Integrated Analytic and Empirical Learning

of Approximations for Intractable Theories

Thomas Paul Ellman

CUCS-511-89

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY
December 1989

ABSTRACT

Integrated Analytic and Empirical Learning of Approximations for Intractable Theories

Thomas Paul Ellman

A powerful new technique for learning to solve intractable problems is presented in this dissertation. Although computational theories can be formulated for many problem-solving tasks, such theories are often intractable because they require excessive computational resources. This research has investigated a strategy of sacrificing the correctness of theories in order to gain tractability in return. Initially intractable theories are approximated by adopting *generic simplifying assumptions*. For instance, one generic assumption asserts that the value of a function is the same for all arguments. Another asserts that a random variable is equally likely to manifest any of its legal values. Although such assumptions are not strictly true, they can greatly simplify otherwise intractable computations. They often result in approximate theories with acceptable levels of accuracy and efficiency.

A program called "*POLLYANNA*" has been developed to investigate this approach to the intractable theory problem. *POLLYANNA* was developed using the card game hearts and a job scheduling problem as test bed domains. This program combines analytic and empirical learning methods in a generate and test framework. During the analytic phase, candidate approximate theories are generated by systematically applying generic simplifying assumptions to an initially intractable theory. Analytic generation results show that many candidates are semantically equivalent to informally stated heuristic decision rules, including the *Dump High Rank* rule in the hearts domain and the *Soonest Scheduling Deadline First* rule in the job scheduling domain. During the empirical phase, candidates are tested against teacher-provided training examples. Measurements of accuracy and efficiency are used to guide a search for approximate theories meeting specified accuracy and efficiency goals. Empirical testing results show the candidates to manifest a gradual tradeoff between accuracy and efficiency. Some approximate theories are efficient but highly prone to errors, while others are more accurate but computationally expensive. Depending on the context in which a computational theory is used, different combinations of accuracy and efficiency are appropriate. This research thus demonstrates a flexible approach to approximate theory formation.

Acknowledgments

I would like to express my sincere thanks to Michael Lebowitz, my advisor. Michael has contributed to this research in many more ways than I can possibly acknowledge. He has been a source of continuous guidance and support throughout my graduate studies at Columbia. I would also like to thank my thesis committee members Jack Mostow and John Kender. Their advice, encouragement and challenging questions have been tremendously helpful to me. I feel most fortunate to have worked with three such dedicated people.

I have made many wonderful friends at Columbia, especially Doree Seligmann, Kevin Matthews, Kenny Wasserman, Andrea Danyluk, Dannie Durand, Michelle Baker and Dan Yellin. I would like to thank them all for sharing with me the many aspects of life as a graduate student. Finally, I would like to thank my parents. They taught me both the discipline I needed to complete this task, and the love of knowledge that has made it ultimately so rewarding.

This research was supported in part by the Defense Advanced Research Projects Agency under contract N00039-84-C-0165.

Junk page.

Chapter 1

Introduction

1.1. Intractability and Machine Learning

1.1.1. The Ubiquity of Intractable Theories

Computational intractability is a fact of life in artificial intelligence. Correct computational theories can be formulated for many problem solving applications. Unfortunately the costs of computation often make them useless in practice. If a computational theory requires inordinate time or space resources, the theory has little or no practical value for solving problems. Even when the theory includes knowledge that is sufficient in principle to solve a problem, computational intractability can render the knowledge unavailable for practical problem solving. Problems of intractability arise in all areas of artificial intelligence, including games [Berliner and Ebeling 89; Pearl 84], planning [Korf 87], databases [Ullman 82], robotics, vision [Tsotsos 87], and natural language understanding [Savitch, et al. 87]. Outside of AI per se, problems of intractability arise frequently in combinatorial optimization, [Papadimitriou and Steiglitz 82; Garey and Johnson 79] and numerical analysis [Ralston and Rabinowitz 78; Kronsjo 79].

1.1.2. A Machine Learning Approach to Intractability

This research has investigated a machine learning approach to the problem of computationally intractable domain theories. It has developed a new method for improving the tractability of such theories. The new technique employs a strategy of sacrificing the accuracy of an intractable theory in order to gain efficiency in return. Initially intractable theories are approximated by adopting simplifying assumptions. Such assumptions are useful because they greatly shorten the otherwise intractable inference process. Unfortunately, the assumptions are not strictly true. The resulting theory may therefore be less accurate than the original. One may nevertheless take an *optimistic* attitude toward such difficulties. The errors may be small or infrequent. When errors do occur, they may be justified by the improvement in computational efficiency.

The new learning technique has been implemented in a program called *"POLLYANNA"*. This system combines analytic and empirical methods in a generate and test framework. During the analytic, generation phase, candidate approximations are formed by systematically applying *generic simplifying assumptions* to the initial intractable theory. In the empirical, testing phase, approximate theories are evaluated against teacher-provided training examples. Measurements of accuracy and efficiency are used to guide a search for approximate theories meeting specified accuracy and efficiency goals.

POLLYANNA is capable of learning an entire spectrum of approximate theories. At one end of the spectrum, the system learns naive theories that are efficient but highly prone to errors. At the other end, it learns more sophisticated theories that are accurate but computationally expensive. Theories at intermediate points are created as well. Depending on the context in which a computational theory is used, different combinations of accuracy and efficiency are appropriate. This research thus demonstrates a highly flexible approach to the intractable theory problem. *POLLYANNA* was developed using the card game hearts and a job scheduling problem as test bed domains.

1.1.3. Human Learning and Intractable Theories

This research has been motivated by a belief that a machine learning approach can deal with the problem of computational intractability. An interesting question arises when intractability is viewed as a learning problem. The question can be illustrated by the following thought experiment: Imagine two students, *A* and *B*, who are learning to play the card game hearts by observing the actions of a teacher who is actually playing the game.[1] Both students are told the rules that determine which cards are legal in a given game situation. Only student *A* is told that the game objective is to *minimize* one's final game score. Student B knows nothing about the objective of the game. Which student is likely to learn faster? One naturally expects student *A* to learn faster, since he knows the goal of the game, while student *B* does not.

The thought experiment illustrates the value of background knowledge in the course of learning from examples. An initial domain theory appears to facilitate the process of learning examples. The two imaginary students differ only in their initial knowledge of the game. If student *A* learns more quickly, the credit must be given

---

[1]The hearts domain will be used as a source of examples throughout this dissertation. The reader who is not familiar with hearts should take a moment to learn about the game. Hearts is normally played with four players. Each player is dealt thirteen cards. At the start of the game, one player is designated to be the leader. The game is divided into thirteen successive tricks. At the start of each trick, the leader plays a card. Then the other players play cards in order going clockwise around the circle. Each player must play a card matching the suit of the card played by the leader, if he has such a card in his hand. Otherwise, he may play any card. The player who plays the highest rank card in the same suit as the leader's card will win the trick and become the leader for the next trick. The winner of the trick receives points associated with the cards played in the trick. In the simplest version of the game, each heart card has a point value of one, and the queen of spades has a point value of thirteen. All other cards have zero point value. The game objective is to minimize the number of points in one's score. In a more advanced version, players can "shoot the moon". Whenever one player takes all hearts and the queen of spades, his score is zero and his opponents get twenty-six points each. The option of shooting the moon is absent from the version of hearts used in this research.

to his superior initial theory of the game. A problem arises if one asks how student *A* exploits his superior knowledge. He presumably attempts to interpret the teacher's actions in terms how they contribute to the objective of the game. Nevertheless he certainly cannot *prove* that each of the teacher's choices is optimal. Although such proofs are possible in principle given knowledge of the rules and goals of the hearts game, they are not feasible in practice. They would require an exhaustive analysis of the hearts game space - a process that is computationally intractable. Thus student *A* must utilize his knowledge of hearts in some other manner. Although the experiment clearly illustrates the value of an initial intractable theory, it raises a question regarding the manner in which the theory is actually used to help a student learn from examples.

1.1.4. Heuristics and Intractable Theories

AI literature has traditionally recommended using domain specific *heuristics* to deal with intractability. Although no single definition of "heuristic" has been generally accepted, one author defines heuristics to be a "rule of thumb" for deciding among several alternative courses of action [Pearl 84]. Heuristics are not guaranteed to be correct; however, they are easy to apply and lead to results that are nearly or frequently correct. Heuristics thus represent a compromise between the requirements of efficiency and accuracy. Heuristics have long been studied by investigators interested in problem solving techniques useful for humans and machines [Polya 57; Newell and Simon 72].

In the context of machine learning, it is natural to ask how heuristics can be learned. For domains in which an intractable theory has been formulated, the intractable theory itself presents an interesting possible source of heuristics. Some investigators have suggested that heuristics can be obtained by formulating simplified versions of the original theory. For example, several different heuristics for the N-puzzle[2] can each be obtained by forming distinct simplified versions of the N-puzzle problem space [Gaschnig 79; Pearl 84; Mostow and Prieditis 89]. A potential problem arises when a given intractable theory can be transformed into many different simplified theories, as occurs in the case of the N-puzzle. Each simplified theory may lead to a different heuristic. The different heuristics may offer conflicting advice. Some additional information will then be needed to select among the candidate heuristics. When training examples are available, the candidates might be evaluated according to their consistency with the training examples.

[2]The *15-Puzzle* is a game played by moving *15* square tiles around a *4* by *4* rectangular grid in order to reach a specified goal configuration.

1.1.5. Dimensions of Machine Learning

POLLYANNA uses a combination of two distinct criteria for evaluating the behavior of computational theories. One criterion is efficiency. Efficiency is an issue right from the outset since the initial domain theory is intractable. A second criterion is accuracy. Accuracy becomes an issue as a result of approximations used to improve efficiency. Most previous machine learning techniques focus on either accuracy or efficiency, but not both. In order to achieve learning objectives involving both criteria, *POLLYANNA* uses an integrated combination of two types of machine learning techniques.

Empirical learning research has been concerned primarily with the *accuracy* of computational theories [Winston 72; Michalski 80; Lebowitz 83; Angluin and Smith 83; Cohen and Feigenbaum 82; Michalski 83; Michalski et al. 83; Mitchell 82]. This research has developed systems that are initially endowed with little or no background knowledge of the application domain. Such systems rely mainly on training examples to formulate hypotheses about the domain. The hypotheses are evaluated in terms of the accuracy with which they can explain or predict examples. *Analytic* learning research has focused more on the *efficiency* of computational theories [Mitchell et al. 86; Ellman 89a; Minton 88; Laird et al. 86]. This research has developed systems that are initially supplied with an accurate but inefficient theory of the application domain. Learning methods are therefore directed at improving the efficiency of such theories.

Empirical and analytic learning are typically understood in terms of distinct models of learning. Empirical learning can usually be characterized at the *knowledge level* [Dietterich 86]. The knowledge level is an abstraction that can be used to define the knowledge of a system [Newell 82]. Viewed from the knowledge level, a system is presumed to know all the facts contained in the deductive closure of its knowledge base. Considerations regarding the computational cost of drawing inferences are ignored. Efficiency cannot be analyzed from the knowledge level. When learning systems are understood in this way, efficiency is ignored, and accuracy becomes of paramount importance.

Analytic learning systems are usually understood in terms of the *operationality* of the computational theories they produce. No single definition of "operationality" has been accepted by the machine learning community, although several have been suggested [Mostow 81; Mostow 83; Keller 88; DeJong and Mooney 86; Minton 88; Segre 88; Dietterich and Bennett 88]. Using a simple definition, operationality measures the degree to which a theory is (a) usable (executable, compilable, interpretable, etc.) on a real machine and (b) efficient in its use of time and space resources. In any case, the *truth* of a theory is not usually considered relevant to its degree of operationality. When learning systems are understood in terms of operationality, accuracy is ignored and efficiency becomes more important. The typical features of analytic and empirical learning are summarized in Figure 1-1.

	EMPIRICAL LEARNING	ANALYTIC LEARNING
BACKGROUND KNOWLEDGE:	Little	Much
TRAINING EXAMPLES:	Many	Few
EVALUATION CRITERION:	Accuracy	Efficiency
LEARNING MODEL:	Knowledge Level	Operationality

Figure 1-1: Empirical v. Analytic Learning

POLLYANNA is an integrated learning system. It manifests features of both analytic and empirical learning. The system is analytic in its reliance on considerable amounts of background knowledge. The initial intractable theory is one type of background knowledge used in *POLLYANNA*. The system is also empirical in its use of large numbers of training examples. The behavior of *POLLYANNA* can be analyzed using both of the two learning models described above. Learning may be characterized in terms of improved operationality and learning at the knowledge level.

1.1.6. Questions Addressed by this Research

The two principal questions addressed by this research are shown in Figure 1-2. The first question is framed in terms of empirical learning and is illustrated by the thought experiment described above. It asks how an intractable theory can be used to guide a student in the course of learning from examples. This question is answered by showing how an intractable theory can constrain the set of hypotheses to be considered in the course of empirical learning. The second question is framed in terms of analytic learning. It asks how heuristics can be extracted from a computationally intractable theory. This question is answered by finding operations that can be applied to intractable theories to produce useful heuristics.

- *How can an intractable theory help a student to learn from examples?*
- *How can heuristics be extracted from an intractable theory?*

Figure 1-2: Questions Addressed by this Research

1.2. The Intractable Theory Problem

The terms "theory" and "intractable" are each used in contexts inside and outside of the machine learning field. The term "theory" has become popular in the machine learning community ever since it was used in [Mitchell et al. 86]. It also has a technical meaning in the field of mathematical logic and is used widely in colloquial language. The term "intractable" has a technical meaning in the field of computational complexity. These terms therefore carry considerable semantic baggage that can lead to confusion in the course of discussing "intractable theory problem". To avoid confusion, an effort will be made to clarify the use of these and other terms in this dissertation.

1.2.1. Computational Theories

In the present context, the term "theory" is intended to refer to any computationally effective knowledge base. A knowledge base is said to be "computationally effective" for a given class of problems if there exists some effective procedure for using the knowledge base to solve the class of problems. A theory may be viewed as an algorithm represented in a declarative manner. The algorithm is represented by two components: (a) a knowledge base (theory) and (b) an inference engine (effective procedure). The notion of a "computational theory" is used here instead of the more standard "algorithm" for two reasons. To begin with, machine learning methods are generally more suited to declarative than procedural knowledge representations. The techniques to be presented in this dissertation are no exception. Furthermore, the distinction between knowledge base and inference engine will prove to be important in clarifying various types of intractable theory problems. The term "theory" will be used interchangeably with the term "computational theory".

1.2.2. Completeness and Consistency

Definitions of completeness and consistency will be useful both for defining the intractable theory problem and for discussing methods of attacking it. These terms can be defined in the following way for theories that describe functions: Let T be a theory that defines a function F mapping a domain D into range R. Theory T is said to be *complete* provided it entails some statement of the form $y=F(x)$ for each x in the domain D, i.e., it assigns *at least* one value to each member of the domain. A theory T is said to be *consistent* if it entails no statement of the form $\{y_1=F(x)\}\wedge\{y_2=F(x)\}$ for any x in domain D unless $y_1=y_2$, i.e., it assigns *at most* one value to each member of the domain. A theory can be incomplete in two distinct ways: (1) A theory suffers from limited *scope* if it fails to entail any statement of the form $y=F(x)$ for one or more values of x in the domain D, i.e., the value of $F(x)$ is not specified for some member of the domain. (2) A theory suffers from limited *precision* if it entails only statements of the form $\{y_1=F(x)\}\vee...\vee\{y_n=F(x)\}$, where $n>1$ and each y_i is distinct, for one or more values of x in the domain D, i.e., the value of $F(x)$ is only partially specified for some member of the domain. These definitions can be extended to theories that define concepts by taking $F(x)$ to be a boolean concept membership function.

1.2.3. Approximations, Approximate Theories and Heuristics

The terms "approximation" and "approximate theory" will be used repeatedly in this dissertation. The term "approximate theory" will be used to describe the result of transforming an initially correct theory by introducing one or more "approximations". Individual approximations differ from full approximate theories. An individual approximation will not necessarily be a complete theory. Incompleteness will often occur because the approximation only describes solutions to subproblems, without giving solutions to top level problems. In contrast to this, an approximate theory is complete. It specifies precise solutions to all problems drawn from the domain under study.

The term "heuristic" will also be used repeatedly in this dissertation. Much of this research is directed at showing that many approximate theories are semantically equivalent to intuitively plausible heuristics. Despite the equivalence, the terms "heuristic" and "approximate theory" will be used differently. The term "heuristic" will refer to rules of thumb that are described using informal terms. The term "approximate theory" refers to a formal representation. A single heuristic also need not be as complete as a full approximate theory. For example, it may apply only to some problems drawn from the domain under study.

1.2.4. A Pragmatic Notion of Intractability

This research is concerned with a pragmatic notion of intractability. A computational theory is considered to be "tractable" or "intractable" relative to a specific performance *context* in which the theory is used [Keller 87]. The context is characterized by (a) specified computational resource constraints and (b) a set of problems to be solved using the theory. Typical resources include time and space consumed during computations. These quantities are assumed to be measured in absolute terms, e.g., CPU seconds and bytes. Resource constraints could be formulated in terms of either worst case or average case behavior. A theory is then said to be tractable provided it can solve the specified problems without exceeding the absolute resource limits. Otherwise the theory is not tractable.

The pragmatic definition of intractability can be illustrated with two examples. Consider the case of a computational theory used to monitor radar data as part of an air traffic control system. The theory might be used to detect impending collisions between aircraft. In such a performance context, the computation would be required to meet some real time constraints. The theory might or might not be intractable, depending on the specific time constraints that are relevant. As a second example, consider a computational theory of chess that operates by exhaustively searching the entire game tree. There does exist some finite time period sufficient for carrying out such a computation; however, the computation cannot be performed within any resource limitations likely to be available in real life. The theory is therefore intractable for all realistic performance contexts.

The term "pragmatic" is used to distinguish this idea from the standard complexity-theoretic notion of tractable and intractable algorithms. The standard definition assumes an infinite family of problems associated with a growth parameter, N, that measures the size of problems in the family. Furthermore, the standard definition concerns the asymptotic behavior of the functions $T(N)$ and $S(N)$ that measure time and space resources consumed by the algorithm as a function of the problem size N. An algorithm is considered tractable if and only if $T(N)$ is $O(P(N))$ for some polynomial $P(N)$. Since $S(N)$ is necessarily $O(T(N))$, this implies that $S(N)$ is $O(P(N))$ as well.

The pragmatic notion differs in two principal respects. Instead of considering an infinite family of problems with unbounded sizes, the pragmatic notion considers only a finite set *P* of specific problems, all of which may in fact be the same size. Instead of examining the *asymptotic* dependence of time and space on a size parameter, the pragmatic notion simply asks whether each problem from the set *P* can be solved within *absolute* time and space limits. Asymptotic complexity describes the behavior of algorithms only in the limiting case, as the problem size parameter *N* goes to infinity. When learning is aimed at a specific performance context, the pragmatic notion is more relevant than the complexity theoretic one.[3] Notice also that this research is concerned with intractable *theories* not intractable *problems*. Thus the intractable theory problem corresponds to the standard notion of *algorithm* complexity. It does not address the deeper notion of *problem* complexity, i.e., the complexity of an optimal algorithm for the problem. Although the obvious computational theories of hearts and chess are intractable, there may exist other, highly efficient theories that solve the same hearts or chess *problems*.

1.2.5. Problem Specification

A specification of the intractable theory problem *(ITP)* is presented in Figure 1-3. The problem takes three inputs: an intractable theory *T*, a training set *E* and a set of accuracy and/or efficiency goals *G*. The learning system must find a simplified, approximate theory *T'*, with the same scope and precision as the original theory *T*, that meets specified accuracy and efficiency goals.

Given: **a. An intractable theory *T*.**
b. A set of training examples *E*.
c. Accuracy and efficiency goals *G*.

Find: **An approximate theory *T'*, with the same scope and precision as the initial theory *T*, that satisfies the accuracy and efficiency goals *G*.**

Figure 1-3: The Intractable Theory Problem

The initial theory is assumed to describe some function *F* that maps specific problems *p* to solutions *F(p)*. Each training example is a pair of the form *(p,F(p))*.[4] The training example set *E* is intended to characterize the types of problems typically encountered in practice. During the performance phase, after learning is complete, the computational theory will be used to solve problems drawn from some distribution *D*. The distribution describes the rates at which different types of problems are encountered in practice. The training example set *E* is drawn from the same distribution *D*.

[3] As a speaker unknown to the author once said: "We don't live in asymptopia.".

[4] The requirement that *T* describe a function *F(p)* is introduced merely to enable making a distinction between problem instances and the theory that solves problem instances. The theory need not be implemented as a function.

The accuracy and efficiency goals can be formulated in a variety of ways. (See Chapter 4). One formulation would require finding an approximate theory *T’* meeting specified average case accuracy and efficiency thresholds. Another formulation would ask for a set of theories with Pareto optimal values of average case accuracy and efficiency. A theory is considered to be "Pareto optimal" if no other theory manifests superior levels of both accuracy and efficiency. In either formulation, the learning goals would specify that average accuracy and efficiency levels be achieved relative to the distribution *D* of problems encountered during the performance phase. These values can not be directly measured during the learning phase, since the system does not have access to the underlying problem distribution *D*. Since the example set *E* is drawn from the same distribution *D*, the average values can be *estimated* by testing theory *T’* against the training set *E*.[5]

A final constraint requires the goal theory *T’* to have the same scope and precision as the initial theory *T*. If theory *T* is a complete description of a function *F* mapping a set *P* of problems into a set *S* of solutions, $F{:}P \rightarrow S$, then theory *T’* must also completely describe a function *F’* mapping *P* into *S* as well, $F'{:}P \rightarrow S$. Thus theories *T* and *T’* both solve the same class *P* of problems. For each problem they both return a single result contained in the set *S* of well formed solutions.

In the absence of the scope and precision requirements, a rote learning system might be used to solve the intractable theory problem. It would simply memorize the training set *E* and construct a simplified theory *T’* that operates by table lookup. Given a test problem *p*, the theory *T’* checks whether a pair of the form *(p,F(p))* is found in the table. As a result of the scope and precision requirements, the rote learning approach is not sufficient. The learning system is required to construct a theory *T’* that returns some value of *F(p)* for each problem *p* that it has not seen before. It must therefore make inductive leaps that go beyond the observed training examples.

1.2.6. An Example Intractable Theory Problem

In order to make this definition more concrete, consider the card game hearts as an example. A computational theory of hearts can be formulated in terms of a choice function *Choice(situation)* that takes a description of a game *situation* as input. The situation might be represented as a record of the game history up to the current game state, along with the player’s current hand. This function would return an optimal card choice for the current situation. Such a theory would be intractable due to the need to consider all the ways the cards might be dealt, along with many combinations of players’ future card choices. Training examples might be represented in terms of

[5] An alternate approach would have a teacher provide examples drawn from a special distribution constructed for pedagogical purposes. The learning system would presumably then use the examples in some manner other than simply estimating average values of accuracy and efficiency.

(situation,action) pairs that describe the behavior of a human expert. Efficiency and accuracy goals might be defined respectively in terms of CPU time and the average rate at which an approximate theory makes the same choice as the teacher.

1.2.7. Trading Accuracy for Efficiency

The *ITP* specification in Figure 1-3 makes accuracy and efficiency levels into explicit parameters to the intractable theory problem. This formulation allows for a particular approach to dealing with intractability. Theories can be made tractable by sacrificing accuracy in order to gain efficiency in return. By relaxing the requirement of absolute accuracy, this formulation enables the resulting theory to have greater efficiency than would otherwise be possible. The accuracy goals indicate the amount of error that will be tolerated in order to improve efficiency. The efficiency goals indicate the improvement that is needed. The *ITP* formulation thus implicitly contains the idea of a tradeoff between accuracy and efficiency.

1.2.8. Learning in Context

A solution to the *ITP* requires tailoring the initial theory *T* to suit a specific performance context. Context enters into the *ITP* in two ways. The context is described in part by the training set *E*, which characterizes the distribution of problems to be encountered in practice. The example problems in the training set *E* may not require the full complexity of the initial intractable theory *T*. In such cases it may be possible for a simpler theory *T*' to solve most typical problems accurately. The theory *T'* can omit those features of the initial theory *T* that contribute to intractability, but are rarely important in the target context. Although the theories *T* and *T'* might differ on many of the problems that are theoretically possible, they will nevertheless yield similar or identical results for most typical problems.

The performance context is further characterized by the accuracy and efficiency goals. Depending on the requirements of the target context, varying levels of accuracy and efficiency will be required. In some situations accuracy will be a priority and efficiency will be only a secondary concern. For example, a medical diagnosis system might place greater emphasis on accuracy than efficiency. In other situations the context may dictate tight real time and/or space constraints, leaving accuracy of secondary importance. For example, a chess playing program might be required to choose a move before the chess clock runs out. Accuracy would be a secondary concern since failure to meet the deadline means losing the game. The accuracy and efficiency goals specify the relative importance of these two measures. They indicate how the simplified theory *T'* should strike a balance these competing concerns.

1.2.9. Dual Views of the Intractable Theory Problem

The intractable theory problem can be interpreted from two distinct points of view. The first interprets it as an empirical problem of learning from examples. The training examples are seen as the primary input. The learning system is required to generate a goal theory *T'* that predicts the examples with some specified degree of accuracy. Since the goal theory must have the same scope as the initial theory *T*, the system must make hypotheses that go beyond the observed examples. Inductive bias is required to choose between hypotheses that are equally consistent with the observed examples. The initial intractable theory *T* is seen as a source of the required bias. The efficiency goals are seen as an additional constraint.

The second viewpoint interprets the *ITP* as an analytic learning problem. The initial intractable theory *T* is seen as the primary input. The learning system must transform *T* into a simpler theory *T'*. A given initial theory *T* may be simplified in a variety of ways, depending on the simplification methods available to the learning system. Some method is required to select among alternative simplifications. Additional constraints result from requiring the simplified theory *T'* to meet the specified accuracy goals.

	ANALYTIC VIEW	EMPIRICAL VIEW
PRIMARY INPUT:	Initial Theory	Training Examples
PRIMARY GOAL:	Efficiency	Accuracy
BIAS/CONSTRAINTS:	Examples / Accuracy	Initial Theory / Efficiency

Figure 1-4: Two Views of the Intractable Theory Problem

1.3. The Proposed Solution

The intractable theory problem requires some method of transforming an initially correct domain theory *T* into a simpler theory *T'*. The problem definition does not require this transformation to preserve the truth of the initial theory. Theory *T'* may be only an approximation to *T*. By allowing transformations to sacrifice accuracy, the *ITP* definition opens up the possibility of greater gains in efficiency than can be attained through truth-preserving operations. A solution to the *ITP* must exploit this opportunity.

1.3.1. Generic Simplifying Assumptions (*GSAs*)

This research has developed a solution based on the concept of *generic simplifying assumptions (GSAs)*. Generic simplifying assumptions have three defining properties. (1) A *GSA* is a schema. It can be instantiated in a variety of ways, depending on the domain of application. For this reason, *GSAs* are considered to be *generic*. (2) When introduced into a computational theory, a *GSA* has the effect of *simplifying* the process of making inferences. Problems can therefore be solved more efficiently than before. (3) Each *GSA* is an *assumption*, i.e., it is not necessarily true.

Examples of generic simplifying assumptions are defined informally in Figure 1-5. Notice that each is described in a domain independent manner. Each refers to domain independent concepts such as functions and random variables. These terms can be instantiated in many different ways, depending on the application domain. Notice also that they are not necessarily true. For example, the *GSA* entitled "Function Invariance" (*FI*) treats a function *F(x)* as a constant, despite the fact that *F(x)* may actually depend on the value of *x*. Finally, notice that each *GSA* simplifies the process of carrying out computations. For example, the *FI* assumption avoids the need to actually compute *F(x)*. The simplifying effect of the others is real, but perhaps less obvious. The generic assumptions described in Figure 1-5 are verbal characterizations of *GSAs* defined formally in Chapter 2. They are all drawn from a family of generic simplifying assumptions called "the *PT-GSA* family". This family of *GSAs* will be presented in Chapter 2.

- *Function Invariance (FI):* Given a function *F(x)* that is expensive to compute, assume that *F(x)* equals some constant *C* for all values of *x*.
- *Equiprobable Random Variables (EP):* Given a random variable *v* with range *R*, assume that *v* is equally likely to manifest any value *R*.
- *Probabilistic Independence (IN):* Given two random variables *v* and *w*, assume that they are probabilistically independent.

Figure 1-5: Generic Simplifying Assumptions

Generic simplifying assumptions can be used to implement a tradeoff between accuracy and efficiency. Although *GSAs* are not strictly true in general, they may nevertheless be nearly true for most problems encountered in the intended performance context. Even when false they may not critically impact the results of a computation. *GSAs* allow portions of the initial theory to be preserved intact, while others are approximated. For example, the *FI* assumption in Figure 1-5 approximates only a single function *F(x)* with a constant and leaves other functions unchanged. This approximation would likely be useful if the value of *F(x)* does not vary greatly, or if the final result of the computation is not particularly sensitive to the value of *F(x)*. When used to simplify the non-critical parts of an intractable theory, *GSAs* can potentially achieve dramatic gains in efficiency while maintaining acceptable levels of accuracy.

1.3.2. The *POLLYANNA* Architecture

The proposed solution to the intractable theory problem has been implemented in the *POLLYANNA* program. This program uses a combination of analytic and empirical learning techniques organized in a generate and test architecture. During the analytic (generation) phase, the system generates a set of candidate approximate theories. Approximate theories are generated by systematically applying generic simplifying assumptions to the initially intractable theory. During the empirical (testing) phase, candidate theories are tested against training examples to obtain measurements of accuracy and efficiency. The measurements are to select approximate theories that meet the specified accuracy and efficiency goals.

An architectural view of *POLLYANNA* is shown in Figure 1-6. The learning process begins with the *approximation generation (AG)* phase. This module generates candidate approximations by drawing on three distinct types of knowledge: (a) the initial intractable theory; (b) generic simplifying assumptions; (c) truth preserving reformulations. During the subsequent *theory space generation (TSG)* phase, the system combines individual approximations to build complete approximate theories. The approximate theories are also organized into a search space. The *TSG* module attempts to partially order approximate theories according to computational efficiency, so that efficiency will fall monotonically along paths in the space. For this purpose, the system draws upon knowledge about the impact of generic simplifying assumptions, in order to compare efficiency levels of different theories. In the final *theory space search (TSS)* phase, the system searches through the theory space to find approximate theories that meet specified accuracy and efficiency goals. The search is guided by empirical measurements. These measurements test both the accuracy with which candidate theories explain training examples, and the efficiency of the computation used to build the explanations. The *approximation generation* and *theory space generation* modules are "analytic" since they operate in the absence of training examples. In contrast to this, the *theory space search* module is "empirical" since it makes direct use of examples.

As the system is currently implemented, the three modules are invoked consecutively, as distinct phases of operation. There is no reason in principle for the phases to be separate. Instead of generating all the candidates in advance of any testing, generation and testing could be interleaved. Candidates could be generated as testing indicates they are needed. Thus the *theory space search* module would invoke the *theory space generation* module as new portions of the search space need to be generated. The *theory space search* module would invoke the *approximation generation* module as new individual approximations are needed. Under such an arrangement, data would flow left to right while control information would flow right to left in Figure 1-6.

POLLYANNA may be seen as a solution to the intractable theory problem by comparing the definition in

Figure 1-6: The *POLLYANNA* Architecture

Figure 1-3 to the architectural diagram in Figure 1-6. The givens of the *ITP* include the intractable theory, the training examples and the accuracy/efficiency goals. *POLLYANNA* takes these as explicit inputs. The *ITP* definition requires finding approximate theories meeting the specified accuracy and efficiency goals, and *POLLYANNA* produces these as explicit outputs. Notice that *POLLYANNA* also requires three types of information that are not listed under the "givens" of the intractable theory problem. These include: (a) generic simplifying assumptions, (b) reformulation knowledge and (c) assumption impact knowledge. These three databases are intended to be domain independent, permanent parts of the system. In order to support this view, considerable effort will be made to demonstrate the domain independence of these three types of information.[6]

POLLYANNA embodies an *optimistic* approach to the problem of intractability. The system is optimistic in two distinct ways. At the most general level, *POLLYANNA* takes an optimistic attitude toward generic simplifying assumptions, like the ones shown in Figure 1-5. Although these assumptions are not necessarily true, and are obviously false in some cases, *POLLYANNA* adopts them anyway. These assumptions serve to simplify expensive parts of computations. The simplified computation may not critically impact the accuracy of final answer. *POLLYANNA* thus optimistically expects the simplified theory to be nearly correct, most of the time. *POLLYANNA*

[6]In the current implementation, *POLLYANNA* uses one additional input not shown in Figure 1-6. The approximation generator takes input from an interactive user. A human user supplies control information to guide the process of generating assumptions. (See Chapter 2.)

also uses an optimistic search strategy during the empirical phase of learning. Taking advantage of the monotonic organization of the theory space, *POLLYANNA* begins searching with the simplest, most efficient theory in the space. Less efficient theories are considered only after simpler ones are refuted by training examples. This strategy is a variant of Occam's Razor. *POLLYANNA* prefers to adopt the *simplest* hypothesis that is consistent with training examples. Simplicity is measured in terms of the efficiency of computational theories.

1.3.2.1. Learning as Search in *POLLYANNA*

POLLYANNA can be compared to the version space model for analyzing empirical learning as a search problem [Mitchell 82]. This model presupposes a search space of candidate hypotheses to be considered in the course of learning. Each hypothesis describes a different concept. The concepts are partially ordered according to the concept inclusion relation. (Concept *A* is said to include concept *B* if every instance of *B* is also an instance of *A*.) Given such a search space, a learning algorithm can conduct a search process, using training examples to guide the search. Many learning algorithms can be characterized in terms of the control strategies they use to search through the space of candidates.

Some features of *POLLYANNA* fit nicely into the version space model, while others depart from the model in significant ways. The principal similarity involves the empirical theory space search (*TSS*) component of *POLLYANNA*. It performs empirical learning by searching through a space of candidate hypotheses - approximate theories - using training examples to guide the search. The principal differences involve the organizational structure of *POLLYANNA*'s theory space, and the manner in which the space is generated by the approximation generation (*AG*) and theory space generation (*TSG*) modules. The candidate hypotheses in *POLLYANNA* need not represent concept descriptions. Approximate theories may represent concepts, functions or other objects. *POLLYANNA* attempts to partially order candidates according to efficiency rather than concept inclusion, so that efficiency will fall monotonically along paths through the space. Computational efficiency does not enter at all into the version space model. *POLLYANNA* also constructs the space of candidates, by applying generic simplifying assumptions to an initial domain theory. The version space model simply assumes the existence of a set of candidate hypotheses. It does not deal with the issue of generating a candidate hypothesis space from some body of background knowledge. These differences suggest a need to extend the version space model to encompass systems such as *POLLYANNA*.

Two other models of learning are also worth examining in relation to *POLLYANNA*. The probably approximately correct (*PAC*) model [Valiant 84] is used to analyze the information complexity of learning classes of concepts. *PAC* considers only the accuracy of learned concepts. Efficiency of concept descriptions is ignored by the *PAC* model. An approach to analyzing learning in the sense of improving efficiency is described in [Natarajan and Tadepalli 88]. This model considers only efficiency and ignores the accuracy of computational theories. No similar models have yet been proposed to analyze systems that learn by implementing a tradeoff between the dimensions of accuracy and efficiency.

1.3.2.2. Inductive Bias in *POLLYANNA*

The behavior of *POLLYANNA* can also be analyzed in terms of inductive bias. The concept of bias will be useful later, in the course of formulating claims about the role of empirical learning in *POLLYANNA*. The standard definition of bias is found in [Mitchell 80]. It defines an "unbiased hypothesis space" in the context of concept learning systems. If instances of the concept are drawn from some known set *I*, then the unbiased hypothesis space contains one concept for every subset of *I*. Any proper subset of the unbiased space is called a "biased hypothesis space". In comparing two hypothesis spaces, *H* and *H'*, the space *H* is said to have a stronger (weaker) bias if the cardinality of *H* is less than (greater than) the cardinality of *H'*.

The standard definition requires a modification before it can be applied to the intractable theory problem formulated in Figure 1-3. The standard definition concerns concept learning programs. In contrast to this, the intractable theory problem has been formulated as a function learning task. The standard definition must therefore be extended from concepts to functions. This extension is straightforward. Consider a candidate hypothesis space *H* that describes functions mapping problems in the set *P* to solutions in the set *S*. The space *H* is said to be "unbiased" if it contains every possible function that can be defined over the sets *P* and *S* of problems and solutions, i.e., $H=[S \rightarrow P]$. Any proper subset of *H* would then be called a "biased" hypothesis space. This reduces to the standard definition whenever *P* is a set *I* of instances, and *S* is the set $\{T,F\}$ of possible concept membership classifications.

In the context of *POLLYANNA*, the candidate hypothesis space *H* corresponds to the *theory space* produced jointly by the *AG* and *TSG* modules. Experimental results will show that these modules generate biased theory spaces. Only a proper subset of the unbiased space is generated. The theory space is also *dependent* on the three inputs to the *AG* module, the intractable theory (*IT*), the generic simplifying assumptions (*GSAs*) and the reformulation knowledge (*R*). The *GSAs* and *Rs* constitute a domain independent source of bias in *POLLYANNA*. The initial theory (*IT*) is a domain dependent source of inductive bias. When the initial theory is changed, the *AG* and *TSG* modules produce a different theory space. In this respect, *POLLYANNA* contrasts with traditional concept learning programs that use only a fixed candidate hypothesis space.

1.4. Research Methodology

1.4.1. Test Domains

POLLYANNA was developed mainly using the card game hearts [Gibson 74] as a test bed domain. (See Section 2.4.2.) Hearts was chosen as the primary domain for several reasons. To begin with, this domain is obviously intractable. (The precise cause of this intractability will be discussed in Chapter 2.) Despite this intractability, a variety of relatively simple heuristics for hearts are known to human card players [Andrews 83]. (Example heuristics will be presented in Chapter 2.) The simplicity of these heuristics contrasts dramatically with the complexity of exhaustively analyzing the game. This contrast naturally suggests investigating the relation between the simple heuristics and the intractable theory.

The hearts domain is also a good "fruit fly". It exhibits features that make it a relatively manageable subject of experimentation. As with most games, the rules are relatively easy to state. For this reason, an exact intractable domain theory can be constructed without great difficulty. (The choice of representation is nevertheless important, as discussed in Chapter 2.) Training examples are also readily available from human card players and hearts game playing programs. (See Chapter 4.)

Hearts also exhibits features that make it more realistic than many games. At any point in the game, each player has only incomplete information about the current game state. Part of the difficulty of hearts is a consequence of the need to deal with incomplete information. Incomplete information is a problem typically encountered in real life problem solving. As an example, consider the task of planning a trip on the New York City subway system. One can never predict precisely when a train will arrive, or how long it will take to reach its destination. Incomplete information thus makes hearts more like real life problem solving than would be the case using a domain with complete information. (Despite the incomplete information, the initial theory need not be incomplete, as shown in Chapter 2.)

A job scheduling domain was used as a secondary test bed domain. (See Section 2.4.3.) This domain was chosen for two main reasons. The scheduling problem is sufficiently similar in structure to hearts that the scheduling theory could be written in the formalism used in representing the hearts domain theory. The scheduling theory could therefore be approximated using the same set of generic simplifying assumptions as used in learning heuristics for hearts. Scheduling is rather unlike hearts in being a serious domain. The scheduling domain results therefore provide evidence of *POLLYANNA*'s value in domains of practical importance.

1.4.2. Key Implementation Tasks

Several distinct implementation tasks were involved in realizing *POLLYANNA* as a solution to the intractable theory problem. One key task involved the *approximation generation* process (Figure 1-6). In order to implement this component, it was necessary first to identify an appropriate set of generic simplifying assumptions. The set was required to be capable of deriving known hearts heuristics starting from an exact, intractable theory of the game. The *GSAs* in Figure 1-5 were not the first ones considered for the hearts domain. They are the end result of a long process of studying various formulations of domain theories, assumptions and heuristics. The family of *GSAs* resulting from this investigation has been given the name "*PT-GSAs*", i.e., probability theory generic simplifying assumptions. (See Section 2.5.) They represent an elaboration and formalization of the list in Figure 1-5.

