*	*	*

5.3 Part 3: Relation Instance Database Entries

The relation instance database is a collection of tables for relations, each describing a collection of relations that hold between objects, and each of which is a relation instance. A relation instance database entry is a row in a table, such as that in the one line table of table 5.3, where "predicate121" is a category of item of which item202 is a member. The ID "tuple6781" is a key used by the system to identify the tuple. The table entry implies the truth of the relation instance "relation103(item202, item157)". Table 5.4 is the previous table as it might appear if the entries were not generic (where item202 is, for example, a person named "Mary).

Table 5.3: Table for relation103

ID:	predicate121	predicate103
tuple6781	item202	item551
*	*	*
*	*	*
*	*	*

Table 5.4: Non-Generic Table for owns

ID:	owner	owned
tuple6781	item202	item551
*	*	*
*	*	*
*	*	*

Together, the predicate instance database and the relation instance database comprise the collection of ground facts contained in the KB.

5.4 Part 4: is-a Hierarchy Entries

The is-a hierarchy is a tree-structure of entries of the form:

predicate103: (ID, attribute138, attribute118, attribute184) is-a:
  predicate102: (ID, attribute138, attribute118)

For example, predicate103 might be "mammal", predicate102 "animal", and the extra attribute184 might be "fur color", which attribute is not relevant to non-mammals. Assuming that attribute138 is "number of legs", that attribute118 is "number of eyes" and that we have an instance of mammal such that mammal("Fido", 4, 2, brown) the presence of this entry in the is-a hierarchy means that:

animal("Fido", 4, 2) ← mammal("Fido", 4, 2, brown)

In general, given an entry of the form:

predicateX: (ID, $attribute_1$, ..., $attribute_m$, $attribute_{m+1}$, ..., $attribute_n$) is-a:
  predicateY: (ID, $attribute_1$, ..., $attribute_m$)

the entry means that:

predicateY(ID, $attribute_1$, ..., $attribute_m$) ←
  predicateX(ID, $attribute_1$, ..., $attribute_m$, $attribute_{m+1}$, ..., $attribute_n$)

5.5 Part 5: Rulebase Entries

The rulebase is a logic program, which is a collection of rules. The general form of a rule is:

$$P_1 \leftarrow P_2, ..., P_m, \neg Q_1, ..., \neg Q_n$$

where $P_1, ..., P_m$ and $Q_1, ..., Q_n$ are predicates, and all variables appearing in the rule are understood to be universally quantified.

Our rulebase is stratified and all of the rules that it contains are safe. A logic program is "stratified" when it does not contain sets of rules such that a predicate instance appearing as the head of a rule can derive its own negation. For, example, the two rules "predicate1 (...) ← predicate2 (...)" and "predicate2 (...) ← ¬predicate1 (...)" (the arguments are delete for brevity's sake) could not both be present at the same time. A rule is "safe" when all variables appearing in the head of the rule also appear in the body. A sample rule follows:

predicate102 (item456, variable244, value1255) $\leftarrow$
predicate101 (item456, variable244)
predicate110 (item457, value1320, value1258, value674)
relation118 (item456, item457)

where an argument like "item456" functions as a variable which will be replaced with the item number for a real item when the rule is applied.

5.6 Queries

The general form of a query (Q) is:

$$L_1 \wedge L_2 \wedge ...L_n$$

where $L_1, ..., L_n$ are literals (negated or non-negated predicates) and $y_1, ..., y_m$ are all of the free variables in Q (more on this in the next section). A sample follows:

p121 (item438: >variable790, value220, value914, value822, value384) $\wedge$
p107 (item439: value1329, value1255, variable791, value1026, value572) $\wedge$
r132 (item438, item439)

where arguments like "item438" function as they do in rulebase entries and and the ">" is used to designate the variable whose value the user is interested in.

5.7 Answers

Together the attribute base, predicate instance datebase, relation instance database, is-a hierarchy and rulebase entries make up the knowledge base (KB). A variable assignment is an assignment, to a set of variables in our language, of either a value appearing in the attribute base or an item appearing in the predicate instance database. A query (Q) can be answered if

$$KB \models \exists Y Q$$

where Y is the set of all free variables appearing in Q. When this is the case, and V is the variable assignment for Y then V is an answer for Q. The answer presented to the user is the subset of V that the user indicated, in the query, that he wished returned to him.

Chapter 6

Training the System

In two places in our system we make use of PHEs. In each case the formula used to derive the estimates is:

$$\frac{\prod_{i=1}^{t}[P(success|property_i)]}{P(success)^{t-1}} \tag{6.1}$$

(where I have replaced the 'n' with 't' to avoid confusion with another 'n' appearing in a formula below). Each term in this formula is a probability whose value must be discovered by application of the system to a set of training samples. For example, to discover the value of $P(success)$, which term appears in the denominator of (6.1), we must test many samples and see for what proportion of the samples the tests are successful. This proportion is the probability denoted by this term.

6.1 Training the System to Choose Subqueries for Further Processing

Our application of the formula is to derive the probability of success associated with a subquery, so that the best subqueries may be chosen for processing from a set of subqueries. When used in this fashion we say that success is an event that occurs "when a subquery is a member of a SSP" (see chapter 2). Using this definition, and a training set of limited size, we can approximate the value of formula 6.1 by:

$$\frac{\prod_{i=1}^{t}\left[\frac{number\,of\,subqueries\,generated,\,with\,property_i,\,that\,fall\,in\,a\,SSP}{number\,of\,subqueries\,generated\,with\,property_i}\right]}{\left(\frac{number\,of\,subqueries\,generated\,that\,fall\,in\,a\,SSP}{number\,of\,subqueries\,generated}\right)^{t-1}}$$

Corresponding to the numerator, the system will contain a table of cells such that there is one cell for each value that each $property_i$ can take. Each cell will contain two entries. One will record the number of subqueries generated by the system that have the requisite value for $property_i$. The other will record the number of those that have the further property of having been a member of a SSP.

Corresponding to the denominator, the system will contain two entries. One to record the number of subqueries generated by the system. The other to record the number of those that have the further property of having been a member of a SSP.

Training the system consists of processing enough sample queries to generate useful values for the table entries mentioned above. These values will be used to compute the necessary probabilities for use during query processing. The algorithm shown in figure 6.1 will be used for this purpose.

```
do:
  generate sample query;
  process sample query;
  for (each subquery in the resulting search tree)
    increment table entries for:
      subqueries generated;
      subqueries generated with the current subquery's properties;
  end for;
  if (an answer has been found) then
    for (each subquery in the resulting search tree in a SSP)
      increment table entries for:
        subqueries generated that fall in a SSP;
        subqueries generated, with the current subquery's properties, that fall in a SSP;
    end for;
  end if;
until (enough samples have been processed);
```

Figure 6.1: Method for Training the System to Choose Subqueries

Figure 6.2 represents a tree at the point at which the query processing algorithm has stopped. Note the three subqueries which are marked with an '$*$'. Let us assume that these three queries share the same value for some $property_i$. One of them is on a SSP. From this tree we may extract information to the extent that the probability of success (that a node will be on a SSP), for a node with the given value for $property_i$, with regards to this training example is 1/3.

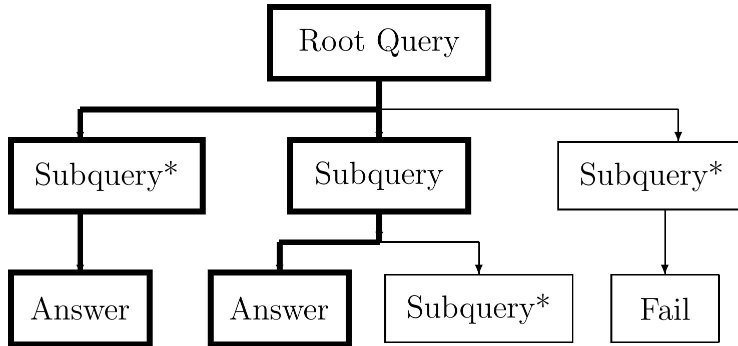

Figure 6.2: Approach for Training the System to Choose Subqueries

6.2 How Big Should the Set of Training Examples Be?

The question is: how many samples must we test, in determining the value of each term in formula 6.1, so that the value returned by it has a reasonable degree of certainty associated with it?

For individual probabilities the relationship between the number of samples, the confidence interval and the error is given by the following formula:

$$n = \frac{z^2_{\alpha/2} p(1-p)}{\varepsilon^2} \tag{6.2}$$

where n is the size of the sample set, α is 1 - the confidence coefficient (a confidence coefficient of .95 corresponds to a confidence level of 95%), p is the proportion of samples for which the test is a success, and ε is the absolute size of the error. This formula appears in many introductory statistics texts and states that with n samples there is a percent chance equal to the confidence level that the absolute error in p will be $< \varepsilon$. It is important to note that this formula cannot be used unless it is certain that the number of samples is large enough to validate the formula's approximation of the binomial distribution by the normal distribution, which requires that both of the following constraints hold true:

$$np > 5$$

$$n(1-p) > 5$$

Using formula 6.2 and these constraints we can compute, for any number of samples, confidence level and proportion, the maximum size of the error that we expect to see.

Both the numerator and the denominator of formula 6.1 are a product of individual probabilities with associated error terms. For the numerator:

$$\prod_{i=1}^{t}[P_i \pm \varepsilon_i] = P_{numerator} \pm \varepsilon_{numerator} \tag{6.3}$$

where P_i is the idealized value of the probability $P(success|property_i)$, which appears in the numerator of formula 6.1, as it would be if the value of $P(success|property_i)$ was error free. ε_i is the amount of error associated with $P(success|property_i)$, effectively $\mid P_i - P(success|property_i) \mid$.