An even greater difficulty involved developing a representation for *GSAs* and domain theories. The verbal formulation of the *GSAs* shown in Figure 1-5 is not sufficient for a machine to generate approximations. A complete implementation required finding a representation that would be operational for the task of generating approximations. The *GSA* representation is required to interface with the initial domain theory representation. The *GSAs* and domain theory must interact in a manner that actually results in more efficient computational theories. This investigation resulted in a representation for theories and assumptions called the "*PT-FORMALISM*", i.e., probability theory formalism. (See Section 2.4.1.)[7]

1.4.3. History and Status of the Implementation

The implementation process occurred in three main phases. In the first implementation, only the *theory space generation (TSG)* and *theory space search (TSS)* modules were built. In the absence of machine generated output from the *approximation generator (AG)* module, hand coded inputs to the *TSG* module were used instead. These consisted of sets of hand coded candidate approximations represented in terms of generic functions, i.e., functions that are associated with multiple definitions [Cardelli 85; Stefik and Bobrow 86]. (See Section 2.3.5). The *TSG* and *TSS* module were then run together to collect accuracy and efficiency statistics, as reported in [Ellman 88].

The second round of implementation involved creating a *mechanical* version of the *approximation generator (AG)*. This module served to generate the type of generic functions used as input to the *TSG* module. The *AG* system is considered "mechanical" but not "automatic" because it uses some guidance from a human being. The mechanical *AG* system was used to generate approximations for the hearts domain, as reported in [Ellman 89b]. The results were run through the *TSG* and *TSS* systems to collect accuracy and efficiency statistics, as reported in [Ellman 89c].

[7]The term "formalism" is used here to refer to a language for representing a theory. The term "representation" will be used to describe a particular encoding of a domain theory in terms of some formalism.

The final round of implementation involved encoding a domain theory for the scheduling domain. The previously implemented *AG* module was then used to generate approximations for the scheduling domain. Some of the resulting approximations were shown to be semantically equivalent to plausible scheduling heuristics. The scheduling approximations were not tested empirically in the *TSG* and *TSS* modules. This domain was implemented mainly to demonstrate the domain independence of the generic simplifying assumptions and truth-preserving reformulations used to generate approximate theories.

1.5. Related Research in Machine Learning

POLLYANNA can be compared to several other machine learning techniques for improving computational efficiency. One comparison involves Explanation-Based Generalization (*EBG*) [Mitchell et al. 86] and similar systems. A conceptual framework will be developed to describe the types of intractable theory problems which can and cannot be handled by *EBG*. An important class of intractable theory problems will be seen to lie outside the scope of *EBG* and similar systems. This class represents potential range of application for *POLLYANNA*. A second comparison involves other contemporary research on learning approximations to improve the efficiency of intractable theories. *POLLYANNA* will be compared to such efforts along several dimensions. These comparisons will serve to illustrate the range of approaches to approximate theory formation that are currently under investigation.

1.5.1. EBG and the Intractable Theory Problem

EBG attempts to improve efficiency by solving and analyzing the solutions to example problems. *EBG* learns from training examples in a two step process. In the first step, it finds an operator sequence representing a valid solution to the example problem. In the second step, the sequence is compiled into a single operator or "chunk" that has the same effect as the original solution sequence. When *EBG* is applied selectively, it can produce compiled operators that improve the efficiency of a problem solver [Minton 88]. A number of other systems use learning techniques that are essentially equivalent to *EBG*. These include *Soar* [Laird, et al. 87], *Prodigy* [Minton 88], *Genesis* [DeJong and Mooney 86], *STRIPS* [Fikes et al. 72] and *LEX-II* [Mitchell 83]. (The similarities among these systems are discussed in [Ellman 89a].) One important variation allows solutions to be provided by a human teacher, rather than from internal problem solving [DeJong and Mooney 86]. In order to operate successfully, these systems all require operator sequences representing the solutions to sample problems, regardless of the source of such solutions.

The intractable theory problem was originally defined in the context of *EBG* [Mitchell et al. 86]. The authors correctly observed that *EBG* fails to operate when training example problems cannot be solved using limited computational resources. This observation left an unfortunate impression that the "intractable theory problem" properly belongs to the field of explanation-based learning. This dissertation takes a rather different view. Intractability is hardly a problem confined to explanation-based learning. It occurs throughout the field of computer science. It is ironic that the intractable theory problem was "discovered" by people working on explanation-based learning. Considering that *EBG* improves *only* the efficiency of computational theories, one would have expected intractability to have been a prime concern from the outset.

1.5.1.1. Preserving Truth v. Trading Accuracy for Efficiency

EBG is a truth-preserving process [Dietterich 86]. The correctness of each compiled operator is a logical consequence of the correctness of the initial domain theory. *EBG* systems attempt to improve efficiency without any sacrifice in accuracy. This approach does not fully exploit the possibilities offered by the *ITP* formulation in Figure 1-3. The *ITP* definition allows application of non truth-preserving operations to the initial domain theory. It allows accuracy to be sacrificed in order to gain efficiency in return. These observations suggest that *EBG* is limited in two respects. By strictly preserving accuracy, *EBG* may be unable to improve efficiency as much as alternative methods that are not so restricted. When faced with an intractable theory that cannot be made suitably efficient without introducing errors, *EBG* fails entirely.

1.5.1.2. Deterministic and Non-Deterministic Intractability

Theories can be intractable for two distinct reasons. One type of intractability occurs when problems are solved by searching in a large space of potential solutions. Although each potential solution is small and easily tested against a search goal, the overall process can be intractable if the space of candidate solutions is too large. Thus given a pair $(p,F(p))$, the task verifying the value of $F(p)$ may be tractable even when the task of finding $F(p)$ given p alone is not tractable. A more difficult situation occurs when the task of verifying solutions is also intractable. In this case both the task of verifying problem/solution pairs, $(p,F(p))$, and computing solutions $F(p)$ from p, are intractable.

The two types of intractability can be defined in terms of the difference between deterministic and non-deterministic computations [Garey and Johnson 79]. A theory shall be designated "deterministically intractable" when the task of computing $F(p)$ from p is intractable. This sort of theory cannot efficiently solve problems when the inference engine runs on a normal deterministic machine. It nevertheless might solve problems efficiently when the inference engine runs on a non-deterministic machine. If the task of verifying a problem/solution pair $(p,F(P))$ is tractable, then a non-deterministic machine can "guess" the correct solution and then (deterministically) verify the guess. Thus a theory can be deterministically intractable, but non-deterministically tractable. A theory shall be designated "non-deterministically intractable" when both the task of computing *F(p)* and the task of verifying *(p,F(p))* are intractable. This sort of theory cannot be used efficiently on either a normal or non-deterministic machine. These definitions are summarized in Figure 1-7.[8]

Deterministic:	**Intractable to Compute:** *F(p)* **from** *p*
Non-Deterministic:	**Intractable to Verify:** *(p,F(p))*

Figure 1-7: Types of Intractability

Consider the N-puzzle as an example. Suppose the solution requires finding any *legal* operator sequence mapping the initial state to the goal state. This problem is deterministically intractable. An acceptable solution is hard to find; however, the correctness of a proposed solution is easily checked. Suppose instead that the solution requires finding an *optimal* operator sequence mapping the initial state to the goal state. This version is non-deterministically intractable since the task of verifying the optimality of a proposed operator sequence is difficult. In order to verify optimality, one must consider a large space of alternative operator sequences.

The distinction between deterministic and non-deterministic intractability is significant in the context of machine learning. In particular, both *EBG* and the similar systems described above are at least formally applicable to theories that are non-deterministically tractable, but deterministically intractable. A human teacher may serve as the "oracle" that makes guesses guiding a non-deterministic machine. The teacher would provide solutions to problems. The system would then prove the correctness of the teacher’s solution and perform explanation-based generalization on the proof. This arrangement does not operate in the case of non-deterministic intractability. When the process of verifying a candidate solution is itself intractable, the system will be unable to the construct the proof required for *EBG*.

1.5.1.3. Absolute and Relative Intractability

A rough distinction can be drawn between theories that are *absolutely* intractable and those that are *relatively* intractable. To make the difference clear, imagine a scenario in which a set of computational resources R_L is available to be used for the purpose of learning heuristics for an intractable theory. Imagine also that a different set R_P of resources will be available to solve problems during the performance period, after learning has been completed. One might be willing to expend considerably greater resources on each example problem during the

---

[8]The set of domain theories that are non-deterministically tractable corresponds roughly to the complexity class *NP*, except that the present context is concerned with pragmatic theory efficiency, not asymptotic problem complexity. The distinction deterministic and non-deterministic intractability corresponds roughly to the difference between "large search space" intractability and "small steps" intractability, presented in [Rajamoney and DeJong 87].

learning phase, with the intention of amortizing the cost of learning over a long performance phase. Suppose therefore that R_L is greater than R_P. A theory is said to be "relatively intractable" if it can successfully solve problems during the learning phase with resources R_L, but cannot solve problems during the performance phase with resources R_P. A theory is said to be "absolutely intractable" if it cannot solve problems during either the learning or performance phases, using either resources R_L or R_P.[9]

As an example of a relatively intractable theory, consider the air traffic control system described above. Although an actual performance program would face tight real time constraints (R_P), the learning phase may occur with much weaker resource constraints (R_L). Learning could occur off line using simulated problem data. Real time constraints could be relaxed during off-line learning. Tight real time constraints would be imposed later during the performance phase. In contrast to this, the problem of solving mid game positions in chess is absolutely intractable, for all reasonable choices of R_L and R_P.

The distinction between absolute and relative intractability is useful for several reasons. To begin with, both *EBG* and the similar systems described above may be applicable to theories that are relatively but not absolutely intractable. In some situations *EBG* might be performed without exceeding the resources R_L allocated to the learning phase. It could even result in chunks that improve efficiency enough to solve problems without exceeding the resources R_P allocated to the performance phase. *EBG* is not applicable to theories that are absolutely intractable.[10]

The relative/absolute distinction is interesting for another reason. When the initial domain theory is only relatively intractable, teacher-provided training examples may not be needed. During the learning phase, training examples could be generated by using the initial (relatively intractable) theory, without exceeding the resources R_L allocated to the learning phase. To be precise, the initial theory can generate the solution *F(p)* of each typical problem *p*; however, a collection of typical problems must still be supplied to the system. By processing these training examples, a learning system could produce heuristics that would then be used during the performance phase, when the tighter resource bound R_P prohibits using the initial theory. Relative intractability allows for machine generation of examples regardless of the actual learning technique, i.e., whether it be *EBG* or something else.

In cases of absolute intractability, the learning system must rely on human teachers to supply both the problem *p* and the solution *F(p)* components of the training examples. Humans themselves cannot compute exact solutions

---

[9]Strictly speaking the terms "relative" and "absolute" should be applied to *(theory,context)* pairs.

[10]Except in the situation described above, when the theory suffers only from deterministic intractability and the teacher is available to make non-deterministic choices.

using an absolutely intractable theory. They presumably rely on their experience to generate solutions to example problems. The human generated examples are therefore not guaranteed to be consistent with the intractable domain theory. Erroneous examples are not a fatal problem in the context of approximate theory formation. Approximate theories are not expected to be entirely consistent with either correct or erroneous training examples. Systems for learning approximations are therefore likely to tolerate noisy training information.

1.5.1.4. The Scope of EBG

Explanation-based generalization is applicable to domain theories that are deterministically intractable (but non-deterministically tractable) or relatively intractable (but not absolutely intractable). Of the four combinations {deterministic, non-deterministic} *x* {relative, absolute}, *EBG* fails to apply only when the theory suffers from absolute, non-deterministic intractability. The same application conditions apply to the systems mentioned above that use essentially the same techniques as *EBG*. This observation concerns only the question of whether *EBG* can produce some sort of learning results for a given intractable theory. As shown in [Minton 88], the *EBG* method does not always lead to improved efficiency whenever the method applies. *EBG* may fail to yield useful results when applied to any of the four combinations. For this reason, the approximation methods presented in this dissertation are potentially relevant to all four types of intractable theory.

1.5.2. Research on Learning Heuristics

Interesting comparisons can be drawn between the heuristic generation framework used in *POLLYANNA* and other techniques for generating heuristics. For example, several investigators have discussed techniques for generating heuristics from simplified domain models [Gaschnig 79; Guida and Somalvico 79; Pearl 84; Kibler 85]. They examine the problem of generating heuristic evaluation functions *H(s)* that estimate the distance *D(s)* from some state *s* to a goal state. Given an intractable search space *S*, they transform *S* to a simpler search space *S'* for which an easily computable and exactly correct function *D'* is known. For example, *S'* might be an edge-subgraph (edge-supergraph) of *S*, i.e., *S'* contains all the states of *S*, but only a subset (superset) of the edges in *S*. After constructing a simplified space *S'*, and evaluation function *D'*, the function *D'* is used as the heuristic *H* for the original space *S*. (If *S'* is an edge-supergraph, the function *D'* is an admissible evaluation function for the original space *S*.) *POLLYANNA* does something similar using generic simplifying assumptions. When *GSAs* and reformulations are applied to an intractable theory *T*, the result is a new, approximate theory *T'*. The approximate theory *T'* is then used to solve problems the exact theory was originally designed to handle. The original theory *T* in *POLLYANNA* corresponds to the original search space *S* used by Gaschnig and Pearl. The approximate theory *T'* corresponds to the simplified search space *S'*. All of these methods are conceptually related to backward error analysis [Ralston and Rabinowitz 78], a technique for analyzing the error of numerical approximation techniques.

Given some problem *P* for which an algorithm computes an approximation *A'* to the exact answer *A*, backward error analysis finds a perturbed problem *P'* for which *A'* is an exactly correct solution. The algorithm's error is then analyzed in terms of the difference between the original problem *P* and the perturbed problem *P'*.

1.5.3. Research on Learning Approximations

POLLYANNA is similar in spirit to other contemporary research on the intractable theory problem. Recent machine learning research in this area has been variously described in terms of learning "approximations" [Keller 87; Bennett 89; Chase et al. 89], "abstractions" [Doyle 88; Mostow and Prieditis 89; Tadepalli 89; Unruh and Rosenbloom 89; Knoblock 89; Mohan and Tong 89], "simplifications" [Chien 89], and "heuristics" [Mostow and Fawcett 87; Mostow and Prieditis 89]. Other efforts at handling intractable theories not using such terminology include [Gupta 87; Hammond et al. 88]. These investigations share several common features. Each deals with a computationally intractable domain. Since an exact theory of the domain cannot be used in practice, a more efficient theory is used instead. The efficient theory is called an "approximation" because it is not guaranteed to be completely correct. Each approximate theory is related to the exact intractable theory by a specified set of formal operations. In some cases the formal operations are described only in verbal terms, whereas in other cases, the operations are actually implemented.

Each of the above mentioned systems can be naturally characterized in terms of the specific formal operations that relate the approximate theory to the exact one. For each system one may ask what are the generic simplifying assumptions that are used explicitly or implicitly in the design of the system. One type of operation introduces an assumption that functions or expressions return values that are invariant with respect to their arguments. These include the *truify* and *falsify* operations [Keller 87], *function invariance* [Ellman 88] and *freeze* [Mostow and Fawcett 87]. Another type of operation causes a problem solver to ignore the truth value of certain predicates, operator preconditions or goal conditions [Doyle 88; Chase et al. 89; Knoblock 89; Unruh and Rosenbloom 89; Mostow and Prieditis 89]. Some plan learning systems assume that certain types of subgoals do not interfere with each other [Gupta 87; Hammond et al. 88; Chien 89]. Other plan learning systems use assumptions of limited counter-planning by adversaries [Chien 87; Tadepalli 89]. Some systems use truth-preserving operations as well as generic simplifying assumptions in order to generate approximate theories [Ellman 89b; Mostow and Prieditis 89].

Systems for learning approximations can also be characterized in terms of the architectural features of the learning and performance components. Several learn by searching through a space in which each state corresponds to a different approximate theory [Keller 87; Ellman 88; Mostow and Fawcett 87; Mostow and Prieditis 89]. Systems that use empirical information to guide the process of selecting approximations include [Keller 87; Ellman

88; Chase et al. 89]. Failure driven refinement has been used as the basic learning method in several systems for learning approximations [Gupta 87; Hammond et al. 88; Chien 89]. Using this approach, a simple highly approximate theory is used until a failure is generated. Analysis of the failure leads to revising or retracting approximations. Considerable research on the intractable theory problem comes out of the explanation-based learning tradition. These systems learn approximations mainly as a means to facilitate the process of forming chunks or schemata [Doyle 88; Hammond et al. 88; Chien 89; Unruh and Rosenbloom 89; Tadepalli 89; Bennett 89].

Methods of learning approximations can also be compared in terms of the impact of approximations on the overall quality of solutions obtained in the course of solving problems. Although each of the above mentioned systems involves some sort of approximation, not all approximations lead to the same types of errors. Approximation errors can be classified according to whether they occur in *local* or *global* computations. Local computations produce intermediate results that are used in the course of *global* computations that produce final results. Some systems make only local errors. Errors in local computations do not always impact the accuracy of the final results. This typically occurs in cases where the approximations are applied to search control decisions [Mostow and Prieditis 89; Chase et al. 89; Unruh and Rosenbloom 89] or when a hierarchy of approximations is used in a process of stepwise refinement [Knoblock 89; Mohan and Tong 89]. In these systems, local errors can slow the process of finding a solution. The final solution is nevertheless guaranteed to meet the goal criterion. It may also be optimal [Mostow and Prieditis 89]. Other systems make truly global approximations [Ellman 88; Tadepalli 89]. In these cases the final results produced by the approximate theory may well be in error.

1.6. Thesis of the Dissertation

1.6.1. Viability of the *POLLYANNA* Methodology

This research has been directed at establishing the viability of a particular solution to the intractable theory problem. The proposed solution, embodied in *POLLYANNA*, involves two key features. The most important feature concerns the method of generating approximations. Approximate theories are generated by systematically applying generic simplifying assumptions to an intractable theory. This generation process is not perfect. It produces both good and bad approximations. The system therefore requires a second key feature. It relies on empirical testing of approximations in order to separate the good ones from the bad ones. The principal claim of this dissertation asserts the viability of *POLLYANNA* as a solution to the intractable theory problem:

Claim #1: The intractable theory problem can be solved by analytically generating approximations from generic simplifying assumptions and empirically testing approximations against training examples.

The principal claim must be qualified in several respects. The proposed solution is not expected to solve all instances of the intractable theory problem. The approach will no doubt fail for some application domains, and some types of theory formalisms. Nevertheless, the approach is expected to succeed *in a non-trivial set of application domains and theory formalisms*. In addition, the *POLLYANNA* implementation is not claimed to be a complete solution. It may nevertheless be seen as a significant first step toward a complete solution. A later section illustrates the manner in which *POLLYANNA* may be seen as a first step, and will identify some of the hurdles that must be overcome to extend *POLLYANNA* into a more complete solution.

The following sections elaborate on the principal claim of the thesis. In so doing, they distinguish between claims and conjectures. Statements labeled "claim" are supported by either analytic or experimental results. Statements labeled "conjecture" are not directly supported. They are offered as interesting possibilities to be confirmed or refuted by future research. Statements in the following sections can also be construed in both a narrow and broad sense. In the narrow sense, they make assertions about a particular formalism for representing intractable theories, the *PT-FORMALISM*, and a particular set of generic simplifying assumptions, the *PT-GSAs*. Some also make assertions about specific results from the hearts or scheduling domains. All the direct experimental evidence pertains only to these narrow assertions. Nevertheless, for each narrow statement about particular generic simplifying assumptions, knowledge representations or domains, one may formulate a corresponding general statement. The more general statement asserts the existence of other generic simplifying assumptions, knowledge representations or application domains, that have the same properties. The narrow interpretation of each statement should be taken as a claim. The broad interpretation of each should be considered a conjecture.

1.6.2. Subsidiary Claims

The principal claim will be supported indirectly through a number of subsidiary claims. The subsidiary claims assert some specific properties of the *POLLYANNA* system. They fall into two groups. The first group of claims concerns properties of the analytic, generation component of *POLLYANNA*, i.e., the *approximation generator* and *theory space generator*. They establish that the generator is capable of producing approximate theories with important desirable properties. They also assert that some undesirable approximate theories are generated as well. The second group of claims concerns properties of the system as a whole, i.e., including the empirical, *theory space search* component. They assert that empirical methods are capable of sorting out the useful approximate theories from the useless ones. The subsidiary claims will be directly supported by experimental data. Taken together, they support the primary claim described above.

1.6.2.1. Properties of Theories Generated from *GSAs*

The thesis begins by claiming that certain types of theories can be derived from generic simplifying assumptions in the approximation generator of *POLLYANNA*. These are existence claims. They state that certain approximate theories lie somewhere in the large space of all possible theories that can be derived by the approximation generator. Each derivation uses only an initial intractable theory, *IT*, generic simplifying assumptions, *GSAs*, and truth-preserving reformulations, *Rs*. These claims assert properties of the *best* or *worst* approximate theories that can be generated. They do not address the issue of what other approximate theories can be generated as well.

Subsidiary Claim #1: Approximate theories with the following properties can be derived from an intractable theory (IT) through the application of generic simplifying assumptions (GSAs) and truth preserving reformulations (Rs):

- *Semantic Properties: Many derived approximations are semantically equivalent to plausible heuristics. Some are semantically equivalent to previously known novice level heuristics. Others are comparable in accuracy to, but semantically distinct from, previously known novice level heuristics.*
- *Operationality Properties: All derived approximate theories are operationally usable. They are expressed in a language that is executable on a real machine. They require only the same input data that is required by the initial intractable theories.*
- *Context Sensitivity Properties: Sets of derived approximate theories exhibit accuracy and efficiency levels defining a flexible tradeoff between these two goals. Performance can be tailored to suit a variety of performance contexts by selecting a theory that appropriately balances the competing goals of accuracy and efficiency.*
- *Accuracy Properties: The derived approximations have widely varying levels of accuracy. The best are as accurate as previously known novice level heuristics. The worst are less accurate than random guessing.*
- *Efficiency Properties: The derived approximations have widely varying levels of efficiency. All are more efficient than the initial absolutely intractable theory. Each is efficient enough to be used in some natural performance contexts.*
- *Pareto Optimality Properties: Some derived approximations are Pareto optimal, while others are not. Pareto optimal theories dominate the non Pareto optimal ones in both accuracy and efficiency.*

Claims about properties of approximate theories generated by *GSAs* (Subsidiary Claim #1) will be supported by mechanically deriving various approximate theories using the approximation generator of *POLLYANNA*. Each derivation will be seen to depend only on the initial theory, generic simplifying assumptions and truth-preserving reformulations. Semantic properties shall be demonstrated by informal proofs showing that some derived theories are semantically equivalent to informally stated heuristics. Derived approximate theories will also be empirically tested against training examples. The operationality property will be supported by actually executing derived theories on a real machine.[11] Accuracy, efficiency, Pareto optimality and context sensitivity properties will be

[11] A Symbolics 3600 LISP Machine.

supported by collecting measurements of the accuracy and efficiency of each approximate theory as measured against training examples.

The semantic property claim can be illustrated by pointing to specific heuristics generated by *POLLYANNA*. Examples of such heuristics are listed in Figure 1-8. The heuristics are characterized here in verbal terms. Equivalence between the verbal characterizations and the symbolic expressions generated by *POLLYANNA* will be demonstrated in Chapter 3. The results described in Figure 1-8 indicate that *POLLYANNA* is capable of generating approximate theories that are semantically equivalent to plausible or novice level heuristics.

HEARTS HEURISTICS:

Lose the Current Trick:
"If some card is defeated by a card already on the table, then play any such defeated card."

Lose and Dump High Ranks:
"If some card is defeated by a card already on the table, then play a defeated card of maximal rank."

Lose and Dump High Point Values:
"If some card is defeated by a card already on the table, then play a defeated card of maximal point value."

SCHEDULING HEURISTICS:

Soonest Scheduling Deadline First:
"Choose a job with the soonest scheduling deadline."

Most Critical Job First:
"Choose a job that best contributes to satisfying preconditions of other jobs."

Figure 1-8: A Selection of the Heuristics Generated by *POLLYANNA*

These results are significant for two main reasons. One concerns the manner in which each heuristic was derived. Each heuristic results from a derivation that applies *only* domain independent operations (generic simplifying assumptions and truth-preserving reformulations) to an initially intractable domain theory. The second concerns the order in which the two domains were studied. All the domain independent operations (*GSAs* and *Rs*) were developed and implemented in the course of investigating the hearts domain. The scheduling domain was investigated after the implementation was complete. Thus the operations were designed to derive known heuristics in the hearts domain. Once designed, they derived heuristics for the scheduling domain as well.

1.6.2.2. The Importance of Empirical Learning

Empirical learning is useful in *POLLYANNA* because of the generative properties of generic simplifying assumptions. Approximate theories with widely varying levels of accuracy and efficiency are generated by generic simplifying assumptions. Some approximate theories are Pareto optimal, while others are not. *GSAs* are considered an "imperfect" generator because they produce both Pareto optimal and non Pareto optimal approximations. Empirical tests of accuracy and efficiency are therefore useful. They distinguish the Pareto optimal approximate theories from the non Pareto optimal ones.

Empirical learning is also claimed to be feasible in *POLLYANNA*. Feasibility depends on the information and computation costs of testing candidates against training examples. Depending on the number of candidates to be tested, empirical learning may require inordinate amounts of training information and computation time. The number of candidates depends in part on the generative properties of generic simplifying assumptions. It also depends on control strategies used to guide the process of generating approximations from generic simplifying assumptions. The existence of suitable control strategies therefore becomes the subject of an important claim about *POLLYANNA*.

Controlled *GSA* application leads to a better generator than some alternatives that can be envisioned. One alternative is an unbiased generator. An unbiased generator produces at least one candidate theory for each member of the unbiased hypothesis space. (See Section 1.3.2.2.) It therefore produces least one candidate semantically equivalent to each approximate theory that can be generated from *GSAs*. An unbiased generator might therefore be claimed equal or superior to a generator using *GSAs*; however, the unbiased generator also produces vast numbers of candidates not generated by *GSAs*. A more strongly biased hypothesis space results from controlled use of *GSAs* to generate candidate theories. The costs of empirical testing are therefore much lower when candidates are generated from controlled application generic simplifying assumptions to an intractable domain theory.

Subsidiary Claim #2: Empirical learning is both useful and feasible in a system that generates approximations from generic simplifying assumptions.

- *Empirical learning is useful because GSAs are an imperfect generator of approximations. Empirical tests can distinguish Pareto optimal theories from non Pareto optimal ones.*
- *Empirical learning is feasible because there exist suitably selective strategies for controlling application of GSAs in a generator of approximations.*
    - *Controlled GSA application leads to a theory space that is more strongly biased than an unbiased space.*
    - *Controlled GSA application leads to sufficiently few candidate theories so that empirical testing is feasible within reasonable information and computation resources.*

Empirical learning serves an additional purpose in *POLLYANNA*. Empirical measurements generate data to precisely quantify the tradeoff between accuracy and efficiency. To achieve a given level of efficiency, one may ask how much accuracy must be sacrificed. For a given level of accuracy, one may ask how much efficiency must be sacrificed. Empirical learning collects data that provides precise quantitative answers to these questions. The empirical data can be used to select approximations that best balance the competing concerns of accuracy and efficiency.

Subsidiary Claim #3: Empirical learning helps to manage the tradeoff between accuracy and efficiency.
- *Empirical measurements serve to quantify the tradeoff between accuracy and efficiency.*
- *Empirical measurements can be used to select an approximate theory that best balances the competing concerns of accuracy and efficiency.*

The claim that empirical learning is useful (Subsidiary Claim #2) will be supported by data from the empirical theory space search component of *POLLYANNA*. The usefulness of empirical learning will be supported by accuracy and efficiency measurements showing the existence of both Pareto optimal and non Pareto optimal approximate theories. The feasibility of empirical learning (Subsidiary Claim #2) will be supported by analyzing the computation and information requirements of empirical testing. These requirements will be shown to depend on the number of candidate theories to be tested. Analytic bounds on the sizes of candidate theory spaces will therefore prove relevant to this claim. Such bounds will be derived by analyzing several possible control strategies that limit generation of candidate theories. The claim that empirical learning quantifies the accuracy/efficiency tradeoff (Subsidiary Claim #3) will be supported by using accuracy and efficiency measurements to construct a curve that precisely described the terms of the tradeoff.

1.6.3. The Generality of *GSAs*

The *generic* nature of *GSAs* now becomes the subject of a specific claim. Two distinct *types* of generality are claimed. A *GSA* exhibits *formal* generality to the extent that it can be instantiated in a variety of situations. A *GSA* exhibits *useful* generality to the extent that it leads to accurate and efficient approximate theories in a variety of situations. Different *degrees* of generality are also claimed. Generality within a domain results from varying the parts of a theory to which *GSAs* are applied. Generality across domains involves varying the application domain itself.

Claim #2: Generic simplifying assumptions are domain independent means of generating approximations for intractable theories:
- *All GSAs exhibit formal, prima facie generality across domains.*
- *Each GSA can be usefully applied in multiple ways to a single domain theory.*

- *Each GSA can be usefully applied in multiple domains.*

The claim of generality (Claim #2) will be supported by generating approximate theories in a variety of situations. Generality across domains will be demonstrated by using the *PT-GSAs* to generate approximate theories that are equivalent to plausible heuristics for both the hearts and scheduling domains. A more detailed analysis will compare individual *PT-GSA* operator usage rates in each domain. Generality within single domains will be demonstrated by using the *PT-GSAs* to derive multiple, mutually inconsistent approximations from a single domain theory. The claim of formal, prima facie generality will be supported by exhibiting the *PT-GSA* operator definitions.

The claim of useful generality is a matter of degree. The *PT-GSA* family of generic simplifying assumptions is claimed be useful for a significant class of application domains. It is not claimed to be useful for all application domains. The *PT-GSAs* appear best suited to applications that can be structured as a sequence of decisions. The general methodology of *POLLYANNA* appears best suited to applications that involve hierarchical problem-solving. These types of problems will be characterized in Chapter 6, which discusses the useful range of application of *POLLYANNA*.

1.6.4. A Principled Method of Deriving Heuristics

POLLYANNA uses a principled method to derive heuristics. The process is considered "principled" because it explicitly identifies both the facts and the assumptions on which each generated heuristic depends. Each derivation depends only on facts from the initial theory *IT*, generic simplifying assumptions *GSAs* and truth-preserving reformulations *Rs*. The reformulations *Rs* are all theorems of logic, algebra or probability theory. Each derivation is therefore logically sound in the following sense: If the initial theory is correct and the instantiated *GSAs* are assumed to be true, the heuristic follows as a logically sound conclusion.

Claim #3: Generic simplifying assumptions are a principled means of deriving heuristics from intractable theories:

- *Each derivation identifies a set of facts from the domain theory on which the heuristic depends.*
- *Each derivation identifies a set of assumptions on which the heuristic depends.*
- *Each heuristic follows as a logically sound consequence of the facts from the initial theory and the instantiated generic simplifying assumptions.*

The claim that *GSAs* afford a *principled* means of deriving heuristics (Claim #3) will be supported by exhibiting the derivations of various approximate theories. Each derivation will be seen to depend on only the initial theory and a set of transformation rules. Each transformation rule will be seen to describe either a generic simplifying assumption *GSA* or a truth-preserving reformulation *R*. The correctness of each reformulation will be demonstrated informally, in cases where such correctness is not self-evident. The derivations will therefore be logically sound, provided the initial theory and the instantiated *GSAs* are assumed to be true.

1.6.5. Analytic Reasoning about Properties of Approximations

Claim #4: Analytic methods can organize approximate theories into a search space with useful structural properties.

- *Analytic methods can partially order approximate theories to generate a space that is monotonic in efficiency.*
- *Analytic methods can partially order approximate theories to generate a space that is monotonic in accuracy. (Conjecture.)*
- *The search for accurate, efficient theories is facilitated by the monotonic theory space organization.*

Generic simplifying assumptions (*GSAs*) will be associated with theorems that describe their impact on the efficiency of computational theories. Efficiency impact theorems will be formulated and proved. (Each proof will depend on assumptions about properties of the intractable domain theory to which the *GSA* is applied.) Efficiency impact theorems are implemented as rules that can partially order theories according to computational efficiency, resulting in a space that is monotonic in efficiency. Theory space search algorithms will be seen to exploit such monotonicity to avoid exhaustive search.

1.6.6. Interaction of Approximation and Reformulation

Claim #5: Truth preserving reformulations are required to fully exploit the potential of generating approximate theories from generic simplifying assumptions.

- *Reformulations enable subsequent application of efficiency improving GSAs.*
- *GSAs enable subsequent application of efficiency improving reformulations.*

1.6.7. Comparison to Alternative Methods

POLLYANNA is naturally compared to explanation-based generalization, since EBG is a method for improving the efficiency of computational theories. As argued above, *POLLYANNA* applies to some types of intractable theory problems to which EBG cannot be applied at all. For those intractable theory problems to which both methods apply, EBG is restricted to preserving the accuracy of the initial theory. *POLLYANNA* might therefore achieve superior results in these cases as well.

Claim #6: The machine learning methodology embodied in POLLYANNA is superior to explanation-based generalization for some applications:

- *POLLYANNA can be usefully applied to absolutely, non-deterministically intractable theories. EBG fails to produce any results whatsoever for such theories.*

- *For some domain theories to which both EBG and POLLYANNA can be formally applied, POLLYANNA will produce more efficient theories by sacrificing the requirement of perfect accuracy. (Conjecture.)*

Support for the claim that *POLLYANNA* applies to computational theories not handled by *EBG* (Claim #6) will be provided by the hearts domain implementation. The initial hearts theory will be shown to be absolutely, nondeterministically intractable, for all reasonable resource bounds. *EBG* will be shown not to apply to the hearts domain theory used in this research. In the course of supporting Subsidiary Claim #1, *POLLYANNA* will be seen to produce approximate theories that are equivalent to plausible hearts heuristics, and that are efficient enough to be used in some realistic performance contexts. *POLLYANNA* will thus be shown to handle at least one domain theory not handled by *EBG*.

EBG has been criticized for failing to perform knowledge level learning [Dietterich 86]. As described above, a system is said to perform knowledge level learning only when the deductive closure of its initial knowledge base changes over time. *EBG* is a truth-preserving process. All the learned schemata are logical consequences of the initial knowledge base. The deductive closure does not change and *EBG* does not learn at the knowledge level. *POLLYANNA* is not subject to this criticism. This claim will be demonstrated by generating learning curves from the data collecting during the empirical, theory space search phase. Learning curves will demonstrate that improvements in accuracy result from increasing the numbers of training examples supplied to the system.

Claim #7: POLLYANNA learns at the knowledge level

It is also interesting to compare *POLLYANNA* to purely empirical learning methods. The power of *POLLYANNA* resides in its ability to generate a space of candidate approximate theories, starting with an initially intractable theory. This hypothesis space is biased, since it does not contain all possible hypotheses. The bias depends explicitly on the initial intractable theory. When the initial theory is changed, the candidate theory space changes as well. For this reason, *POLLYANNA* is conjectured to learn more quickly or more accurately in most domains than a purely empirical learning system with a fixed space of candidate hypotheses.

Conjecture #1: Heuristics for intractable theories can be learned more accurately or with fewer examples using the approach embodied in POLLYANNA than is possible alternative systems that use a fixed hypothesis space.

A final comparison involves other systems that use an initial domain theory to guide the learning process. *POLLYANNA* manifests only one of many possible methods for generating candidate hypotheses from an initial domain theory. Other integrated analytic/empirical learning systems appear to perform a similar task. Several such systems are surveyed in [Ellman 89a; Danyluk 89]. Comparisons to such systems are difficult because they use such diverse representations and architectures. In any case, only future research will determine whether *POLLYANNA* performs as well as or better than alternative integrated learning approaches to the intractable theory problem.

1.6.8. The Power of a Paradigm

POLLYANNA may be viewed as one step toward a full solution to the intractable theory problem. Considerable efforts will be made in this dissertation to show how *POLLYANNA* makes progress toward that goal. The research underlying *POLLYANNA* has produced results that are specialized to one formalism, the *PT-FORMALISM*, and two domains, hearts and scheduling. Several distinct research tasks were faced and completed in the course of achieving these results. These included the identification and representation of certain types of knowledge, (e.g., *GSAs*), and proving some theorems about their properties, among other things. The final chapter of this document will enumerate and characterize these research tasks. The tasks will be seen as instances of a general research paradigm [Kuhn 70]. The paradigm will illustrate how the present results might be extended to other knowledge representation formalisms and other application domains.

Research in the *POLLYANNA* paradigm is guided by a fundamental hypothesis about the nature of heuristics. Based on the experience of *POLLYANNA*, all problem-solving heuristics are conjectured to be derivable from intractable theories, using only generic simplifying assumptions and truth-preserving reformulations. Although this conjecture will probably fail to be completely correct, the reasons for such failure will likely be illuminating. In any case, the conjecture serves the useful purpose of focusing research into machine learning solutions to the intractable theory problem.

Conjecture #2: All problem-solving heuristics can be derived by applying generic simplifying assumptions and truth-preserving reformulations to an initially intractable theory.

Conjecture #3: POLLYANNA instantiates a research paradigm, the articulation of which will eventually produce a solution to the intractable theory problem.

1.7. Recurring Themes of this Research

A Model of Plausible Reasoning:

This research has developed a powerful technique for reconstruction of heuristics. The method involves deriving each heuristic *H* from a correct but intractable theory *IT*, generic simplifying assumptions *GSAs* and truth-preserving reformulations *Rs*. Such derivations are said to be "principled" because they identify both domain knowledge and assumptions on which heuristics depend. The derivations are logically sound, provided the domain theory and instantiated generic simplifying assumptions are assumed to be true. The derivations indicate how plausible heuristics are derivable from correct but intractable domain theories under generic simplifying assumptions.

A Flexible Tradeoff between Accuracy and Efficiency:

A particular strategy for dealing with intractability has been developed by this research. The strategy involves sacrificing perfect accuracy in order to gain efficiency in return. Results show that the tradeoff can be carried out in various ways. Many combinations of accuracy and efficiency can be attained. Depending on the requirements of the performance context, various different combinations may be appropriate. The approach thus achieves flexible and context sensitive tradeoff between the competing goals of accuracy and efficiency.

Integrated Models of Integrated Learning:

POLLYANNA illustrates the power of integrated analytic/empirical learning. It also illustrates the limitations of existing models of the learning process. Neither *knowledge level learning*, with its focus on *accuracy*, nor *operationality*, with its focus on *efficiency*, is sufficient for characterizing and evaluating the behavior of *POLLYANNA*. This research thus illustrates the need for learning models that combine multiple measures of performance.

A Style of Learning:

POLLYANNA uses analytic methods to divide computations into sub-problems, and sub-subproblems, etc. Empirical methods enable the system to distinguish between important, critical subproblems, and unimportant, non-critical subproblems. Subproblems are approximated separately. Non-critical ones are approximated to a large degree. Critical subproblems are approximated very little, if at all. The system learns to selectively deploy limited computational resources. Resources are allocated away from non-critical tasks, and toward the more critical parts of a computation.

1.8. Reader's Guide to the Dissertation

The following chapters can be grouped into two parts that correspond roughly to the analytic (generate) and empirical (test) components of *POLLYANNA*. The analytic portion is discussed mainly in Chapters 2 and 3. The architecture of *POLLYANNA*'s *approximation generator* is discussed in Chapter 2. Experimental results from the this module are presented in Chapter 3. The empirical portion is discussed mainly in Chapters 4 and 5. The architectures of *POLLYANNA*'s *theory space generation* and *theory space search* modules are discussed in Chapter 4. (Although *theory space generation* is an analytic process, it is more easily discussed along with the *theory space search* module.) Experimental results from these modules are presented in Chapter 5.

Chapter 2, entitled "Analytic Generation of Heuristics", presents a general framework for deriving heuristics from intractable theories. The framework involves the interaction of intractable theories, generic simplifying assumptions and truth-preserving reformulation knowledge. The framework illustrates the principled nature of *POLLYANNA*'s derivations (Claim #3). This chapter also elaborates the notion of generic simplifying assumptions presented above. The *PT-GSA* family of generic simplifying assumptions is presented and discussed. The *PT-FORMALISM* for representing domain theories is discussed along with representations of the hearts and scheduling theories. The hearts domain theory in particular is shown to be absolutely, non-deterministically intractable, and therefore not amenable explanation-based generalization *(EBG)* (Claim #6). Truth preserving reformulations used in *POLLYANNA* are presented along with a discussion of the manner in which reformulations interact with assumptions (Claim #5). Search control issues are also discussed in this chapter.

Chapter 3, entitled "Analytic Learning Results in *POLLYANNA*", demonstrates the behavior of *POLLYANNA*'s approximation generator. Examples of approximate theories the system can generate for the heart and scheduling domains are presented. Informal proofs are used to show these approximate theories to be equivalent to some known heuristics as well as some previously unknown heuristics. The claims regarding semantics (Subsidiary Claim #1) and generality (Claim #2) are supported by the results presented in this chapter.

Chapter 4 entitled "Empirical Testing of Heuristics" begins by discussing the goals of learning. *POLLYANNA* is shown to handle various types of accuracy and efficiency goals. The choice of an appropriate goal is seen to depend on the intended performance context. This chapter also includes discussions of both the *theory space generation* and *theory space search* modules of *POLLYANNA*. The sections on *theory space generation* show how *POLLYANNA* combines individual approximations to form complete approximate theories. They also describe how the system attempts to generate a search space that is monotonic in the efficiency of approximate theories (Claim #4). The theory space generator is shown to depend on theorems describing the impact of *PT-GSAs* on the efficiency of approximate theories. The relevant theorems are presented and proved. An example of a theory space generated by this module is presented as well. Analytic bounds in the sizes of generated theory spaces are derived in this chapter. Such bounds support the claim that empirical learning is feasible in *POLLYANNA* (Subsidiary Claim #2). The section on *theory space search* will discuss algorithms for searching a space of candidate theories. Several algorithms are presented. Each handles different types of learning goals.

Chapter 5 entitled "Empirical Learning Results in *POLLYANNA*" presents the results of empirically testing heuristics. *Theory space search* results for the hearts domain are presented. Measurements of accuracy and efficiency for approximate hearts theories are analyzed and discussed. These results directly support the claimed operationality, accuracy, efficiency, Pareto optimality and context sensitivity properties of theories generated from generic simplifying assumptions (Subsidiary Claim #1). The empirical results also support the claim regarding the usefulness and feasibility of empirical learning (Subsidiary Claim #2), the role of empirical learning in managing the tradeoff between accuracy and efficiency (Subsidiary Claim #3) and the fact of knowledge level learning in *POLLYANNA* (Claim #7).

Chapter 6 entitled "Conclusion", examines the claims presented above. Each claim is evaluated in light of the experimental results obtained from *POLLYANNA*. This chapter also describes how *POLLYANNA* can be viewed as a first step toward a complete solution to the intractable theory problem. A paradigm for future research is presented. The paradigm specifies the sorts of research tasks necessary to extend *POLLYANNA*'s results to new formalisms and application domains.

Chapter 2

Analytic Generation of Heuristics

2.1. Introduction

2.1.1. A View of Heuristics

The concept of *heuristic* has a long history in Artificial Intelligence. AI has usually considered human experts to be the source of heuristics. Heuristics supplied by human experts are considered to capture insights that the expert has gleaned from a long experience of working in the application domain. This traditional view neglects the relation between heuristics and underlying theories of the domain under study. An alternative view examines heuristics in relation to computationally intractable theories. A variety of heuristics can be derived by applying formal operations to intractable domain theories. The formal operations introduce approximations that result in tractable approximate theories that are equivalent to heuristics.

The alternative viewpoint offers two major advantages. The first concerns machine learning of heuristics. After identifying the operations that can transform exact, intractable theories into approximate, heuristic theories, it becomes possible to mechanically or automatically learn heuristics. The second advantage concerns previously known heuristics. Starting from an intractable domain theory, a series of formal operations can be used to *reconstruct* previously known heuristics. Reconstruction often yields considerable insight into the rationale underlying a known heuristic. It identifies the domain knowledge on which the heuristic depends. It also identifies assumptions that are sufficient to prove correctness of the heuristic.

2.1.2. Overview

This chapter discusses the analytic learning process of generating heuristics in *POLLYANNA*. A general framework for deriving heuristics is presented first, in Section 2.2. Particular attention is paid to the types of knowledge involved in generation of heuristics. The framework is illustrated through informal derivation of an example heuristic. The example illustrates the central role of *generic simplifying assumptions (GSAs)* in generation of heuristics (Figure 1-5). This section also shows some of the advantages and disadvantages of using *GSAs* to generate heuristics.

The architecture of *POLLYANNA*'s *approximation generation* module is described in Section 2.3. The generator is described in terms of a problem space model. Domain theories correspond to problem states. Generic simplifying assumptions are implemented as operators that transform problem states. The representation of domain theories for hearts and job scheduling are presented in Section 2.4. These two domain theories are shown to be instances of a general formalism for representing theories: the *PT-FORMALISM*. The power and generality of this formalism is also discussed in Section 2.4.

A family of generic simplifying assumptions is presented in Section 2.5. The *PT-GSAs* are a formal implementation and elaboration of the *GSAs* described verbally in Figure 1-5. The *PT-GSAs* are designed to interface with domain theories written in the *PT-FORMALISM*. The semantics of each *PT-GSA* is discussed. Examples of corresponding specific assumptions from the hearts domain are presented. The types of truth preserving reformulation knowledge used in *POLLYANNA* are discussed in Section 2.6.

Search control methods for *POLLYANNA*'s approximation generator are discussed in Section 2.7. Problems with blind search are considered. An automatic search control strategy is proposed along with a discussion of possible extensions to the strategy. A relation between the issue of search control and the feasibility of empirical learning is discussed as well.

2.2. *H = IT + GSA + R*: A Framework for Generating Heuristics

POLLYANNA uses a domain independent mechanism to derive heuristics from intractable domain theories. A general framework for such derivations is described in Figure 2-1. This diagram shows three types of knowledge that are used in the derivation process: An *intractable theory, reformulation knowledge* and *generic simplifying assumptions*. The intractable theory *(IT)* is a domain-dependent input to *POLLYANNA*'s heuristic generation process. The reformulation knowledge *(R)* consists of domain independent theorems of logic, set theory, algebra and probability theory. The reformulations are designated "truth-preserving" because they are guaranteed to be true regardless of the application domain. A generic simplifying assumption *(GSA)* is a schema describing a class of simplifying assumptions. Each specific simplifying assumption is an assertion that is not necessarily true. It may nevertheless be useful if it simplifies an intractable theory without introducing too many errors. A *GSA* is considered to be "generic" because it can be instantiated in a variety of domains or in a variety of ways in a single domain. Some examples of *GSAs* were shown in Figure 1-5. These three types of knowledge will be discussed in more detail in Sections 2.4, 2.5 and 2.6.

Figure 2-1: Heuristic Generation Framework

2.2.1. An Example Heuristic

The process of generating heuristics will be illustrated using the card game hearts. A heuristic for playing hearts will be presented along with an informal derivation. The example heuristic and derivation are intended to serve several purposes. To begin with, they will provide intuitive support for the idea that heuristics *can* be derived from intractable theories. The derivation will also illustrate each of the three types of knowledge, *IT, GSAs and Rs*, and their respective roles in the process of deriving heuristics. The *GSAs* described earlier in Figure 1-5 will be used to derive a simple heuristic for the card game hearts. The derivation will thus motivate the attention given to these particular generic simplifying assumptions.

A situation from a hearts game is shown in Figure 2-2. In this situation, the ♦King has just been played.

According to the rules, the player can legally play only one of ♦QUEEN, ♦4 or ♦2. The player chooses to play his ♦QUEEN. In so doing, he might be using the heuristic described in Figure 2-3, the *Dump High Rank Rule*. The heuristic suggests playing the highest rank legal card in situations like the current one, when all legal card choices cause the player to lose the current trick.

```
Problem Situation:

    Lead Suit:        ♦

    Cards on Table:   ♦ KING

    Player's Hand:    ♠ JACK, 7
                      ♥ ACE, 10
                      ♦ QUEEN, 4, 2
                      ♣ JACK, 9

Player's Choice:      ♦ QUEEN
```

Figure 2-2: A Hearts Problem and Solution

```
IF:   All legal card choices lose
      the current trick,

THEN: Play a legal card of HIGHEST RANK.
```

Figure 2-3: Dump High Rank Rule

The heuristic in Figure 2-3 can be understood using the following informal explanation: *"If all legal choices lose the current trick, the player's choice should be made with an eye toward future tricks. By playing the highest rank card, the player is left with relatively lower cards in his hand. The lower cards will result in winning fewer future tricks. The player's final score will therefore be lower. Since the game objective is to minimize the final score, it is best to play the highest rank card."*

POLLYANNA derives this heuristic by building a derivation structured like the abstract one in Figure 2-1. The derivation uses facts from the rules of the game *(IT)*, generic simplifying assumptions *(GSAs)* and truth-preserving reformulation knowledge *(Rs)*. *POLLYANNA*'s derivation is a formal version of the verbal explanation given above. The verbal explanation is *plausible*, but not particularly rigorous. *POLLYANNA*'s derivation is a *logically sound* proof, provided that one accepts the initial theory and the simplifying assumptions as givens. The derivation framework, *H=IT+GSA+R*, is an attempt to model *plausible explanation* in terms of sound proofs based on generic simplifying assumptions.

2.2.2. Derivation of the Example Heuristic

The power of generic simplifying assumptions *GSAs* is illustrated by the following informal derivation. This derivation uses the *GSAs* in Figure 1-5 to derive the *Dump High Rank Rule* shown in Figure 2-3. It was constructed by hand. *POLLYANNA* itself uses a more lengthy and complex derivation to generate an approximate theory equivalent to the *Dump High Rank Rule*. (The trace of an implemented derivation is shown in Appendix I.) The informal derivation is presented here to motivate further discussion of generic simplifying assumptions.

The hearts theory is presumed to be represented in terms of an evaluation function, *Exp-Game-Score(c)*, that computes the player's expected score at the end of the game, given that he plays card *c* in the current trick. Using this intractable theory, the player would choose cards by finding one that minimizes this expectation value. In the course of the derivation, the theory is simplified by replacing various exact sub-expressions with approximate versions. Generic simplifying assumptions are used to determine which expressions to simplify, and how they should be simplified. The following derivation is an abridged version of the complete one, which appears in Appendix A.

- The player chooses a legal card that minimizes his expected score. He uses an evaluation function that returns his expected score at the end of the game, given that he plays card *c* in the current trick:

```
Choice = Minimize(Legal-Cards, Exp-Game-Score)

Exp-Game-Score(c) =

 = Current-Score
   + Exp-Current-Trick-Score(c)
   + ∑ (t in Tricks-Left) Exp-Future-Trick-Score(t,c)
```

- The player's expected score on trick *t* can be written in terms of the probability of winning the trick and the expected value of the trick. This requires an assumption of probabilistic independence:

```
Exp-Future-Trick-Score(t,c) = Prob-Win(t,c) * Exp-Trick-Value(t,c)
```

IN Assumption: The probability of winning the trick is statistically independent of the trick value.

- To compute the expected value of a trick, the player simply uses an average value, treating the function as a constant:

```
Exp-Trick-Value(t,c) = TOTAL-DECK-VALUE / NUMBER-OF-TRICKS
```

FI Assumption: The expected trick value is a constant, equal to the average value of a trick.

- The probability of winning a trick can be expressed as a summation over all the cards that might be played and the range of possible lead suits. For each combination, the player must determine whether he wins the trick:

```
Prob-Win(t,c) = ∑ (c1,c2,c3,c4 in DECK, s in suits)
                   Prob-Trick(c1,c2,c3,c4,s,t,c)
                   * [If Win(c1,c2,c3,c4,s) then 1 else 0]
```

- The player must now compute the odds that he will play *c1*, his opponents will play *c2*, *c3* and *c4* and that the lead suit will be *s*. Using an assumption of probabilistic independence, this may be written as a product of individual probabilities:

```
Prob-Trick(c1,c2,c3,c4,s,t,c) = Prob-Choice(me,c1,t,c)
                                * Prob-Choice(left(me),c2,t,c)
                                * Prob-Choice(right(me),c3,t,c)
                                * Prob-Choice(across(me),c4,t,c)
                                * Prob-Lead-Suit(s,t,c)
```

IN Assumption: The four card choices and the lead suit are statistically independent.

- In order to compute the odds of card choices and lead suits in future tricks, the player assumes that sets of possible alternatives are equally likely:

```
Prob-Choice(me,k,t,c)  = If Member(k,Hand(me) - {c})
                            then 1 / |Hand(me) - {c}|
                            else 0
```

EP Assumption: The player is equally likely to play any card remaining in his hand.

```
Prob-Choice(p,k,t,c) = 1 / |DECK|  (If p is an opponent.)
```

EP Assumption: An opponent is equally likely to play any card in the deck.

```
Prob-Lead-Suit(s,t,c) = 1 / |SUITS|
```

EP Assumption: The lead suit is equally likely to be any suit.

- These function definitions can be combined into a more compact form. (See Appendix A.) After evaluating the constant expressions and manipulating summations, an expression in the following form is obtained: (The constants *K1* and *K2* are both positive numbers.)

```
Exp-Game-Score(c) =

 = Current-Score
   + Exp-Current-Trick-Score(c)
   + K1 * ∑ (k in Hand(me) - {c}) [Rank(k) + K2]^3
```

This approximate evaluation function is equivalent to the heuristic in Figure 2-3. When used to choose cards in the course of a game, it leads to the same choices as would be generated by the heuristic rule. In order to demonstrate the equivalence, one must show that the *Exp-Game-Score(c)* is minimal when *c* is the highest rank legal card in *Hand(me)*, provided all legal cards lose the current trick. If all legal cards lose the current trick, then *Exp-Current-Trick-Score(c)* is identically zero. The evaluation function *Exp-Game-Score(c)* reduces to a summation over the unplayed cards *Hand(me) - {c}*. Each term has the form $(Rank(k) + K2)^3$. (The cubic exponent reflects the number of opponents.) The summation is minimized when the played card *c* has maximal rank.

The derivation makes use of each of the three types of knowledge that were mentioned as part of *POLLYANNA*'s framework for generating heuristics. Some of the function definitions were taken directly from the initial intractable theory *(IT)*. For example, derivation uses the function *Win(c1,c2,c3,c4,s)*, taken from the initial theory, to determine whether card *c1* wins the trick, given opponents' choices *c2*, *c3*, *c4* and a lead suit *s*. The derivation also uses some domain independent reformulation knowledge *(R)*. This knowledge includes rules to manipulate summations and evaluate constant expressions, as well as the definitions of domain independent arithmetic and logical operations.

The derivation also uses each of the generic simplifying assumptions *(GSAs)* described in Figure 1-5. *Function Invariance (FI)*, was used to ignore the expected trick value of future tricks. *Probabilistic Independence (IN)*, was used twice: (1) Assuming independence of winning a future trick and the value of the trick; (2) Assuming independence of the opponents' future choices. *Equiprobable Random Variables (EP)*, was used three times: (1) Assuming the player is equally likely to play any card remaining in his hand; (2) Assuming opponents are equally likely to play any card in the deck; (3) Assuming the lead suit is equally likely to be any suit.

2.2.3. A Second Example Heuristic

The power of generic simplifying assumptions can be further illustrated with a second example heuristic from hearts. Consider the heuristic shown in Figure 2-4, the *Dump High Point Value Rule*. This heuristic recommends playing a card of highest point value, when all legal card choices lose the current trick. It can be derived using the same set of generic simplifying assumptions as were used to derive the heuristic in Figure 2-4. In the new derivation, one simply interchanges roles *Prob-Win* and *Exp-Trick-Value* in the first derivation. Now the probability of winning a trick is approximated as a constant. The expected trick value is calculated by summing over all possible combinations of four cards. As in the first derivation, the player assumes each opponent is equally likely to play any card in the deck, and that he himself is equally likely to play any card remaining in his hand. A complete derivation is found in Appendix A. The following evaluation function is the result:

```
Exp-Game-Score(c) =

 = Current-Score
   + Exp-Current-Trick-Score(c)
   + K1 + K2 * ∑ (k in Hand(me) - {c}) Point-Value(k)
```

Whenever all legal cards lose the current trick, the expected current trick score is identically zero. In such cases, the expected game score increases in proportion to the total point value of cards remaining in the player's hand. This quantity is minimized by playing a legal card of maximal point value.

```
IF:   All legal card choices lose
      the current trick,

THEN: Play a legal card of HIGHEST POINT VALUE.
```

Figure 2-4: Dump High Point Value Rule

2.2.4. The Generality of *GSAs*: Positive and Negative Consequences

The generality of generic simplifying assumptions is clearly illustrated by the two derivations. Using only the initial intractable theory (*IT*) and truth-preserving reformulations (*Rs*), a small set of generic simplifying assumptions (*GSAs*) is sufficient to derive simple but plausible heuristics for hearts. Notice that the two heuristics are mutually inconsistent. In most card playing situations the card of highest rank is not the same as the card of highest point value. The *Dump High Rank Rule* recommends a different choice than the *Dump High Point Value Rule*. Nevertheless, they were both derived using the same set of generic simplifying assumptions. The two derivations used different *instantiations* of the same *GSAs*. The *GSAs* in Figure 1-5 refer to *general* parameters like functions and random variables. Distinct instantiations result from selecting different *specific* functions and random variables. For example, both derivations used the generic assumption of *Function Invariance* (*FI*), (Figure 1-5). They differ on the specific function assumed to be invariant. Generic simplifying assumptions thus exhibit a property called "generality within the context of a single domain theory". A set of *GSAs* can derive multiple, mutually inconsistent heuristics, when applied to a single intractable theory. The following sections show that both positive and negative consequences result from the generality of *GSAs*.

2.2.4.1. Discovery of New Heuristics

Generic simplifying assumptions can be used to discover new heuristics. Discovery is possible as a result of the generality of *GSAs*. The two derivations showed that more than one heuristic can be generated from a single set of *GSAs*. In the context of a single domain theory, a wide variety of heuristics can be generated by systematically instantiating a single set of *GSAs*, with the possibility of discovering previously unknown heuristics for the original domain. Examples of new heuristics discovered through such a process in the domain of hearts will be presented in Chapter 3. The *GSAs* in Figure 1-5 do not refer to any domain specific concepts. They exhibit a property called "prima facie generality across domains". They can be used to generate heuristics in domains beyond those for which they were originally designed, with the possibility of discovering previously unknown heuristics for the new domain as well. Examples of heuristics discovered by applying them to the domain of job scheduling will be presented in Chapter 3.

2.2.4.2. The Value of Empirical Learning

Negative consequences also result from the generality of generic simplifying assumptions. When heuristics are generated from a set of *GSAs*, many of the heuristics are mutually inconsistent, as are the ones in Figures 2-3 and 2-4. Furthermore, the heuristics are not equally good. A set of *GSAs* can produce heuristics that vary widely in their cost of application and degree of accuracy. For each good heuristic generated from a set of *GSAs*, many bad ones are generated as well. Evidence for this fact will be presented in Chapter 5. The *GSAs* are too general to distinguish between good and bad heuristics. *POLLYANNA* therefore uses empirical methods to evaluate heuristics generated from generic simplifying assumptions. The analytic/empirical architecture of *POLLYANNA* is a consequence of the fact that *GSAs* generate both good and bad heuristics.

2.2.5. Deep v. Shallow Representations of Heuristics

A distinction between deep and shallow representations of heuristics is illustrated by the two hearts derivations. The *IF-THEN* rules in Figures 2-3 and 2-4 are both *shallow* representations. They are called "shallow" because neither gives any hint of the source of heuristic. The *IF-THEN* rules represent results of derivations in compiled form. In contrast to the *IF-THEN* rules, the two derivations are *deep* representations. They are called "deep" because they identify both the facts from the initial theory as well as the assumptions on which each heuristic depends. When these derivations are implemented in a formal system, the supporting facts and assumptions can each determined by tracing dependencies. Starting at the derived heuristic, one follows support links backwards through the derivation structure to find facts in the initial theory or instantiated *GSAs* on which the derivation depends.

Deep representations may facilitate maintenance of heuristics when there is a change in the underlying domain theory. In order to illustrate the maintenance process, consider what would happen were hearts rules were changed so that low cards defeat high cards to win tricks. The derivation of the *Dump High Rank Rule* makes use of the rules determining who wins a trick. (See Appendix A.) One would therefore expect the heuristic to change, presumably into a *Dump Low Rank Rule*. In contrast to this, the derivation of the *Dump High Point Value Rule* did not use the rules determining who wins a trick. This heuristic need not be modified to reflect the proposed change in the hearts rules. These observations suggest using derivational analogy [Carbonell 86; Mostow 89], as a means of maintaining heuristics. Whenever the theory changes, system would apply to the new theory the same series of formal operations as were used in the original derivation. In some cases, the same set of *GSAs* and *Rs* might suffice to generate an appropriate new heuristic after a change in the domain theory.

Derivational analogy *(DA)* presents a problem for *POLLYANNA* that does not arise in other systems that use this technique. *POLLYANNA* is distinguished by its use of non-truth-preserving transformations, as well as its reliance on empirical testing. Traditional *DA* systems apply truth-preserving transformations and do not rely on empirical testing. *POLLYANNA* can use the *DA* method to update the set of candidate approximate theories; however, the data obtained from empirical testing will no longer be valid. The revised candidate theories need not have the same accuracy and efficiency properties as the corresponding original ones. The empirical learning phase must be repeated. Additional research is needed to test the feasibility of performing derivational analogy in the context of *POLLYANNA*.

Deep representations also offer additional advantages. Heuristics may be more easily explained to humans when they are derived from explicit domain knowledge [Swartout 83]. Heuristics generated by *POLLYANNA* are explainable in terms of facts from the domain theory *IT*, generic simplifying assumptions *GSAs* and a series of truth-preserving reformulations *Rs*. No such explanation is possible given only the compiled, shallow representation in terms of *IF-THEN* rules. Deep representations also support debugging of heuristics. When a heuristic is observed to fail in practice, the derivation can be used to trace the failure back to one or more simplifying assumptions that might be responsible [Doyle 79]. By selectively retracting assumptions, various revisions of the heuristic can be generated. An essentially equivalent technique is used in the theory space search module of *POLLYANNA*. (See Chapter 4.)

POLLYANNA is naturally compared to several other systems for deriving programs from basic domain knowledge. Examples include the *XPLAIN* program [Swartout 83], the Φ*NIX* project [Barstow 85] and *KIDS* [Smith 88]. All of these systems, *XPLAIN*, Φ*NIX*, *KIDS* and *POLLYANNA*, can be viewed as forms of "transformational implementation" [Balzer et al. 76]. Under this programming paradigm, programs are generated from initial specifications by applying a series of truth-preserving transformations. These systems all share the advantages offered by deep representations for the tasks of maintaining, debugging and explaining programs to humans. *POLLYANNA* differs from the standard transformational paradigm in one important respect. *POLLYANNA* uses transformations that are not truth-preserving, i.e., generic simplifying assumptions, in addition to truth-preserving transformations, i.e., reformulation knowledge.

2.2.6. Development of *GSAs* through Reconstruction of Heuristics

The generic simplifying assumptions in Figure 1-5 were formulated in a process that involved *reconstruction* of known heuristics. The development process began by investigating heuristics for the hearts game. Sample heuristics were obtained by analyzing protocols of hearts games played by humans. The protocols contained records of players' card choices as well as players' verbal explanations of their decisions. The protocols were then analyzed to extract heuristics that were either mentioned explicitly in the explanations, or which be sufficient to generate the same card playing behavior. Some additional heuristics were obtained from a book on hearts [Andrews 83]. The *Dump High Rank Rule* and the *Dump High Point Value Rule* were included among the heuristics assembled. The assembled heuristics were then analyzed in the following way: For each heuristic *H*, an attempt was made to formally prove *H* using facts from the initial theory (*IT*), simplifying assumptions (*SAs*) and reformulations (*Rs*). This process resulted in several informal derivations, including the two shown above. Each derivation used *specific* simplifying assumptions relevant to the hearts domain, (e.g. (1) An opponent is equally likely to play any card in the deck; (2) Expected trick value is a constant.) The *generic* simplifying assumptions were then formulated as domain independent versions of the specific assumptions used in the derivations of known heuristics.

2.2.7. Goal Heuristics for Hearts

A collection of sample hearts heuristics is shown in Appendix B. These heuristics were originally proposed as a benchmark for evaluating the performance of *POLLYANNA* [Ellman 87]. They were formulated before implementation of *POLLYANNA* was actually begun. Experimental results from the *approximation generation* phase will show that most - but not all - of these heuristics can be generated by *POLLYANNA*. Results will also show that *POLLYANNA* can generate some heuristics not found in the original goal set. A comparison between these goal heuristics and the ones generated in *POLLYANNA* will be presented in Chapter 3.

The sample heuristics were formulated through an analysis of the human game-playing protocols mentioned above. They are written as a collection of *IF-THEN* rules. These rules implement a complete strategy for playing hearts, i.e., they specify a card choice for every possible game state. Each card choice is made to satisfy the primary objective of minimizing the number of points taken during the current trick. In order to optimize the current trick, they first pare down the choices to those cards guaranteed to take zero points in the current trick. If no such card is available, they play cards of minimal point value. Among cards of equal point value, they choose cards of minimal rank. If at least some cards are guaranteed to take zero points in the current trick, the rules optimize for future tricks, by dumping dangerous cards. The first priority is to dump cards of high point value. The second priority is to dump cards of high rank. The rules are written assuming a version of the game in which the only point cards are hearts. All other cards have zero point value.

2.3. Architecture of the Approximation Generator

The organization of *POLLYANNA*’s *approximation generation* module is shown in Figure 2-5. The diagram shows the three types of knowledge that are used to generate approximations: the initial theory, generic simplifying assumptions and reformulation knowledge. The architecture is naturally understood in terms of a problem space model. At each point in time, a problem state is defined by the current inventory of approximations. New problem states are created by application of operators to the current state. Two types of operators are used. One type of operator is used to implement generic simplifying assumptions. The other type is used to implement truth preserving reformulations.

In the current implementation all search control decisions are made by the human user. During each operation cycle, a human user chooses which operator to apply. Since each operator can usually be applied in more than one way, the user also selects a particular instantiation of the operator. After the user makes these decisions, *POLLYANNA* applies the operator and updates the problem state. This implementation is called a "mechanical"

Figure 2-5: Approximation Generator Architecture

process of generating heuristics because the operations are carried out by the system. It is not considered "automatic" because control decisions are made by humans. The burden of generating heuristics is shared between *POLLYANNA* and the human user. The possibility of building a completely automatic approximation generator will be discussed in Section 2.7.

2.3.1. Representation of Problem States

The representation of a problem state is described in Figure 2-6. A state is formally defined as a table that maps function names to sets of lambda expressions. For each function name *F*, the table returns a set of lambda expressions called *Versions(F)*. These lambda expressions represent various alternative definitions of the function *F*. In the initial state, each function has only the single definition appearing in the intractable domain theory.[12] Each

[12]Although this is the normal case, there is no reason why the initial theory designer could not provide multiple initial definitions. The alternate definitions might be semantically equivalent expressions that make the representation redundant. The alternate definitions might also represent specific approximations that the user wants the system to consider.

operator application has the effect of creating new versions of one or more functions. The various versions can differ from each other in either of two ways. One version may have been created by a reformulation operator. It will have the same semantics as the version from which it was created, but will have a different representation. One version may also be an approximation of another, having been created by application of a *GSA* operator to the original version. Complete approximate theories are formed by selecting exactly one version of each function.[13]

Versions:Functions → <Set of λ-Expressions>

Functions:	**Versions:**
f	$\mathbf{f}_0$, $\mathbf{f}_1$, $\mathbf{f}_2$
g	$\mathbf{g}_0$, $\mathbf{g}_1$
h	$\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, $\mathbf{h}_3$

Initial State: One λ-Exp for each function.

Later State: Many λ-Exps for each function.

Figure 2-6: Approximation Generator Problem State

2.3.2. The Operator Application Cycle

Most of the operators are represented as declarative rewrite rules of the form: *LHS* → *RHS*, where *LHS* and *RHS* are patterns describing algebraic expressions. The pattern language is described in Appendix F. These operators are applied in the following manner. First the user selects an operator *O*, a function name *F*, and a version *V* from the set *Versions(F)* of possible definitions of function *F*. The system then presents a list of the sub-expressions of *V* to which the operator is applicable. The user then chooses one of the sub-expressions *E*. The system then applies operator *O* to transform expression *E* to a new expression *E'*. A new version *V'* is created by substituting *E'* for *E* in *V*. Finally the new version *V* is added to the set *Versions(F)*.

The *GSA* and reformulation operator definitions are found in Appendices F and G respectively. A total of 39 operators are defined. These include 10 *GSA* operators and 29 reformulation operators. Only 29 operators were actually used in the hearts and scheduling domains. These included 8 *GSA* operators and 21 reformulation operators. The declarative pattern language was used to implement 33 of the 39 operators. A procedural implementation was used to implement the remaining 6 operators. In particular, procedural methods were used for operators implementing *Fold* and *UnFold* operations, described in Section 2.6 below. A procedural method was also used to implement one version of the *Function Invariance (FI) GSA*.

POLLYANNA maintains a record of the derivation used to generate approximations. For each new version *V'*,

[13]The approximation generation search space is commutative in the sense described in [Nilsson 80], (page 35). For this reason only the current problem state needs to be maintained. Backtracking is never required.

the record indicates the parent version *V* from which *V'* was created. The record also indicates the operator *P* that created *V'* and any instantiation data used in applying *P* to *V*. The derivation record is useful for two reasons. For each function version, the record provides a means of tracing a series of parent links. The parent links can be used to determine the parts of the initial theory on which the given version depends. The parent links might therefore be used to help update an approximation in the face of changes to the initial theory. (See Section 2.2.5.) The derivation record is also important for the process of generating a theory space, to be described in Chapter 4.

2.3.3. Local v. Global Theory Operations

POLLYANNA's approximation generator can be analyzed in terms of a distinction between *local* and *global* operations. *POLLYANNA* uses purely local operations in the *approximation generation* phase. All of the *GSA* and reformulation operators are applied directly to individual function definitions. When an operator *P* is applied to version *V* of function *F*, the operation proceeds without reference to other functions that interact with *V*. Other functions may call *F* or be called by version *V* of function *F*. These other functions are not involved in the operation performed on version *V*. One can imagine alternative implementations based on global operations. Global operators would apply to entire theories, rather than individual functions. Were *POLLYANNA* to use global operations, there would no longer be a distinction between generating individual approximations and generating complete approximate theories. The distinction between *approximation generation* and *theory space generation* would be blurred.

Advantages and disadvantages result from using purely local operations. The chief advantage concerns the efficiency of the learning process. Local operations permit operator sequences to be compiled. They need not be repeated when individual approximations are combined to form complete approximate theories, during the *theory space generation* phase. Suppose several theories, T_1 ... T_n, are generated using operator sequences S_1...S_n respectively. If all of the sequences S_i share a common subsequence *S*, the sequence *S* needs to be applied only once. It need not be repeated for each distinct theory.[14]

The chief disadvantage involves predicting the effects of operators, particularly *GSA* operators. The *theory space search* (*TSG*) module faces the task of partially ordering approximate theories according to efficiency. The partial order involves comparisons of the global efficiency of theories, rather than the local efficiency of individual functions. In order to make such comparisons, the *TSG* module must reason about the impact of local operations on

[14]Perhaps an even greater efficiency results from separation of the *theory space search* phase from the two earlier *approximation generation* and *theory space generation phases*. Approximate theories are fully compiled in advance of empirical testing. Operator sequences therefore need not be repeated for each training example against which a theory is tested.

the global efficiency of theories. As will be demonstrated in Chapter 4, this requires the *TSG* module to make assumptions about the manner in which different functions interact with each other.

2.3.4. Classification of Knowledge in *POLLYANNA*

Interesting insights result from classifying the knowledge used to generate approximations in *POLLYANNA*. The knowledge can be classified along several different dimensions, as shown in Figure 2-7. One dimension distinguishes between *operators*, which are used during the *approximation generator* phase, and *functions*, which are evaluated during the empirical *theory space search* phase. Operators and functions thus correspond to the distinction between compile time and run time computations. Another dimension involves the distinction between *domain dependent* knowledge, such as the initial intractable theory, and *domain independent* knowledge that is a permanent part of the system. A third dimension involves the distinction between *correct* and *approximate* knowledge.[15]

The result of fitting *POLLYANNA* into this classification scheme is shown in Figure 2-7. Notice that four of the eight combinations are actually used in *POLLYANNA*. Three of the four have been discussed before: the intractable theory, the generic simplifying assumptions and reformulation knowledge. *POLLYANNA* also uses a collection of primitive function definitions. These primitive functions define the language in which domain theories must be written for use in *POLLYANNA*. These functions will be described in Section 2.4.

COMPILE TIME KNOWLEDGE (Operators):

	DOMAIN INDEPENDENT:	DOMAIN DEPENDENT:
CORRECT:	Reformulation Operators	?
APPROXIMATE:	GSA Operators	?

RUN TIME KNOWLEDGE (Functions):

	DOMAIN INDEPENDENT:	DOMAIN DEPENDENT:
CORRECT:	Primitive Functions	Intractable Domain Theory
APPROXIMATE:	?	?

Figure 2-7: Classification of Knowledge in *POLLYANNA*

The classification table in Figure 2-7 shows four blank spaces. These blanks indicate ways in which

[15]Two additional dimensions include (1) Declarative knowledge v. control knowledge and (2) Domain-level knowledge and meta-level knowledge.

POLLYANNA might be extended to use additional types of knowledge. Notice that *POLLYANNA* does not use any domain dependent approximate knowledge at all. The architecture could easily incorporate such knowledge. The user need simply supply approximations that he would like the system to use in the course of generating heuristics. These could be coded in the form of approximate function definitions, to be used at run time, or domain specific approximation operators, to be used at compile time. Another extension would allow the user to provide correct, domain specific operators to be used at compile time. These would define preferred reformulations that he would like the system to use. A final extension would add domain independent approximate knowledge to be used at run time. *POLLYANNA* would then include functions that implement generic simplifying assumptions, as well as operators. This analysis indicates considerable flexibility in the *POLLYANNA* architecture that has not yet been exploited. Were it desirable, any of the blank spaces could be filled by the builders of knowledge bases used in *POLLYANNA*.

2.3.5. Theories, Classes and Generic Functions

Complete approximate theories are implemented in *POLLYANNA* using the techniques of object-oriented programming [Cardelli 85; Stefik and Bobrow 86]. Each candidate approximate theory is implemented as a distinct *class*. The definition of a class specifies which version is used for each function appearing in the final state of the approximation generator. Each function in the final state is implemented as a *generic* function. Every generic function takes one extra argument indicating the class or theory to be used in evaluating the function, i.e., $f(x)$ becomes $f(t,x)$. The theory or class argument t implicitly specifies the version of each function to be used in the course of evaluation.

Generic functions are a useful representation for several reasons. To begin with, many of the advantages that have made generic functions useful in software engineering carry over to the present context. Generic functions are facilitate orthogonality, in the sense that theories remain well formed when arbitrary combinations of versions are combined. They facilitate modularity by decomposing a domain theory into parts with relatively simple interactions. Generic functions also provide simple hooks for inserting hand-coded assumptions, should the need arise.

The generic function framework is more general than might appear at first. Were the concept of "generic function" replaced with "generic predicate", the formalism could be extended to apply to Horn clause theories as well as theories written in terms of lambda expressions. By speaking in terms of "generic subproblems", the formalism could be generalized to include arbitrary AND/OR spaces or any sort of problem solving by hierarchical decomposition.

2.4. Representation of Intractable Theories

2.4.1. The *PT-FORMALISM*: An Abstract Class of Domain Theories

A definition of the *PT-FORMALISM* for representing domain theories is presented in this section. The definition specifies which types of constructs are allowed in theories designed for use in *POLLYANNA*. The *PT-FORMALISM* was specifically designed to interface with the *PT-GSA* family of generic simplifying assumptions. It indicates the range of theories to which *PT-GSAs* are *formally* applicable. Analysis of the *PT-FORMALISM* provides support for the claim of *formal* generality of *PT-GSAs* (Claim #2 in Section 1.6.3). The claim of *useful* generality is supported only by experimental results.

The *PT-FORMALISM* is described in Figure 2-8. In order to represent a theory in this formalism, one must specify a finite event space *E* and provide a means of computing a probability distribution *D(e)* defined over the event space. The event space *E* represents the basic random variables in the domain theory. An event space could be the set of possible values of a single variable. It could also be a cartesian product representing the range of a collection of variables. In the hearts domain, the event space *E* is taken to be the set of all possible initial deals. In the scheduling domain, the event space corresponds to the set of all admissible schedules. Additional random variables may be defined as functions of the event space *E*. Each additional random variable is defined by a λ-expression. A set of deterministic functions may also be defined using λ-expressions. The λ-expressions must be written in a restricted language. Each λ-expression may use only the purely declarative constructs of *function composition*, *recursion* and *explicit transformation*, i.e. permuting or duplicating arguments. The λ-expressions may refer to random or deterministic functions defined by λ-expressions, or to primitive functions.

- A finite event space *E*.
- A probability distribution $D{:}E \rightarrow [0,1]$.
- A set of functions with finite domains and ranges:
  - Random variable functions, $r_1,...,r_n$, each of which takes the random event variable *e* as an argument.
  - Deterministic functions, $d_1,...,d_m$, each of which does not take the event variable *e* as an argument.
- A set of λ-expressions providing a definition for each function.
  - Definitions involve the operations of function composition, recursion and/or explicit transformation.
  - Definitions refer only to defined random or deterministic functions or to primitives.
  - Primitives include $Prob[\lambda(e)B(e)|G]$ and $Exp[\lambda(e)N(e)|G]$, among others.

Figure 2-8: The *PT-FORMALISM* for Representing Domain Theories

The primitive functions permitted in the *PT-FORMALISM* are listed in Appendix E. The primitives include standard *LISP* functions, reduction of binary operations, functions to manipulate sets represented as lists, and functions to compute probabilities and expectation values. Several primitives used to compute probabilities and expectation values are worth mentioning. The notation *Exp[λ(e)N(e)|G]* signifies the expectation value of some numeric function *N(e)*, viewed as a function of the event space. Likewise *Prob[λ(e)B(e)|G]* is the probability that a boolean function *B(e)* is true. Both *Prob* and *Exp* have a second argument indicating the *givens* of a conditional expectation or probability. The *givens* describe constraints on the event space to be used in computing conditional probabilities and expectation values. A *given* structure is conceptually a list of pairs. Each pair associates a random variable with a list of possible values. The random variable is constrained to take on one of the values on the list. Random variables not appearing on the list are free to take any value in their ranges. Other primitives used to compute probabilities and expectation values include: *Known*, *Evaluate*, *Given* and *And-Givens*, that are associated with computations of probabilities and expectation values. *Known[v|G]* tests whether the value of variable *v* is directly specified in the givens *G*. *Evaluate[v|G]* returns the value of variable *v*, if that value is directly specified in the givens *G*. *Given(v,w)* creates a given structure that specifies *w* as the value of variable *v*. *And-Givens(G1,G2)* creates a given structure that is the logical conjunction of givens *G1* and *G2*. These functions are described in Appendix E.

The primitives and construction rules for writing λ-expressions are important for the successful operation of the approximation generator in *POLLYANNA*. They guarantee that certain type of reformulation operators will applicable, i.e., those that reformulate probabilities and expectations of composite functions. (See Section 2.6.2.) Those reformulation operators played a crucial role in generating the hearts and scheduling heuristics described in Chapter 3. They are also required for successful implementation of the automatic control strategy proposed in Section 2.7.