For the denominator:

$$(P_{success} \pm \varepsilon_{success})^{t-1} = P_{denominator} \pm \varepsilon_{denominator} \tag{6.4}$$

where $P_{success}$ is the idealized value of the probability $P(success)$, which appears in the denominator of formula 6.1, as it would be if the value of $P(success)$ was error free. $\varepsilon_{success}$ is the amount of error associated with $P(success)$, effectively $\mid P_{success} - P(success) \mid$.

Thus, the formula itself is of the form:

$$\frac{P_{numerator} \pm \varepsilon_{numerator}}{P_{denominator} \pm \varepsilon_{denominator}} \tag{6.5}$$

and returns the final result $(P_{final} \pm \varepsilon_{final})$ that we are looking for.

What we need to know is, for the range of probabilities that we would like to be able to deal with, and the number of training samples that we expect, realistically, to be able to deal with, what will ε_{final} be in the worst case.

The final error (ε_{final}) can be derived from formula 6.5 above, via the equation

$$\frac{P_{numerator} \pm \varepsilon_{numerator}}{P_{denominator} \pm \varepsilon_{denominator}} = P_{final} \pm \varepsilon_{final} \tag{6.6}$$

Which, when adjusted for this purpose gives us:

$$\frac{P_{numerator} \pm \varepsilon_{numerator}}{P_{denominator} \pm \varepsilon_{denominator}} - P_{final} = \pm\varepsilon_{final}$$

Substitution from 6.3 and 6.4 above gives us:

$$\frac{\prod_{i=1}^{t}[P_i \pm \varepsilon_i]}{(P_{success} \pm \varepsilon_{success})^{t-1}} - P_{final} = \pm\varepsilon_{final}$$

Under ideal circumstances $P_{final} = \frac{\prod_{i=1}^{t} P_i}{P_{success}^{t-1}}$ (which is what we would get if the value of every ε_i and $\varepsilon_{success} = 0$). Inserting this into the previous equation gives us:

$$\frac{\prod_{i=1}^{t}[P_i \pm \varepsilon_i]}{(P_{success} \pm \varepsilon_{success})^{t-1}} - \frac{\prod_{i=1}^{t} P_i}{P_{success}^{t-1}} = \pm\varepsilon_{final}$$

6.3 Case 1 ($+\varepsilon_{final}$): The Math

To get the worst-case error for a given set of sample set parameters we must split this into two cases based on the sign of ε_{final}. The first case occurs when:

$$\frac{\prod_{i=1}^{t}[P_i \pm \varepsilon_i]}{(P_{success} \pm \varepsilon_{success})^{t-1}} - \frac{\prod_{i=1}^{t} P_i}{P_{success}^{t-1}} = \varepsilon_{final}$$

What we wish is a simple formula that will allow us to generate a worst case estimate of the value of ε_{final}. Unfortunately in the equation above ε_{final} is a function of too many different probabilities (P_i, where $i = 1...t$) for this equation to be useful. Fortunately there is a simple variant of this equation, gained by freezing the signs and probabilities at convenient values, that will serve:

$$\frac{(P_{max} + \varepsilon_{max})^t}{(P_{success} - \varepsilon_{success})^{t-1}} - \frac{P_{max}^t}{P_{success}^{t-1}} = \varepsilon_{worstcase} \tag{6.7}$$

where $P_{max} = max(P_1, ..., P_t)$ and $\varepsilon_{max} = max(\varepsilon_1, ..., \varepsilon_t)$.

In order to prove that this equation truly yields a worst case estimate we need to prove that $\varepsilon_{worstcase} \geq \varepsilon_{final}$ which, when written out, is:

$$\frac{(P_{max} + \varepsilon_{max})^t}{(P_{success} - \varepsilon_{success})^{t-1}} - \frac{P_{max}^t}{P_{success}^{t-1}} \geq \frac{\prod_{i=1}^{t}[P_i \pm \varepsilon_i]}{(P_{success} \pm \varepsilon_{success})^{t-1}} - \frac{\prod_{i=1}^{t} P_i}{P_{success}^{t-1}} \tag{6.8}$$

In proving this it is useful to note that, given that for all i $\varepsilon_i < P_i$ it must be true that:

$$\frac{\prod_{i=1}^{t}[P_i + \varepsilon_i]}{(P_{success} - \varepsilon_{success})^{t-1}} - \frac{\prod_{i=1}^{t} P_i}{P_{success}^{t-1}} \geq \frac{\prod_{i=1}^{t}[P_i \pm \varepsilon_i]}{(P_{success} \pm \varepsilon_{success})^{t-1}} - \frac{\prod_{i=1}^{t} P_i}{P_{success}^{t-1}}$$

This allows us to simplify 6.8 to:

$$\frac{(P_{max} + \varepsilon_{max})^t}{(P_{success} - \varepsilon_{success})^{t-1}} - \frac{P_{max}^t}{P_{success}^{t-1}} \geq \frac{\prod_{i=1}^{t}[P_i + \varepsilon_i]}{(P_{success} - \varepsilon_{success})^{t-1}} - \frac{\prod_{i=1}^{t} P_i}{P_{success}^{t-1}} =$$

$$\frac{(P_{max} + \varepsilon_{max})^t - \prod_{i=1}^{t}(P_i + \varepsilon_i)}{(P_{success} - \varepsilon_{success})^{t-1}} \geq \frac{P_{max}^t - \prod_{i=1}^{t} P_i}{P_{success}^{t-1}}$$

Comparing the denominators we can see that:

$$(P_{success} - \varepsilon_{success})^{t-1} \leq P_{success}^{t-1}$$

is true. Comparing the numerators to show that:

$$(P_{max} + \varepsilon_{max})^t - \prod_{i=1}^{t}(P_i + \varepsilon_i) \geq P_{max}^t - \prod_{i=1}^{t} P_i$$

is true takes a bit more work. First we will rearrange the terms to get:

$$(P_{max} + \varepsilon_{max})^t - P_{max}^t \geq \prod_{i=1}^{t}(P_i + \varepsilon_i) - \prod_{i=1}^{t} P_i \tag{6.9}$$

then we will make use of the following equation:

$$\prod_{i=1}^{t}(X_i) - \prod_{i=1}^{t}(Y_i) = \sum_{j=1}^{t}\left[\prod_{i=1}^{j-1}(Y_i)(X_j - Y_j)\prod_{i=j+1}^{t}(X_i)\right] \tag{6.10}$$

where X and Y are lists of t numbers. This equation is proven true by the following sequence of algebraic manipulations:

$$\sum_{j=1}^{t}\left[\prod_{i=1}^{j-1}(Y_i)(X_j - Y_j)\prod_{i=j+1}^{t}(X_i)\right] =$$

$$\sum_{j=1}^{t}\left[X_j\prod_{i=1}^{j-1}(Y_i)\prod_{i=j+1}^{t}(X_i) - Y_j\prod_{i=1}^{j-1}(Y_i)\prod_{i=j+1}^{t}(X_i)\right] =$$

$$\sum_{j=1}^{t}\left[\prod_{i=1}^{j-1}(Y_i)\prod_{i=j}^{t}(X_i) - \prod_{i=1}^{j}(Y_i)\prod_{i=j+1}^{t}(X_i)\right] =$$

$$\sum_{j=1}^{t}\left[\prod_{i=1}^{j-1}(Y_i)\prod_{i=j}^{t}(X_i)\right] - \sum_{j=1}^{t}\left[\prod_{i=1}^{j}(Y_i)\prod_{i=j+1}^{t}(X_i)\right] =$$

$$\left\{\prod_{i=1}^{t}(X_i) + \sum_{j=2}^{t}\left[\prod_{i=1}^{j-1}(Y_i)\prod_{i=j}^{t}(X_i)\right]\right\} - \left\{\sum_{j=1}^{t-1}\left[\prod_{i=1}^{j}(Y_i)\prod_{i=j+1}^{t}(X_i)\right] + \prod_{i=1}^{t}(Y_i)\right\} =$$

$$\left\{\prod_{i=1}^{t}(X_i) + \sum_{j=1}^{t-1}\left[\prod_{i=1}^{j}(Y_i)\prod_{i=j+1}^{t}(X_i)\right]\right\} - \left\{\sum_{j=1}^{t-1}\left[\prod_{i=1}^{j}(Y_i)\prod_{i=j+1}^{t}(X_i)\right] + \prod_{i=1}^{t}(Y_i)\right\} =$$

$$\prod_{i=1}^{t}(X_i) - \prod_{i=1}^{t}(Y_i)$$

Applying equation 6.10 to 6.9 we get:

$$\sum_{j=1}^{t}\left[\prod_{i=1}^{j-1}(P_{max})(\varepsilon_{max})\prod_{i=j+1}^{t}(P_{max}+\varepsilon_{max})\right] \geq \sum_{j=1}^{t}\left[\prod_{i=1}^{j-1}(P_i)(\varepsilon_j)\prod_{i=j+1}^{t}(P_i+\varepsilon_i)\right] \tag{6.11}$$

Which, since each term in the summation on the left can be seen to be $\geq$ to its associated term in the summation on the right, can be seen to be true.

This, together with the inequality seen to hold between the numerators of inequality 6.8, demonstrates inequality 6.8 to be true, providing us with:

Lemma 1 $$\frac{(P_{max}+\varepsilon_{max})^t}{(P_{success}-\varepsilon_{success})^{t-1}} - \frac{P_{max}^t}{P_{success}^{t-1}} \geq \frac{\prod_{i=1}^{t}[P_i \pm \varepsilon_i]}{(P_{success}\pm\varepsilon_{success})^{t-1}} - \frac{\prod_{i=1}^{t} P_i}{P_{success}^{t-1}}$$

Which tells us that our formula for deriving worst-case error values for our probabilities:

$$\frac{(P_{max}+\varepsilon_{max})^t}{(P_{success}-\varepsilon_{success})^{t-1}} - \frac{P_{max}^t}{P_{success}^{t-1}}$$

truly yields worst-case results in all possible situations involving the first case.

6.4 Case 1 ($+\varepsilon_{final}$): The Resulting Numbers

The next step is to show that, when training with a tractable number of samples, our worst-case estimator (for this case) yields a usable degree of certainty for the resulting probabilities. Taken from our experiment, reasonable values for the various parts of formulas 6.2 and 6.1 are, for the numerator: n = 40,000, α = .05 and t = 4 and for the denominator: n = 400,000, α = .05 and t = 4. Feeding these values into formula 6.2 along with various values for P provides the necessary values for use in the various parts of equation 6.7 (see tables 6.1 and 6.2). From the contents of tables 6.1 and 6.2 we can generate, using equation 6.7, the table below, which shows the values of ε_{final} as a function of P_{max} and $P_{success}$ (see table 6.3).

Table 6.1: Case1: Values for the Numerator

P_{max}	P_{max}^t	$\approx \varepsilon_{max}$	$(P_{max} + \approx \varepsilon_{max})^t$
.01	.00000001	.000975	.0000000145
.1	.0001	.00294	.000112
.3	.0081	.00449	.00860
.5	.0625	.00490	.0650
.7	.2401	.00490	.247
.9	.6561	.00490	.671
.99	.96059601	.00490	.980

Table 6.2: Case1: Values for the Denominator

$P_{success}$	$P_{success}^{t-1}$	$\approx \varepsilon_{success}$	$(P_{success} - \approx \varepsilon_{success})^{t-1}$
.01	.000001	.000308	.000000910
.1	.001	.000930	.000972
.3	.027	.00142	.0266
.5	.125	.00155	.124
.7	.343	.00142	.341
.9	.729	.000930	.727
.99	.970299	.000308	.969

Table 6.3: Case 1: ε_{final} as a function of P_{max} and $P_{success}$

		P_{max}						
		.01	.1	.3	.5	.7	.9	.99
$P_{success}$	.01	.00593	X	X	X	X	X	X
	.1	.00000492	.0152	X	X	X	X	X
	.3	.000000175	.000604	.0233	X	X	X	X
	.5	.0000000369	.000103	.00455	.0242	X	X	X
	.7	.0000000134	.0000369	.00160	.00840	.0243	X	X
	.9	.00000000623	.0000169	.000718	.00367	.0104	.0230	X
	.99	.00000000466	.0000125	.000527	.00267	.00745	.0163	.0214

6.5 Case 2 ($-\varepsilon_{final}$): The Math

The second case occurs when:

$$\frac{\prod_{i=1}^{t}[P_i \pm \varepsilon_i]}{(P_{success} \pm \varepsilon_{success})^{t-1}} - \frac{\prod_{i=1}^{t} P_i}{P_{success}^{t-1}} = -\varepsilon_{final}$$

or:

$$\frac{\prod_{i=1}^{t} P_i}{P_{success}^{t-1}} - \frac{\prod_{i=1}^{t}[P_i \pm \varepsilon_i]}{(P_{success} \pm \varepsilon_{success})^{t-1}} = \varepsilon_{final}$$

What we wish is a simple formula that will allow us to generate a worst case estimate of the value of ε_{final}. Unfortunately in the equation above ε_{final} is a function of too many different probabilities (P_i, where $i = 1...t$) for this equation to be useful. Fortunately, as before, there is a simple variant of this equation, gained by freezing the signs and probabilities at convenient values, that will serve:

$$\frac{P_{max}^{t}}{P_{success}^{t-1}} - \frac{(P_{max} - \varepsilon_{max})^{t}}{(P_{success} + \varepsilon_{success})^{t-1}} = \varepsilon_{worstcase} \tag{6.12}$$

In order to prove that this equation truly yields a worst case estimate we need to prove that $\varepsilon_{worstcase} \geq \varepsilon_{final}$ which, when written out, is:

$$\frac{P_{max}^t}{P_{success}^{t-1}} - \frac{(P_{max} - \varepsilon_{max})^t}{(P_{success} + \varepsilon_{success})^{t-1}} \geq \frac{\prod_{i=1}^t P_i}{P_{success}^{t-1}} - \frac{\prod_{i=1}^t [P_i \pm \varepsilon_i]}{(P_{success} \pm \varepsilon_{success})^{t-1}} \tag{6.13}$$

In proving this it is useful to note that, given that for all i $\varepsilon_i < P_i$ it must be true that:

$$\frac{\prod_{i=1}^t P_i}{P_{success}^{t-1}} - \frac{\prod_{i=1}^t [P_i - \varepsilon_i]}{(P_{success} + \varepsilon_{success})^{t-1}} \geq \frac{\prod_{i=1}^t P_i}{P_{success}^{t-1}} - \frac{\prod_{i=1}^t [P_i \pm \varepsilon_i]}{(P_{success} \pm \varepsilon_{success})^{t-1}}$$

This allows us to simplify 6.13 to:

$$\frac{P_{max}^t}{P_{success}^{t-1}} - \frac{(P_{max} - \varepsilon_{max})^t}{(P_{success} + \varepsilon_{success})^{t-1}} \geq \frac{\prod_{i=1}^t P_i}{P_{success}^{t-1}} - \frac{\prod_{i=1}^t [P_i - \varepsilon_i]}{(P_{success} + \varepsilon_{success})^{t-1}}$$

or:

$$\frac{P_{max}^t - \prod_{i=1}^t P_i}{P_{success}^{t-1}} \geq \frac{(P_{max} - \varepsilon_{max})^t - \prod_{i=1}^t (P_i - \varepsilon_i)}{(P_{success} + \varepsilon_{success})^{t-1}}$$

Comparing the denominators we can see that:

$$P_{success}^{t-1} \leq (P_{success} + \varepsilon_{success})^{t-1}$$

is true. Next we must compare the numerators and show that:

$$P_{max}^t - \prod_{i=1}^t P_i \geq (P_{max} - \varepsilon_{max})^t - \prod_{i=1}^t (P_i - \varepsilon_i) \tag{6.14}$$

As we did for the first case, we apply equation 6.10 to 6.14 and get:

$$\sum_{j=1}^t \left[\prod_{i=1}^{j-1} (P_i)(P_{max} - P_j) \prod_{i=j+1}^t (P_{max}) \right] \geq$$
$$\sum_{j=1}^t \left[\prod_{i=1}^{j-1} (P_i - \varepsilon_i)((P_{max} - \varepsilon_{max}) - (P_j - \varepsilon_j)) \prod_{i=j+1}^t (P_{max} - \varepsilon_{max}) \right] =$$

$$\sum_{j=1}^{t}\left[\prod_{i=1}^{j-1}(P_i)(P_{max}-P_j)\prod_{i=j+1}^{t}(P_{max})\right] \geq$$
$$\sum_{j=1}^{t}\left[\prod_{i=1}^{j-1}(P_i-\varepsilon_i)((P_{max}-P_j)-(\varepsilon_{max}-\varepsilon_j))\prod_{i=j+1}^{t}(P_{max}-\varepsilon_{max})\right]$$

Which, since each term in the summation on the left can be seen to be $\geq$ to its associated term in the summation on the right, can be seen to be true. Together with the inequality seen to hold between the numerators of inequality 6.13, this demonstrates inequality 6.13 to be true, providing us with:

Lemma 2 $$\frac{P_{max}^t}{P_{success}^{t-1}} - \frac{(P_{max}-\varepsilon_{max})^t}{(P_{success}+\varepsilon_{success})^{t-1}} \geq \frac{\prod_{i=1}^{t} P_i}{P_{success}^{t-1}} - \frac{\prod_{i=1}^{t}[P_i \pm \varepsilon_i]}{(P_{success} \pm \varepsilon_{success})^{t-1}}$$

From which we can conclude that our formula:

$$\frac{P_{max}^t}{P_{success}^{t-1}} - \frac{(P_{max}-\varepsilon_{max})^t}{(P_{success}+\varepsilon_{success})^{t-1}}$$

for deriving worst-case error values for our probabilities truly yields worst-case results in all possible situations involving the second case.

The next step is to show that, when training with a tractable number of samples, our worst case estimator yields a usable degree of certainty for the resulting probabilities.

6.6 Case 2 ($-\varepsilon_{final}$): The Resulting Numbers

Taken from our experiment, reasonable values for the various parts of formulas 6.2 and 6.1 above are, for the numerator: n = 40,000, α = .05 and t = 4 and for the denominator: n = 400,000, α = .05 and t = 4. Feeding these values into formula 6.2 along with various values for P provides us with the necessary values for use in the various parts of equation 6.12 (see tables 6.4 and 6.5).