The primitives and construction rules also specify the representation language for approximate theories generated by the *approximation generation* and *theory space generation* modules in *POLLYANNA*. These approximate theories are executed during the *theory space search* phase. In order for approximate theories to run, all the primitives used in approximate theories must be supported in *POLLYANNA*'s run time environment. The run time environment therefore includes definitions of the unusual primitives such as *Known, Evaluate, Given* and *And-Givens*, that are associated with computations of probabilities and expectation values.

2.4.2. A Theory of Hearts

2.4.2.1. Issues in Formulating a Hearts Theory

Several issues must be addressed in order to represent a theory of hearts. The hearts domain suffers from problems of incomplete knowledge as well as the problem of intractability. Two different problems of incomplete knowledge occur in hearts. In any given game situation, players have incomplete information about the state of the game. A player's information is incomplete because he does not know the cards in his opponents' hands. In addition to this problem of incompleteness, another occurs as well. Even if players knew each other's hands, they would be unable to predict each other's behavior. This problem occurs because hearts is played by humans. In the absence of a model of human behavior, it is not possible to predict one's opponents' card choices. For the purpose of studying the intractable theory problem, it is desirable to have a hearts domain theory that is complete. (See Section 1.2.2.) Methods of dealing with both types of incompleteness will be discussed below.

The problem of human behavior can be remedied by assuming a computational model of human players. Such a model could take the form of a function *Choice*. This function would take as input a parameter defining the current game *state*, i.e., all the publicly visible information, as well as a description of the player's *hand*, i.e., the player's private information. The function would return the card choice that the player would make in the given *state*, assuming he held the specified *hand*. In order to implement such a function, one must make assumptions about human behavior. One approach would assume optimal play by each player.[16] An alternate approach would use a model describe sub-optimal behavior by each player. The hearts theory used in *POLLYANNA* takes this latter approach. Although the theory does include a computational model each player, the model is not claimed to describe optimal card playing behavior.

In order to deal with the problem of incomplete information, hearts can be cast into the framework of probability theory. Although opponents' specific card choices cannot be predicted without knowledge of their hands, the probability of each card choice may nevertheless be computable. Consider first what would happen if the initial card deal were somehow known. A function *card(p,t,d)* can be defined in the following way to compute the card played by player *p* in trick *t* assuming *d* is the initial deal. The function *card* operates by using *choice* to simulate the history of the game. The initial game state is defined by the initial deal *d*. Each subsequent state can be computed using the *choice* function to determine what card was played, and updating the game state description. *Card* returns the value computed by *choice* upon reaching the state corresponding to player *p*'s turn in trick *t*.

---

[16]Optimality could be defined in the game theoretic sense. An optimal strategy against optimal opponents would correspond to an equilibrium four-tuple in the four dimensional matrix of hearts game strategies [Luce and Raiffa 57]. Not all games have an equilibrium solution. Others have more than one equilibrium solution. This approach might or might not be feasible in the game of hearts, depending on whether there exists a unique equilibrium solution.

After defining the functions *Choice* and *Card*, the probabilities of player's card choices could be computed in the following way: Let *Prob[λ(d)c=card(p,t,d)|g]* represent the probability that player *p* plays card *c* in trick *t* given information *g*. The given parameter *g* includes both the public information visible to everyone and the private information known to the player carrying out the computation. The computation begins by determining which of the many possible deals are consistent with information *g*, i.e., which deals lead to games consistent with the public information, and which deals define the computing player's hand to be consistent with his private information. Such consistency tests are possible because the entire game history is computable in principle from the initial deal alone through repeated use of the *Card* and *Choice* functions. After determining the set of consistent deals, the probability of card choice *c* is determined by assuming each of the consistent deals is equally likely. *Prob[λ(d)c=card(p,t,d)|g]* is just the fraction of the consistent deals for which *card(p,t,d)* equals *c*. Expectation values of functions that depend on the values of *Card* can in principle be computed by similar methods.

2.4.2.2. Representation of the Hearts Theory

A portion of a computational hearts theory is shown in Figure 2-9. This theory was designed to implement to overall computation strategy described above. It describes a function *Choice(p,t,hand,state)* that a tells a player *p* what card he should play in trick *t*, given his *hand* and a description of the current game *state*. This function operates by minimizing the evaluation function *Exp-Game-Score* over all the *Legal-Choices*. The evaluation function *Exp-Game-Score* computes the conditional expectation value of the random variable *Game-Score*, assuming that a given card *c* is chosen for play. The random variable *Game-Score* is expressed as a function of the random variables *Trick-Score*, *Win* and *Trick-Value*. These in turn are written as functions the random variable *Card* and other random variables as well. The structure of relationships among random variables is summarized in Figure 2-10. Each random variable is ultimately a function of the initial game deal *d*, which represents the underlying probabilistic event space.

The functions in Figure 2-9 are represented in *POLLYANNA* using purely declarative λ-Expressions. In addition to the λ-Expressions, *POLLYANNA* requires one additional type of information. For each random variable in the domain theory, the system must be provided with a list of values representing the range of that variable. The value range is used by the functions *Prob[]*, *Exp[]* and is also required by some of the generic simplifying assumptions, as described in Section 2.5. The hearts theory shown in Figure 2-9 is a slightly simplified version of the complete hearts theory found in Appendix C.

A careful reading of the definitions in Figure 2-9 suggests a circularity. The function *Choice(p,t,d)* appears to indirectly call *Choice(p,t,d)*. The circularity is potentially avoided by suitable definitions of the functions *Exp* and *Prob*, which compute probabilities or expectation values of variables appearing in the random variable tree in Figure

```
Choice(p,t,hand,state) =
 =  Minimize( λ(c) Exp-Game-Score(p,t,hand,state,c),
              Legal-Choices(p,t,hand,state) )

Exp-Game-Score(p,t,hand,state,c) =
 = Exp[λ(d) Game-Score(p,d) | Next-State(p,t,hand,state,c)]

Game-Score(p,d) = ∑ (t in TRICKS) Trick-Score(p,t,d)

Trick-Score(p,t,d) = If Win(p,t,d) then Trick-Value(t,d) else 0

Trick-Value(t,d) = Point-Value(Card(Tom,t,d))
                   + Point-Value(Card(Dick,t,d))
                   + Point-Value(Card(Harry,t,d))
                   + Point-Value(Card(Moe,t,d))

Win(p,t,d) =
  = Defeats(Card(p,t,d),Card(Left(p),t,d),Lead-Suit(t,d))
    ∧ Defeats(Card(p,t,d),Card(Right(p),t,d),Lead-Suit(t,d))
    ∧ Defeats(Card(p,t,d),Card(Across(p),t,d),Lead-Suit(t,d))

Card(p,t,d) = Choice(p,t,Hand(p,t,d),Game-State(p,t,d))
```

Figure 2-9: Definitions of Some Hearts Functions

2-10. For example, consider a computation of *Prob[λv=f(e)|G]* the probability that *f* has value *v* given information *G*. The *Prob* function would first check to see whether a value for *f* is directly specified in the givens *G*. If so, the answer is zero or one, depending on whether *f* is assigned a value of *v* in *G*. If not, then the definition of *f* would be unfolded in order to express the probability computation in terms of new variables lower than *f* in the random variable tree. The new variables would then be checked against the givens, unfolded and so on. The process would terminate when the unfolding process encounters variables defined in the givens *G*, or the basic underlying random variable *d* representing the initial deal. Variables apparently defined in terms of themselves would not cause infinite loops because of the manner in which *Prob* and *Exp* check the given values *G* before unfolding function definitions. (See Section 2.4.4 regarding the feasibility of testing or verifying this solution to the problem of circularity.)

The hearts theory described in Figure 2-9 and Appendix C is *absolutely* intractable and *non-deterministically* intractable. (See Section 1.5.1.2.) The theory is absolutely intractable because the function *Exp-Game-Score* cannot be evaluated using computational resources that are available in any realistic context of learning. It is non-deterministically intractable because neither verifying the correctness of a card choice, nor verifying the value of *Exp-Game-Score* is feasible within realistic resource constraints. Explanation-based generalization (*EBG*) does not apply to theories that are absolutely, non-deterministically intractable. It therefore cannot be applied to the hearts theory presented here.[17] This observation partially supports the claim that *POLLYANNA* applies to domain theories

---

[17] *EBG* fails to apply for a more superficial reason as well. It operates only on domain theories formulated in Horn clauses [Mitchell et al. 86]. The present hearts theory is written in terms of λ-expressions. Nevertheless it could be translated into Horn clauses in a relatively straight forward manner.

Figure 2-10: Hearts Random Variable Tree

not handled by explanation-based generalization (Claim #6 in Section 1.6.7). *EBG* might nevertheless apply to some other theories of the hearts domain.

2.4.3. A Job Scheduling Theory

A job scheduling domain was used as a secondary test bed to demonstrate the generality of *POLLYANNA*. A specification of the type of problem solved by the scheduling theory is shown in Figure 2-11. A problem instance consists of a collection of jobs, working times, deadlines and job values. The working time of a job represents the amount of time required to complete the job, once work on the job has begun. The deadline of a job represents a point in time by which the job must be completed. The value of the job is simply a measure of the importance of the job. The specification also includes a list of precedence constraints indicating that some jobs are preconditions for other jobs. The precedence constraints define a graph representing the activity network of the scheduling problem. A solution requires finding the optimal uniprocessor schedule. Each schedule is evaluated according to the total value of the jobs completed on time without violation of precedence constraints. This scheduling problem is technically known as "precedence constrained scheduling with individual deadlines" and has been shown to be *NP*-Complete [Garey and Johnson 79].

Given:

- A list of jobs: $J_1,...,J_n$.
- A working time for each job: $W_1,...,W_n$.
- A deadline for each job: $D_1,...,D_n$.
- A value for each job: $V_1,...,V_n$.
- A list of precedence constraints of the form: (J_a,J_b), indicating that job J_a is a precondition for job J_b.

Find: A schedule meeting two requirements:

- Admissibility: The schedule assigns a starting time $T_i \geq 0$ to each job. For each job J_i allocated starting time T_i, no other job is allocated a time in the interval: $[T_i,T_i+W_i)$.
- Optimality: Each admissible schedule is assigned a value. The value is equal to the sum of the values of jobs completed successfully and on time. A job J completes successfully provided all the preconditions of J complete successfully by the time J starts. A job finishes on time provided it meets its deadline. An admissible schedule is optimal provided it has a value equal to or greater than all other admissible schedules.

Figure 2-11: Job Scheduling Problem Definition

A computational scheduling theory is found in Appendix D. The theory is implemented in terms of a probabilistic formalism similar to the one used to represent the hearts domain theory. The functional relationships among random variables in the scheduling theory are illustrated in Figure 2-12. The *Value* of a schedule is computed as the sum of values of individual jobs. The *Job-Value* of an individual job is computed as the product of the three variables *Penalty*, *Preconditions* and *Weight*. The *Penalty* factor is zero or one depending on whether the job completes on time. *Penalty* thus depends on *Completion-Time,* which itself depends on the *Allocation-Time* of a job and the *Working-Time* for that job. The *Preconditions* factor is zero or one depending on whether all the preconditions job *j* are successfully completed by the time job *j* is allocated. A precondition *p* for job *j* is satisfied only if the *Completion-Time* of *p* is no later than the *Allocation-Time* of *j* and the preconditions of *p* itself are also satisfied. The *Weight* of a job simply reflects the importance of that job. The underlying event space is described by a collection of random variables $\lambda(e)Choice(t,e)$ representing the choice of job assigned to each time slot.

The scheduling theory is defined in terms of a function *Evaluation(i,t,g)*, which can be used to order states in the scheduling search space. The parameter *i* represents a particular instance of the general scheduling problem. The

Figure 2-12: Job Scheduling Random Variable Tree

parameter t represents the first free time slot in the current problem state. The parameter g encodes the current problem state, which represents a partially or completely specified schedule. Problem states corresponding to complete schedules are specified by assigning a job to the random variable $\lambda(e)Choice(t,e)$ for each time slot t. Problem states corresponding to partial schedules are specified by assignments of jobs to some, but not all time slots. All problem states are represented givens g that can be used in computing probabilities $Prob[\lambda(e)B(e)|g]$ or expectation values $Exp[\lambda(e)N(e)|g]$. (See Section 2.4.1.)

The function *Evaluation(i,t,g)* returns the *Value* of the best complete schedule that is consistent with the partially or completely specified schedule *g*. It operates by generating all complete schedules *g*' that are consistent with schedule *g*. Each complete schedule *g*' is evaluated by computing *Exp[λ(e)Value(i,e)|g']*, the expectation of the random variable representing the *Value* of the schedule. Since *g'* is a complete schedule, all the underlying random variables λ*(e)Choice(t,e)* are specified in state *g'*. The expectation *Exp[λ(e)Value(i,e)|g']* is therefore equal to the actual *Value* of the complete schedule *g*'. By maximizing this expectation over all complete schedules *g*' that are consistent with *g, Evaluation(i,t,g)* returns the *Value* of the best complete schedule consistent with schedule *g*.

The scheduling theory has asymptotic complexity that is exponential in the number of time slots to be allocated. The computational efficiency of the theory therefore depends on the size of the particular scheduling problem instance to be solved. Notice that the scheduling theory involves optimization, rather than satisficing. Verification of optimality presumably requires systematic examination of all possible solutions. For this reason, the task of verifying the optimality of solutions is probably no easier than the task of finding them in the first place. The scheduling theory will therefore likely be non-deterministically intractable in any context for which it is deterministically intractable.

The scheduling theory shows how a deterministic search problem can be cast into a probabilistic formalism. The key step in this process involves identifying problem states in a search space with the givens that appear in computations of probabilities and expectation values. Variables that constitute the search problem state description are identified with the underlying random variables of a probabilistic event space. State evaluations are performed by generating the leaves of a search tree and computing the expectation value of some optimization criterion at each leaf. Once a deterministic search problem is cast into such a formalism, generic simplifying assumptions from the *PT-GSA* family become applicable. Results from experimenting with the scheduling theory will show that scheduling heuristics can be generated by applying *PT-GSAs* to the scheduling theory.

2.4.4. Testing and Verification of Intractable Theories

Intractable domain theories present particular debugging problems for knowledge engineers. After encoding a computational theory, a knowledge engineer normally tests his representation to determine whether it generates correct answers, or whether it is even executable at all. If the domain theory is relatively (but not absolutely) intractable, it can be executed against test data without exceeding available resources. (See Section 1.5.1.3.) If the domain theory is absolutely intractable, it cannot be executed even once. Direct testing and debugging procedures are therefore not feasible. Proofs of program termination and functional correctness would useful; however, such proofs tend to be extremely difficult for theories of any complexity.

Neither the hearts domain theory, nor the scheduling domain theory was empirically tested for executability or correctness. Nor was either analytically proved to terminate or to be functionally correct. In the absence such empirical tests and analytic proofs, neither theory is guaranteed to be either operationally usable or functionally correct. This limitation is not necessarily a problem. The initial theories are not designed in order to be directly executed. They need only generate approximations that are both operationally usable and that meet accuracy and efficiency goals. These properties of approximate theories *can* be tested in the empirical component of *POLLYANNA*.

2.5. The *PT-GSAs*: A Family of Generic Simplifying Assumptions

The *PT-GSA* family of generic simplifying assumptions is described in Figure 2-13. Three types of *GSAs* are listed: *Function Invariance (FI), Equiprobable Random Variables (EP)* and *Probabilistic Independence (IN)*. Each type includes several distinct versions. The *Function Invariance* group includes assumptions asserting functions or algebraic expressions to be constants. The *Equiprobable Random Variables* group includes assumptions asserting that various sets of alternatives are equally likely. The *Probabilistic Independence* group includes assumptions asserting that sets of random variables are probabilistically independent. The definitions of all *PT-GSA* operators are found in Appendix F.

Function Invariance (FI):

1. $(\forall x)\ F(x) = \text{Constant}$
2. $(\forall x)\ F(x) = F(\text{Constant})$
3. $A + B = A + 0$
4. $A * B = A * 1$

Equiprobable Random Variables (EP):

1. $\text{Prob}[\lambda(e)\ B(e) \mid G] = 1 / 2$
2. $\text{Prob}[\lambda(e)\ v = F(e) \mid G] = 1 / |\text{Range}(F)|$
3. $\text{Exp}[\lambda(e)\ N(e) \mid G] = \text{Average}(\text{Range}(N))$

Probabilistic Independence (IN):

1. $\text{Prob}[\lambda(e)\ P(e) \wedge Q(e) \mid G] =$
   $= \text{Prob}[\lambda(e)\ P(e) \mid G] * \text{Prob}[\lambda(e)\ Q(e) \mid G]$
2. $\text{Exp}[\lambda(e)\ M(e) * N(e) \mid G] =$
   $= \text{Exp}[\lambda(e)\ M(e) \mid G] * \text{Exp}[\lambda(e)\ N(e) \mid G]$

Figure 2-13: The *PT-GSA* Family of Generic Simplifying Assumptions

The *PT-GSAs* are specifically suited to interface with domain theories written in the *PT-FORMALISM*. They apply to computations involving functions, probabilities and expectation values. The family was formulated through a process of reconstructing known heuristics drawn from the hearts domain. (See Section 2.2 above.) Verbal descriptions of specific simplifying assumptions that are instances of *PT-GSAs* are shown in Figure 2-14. Some of these specific assumptions are taken from the derivations of the *Dump High Rank Rule* and the *Dump High Point Value Rule*, found in Appendix A. Others are drawn from the scheduling domain. Longer lists of simplifying assumptions are found in Appendices K and M. These appendices provide verbal paraphrases of specific simplifying assumptions generated *mechanically* by *POLLYANNA* in the hearts and scheduling domains. A discussion of hearts and scheduling heuristics generated by *PT-GSAs* is found in Chapter 3.

Function Invariance (FI):

- The expected trick value is a constant, equal to the average value of a trick.
- The probability of winning the trick is a constant, equal to the average for all players.
- The working time for each job is a constant, equal to one time unit.

Equiprobable Random Variables (EP):

- The player is equally likely to play any card remaining in his hand.
- An opponent is equally likely to play any card in the deck.
- The lead suit is equally likely to be any suit.
- Any job not yet allocated is equally likely to be assigned to any of the remaining time slots.
- The preconditions of each job are equally likely to be satisfied or unsatisfied.

Probabilistic Independence (IN):

- The probability of winning the trick is statistically independent of the trick value.
- The four card choices and the lead suit are statistically independent.
- The probability that a job will complete on time is independent of whether its preconditions will be met.

Figure 2-14: Instances of *PT-GSAs* in Verbal Form

2.5.1. Function Invariance

Assumptions of *Function Invariance (FI)* are used to avoid computing costly functions or algebraic expressions. *FI* is typically applied to functions performing intermediate computations, the results of which are not a final answer, but which are used in subsequent computations in order to determine a final value. Given a theory describing a function $F(x)=G(H_1(x),...,H_n(x))$, an *FI* assumption might be applied to assume $H_i(x)$ is a constant. Such an assumption is likely to be useful when the following conditions are met: (1) The function $H_i(x)$ is expensive to compute; (2) The value of $H_i(x)$ does not vary much when x changes; (3) The function $G(...,H_i(x),...)$ is not very sensitive to the value of $H_i(x)$. When these conditions hold, use of an *FI* assumption will likely result in a large gain in efficiency, with a minimal sacrifice in accuracy.

Four different versions of *Function Invariance (FI)* are shown in Figure 2-13. The versions are closely related, but have slightly varying semantics. The most general version (#1) allows replacing a reference to function *F* with any constant. In the *POLLYANNA* implementation, the particular constant is chosen interactively, by a human user. A related version (#2) requires the constant to be in the range of *F*. It also requires computing function *F* at least once, if the range of *F* is not known.[18] Two specialized versions (#3 and #4) apply to sum and product expressions. They replace sub-expressions with the algebraic identity elements of the top level operation.[19]

A particularly interesting application of *Function Invariance (FI)* occurs when *F(x)* is an evaluation function, i.e., a function used to select an optimal member of from a set of possible choices, by finding one with a minimal (or maximal) value. Suppose an evaluation function has the form $F(x)=G(H_1(x),...,H_n(x))$. Each of the sub-expressions *Hi(x)* represents a distinct "factor" that bears upon the choice of an appropriate value of *x*. When *FI* is used to make $H_i(x)$ a constant, the evaluation function *ignores* the factor $H_i(x)$ and makes a decision based only on the remaining factors. For example, the derivation of the *Dump High Rank Rule* uses *FI* to ignore the expected value of future tricks. (See Figure 2-14 and Appendix A). The resulting evaluation function makes decisions based on the odds of winning future tricks. Likewise, the derivation of the *Dump High Point Value Rule* uses *FI* to ignore the odds of winning future tricks. (See Figure 2-14 and Appendix A). The resulting evaluation function makes decisions based on the expected value of future tricks.

Evaluation functions often permit considerable flexibility in choosing a specific constant to approximate a function. Suppose the function $F(x)=G(H_1(x),...,H_n(x))$ is used to evaluate a set of alternatives, $\{x_1,...,x_m\}$, by selecting an x_i giving the highest value of *F*. When used in this manner, the specific values of $F(x_i)$ do not matter. Only the rank order is important. Depending on the properties of the function *G*, the specific constant used to approximate $H_i(x)$ may not influence the rank order of choices. For example, if *G* is simply a sum over *n* terms, the rank order of choices does not depend on the choice of a constant used to approximate one of the terms. If *G* is a product over *n* factors, only the sign of the chosen constant is relevant. When the choice of a constant does not impact the semantics of the resulting evaluation function, algebraic identity elements (zero or one) are natural choices for two reasons. They minimize computation by avoiding one extra binary operation. They also allow a reformulation that simply drops $H_i(x)$ from the entire expression, producing a new expression that is easier to understand. Algebraic identities are used in the third and fourth versions of *FI* in Figure 2-13.

[18] In the context of *POLLYANNA* there is no semantic difference between the first and second versions of *FI*. Using the *Fold* reformulation operator, version #2 can be made just as powerful as version #1. First *Fold* replaces *F(x)* with *Id(F(x))*, defining *Id* to be equal to the identity function. Then the second version of *FI* replaces *Id(F(x))* with *Id(Constant)*. It also makes no difference whether *FI* is defined to apply to *functions*, as in the first two versions, or to *expressions*, as in the second two versions. The *Fold* and *UnFold* reformulation operators can convert arbitrary expressions into function calls and vice versa. These reformulation operators are described in Section 2.6.

[19] The implemented version of *FI* allows the interactive user to specify an arbitrary constant; however, all results for the hearts and scheduling domains were obtained by following a policy of selecting algebraic identities as the constant values of functions. The same results could therefore have been obtained by implementing versions #3 and #4 to carry out the same policy.

Assumptions similar to *Function Invariance (FI)* have been used elsewhere in Machine Learning. Keller's MetaLEX system uses two generic simplifying assumptions called *Truify* and *Falsify* [Keller 87]. These two operations are used to simplify boolean expressions by replacing sub-expressions by one of the constants *T* or *F*. *Truify* and *Falsify* are most closely related to the second and third versions of *Function Invariance* shown in Figure 2-13. Both can be used to replace a subexpression with the identity element of a binary operator. Application of *Truify* to B in $A \wedge B$ leaves the expression equal to A. Application of *Falsify* to B in $A \vee B$ leaves the expression equal to A. In each case, one subexpression is effectively ignored and dropped from the computation. Another approximation similar to *Function Invariance (FI)* is used to ignore negligible terms in a sum [Bennett 87].

A operation called "Freeze" has been described by Mostow and Fawcett [Mostow and Fawcett 87]. *Freeze* can replace any expression or sub-expression by any constant. In order to maintain data type constraints, *Freeze* might be required to replace an expression with a constant in the range of the top level operation in the expression. Thus integer expressions would be replaced by integer constants, and likewise for real and boolean expressions, etc. Mostow and Fawcett do not actually mention such a constraint. Depending on whether this constraint is imposed, *Freeze* is equivalent to either the first or second version of *Function Invariance* in Figure 2-13.

2.5.2. Equiprobable Random Variables

Equiprobable Random Variables (EP) assumptions are roughly equivalent to the *Principle of Insufficient Reason*, first formulated by Jacob Bernoulli (1654 - 1705) in the Ars Conjectandi [Jaynes 83a]. Bernoulli's principle suggests that each member of a set of mutually exclusive alternatives be considered equally likely, in the absence of any evidence to the contrary. His principle is put to a novel use in the present context. It was originally formulated to deal with lack of knowledge that would favor one alternative over another. In its original form, it applies to the incomplete theory problem. This research has applied Bernoulli's principle to the intractable theory problem. Complete intractable theories contain all the information needed to decide among the alternatives. Computational resource limits render the information unavailable. In this context, Bernoulli's principle might be renamed the "Principle of Intractable Reason".

Assumptions of *Equiprobable Random Variables (EP)* are used to simplify computations of probabilities and expectation values. *EP* assumptions assert that a random variable *F(e)* is equally likely to take any of the values in its legal range, *Range(F)*. A system using *EP* assumptions can avoid computing probability distributions, when computing such distributions would be computationally expensive. Instantiations of *EP* in the hearts and scheduling domains are shown in Figure 2-14. Each of the three hearts assumptions is used in the derivations of both the *Dump High Rank Rule* and the *Dump High Point Value Rule*. (See Appendix A.)

Three different versions of *EP* are shown in Figure 2-13. The first version simplifies computing probabilities of boolean functions *B(e)*. Since boolean functions have a range of size two, *{T,F}*, this version asserts that *B(e)* is likely to be true with probability one half. The second version simplifies computing the probability that an arbitrary random variable *F(e)* will take on a specific value *v* in *Range(F)*. Assuming that all values in *Range(F)* are equally likely, the probability is just *1/|Range(F)|* for each specific value. The last version applies to expressions computing expectation values of numeric random variables *N(e)*. Assuming that all values in *Range(N)* are equally likely, the expectation value is just an average over the range of *N(e)*.

Equiprobable assumptions have often been criticized on the grounds of ambiguity [Luce and Raiffa 57], (page 284). Various different results can be obtained depending on how one defines a set of alternatives that are considered equally likely. The ambiguity can be illustrated using the random variable tree in Figure 2-10. Each node in the tree represents a different random variable appearing in the hearts domain theory. Each node also represents a distinct opportunity to formulate an *EP* assumption asserting that all values of the random variable are equally likely. Application of *EP* to *Game-Score* generates a uniform distribution for that variable. Application of *EP* to *Win* and *Trick-Value* leads to uniform distributions for those variables. A uniform distribution for *Win* and *Trick-Value* implies a different, non-uniform distribution for *Game-Score*. Thus different results are obtained depending on the nodes in the variable tree to which *EP* is applied. When *EP* is applied to nodes near the root of the tree, a highly efficient but inaccurate theory is the result. When *EP* is applied to nodes deeper in the tree, the resulting theory is less efficient, but potentially more accurate. Chapter 3 will illustrate how the resulting approximate theories depend on the random variables to which *EP* is applied.

The ambiguity of *EP* does not present the same problem in the context of *POLLYANNA* that it presents in other contexts. *POLLYANNA* uses *EP* assumptions only for generation of approximate theories. The approximations are then tested empirically before they are accepted. Empirical testing serves to filter out inaccurate *EP* assumptions. Ambiguity may even be desirable in the context of *POLLYANNA*. Different applications of *EP* lead to approximate theories with different levels of efficiency. A particular level of efficiency can be chosen to suit the performance context. Ambiguity may also enhance the goal of accuracy. Accurate *EP* assumptions are more likely to be found when *EP* is applied systematically to a large set of random variables.

The range of possible instantiations of *EP* is greatly influenced by the specific representation of domain theories. Depending on the actual random variables used to formalize a domain, application of *EP* can lead to a variety of different results. Consider the following example from hearts. Suppose $\lambda(d)Win(p,t,d)$ is a boolean random variable indicating whether player *p* wins trick *t*. Applying the first version of *EP* in Figure 2-13 to the expression $Prob[\lambda(d)Win(p,t,d)|G]$, one obtains the result that a specific player *p* has probability *1/2* of winning trick *t*. Now suppose that λ(d)*(d)Winner(t,d)* is a random variable equal to the player who wins trick *t*. The range of *Winner* is a set of four players. After applying the second version of *EP* in Figure 2-13 to the expression *Prob[λ(d)p=Winner(t,d)|G]*, one obtains the result that a specific player *p* has probability *1/4* of winning trick *t*. This example illustrates that the results of *EP* can depend on the detailed encoding of the initial domain theory.[20]

EP assumptions are unnecessarily strong for some situations. Each version of *EP* assumes that *all* values in the range of a variable are equally likely. Easily available information can sometimes eliminate values from the range of a variable, even when the complete distribution is too difficult to compute. For example, consider an instance of *EP* taken from Figure 2-14: *"An opponent is equally likely to play any card in the deck."* This assumption results from applying the second version of *EP* to the random variable *λ(d)Card(p,t,d)*, which has a range equal to the set *DECK*. It ignores the fact that many cards in the deck can be easily eliminated from consideration. A given player *p* knows that an opponent *p'* cannot play any card is already played or is held in player *p*'s own hand. This suggests assuming that all cards in this restricted set are equally likely. A more complex version of *EP* might be formulated to implement this line of reasoning. One approach would define a function *Restricted-Range(v,G)* to determine the values of variable *v* that have non-zero probability given *G*. Various approximate versions of *Restricted-Range(v,G)* would apply different amounts of effort to eliminate values in *Range(v)*. The new version of *EP* would then assert that all values in *Restricted-Range(v,G)* are equally likely.

2.5.3. Probabilistic Independence

Two versions *Probabilistic Independence (IN)* are shown in Figure 2-13. Both assert standard definitions of probabilistic independence. The first version asserts that the probability of a conjunction of boolean variables, $P \wedge Q$, is equal to the probability of *P* times the probability of *Q*. The second version asserts that the expectation value of a product of numeric variables *M* and *N* is equal to the expectation of *M* times the expectation of *N*. Example instantiations of *IN* in the hearts and scheduling domains are shown in Figure 2-14. Each of the two hearts assumptions is used in deriving both the *Dump High Rank Rule* and the *Dump High Point Value Rule*. (See Appendix A.)

Assumptions of *Probabilistic Independence (IN)* are useful for two main reasons. To begin with, application of *IN* creates new opportunities to formulate assumptions of *Equiprobable Random Variables (EP)*. Given an expression in the form *Prob[λ(e)P(e)∧Q(e)|G]* it is not immediately possible to use *EP* on either of the individual variables *P* and *Q*. After applying the first version of *IN*, one obtains an expression involving individual

---

[20]Both the first and second versions of *EP* could actually be applied to the expression *Prob[λ(d)p=Winner(t,d)|G]*, since equality is a boolean valued operator. Application of versions 1 and 2 would result in probabilities of *1/2* and *1/4* respectively.

probabilities of *P* and *Q*. The individual variables can then be the subjects of *EP* assumptions. A similar process occurs using the second version in *IN*. Application of *IN* thus enables one to selectively apply *EP* to individual variables.

Assumptions of *Probabilistic Independence (IN)* also serve to enable subsequent reformulations. Application of *IN* enables subsequent reformulations that factor nested summations into a product of individual summations. This enablement process is outlined in Figure 2-15. The example begins with an expression involving a double, nested summation. Each term in the summation computes the probability that a conjunction of conditions is true. After application of the *IN* operator, the nested summation is factorable into a product of two individual summations. If each summation involves *N* terms, the process effectively reduces the number of calls to *Prob* from an initial value of $2*N^2$ down to a final value of $2*N$.

1. Nested Probability of Conjunction:

Σ **(i) (** Σ **(j) Prob[**λ**(e) p(i,e)** $\wedge$ **q(j,e)])**

2. Apply *Probabilistic Independence (IN)***:**

Σ **(i) (** Σ **(j) Prob[**λ**(e) p(i,e)] * Prob[**λ**(e) q(j,e)])**

3. Factor the Summations:

(Σ **(i) Prob[**λ**(e) p(i,e)]) * (**Σ **(j) Prob[**λ**(e) q(j,e)])**

Figure 2-15: Independence Assumptions Enable Factorization

2.5.4. Efficiency Impact of *PT-GSAs*

Generic simplifying assumptions are intended to improve the efficiency of computations. *Function Invariance* avoids evaluating costly functions. *Equiprobable Random Variables* avoids computation of probability distributions. *Probabilistic Independence* enables subsequent efficiency improving reformulations, i.e., factoring of nested summations. This section has provided only an intuitive idea of the efficiency impact of each *PT-GSA*. A more rigorous analysis of the efficiency impact of each *PT-GSA* will be presented in Chapter 4.

2.5.5. Justification of *PT-GSAs*

The *PT-GSA* family is not the only set of generic assumptions that could be used to simplify theories in the *PT-FORMALISM*. Alternative *GSAs* could have semantics different from the *PT-GSAs* and still improve efficiency to an equal degree. One might reasonably ask: *"Why are the PT-GSAs to be preferred over all others with equal power to improve efficiency?"*. In the context of *POLLYANNA*, generic simplifying assumptions are ultimately justified *a posteriori*, by the results of empirical testing. Nevertheless, an *a priori* justification can be provided for some of the *PT-GSAs*.

The information theoretic concept of *maximum entropy* provides an *a priori* justification for *Equiprobable Random Variables (EP)*. Entropy measures the average uncertainty of a probability distribution [Shannon 48]. Distributions with high entropy have high uncertainty and correspondingly low information content. Given a finite event space, the equiprobable distribution has greater entropy than any alternative probability distribution. An *EP* assumption therefore carries less information than any assumption of an alternative distribution over the same space. *EP* assumptions are as weak as possible in terms of information content. This fact can be used to form an analogy with a principle of logic. A logical formula *A* is said to be weaker than a logical formula *B* if *B* entails *A*, but *A* does not entail *B*. Formula *A* is a safer assumption than formula *B*, since formula *A* is no more likely (and possibly less likely) to be false than formula *B*. By replacing the notion of logical weakness with the notion of weak information, this principle of logic becomes the principle of Maximum entropy. Equiprobable distributions and the principle of maximum entropy have been used in this manner in the information theoretic formulation of statistical mechanics [Jaynes 83b].

Equiprobable distributions can also be justified using invariance arguments. Suppose *v* is a random variable with a range equal to a set *S* of symbols: $\{s_1,...,s_n\}$. If symbols themselves have no significance, assumptions should treat them identically. The assumed distribution *D(v)* should therefore be invariant under all permutations of the set *S*. Only the uniform distribution satisfies this requirement. The invariance argument is similar to arguments offered to justify the use of certain *a priori* parameter distributions in the context of Bayesian inference [Jaynes 83c].

Assumptions of *Probabilistic Independence (IN)* can also be justified in terms of maximum entropy. Suppose one seeks a joint probability distribution *D(x,y)* for the random variables *x* and *y*. Suppose further that the individual distributions: *Dx(x)* and *Dy(y)* are known. Of all the joint distributions *D(x,y)* that yield individual distributions *Dx* and *Dy*, the one of maximum entropy is $D(x,y)=Dx(x)*Dy(y)$, i.e., the distribution in which *x* and *y* are completely independent [Shannon 48]. When faced with a choice of what to assume about the correlation of random variables, the concept of maximum entropy suggests assuming probabilistic independence.

2.5.6. Generalizations of *PT-GSAs*

The *PT-GSAs* may be seen as instances of even more general simplifying assumptions. Some generalizations of the *PT-GSAs* are shown in Figure 2-16. Three *GSAs* called *"Argument Abstraction (AA)"*, *"Equiprobable Ranges (ER)"* and *"Correlation Constraints (CC)"* represent generalizations of the three types of *PT-GSAs* shown in Figure 2-13. These generalizations indicate how the *PT-GSA* family could be extended to include an even broader class of generic simplifying assumptions.

- *Argument Abstraction (AA):* Given a function *F(x)* and a many to one function *Abstract* that maps *Domain(F)* into *Domain(F)*, assume that $(\forall\ x)\ F(x)=F(Abstract(x))$.
- *Equiprobable Ranges (ER):* Given a random variable *v*, let *S* be any subset of *Range(v)*. Assume that *D(v)* is $1/|S|$ if *v* is a member of S, and zero otherwise.
- *Correlation Constraints (CC):* Given two random variables *v* and *w* with individual distributions, *Dv(v)* and *Dw(w)*:
  - Assume *v* and *w* are related by a function *F* so that the joint distribution *D(v,w)* equals *Dv(v)* if *w=F(v)*, and is zero otherwise.
  - Assume *v* and *w* are probabilistically independent so that *D(v,w)=Dv(v)*Dw(w)*.

Figure 2-16: Generalizations of the *PT-GSAs*

Argument Abstraction (AA) is a generalized version of *Function Invariance (FI)*. It uses a many to one function *Abstract* to partition the domain of a function *F* into equivalence classes: $[x] = \{y|Abstract(y)=Abstract(x)\}$. The value of *F(x)* is assumed to be constant across each class. *Function Invariance* falls out as an extreme case in which *Abstract* maps the entire domain of *F* to a single element. *Argument Abstraction* improves efficiency by reducing the number of times *F(x)* must be computed. The function *F* needs only to be computed at most once for each equivalence class defined by *Abstract*.

Two particularly interesting cases of *Argument Abstraction* are worth mentioning. Suppose the argument *x* of *F(x)* is a tuple $(x_1,...,x_n)$. The function *Abstract* can then be defined as a projection that sets some of the x_i components equal to constants. A second (overlapping) special case occurs when the function *F* is defined over a set of spatial coordinates. The function *Abstract* can then impose a physical symmetry constraint on the function *F*, e.g., spherical, cylindrical or translational symmetry, etc. Symmetry constraints result from a function *Abstract* that operates by transforming to new coordinates (spherical, cylindrical, etc.), projecting all but one of the new coordinates, and transforming back to the original coordinates. For example, spherical symmetry results from using the function *Abstract(x,y,z)=(r,0,0)*, where $r=(x^2+y^2+z^2)^{1/2}$.

The efficiency gain of *Argument Abstraction* can be achieved using *memoization* techniques [Mostow and Cohen 85] (See Section 2.6). Memoization operates by storing each computed value of a function *F* in a table. Before evaluating any function call of the form *F(x)*, the system first checks to see if an appropriate value was previously stored in the table. In the absence of *Argument Abstraction,* the system would look for a value stored under the index *F(x)*. Under an *AA* assumption, it would look for a value stored under the index *F(Abstract(x))* instead. *Argument Abstraction* thus improves efficiency by increasing the lookup hit rate achieved with memoization.

Equiprobable Ranges (ER) is a generalized version of *Equiprobable Random Variables (EP)*. In order to formulate an *ER* assumption, one selects a subset *S* of the range of a random variable *v*. Values in the set *S* have probability $1/|S|$, while those outside of *S* have probability zero. *EP* assumptions fall out as an extreme case in which *S=Range(v)*. At the opposite extreme, one can formulate *ER* assumptions in which only a single value has non-zero probability. When the set *S* contains only one element, a single value becomes the *default* for variable *v*, to be used when computing the true distribution *D(v)* is too costly. *Correlation Constraints (CC)* is a generalized version of *Probabilistic Independence (IN)*. This class includes assumptions of complete independence, as well as the opposite assumption of perfect correlation.

2.5.7. Subsumption Relations among *PT-GSAs*

The *PT-GSA* family could, in principle, be reduced to a smaller set of generic simplifying assumptions. This section demonstrates that *Function Invariance (FI)* can be used to implement both *Equiprobable Random Variables (EP)* and *Probabilistic Independence (IN)*. When suitable types of reformulation knowledge are available, *FI* alone can generate all the same results are generated by the entire *PT-GSA* family, i.e., including *EP* and *IN* as well as *FI*.

Equiprobable Random Variables (EP) can be viewed as a special case of *Function Invariance (FI)*. This relationship can be drawn in two different ways. To begin with, *EP* may be viewed as the result of applying *FI* to the function *D(v)*, which is the probability distribution for random variable v. When *FI* is applied in this manner, a normalization constraint can be used to select the actual constant to be $1/|Range(v)|$. The relationship between *FI* and *EP* can also be observed in another context. Consider a function of the form: $F(c)=Prob[\lambda(e)v(e)|G(c)]$. Notice that the *givens* of the conditional probability depend on the argument *c* to the function *F(c)*. Application of *EP* yields the result: $F(c) = 1/|Range(v)|$ The same result could be obtained apply *FI* directly to *F(c),* by choosing a constant value of $1/|Range(v)|$. *EP* is far more restrictive than *FI* in this context, because *EP* constrains the constant to a specific value. *FI* enforces no such constraint.