Table 6.4: Case2: Values for the Numerator

P_{max}	P^t_{max}	$\approx \varepsilon_{max}$	$(P_{max} - \approx \varepsilon_{max})^t$
.01	.00000001	.000975	.00000000663
.1	.0001	.00294	.0000887
.3	.0081	.00449	.00763
.5	.0625	.00490	.0601
.7	.2401	.00490	.233
.9	.6561	.00490	.642
.99	.96059601	.00490	.942

Table 6.5: Case2: Values for the Denominator

$P_{success}$	$P^{t-1}_{success}$	$\approx \varepsilon_{success}$	$(P_{success} + \approx \varepsilon_{success})^{t-1}$
.01	.000001	.000308	.00000110
.1	.001	.000930	.00103
.3	.027	.00142	.0274
.5	.125	.00155	.126
.7	.343	.00142	.345
.9	.729	.000930	.731
.99	.970299	.000308	.971

From tables 6.4 and 6.5 we can generate, using equation 6.12, the table below, which shows the value of ε_{final} as a function of P_{max} and $P_{success}$ (see table 6.6).

Table 6.6: Case 2: ε_{final} as a function of P_{max} and $P_{success}$

		P_{max}						
		.01	.1	.3	.5	.7	.9	.99
$P_{success}$	.01	.00397	X	X	X	X	X	X
	.1	.00000356	.0139	X	X	X	X	X
	.3	.000000128	.000466	.0215	X	X	X	X
	.5	.0000000274	.0000960	.00424	.0230	X	X	X
	.7	.00000000994	.0000344	.00150	.00801	.0246	X	X
	.9	.00000000465	.0000158	.000673	.00352	.0106	.0218	X
	.99	.00000000348	.0000117	.000490	.00252	.00749	.0150	.0199

6.7 The Final Result (math and numbers)

We cannot determine, in advance, which of the two special cases the error will involve. Therefore, to be safe, we must derive a final worst-case error formula which produces, as the error, the maximum of the two errors produced by the formulas we derived from lemmas 1 and 2 . Combining lemmas 1 and 2 gives us the following theorem:

Theorem 1

$$\max(\frac{(P_{max}+\varepsilon_{max})^t}{(P_{success}-\varepsilon_{success})^{t-1}} - \frac{P_{max}^t}{P_{success}^{t-1}}, \frac{P_{max}^t}{P_{success}^{t-1}} - \frac{(P_{max}-\varepsilon_{max})^t}{(P_{success}+\varepsilon_{success})^{t-1}}) \geq$$

$$\max(\frac{\prod_{i=1}^t [P_i \pm \varepsilon_i]}{(P_{success}\pm\varepsilon_{success})^{t-1}} - \frac{\prod_{i=1}^t P_i}{P_{success}^{t-1}}, \frac{\prod_{i=1}^t P_i}{P_{success}^{t-1}} - \frac{\prod_{i=1}^t [P_i \pm \varepsilon_i]}{(P_{success}\pm\varepsilon_{success})^{t-1}})$$

Using this theorem we can derive our final worst-case error formula and use it to generate a final set of worst-case error values for examination (see formula 6.15 and table 6.7).

$$\max(\frac{(P_{max}+\varepsilon_{max})^t}{(P_{success}-\varepsilon_{success})^{t-1}} - \frac{P_{max}^t}{P_{success}^{t-1}}, \frac{P_{max}^t}{P_{success}^{t-1}} - \frac{(P_{max}-\varepsilon_{max})^t}{(P_{success}+\varepsilon_{success})^{t-1}}) \quad (6.15)$$

Table 6.7: Final Values: ε_{final} as a function of P_{max} and $P_{success}$

		P_{max}						
		.01	.1	.3	.5	.7	.9	.99
	.01	.00593	X	X	X	X	X	X
	.1	.00000492	.0152	X	X	X	X	X
	.3	.000000175	.000604	.0233	X	X	X	X
$P_{success}$	.5	.0000000369	.000103	.00455	.0242	X	X	X
	.7	.0000000134	.0000369	.00160	.00840	.0246	X	X
	.9	.00000000623	.0000169	.000718	.00367	.0106	.0230	X
	.99	.00000000466	.0000125	.000527	.00267	.00749	.0163	.0214

6.8 Generating the Training Examples

We will need a large set of randomly generated queries to train the system with. Each training query is generated via the following algorithm:

1. Generate some random predicate instances:
   Predicates are randomly selected and each attribute randomly assigned a ground item or variable as its associated value. A list of the variables used so far is maintained so that some of them can be reused in other randomly generated predicate instances, as circumstances and randomness permits, to create links between the predicate instances.

2. Add in some relation instances connecting some of the predicate instances to each other:
   For each pair of predicate instances already added, see if there exists a relation that can be used to relate them. If so, generate a random number and if the number so indicates use the relation to generate a relation instance, that relates the pair of predicate instances, to add to the query.

3. Break variable-based connections between predicate instances connected by relations:
   If two predicate instances are connected by a relation instance, check to see if they are also connected by sharing a variable. If so, generate a random number, and if it so indicates replace one of the instances of the shared variable with a new variable of the same type (causing the variables to function as wildcards).

4. Choose a relation instance to negate:
   If the query does not already contain a negated relation instance generate a random number and, if the number so indicates, choose one relation instance and negate it.

5. Choose a variable to be the one whose value is being sought:
   Randomly select, from among all variables appearing in the query, one variable to be the one whose value the imaginary user is seeking the value of.

The training set thus created may be pre-screened (using an exhaustive query processing algorithm), and any number of unanswerable queries thrown out, so as to adjust the ratio of answerable-queries to unanswerable-queries as desired.

Chapter 7

The Experiment

Once the system has been trained it will be used to process a randomly generated set of test queries (not taken from the list of training examples).

7.1 The Algorithms to be Tested

Each test query will be processed using the following algorithms:

1. Exhaustive:
   At each step all unprocessed subqueries will be processed (i.e. the algorithm proceeds in breadth-first fashion). As each level of subqueries in the search tree is processed, all of the new subqueries thus generated are checked for the presence of an answer.

2. Random:
   At each step the best subquery to process next will be chosen randomly from the set of all unprocessed subqueries.

3. Best-First:
   At each step the best subquery to process next will be chosen from the set of all unprocessed subqueries using the probabilities generated by our use of PHEs (see section 4.3).

4. Random-Best-First:
   The same as Best-First, only the probabilities used will be random numbers generated using a flat distribution between 0 and 100.

5. Hybrid:
   At each step a list of potential next-queries-to-process is made by comparing each existing query's probability of success with a random number (generated from a flat distribution from 0 to 100). If the random number is less than the probability of success the query is copied into the new list, and once the new list has been completed the next query to process is selected from it in best-first fashion (see section 4.5).

6. Random-Hybrid:
   The same as Hybrid, only the probabilities used will be random numbers generated using a flat distribution between 0 and 100.

The only exception to the algorithm descriptions above occurs when there is encountered a query whose processing involves the handling of a negated conjunct. Here we use negation-as-failure, so the negated conjunct is un-negated, and the query is processed using a recursive call to a new instance of the current algorithm. This call is completely processed before control passes back to the calling instance, and the results used to determine the status of the query with the negated conjunct.

Each algorithm stops when it has either found an answer or hit an assigned "resource limit". This limit is hit when at least one of the following has occurred:

- the total number of nodes in the tree of subqueries generated so far exceeds 600
- the total number of levels in the tree of subqueries generated so far exceeds 50
- the total number of negation levels (requiring the use of negation-as-failure and a recursive call of the algorithm to another instance of itself to process that query) in the tree of subqueries generated so far exceeds 2

7.2 The Measurements to be Made

Because of the probabilistic nature of some of the algorithms used, each test query will be processed many times, for those algorithms that make use of probabilistic selection, once for the others. The average of all scores generated by an algorithm will be used as a measure of the performance of that algorithm. Items to be measured are:

1. The number of nodes that the algorithm expands, averaged over all queries attempted.
2. The number of nodes that the algorithm expands, averaged over all queries solved.
3. The number of queries solved.
4. The number of nodes in the path leading to the solution found, averaged over all queries solved.

7.3 Training the System

Training the system consists of presenting the system with a large number of randomly generated queries for it to solve. The process of solving a query generates a "query tree". This is a tree structure each node of which is a query. The root of the tree is the original query and every other query is a subquery of the query that is its parent node in the tree, generated by applying some knowledge within the system to the parent query, with the goal of taking the query one step closer to a solution. Once generated, the query tree is examined by the system to determine how the heuristic properties of the various queries in the tree are associated with that query's being on (or not being on) the shortest path leading to a node that represents an answer to the original query. In detail, the process proceeds as follows:

1. Generate 20 random queries, all of which are solvable by the Exhaustive algorithm within the pre-set resource limits (algorithm and resource limits are discussed in section 7.1).
2. For each of these 20 queries:
   (a) Solve the query using the Exhaustive algorithm, generating a query tree in the process.
   (b) For each query in the query tree and heuristic property available to the system: Determine the heuristic value of the query for the given heuristic property, and wether or not the query is on a SSP in the query tree, and update the system's probability data accordingly. For instance, if the query has 3 conjuncts, and is on a SSP, we take the stored values regarding the total number of queries encountered so far with 3 conjuncts (T) and the number of queries encountered so far with 3 conjuncts that were on a SSP (S), and increment both by one. Thus if, previously, the probability that a query with 3 conjuncts was on a SSP was $\frac{S}{T}$, now it becomes $\frac{S+1}{T+1}$.
3. Examine all of the stored data available for all of the heuristic values that can be encountered by the system. If, for at least 80% of the heuristic values, the system has collected enough data for the related probability to be considered trained to within the desired tolerance (in our system we require a 95% probability that no error in probability greater than .003 will occur) then training is finished. If not, then return to step 1 and continue. The formula used to determine if the system has been sufficiently well trained for a specific heuristic value is formula 6.2, and its attendant constraints, as discussed in section 6.2.

In practice the system tended to take from 10 to 12 hours to fully train, on a 1.2Ghz AMD Athlon processor (Thunderboard core), in a process involving a sample set of about 1200 queries (on average) with an average total of approximately 380,000 subqueries examined.

About 80% of the run time was involved in step 1 (see above) as it took many attempts to randomly generate each query whose solution process was both non-trivial (involving a query tree of at least 60 nodes and a solution path at least 5 hops long) and within the preset resource limits (as discussed above). When using a randomly generated set of KBs most randomly generated queries tend to be either trivially unsolvable, or to exceed the resource limits, and are thus thrown out without being used. In practice, between 5 and 10 queries were usually generated for each usable query achieved. As a result of this it is reasonable to assume that in a real-life system, with an already stored set of actual user queries to use for training, a similar level of training could be reached in well under an hour.

7.4 Selecting the Heuristics to be Used

Before running the test data it is necessary to choose from amongst the available 29 heuristics a small subset to be used by the smart algorithms in query processing. To do this the following process was used:

1. Generate 3 queries randomly.
2. For each query, run it through the algorithms several times (for each individual heuristic used alone), to get a reliable figure regarding the performance of each algorithm on each query when used with each prospective heuristic by itself.
3. Generate 10 queries randomly.
4. Select from the list of available heuristics those that, when used in isolation, consistently caused the smart algorithms to perform well. When doing this care must be taken to avoid selecting any pairs of heuristics that are too closely related, like "number of conjuncts" and "number of conjuncts that are predicate instances". This is done to avoid any significant violation of the independence assumptions that went into our mathematics.
5. Using both the smart algorithms and their random counterparts, test the selected heuristics (as a group) on the 10 random queries.
6. If the performance is sufficiently good, commit to the selected heuristics. If not, return to step 4, re-examine the numbers, and try again.
7. Once committed, do performance testing using a newly randomly generated set of test queries. Once we have committed to a set of heuristics in step 6, whatever results the performance testing yields must be accepted. To reselect new heuristics after this point if the performance testing yields poor results would risk retrofitting the data.

Final groups of selected heuristics tended to contain about 4 members. The heuristics that tended to repeatedly show up as being useful were those involving the number of conjuncts in the query, the number of arguments taken by those conjuncts, and the number of arguments that have ground items assigned to them.

7.5 Details of the Experiment

The experiment used to test the techniques proposed in this document proceeded as follows:

1. Randomly generate a set of knowledge bases. This consisted of:
   (a) an attribute base, containing a list of attributes and the values that each can take.
   (b) an is-a hierarchy, containing about 40 categories of items in a tree structure 5-6 levels deep with each node having, on average, 2 children.
   (c) a relationbase, containing about 100 relations for relating items that are instances of the categories contained in the is-a hierarchy.
   (d) a database of tuples relating to the predicates appearing in the is-a hierarchy and the relations in the relationbase. Each tuple is either a collection of attribute values defining one specific item (a member of one of the categories appearing in the is-a hierarchy), or a collection of items, related by the relation with which the tuple is associated.
   (e) a rulebase, containing about 200 rules related to the various predicates and relations already mentioned.
2. Train the system (generate the needed probabilities) as in chapter 7.3.
3. Select the heuristics to be used using the method discussed in section 7.4.
4. Randomly generate 100 queries, using the same method used to randomly generate queries for training the system, as discussed in chapter 7.3.
5. Run each of the test queries through each of the algorithms discussed in section 7.1, running each one through 5 times in the case of the those algorithms that contain a random element.
6. For each run take the measurements discussed in section 7.2, averaging them together as appropriate to get the final results.

The entire procedure was repeated 4 times, for 4 different randomly generated sets of knowledge bases, and the final results averaged together as appropriate.

7.6 A Sample Query

In an actual run, our system randomly generated the following query[1]:

100:
drug (drugNumber?: >name?, manufacturer=Merck, form=liquid, fdaStatus=Approved, ...),
condition (conditionNumber?: temperature=103, pulse=82, diastolicBloodPressure?, ...),
cures (drugNumber?, conditionNumber?)[2]

which, translated into something like English, reads as:

"What is the name of a drug that is manufactured by Merck, liquid in form, approved by the FDA, etc... and cures some condition characterized by a temperature of 103, a pulse of 82, any diastolic blood pressure, etc..."

Using the Hybrid algorithm the processing of the query proceeds as in figure 7.1. The root node is the original query. Each box in the tree represents one subquery and the contents of each box are in the form "node-number:node-score". The blue boxes represent the solution path, with node 142 being the point at which a solution was found.[3] Sample subqueries from the tree, and transitions between them, are provided below.

[1] In the actual output generated by the system the terms relating to drugs and conditions were not used. The system uses generic terms like "r100" instead of the relation "cures". The drug-related terminology was substituted in for the sake of readability.

[2] The marks ", ..." at the end of the lists of arguments are not literal parts of the query, they merely indicate where I cut off listing the values of arguments for the sake of brevity. Similarly, the use of "etc..." in the translation immediately below is not a literal translation of the query into English, but just a placeholder to indicate where the cutting off of arguments took place.

[3] The node-numbers that are missing from the query tree do not indicate existing tree-nodes that were not included in the diagram, which is complete. During query processing the system occasionally attempts to create a subquery, only finding out after this process has begun that no valid subquery can be created at that point. The partially-begun subquery is discarded but the already-used node-number cannot be re-used, and the missing numbers can appear, incorrectly, to be gaps in the query tree.

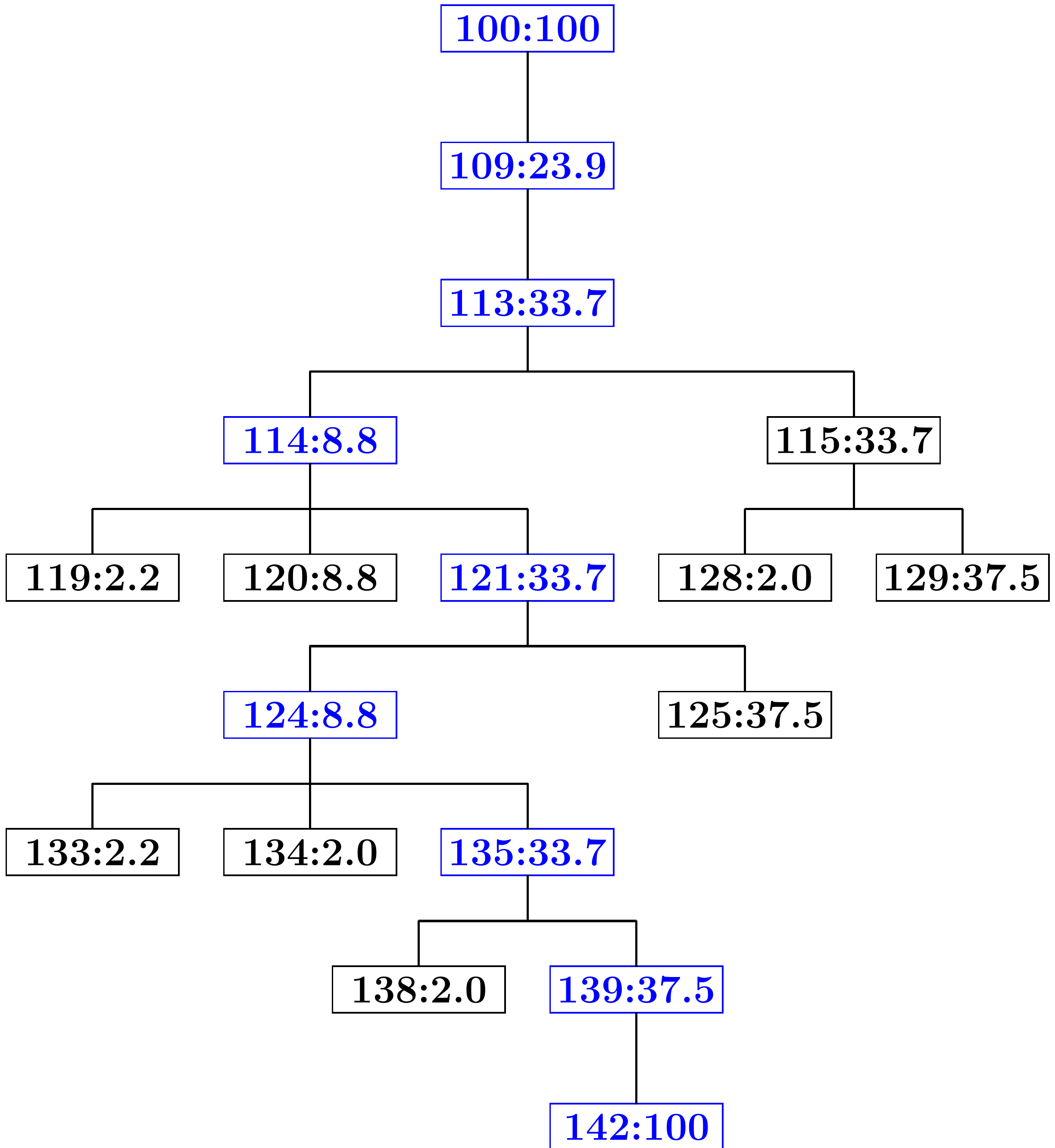

Figure 7.1: Progress Through A Sample Query (The Hybrid Algorithm)

Some of the subqueries in the query tree of figure 7.1, and the transitions between them, are described below.