Probabilistic Independence (IN) can also be viewed as a special case of *Function Invariance (FI)*. This relationship can be observed by considering an alternate definition of probabilistic independence. Two boolean variables *P* and *Q* are probabilistically independent whenever the probability of *P* given *Q* is equally to the global probability of *Q*, i.e., *Prob[P|Q]=Prob[P|True]*. Now consider a function of the form: *F(G)=Prob[λ(e)f(e)|G]*. The given condition *G* is actually an argument to the function *F(G)*. Application of the alternate definition of *IN* yields the result: *F(G)=Prob[λ(e)f(e)|True]*. This result could also be obtained by direct application of *FI* since the simplified expression does not depend on the argument to *F(G)*. *IN* is more restrictive than *FI*, since *IN* constrains the constant to a specific value, whereas *FI* enforces no such constraint.

A skeptic might reasonably ask why the entire *PT-GSA* family is needed, if all the same results can be obtained using *Function Invariance (FI)* alone. The answer is search control. *FI* is more general than *EP* and *IN*. Taken by itself, *FI* generates many more approximations than result from *EP* and *IN*. These additional approximations can add to the costs of empirical testing in the *theory space search* phase. *FI* would be equivalent to *EP* and *IN* only if a search control strategy constrained *FI* to those applications that implement *EP* or *IN*. *EP* and *IN* can be viewed as a means of embedding search control knowledge in the definitions of *GSA* operators.

A summary of subsumption relationships among *GSAs* is shown in Figure 2-17. Two types of subsumption relations are shown in this diagrams. The relations labeled "I" correspond to direct instantiation. For example, *Function Invariance* is a direct instantiation of *Argument Abstraction*, with an appropriate choice of the function *Abstract*. The relations labeled "R" are more complex. They indicate that the general type of assumption can be used to implement the more specific one. The implementation process may involve some reformulation operations. For example, in order to implement *EP* or *IN* with *FI*, a *Fold* operation might be required to define a new function to which *FI* can then be applied.

Equiprobable Random Variables (EP) may be interpreted as an interesting extension of *Function Invariance (FI)*. Assumptions of *Function Invariance (FI)* are especially useful for numeric expressions, in which the chosen constant can be an average value. A problem arises in the case of symbolic expressions with unordered ranges. No natural average value exists in such cases. (What is an average suit in hearts? What is an average job in scheduling?) *Equiprobable Random Variables (EP)* can be used to overcome this difficulty. If computations are implemented as probabilities and expectation values using the *PT-FORMALISM*, the *EP* operator can generate equiprobable averages over the values of symbolic expressions. In the case of numeric expressions, the result is often the same as using *FI* with an average numeric value. *EP* extends the same strategy to expressions with unordered symbolic ranges.

Figure 2-17: Subsumption Relations among *GSAs*

2.6. Reformulation Knowledge

The types of truth-preserving reformulation knowledge used in *POLLYANNA* are summarized in Figure 2-18. A complete list of reformulations is found in Appendix G. Reformulations serve two distinct purposes in the context of *POLLYANNA*. Some reformulations create new opportunities to instantiate generic simplifying assumptions. The reformulations that transform probabilities and expectations of composite functions are examples of this type. (See Figure 2-20). Other reformulations directly improve the efficiency of a domain theory. These are analogous to traditional optimizing compiler techniques that improve program efficiency while preserving semantics. The reformulations that install memoization code are an example of this type. (See Figure 2-21 in Section 2.6.3.)

- Identities of Logic, Set Theory, Algebra.
- Laws of Probability Theory.
- Fold and Unfold Function Definitions.
- Insertion of Memoization Code.

Figure 2-18: Reformulation Knowledge

2.6.1. Folding and Unfolding Function Definitions

The *Fold* and *UnFold* reformulations are defined in Figure 2-19. These operations were originally defined in [Burstall and Darlington 77]. *UnFold* operates on function call expressions. It replaces the function call with the result of substituting actual parameters for formal parameters throughout the body of the function definition.[21] *Fold* performs the inverse operation. It replaces arbitrary expressions with calls to functions. It also defines new functions as a side effect. The actual parameters of the new function call can be any set of sub-expressions of the original expression. *Fold* can therefore be applied to a given initial expression in many different ways. *UnFold* is useful for enabling subsequent applications of the operators that reformulate expectations and probabilities of composite functions, as described below. *Fold* is useful for enabling subsequent applications of *Function Invariance (FI)*. Once a new function is defined, it can be made invariant through application of the *FI* operator.

- *UnFold*: If function $F=\lambda(x_1...x_n)Mac[x_1,...,x_n]$ then replace $F(e_1,...,e_n)$ with the result of expanding $Mac[e_1,...,e_n]$.
- *Fold*: If expression E contains sub-expressions $e_1,...,e_n$ and is syntactically identical with the expansion of $Mac[e_1,...,e_n]$, then replace E with $F(e_1,...,e_n)$ and define a new function: $F=\lambda(x_1,...,x_n)Mac[x_1,...,x_n]$.

Figure 2-19: Folding and Unfolding Function Definitions

Fold and *UnFold* can be used to control the scope of generic simplifying assumptions. Suppose *UnFold* is applied to the function call *F(x)* appearing within some expression *e*. If *GSA* operations are subsequently applied to versions of *F*, they will have no impact on the expression *e*. UnFold thus limits the scope subsequent *GSA* operations. *Fold* can achieve the opposite effect. Suppose *Fold* replaces an expression e_1 with $F(e_2)$, where e_1 is an expression matching the body of *F*. Subsequent *GSA* operations applied to versions of *F* will then impact the expression $f(e_2)$. Fold thus expands the scope of subsequent *GSA* operations. By selectively applying *Fold* and *UnFold*, one can arrange to use a highly simplified version of *F(x)* at points in the theory where an exact value is not important. A more sophisticated version of *F(x)* can be used at points where exact values are critical.

2.6.2. Reformulation of Composite Functions

Two particularly important reformulation operators are defined by the equations in Figure 2-20. These equations describe how to compute a *Probability of a Composite Function (PC)* or an *Expectation of a Composite Function (EC)*. A composite function is simply a function defined as a composition of other functions. The operators *PC* and *EC* serve to replace an expression involving one random variable, *f*, with a new expression involving some other random variable *g*. By generating a probability or expectation value of *g*, these reformulations enable a subsequent application of *EP* to assume all values of the variable *g* are equally likely.

[21]The notation *Mac[x]* is used to designate the result of a macro expansion taking *x* as an argument.

```
Probability of a Composite Function (PC):

  Prob[λ(e) v=f(g(e))] = Σ (x in Range(g))
                           If v=f(x) then Prob[λ(e) x=g(e)] else 0

Expectation of a Composite Function (EC):

  Exp[λ(e) f(g(e))] = Σ (x in Range(g)) f(x) * Prob[λ(e) x=g(e)]
```

Figure 2-20: Probabilities and Expectations of Composite Functions

EC and *PC* usually operate in combination with *UnFold*. For example, when given an initial expression of the form *Prob[λ(e) v=F(e))]*, *UnFold* might expand the definition of *F* to obtain an expression in the form *Prob[λ(e) v=f(g(e))]*. The *PC* operator could then transform this into an expression involving *Prob[λ(e)x=g(e)]*. Repeated application of *UnFold* in combination with *EC* and/or *PC* can transform an expression involving the variable *Game-Score* into expression involving any of the random variables shown in the tree of Figure 2-10. These reformulations are therefore the key to generating a large class of different *EP* assumptions.

2.6.3. Memoization

A definition of *Memoization* is found in Figure 2-21. *Memoization* serves to associate a software cache with a specific function *F(x)* [Bird 80]. Whenever the function *F* is actually evaluated, the result is stored in a table. On subsequent calls of *F(x)*, the memoization code checks whether a value of *F(x)* was previously stored in the table. *Memoization* thus arranges for the function *F* to be evaluated only once for each argument. *Memoization* is especially effective when used in combination with *Argument Abstraction (AA)* or *Function Invariance (FI)*. The lookup hit rate can improve when *Function Invariance (FI)* sets one argument x_i of $F(x_1,...,x_n)$ equal to a constant. It can also improve when a function *Abstract* is used to divide the range *F* into equivalence classes.[22]

```
Initial Definition: F(x) = Expression[x]

New Definition:     F(x) = 1. Let L = Lookup[x,Table].
                           2. If L <> NIL
                                then Return(L)
                                else a. Let L = Expression[x].
                                     b. Store(L,Table).
                                     c. Return(L).
```

Figure 2-21: Insertion of Memo Functions

[22]*Fold* is also quite useful for enhancing the memoization hit rate. Suppose *Fold* is used to replace expressions e_1 and e_2 with two calls to the same function, i.e., $F(args_1)$ and $F(args_2)$. When function *F* is memoized, the results of computing expression $e_1=F(args_1)$ might in some cases be used to avoid computing expression $e_2=F(args_2)$, i.e., whenever $args_1=args_2$.

2.6.4. Soundness of Reformulation Operations

The reformulations used in *POLLYANNA* are all logically *sound*. For most of the reformulation operators, the fact of soundness is obvious from the definition. Soundness is less obvious for a few of the operators. In such non-obvious cases, an explanation of the reformulation is found along with the definition in Appendix G. The operator definitions and associated explanations are offered in support of the claim that *POLLYANNA* produces logically sound derivations of heuristics, provided the instantiated *GSAs* are assumed to be true. The soundness property supports the claim that *POLLYANNA* uses a principled method of generating approximate theories. (See Claim #3 in Section 1.6.4.)

2.6.5. Interaction of Assumption and Reformulation

A summary of the interactions between reformulations and *GSAs* in *POLLYANNA* is found in Figure 2-22. Each of these interactions were discussed above. Three distinct modes of interaction have been identified. (1) Truth preserving reformulations can enable new applications of *GSAs* that were not previously possible. (2) *GSAs* can enable subsequent reformulations that directly improve efficiency. (3) *Fold* and *UnFold* can be used to control the scope of simplifying assumptions. This summary serves to support the claim about interactions of generic simplifying assumptions and truth-preserving reformulations (Claim #5 in Section 1.6.6).

- Reformulations enable efficiency improving *GSAs*:
  - *Fold* enables *Function Invariance*.
  - *Unfold* and *Probability / Expectation of a Composite Function* enable *Equiprobable Random Variables*.
- *GSAs* enable efficiency improving reformulations:
  - *Function Invariance* and *Argument Abstraction* improve the *Memoization* hit rate.
  - *Probabilistic Independence* enables factoring nested summations.
- *Fold* and *UnFold* control the scope of assumptions.

Figure 2-22: Interactions of *GSAs* and Reformulations

2.7. Search Control for Generation of Approximations

2.7.1. Goals for Approximation Generation

The intractable theory problem definition specifies accuracy and efficiency constraints as the goals of learning. (See Figure 1-3 in Section 1.2.5.) These goals are tested empirically in the *theory space search (TSS)* module. The accuracy and efficiency constraints also serve as implicit goals for the *theory space generation (TSG)* and *approximation generation (AG)* modules. In the current implementation, these modules are implemented as temporally distinct phases of learning. In a more integrated system, these phases would overlap in time. When accuracy and efficiency goals are tested empirically by the *TSS* module, the test results would be used to control the *TSG* and *AG* processes as well.

2.7.2. Problems with Blind Search Control

Blind search control would encounter several difficulties when used in *POLLYANNA*'s approximation generator. One difficulty concerns the branching factor that results from certain *GSA* and reformulation operators. The *Fold* operation can be applied in a vast number of different ways to any problem state. Furthermore, one of the *Function Invariance (FI)* operators can be applied in an *infinite* number of ways to each problem state. It allows any function definition to be replaced with *any* constant, supplied by an interactive user. Although these operators have a dramatic effect on the branching factor, they do not represent serious problems of search control. Both operators can be dispensed with entirely. The presence or absence of *Fold* does not impact the set of approximations that can be generated in principle. *Function Invariance* can be restricted to replace expressions with appropriate algebraic identities. Although this would limit the range of theories generated, the restricted forms of *Function Invariance* would suffice to generate all the results obtained so far in *POLLYANNA*.

A more serious problem concerns semantic redundancy of approximations. The *GSA* and reformulation operators can produce many approximate theories that are semantically equivalent, despite superficial syntactic differences. Semantically equivalent theories have the same accuracy levels, but have potentially distinct levels of efficiency. Assuming the most efficient is theory is preferred, the others simply add to the costs of empirical testing, without improving the results. Two distinct types of redundancy are worth mentioning. One type of redundancy occurs when two syntactically different definitions of *F(x)* each return the same value for all arguments. A second type of redundancy occurs when the function *F(x)* is used as an evaluation function. In such cases, only the rank order of values of *F(x)* is significant. Even when distinct values are returned by two versions of an evaluation function *F(x)*, the versions are redundant if their respective rank orders are the same. Examples and sources of semantic redundancy will be discussed below.

2.7.3. A Proposed Automatic Control Strategy

A proposed automatic control strategy for the *approximation generator* is described in Figure 2-23. The control strategy is *automatic* rather than *mechanical* because it chooses operator applications itself, without relying on human intervention. The procedure outputs a table of functions and versions representing the final state of the approximation generator process, as described in Figure 2-6. These function definitions are a proper subset of those that would result from using blind search in the approximation generator. When used in the *theory space generation*

(TSG) module, they result in a smaller theory space than would result from blind search. An analysis of the size of theory spaces resulting from this control strategy is found in Chapter 4.

GENERATE(F,D):

F(x) = Prob[λ(e) v=h(e) |G]

1. **Derive F(x) Version #0: Apply** *Equiprobable Random Variables (EP)* **to h(e) to generate a constant version of F(x).**
2. **Derive F(x) Version #1:**
   a. **Insert a test that checks whether the value of h(e) is already known under givens G.**
   b. **Apply** *Equiprobable Random Variables (EP)* **to h(e) to generate a constant to be returned only when h(e) is unknown.**
3. **If (D > 0) then Derive F(x) Version #2:**
   a. **Use** *Unfold* **to replace h(e) with s(t_1(e),...,t_n(e)).**
   b. **Apply** *Probability of Composite Function (PC)* **to generate a nested summation over the ranges of variables t_1,...,t_n. Each term of the summation involves the conjunctive probability that variables t_1,...,t_n will have a specific combination v_1,...,v_n of values.**
   c. **Apply** *Probabilistic Independence (IN)* **to change the conjunctive probability into a product of probabilities of individual t_i values: Prob[λ(e)v_i=t_i(e)|G].**
   d. **Use** *Fold* **repeatedly to define and insert n calls to functions G_i(x,v_i). Each G_i(x,v_i) computes the probability that t_i has value v_i: i.e., G_i(x,v_i)=Prob[λ(e)v_i=t_i(e)|G].**
4. **If (D > 0) then Derive F(x) Version #3:**
   a. **Insert a test that checks whether the value of h(e) is already known under givens G.**
   b. **Apply the operators** *UnFold*, *Probability of Composite Function (PC)*, *Probabilistic Independence (IN)*, **and** *Fold* **as described above to write F(x) in terms of an expression that calls each G_i(x,v_i) only when h(e) is unknown.**
5. **For (i = 1 ... n) GENERATE(G_i,D-1).**

Figure 2-23: Automatic Control Strategy

The control strategy in Figure 2-23 can be understood in terms of a tree that defines functional relationships among random variables. Examples of such trees are shown in Figures 2-10 and 2-12. The procedure *GENERATE(F,D)* takes a function *F(x)* and a depth bound *D* as input. The function *F* is assumed to compute the probability that a random variable *h(e)* has value *v*. Initially it has only a single definition of the form *F(x)=Prob[λ(e)v=h(e)|G]*. *GENERATE* systematically applies assumptions of *Equiprobable Random Variables (EP)* and *Probabilistic Independence (IN)* to random variables appearing at most depth *D* below *h(e)* in the tree. *GENERATE* is called recursively one time for each random variable at depth *D* or higher.

GENERATE produces four different versions of the function $F(x)$. These versions implement different ways of computing the probability that $h(e)$ has value v. Version number one returns the constant $1/|Range(h)|$. Version number two is nearly the same as version one. It first checks whether the value of $h(e)$ is already known exactly, and returns the constant $1/|Range(h)|$ only when $h(e)$ is unknown. Version number three operates by computing probabilities associated with variables one level deeper than $h(e)$ in the tree. Version number four is nearly the same as version three. It first checks whether the value of $h(e)$ is already known exactly. When $h(e)$ is unknown, it operates by computing probabilities associated with variables one level deeper than $h(e)$ in the tree.[23] A more detailed version of this algorithm is found in Appendix H. A sample approximation generator execution trace is found in Appendix I. The trace illustrates how the proposed procedure *GENERATE* might operate if it were applied to the hearts domain theory in Figure 2-9.

2.7.4. Extensions of the Control Strategy

The procedure *GENERATE* must be extended in several ways to properly handle all theories in the *PT-FORMALISM*. The first extension applies to functions that compute expectation values rather than probabilities: $F(x)=Exp[\lambda(e)h(e)|G]$. A procedure to handle expectation values would look similar to the procedure *GENERATE*; however, the specific operators used will be different from those used to process computations of probabilities. For example, the reformulation operator *Probability of Composite Function (PC)* would be replaced by *Expectation of Composite Function (EC)*. Notice that the *EC* operator transforms an expectation value expression into one involving a probability. (See Figure 2-20.) After the first level of recursion, the extended procedure would reduce $F(x)$ an expression involving only computations of probabilities. The original *GENERATE* procedure in Figure 2-23 can then be applied directly.

Two special types of expectation value computations are worth mentioning. If $h(e)$ is a sum of other random variables, $t_1(e),...,t_n(e)$, a special reformulation rule is used to reduce the expectation of $h(e)$ to a sum of expectation values of each variable $t_i(e)$. No independence assumption is needed, since the expectation of a sum is always equal to the sum of individual expectation values. If $h(e)$ is a product of other random variables, $t_1(e),...,t_n(e)$, a special version of *Probabilistic Independence (IN)* is used to change the expectation of $h(e)$ into a product of expectation values of each variable $t_i(e)$. In both special cases, the extended version of *GENERATE* is called recursively on the expectation of each variable $t_i(e)$.

---

[23]The procedure *GENERATE* can produce even more versions of $F(x)$. The function $s(t_1(e),...,t_n(e))$ that resulted applying *Unfold* to $h(e)$ may not be primitive. If function s is not primitive, then *UnFold* can be applied to this function as well. In general, if $h(e)$ can be unfolded N times, *GENERATE* will produce $N+2$ versions of $F(x)$.

The *PT-FORMALISM* includes some constructs that do not fit into the framework defined in Figure 2-23. Some of these constructs involve *second order* functions, i.e., functions that take λ-expressions as arguments. Others involve functions whose arguments are *sets*. The *GENERATE* procedure found in Appendix H and outlined in Figure 2-23 ignores the possibility of such constructs. It assumes that $h(e)=s(t_1(e),...,t_n(e))$, and formulates a sum over the range of each random variable t_i. If $t_i(e)$ is a λ-expression depending on the event variable *e*, the range of t_i is large and difficult to represent. If $t_i(e)$ is a set, the range of t_i is also large. Such troublesome constructs are handled by invoking reformulation rules that transform expressions involving the troublesome constructs into expressions involving the simpler constructs handled by the *GENERATE* procedure.

Means-ends analysis would provide a natural framework for extending *GENERATE* to handle expectation values and second order constructs. At each level of recursion, the system would set up a separate goal for making each of the four versions produced by *GENERATE*. The four goal types would be further broken down according to the type of expression that defines the function *F(x)* begin processed. (E.g., expectation values would be one special case. Second order operations would lead to additional special cases.) The table of goal types would therefore contain entries of the form: *(Version#i, Expression-Type)*. Each entry would be associated with the appropriate operators for building the indicated version expressions of the indicated type. The means/ends control structure would successively generate each of the four versions by invoking the operators associated with the corresponding goal type.

2.7.5. Restrictions of the Control Strategy

The procedure *GENERATE* can be restricted in several ways that reduce the number of approximations generated. One restriction operates by detecting redundancy among approximations. Some of the function versions produced by the automatic procedure may be semantically equivalent to each other. Cases of such equivalence occur whenever the random variable *h(e)* does not appear in the givens *G* that represent problem states. Tests checking for a known value of *h(e)* are bound to fail. The pairs *(Version#0, Version#1)* and *(Version#2, Version#3)* become semantically equivalent. The restricted procedure *GENERATE-A* in Figure 2-24 is designed to avoid this type of redundancy. It only produces versions testing for known values when the variable *h(e)* can potentially appear in the problem state descriptions. In order to operate, this procedure must be supplied with meta-level information indicating what variables can and cannot appear in problem states.[24]

A further restricted control strategy is described by procedure *GENERATE-B* in Figure 2-24. It generates only

[24]The control restrictions described in Figure 2-24 could be applied to either the original control procedure *GENERATE* or to the extended control procedure.

```
GENERATE-A: (Restricted Tests for Known Values.)

     If (Variable h(e) can appear in the problem state.)
        then Begin
             a. Generate Version#0 and Version#1
             b. If (D > 0) Generate Version#2 and Version#3
             End
        else Begin
             a. Generate Version#0
             b. If (D > 0) Generate Version#2
             End

GENERATE-B: (Distinct Problem State Abstractions.)

     If (Variable h(e) can appear in the problem state.)
        then Begin
             a. Generate Version#0.
             b. If (D = 0)
                   then Generate Version#1
                   else Generate Version#3
             End
        else If (D = 0)
                then Generate Version#0
                else Generate Version#2
```

Figure 2-24: Restricted Control Strategies

approximations that represent distinct problem state abstractions. The behavior of this procedure can be understood by first considering how state abstractions are produced by the original *GENERATE* procedure. Let *V* be the set of random variables that can appear in the problem state. Each variable $v \in V$ corresponds to a distinct node in the random variable tree. All known values of variable *v* are effectively ignored and abstracted out of the problem state whenever either (1) *GENERATE* applies an *EP* operator to the variable *v* directly, or (2) *GENERATE* applies an *EP* operator to an ancestor of *v* in the random variable tree. Each variable *v* can thus be abstracted in many different ways, depending on whether *GENERATE* prunes the tree at node *v* or at one of its ancestors. These multiple abstractions are ultimately a product of the ambiguity in Bernoulli's Principle of Insufficient Reason, implemented by the *EP GSA* operator in *POLLYANNA*. (See Section 2.5.2.)

The procedure *GENERATE-B* produces only a single abstraction of each state variable. It does so by using *EP* to prune the random variable tree only at nodes corresponding to state variables. Two function versions are generated for each state variable reached by the expansion. One version ignores the variable *(Version#0)* and one version looks for a known value (*Version#1* or *Version#3*). Only one version is generated for all other nodes in the tree. This restriction does more than just eliminate redundancy. It omits some approximations that are semantically distinct from any that are generated. The omitted approximations are quite similar to the generated ones. They differ only in the values of functions returning constants. For each omitted approximate theory making *F(x)* equivalent to

F(x)=Constant1, the restricted procedure results in some theory making *F(x)* equivalent to *F(x)=Constant2*, although the constants are not generally the same. If the function *F(x)* is called inside a sub-expression of an evaluation function, the different constants can sometimes change the rank order of choices, leading therefore to different decisions among choices.

The *approximation generator* search control problem can be understood in terms of inductive bias. Blind search control generates a large number of versions of each function. In the *theory space generation* phase, these numerous versions result in a large, weakly biased theory space. A large theory space raises the costs of empirical testing in the *theory space search* phase. Intelligent search control is needed in the *approximation generator* to strengthen the bias of the resulting theory space. The algorithms *GENERATE, GENERATE-A* and *GENERATE-B* in Figures 2-23 and 2-24 are attempts at intelligent control in the *approximation generator* of *POLLYANNA*.

2.7.6. Human v. Automatic Control

Human control was used to experiment with the approximation generator of *POLLYANNA*. (See Section 2.3.) The automatic procedures themselves have not been implemented. The human strategy nevertheless followed a general course that closely resembles the procedure *GENERATE* as modified in Sections 2.7.4 and 2.7.5. Human generated execution traces generally reflect the recursive structure of the automatic procedure. Most function versions appearing in the human traces can be identified with one of the four versions produced by *GENERATE* at each node in the random variable tree. The automatic strategies were actually developed as generalizations and formalizations of regularities appearing in the human generated traces.

A pair of complementary questions must be addressed in comparing the human and automatic control strategies. These questions concern the *weakness* and *strength* of the proposed generation strategies. The first asks whether the control restrictions are weak enough: "Are the automatic control strategies capable of generating all or most of the heuristics that resulted from human control?". This question will be addressed by comparing operator sequences used under human control to those that would result from automatic control. (See Section 3.4.5.) A positive answer to this question will support the claim that a fully automatic version of *POLLYANNA* would generate roughly the same heuristics as resulted from human control.

A second question asks whether the proposed control restrictions are strong enough: "Do the automatic control strategies generate few enough heuristics so that empirical testing is feasible within reasonable resource limitations?". This question will be addressed by deriving analytic bounds on the size of theory spaces that result from the proposed automatic control strategies. Theory spaces generated under human control can then be compared in size to those that would result from automatic control. (See Section 4.4.13.) If automatically generated spaces are not too much larger, the analysis will support the claim that empirical testing would be feasible in a completely automatic version of *POLLYANNA*. (See Subsidiary Claim #2 in Section 1.6.2.2.)

2.8. Properties of *POLLYANNA*'s Approximate Theories

2.8.1. Accuracy v. Efficiency

The tradeoff between accuracy and efficiency can be characterized in terms of the depth to which *GENERATE* expands random variable trees like the ones in Figures 2-10 and 2-12. Each node in the tree represents a distinct random variable for which an assumption of *Equiprobable Random Variables (EP)* can be formulated. When *EP* is applied to a random variable, the subtree under that variable is effectively *pruned*. The values of some variables in the subtree may be known in the current problem state. When the subtree is pruned, the known values are ignored. If the tree is pruned near the root, the resulting theory is very efficient. It is also not very accurate because it ignores most of the information in the current problem state. If *EP* is applied to variables at deeper levels in the tree, the theory becomes more expensive to evaluate. In return for this computational expense, the theory is able to exploit more information from the current problem state. By taking account of more information, the theory becomes potentially more accurate. Empirical tests confirming the existence of this tradeoff are presented in Chapter 5.

2.8.2. Asymptotic Time Complexity

The asymptotic time complexity of *POLLYANNA*'s approximate theories can be determined by considering the types of expressions produced by the *GENERATE* procedure. At each level of recursion, the procedure generates a nested summation over the ranges $R_1,...,R_b$ of variables $t_1,...,t_b$, where b represents the branching factor of the random variable tree. After a series of recursive calls to depth d in the tree, the summations are nested to depth $b*d$, assuming a fixed branching factor. If all random variables have a range ranges of fixed cardinality R, a total of R^{bd} individual terms must be evaluated inside an innermost summation. One innermost summation is generated for each of the b^d nodes at depth d leading to a total of b^d*R^{bd} individual terms. Assuming that constant time is required test whether a variable is *Known* and to evaluate each function s in the expression $s(t_1(e),...,t_b(e))$ of Figure 2-23, the resulting approximate theory has running time $O(b^d*R^{bd})$. For any given approximate theory, the parameters b and d are fixed. The range parameter R may vary depending on problem instances. In the hearts domain, a typical range parameter is the number of cards in the deck. In the scheduling domain, typical range parameters are the number of jobs and the number of time slots. A family of problem instances could be parameterized by the sizes of these ranges, e.g., decks with many or few cards, scheduling many or few jobs. The range parameter R thus may therefore represent a natural measure of problem size. The complexity thus reduces to $O(R^{bd})$. All approximate theories generated by *POLLYANNA* run in time that is polynomial in the cardinality of ranges of random variables.

Variations on the automatic control procedure can produce approximate theories with lower time complexity. Suppose that all summations on the same tree level can be factored completely. Factoring converts a summation nested to depth b over the ranges $R_1,...,R_b$ of variables $t_1,...,t_b$ into a product of b single summations over each individual range. The resulting approximate theory will have time complexity $O(R^d)$, since all summation nesting occurs between levels of the tree. An even more favorable situation occurs when the top level function computes an expectation value and all functions $s(t_1,...,t_b)$ are either sums or products. All nested summations can be avoided in such cases. The resulting theory first computes the expectation value of each random variable at depth d, using either an equiprobable average or a known value. The random variable expression tree is then directly evaluated using these expectation values. The entire process requires only constant time.

2.9. Summary of Analytic Generation Techniques

- **A Framework for Generating Heuristics:** *POLLYANNA* derives heuristics using three types of knowledge: An intractable domain theory (*IT*), generic simplifying assumptions (*GSA*) and truth-preserving reformulation knowledge (*R*). The power of generic simplifying assumptions is informally illustrated by deriving two hearts heuristics. Derivations of mutually inconsistent heuristics result from using distinct instantiations of the same *GSAs*. Generic simplifying assumptions are developed by a process of reconstructing known heuristics. Reconstruction often yields insight into the underlying rationale of heuristics.
- **The PT-GSA Family of Generic Simplifying Assumptions:** The *PT-GSAs* include *Function Invariance (FI), Equiprobable Random Variables (EP)* and *Probabilistic Independence (IN)*. *FI* avoids function evaluations by asserting the value of a function to be independent of its arguments. *EP* avoids computing probabilities and expectation values by asserting that a random variable is equally likely to take any of its legal values. *IN* enables factoring summations by asserting that two random variables are probabilistically independent.
- **Reformulation Knowledge:** Truth-preserving reformulations associated with the *PT-GSA* family include theorems of logic, set theory, algebra and probability theory. Two key reformulations include *Probability of a Composite Function (PC)* and *Expectation of a Composite Function (EC)*. These reformulations serve to transform probabilities or expectations involving one random variable into a probability or expectation involving another random variable. Additional reformulations include *Fold, UnFold* and *Memoization*.
- **A Formalism for Representing Domain Theories:** The *PT-FORMALISM* is a language for representing domain theories. The formalism includes an event space, a probability distribution and a collection of functions that define random variables. Domain theories for hearts and job scheduling are represented in the *PT-FORMALISM*. The representations are illustrated in terms of a tree of functional relationships among random variables. The formalism is designed to interface with the *PT-GSA* family of generic simplifying assumptions.
- **A Problem Space Architecture:** *POLLYANNA* uses a problem space architecture to generate approximations. Each problem state is represented as a table that provides definitions of functions appearing in the domain theory. The initial state has only one definition of each function. Subsequent states provide several definitions for each function. Generic simplifying assumptions and truth-preserving reformulations are implemented as operators that transform problem states. Each operation adds one or more new definitions of some functions. A human user makes search control decisions by

choosing operators to apply. The system then mechanically carries out the selected operator applications. The implemented system is therefore mechanical, but not automatic.

- **Proposed Control Strategies:** The proposed automatic control procedure *GENERATE* is generalization of experience from using human control. *GENERATE* can be understood in terms of the tree of random variables appearing in the initial intractable theory. This procedure systematically applies *Equiprobable Random Variables (EP)* assumptions to random variables throughout the tree. The proposed procedures *GENERATE-A* and *GENERATE-B* each implement control strategies that are more restrictive than the strategy implemented by *GENERATE*. A tradeoff between accuracy and efficiency is illustrated by the operation of these procedures. Highly efficient but inaccurate theories result from applying *EP* near the root of the tree. Less efficient but potentially more accurate theories result from applying *EP* at deeper levels. All approximate theories produced by *GENERATE* can be shown run in time that is polynomial in the cardinality of ranges of random variables.

Chapter 3

Analytic Learning Results in *POLLYANNA*

3.1. Introduction

The results of analytic learning can be characterized using a technique called "semantic analysis". Semantic analysis serves to demonstrate that approximate theories represented symbolically in *POLLYANNA* are semantically equivalent to informally stated heuristics. Semantic analysis is an important tool for evaluating individual approximate theories generated by *POLLYANNA*. Although approximate theories should ultimately be evaluated in terms of accuracy and efficiency goals derived from a specific performance context, such goals do not necessarily carry over into other contexts. Semantic analysis serves as a substitute for specific accuracy and efficiency goals. Although semantic analysis is less precise than measurements of accuracy and efficiency, it has the advantage of being independent of any particular performance context.

Once an approximate theory has been shown semantically equivalent to an informally stated heuristic, the heuristic can be evaluated in terms of quality and originality. Several standards of quality are possible. The minimal standard asks whether the heuristic is intuitively plausible. A stricter standard asks whether the heuristic generates behavior on the level of a human novice. The strictest standard compares the heuristic to those used by human experts. Several standards of originality are also possible. The minimal standard asks whether the heuristic is novel to the system designer, i.e., did the designer anticipate the heuristic prior to implementation. Heuristics failing to meet this standard shall be designated "reconstructed" heuristics. Those that meet the standard shall be designated either "discovered" heuristics or "rediscovered" heuristics. A heuristic shall be designated "rediscovered" if it was previously known to some humans. A heuristic shall be designated "discovered" if it is not known to have ever been previously formulated by anyone.

POLLYANNA generates approximate theories that are semantically equivalent to a variety of hearts and scheduling heuristics. Generated heuristics for hearts include the *Dump High Rank Rule* and the *Dump High Point Value Rule*. Additional generated hearts heuristics include extensions to the two dumping rules as well as more complex formulae for striking a balance among multiple decision criteria. Generated scheduling heuristics include *Soonest Scheduling Deadline First*, *Enable Many Jobs* and *Enable a Well Prepared Job*, among others.

Heuristics generated by *POLLYANNA* fall into several of the categories defined above. In terms of quality, some generated heuristics are not plausible. Others are merely plausible. Still others generate novice level behavior. In terms of originality, some heuristics were merely reconstructed by the system. Others were apparently rediscovered by the system, i.e., they were not anticipated by the designer, but they were probably known previously to some humans. Still others are probably discoveries, i.e., they appear unlikely to have previously been formulated by anyone. These results are especially significant considering the manner in which the heuristics were generated. All the heuristics resulted from applying generic simplifying assumptions and truth-preserving reformulations to initially intractable theories.

3.2. Overview

Semantic analysis of approximate theories generated by *POLLYANNA* is summarized in Section 3.3. Semantically equivalent hearts and scheduling heuristics will be presented in Sections 3.3.1 and 3.3.2 respectively. The semantic analysis is supported by several appendices. The actual output from the approximation generator is presented in Appendix J for the hearts domain, and in Appendix L for the scheduling domain. The output data is analyzed in Appendix K, for the hearts domain, and Appendix M for the scheduling domain. The analyses involve straightforward but lengthy algebraic manipulations. These analyses were carried out by hand. They derive simpler expressions that are semantically equivalent to the approximation generator output. The approximation generator output and semantic analyses are also summarized in the main body of the text, in Sections 3.3.1 and 3.3.2 below. The discussion will show that *POLLYANNA*'s approximate theories are equivalent to a variety of plausible heuristics described in verbal terms.

Heuristics generated by *POLLYANNA* will be evaluated in Section 3.4. *POLLYANNA*'s hearts heuristics will be compared to a set of previously formulated goal heuristics found in Appendix B. *POLLYANNA* will be seen to generate some, but not all, of the goal heuristics. Rough comparisons between *POLLYANNA*'s scheduling heuristics and known heuristics appearing in the scheduling literature will be made in Section 3.4.2. The quality and originality of generated heuristics will be discussed in Sections 3.4.4 and 3.4.3. The evaluation in Section 3.4 serves to support the claim of semantic properties of approximate theories generated from generic simplifying assumptions (Subsidiary Claim #1 in Section 1.6.2.1).

The generality of *POLLYANNA*'s heuristic generation techniques will be analyzed in Section 3.5. Statistics summarizing the usage of *GSA* and reformulation operators will be presented. Generality within domains will be demonstrated by showing that multiple, mutually inconsistent heuristics result from different instantiations of *GSA* operators. Generality across domains will be demonstrated by comparing the usage rates of each operator in the hearts and scheduling domains. This section serves to support the claim of generality of generic simplifying assumptions (Claim #2 in Section 1.6.3). A summary of lessons learned from implementing *POLLYANNA*'s approximation generator will be presented in Section 3.6.

3.3. Heuristics Generated in *POLLYANNA*

3.3.1. Heuristics for Hearts

3.3.1.1. Approximation Generator Output in Hearts

A summary of one final state produced by *POLLYANNA*'s approximation generator for the hearts domain is found in Figure 3-1. The principal functions in the final state are listed along with a description of referencing relationships. The diagram also indicates which functions have multiple versions, and the number of versions of each. Most of the hearts functions have only one version. The ones with multiple versions are described in Figure 3-3. All hearts functions happen to have at most two versions. Results presented in Chapter 4 will show that a total of 72 approximate theories result from systematically combining versions of hearts functions. Notice that the reference tree in Figure 3-1 is similar, but not identical to the random variable tree in Figure 2-10. For each variable *VAR* in the random variable tree, a function *EXP-VAR* or *PROB-VAR* appears in the reference tree. The reference tree also includes additional functions, e.g., those that compute probabilities of conjunctions of random variable values. The actual output of *POLLYANNA*'s approximation generator for the hearts domain is found in Appendix J. [25] This data was generated using the complex hearts theory representation found in Appendix C.[26]

All the approximate hearts theories share a common general structure, which is described in Figure 3-2. Each theory describes an evaluation function *Exp-Game-Score(c,p,tr,private,public)* that computes player *p*'s expected game score, given that he plays card *c* in trick *tr*, the current trick. The additional arguments represent player *p*'s hand (*private*) and game history (*public*). *Exp-Game-Score* written as a sum of four parts: *Exp-Past-Score*, *Exp-Current-Score*, *Exp-Next-Score*, and *Exp-Future-Score*. These functions respectively compute player *p*'s expected score for past tricks, the current trick, the next trick, and all remaining tricks. The terms *Exp-Past-Score* and *Exp-Future-Score* are equal to constants, as a result of an assumption of *Function Invariance (FI)*. The terms *Exp-Current-Score* and *Exp-Next-Score* are each equal to the odds of winning times the expected trick value. These

[25]Only a partial description of the final state is found in Appendix J. All versions computing exact probabilities or expectation values are intractable, and were therefore omitted.

[26]The complex hearts theory differs in two principal respects from the simpler version in Figure 2-9. It uses an extensional representation that facilitates making separate approximations for different tricks of the game. It also uses numeric rather than boolean functions, where possible. The resulting theory tends to involve expectation values of numeric variables rather than probabilities of boolean variables. Expectation values more often lead to factorable summations, resulting in more efficient approximate theories.

```
Exp-Game-Score
 |
 |----Exp-Past-Score
 |----Exp-Future-Score
 |
 |----Exp-Current-Score
 |     |----Exp-Win [2]
 |     |     |----Exp-Defeats
 |     |           |----Prob-Known-Card
 |     |           |
 |     |           |----Exp-Defeats-Sum-Sum
 |     |                 |----Prob-Lead-Suit
 |     |                 |     |----Prob-Lead-Card [2]
 |     |                 |
 |     |                 |----Exp-Defeats-Sum
 |     |                       |----Prob-Card-Invariant
 |     |
 |     |----Exp-Trick-Value [2]
 |           |----Exp-Point-Value [2]
 |           |     |----Prob-Card [2]
 |           |
 |           |----Exp-Point-Value-Invariant
 |                 |----Prob-Card-Invariant
 |
 |----Exp-Next-Score
       |----Exp-Next-Win [2]
       |     |----Exp-Next-Defeats
       |           |----Prob-My-Next-Card
       |           |----Prob-Next-Defeats-In-Suit-Invariant
       |           |----Prob-Next-In-Lead-Suit-Invariant
       |
       |----Exp-Next-Trick-Value [2]
             |----Exp-My-Next-Point-Value [2]
             |     |----Prob-My-Next-Card
             |
             |----Exp-Op-Next-Point-Value
```

Figure 3-1: Reference Relations among Hearts Functions

versions resulted from applying *Probabilistic Independence (IN)* to the random variables *Win* and *Trick-Value* in the current and next tricks. All approximate theories therefore involve the four key functions *Exp-Win, Exp-Trick-Value Exp-Next-Win* and *Exp-Next-Trick-Value*.