100:
drug (drugNumber?: >name?, manufacturer=Merck, form=liquid, fdaStatus=Approved, ...),
condition (conditionNumber?: temperature=103, pulse=82, diastolicBloodPressure?, ...),
cures (drugNumber?, conditionNumber?)

becomes:

109:
answer=CureAll
condition (conditionNumber?: temperature=103, pulse=82, diastolicBloodPressure?, ...),
cures (drugNumber201, conditionNumber?)

via a predicate-instance database entry of the form:

drug (drugNumber201: name=CureAll, manufacturer=Merck, form=liquid,
fdaStatus=Approved, ...)

113:
answer=CureAll
cures (drugNumber201, conditionNumber137)

becomes:

114:
answer=CureAll
cures (drugNumber201, conditionNumber?),
causedBy (conditionNumber137, conditionNumber?)

via a rulebase entry of the form:

cures (drug?, condition1?) ←
cures (drug?, condition2?),
causedBy (condition1?, condition2?)

114:
answer=CureAll
cures (drugNumber201, conditionNumber?),
causedBy (conditionNumber137, conditionNumber?)

becomes:

121:
answer=CureAll
causedBy (conditionNumber137, conditionNumber133)

via a relation-instance database entry of the form:

cures (drugNumber201, conditionNumber133)

7.7 The Results

The final results are broken down into sections depending on the specific measurement being discussed.

7.7.1 Nodes Expanded Averaged Over All Queries Solved

A node is "expanded" if the system processes the associated subquery and generates whatever children of that subquery that it can, in an attempt to get closer to a solution. Expanding subqueries is where the system does its query processing "work" and as such we would like to reduce as much as possible the number of subqueries that must be expanded to reach a solution.

The numbers relating to algorithm performance can be seen in table 7.1 (the "R" in an algorithm name being short for "random"):

"Nodes Expanded" represents the average number of nodes that were expanded by the system in solving those queries for which a solution was actually found (i.e. excluding from consideration those queries for which the system "gave up" due to query processing reaching the preset resource limit). This information is presented in graphical form in figure 7.2.

Our two benchmark random algorithms (R-Best-First and R-Hybrid) answer around 60% of the queries presented. For the rest of the queries they go off on "false trails" through the query tree, giving up when the resource limit is reached. Our two "smart" algorithms (Best-First and Hybrid) do much better, heading more directly for the answers (note the reduced number of nodes expanded) for a far greater number of queries (averaging around 86%).

Table 7.1: Algorithm Performance for Solved Queries (the numbers)

Algorithm	Queries Solved	Nodes Expanded
Exhaustive	100.0	109.0
Random	79.4	48.1
Best-First	82.8	25.3
R-Best-First	57.1	35.9
Hybrid	88.7	30.6
R-Hybrid	60.1	39.9

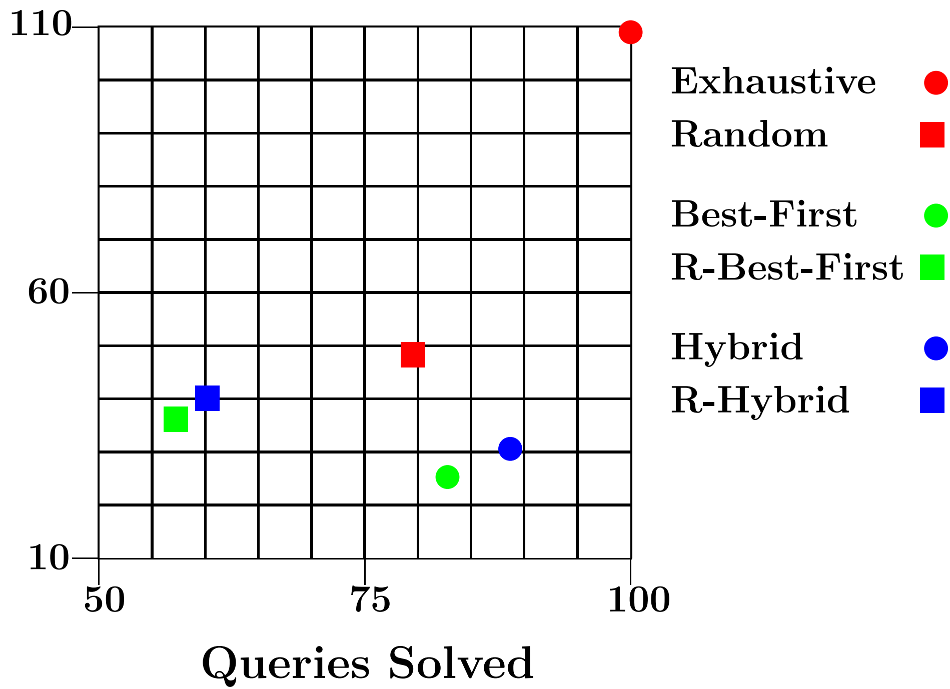

Figure 7.2: Algorithm Performance (for solved queries)

It is interesting to note that, despite what might be expected, the Hybrid algorithm seems to outperform Best-First, answering almost 6% more queries without a large increase in the number of nodes expended. This is a consequence of the "unforgiving" nature of Best-First, which depends on having access to very accurate heuristic values when traversing a large search space for which the scores for the individual nodes are similar. Looking at figure 7.3, taken from an actual run, we can see how this happens in practice.

When node 122 is reached (the contents of each box in the figure are in the format "node-number:node-score") two children are generated, numbered 123 and 124. The Best-First algorithm chooses node 123 to process next, and quickly generates more children and grandchildren whose scores are slightly greater than those of node 124, preventing 124 from being taken before the resource limit is reached. In contrast, during many runs of Hybrid node 124 is chosen for expansion first. In many others 123 is chosen first but the probabilistic nature of Hybrid causes it to go back and expand node 124 soon anyway, unless the probabilities for the subnodes of node 123 are much higher, in which case they are fairly likely to lead to a solution anyway. If node 124 turns out to be a bad move, then its subnodes are likely to have even lower probabilities, causing Hybrid to refocus its attention on the subnodes of 123. The result of this is exactly what you would expect, that Hybrid solves more queries, but does extra-work doing so as more false-trails are followed. As we can see, in practice the trade-off turns out to be a good one.

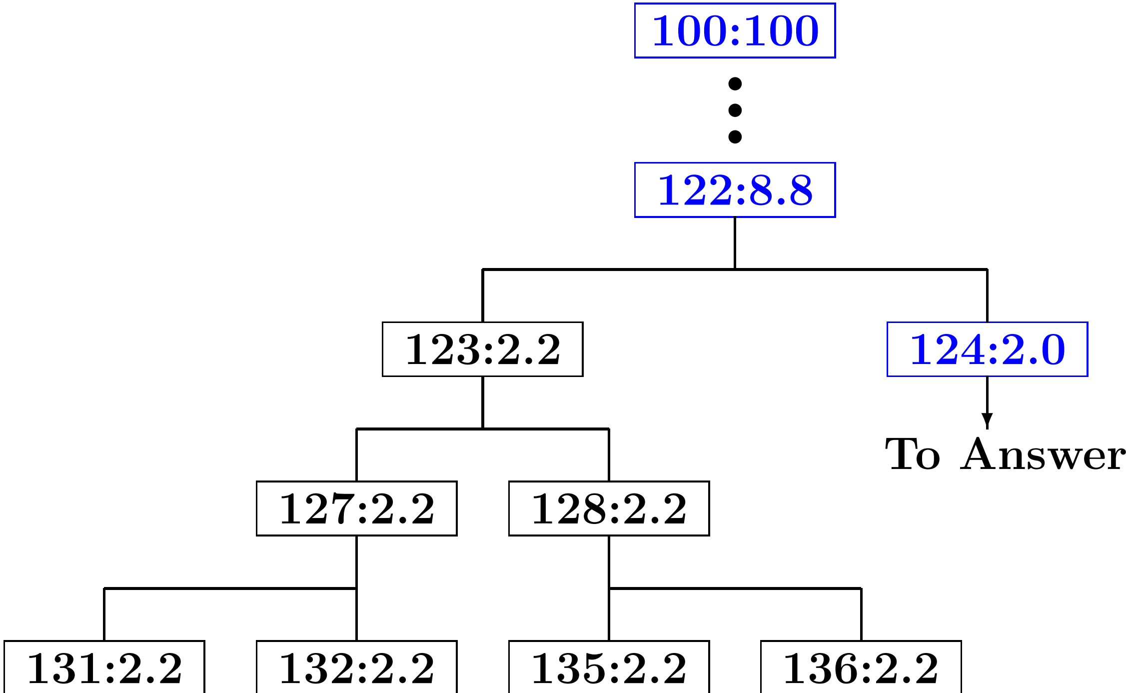

Figure 7.3: Hybrid vs. Best-First Algorithm Comparison

The Exhaustive algorithm finds solutions for all queries, but does it at a high price, its breadth-first nature forcing it to expand a large number of nodes in the process. The Random algorithm is the best performing of the non-PHE-using, probabilistic algorithms. Its random nature prevents it from competing successfully with the "smart" algorithms, but because subqueries are never assigned specific probabilities it cannot run into the kind of problem that prevents Best-First from outperforming Hybrid. Specifically, it cannot suffer from the problem of mistakenly assigning, to at least one subquery on a good path, a randomly generated, overly low probability of success. Mistakes of this type are likely to cause the good path to be ignored in favor of lessor paths, preventing an answer from being found within the resource limits, hence the Random algorithm's tendency to solve significantly more queries than either Random-Best-First or Random-Hybrid.

7.7.2 Nodes Expanded Averaged Over All Queries Attempted

The figures below can be most easily understood in terms of the previous section. A query that is not solved has a much higher cost than one that is, because it keeps expanding nodes up until the resource limit is reached. The larger the percentage of queries an algorithm does not solve the greater its performance suffers because of this. When measuring the number of nodes expanded over all queries attempted (not just the ones solved) we take this extra cost into consideration. These results are the most important, as they represent most accurately the system's performance as an actual user will see it.

This section's graph is, in effect, the previous section's graph with each dot raised by an amount indicating how much it has suffered from this effect. Thus, this graph is much like the one of the previous section, save that each dot has been raised (its y-axis value increased), with the dots that were farthest towards the left in the last section's graph having been raised the most, since they solved the smallest percentage of queries. The numbers can be seen in table 7.2.

Herein, "Nodes Expanded" represents the average number of nodes that were expanded by the system in attempting to solve a query. This information is presented in graphical form in figure 7.4.

Alternatively, we may combine our two coordinates and derive a single numerical score indicating the relative performance of the various algorithms via the formula $score = \frac{queries\ solved}{nodes\ expanded}$. This yields the numbers in table 7.3.

7.7.3 Solution Path Length Averaged Over All Queries Solved

Many of the queries presented to the system have more than one possible solution, or may have multiple paths of different lengths that lead to the same solution. Here we look at how each algorithm performed in terms of finding better or worse solutions to the queries (by

Table 7.2: Algorithm Performance for All Queries (the numbers)

Algorithm	Queries Solved	Nodes Expanded
Exhaustive	100.0	109.0
Random	79.4	76.1
Best-First	82.8	50.9
R-Best-First	57.1	95.5
Hybrid	88.7	49.1
R-Hybrid	60.1	95.5

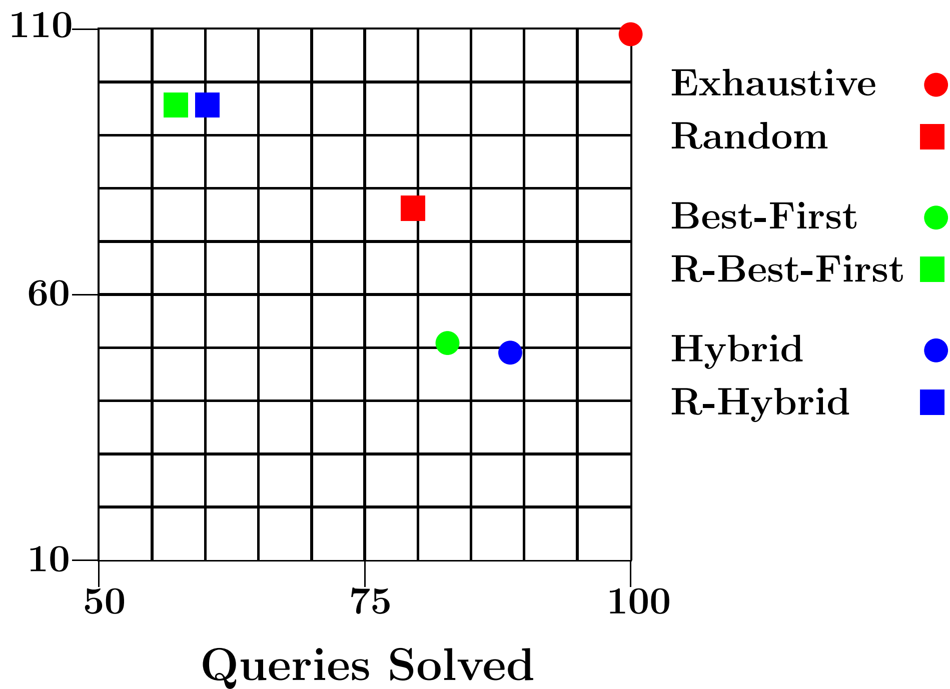

Figure 7.4: Algorithm Performance (for all queries)

which we mean solutions closer to or farther from the initial query in terms of the number of processing steps required to reach it).

Regarding average path length the algorithms perform as common sense would have us predict (given their performance in the previous two sections). Exhaustive always finds the shortest possible path to every solution. The smart algorithm Best-First does slightly better than this, because it is not finding solutions for all of the queries and it tends to not solve queries whose shortest possible solution path is longer than most. This is because if there is one too-low query score in a long path then the path will never be taken to completion, an event more likely to occur in long solution paths than short ones. As before, Hybrid solves more queries than Best-First, but at the cost of not performing quite as well. R-Best-First and R-Hybrid continue to perform similarly and poorly and Random, because of its tendency to build query trees in a somewhat breadth-first fashion, continues to outperform R-Best-First and R-Hybrid without doing as well as their smarter counterparts.

The numbers can be seen in table 7.4 and the graphical representation of the data in figure 7.5.

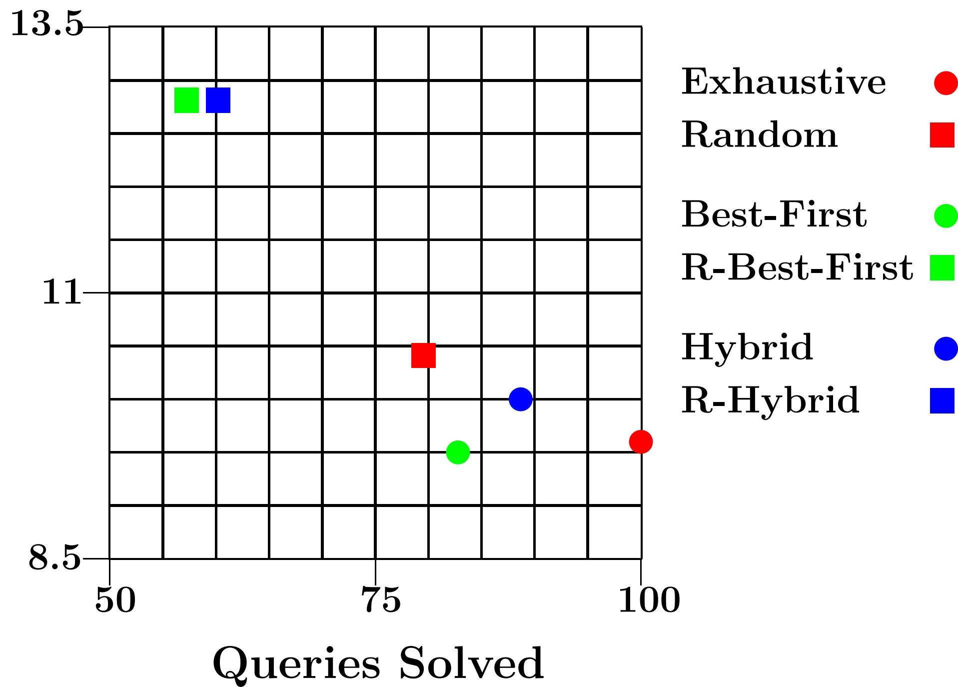

Figure 7.5: Nodes In Solution Path

Table 7.3: Algorithm Scores for All Queries

Algorithm	Score
Exhaustive	0.9
Random	1.0
Best-First	1.6
R-Best-First	0.6
Hybrid	1.8
R-Hybrid	0.6

Table 7.4: Path Length for Solved Queries (the numbers)

Algorithm	Queries Solved	Nodes In Path
Exhaustive	100.0	9.6
Random	79.4	10.4
Best-First	82.8	9.5
R-Best-First	57.1	12.8
Hybrid	88.7	10.0
R-Hybrid	60.1	12.8

Chapter 8

Summary

In this dissertation we have presented a collection of new techniques that aid in the process of carrying out query processing, in a complex KBS, without being overcome by the problem of combinatorial explosion. Theory was presented that justifies the use of the new techniques, and a KBS was built and the new techniques tested therein. The final results of the testing demonstrate that the techniques work in practice as well as in theory. Among the new developments presented are:

1. A probabilistic algorithm for carrying out PHE-based query processing, which algorithm outperforms best-first in this problem domain.
2. A mathematical analysis of the new algorithm that introduces the notion of the "error function", which function may provide a new tool for comparing the expected performance of probabilistic algorithms for selecting the next node to search in a problem space.
3. A technique for determining, in advance, the number of training examples necessary to train a PHE-based query processing system for any desired level of statistical confidence in its answers.
4. A demonstration of the effectiveness of using the concept of the shortest successful path to a solution, as opposed to the presence of a solution, as a training criteria.
5. A demonstration of the fact that PHE-based algorithms can be made to work well with no lookahead, a result which contradicts the conclusions presented in the PHE-related literature to date, and significantly expands the potential usefulness of this technique.
6. A demonstration of the effectiveness of the use of PHEs with multiple (more than 2) heuristic properties being used simultaneously on a single node, as opposed to 1 or (in a very recent paper) 2 heuristic properties used with multiple lookahead nodes.

Chapter 9

Future Work

Future work on this project is expected to include:

1. Automation of the process of heuristic selection. Heuristic selection is done by a process that is mostly mechanical in nature. It should be possible to turn this process entirely over to the computer with only minor modifications to the presently-used technique.

2. The use of negative heuristics to help the system avoid selecting subqueries that would be bad ones to expand next, in addition to using positive heuristics to help the system in selecting subqueries that would be good ones to expand next (which possibility was mentioned in chapter 2).

3. Investigation into the possibility that it is more important for some nodes in the search space to be assigned accurate scores than others. During programming a bug caused one of the query processing algorithms to operate in this fashion, assigning random scores to some nodes and accurate scores to others, which operation yielded significantly better results than were expected.