Multiple versions of the key functions *Exp-Win*, *Exp-Trick-Value*, *Exp-Next-Win* and *Exp-Next-Trick-Value* are described in Figures 3-4 and 3-5. Each function has a naive version and a sophisticated version. The naive version is equal to a constant that results from applying *Equiprobable Random Variables (EP)* to the corresponding random variable, i.e., an instance of $\lambda(d)Win(p,t,d)$ or $\lambda(d)Trick\text{-}Value(p,t,d)$. Each sophisticated version is described by an algebraic expression that is semantically equivalent to, but syntactically quite different from the definition in Appendix J. Derivations of the simpler, equivalent forms are found in Appendix K. Lists of all the

```
Exp-Game-Score(c,p,tr,private,public) =

  = Exp-Past-Score(c,p,tr,private,public)
    + Exp-Current-Score(c,p,tr,private,public)
    + Exp-Next-Score(c,p,tr,private,public)
    + Exp-Future-Score(c,p,tr,private,public)

  = Exp-Current-Score(c,p,tr,private,public)
    + Exp-Next-Score(c,p,tr,private,public)
    + Constant

  = Exp-Win(c,p,tr,private,public)
    * Exp-Trick-Value(c,p,tr,private,public)
                  +
    Exp-Next-Win(c,p,tr,private,public)
    * Exp-Next-Trick-Value(c,p,tr,private,public)
                  +
                Constant
```

Figure 3-2: Structure of Approximate Hearts Theories

- *Exp-Win(c,p,t,private,public) (EWN)*: The probability that player *p* wins trick *t*, the current trick.
- *Exp-Trick-Value(c,p,t,private,public) (ETV)*: The expected total point value of trick *t*, the current trick.
- *Exp-Next-Win(c,p,t,private,public) (ENW)*: The probability that player *p* wins trick *t+1*, the next trick.
- *Exp-Next-Trick-Value(c,p,t,private,public) (ENT)*: The expected total point value of trick *t+1*, the next trick.
- *Prob-Lead-Card(c,p,t,private,public,c') (PLC)*: The probability that *c'* is the lead card for trick *t*, the current trick.
- *Exp-Point-Value(c,p,t,private,public,p') (EPV)*: The expected point value of the card played by *p'* in trick *t*, the current trick.
- *Prob-Card(c,p,t,private,public,c',p') (PCD)*: The probability that *c'* is the card played by player *p'* in trick *t*, the current trick.
- *Exp-My-Next-Point-Value(c,p,t,private,public) (ENP)*: The expected point value of the card played by player *p* in trick *t+1* in the next trick.

Figure 3-3: Hearts Functions with Multiple Versions

individual assumptions used in each version are also found in Appendix K.[27]

Exp-Win(c,p,tr,...) computes the probability that player *p* wins the current trick *tr*, given that he plays card *c* in the current trick. The sophisticated version *EWN-1* results from using *Equiprobable Random Variables (EP)* to assume that each opponent *p*' is equally likely to play any card in the deck, if his choice is not already known. (*EP* is applied to the variable λ*(d)Card(p',tr,d)*.) The resulting function returns zero if card *c* is defeated by some opponent's known card choice *c*' that has already been played. If not then *Exp-Win* is estimated by multiplying *N*

[27]Some of the descriptions in Figure 3-4 are based on the assumption that the most sophisticated versions are used for each of the other four functions with multiple versions: *Prob-Lead-Card, Exp-Point-Value Prob-Card Exp-My-Next-Point-Value*. See Appendix J for definitions of the sophisticated versions *PLC-1, EPV-1, PCD-1* and *ENP-1*.

Exp-Win(c,p,tr,private,public):

EWN-0: = Average(Range(Win))

EWN-1: = If [(∀ c′ in Table) Defeats(c,c′,Lead-Suit)]
then $\{A_1 * [A_2 + \mathrm{Rank}(c)]\}^N$
else 0

Exp-Trick-Value(c,p,tr,private,public):

ETV-0: = Average(Range(Trick-Value))

ETV-1: = $B_1 + (N * B_2) + \mathrm{Point\text{-}Value}(c)$

Exp-Next-Win(c,p,tr,private,public):

ENW-0: = Average(Range(Win))

ENW-1: = $\{C_1 + C_2 * \mathrm{Rank\text{-}Sum}(\mathrm{Hand}(p)-\{c\})\}^3$

Exp-Next-Trick-Value(c,p,tr,private,public):

ENT-0: = Average(Range(Trick-Value))

ENT-1: = $D_1 + D_2 * \mathrm{Point\text{-}Sum}(\mathrm{Hand}(p)-\{c\})$

Figure 3-4: Versions of Selected Hearts Functions

N	= Number of opponents yet to play.
Table	= Cards already played in current trick.
Lead-Suit	= The lead suit for the current trick.
Defeats(c,c′,s)	= True if card c defeats c′ given lead suit s.
Rank-Sum(set)	= Sum of ranks in set.
Point-Sum(set)	= Sum of point values in set.

A_1 = 1 / |Range(Card)|
A_2 = The number of different cards that could be defeated by a card c when Suit(c) is the lead suit.

B_1 = Point-Sum(Table)
B_2 = The average point value per card.

$C_1 = A_2 * |\mathrm{Hand}(p)-\{c\}| / C_2$
$C_2 = 1 / (D_2 * |\mathrm{Range}(\mathrm{Suit})| * |\mathrm{Range}(\mathrm{Card})|)$

$D_1 = 3 * \mathrm{Average}(\mathrm{Range}(\mathrm{Point\text{-}Value}))$
$D_2 = 1 / (2 * |\mathrm{Range}(\mathrm{Card})|)$

Figure 3-5: Terms used in Hearts Function Definitions

factors, each representing the odds that player *p* defeats one of the *N* opponents yet to play. Player *p*'s card choice *c* can defeat exactly $A_2 + Rank(c)$ of an opponent's possible choices. Each of an opponent's possible choices has a probability A_1. The odds of defeating one remaining opponent are therefore $A_1*[A_2+Rank(c)]$. The probability of defeating all *N* remaining opponents and winning the trick is therefore $\{A_1*[A_2+Rank(c)]\}^N$.

Exp-Trick-Value(c,p,tr,...) computes the expected value of the current trick *tr*, given that player *p* plays card *c* in the current trick. The sophisticated version *ETV-1* results from using *Equiprobable Random Variables (EP)* to assume that each opponent *p*' is equally likely to play any card in the deck, if his choice is not already known. (*EP* is applied to the variable λ*(d)Card(p',tr,d)*.) The resulting function simply adds up the point values of cards already played by opponents to get a constant B_1. For cards of opponents yet to play, the number of remaining opponents *N* is multiplied by B_2 the average point value of a card. Adding *Point-Value(c)* for player *p*'s card choice *c*, the expected trick value is $B_1+(N*B_2)+$*Point-Value(c)*.

Exp-Next-Win(c,p,tr,...) computes the probability that player *p* wins the next trick *tr+1*, given that he plays card *c* in the current trick. The sophisticated version *ENW-1* results from using *Equiprobable Random Variables (EP)* to assume that each opponent *p*' is equally likely to play any card in the deck during the next trick. (*EP* is applied to the variable λ*(d)Card(p',tr+1,d)*.) *EP* is also used to assume that player *p* is himself equally likely to play any card in *Hand(p)-{c}*, the cards remaining in his hand after he plays card *c* in the current trick. Yet another *EP* assumption asserts that all lead suits are equally likely in the next trick. (*EP* is applied to the variable λ*(d)Lead-Suit(tr+1,d)*.) The resulting function estimates *Exp-Next-Win* by multiplying three factors, each representing the odds that player *p* defeats one of the three opponents in the next trick. Each of player *p*'s possible choices *c'* in *Hand(p)-{c}* can defeat exactly A_2 + *Rank(c'*) of an opponent's possible choices. Since each of player *p*'s choices in *Hand(p)-{c}* has the same probability, the odds of defeating a single opponent are a linear function of the sum of ranks in *Hand(p)-{c}*. The probability of defeating three opponents and winning the next trick is therefore a cubic polynomial function of *Rank-Sum(Hand(p)-{c})*.

Exp-Next-Trick-Value(c,p,tr,...) computes the expected value of the next trick *tr+1*, given that player *p* plays card *c* in the current trick. The sophisticated version *ENT-1* results from using *Equiprobable Random Variables (EP)* to assume that each opponent *p*' is equally likely to play a card of each distinct point value. (*EP* is applied to the variable λ*(d)Point-Value(Card(p',tr+1,d))*.) *EP* is also used to assume that player *p* is himself equally likely to play any card in *Hand(p)-{c}*, the cards remaining in his hand after he plays card *c* in the current trick. The resulting function simply adds the expected point value for each opponent to the expected point value for player *p*. The expected point value for each opponent is a constant equal to an equiprobable average over the range of the function *Point-Value*. The expected point value for player *p* is proportional to *Point-Sum(Hand(p)-{c}),* since he is equally likely to play any card in *Hand(p)-{c})*. The expected value of the next trick is therefore a linear function of *Point-Sum(Hand(p)-{c})*.

3.3.1.2. Equivalence of Hearts Heuristics and Approximate Theories

A selection of the approximate hearts theories generated by *POLLYANNA* is shown in Figures 3-6 and 3-7. Each approximate theory is described by listing a version of each of the four key functions: *Exp-Win*, *Exp-Trick-Value Exp-Next-Win* and *Exp-Next-Trick-Value*.[28] Each theory is also associated with a heuristic described in verbal form. Although the theories and heuristics are described in different terms, they are nevertheless equivalent in the following sense. In any game situation to which a heuristic applies, it recommends exactly the same card choices that would be selected by the corresponding approximate theory. Two distinct types of reasoning are used to demonstrate the equivalence of these heuristics and approximate theories. For most of the heuristics, the equivalence follows from reasoning about monotonicity properties of the functions in Figure 3-4. Such heuristics describe simple criteria for minimizing monotonic functions. All the heuristics in Figure 3-6 fall into this group. In a few additional cases, the equivalence can only be demonstrated by considering the magnitudes of values computed by the functions in Figure 3-4. The heuristics in Figure 3-7 fall into this group.

The simplest approximate theory is *T0*, which results from selecting the naive version of all four functions. This theory defines an evaluation function that always returns a constant. All card choices are considered equally good, so the corresponding heuristic says to play any card (*H0* = *"Random Play"*). Approximate hearts theories *T1*, *T2*, and *T3* are all equivalent to very simple heuristics. Each uses the sophisticated version of only one function, while relying on naive versions for the other three functions. Hearts theory *T1* uses the sophisticated version of *Exp-Trick-Value*. Since this function is monotonically increasing in *Point-Value(c)*, the corresponding heuristic says to play a card of minimal point value (*H1* = *"Minimize Points"*). This heuristic attempts to avoid taking points in the current trick by never playing point cards if possible. Hearts theory *T2* uses the sophisticated version of *Exp-Next-Trick-Value*. Since this function is monotonically decreasing in *Point-Value(c)*, the corresponding heuristic says to play a card of maximal point value (*H2* = *"Maximize Points"*). This heuristic attempts to avoid taking points later by getting rid of them now. Hearts theory *T3* uses the sophisticated version of *Exp-Next-Win*. Since this function is monotonically decreasing in *Rank(c)*, the corresponding heuristics says to play a card of maximal rank (*H3* = *"Maximize Rank"*). This heuristic attempts to avoid taking points later by getting rid of high cards now. Heuristics *H1*, *H2* and *H3* are all quite simple because the corresponding theories *T1*, *T2* and *T3* are monotonic in either *Rank(c)* or *Pont-Value(c)*.

More complex behavior results from hearts theory *T4*, which uses the sophisticated version of *Exp-Win*. This theory focuses on losing the current trick, so the corresponding heuristic *H4* is called *"Lose the Current Trick"*.

---

[28]A complete theory specification requires designating versions of each of the other four functions: *Prob-Lead-Card*, *Exp-Point-Value Prob-Card Exp-My-Next-Point-Value*. This analysis assumes the most sophisticated version of each: *PLC-1, EPV-1, PCD-1* and *ENP-1*.

```
0. Random Play:
   Theory:    (EWN-0, ETV-0, ENW-0, ENT-0)
   H0:        Play a any card.

1. Minimize Points:
   Theory:    (EWN-0, ETV-1, ENW-0, ENT-0)
   H1:        Play a card of minimal point value.

2. Maximize Points:
   Theory:    (EWN-0, ETV-0, ENW-0, ENT-1)
   H2:        Play a card of maximal point value.

3. Maximize Rank:
   Theory:    (EWN-0, ETV-0, ENW-1, ENT-0)
   H3:        Play a card of maximal rank.

4. Lose the Current Trick:

   Theory:    (EWN-1, ETV-0, ENW-0, ENT-0)

   H4a:       IF:   Some legal card is defeated by an opponent's
                    card already on the table.
              THEN: Play any such card.

   H4b:       IF:   Playing last and no legal card is defeated by an
                    opponent's card already on the table.
              THEN: Play any legal card.

   H4c:       IF:   Not playing last and no legal card is defeated by
                    an opponent's card already on the table.
              THEN: Play a card of minimal rank.

5. Lose and Dump High Ranks:

   Theory:    (EWN-1, ETV-0, ENW-1, ENT-0)

   H5a:       IF:   All legal cards are defeated by opponent's
                    cards already on the table.
              THEN: Play a card of maximal rank.

   H5b:       IF:   Playing last and no legal card is defeated by
                    an opponent's card on the table.
              THEN: Play a card of maximal rank.

6. Lose and Dump High Point Values:

   Theory:    (EWN-1, ETV-0, ENW-0, ENT-1)

   H6a:       IF:   All legal cards are defeated by opponents'
                    cards already on the table.
              THEN: Play a card of maximal point value.

   H6b:       IF:   Playing last and no legal card is defeated by
                    an opponent's card on the table.
              THEN: Play a card of maximal point value.
```

Figure 3-6: Hearts Heuristics and Equivalent Approximate Theories

Theory *T4* returns the minimal value of zero when card *c* is defeated by an opponent's card already on the table. Heuristic *H4a* therefore recommends playing a card that is guaranteed to lose the current trick, whenever some such card is legal. When no legal card is guaranteed to lose, theory *T4* reduces to the quantity: $\{A_1 * [A_2 + Rank(c)]\}^N$, where A_1 and A_2 are constants, and *N* is the number of opponents yet to play. If player *p* is playing last, then *N* is zero and this quantity is a constant. All card choices are then considered to be equally good. Heuristic *H4b* therefore recommends any legal card under these circumstances. In all other cases, theory *T4* is a monotonically increasing function of *Rank(c)*. Heuristic *H4c* therefore recommends a legal card of minimal rank, whenever no card is guaranteed to lose and player *p* is not playing last.

Hearts theories *T5* and *T6* result from combining the sophisticated version of *Exp-Win* with sophisticated versions of *Exp-Next-Win* and *Exp-Next-Trick-Value* respectively. These theories focus on losing the current trick, and also consider one aspect of the next trick. Theory *T5* considers the probability of winning the next trick. The corresponding heuristic *H5* is called *"Lose and Dump High Ranks"*. Theory *T6* considers the expected point value of the next trick. The corresponding heuristic *H6* is called *"Lose and Dump High Point Values"*.

Partial descriptions of theories *T5* and *T6* are given by the verbal heuristics *H5a*, *H5b*, *H6a* and *H6b*. These heuristics apply to cases in which the function *Exp-Win* returns a constant value for all card choices. Decisions are therefore made by minimizing *Exp-Next-Win*, in theory *T5*, and by minimizing *Exp-Next-Trick-Value*, in theory *T6*. Heuristics *H5a* and *H6a* are respectively equivalent to the *Dump High Rank Rule* (Figure 2-3) and the *Dump High Point Value Rule* (Figure 2-4), described in Chapter 2. Heuristic *H5b* is a natural extension of the *Dump High Rank Rule*. It recommends getting rid of high rank cards when all legal cards are guaranteed to win the current trick. Heuristic *H6b* is perhaps a foolish extension of the *Dump High Point Value Rule*. It recommends playing high point value cards when all legal cards are guaranteed to win the current trick. Although a high point value card might not be taken in future tricks, it definitely will be taken in the current trick. Theory *T5* unfortunately overlooks this fact by setting *Exp-Trick-Value* to a constant.

The behavior of hearts theories *T5* and *T6* cannot be completely described by reasoning about monotonicity properties of functions. The difficult cases arise when conflicting advice is offered by terms corresponding to the current and next tricks of the game. Heuristics *H5a*, *H5b*, *H6a* and *H6b* were derived by assuming *Exp-Win(c,...)* is the same for all legal choices so that decisions could be made by minimizing *Exp-Next-Win* or *Exp-Next-Trick-Value* corresponding to the next trick of the game. When *Exp-Win(c,...)* varies according to the card choice *c*, both the current and next tricks of the game must be considered. Terms corresponding to the current and next tricks can offer conflicting advice. Notice that *Exp-Win* is minimized for cards of low rank, while *Exp-Next-Win* is minimized for cards of high rank. Likewise *Exp-Win* and *Exp-Next-Trick-Value* can be minimized by different cards, since a lowest rank card does not generally have greatest point value. Theories *T5* and *T6* can therefore be completely described only by considering the numerical magnitudes of functions.

5. *Lose and Dump High Ranks:*

```
Theory:    (EWN-1, ETV-0, ENW-1, ENT-0)

H5c:       IF:   Some legal cards are defeated by opponents'
                 cards already on the table.
           THEN: Play a maximal rank defeated card.
```

6. *Lose and Dump High Point Values:*

```
Theory:    (EWN-1, ETV-0, ENW-0, ENT-1)

H6c:       IF:   Some legal cards are defeated by opponents'
                 cards already on the table.
           THEN: Play a maximal point value defeated card.
```

Figure 3-7: Additional Cases of Approximate Hearts Theories

Heuristics *H5c* and *H6c* in Figure 3-7 handle the additional cases of hearts theories *T5* and *T6*. They apply to situations in which conflicting advice is offered by current and next trick terms in the evaluation functions. Both heuristics implicitly assign the current trick greater priority. Each recommends first finding the set of cards guaranteed to lose the current trick. These heuristics optimize the next trick only *after* optimizing the current trick. Heuristic *H5c* says to select a maximal rank card from the guaranteed losers. Heuristic *H6c* says to select a maximal point value card from the guaranteed losers. Heuristics *H5c* and *H6c* were derived by manually examining the numerical values returned by theories *T5* and *T6*.[29] They should not be surprising, considering the small numerical values of *Exp-Next-Win* and *Exp-Next-Trick-Value*.

POLLYANNA generates some hearts theories that cannot be easily described in terms of verbal heuristics. In particular, some theories cannot be written in terms of simple *IF-THEN* rules. Two such approximate theories are shown in Figure 3-8. Hearts theory *T7* uses the sophisticated versions of *Exp-Win* and *Exp-Trick-Value* to analyze the current trick, while ignoring the next trick entirely. It is therefore called *"Optimize the Current Trick"*. Hearts theory *T8* uses the sophisticated versions of all four functions. It is therefore called *"Optimize the Current and Next Tricks"*. These theories are difficult to describe due to the details of the numerical formulae they use to combine four decision criteria: *Exp-Win*, *Exp-Trick-Value*, *Exp-Next-Win*, *Exp-Next-Trick-Value*. The criteria are combined in a manner such that no one factor is strictly more important than the others. The two theories therefore cannot be

[29]Heuristic *H5c* was derived by manually differentiating the evaluation function of theory *T5* with respect to *Rank(c)*, and then showing that the derivative is always positive. Heuristic *H6c* resulted from considering the most extreme case, i.e., when the lead card is ♠Jack, player *p* is playing third and is deciding between the ♠10 and the ♠Queen. *Exp-Win* is minimized by the ♠10, while *Exp-Next-Trick-Value* is minimized by the ♠Queen. Theory *T5* recommends playing the ♠10, to avoid winning the current trick. (The right choice, for entirely the wrong reason.)

reduced to a set of *IF-THEN* rules that implement successive optimization over these four criteria. They specify a true balancing of conflicting goals.

7. *Optimize Current Trick:*

Theory: (EWN-1, ETV-1, ENW-0, ENT-0)

```
Exp-Game-Score(c,p,t,...) =

 = If [(∀ c' in Table) Defeats(c,c',Lead-Suit)]
      then [B1 + (N*B2) + Point-Value(c)] * [A1*{A2 + Rank(c)}]^N
      else 0
         +
      Constant
```

8. *Optimize Current and Next Tricks:*

Theory: (EWN-1, ETV-1, ENW-1, ENT-1)

```
Exp-Game-Score(c,p,t,...) =

 = {If [(∀ c' in Table) Defeats(c,c',Lead-Suit)]
       then [B1 + (N*B2) + Point-Value(c)] * [A1*{A2 + Rank(c)}]^N
       else 0 }
          +
   {[D1 + D2*Point-Sum(Hand(p)-{c})] * [C1 + C2*Rank-Sum(Hand(p)-{c})]^3}
```

Figure 3-8: Sophisticated Hearts Theories

The difficulty of verbal characterization is illustrated by hearts theory *T7* (*Optimize the Current Trick*). It provides a numerical formula for balancing the goals of minimizing trick value and minimizing the odds of winning the trick. Consider what happens when player *p* is the leader for the current trick. Cards of low rank will more likely avoid winning the trick. Cards of low point value will likely lead to a lower trick value. It turns out that leading the ♥2 is preferred to the ♦10 and to other diamonds of higher rank, i.e., a lower rank card is preferred. It also turns out that leading the ♥2 is not preferred to the ♦9 or to other diamonds of lower rank, i.e., lower point value cards are preferred. Neither of the two conflicting goals is strictly more important than the other.

A different story is told by hearts theory *T8* (*Optimize the Current and Next Tricks*). This theory provides a formula for balancing the goals of optimizing the current and next tricks of the game. As it turns out, theory *T8* always considers optimizing the current trick to be more important than optimizing the next trick. Theory *T8* also provides a formula for balancing the goals of next trick value and winning the next trick. As it turns out, theory *T8* always considers the expected next trick value to be more important than the probability of winning the next trick. High point value cards are therefore dumped before cards of high rank.

3.3.1.3. The Generality of Hearts Heuristics

Generality of a heuristic can be analyzed by examining which parts of the domain theory were used in either (a) mechanical derivations of approximations or (b) manual proofs of semantic equivalence between approximations and verbal heuristics. Suppose the λ-expression defining a function *f* from the initial domain theory was never touched by any of the *GSA* or reformulation operators used in a mechanical derivation. Any approximate theory resulting from such a derivation will be independent of the manner in which *f* was defined, i.e., the same approximate theory would be derivable if the definition of *f* were modified or entirely absent. If the definition of *f* was also not used in proving semantic equivalence to verbal heuristics, *POLLYANNA* would capable of generating approximate theories equivalent to the same verbal heuristics as well. The verbal heuristics will be general enough to apply to alternate versions of the domain theory in which the function *f* has a different definition.

POLLYANNA's approximate hearts theories and associated heuristics are more general than they might at first appear. As an example of such generality, consider the *Point-Value* function. The implemented version of this function assigns one point to each heart, thirteen points to the queen of spades and zero points to all other cards. (See Appendix C.) This definition could be changed to define a simpler version of hearts in which the queen of spades is worth zero points. It could also changed to define a more advanced version of the game in which the jack of diamonds is worth negative ten points. The definition of *Point-Value* was not used in any of the mechanical derivations approximate hearts theories. All the approximate hearts functions appearing in the approximation generator final state would remain derivable if the *Point-Value* function were changed. (See Appendix J.) Instead of actually replaying the derivations, it would suffice to replace the old definition of *Point-Value* with new revised definition, in the final state of the approximation generator.

The definition of *Point-Value* was also not used in manually deriving the simpler algebraic forms of approximate hearts functions that are shown in Figures 3-4 and 3-8. (See Appendix K). These equivalent forms thus remain invariant with respect to changes in the definition of *Point-Value*. The definition was also not used in proving semantic equivalence to any of the verbal heuristics that depend only on monotonicity properties of the functions in Figure 3-4, i.e., the heuristics in Figure 3-6. The definition of *Point-Value was* used in deriving the two verbal heuristics that cannot be analyzed by monotonicity properties alone, i.e., the heuristics in Figure 3-7. These heuristics might be different if the definition of *Point-Value* were changed.

3.3.1.4. Additional Heuristics for Hearts

Additional hearts heuristics would result from using *POLLYANNA*'s approximation generator to explore deeper regions of the hearts random variable tree in Figure 2-10. Although the approximation generator has not been used for this purpose, it is interesting to speculate about the types of theories that might be generated. For example, more sophisticated theories would likely result from retracting the assumption that opponents are equally likely to play any card in the deck. (*Equiprobable Random Variables (EP)* applied to the random variable $\lambda(d)Card(p,t,d)$.) By unfolding the definition of *Card*, such theories could attempt to analyze which cards would be legal and optimal for opponents to play. By considering that opponents cannot *legally* play cards that appeared in previous tricks, approximate theories might be sensitive to the past history of the game. Such theories would assume opponents are equally likely to play any card *out*, i.e., unplayed cards not in the current player *p*'s hand. The derivations of these theories would be quite similar to many of the derivations used to generate the approximate theories described above. These theories could probably be generated by a straightforward application of *POLLYANNA*'s existing methods. In contrast to this, serious difficulties must be overcome in order to generate theories that analyze the *optimality* of opponents' choices. If such theories can be generated by *POLLYANNA*, they would likely be inefficient. (These difficulties will be described in Section 3.6.)

3.3.2. Heuristics for Job Scheduling

3.3.2.1. Approximation Generator Output in Scheduling

A summary of one final state produced by *POLLYANNA*'s approximation generator for the scheduling domain is found in Figure 3-9. The principal functions in the final state are listed along with a description of referencing relationships. The diagram also indicates which functions have multiple versions, and the number of versions of each. A total of 24 distinct approximate theories can be formed by systematically combining versions of scheduling functions. Some of the key scheduling functions are defined in Figure 3-11. Notice that the reference tree in Figure 3-9 is similar, but not identical to the random variable tree in Figure 2-12. The actual output of *POLLYANNA*'s approximation generator for the scheduling domain is found in Appendix L.[30] This data was generated using the scheduling theory representation found in Appendix D.

The top level function *Evaluation(i,t,g)* is used to evaluate a state *g* that appears in the search space of a specific instance *i* of the general job scheduling problem defined in Figure 2-11. *Evaluation* could be used to make search control decisions for a variety of search algorithms, including *A**, hill climbing or beam search, among

[30]Only a partial description of the final state is found in Appendix L. All versions computing exact probabilities or expectation values are intractable, and were therefore omitted.

```
Evaluation [3]
 |
 |---Eval-Static
 Or
 |---Evaluation-A
 |    |---Eval-Static
 Or
 |---Evaluation-B
      |---Evaluation-A
           |---Eval-Static

Eval-Static
 |
 |---Exp-Job-Value
      |---Exp-Preconditions [2]
      |    |---Exp-Check-Precond
      |         |---Prob-Prec-Comp-Alloc
      |              |---Prob-Prec
      |              |---Prob-Comp [2]
      |              |    |---Prob-Alloc
      |              |
      |              |---Prob-Alloc
      |
      |---Exp-Penalty [2]
      |    |---Prob-Comp [2]
      |         |---Prob-Alloc
      |
      |---Exp-Weight
```

Figure 3-9: Reference Relations among Scheduling Functions

others. The parameter g represents a partial schedule in which all slots at times earlier than t have been allocated to jobs. The parameter t represents the time of the first open slot. *Evaluation* estimates the value of the best schedule that is consistent with the job assignments described by the state parameter g. Several different versions of *Evaluation* appear in the approximation generator final state: *EVL-0, EVL-1* and *EVL-2*. These versions respectively call the functions: *Eval-Static*, *Evaluation-A* and *Evaluation-B*. They perform static evaluation of states after looking ahead *0*, *1*, or *2* levels in the search tree.

The static evaluator has the form *Eval-Static(i,g)*. It computes the conditional expectation value of the random variable $\lambda(e)Value(i,e)$, given the job assignments specified by the parameter g. *Eval-Static(i,g)* is simply a sum over all jobs j of the expected job value for each job: *Exp-Job-Value(i,g,j)*. Using an assumption of *Probabilistic Independence (IN)*, *Exp-Job-Value* can be written as the product of three other expectation values *Exp-Penalty*, *Exp-Preconditions* and *Exp-Weight*. *Exp-Penalty* computes the probability that a job is successfully completed by its deadline. *Exp-Preconditions* computes the probability that all the preconditions of a job j will be successfully completed by the time t to which job j is assigned. *Exp-Weight* is just the expectation of the random variable *Weight*, which reflects the importance of a job.

```
Eval-Static(i,g) = ∑ (j in Jobs(i)) Exp-Job-Value(i,g,j)

Exp-Job-Value(i,g,j) = Exp-Preconditions(i,g,j)
                              * Exp-Penalty(i,g,j)
                              * Exp-Weight(i,g,j)
```

Figure 3-10: Scheduling Static Evaluation Function

Key scheduling functions called by the static evaluator are described in Figure 3-11. The functions *Exp-Penalty*, and *Exp-Preconditions* are particularly interesting. Each has one naive and one sophisticated version. Selecting the sophisticated version *EPN-1* of *Exp-Penalty* and the naive version *EPC-0* of *Exp-Preconditions*, the resulting theory will focus more on the issue of job completion deadlines and will ignore the issue of issue of satisfying preconditions. Selecting the sophisticated version *EPC-1* of *Exp-Preconditions* and the naive version *EPN-0* of *Exp-Penalty*, the resulting theory will focus more on the issue of satisfying preconditions, and will ignore the issue of job completion deadlines. The sophisticated versions of *Exp-Penalty,* and *Exp-Preconditions* both directly or indirectly call the function *Prob-Alloc*, to determine the odds that a job *j* is assigned to a specific time *t*. The function *Prob-Alloc* has only one version. It uses *Equiprobable Random Variables (EP)* to assume that all unscheduled jobs are equally likely to be assigned to any open time slot.

- *Exp-Penalty(i,g,j) (EPN-0, EPN-1):* The probability that job *j* is finished by its completion deadline. The naive version *EPN-0* uses *Equiprobable Random Variables (EP)* to assume that all jobs are as likely as not to complete on time. The sophisticated version *EPN-1* operates by considering each time slot *t*, and calling *Prob-Comp(i,g,j,t)* to determine the odds that job *j* completes at time *t* precisely.
- *Exp-Preconditions(i,g,j) (EPC-0, EPC-1):* The probability that all the preconditions of job *j* are successfully completed by the time to which job *j* is assigned. The naive version *EPC-0* uses *Equiprobable Random Variables (EP)* to assume that the preconditions of all jobs are as likely as not to be satisfied. The sophisticated version *EPC-1* operates by computing the product of the probabilities that each precondition *p* of job *j* is satisfied. These in turn are computed by considering all possible completion times t_c for precondition *p*, all possible allocation times t_a for job *j*. The functions *Prob-Comp* and *Prob-Alloc* are called to determine the probability of each pair (t_c,t_a). The sophisticated version *EPC-1* also uses *EP* to assume that the preconditions of preconditions are as likely as not to be satisfied.
- *Prob-Comp(i,g,j,t) (PCM-0, PCM-1):* The probability that job *j* finishes at time *t* precisely. The naive and sophisticated versions both call the function *Prob-Alloc(g,j,t')* to determine the odds that job *j* is assigned to a time *t'* that would result in completion at time *t*. The naive version *PCM-0* uses *Function Invariance (FI)* to assume that all jobs require the same amount of time (zero) to complete once they begin. The sophisticated version *PCM-1* takes account of the fact that jobs require differing amounts of working time.
- *Prob-Alloc(g,j,t):* The probability that job *j* is assigned to begin at time slot *t*. This function has only one version. It first checks the state parameter *g* to see if a time slot was already allocated to job *j*, or if a job was already assigned to time *t*. If such cases, the probability is determined to be either zero or one, depending on whether job *j* and time *t* are matched in state *g*. In the remaining cases, *Prob-Alloc* uses an assumption of *Equiprobable Random Variables (EP),* and returns the constant *1/|Range(Allocation-Time)|*. Unscheduled jobs are therefore assumed equally likely to be allocated any one of the open time slots.

Figure 3-11: Selected Scheduling Functions

3.3.2.2. Equivalence of Scheduling Heuristics and Approximate Theories

Several interesting heuristics result from the data produced by *POLLYANNA*'s approximation generator for the scheduling domain. Two scheduling heuristics will be discussed in this section. The *Soonest Scheduling Deadline First* heuristic is described in Figure 3-12. The *Most Critical Job First* heuristic is described in Figure 3-13. Each heuristic is associated with an approximate theory. The approximate theories are described as tuples listing versions of scheduling functions appearing the final state of the approximation generator described in Appendix L. Each theory is also described by an equation that defines the semantics of *Exp-Job-Value* under the corresponding approximate theory. These equations are semantically equivalent to actual definitions appearing in Appendix L. The simpler equivalent forms are derived in Appendix M.

The *Soonest Scheduling Deadline First* heuristic is described in Figure 3-12. Under this heuristic, jobs are ranked according to their scheduling deadlines. A scheduling deadline is the latest time slot to which a job can be assigned and still complete on time. The heuristic is equivalent to *Scheduling Theory #1* also described in Figure 3-12. This approximate theory results from selecting the sophisticated version *EPN-1* of *Exp-Penalty* and the naive version *EPC-0* of *Exp-Preconditions*. It focuses on the issue of job completion times, and ignores the issue of preconditions. It uses the sophisticated version *PCM-1* of *Prob-Comp*, which takes account of the varying amounts of work required to perform jobs.

A simplified equivalent form of *Scheduling Theory #1* is shown in Figure 3-12. (See Appendix M.) It defines a version of *Exp-Job-Value*, which is called by *Eval-Static* to compute the expected value of each job. (See Figure 3-10.) To find the expected value of a job *j*, this function really only computes *Exp-Penalty*, the probability that job *j* completes on time. The other factors in *Exp-Job-Value* are set to constants. To determine *Exp-Penalty*, this function first checks to see if job *j* has already been allocated a time slot in state *g*. If job *j* has been assigned to a slot, the expected job value is a positive constant or zero, depending on whether *j* completes by its deadline. If job *j* has not yet been assigned to a slot, the expected job value is computed using *Equiprobable Random Variables (EP)* to assume that *j* is equally likely to be allocated any one of the open time slots. Job *j* will complete on time only if allocated a slot no later than its scheduling deadline. The expected value is therefore proportional to the number of open time slots up to, and including, job *j*'s scheduling deadline. The specific simplifying assumptions used to generate *Scheduling Theory #1* are listed in Appendix M.

The behavior of *Scheduling Theory #1* can be understood by considering the role of *Exp-Job-Value* in the evaluation function *Eval-Static* described in Figure 3-10. Suppose that *Eval-Static(i,g)* is used to compare two states G_a and G_b that respectively result from assigning jobs J_a and J_b to the current time slot *Now(g)* in state *g*. Notice

Soonest Scheduling Deadline First:

```
IF (Some unscheduled job has not yet passed its scheduling deadline)
   THEN (Select an unscheduled job with a soonest scheduling deadline)
   ELSE (Select any unscheduled job).
```

Scheduling Theory #1: (EVL-0, EPN-1, EPC-0, PCM-1)

```
Exp-Job-Value(i,g,j) =

 = If (Job j is assigned to some slot t in state g)
      then {If [t + Working-Time(j,i) ≤ Deadline(j,i)]
               then A0 * A1
               else 0 }
      else { A0 * A1 * A2 * [S-Deadline(j,i) + 1 - Now(g)] }
```

Definitions:

```
Deadline(j,i) = Job j's completion deadline.
Working-Time(j,i) = The time to perform job j.
S-Deadline(j,i) = Deadline(j,i) - Working-Time(j,i)
Now(g) = The time of the first open slot in state g.

A0 = Average(Range(Preconditions))
A1 = Average(Range(Weight))
A2 = 1/|Range(Allocation-Time)|
```

Figure 3-12: *Soonest Scheduling Deadline First*

that the expected values *Exp-Job-Value* for all jobs other than J_a and J_b are the same in states G_a and G_b. These terms can therefore be ignored in the comparison of states G_a and G_b. Suppose further that neither J_a nor J_b is past its scheduling deadline. Therefore the expected value of J_a in state G_a and the expected value of J_b in state G_b are both equal to $A_0{*}A_1$, as indicated in Figure 3-12. Now suppose that the scheduling deadline of J_a is quite near while the scheduling deadline of J_b is far away. Job J_a is unlikely to complete on time if not scheduled now. The expected value of J_a in state G_b is therefore quite small. Job J_b is likely to complete on time regardless of whether it gets scheduled now. The expected value of J_b in state G_a is therefore rather large. In comparing the states G_a and G_b, *Static-Eval* determines that a choice of J_a improves the expected value of J_a more than a choice of J_b improves the expected value of J_b. State G_a is therefore ranked higher than state G_b. A job with an early scheduling deadline is preferred over a job with a late scheduling deadline.

The *Most Critical Job First* heuristic is described in Figure 3-13. It recommends picking a job that best contributes to satisfying preconditions of other jobs. This version of the heuristic does not specify precisely how to determine the job that best contributes to satisfying preconditions. Two specialized versions of this heuristic are more precise, but they apply only to some states in the search spaces of scheduling problems. They are shown in Figure 3-14 and will be discussed below. The *Most Critical Job First* heuristic is equivalent to *Scheduling Theory*

#2, also described in Figure 3-13. This approximate theory results from selecting the sophisticated version *EPC-1* of *Exp-Preconditions* and the naive version *EPN-0* of *Exp-Penalty*. It focuses on the issue of satisfying preconditions and ignores the issue of job completion times. It uses the naive version *PCM-0* of *Prob-Comp*, which assumes all jobs require the same amount of time (zero) to complete after they begin.

Most Critical Job First:

```
 Choose an unscheduled job that best contributes to satisfying
 preconditions of other unscheduled jobs.


Scheduling Theory #2: (EVL-0, EPN-0, EPC-1, PCM-1)

 Exp-Job-Value(i,g,j) =

   = B0 * B1 * { B2^|Requirements(j,i)| }
     * Π (p in Requirements(j,i))
         Case1: [Jobs j and p are allocated slots tj and tp in state g.]
                Return {If (tp ≤ tj) then 1 else 0 }
         Case2: [Only j is allocated a slot in state g.]
                Return 0
         Case3: [Only p is allocated a slot in state g.]
                Return { B3 * Slots(g) }
         Case4: [Neither j nor p is allocated a slot in state g.]
                Return { ( B3^2 ) * Slots(g) * (Slots(g) - 1) / 2 }

Definitions:

 Requirements(j,i) = The preconditions of job j.
 Slots(g) = The number of open time slots in state g.

 B0 = Average(Range(Penalty))
 B1 = Average(Range(Weight))
 B2 = 1 / 2
 B3 = 1 / |Range(Allocation-Time)|
```

Figure 3-13: *Most Critical Job First*

A simplified equivalent form of *Scheduling Theory #2* is shown in Figure 3-13. (See Appendix M.) It defines a version of *Exp-Job-Value*, which is called by *Eval-Static* to compute the expected value of each job. (See Figure 3-10.) To find the expected value of a job *j*, this function really only computes *Exp-Preconditions*, the probability that every precondition of job *j* is successfully completed by the time to which job *j* is assigned. The other factors in *Exp-Job-Value* are set to constants. Using an assumption of *Probabilistic Independence (IN)*, *Exp-Preconditions* is computed as the product of the probabilities that each individual precondition is satisfied. These in turn are computed by using *Equiprobable Random Variables (EP)* to assume that each unassigned job *j* or unassigned precondition *p* is equally likely to be assigned to any open time slot. An additional *EP* assumption asserts that the (recursive) preconditions of any precondition are as likely as not to be satisfied.[31]The specific simplifying assumptions used to generate *Scheduling Theory #2* are listed in Appendix M.