4. The use of PHEs for other phases of the query processing process, such as the selection of which conjunct is the best one to process next, and which KBs should be instructed to work on the subquery currently being expanded.

5. Facilities for the assigning of confidence levels to individual KBs or their contents, for use in selecting how a query is to be processed in very large scale KBSs with large numbers of KB components.

Bibliography

[Ali 93] Ali, S., "A 'Natural Logic' for Natural Language Processing and Knowledge Representation", Ph.D. Dissertation, State University of New York at Buffalo, November 1993.

[Bres 96] Bresina, J., "Heuristic-Biased Stochastic Sampling", Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96), 1996, pps. 271-278.

[Chak 86] Chakrabarti, P., Ghose, S., DeSarkar, S., "Heuristic Search Through Islands", Artificial Intelligence, 29, 1986, pps. 339-347.

[Clar 97] Clark, P., Porter, B. "Building Concept Representations from Reusable Components", AAAI, 1997, pps. 369-376.

[Craw 91] Crawford, J., Kuipers, B., "Algernon - A Tractable System for Knowledge Representation", SIGART Bulletin, Vol. 2, No. 3, 1991, pps. 35-44.

[Domi 97] Domingos, P., Pazzani, M., "On the Optimality of the Simple

Bayesian Classifier under Zero-One Loss", Machine Learning, No. 29, 1997, pps. 103-130.

[Falk 91] Falkenhainer, B., Forbus, K., "Compositional Modeling: Finding the Right Model for the Job", Artificial Intelligence, No. 51, 1991, pps. 95-143.

[Freu 97] Freund, R., Wilson, W., "Statistical Methods", Academic Press, 1997.

[Gold 96] Golding, A., Rosenbloom, P., "Improving Accuracy by Combining Rule-Based and Case-Based Reasoning", Artificial Intelligence, No. 87, 1996, pps. 215-254.

[Grub 93] Gruber, T., "A Translation Approach to Portable Ontology Specifications", Knowledge Acquisition, Vol. 5, No. 2, 1993, pps. 199-220.

[Grub 96] Gruber, T., Olsen, G., "The Configuration Design Ontologies and the VT Elevator Domain Theory", International Journal of Human and Computer Studies, No. 44, 1996, pps. 569-598.

[Hank 90] Hanks, S., "Controlling Inference in Planning Systems: Who, What, When, Why, and How", Proceedings of the AAAI Spring Symposium on Planning in Uncertain, Unpredictable, or Changing Environments, 1990.

[Hans 89] Hansson, O., Mayer, A., "Heuristic Search as Evidential Reasoning", Proceedings of the Fifth AAAI Workshop on Uncertainty in AI, Windsor, Ontario, 1989, pps. 152-161.

[Hans 90a] Hansson, O., Mayer, A., "Probabilistic Heuristic Estimates", Annals of Mathematics and Artificial Intelligence, No. 2, 1990, pps. 209-220.

[Hans 90b] Hansson, O., Mayer, A., Russell, S., "Decision-Theoretic Planning in BPS", Proceedings of the AAAI Spring Symposium on Planning in Uncertain, Unpredictable, or Changing Environments, 1990.

[Hans 98] Hansson, O., "Bayesian Problem-Solving Applied to Scheduling", Ph.D. Dissertation, University of California at Berkeley, Fall 1998.

[Hart 68] Hart, P., Nilsson, N., Raphael, B., "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", IEEE Transactions on Systems Science and Cybernetics, Vol. 4, No. 2, 1968, pps. 100-107.

[Haye 81] Hayes, P., "The Logic of Frames", appearing in: "Readings in Artificial Intelligence", edited by: Webber, B., Nilsson, N., Tioga, 1981.

[Jaga 89] Jagannathan, V., Dodhiawala, R., Baum, L., (editors) "Blackboard Architectures and Applications", Academic Press, 1989.

[Kahn 73] Kahneman, D., Tversky, A., "On the Psychology of Prediction", Psychological Review, Vol. 80, No. 4, 1973, pps. 237-251.

[Kwa 89] Kwa, J., "BS^*: An Admissible Bidirectional Staged Heuristic Search Algorithm", Artificial Intelligence, No. 38, 1989, pps. 95-109.

[Lark 80] Larkin, J., McDermott, J., Simon, D., Simon, H., "Models of Competence in Solving Physics Problems", Cognitive Science, No. 4, 1980, pps. 317-345.

[Lena 90] Lenat, D., Guha, R., "Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project", Addison-Wesley, 1990.

[Lena 95] Lenat, D., "CYC: A Large-Scale Investment in Knowledge Infrastructure", Communications of the ACM, Vol. 38, No. 11, November 1995, pps. 32-38.

[Lloy 87] Lloyd, J., "Foundations of Logic Programming", Springer-Verlag, 1987.

[Maye 94] Mayer, A., "Rational Search", Ph.D. Dissertation, University of California at Berkeley, December 1994.

[Mich67] Michie, D., "Strategy-Building with the Graph Traverser", Appears In: "Machine Intelligence 1", edited by: Collins, N., Michie, D., Publisher: Oliver and Boyd, 1967, pps. 135-152. Also found as: Proceedings of the First International Machine Intelligence Workshop, 1965.

[Mugg 00] Muggleton, S., "Semantics and Derivation for Stochastic Logic Programs", submitted to The Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI-2000): Workshop on Fusion of Domain Knowledge with Data for Decision Support, appears in

Proceedings of the UAI-2000 Workshop on Fusion of Domain Knowledge with Data for Decision Support, Dybowski, R., (editor).

[Mugg 01] Muggleton, S., "Stochastic Logic Programs", Submitted to The Journal of Logic Programming, 2001 (accepted subject to revision). Note: From vol. 47 (2001) on this journal has been retitled "The Journal of Logic and Algebraic Programming". This is an extended version of a paper of the same title which appeared in "Advances in Inductive Logic Programming", De Raedt, L., (editor), IOS Press, 1995.

[Nels 90] Nelson, P., Dillenburg, J., "Multiple Level Island Search", Proceedings of the Fifth Rocky Mountain Conference on Artificial Intelligence: Pragmatics in Artificial Intelligence, 1990, pps. 231-238.

[Nels 92] Nelson, P., Toptsis, A., "Unidirectional and Bidirectional Search Algorithms", IEEE Software, Vol. 9, No. 2, March 1992, pps. 77-83.

[Papa 95] Papakonstantinou, Y., Garcia-Molina, H., Widom, J., "Object Exchange Across Heterogeneous Information Sources", Proceedings of the IEEE 11th International Conference on Data Engineering, 1995, pps. 251-260.

[Pate 86] Patel, V., Groen, G., "Knowledge Based Solution Strategies in Medical Reasoning", Cognitive Science, No. 10, 1986, pps. 91-116.

[Pear 84] Pearl, J., "Heuristics: Intelligent Search Strategies for Computer Problem Solving", Addison-Wesley, 1984.

[Pear 88] Pearl, J., "Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference", Morgan Kaufmann, 1988.

[Pohl 71] Pohl, I., "Bi-Directional Search", appearing in "Machine Intelligence 6", Editors: Meltzer, B., Michie, D., Edinburgh University Press, 1971, pps. 127-140. Can also be found as: "Proceedings of the Sixth Annual Machine Intelligence Workshop", Edinburgh, 1970.

[Rick 97] Rickel, J., Porter, B., "Automated Modeling of Complex Systems to Answer Prediction Questions", Artificial Intelligence, No. 93, 1997, pps. 201-260.

[Russ 95] Russell, S., Norvig, P., "Artificial Intelligence: A Modern Approach", Prentice-Hall, 1995.

[Shan 50] Shannon, C., "Programming a Computer for Playing Chess", Philosophical Magazine, Vol. 41, No. 4, 1950, pps. 256-275.

[Smir 96] Smirnov, Y., Koenig, S., Veloso, M., Simmons, R., "Efficient Goal-Directed Exploration", Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96), 1996, pps. 292-297.

[Sun 95] Sun, R., "Robust Reasoning: Integrating Rule-Based and Similarity-Based Reasoning", Artificial Intelligence, No. 75, 1995, pps. 241-295.

[Tver 74] Tversky, A., Kahneman, D., "Judgement under Uncertainty: Heuristics and Biases", Science (New Series), Vol. 185,

Issue 4157, September 27, 1974, pps. 1124-1131.

[van 97] van Heijst, G., Schreiber, A., Wielinga, B., "Using Explicit Ontologies in KBS Development", International Journal of Human and Computer Studies, No. 45, 1997, pps. 183-292.

Vita

Kevin Grant was born in New Orleans, Louisiana, on March 10, 1965. During his college years he obtained a bachelor of arts degree in philosophy, a bachelor of science degree in computer science and a master of science degree in computer science all from the University of New Orleans. His master's studies were concentrated in the area of database theory and his thesis was concerned with the problem of performing the transitive closure operation (a very useful but computationally very expensive database operation) in an efficient fashion.

While in the master's program he co-authored two papers. The topic of the first was the efficient discovery of and deletion of duplicate database entries in parallel database systems and was published in a peer-reviewed journal. The second concerned the performance of various transitive closure algorithms in theory and practice and was delivered in person at a small conference in Lafayette, Louisiana.

Upon completion of his master's degree he was awarded a fellowship to attend Louisiana State University as a doctoral candidate in the department of computer science (with a graduate minor in psychology). His studies were concentrated in the areas of artificial intelligence and machine learning and culminated in the submission of the dissertation of which this vita is a part.

The cumulative GPA for all graduate work done over the course of his lifetime to date is a perfect 4.0. In January of 2004 he will begin the process of extracting the contents of this dissertation into a collection of publications.