The simplified version of *Exp-Job-Value* is written in the form of a case statement. (See Figure 3-13). The four cases depend on whether job *j* and/or precondition *p* are allocated slots in the current state *g*. If both *j* and *p* are assigned *(Case1)*, to times t_j and t_p, the precondition is satisfied if and only if $t_p \le t_j$. Using the assumption that jobs require zero working time, precondition *p* needs only to begin by the time job *j* begins. If *j* is assigned and *p* is not *(Case2)*, the precondition cannot be satisfied, since *p* must be eventually be assigned to time later than the starting time for job *j*. If *p* is allocated and *j* is not *(Case3)*, then the precondition is satisfied for each open time slot to which *j* might eventually be assigned. In this case the precondition is satisfied with probability proportional to *Slots(g)*, the number of open time slots in state *g*. If neither *j* nor *p* is assigned *(Case4)*, the precondition is satisfied whenever *j* and *p* are eventually allocated times t_p and t_j such that $t_p \le t_j$. Since jobs *j* and *p* are equally likely to be allocated any pair of open slots, the precondition is satisfied with probability proportional to *Slots(g)* choose two. Notice that a single precondition *p* of an unassigned job *j* is more likely to be satisfied under *Case3*, in which *p* is assigned, than under *Case4*, in which *p* is not yet assigned. (Since Slots(g) must be less than |Range(Allocation-Time)|, the total number of time slots.)

Two special cases of *Most Critical Job First* are described in Figure 3-14. These two heuristics describe the behavior of *Scheduling Theory #2* under restricted circumstances. The heuristic called "Enable Many Jobs" applies to states in which every unassigned job has at most one precondition. The heuristic called "Enable a Well Prepared Job" applies to states in which every unassigned job is a precondition for at most one other unassigned job. Taken together these rules apply to all states for which the unassigned portion of the activity network has a tree structure. (See Section 2.4.3.) Under such conditions, the behavior of *Scheduling Theory #2* can be described more concisely. The approximate theory itself is more general than the special case heuristics in Figure 3-14. It can be used to evaluate any state in terms of the degree to which preconditions of jobs are likely to be satisfied.

The *Enable Many Jobs* rule can be understood by considering the role of *Exp-Job-Value* in the evaluation function *Eval-Static* described in Figure 3-10. Suppose *Eval-Static(i,g)* is used to compare two states G_a and G_b that respectively result from assigning jobs J_a and J_b to the current time slot *Now(g)* in state *g*. Suppose further that sets $A=\{a_1,...,a_n\}$ and $B=\{b_1,...,b_m\}$ hold all the unscheduled jobs enabled by J_a and J_b respectively. Notice that the expected values *Exp-Job-Value* for all jobs outside of sets *A* and *B* are the same in states G_a and G_b and can

[31]The factor of $B_2^{|Requirements(j,i)|}$ in the definition of *Exp-Job-Value* in Figure 3-13 results from the *EP* assumption asserting that the recursive preconditions of each precondition *p* itself are as likely as not to be satisfied.

Enable Many Jobs:

```
IF:   All unscheduled jobs have at most one precondition.
THEN: Choose an unscheduled job that is a precondition to the
      greatest number of other unscheduled jobs.
```

Enable a Well Prepared Job:

```
IF:   All unscheduled jobs enable at most one unscheduled job.
THEN: Choose a job that is a precondition to a best
      prepared unscheduled job.
```

Figure 3-14: Special Cases of *Most Critical Job First*

therefore be ignored. Assuming the application condition of *Enable Many Jobs*, each a_i and each b_i has exactly one precondition. The product appearing in *Exp-Job-Value* therefore reduces to a single factor. (See Figure 3-13.)

When *Exp-Job-Value* is called with state G_a as parameter, all jobs in set A fall under *Case3* and have a relatively high expected job value, while all jobs in set B fall under *Case4* and have a relatively low expected job value. Likewise when *Exp-Job-Value* is called with state G_b as parameter, all jobs in set B fall under *Case3*, while all jobs in set A fall under *Case4*. Suppose that J_a enables more jobs than J_b. A choice of J_a therefore places more jobs under *Case3* resulting in high expected value. A choice of J_b places more jobs under *Case4* resulting in low expected value. State G_a therefore receives a higher evaluation than state G_b. A job enabling many unscheduled jobs is preferred over a job enabling few unscheduled jobs.

The *Enable a Well Prepared Job* rule can be also understood by considering the role of *Exp-Job-Value* in the evaluation function *Eval-Static* described in Figure 3-10. Suppose *Eval-Static(i,g)* is used to compare two states G_a and G_b that respectively result from assigning jobs J_a and J_b to the current time slot *Now(g)* in state *g*. Suppose further that job J_a enables job A and that job J_b enables job B. Assuming the application condition of *Enable a Well Prepared Job*, jobs J_a and J_b must serve as preconditions to no other jobs. States G_a and G_b can therefore be compared in terms of the expected values of jobs A and B alone. The expected values of all other jobs can be ignored.

Job A fares better in state G_a than in state G_b. It falls under *Case3* with high value in state G_a and falls under *Case4* with low value in state G_b. The difference between the value of A in state G_a and the value of A in state G_b is proportional to the probability that all preconditions of A other than J_a are satisfied. Likewise job B does better in state G_b than in state G_a. The difference is again proportional to the probability that all preconditions of B other than J_b are satisfied. Job A is considered "better prepared" than job B if the additional preconditions of A are more likely to be satisfied than are the additional preconditions of B. If job A is better prepared that job B, then state G_a receives a higher evaluation than state G_b. A job enabling a well prepared job is therefore preferred over a job enabling a poorly prepared job.

3.3.2.3. Additional Heuristics for Job Scheduling

POLLYANNA generates several interesting variations on the scheduling heuristics described above. Consider what happens to *Soonest Scheduling Deadline First* when the naive version of *Prob-Comp (PCM-0)* is used instead of the sophisticated one *(PCM-1)* in *Scheduling Theory #1*. The naive version assumes all jobs take the same amount of time (zero) to complete after they are started. All scheduling deadlines are therefore equal to job completion deadlines. The resulting theory ranks jobs according to completion deadlines: *Soonest Completion Deadline First*. This theory would probably be most useful when the working time for each job is difficult to compute.

Consider also what happens to *Most Critical Job First* when the sophisticated version of *Prob-Comp (PCM-1)* is used instead of the naive one *(PCM-0)* in *Scheduling Theory #2*. The sophisticated version actually takes account of the different working times for each job. The resulting approximate theory can be simplified into a form quite similar to the one in Figure 3-13. The first case will test whether the allocation time t_p of precondition p is at least *Working-Time(p,i)* before the allocation time t_j of job j. The third and fourth cases will be changed by substituting *Slots(g)-Working-Time(p,i)* for *Slots(g)*. The resulting theory tends to avoid choosing to schedule a job with a long working time. Jobs taking a long time to complete are not likely to finish soon enough to effectively fulfill preconditions of other jobs: *Most Critical Short Job First*.

Additional scheduling heuristics would result from using *POLLYANNA*'s approximation generator to explore deeper regions of the scheduling random variable tree in Figure 2-12. For example, a more sophisticated analysis of precondition satisfaction would follow from retracting the assumption that all preconditions of other preconditions are as likely as not to be satisfied. (*Equiprobable Random Variables (EP)* applied to recursive calls of the random variable *Preconditions*). The resulting approximate theories would reason about preconditions chains up to length N in the activity network, after unfolding the random variable *Preconditions* through N levels of recursion.

3.4. Covered and Discovered Heuristics

3.4.1. Coverage of the Heuristic Hearts Rule Set

POLLYANNA's analytic learning capabilities can be evaluated by comparing approximate hearts theories to the benchmark heuristic rule set found in Appendix B. The benchmark implements a complete card playing strategy that was described in Section 2.2.7. The approximate theories are quite similar to the benchmark in some respects. Both use the same general criteria for choosing cards, i.e., minimizing the odds of winning and the expected trick value in the current and future tricks. One particular area of overlap involves the *Dump High Rank Rule* and the *Dump High Point Value Rule*. These two rules are implicit in both the benchmark rule set and in some approximate theories generated by *POLLYANNA*.

The approximate theories differ from the benchmark in their method of combining the four decision criteria. The approximate theories combine the four criteria algebraically in the form of the evaluation function shown in Figure 3-2. The evaluation function allows a continuous tradeoff between competing factors. In contrast to this, the benchmark rules implement a priority scheme for jointly optimizing the different criteria. The criteria are ordered so that primary criteria are optimized first, while secondary criteria are used to break ties. The approximate theories reduce to a priority scheme in some cases, e.g. the current trick terms always outweigh the next tricks. Within the current trick term, neither the expected trick value nor the probability of winning is strictly more important than the other.

The most sophisticated approximate hearts theory (*T8* = "Optimize Current and Next Tricks") is compared to the benchmark rule set in Appendix B. Theory *T8* generates exactly the same card choices as eight out of the nine rules. The inconsistency occurs when a player is deciding whether to lead a low heart or a high non-heart. The rules prefer leading any non-heart to leading any heart. Hearts theory *T8* may prefer either hearts or non-hearts depending on their ranks. The rules assert the priority of trick value over the odds of winning. The theory attempts to balance the two concerns.

POLLYANNA most sophisticated approximate hearts theory (*T8* = "Optimize Current and Next Tricks") is actually much more general than the benchmark rule set. It applies to a variety of versions of the game that differ only in the point values of cards. (See Section 3.3.1.3.) In contrast, the goal rules were written for a version in which hearts have one point each, and no other cards have any point value. They take a simple form because all point cards are in one suit. A much larger rule set is required to implement the same strategy if the queen of spades is worth thirteen points. The approximate theory *T8* captures this and other versions for free. *POLLYANNA*'s most sophisticated theory thus extends the benchmark strategy to more complex versions of the game.

3.4.2. Coverage of Known Scheduling Heuristics

Comparisons between *POLLYANNA*'s scheduling heuristics and known human scheduling heuristics are difficult to make. Differences in problem formulation are the principal difficulty. The scheduling problem implemented in *POLLYANNA* is somewhat simpler than the ones usually considered in the scheduling literature. Scheduling research typically studies problems of allocating multiple processors to multiple jobs [Stokey 89], in contrast to the single processor problem implemented in *POLLYANNA* (See Figure 2-11). Real life scheduling problems usually to involve simultaneous consideration of many distinct types of constraints and preference criteria [Smith et al. 86]. In contrast to this, the scheduling problem investigated here involves only one preference criterion (the value of the schedule) and two types of constraints (preconditions and deadlines).

Despite differences in problem formulation, some rough comparisons can be drawn. *POLLYANNA*'s *Soonest Scheduling Deadline First* heuristic is similar in flavor to some heuristics appearing in the literature. One such similarity involves heuristics that measure the *slack* associated with a single job [Davis and Patterson 75]. Job slack is computed through analysis of the project activity network. Slack is defined as the difference between (a) the latest possible starting time for a job that still allows a the whole project to complete by a project deadline and (b) the earliest time at which all preconditions of a job can possibly be complete. Slack heuristics are similar to *Soonest Scheduling Deadline First*, since both measure the range of starting times that are consistent with some deadline; however, slack is defined in terms of project deadline, rather than the deadline of an individual job. A more distant similarity involves heuristics that measure *tightness* of various constraints bearing on an individual job, and choose tightest jobs first [Fox et al. 89]. These comparisons are meant only to be suggestive. The heuristics generated by *POLLYANNA* cannot be claimed semantically equivalent to those involving slack and tightness, at the very least because these previously known heuristics were devised for different and more complex scheduling problems. The similarities do suggest that job scheduling would be a fruitful application for further developing the approximation techniques used in *POLLYANNA*.

3.4.3. The Originality of Generated Heuristics

Originality of heuristics in *POLLYANNA* is a direct consequence of the generality of generic simplifying assumptions. *POLLYANNA*'s *GSAs* were developed by attempting to reconstruct known hearts heuristics. Most of the reconstruction effort focused on the *Dump High Rank Rule* and the *Dump High Point Value Rule*. The resulting *PT-GSAs* and *PT-FORMALISM* turned out to be significantly more general than the two hearts heuristics. New heuristics resulted from systematically applying the *PT-GSAs* to various different parts of domain theories. Heuristics generated by *POLLYANNA* can be classified according to three levels of originality: "reconstructed", "rediscovered" and "discovered". (See Section 3.1.) This classification is summarized in Figure 3-15.

Heuristics anticipated by the designer of *POLLYANNA* are designated "reconstructed heuristics". Reconstructed hearts heuristics include *H4* (*Lose the Current Trick*) and *H5* (*Lose and Dump High Ranks*) as well as some cases of heuristic *H6* (*Lose and Dump High Ranks*). In particular, this category includes hearts heuristics *H5a* and *H6a*, which are respectively equivalent to the *Dump High Rank Rule* and the *Dump High Point Value Rule*. Almost all the rules in the goal set of heuristics also fall into this group of "reconstructed" heuristics. (See Appendix B.)

Heuristics not anticipated by the designer of *POLLYANNA* are considered to be either "discovered" or "rediscovered". Distinctions between these two categories are sometimes difficult to make. If a generated heuristic

RECONSTRUCTED:		
Hearts:	*Lose the Current Trick.*	**(H4)**
	Lose and Dump High Ranks.	**(H5)**
	Lose and Dump High Point Values.	**(Cases H6a,c)**
	Dump High Rank Rule.	**(Case H5a)**
	Dump High Point Value Rule.	**(Case H6a)**
	Most of Heuristic Goal Set.	
Scheduling:	*Soonest Deadline First.*	
REDISCOVERED:		
Hearts:	*Random Play.*	**(H0)**
	Minimize Points.	**(H1)**
	Maximize Points.	**(H2)**
	Maximize Rank.	**(H3)**
	Lose and Dump High Point Values.	**(Case H6b)**
Scheduling:	*Soonest Scheduling Deadline First.*	
	Enable Many Jobs.	
	Enable a Well Prepared Job.	
DISCOVERED:		
Hearts:	*Optimize Current Trick.*	**(T7)**
	Optimize Current and Next Tricks.	**(T8)**
Scheduling:	**None**	

Figure 3-15: Originality of Generated Heuristics

can be shown semantically equivalent to a previously published heuristic, this counts as direct evidence that the heuristic was rediscovered by the system. The problem arises when no equivalent published heuristic can be found. Absence of a published equivalent is only indirect evidence that the heuristic was discovered by the system. Direct evidence of discovery is not possible. All of the heuristics discovered or rediscovered by *POLLYANNA* fall into this category. The distinctions between discovered and rediscovered heuristics made in the following discussion must therefore be considered only to be *estimates* of the exact level originality.

POLLYANNA is particularly good at rediscovering heuristics at the extreme naive end of the spectrum. Two extreme examples are heuristics *H0* (*Random Play*), *H1* (*Minimize Points)* and *H2* (*Maximize Points*) in Figure 3-6. Heuristic *H0* says a player should simply play any card. Heuristic *H1* says he should play any card of minimal point value. Heuristic *H2* says he should play any card of maximal point value. These heuristics are so naive that they tend to be overlooked by humans. Nevertheless they could in principle be useful in a performance context with extremely tight computational resource constraints, e.g., when playing a game with time clocks, and one's time is about to run out.

POLLYANNA has also rediscovered some negative hearts heuristics, i.e., heuristics that are worse than playing randomly. One such example is heuristic *H3* (*Maximize Rank*) in Figure 3-6, which says to play a maximal rank card. This heuristic tries to avoid winning the next trick in a way that indirectly tends to win the current trick. Negative card playing behavior also results from heuristic *H6b*, a special case of *Lose and Dump High Point Values*, shown in Figure 3-6. Heuristic *H6b* says that a player should choose a maximal point value card when he is guaranteed to win the current trick. This strategy diminishes the expected point value of the next trick in a way that indirectly interferes with optimizing the current trick. These two negative heuristics both result from pairs of decision criteria that offer opposite advice. Negative behavior results from ignoring the more important criterion and focusing on the less important conflicting one.

POLLYANNA's most significant discoveries in hearts are the algebraic evaluation functions implemented in theories *T7* (*Optimize the Current Trick*) and *T8* (*Optimize the Current and Next Tricks*), shown in Figure 3-8. Both of these theories involve a previously unknown numerical formula to strike a balance among the decision criteria: *Exp-Win*, *Exp-Trick-Value*, *Exp-Next-Win*, *Exp-Next-Trick-Value*. None of the separate criteria is entirely new. As described above, all four criteria appear implicitly in the benchmark rule set. In addition, the monotonicity properties of all four functions were envisioned by the designer of *POLLYANNA* prior to implementation. Nevertheless *POLLYANNA* should be credited with discovering the specific monotonic functions appearing in the approximate theory, (i.e., linear functions of point value and polynomial functions of rank), along with numerical constants determining the weight of each factor and the method of combining them into a complete evaluation function. *POLLYANNA* thus discovered numerical formulae that strike a balance between competing factors in a decision process. *POLLYANNA* also gets credit for discovering a way of extending the benchmark rule from a simple version of the game to a more complex versions of the game.

Results from the job scheduling domain generally fall into the category of rediscovered heuristics. None of these heuristics were anticipated by the system designer prior to implementation of the *PT-GSAs* and *PT-FORMALISM*. The scheduling domain was selected after implementation of the *PT-GSAs* and *PT-FORMALISM* was complete. Rediscovered scheduling heuristics include <Soonest Scheduling Deadline First>, shown in Figure 3-12. They also include *Enable Many Jobs* and *Enable a Well Prepared Job*, shown in Figure 3-14. These heuristics are not claimed to be discoveries. Their very simplicity suggests that they were previously known to some humans.[32]

[32] The *Soonest (Completion) Deadline First* heuristic was envisioned by the system designer after implementation of the *PT-GSAs* and *PT-FORMALISM*, but prior to implementation of the scheduling domain theory. This heuristic might therefore be placed in the category of reconstructed heuristics. Some type of heuristic focusing on preconditions was also envisioned by the designer prior to implementation of the scheduling theory; however, the two special cases of *Most Critical Job First* were not anticipated. The equivalence of *Enable Many Jobs* and *Enable a Well Prepared Job* to *Scheduling Theory #2* was recognized and proved after the implementation was complete.

3.4.4. The Quality of Generated Heuristics

The heuristics generated by *POLLYANNA* can be classified according to four levels of quality: not plausible, plausible, novice, expert. (See Section 3.1.) This classification is summarized in Figure 3-16. *POLLYANNA* has actually generated heuristics falling into only the first three categories: not plausible, plausible and novice. Examples of heuristics that are not plausible include *H0* (*Random Play*) and *H2* (*Maximize Points*), shown in Figure 3-6. They also include the two negative hearts heuristics *H3* (*Maximize Rank* and *H6b*, a special case of *Lose and Dump High Point Values*, also shown in Figure 3-6.

NOT PLAUSIBLE:

Hearts:	*Random Play.*	**(H0)**
	Maximize Points.	**(H2)**
	Maximize Rank.	**(H3)**
	Lose and Dump High Point Values.	**(Case H6b)**
Scheduling:	**None**	

PLAUSIBLE:

Hearts:	*Minimize Points.*	**(H1)**
	Optimize Current Trick.	**(T7)**
	Optimize Current and Next Tricks.	**(T8)**
Scheduling:	*Soonest Scheduling Deadline First.*	
	Enable Many Jobs.	
	Enable a Well Prepared Job.	

NOVICE:

Hearts:	*Lose the Current Trick.*	**(H4)**
	Lose and Dump High Ranks.	**(H5)**
	Lose and Dump High Point Values.	**(Cases H6a,c)**
	Dump High Rank Rule.	**(Case H5a)**
	Dump High Point Value Rule.	**(Case H6a)**
Scheduling:	**None**	

EXPERT:

Hearts:	**None**
Scheduling:	**None**

Figure 3-16: Quality of Generated Heuristics

Examples of merely plausible heuristics include *H1* (*Minimize Rank*) shown in Figure 3-6. They also include the numerical formulae for balancing decision criteria, provided by hearts theories *T7* (*Optimize the Current Trick* and *T8* (*Optimize the Current and Next Tricks*, shown in Figure 3-8. Examples of merely plausible heuristics also include all the scheduling heuristics described above, e.g. *Soonest Scheduling Deadline First* (Figure 3-12), as well as *Enable Many Jobs* and *Enable a Well Prepared Job* (Figure 3-14).

Examples of novice level heuristics include the *H4* (*Lose the Current Trick*, *H5* (*Lose and Dump High Ranks*) and two special cases of *Lose and Dump High Ranks* (*H6a,c*), shown in Figures 3-6 and 3-7. Novice level heuristics also include the *Dump High Rank Rule (H5a)*, the *Dump High Point Value Rule (H6a)*, which are just special cases of *H5* and *H6*. All of these novice level hearts heuristics appeared in protocols of hearts games played by human novices. Expert level heuristics have not yet been generated by *POLLYANNA*. The reasons for this limitation are discussed in Section 3.6.

3.4.5. Coverage Properties of Automatic Control Strategies

Heuristics generated under human control can be compared to the ones that would result from automatic control strategies. (See Section 2.7 and Figure 2-23.) The comparison is important for establishing the feasibility of a completely automatic version of *POLLYANNA*. The feasibility of complete automation depends in part on the properties of the best heuristics generated by automatic control strategies. Automatic control would ideally generate heuristics that are equivalent or superior to human generated ones. Feasibility depends also on how many other heuristics are generated under automatic control, and the consequent difficulty of separating the desirable ones from the undesirable ones. This second issue is addressed in Section 5.5.

The control procedure *GENERATE* would produce most of the heuristics that resulted from human control, provided the procedure were extended to handle special constructs as described in Section 2.7.4. This claim is supported by comparing the human generated function versions with the four versions (*Version#i (i=0...3)*) generated at each level of recursion by the procedure *GENERATE*. With a few exceptions, the human generated definitions can each be identified with a specific *Version#i* for some variable *h(e)* appearing in the hearts or scheduling random variable tree. The correspondence of versions can be observed in a general way by noticing that the function reference trees in Figures 3-1 and 3-9 are structurally similar to the random variable trees in Figures 2-10 and 2-12. A more detailed comparison requires examining the actual approximation generator output in Appendices J and L.[33]

The extended control procedure would actually produce results that differ from the human generated heuristics in some minor ways. The automatic procedure would not factor summations or carry out some *Function Invariance (FI)* applications that were performed under human control. Failure to factor summations impacts only the efficiency but not the semantics of approximate theories. The missing applications of *FI* are actually truth-

---

[33]The reformulations *Fold* and *UnFold* must be applied to some of the definitions appearing in Appendices J and L to make them correspond to the four versions described in Figure 2-23. The human generated approximations occasionally include extra *Fold* or *UnFold* operations that would not be applied by *GENERATE*.

preserving operations applied to functions that were invariant anyway. *FI* was used in such cases only to enhance opportunities for memoization. The extended control procedure would therefore generate heuristics that are less efficient than the human generated ones, but semantically equivalent nevertheless.

The restricted control procedures *GENERATE-A* and *GENERATE-B* can also be compared to human control strategies. (See Section 2.7.5 and 2-24.) These two procedures provide restrictions that would limit the approximations produced by the extended version of *GENERATE*. The control restrictions embodied in procedure *GENERATE-A* would not cause omission of any human generated heuristics that would otherwise result from using the extended *GENERATE* procedure. The procedure *GENERATE-A* uses meta-knowledge about the representation of problem states to limit insertion of tests for known values of random variables. Exactly the same strategy was used under human control. Tests for known values were inserted under human control only when the variable in question could potentially appear in the problem state. The control restrictions embodied in *GENERATE-A* would therefore not limit coverage of the heuristics generated under human control.

Procedure *GENERATE-B* would result in a set of heuristics including ones similar to those generated under human control, but which are not identical to the human generated ones. This procedure manifests a strategy that produces only distinct problem state abstractions. The human control strategy was not so restricted; however, the differences have little or no consequence in most cases. The automatic and human results would differ only in the values of functions that are equal to constants. *GENERATE-B* would produce approximations with all the same monotonicity properties as those generated under human control. All the hearts heuristics depending only on reasoning about monotonicity would follow as well, i.e., *H1, H2, H3, H4a, H4b, H4c, H5a, H5b, H6a* and *H6b*. Only the heuristics that involve balancing of competing factors would be different, e.g., *H5c* and *H6c*. The automatic procedure would generate different numerical constants, leading to heuristics that strike a different balance between competing factors.

3.5. Generality of the Approximation Generator

The generality of *POLLYANNA*'s approximation generator has been demonstrated by the results obtained in the hearts and scheduling domains. Domain theories for both hearts and scheduling were encoded in the *PT-FORMALISM*. Approximations were generated by applying generic simplifying assumptions from the *PT-GSA* family to each domain theory. The resulting approximations were combined in various ways to form approximate theories. Informal derivations were used to show the approximate theories to be equivalent to plausible heuristics. *POLLYANNA* is thus capable of generating heuristics in at least two distinct domains.

A more detailed evaluation of generality will be presented in this section. Two different types of generality will be considered. Generality *within* a domain is analyzed by comparing the *specific* simplifying assumptions used to generate each of several different approximate theories for the domain. Generality *across* domains is analyzed by comparing the number of times each *GSA* and reformulation operator was used in each domain. The detailed analysis will provide further evidence of *POLLYANNA*'s generality. It will also illustrate the major issues that must be faced in order to further extend *POLLYANNA*'s range of application.

3.5.1. Generality within Domains

Generality within each of the two domains is illustrated by the tables in Figures 3-17 and 3-18. The rows of each table correspond to specific simplifying assumptions used to generate the approximate theories described above. The specific assumptions are listed by number and by the *GSA* operator used to generate the assumptions. Verbal paraphrases of these simplifying assumptions are found in Appendices K and M. Definitions of the *GSA* operators are found in Appendix F. The columns of each table correspond to the approximate theories described above. A cell in row *i* column *j* is marked with an *x* is assumption *i* was used in generating theory *j*.

Generality within a domain occurs at two different levels. Some generality occurs at the level of *generic* assumptions. Notice that most *GSA* operators were used multiple times in each domain. Every distinct use represents a different instantiation of the generic assumption. Some generality also occurs at the level of *specific* assumptions. Notice that every specific assumption is used in more than one approximate theory. For example, hearts assumption number eleven is an *EP* assumption asserting that an opponent is equally likely to play any card in the deck, if his choice is not already known. This assumption is used by four of the six theories listed in Figure 3-17. Scheduling assumption number five is an *EP* assumption asserting that an unscheduled job is equally likely to be allocated any of the open time slots. This assumption is used by both of the scheduling theories. Consider that all approximate theories generated by *POLLYANNA* lead to different sorts of behavior. Both the six hearts theories and the two scheduling theories are mutually inconsistent. This variety results in part from multiple instantiations of individual generic assumptions. The variety also results partly from various different combinations of specific assumptions.

SA:	Hearts Theory: 1	2	3	4	5	6	GSA Operator:
1	x	x	x	x	x	x	Function-Invariance:Constant (FI)
2	x	x	x	x	x	x	Function-Invariance:Constant (FI)
3	x	x	x	x	x	x	Independence-of-Product:Binary (IN)
4	x	x	x	x	x	x	Independence-of-Product:Binary (IN)
5	x	x	x	-	-	-	Equiprobable:Numeric (EP)
6	-	x	x	x	x	x	Equiprobable:Numeric (EP)
7	x	x	-	x	-	x	Equiprobable:Numeric (EP)
8	x	-	x	x	x	-	Equiprobable:Numeric (EP)
9	-	-	-	x	x	x	Independence-Of-Product:Nary (IN)
10	-	-	x	-	x	-	Independence-Of-Product:Nary (IN)
11	x	-	-	x	x	x	Equiprobable:Equality (EP)
12	-	-	x	-	x	-	Equiprobable:Equality (EP)
13	-	-	-	x	x	x	Equiprobable:Equality (EP)
14	x	-	-	x	x	x	Function-Invariance:Constant (FI)
15	-	x	x	-	x	x	Function-Invariance:Constant (FI)
16	-	-	-	x	x	x	Independence:Three-Conjuncts (IN)
17	-	-	x	-	x	-	Independence:Three-Conjuncts (IN)
18	-	-	x	-	x	-	Equiprobable:Equality (EP)
19	-	x	-	-	-	x	Equiprobable:Numeric (EP)
20	-	x	x	-	x	x	Independence:Two-Conjuncts (IN)
21a	-	x	x	-	x	x	Independence:Two-Conjuncts (IN)
21b	-	x	x	-	x	x	Independence:Three-Conjuncts (IN)
22	-	x	x	-	x	x	Independence:Two-Conjuncts (IN)
23	-	x	x	-	x	x	Function-Invariance:Constant (FI)
24	-	x	x	-	x	x	Function-Invariance:Constant (FI)
25	-	x	x	-	x	x	Equiprobable:Numeric (EP)
26	-	x	x	-	x	x	Equiprobable:Boolean (EP)
27	-	x	x	-	x	x	Equiprobable:Boolean (EP)
28	-	x	x	-	x	x	Equiprobable:Equality (EP)
29	-	x	x	-	x	x	Equiprobable:Boolean (EP)
30	-	x	x	-	x	x	Equiprobable:Boolean (EP)

Figure 3-17: Specific Assumption Generality in the Hearts Domain

SA:	Scheduling Theory: 1	2	GSA Operator:
1	x	-	Equiprobable:Numeric (EP)
2	-	x	Equiprobable:Numeric (EP)
3	x	x	Equiprobable:Numeric (EP)
4	-	x	Equiprobable:Boolean (EP)
5	x	x	Equiprobable:Equality (EP)
6	x	x	Function-Invariance:Constant (FI)
7	-	x	Function-Invariance:Constant (FI)
8a	x	x	Independence-of-Product:Binary (IN)
8b	x	x	Independence-of-Product:Binary (IN)
9	-	x	Independence-of-Product:Nary (IN)
10	-	x	Independence:Three-Conjuncts (IN)
11	x	x	Independence:Two-Conjuncts (IN)
12	-	x	Independence:Two-Conjuncts (IN)
13	-	x	Function-Invariance:Constant (FI)

Figure 3-18: Specific Assumption Generality in the Scheduling Domain

3.5.2. Generality across Domains

Generality across domains will be analyzed by separately considering the *GSA* operators and reformulation operators used in *POLLYANNA*. The *GSA* operators can be analyzed using the table in Figure 3-19. This table shows the number of times individual *GSA* operators were used in each of the two domains. Notice that each major type of GSA, *Function Invariance (FI), Equiprobable Random Variables (EP)* and *Probabilistic Independence (IN),* was instantiated several times in each domain. Notice also that every individual *GSA* operator was used at least once in each domain. The *GSA* operators were developed and implemented for the purpose of generating heuristics in the hearts domain. Implementation of *GSAs* was complete before investigation of the scheduling domain was begun. Nevertheless the exact same operators were applied to generate plausible heuristics in the scheduling domain as well.

GSA OPERATOR:	HEARTS:	SCHEDULING:
Equiprobable:Boolean (EP)	4	1
Equiprobable:Equality (EP)	5	1
Equiprobable:Numeric (EP)	6	3
Independence-of-Product:Binary (IN)	2	2
Independence-of-Product:Nary (IN)	2	1
Independence:Two-Conjuncts (IN)	3	2
Independence:Three-Conjuncts (IN)	3	1
Function-Invariance:Constant (FI)	6	3

Figure 3-19: *GSA* Operator Usage Comparison Across Domains

The generality of *POLLYANNA*'s approximation generator can be further analyzed by considering usage rates of the reformulation operators. The table in Figure 3-20 shows the numbers of times that individual reformulation operators were used in each of the two domains. Notice first that several reformulations are used multiple times in each domain. These include the key operators implementing *Probability/Expectation of Composite Functions (PC/EC)* along with the *Fold* and *UnFold* reformulations. These usage rates demonstrate that some of the reformulations exhibit considerable generality across domains.

The reformulation operators exhibit *less* generality across domains than the *GSA* operators. Notice that several reformulation operators were used in one domain only. Lack of generality among reformulations is not significant in most cases. For example, no effort was made to factor summations appearing in approximate scheduling theories. This explains the failure to use the operator for factoring summations, the operator for reversing the order of summations as well as the operators implementing commutative and associative properties of multiplication. Furthermore, some reformulations implement conceptually the same operations on functions with different numbers of arguments. When these differences among operators are ignored, the reformulations are seen to have more generality across domains.

REFORMULATION OPERATOR:	HEARTS:	SCHEDULING:
Expectation-of-Sum:Binary	2	0
Expectation-of-Sum:Nary	2	1
Composed-Exp: One Arg	2	1
Composed-Exp: Three Args	2	1
Composed-Prob: One Arg	1	1
Composed-Prob: Two Args	1	0
Factor-Summation	9	0
Reverse-Summation-Order	2	0
Commute-Product	4	0
Associate-Product	9	1
Probability-If-Known:Equality	5	2
Expectation-If-Known	0	1
Prob-Boolean-to-Prob-Equality	1	0
Member of Set	1	0
Prob of Random Member	1	0
Prob of Unique Member	0	2
Fold Function Definition: Atomic Args	112	36
Fold Function Definition: Selected Args	4	2
Fold Existing Function Definition	17	5
Unfold Function Definition	32	9
Unfold Lambda Definition	3	2

Figure 3-20: Reformulation Operator Usage Comparison Across Domains

A potentially significant problem is presented by a few of the reformulations that were used in one domain only. The most troubling cases involve reformulations specially designed to manipulate probabilities involving second order functions. These special reformulations were written to handle functions that appeared in one domain theory but not the other. Were *POLLYANNA* applied to a third domain, these results suggest that even more reformulations would have to be developed. As the system is applied to more and more domains, the reformulation operator set would hopefully stabilize and the need for new operators would diminish. This remains to be demonstrated by future research.

3.6. The Lessons of Analytic Learning

3.6.1. Limitations and Implementation Difficulties

POLLYANNA has successfully generated heuristics that perform only on the level of a human novice. Efficient heuristics with expert level performance have not been generated. The reasons for such limits will be explained in the following sections. The diagnoses fall into three main groups. Some of the limitations would apparently be overcome by adding new *GSA* or reformulation operators the ones currently implemented. Others would probably be remedied by using domain theories that are encoded differently, but still within the *PT-FORMALISM*. Still other difficulties could probably not be overcome without using a representation providing a richer set of constructs than are available in the *PT-FORMALISM*.

The approximation generator is quite capable of producing heuristics that are equivalent or superior in accuracy to expert level heuristics. This follows as a trivial consequence of the fact that the exact initial domain theory itself appears in the space. Even ignoring the initial theory, the approximation generator produces a series of theories that gradually converge on the exact one, as the random variable tree is expanded to greater and greater depth. The problem lies not in the accuracy of such theories. A successful system must generate expert heuristics that are efficient as well. All the expert level theories that can currently be generated are grossly inefficient. The efficiency of expert level heuristics appears to be limited by two main factors. Considerable inefficiency results from all the nested summations introduced by the approximation generator. Much greater inefficiency results from efficiency cliffs that appear in the problem space. These factors will be discussed below.

3.6.1.1. Inefficient Nested Summations

Many approximate theories generated by *POLLYANNA* are highly inefficient. The efficiency problems are partly attributable to deeply nested summations introduced by the operators implementing *Probability/Expectation of Composite Functions (EC/PC)*. The complexity analysis in Section 2.8.2 demonstrated that the approximate theories always have polynomial time complexity $O(R^N)$, where R is the range size of a typical random variable, and N is the depth of the most deeply nested summation. This represents a considerable improvement for NP-Complete problems like job scheduling. Nevertheless, even polynomial time theories can be rather slow. For example, some approximate hearts theories have doubly nested summations over all cards in the deck. Some approximate scheduling theories have tripley nested summations over all time slots.

Most of the inefficiency can be removed by suitably reformulating the approximate theories. Consider the semantically equivalent versions presented in Figures 3-4, 3-12 and 3-13. Were these versions implemented as written, they would be much more efficient than *POLLYANNA*'s approximate theories. The simpler, equivalent versions were derived by applying truth preserving reformulations by hand to the approximation generator output.

At least some of these reformulations could in principle be implemented in *POLLYANNA*'s approximation generator. For example, many simplifications occur in summations with an index i taken over expressions proportional to the Kroneker delta function δ_{ij}.[34] These summations can be removed since at most a single term in which $i=j$ can be non-zero. Were these and other reformulations implemented in the approximation generator, the resulting theories would be considerably more efficient.

Even greater efficiency would result from reformulations that reason about monotonicity properties of evaluation functions. Many approximate theories generated by *POLLYANNA* are equivalent to simple *IF-THEN* rules, as shown in Section 3.3.1.2. These rules can often be derived by showing functions to be monotonic in certain variables. When monotonic functions are used to evaluate choices, monotonicity implies simple minimization criteria. Reasoning about monotonicity would be more difficult to implement; however, it appears possible in principle.

3.6.1.2. Efficiency Cliffs

Efficiency cliffs arise when the approximation generator encounters random variables with extremely large ranges. Probabilities or expectations of such variables are computed by summing over their entire ranges. In the hearts domain, the hand of an opponent and the initial deal itself are both variables with extremely large ranges. Summations over these ranges can be generated if the random variable tree in Figure 2-10 is unfolded to sufficient depth. The resulting approximate theory operates by considering all possible hands or all possible deals one by one. When such a variable is encountered, efficiency falls off a cliff.

POLLYANNA might eliminate efficiency cliffs using an approach based on the *GSA Argument Abstraction (AA)*. (See Section 2.5.6). Instead of considering every possible hand or deal, abstraction would be used to divide these random variables into equivalence classes. Probabilities or expectation values would be computed by summing over all classes. Human hearts players appear to use abstraction in a similar manner. For example, they often speak of considering the possible splits, i.e., the number of cards in each suit held by each opponent, rather then each specific hand that might be held by an opponent. Splits are an abstraction of hands, since many different hands correspond to a single split. In order for *POLLYANNA* to generate such abstractions, random variables must be represented in a suitable way. The information to be ignored by abstraction (the specific cards) should be separated from the information to be preserved (the number of cards in each suit). Attempts to use *AA* for this purpose in *POLLYANNA* would undoubtably hinge on the representation of domain theories.

---

[34]The Kroneker delta function δ_{ij} is defined to be one if $i=j$ and zero otherwise.

Efficiency cliffs might also be prevented by using a different representation of the hearts domain theory. The alternative representation would avoid using random variables with vast ranges. For example, the variables representing splits might be built into the hearts theory from the beginning. Assuming such a representation exists, this approach would place special demands on humans who encode the initial domain theory. They would have to design domain theories with an eye toward the needs of *POLLYANNA*'s approximation generator. (See Section 3.6.2).

3.6.1.3. Variables with Unknown or Infinite Ranges

POLLYANNA's approximation generator requires a description of the range of each random variable appearing in the initial domain theory. This information must be supplied by humans. In some cases, humans may not be able to specify a finite range for a variable appearing in the theory. This problem can occur when the range is either infinite, or finite but unknown. For example, a variable might defined by an arithmetic function *(+,-,*,/)* or a probabilistic function *(Prob[], Exp[])* which has an infinite range. Even when only a finite set of values can arise in practice, the specific set may be difficult to determine.

As an extreme example, consider that *Exp-Game-Score* is itself a random variable appearing in the hearts domain theory. The value of *Exp-Game-Score* is computed by one's opponents as they choose their cards. Their computations depend on the way the cards were dealt. The range of this variable must be considered by approximate theories that consider optimal, rather than random behavior by opponents. This variable can have only a finite number of different values, since hearts is a finite game. The exact set of values is nevertheless rather difficult to determine. It is also likely to be rather large.

3.6.1.4. Second Order Mapping Functions

Second order mapping functions present special difficulties for *POLLYANNA*'s approximation generator. For example, consider the function *Minimize(S,F)*, which finds a member *x* of set *S* that has a minimal value of *F(x)*. In order to apply composite *Probability/Expectation of Composite Functions (EC/PC)* to an expression of the form *Prob[λ(e)v=Minimize(S,F)]*, the ranges of *S* and *F* must be known. The set *S* will typically have a large range, e.g., the number of possible hands in hearts. The function *F* may itself be a random variable, the range of which is a set of functions that depend on the event space variable *e*, e.g., the evaluation function used by an opponent. This set is likely to be both large and unknown or difficult to represent. Similar problems are presented by functions such as *For-All*, *Exists* and *Set-of* among others. These problems were handled by two methods in *POLLYANNA*. Domain theories were written to avoid using these second order functions whenever possible. In cases when such functions could not be avoided, special case reformulation rules were written to substitute for *Probability/Expectation of Composite Functions (EC/PC)*. (See Appendix G.)

3.6.1.5. Normalization of Probabilities

POLLYANNA's approximations often result in probabilities that are not normalized. This problem arises when *Equiprobable Random Variables (EP)* is used to set $Prob[\lambda(e)v{=}F(e)]$ equal to $1/|Range(F)|$ for some values of v, while an exact probability of zero or one is used for values of v that are known to be definite or impossible. As a result, the probabilities do not add up to one. As an example, suppose the variable $\lambda(e)F(e)$ is known to have value v_1, so that $Prob[\lambda(e)v_1{=}F(e)]$ is equal to one. For other values of v_i, the value of $Prob[\lambda(e)v_i{=}F(e)]$ is assumed to be $1/|Range(F)|$. The probabilities will therefore sum to a number greater than one. If the value v_1 were known to be impossible for $\lambda(e)F(e)$, then $Prob[\lambda(e)v_1{=}F(e)]$ would be zero, and the probabilities would sum to a number smaller than one. This accounts in part for the extremely small probabilities of winning the next trick that arise in some approximate hearts theories. Lack of normalization does not impact the monotonicity properties of individual decision criteria; however, it does impact the relative weights given to each when they are combined into a single evaluation function. Normalization could be maintained by revising *Probability/Expectation of Composite Functions (EC/PC)* to compute and insert normalization constants. Empirical testing would be useful for determining whether improvements in accuracy would be sufficient to offset the computational expense of normalization.

3.6.1.6. Learning Special Case Approximations

POLLYANNA generates heuristics that are different in flavor from those used by humans. A book on hearts presents heuristics in the form of rules that apply to special situations [Andrews 83]. Approximate theories generated by *POLLYANNA* have the form of evaluation functions that apply to all situations. Semantic analysis shows that the evaluation functions generate different behavior in different situations. In both the hearts and scheduling domains, some approximate theories were shown equivalent to special case rules, however, special cases not appear explicitly in the approximate theories generated by *POLLYANNA*.

Special case approximations would result from adding some additional reformulations to those already used in *POLLYANNA*. One approach would use reformulations that unwind summations (Σ) or products (Π) over sets. For example, in hearts a summation over the set of all tricks could be replaced by a sum of thirteen explicit terms. A product over all opponents could be replaced by three explicit factors. After creating an extensional sum or product, distinct approximations could be applied to each term or factor. Different methods could be used to analyze each trick in the game, or each of the opponents to the left, right and across the table. For example, this might lead to different strategies to use during the early, middle or later parts of the hearts game.

The failure to generate special case rules should ultimately be blamed on limitations of the *PT-FORMALISM* itself. Theories represented as algebraic expressions generally lack the expressive power needed to describe case situations. Although the *PT-FORMALISM* contains constructions for conditional evaluation of expressions, these conditionals seem to lack the necessary descriptive powers. Generation of true special case heuristics would probably require extending the *PT-FORMALISM* with planning constructs. It might also require applying the *POLLYANNA* methodology to a different formalism altogether.

3.6.1.7. Learning Weights for Decision Criteria

POLLYANNA has only rudimentary capabilities for controlling the weights associated with individual decision criteria in evaluation functions. The system is quite successful at generating sub-expressions representing the individual criteria. It also generates numerical constants that determine the relative weights associated with the decision criteria implemented by each sub-expression; however, the constants often seem to be quite arbitrary. They also tend to be generated at only a few discrete intervals. The system has no method of continuously varying the relative weights of the decision criteria. These observations suggest combining *POLLYANNA* with techniques for learning the weights of individual terms in evaluation functions. *POLLYANNA* would generate the terms. Parameter learning methods would find appropriate weights.

3.6.2. Strategies for Representation of Domain Theories

The success of *POLLYANNA* depends ultimately on the representation of domain theories. The system is actually quite brittle. Representation details have a dramatic impact on the kinds of approximations that are produced by the system. *POLLYANNA* is similar in this respect to most or all other learning programs. The representation of domain knowledge has long been recognized as a source of bias in the learning process [Mitchell 80; Utgoff 86].

Representation brittleness can be attacked in several ways. One approach would attempt to develop robust learning techniques that are not so sensitive to representation details. Another approach would attempt to automatically reformulate the domain theory from an inappropriate representation into a suitable one. A third approach would develop a set of guidelines to be used in constructing domain theories. These guidelines would help knowledge engineers to select suitable representations of domain theories.

A preliminary set of representation guidelines is presented below. These are not intended to be general theory of good and bad representations. They apply only to the problem of representing theories for use in systems that learn approximations from generic simplifying assumptions. Some apply specifically for writing theories within the *PT-FORMALISM* to be approximated using *PT-GSAs*. In some cases, the guidelines are directly contrary to criteria that would be appropriate in other contexts.

3.6.2.1. Transparency and Efficiency of Representations

POLLYANNA places a premium on transparent, declarative representations. It requires theories to be written in the *PT-FORMALISM*. Functions must be defined using purely declarative λ-expressions, among other things. Declarative representations are necessary in order that certain key reformulations be available, i.e., *Probability/Expectation of Composite Functions (EC/PC)*. Declarative representations may be far less efficient than alternative, procedural representations. For example, the function *Minimize(S,F)* was written in a way that calls the evaluation function *F* twice for each member of the set *S*. (See Appendix C). A more efficient version would avoid repeated evaluation; however, it would not be so transparent and the necessary reformulations would most likely not apply. Transparency is more important than efficiency, in the context of *POLLYANNA*. This ironic situation obtains because the intractable theories are designed to be reformulated and approximated. They are not intended to be directly executed.

3.6.2.2. Incomplete and Redundant Representations

Domain theories used in *POLLYANNA* do not have to be complete. In particular, a domain theory need not provide definitions for all the random variables appearing in the theory. Missing definitions simply limit the set of reformulations and approximations that are available. If the approximation generator does not expand the random variable tree deep enough, undefined variables will not be referenced by the resulting approximate theories. Even when the tree is expanded down to the level of an undefined variable, a usable theory may result. The variable in question must either be known in the current problem state, or else it must be the subject of an assumption of *Equiprobable Random Variables (EP)*. For example, consider the *Choice* function that defines how opponents pick their cards. This function was never referenced in the derivations of the approximate theories described above. If this function were inadvertently left out, or entirely unknown, all the same approximate theories could have been generated.

Domain theories can provide redundant definitions of random variables. Consider the two variables *Win* and *Winner* discussed in Section 2.5.2. The boolean variable *Win(p,...)* is true when player *p* wins a trick. The symbolic variable *Winner(t,...)* is equal to the winner of a trick. These alternative representations were seen to produce different results when used as the subjects of *EP* assumptions. Were both included in the initial domain theory, the range of possible approximations would be wider than if the theory included either one alone. Knowledge engineers are therefore free to provide multiple definitions of functions, each of which carves up the event space in a different way. The alternative definitions need not even be mutually consistent. *POLLYANNA*'s theory space generation module arranges that only one definition is used at a time. (See Chapter 4.)

3.6.2.3. Intensional and Extensional Representations

Intensional representations were used heavily in representing the hearts and scheduling domain theories. Intensional representations are expressed in terms of operations applied to entire sets. Examples include the second order "reduction" operations *Sum* (Σ) and *Product* (Π), which apply binary associative operations after mapping a function over a set. Alternative extensional representations would have involved writing syntactically distinct expressions for each member of the set, e.g., an expression for each term in a summation (Σ) or each factor in a product (Π). Intensional forms were used to guarantee that identical approximations are applied to each term in a summation or each factor in a product. The space of approximate theories was thereby greatly reduced, in comparison to the one that would result from separately approximating each sub-expression. In the absence of a good reason for treating sub-expressions differently, domain theories should use intensional representations as much as possible.

Extensional representations are useful in some cases. The hearts theory in Appendix C was explicitly designed to facilitate separate treatment for terms corresponding to the current and next tricks of the game. It might also be useful to generate separate approximations of expected *Trick-Value* for each of the thirteen tricks as described above. Consider what would happen if *N* different versions of *Exp-Job-Value* were used to analyze each of the 1st, 2nd,...,Nth jobs in a scheduling problem. The effort would be entirely wasted if jobs were listed in random order. Were the jobs consistently ordered in some meaningful way, e.g., the most important jobs listed first, the effort might produce useful results.

A knowledge representation principle can be formulated by considering the notion of critical and non-critical subproblems. *POLLYANNA*'s approximate theories tend to focus on particular subproblems, sub-expressions or decision criteria that are involved in a computation. Each theory implicitly takes a position regarding which subproblems are critical and worthy of attention, and which are non-critical and subject to approximation. In order to divide subproblems into groups of differing importance, the system requires some means of describing sets of subproblems. For this purpose *POLLYANNA* relies on the initial theory representation. Expressions grouped together using intensional forms are treated identically. Expressions written separately using extensional forms are potentially subject to different approximations. *POLLYANNA* will be most effective when subproblems of equal importance are grouped together under intensional forms. Subproblems of differing importance should be separated using extensional forms. The intensional / extensional distinction should thus line up with the *natural kinds* of the domain.

3.7. Summary of Analytic Generation Results

- **Semantic Equivalence of Heuristics and Approximate Theories:** Semantic analysis provides a means of analyzing the quality of approximations generated by *POLLYANNA*. The system generates many approximate hearts and scheduling theories that are semantically equivalent to intuitively plausible heuristics. Some approximate theories are easily paraphrased by analyzing the monotonicity properties of approximate function definitions. Other approximate theories are too complex to paraphrase because they implement a numerical balance among multiple decision criteria. Semantic analysis shows that some generated heuristics are intuitively plausible. Some also generate performance on the level of a human novice. All generated heuristics are nevertheless relatively naive in comparison to expert human performance. Even the naive results are significant considering they are generated from an initially intractable theory using domain independent techniques.

- **Analytically Generated Hearts and Scheduling Heuristics:** Generated hearts heuristics include the *Dump High Rank Rule*, and *Dump High Point Value Rule*, along with extensions to these rules. Nearly all members in a set of goal heuristics are generated, along with generalizations extending the goal heuristics to more complex hearts games. Generated scheduling heuristics *Soonest Scheduling Deadline First*, *Enable Many Jobs* and *Enable a Well Prepared Job*. Some generated heuristics are merely "reconstructions" of heuristics known to the system designer prior to implementation. Others are "rediscoveries" of heuristics not known to the designer, but previously known to some humans. Still others are "discoveries" of heuristics that were probably never previously formulated by anyone.

- **Generality of the Approximation Generator:** Generality and domain independence are analyzed by examining statistics summarizing the usage of individual generic simplifying assumptions and truth-preserving reformulations. Operator usage comparisons between the hearts and scheduling domains clearly demonstrate the domain independent nature of generic simplifying assumptions. Comparisons also demonstrate a degree of domain independence for the truth-preserving reformulations; however, the reformulations appear to be less general than the generic simplifying assumptions.

- **The Lessons of Analytic Learning:** *POLLYANNA* has difficulty in generating efficient versions of expert level heuristics. Expert results are impeded by efficiency cliffs that drastically raise the costs of executing approximate theories. One potential remedy would develop new generic simplifying assumptions that implement abstraction techniques. Another approach would develop new reformulations that facilitate development of context sensitive heuristics. A third approach would use a richer knowledge representation formalism.

- **Knowledge Representation Strategies:** Guidelines for representation of domain theories summarize the experience of implementing the hearts and scheduling domains. *POLLYANNA* places a greater premium on transparent, declarative representations, than on computational efficiency of the initial theory. Some types of incomplete and redundant representations can be handled by the approximation generator. The generation process is facilitated by a judicious choice between intensional and extensional representations.

Chapter 4

Empirical Testing of Heuristics

4.1. Introduction

The role of empirical learning is best understood by viewing *POLLYANNA* in terms of a generate and test architecture. Analytic methods are used to generate candidate theories in the *approximation generation (AG)* and *theory space generation (TSG)* phases. These methods are considered "analytic" because they operate in the absence of training examples. They generate candidate theories with widely varying levels of accuracy and efficiency. Desirable candidates are generated along with many undesirable ones. Empirical methods are therefore used to test the accuracy and efficiency of candidates during the *theory space search (TSS)* phase. The search process is considered "empirical" because it uses training examples to evaluate candidate theories. The search is directed at finding theories meeting explicit accuracy and efficiency goals. (See Figure 1-6).

Empirical testing is a potentially costly process. Exhaustive testing of all candidates is prohibitively expensive in many application contexts. Techniques for intelligent control and organization of search are therefore important. Theory space generation (*TSG*) and theory space search (*TSS*) are both designed to limit the costs of empirical testing. *TSG* attempts to generate theory spaces monotonic in the efficiency of candidate theories. Monotonicity facilitates search for theories meeting explicit efficiency goals. *TSS* uses accuracy and efficiency measurements to control the search process.

4.2. Overview

Learning is guided by explicit accuracy and efficiency goals in *POLLYANNA*. A variety of different goal types is supported. Various different types of goals may be appropriate depending on the context in which learning occurs. These learning goals have a significant impact on the design of both the *TSS* and *TSG* modules. They impact the organization of search spaces produced by *TSG*. They also influence the search control techniques used in *TSS*. Accuracy and efficiency goals are therefore discussed first, in Section 4.3.

Theory space generation (*TSG*) is discussed next, in Section 4.4. *TSG* is viewed as an analytic reasoning process in Section 4.4.2. Analytic methods are used to reason about efficiency, equivalence, completeness and consistency of candidate approximate theories. The value of a monotonic theory space is discussed in Section 4.4.3. Techniques for generating monotonic theory spaces are discussed in Section 4.4.4. Theorems describing the efficiency impact of generic simplifying assumptions are presented and proved, in Section 4.4.6. Analytic methods of reasoning about equivalence of candidate theories are discussed in Section 4.4.9 The actual *TSG* algorithm is presented in Section 4.4.11. The sizes of various candidate theory spaces are analyzed in Section 4.4.13.

Theory space search (*TSS*) is discussed next, in Section 4.5. Several different search algorithms are presented, in Section 4.5.1. Each algorithm handles a different type of learning goal. Each uses a search control strategy that is appropriate the particular goal type. Methods of representing training examples in the hearts domain are discussed in Section 4.5.2.1. Methods used to generate hearts examples are discussed in Section 4.5.2.2. Techniques for measuring the accuracy and efficiency of candidate theories are discussed in Sections 4.5.2.4 and 4.5.2.5. Results of empirical learning are discussed in Chapter 5.

4.3. Learning Goals and Contextual Knowledge

Computational theories can be evaluated in a variety of different ways. The two criteria of accuracy and efficiency were explicitly mentioned in the intractable theory problem definition. (See Figure 1-3.) Additional evaluation criteria have been proposed by other investigators. These include *scope* and *operationality requirements* [Mostow and Fawcett 87], as well as *generality*, *robustness*, *recoverability* and *obviousness* [Segre 88]. No single criterion is suitable for all situations [Keller 88]. Various different measures of performance may be appropriate depending on the context in which a performance program is intended to be used. This issue was earlier discussed using the two examples of chess and medical diagnosis. (See Section 1.2.8.) These examples illustrated that accuracy and efficiency could each be given priority in some situations. The appropriate *balance* between accuracy and efficiency will depend on the context in which a computational theory is used.

In the absence of evaluation criteria that apply to all contexts, learning programs should be *sensitive* to the specific context in which the performance program will be used [Keller 87]. They should modify their behavior appropriately when they are given *contextual knowledge* describing the context in which learning takes place. Contextual knowledge is supplied to *POLLYANNA* in the form of accuracy and efficiency goals. These goals are supplied to the theory space search (*TSS*) module. *POLLYANNA* generates various different results depending on these learning goal parameters. Evidence for this fact will be presented in Chapter 5. The results of learning in *POLLYANNA* will thus be shown to depend on contextual knowledge.

4.3.1. Accuracy and Efficiency Goals for *POLLYANNA*

The learning goals handled by *POLLYANNA* are listed in Figure 4-1. All of these goals are instances of a general goal schema, also described in Figure 4-1. The schema requires finding a theory that is optimal on measures $M_1(t),...,M_n(t)$ subject to constraints $C_1(t),...,C_n(t)$. Possible optimization measures include *Cost(t)* and/or *Error-Rate(t)*.[35] Possible constraints include absolute bounds on *Cost(t)* and/or *Error-Rate(t)*. Depending on which optimization measures and constraints are included, a total of sixteen different goal types can be formulated. Of the sixteen possible combinations, only eleven represent meaningful goals. A theory space search algorithm is associated with each goal in Figure 4-1. These algorithms will be discussed in Section 4.5.

General Goal Schema:

Minimize Measures $M_1(t),...,M_n(t)$ under Constraints $C_1(t),...,C_n(t)$:

Measures:	$M_1(t)$: Error-Rate(t)	
	$M_2(t)$: Cost(t)	
Constraints:	$C_1(t)$: Error-Rate(t) $\leq K_1$.	
	$C_2(t)$: Cost(t) $\leq K_2$	

Algorithm:	**Goal Type:**
TSS-C	A. Error-Rate(t) $\leq K_1$
TSS-C	B. Cost(t) $\leq K_2$
TSS-C	C. Error-Rate(t) $\leq K_1$ and Cost(t) $\leq K_2$
IT	D. Minimize Error-Rate(t)
TSS-E	E. Minimize Error-Rate(t) subject to Cost(t) $\leq K_2$
Root	F. Minimize Cost(t)
TSS-G	G. Minimize Cost(t) subject to Error-Rate(t) $\leq K_1$
TSS-K	H. Pareto Optimize Cost(t) and Error-Rate(t).
TSS-K	I. Pareto Optimize Cost(t) and Error-Rate(t) under constraint Error-Rate(t) $\leq K_1$.
TSS-K	J. Pareto Optimize Cost(t) and Error-Rate(t) under constraint Cost(t) $\leq K_2$.
TSS-K	K. Pareto Optimize Cost(t) and Error-Rate(t) under constraints Error-Rate(t) $\leq K_1$ and Cost(t) $\leq K_2$

Definitions:

```
TSS-C = Algorithm for Goals A, B, C.
TSS-E = Algorithm for Goal E.
TSS-G = Algorithm for Goal G.
TSS-K = Algorithm for Goals H, I, J, K.
Root  = Take the theory space root.
IT    = Take the initial intractable theory.
```

Figure 4-1: Learning Goals and Theory Space Search Algorithms

[35]The terms "Cost" and "Error-Rate" refer to the opposites of efficiency and accuracy.

Several of *POLLYANNA*'s goals require optimizing a combination of two measures. These goals are interpreted in terms of *Pareto optimization*, i.e., finding theories that are Pareto optimal with respect to a collection of measures. A theory t is said to be "Pareto optimal" over set S on measures M_1(t),... $M_n(t)$ if there exists no member of S that strictly dominates theory t, i.e., no theory t' is equal to or better than t on each measure, and is better than t on at least one measure. Pareto optimization reduces to ordinary optimization when only one measure is used. The goals of types *I, J,* and *K* require some interpretation. They ask for all theories that meet two conditions: They must satisfy constraints $C_1(t),...,C_n(t)$. They must also be Pareto optimal over the set of theories meeting these constraints. The goals do not ask for *all* Pareto optimal theories in the entire space. Nor do they ask for theories that are Pareto optimal over the *entire* theory space generated by the analytic learning components of *POLLYANNA*.

POLLYANNA is capable of absolutely achieving certain types of learning goals. Others can only be satisfied in relative terms. Learning goals *D,E,F,G,H,I,J* and *K* involve optimization of either accuracy or efficiency or both. Such goals are satisfied only relative to the set of candidate theories generated in the analytic phase of learning. The candidates depend on the *GSA* operators, reformulation operators and control strategy used in the approximation generator. Different approximation or reformulation methods might produce superior results. In the absence of proofs setting bounds on the efficiency and accuracy of theories, no guarantees of absolute optimality are possible. A more favorable situation obtains regarding learning goals *A, B* and *C*. These involve accuracy or efficiency thresholds, but no optimization. Such thresholds can be directly tested against training examples. If the system can find a theory meeting these thresholds, the learning goals are achieved in the absolute sense. When the system fails to find a such a goal theory, the failure should be interpreted relative to the available candidate theories. Theories meeting the threshold may yet exist.

4.3.2. Defining "Accuracy" and "Efficiency"

The "accuracy" and "efficiency" of computational theories can each be defined in a variety of ways. The accuracy of an evaluation function $f(x)$ might be defined as the mean square difference between the computed value of $f(x)$ and the true value An alternative definition might involve only errors in the rank order of values of $f(x)$. A third definition might count only values of x falsely considered optimal or falsely considered sub-optimal. Accuracy of concept descriptions can be defined in terms of false positives, false negatives or both. The "efficiency" of computational theories can also be defined in many ways. Time and space are two separate dimensions of efficiency. Each of these can be defined in absolute terms, e.g., CPU millenia or tape warehouses, or in terms of asymptotic complexity, e.g., $O(R^N)$. Both time and space efficiency can be defined in worst case, best case or average case versions. Average, best and worst case accuracy and efficiency would be defined relative to the distribution of problems to be solved in the performance phase. (See Section 1.2.5.)

Appropriate definitions of accuracy and efficiency depend ultimately on considerations that are external to the learning system. Just as the relative importance of accuracy and efficiency must be determined by context, so too are the specific definitions of each. The goal table in Figure 4-1 could in principle be expanded to incorporate such context sensitivity. In addition to thresholds and minimization criteria, specific methods for measuring accuracy and efficiency could be supplied as parameters to a learning system. *POLLYANNA* does not currently provide such flexibility. Specific definitions of accuracy and efficiency along with the techniques used in *POLLYANNA* to measure each will be discussed in Sections 4.4.5, 4.5.2.4 and 4.5.2.5.

4.4. Theory Space Generation (*TSG*)

4.4.1. The Role of *TSG* in *POLLYANNA*

Theory space generation (*TSG*) can be understood in terms of its position in the *POLLYANNA* architecture. The *TSG* module occupies a position in between the approximation generator (*AG*) and theory space search (*TSS*) modules. (See Figure 1-6). *TSG* gets its input from the final state of the approximation generator. This data is organized as a table mapping each function name to one or more λ-expressions representing possible versions of the function. (See Figure 2-6.) *TSG* systematically combines function versions into complete candidate theories. Complete theories correspond to tuples drawn from the Cartesian product of version sets. *TSG* also serves to organize candidate theories into a search space. The space is defined by a function *Successors(t)* that takes a candidate theory *t* as argument, and returns a set of other candidates representing refinements of theory *t*. The exact sense in which one theory is said to be a "refinement" of another will be discussed below. The function *Successors(t)* is used by *POLLYANNA*'s theory space search *TSS* module to conduct empirical testing of candidate theories.

4.4.2. *TSG* as an Analytic Reasoning Process

Theory space generation (*TSG*) is an *analytic* reasoning process. It operates entirely in the absence of training examples. (See Figure 1-6.) Several types of analytic reasoning are performed by the *TSG* module. Both facilitate the subsequent theory space search (*TSS*) process. One type concerns the structure of candidate theory spaces. *POLLYANNA*'s theory space search *TSS* module exploits *monotonic* organization of the theory space. Portions of the space can be pruned if the space is monotonic in the efficiency of candidate theories. Monotonicity can be achieved by reasoning about the relative efficiency of theories. A second type concerns redundancy among candidate theories. A great many equivalent theories are contained in the full Cartesian product of version sets. *TSG* attempts to avoid generating more than one copy of equivalent theories. Redundancy can thus be limited by reasoning about equivalence among theories. Both types of analytic reasoning facilitate the subsequent theory space search (*TSS*) process in *POLLYANNA*.

Analytic reasoning about efficiency or equivalence is quite difficult. *POLLYANNA* therefore relies on highly specialized methods for making such analytic comparisons. These methods rely in the manner in which theories are derived by the approximation generator and they apply only to theories so derived. The analytic techniques are only partially successful. *TSG* is able to compare efficiency levels for some pairs of theories, but not for all pairs. *TSG* succeeds in detecting only some special cases of equivalent theories. These limitations are not fatal. They may merely result in an empirical testing process that is more costly than otherwise.

4.4.3. The Value of Monotonic Theory Spaces

Two definitions of theory space monotonicity are shown in Figure 4-2. The first definition describes spaces that are monotonically increasing in some numerical measure *M(t)* of a property of candidate theories. The second definition describes spaces that are monotonically decreasing in the truth value of some boolean constraint *C(t)* defined on candidate theories. Both definitions are written in terms of the binary relation $Refinement(t_1,t_2)$ among theories. This relation is implicitly defined by the *Successors* function, i.e., $Refinement(t_1,t_2)$ if and only if $t_2 \in Successors(t_1)$. In each case, the value of *M(t)* or *C(t)* changes monotonically along paths through the candidate theory space.

- Monotonically Increasing in Measure *M(t)*:

$$(\forall\ t_1,t_2)\ \mathtt{Refinement}(t_1,t_2) \Rightarrow [M(t_1) \leq M(t_2)]$$

- Monotonically Decreasing in Constraint *C(t)*:

$$(\forall\ t_1,t_2)\ \mathtt{Refinement}(t_1,t_2) \Rightarrow [(C(t_1) \Leftarrow (C(t_2))]$$

Figure 4-2: Types of Monotonicity Properties

The value of monotonicity can be understood in terms of the learning goals shown in Figure 4-1. Learning goals are specified in terms of minimization measures $M_i(t)$ and/or constraints $C_i(t)$. Consider goal types *E* and *G*, both of which have the form: "Minimize measure $M_i(t)$ subject to constraint $C_i(t)$.". A search for theories satisfying this goal is facilitated when the space is monotonic in either $M_i(t)$ or $C_i(t)$. Suppose the theory space is monotonically decreasing in the truth value of constraint $C_i(t)$. If a search algorithm encounters a theory *t* that fails to meet constraint $C_i(t)$, all of the successors of theory *t* can be pruned. Monotonicity guarantees that the successors of *t* will also fail to meet constraint $C_i(t)$. (See Algorithm *TSS-E* in Section 4.5.) Now suppose the space is monotonically increasing in the value of measure $M_i(t)$. An efficient best first search algorithm results from expanding nodes in order of increasing values of $M_i(t)$ and using $C_i(t)$ as the termination condition. (See Algorithm *TSS-G* in Section 4.5.)

POLLYANNA attempts to generate theory spaces that are monotonic in the efficiency of candidate approximate theories. In such a monotonic theory space, the most efficient theory is found at the root of the space. Efficiency falls monotonically along paths in the space. All the learning goals and corresponding search algorithms in Figure 4-1 are facilitated using a search space that is monotonic in efficiency. (See Section 4.5.) A theory space monotonic in accuracy would also facilitate the goals and algorithms listed in Figure 4-1; however, *POLLYANNA* does not generate such search spaces. Additional research would be needed to develop analytic methods of reasoning about the relative accuracy of computational theories.

4.4.4. Generation of a Monotonic Theory Space

Analytic comparisons of efficiency are required in order to generate a theory space that is monotonic in efficiency. *TSG* analyzes efficiency by comparing the versions of individual functions that appear in the final state of the approximation generator. Comparisons of versions are guided by *efficiency impact rules*. Each rule describes the impact of a single generic simplifying assumption on the efficiency of approximate theories. These rules are used to construct a relation *Primitive-Refinement*(f,f_i,f_j) over the versions of each function. The primitive refinement relation defines a distinct partial order for each function *f* appearing in the final state of the approximation generator. *Primitive-Refinement*(f,f_i,f_j) implies that version f_i is at least as efficient as f_j, and possibly more efficient as well. Comparisons of complete theories follow from comparing the versions of individual functions. Theory t_1 is inferred to be more efficient than theory t_2 provided the *Primitive-Refinement* relation holds between all pairs of corresponding versions in t_1 and t_2. The semantics of *Primitive-Refinement* will be defined more precisely below.

The *Primitive-Refinement* relation affords a simple specification of a function *Successors(t)* that generates a monotonic theory space. Starting with an initial theory t_1, a successor of t_1 is constructed in the following way: First pick a function *f* that is defined by version f_1 in theory t_1. Find some other version f_2 such that *Primitive-Refinement*(f,f_1,f_2). Then construct theory t_2 by replacing version f_1 with version f_2. Successor theories are thus constructed by replacing versions with their primitive refinements. The *Successor* function and corresponding *Refinement* relation are defined in Figure 4-3. These definitions represent a specification of the *TSG* algorithm to be presented in Section 4.4.11.

One additional constraint should be added to the specification in Figure 4-3. The specification alone does not guarantee the strongest possible ordering of candidate theories. The new constraint arranges that theory *Successors(t)* returns only *minimal* refinements of theory *t*. It requires that a version f_1 be replaced by another version f_2 that is a minimal primitive refinement, i.e., there is no other version f_i such that *Primitive-Refinement*(f,f_1,f_i) and *Primitive-Refinement*(f,f_i,f_2). This constraint can be implemented by computing the

- *Successor*: The successors of theory t include all theories t' such that *Refinement(t,t')*:

$$(\forall\ t)\ \mathbf{Successors(t) = \{t'|Refinement(t,t')\}}$$

- *Refinement*: A theory t_1 is refined into theory t_2 by replacing the existing version of some function f with a primitive refinement:

$$\begin{aligned}&(\forall\ t_1,t_2)\\ &\mathrm{Refinement}(t_1,t_2) \Leftrightarrow (\exists\ f,f_1,f_2)\ \mathrm{Variant}(t_1,t_2,f,f_1,f_2)\\ &\qquad \wedge\ \mathrm{Primitive\text{-}Refinement}(f,f_1,f_2)\end{aligned}$$

- *Variant*: Two theories are variants of each other if and only if they differ in the definition of exactly one function.

$$\begin{aligned}&(\forall\ t_1,t_2,f,f_1,f_2)\\ &\mathrm{Variant}(t_1,t_2,f,f_1,f_2) \Leftrightarrow t_1(f)=f_1 \wedge t_2(f)=f_2\\ &\qquad \wedge\ (\forall\ n\neq f)\ t_1(n)=t_2(n)\end{aligned}$$

t(f) = The version of function f used in theory t.

Figure 4-3: Specification of the Theory Space

reflexive transitive reduction of *Primitive-Refinement*, i.e., finding a minimal subset of the *Primitive-Refinement* relation with the same reflexive, transitive closure as the original relation.

A precise semantics for the *Primitive-Refinement* relation can now be formulated. A condition on the *Primitive-Refinement* relation is shown in Figure 4-4. The condition guarantees monotonicity of a theory space generated according to the specification in Figure 4-3. The condition can be summarized as follows: Suppose that two theories t_1 and t_2 differ only in the definition of a single function f. Suppose further that t_1 uses version f_1 and t_2 uses version f_2 such that *Primitive-Refinement*(f,f_1,f_2). Under these circumstances, the condition requires that $Cost(t_1) \leq Cost(t_2)$. The cost relationship must hold regardless of the other versions used in theories t_1 and t_2. Theory space monotonicity is a consequence of the *Primitive-Refinement* semantics in Figure 4-4 and the theory space specification in Figure 4-3. The consequent monotonicity condition is stated in Figure 4-5.[36]

$$\begin{aligned}&(\forall\ t_1,t_2,f,f_1,f_2)\ [\mathrm{Cost}(t_1) \leq \mathrm{Cost}(t_2)] \Leftarrow \mathrm{Variant}(t_1,t_2,f,f_1,f_2)\\ &\qquad \wedge\ \mathrm{Primitive\text{-}Refinement}(f,f_1,f_2)\end{aligned}$$

Figure 4-4: Semantics of *Primitive-Refinement*

$$(\forall\ t_1,t_2)\ [\mathrm{Cost}(t_1) \leq \mathrm{Cost}(t_2)] \Leftarrow \mathrm{Refinement}(t_1,t_2)$$

Figure 4-5: Theory Space Efficiency Monotonicity Condition

[36] The semantics prescribe *necessary* conditions, which must hold whenever a primitive refinement relation is asserted. They do not prescribe *sufficient* conditions for asserting the relation. The corresponding sufficient conditions would require that *Primitive-Refinement*(f,f_i,f_j) be inferred whenever f_i leads to more efficient theories than f_j. Such inferences are not possible since *TSG* cannot compare the efficiency of all pairs of function versions.

4.4.5. Defining Analytic "Efficiency"

A precise definition of computational efficiency is required in order to reason analytically about the efficiency of theories. Two possible approaches include *absolute* and *asymptotic* analysis of efficiency. Each has advantages as well as serious drawbacks. Asymptotic complexity analysis is useful for obtaining machine independent results; however, it makes only very gross comparisons between algorithms. Asymptotic comparisons are insensitive to adding or multiplying costs by arbitrary constants. They also require formulating families of problems with associated problem size parameters. Even when this is possible, there the asymptotic comparisons may not be relevant to the distribution appearing in a specific performance context. Absolute analysis presents the opposite advantages and disadvantages. Precise comparisons are possible; however, the results are highly dependent on specific machine architectures or software implementations. Furthermore analytic comparisons of absolute time or space would likely get bogged down in the details of machine and software implementation.

POLLYANNA is based on an intermediate approach. A notion of *abstract absolute cost* is used to capture the advantages of the two methods described above. This model presupposes a list of primitive operations and primitive data types. The time cost of a theory is defined in terms of the number of applications of each primitive operation. *Time* is conceptually a vector that maps operations to integers. For each operation *OP*, the integer *Time(OP)* equals the number of applications of the primitive operation *OP*. The space cost of a theory could be defined in several different ways. One approach defines space cost in terms of the number of instances of each primitive data type and the lifetimes of those instances. *Space* is conceptually a vector that maps data types to integers. For each data type *DT* the integer *Space(DT)* equals the sum of over all time intervals of the number of instances of the primitive data type *DT* that are currently alive and therefore occupying space.[37] *POLLYANNA* makes analytic comparisons of time efficiency only. Analytic reasoning about space efficiency would require additional research.

A degree of machine independence results from measuring cost in terms of primitive operations and primitive data types. The *Time* and *Space* vectors will be independent of the manner in which primitive operations and primitive data types are implemented on a specific machine. Some difficulties also result from this model. How are the costs of two theories compared when costs are vectors with multiple dimensions? A possible solution involves reasoning about uniform dominance. A theory t_1 is considered to have lower time cost than theory t_2 only if the value of *Time(OP)* is as low or lower, for all primitive operations *OP*, under theory t_1 than under theory t_2. Likewise, a theory t_1 is considered to have lower space cost than theory t_2 only if the value of *Space(OP)* is as low

[37]An alternative definition would involve only the maximum space required at any single point in time. The appropriate definition of space cost depends ultimately on the context in which a computational theory will be used.

or lower, for all primitive data types *DT*, under theory t_1 than under theory t_2. This condition of uniform dominance is summarized in Figure 4-6. A second difficulty arises if primitive operations take varying amounts real time depending on their parameters, or when primitive data types use up varying amounts of real space depending on their instance variables. The solution is to define lower level primitives; however, some machine independence may be sacrificed.

Comparing Operation Counts:

$$\mathbf{TCost}(t_1) \leq \mathbf{Cost}(t_2) \Leftarrow (\forall\ p \text{ in Operations})\ \mathbf{Time}(t_1,p) \leq \mathbf{Time}(t_2,p)$$

Comparing Data Object Counts:

$$\mathbf{SCost}(t_1) \leq \mathbf{Cost}(t_2) \Leftarrow (\forall\ t \text{ in Data-Types})\ \mathbf{Space}(t_1,t) \leq \mathbf{Space}(t_2,t)$$

Figure 4-6: Uniform Dominance Cost Comparisons

4.4.6. Efficiency Impact Theorems

Efficiency impact theorems serve to justify *POLLYANNA*'s methods for analytically comparing costs of computational theories. Such comparisons are needed in order to construct a theory space monotonic in efficiency. The required theorems can be determined by examining the semantics of the primitive refinement relation shown in Figure 4-4. Each theorem assumes the existence of two theories t_1 and t_2 that differ only in the definition of a single function, i.e., *Variant*(t_1,t_2,f,f_1,f_2). The function versions f_1 and f_2 are assumed to differ by the application of a *GSA* operator. Under these conditions, the impact theorems assert that theory t_1 has equal or lower cost than theory t_2. If such a theorem can be proved, the primitive refinement relation can be inferred to hold between any two versions f_1 and f_2 that differ only by application of the *GSA* operator. Efficiency impact theorems support the claim that analytic methods can partially order approximate theories according to computational efficiency (Claim #4 in Section 1.6.5).

Efficiency impact theorems enable the system to infer the *global* changes in efficiency that result from *local* modifications of individual functions. The proofs can be tricky due to the potential interactions between different parts of theories. When version f_1 is replaced by version f_2, cost must rise regardless of the definitions of all the other functions appearing in theories t_1 and t_2. For this reason, the impact theorems do not hold in the absence of any constraints on the manner in which different functions interact with each other. Each theorem will be supported by *auxiliary assumptions* that constrain such interactions. These assumptions will be seen to hold in many but not all situations.

The efficiency impact theorem for *Function Invariance (FI)* will be presented first. The *FI* theorem uses the cost measure *Ecost* shown in Figure 4-7. *Ecost(t,e,f)* represents the cost of evaluating function *f(e)* under theory *t* for an individual example argument *e*. The impact theorem for *Argument Abstraction (AA)* will be presented second. The *AA* theorem will be seen to depend in part on the *FI* theorem. It uses the cost measure *Acost*, also shown in Figure 4-7. *Acost(t,f)* represents an average cost taken over some set of examples. The impact of *Equiprobable Random Variables (EP)* will follow as a special case of the *FI* theorem. An efficiency impact theorem for *Probabilistic Independence (IN)* was not necessary, for reasons to be discussed below.

- *Ecost(t,e,f)*: The cost of evaluating function *f* applied to example *e* using theory *t*, including the costs associated with calls to other functions defined in theory *t*. Let *Etime* and *Espace* refer to time and space costs respectively.
- *Acost(t,f)*: The average cost of evaluating function *f* using theory *t*, averaged over all examples in the scope of theory *t*. Let *Atime* and *Aspace* refer to time and space costs respectively.

Figure 4-7: Cost Function Definitions

The proofs of impact theorems involve many comparisons of costs associated with different theories. For each comparison of the form: $Cost(t_i)=Cost(t_j)$ or $Cost(t_i) \leq Cost(t_j)$, each term *Cost(t)* may be viewed either as a scalar quantity, or as a multidimensional cost vector. All steps of the proofs will hold in either case. If costs are viewed as vectors, then assertions of equality/inequality must be taken to mean uniform equality/inequality across all dimensions, as defined in Figure 4-6. Likewise, all sums or differences between costs must be viewed as vector operations.

4.4.6.1. Impact of Function Invariance

The efficiency impact theorem for *Function Invariance (FI)* is stated in Figure 4-8. The theorem assumes two versions g_1 and g_2 of function *g*. Version g_1 simply returns a constant, while version g_2 computes an arbitrary expression. The first form of *Function Invariance (FI)* shown in Figure 2-13 could have been used to create version g_1 from version g_2. The theorem further assumes that theories t_1 and t_2 use versions g_1 and g_2 respectively, and agree on all other function versions. Under these conditions, the theorem asserts that theory t_1 is at least as efficient as theory t_2 on evaluations of any function *f* on any example argument *e*.

Given: a. Version $g_1(x)$ = Constant
b. Version $g_2(x)$ = Expression[x] (Expression involving "x")
c. Theories t_1 and t_2 such that Variant(t_1,t_2,g,g_1,g_2)

Prove: $(\forall\ f,e)\ Etime(f,e,t_1) \leq Etime(f,e,t_2)$

Figure 4-8: Efficiency Impact of Function Invariance

1. Consider the tree of function calls generated during evaluation of *f(e)* in theory *t*. Let *G(t), A(t),* and *D(t)* be a partition of the nodes in the evaluation tree for theory *t*. The set *G* includes all nodes directly calling function *g* that are not called recursively by some other call to *g*. The sets *A* and *D* include all ancestors and descendants of nodes in G respectively.
2. Express *Etime(f,e,t)* as a sum over all the nodes of the evaluation tree. For each node *n*, let *C(t,n)* be the cost directly attributable to node *n*, excluding costs associated with calls to children of node *n*. The sum may be decomposed into parts associated with sets A, G and D:

$$(\forall\ f,e)\ \mathtt{Etime}(f,e,t) = \mathtt{Asum}(t) + \mathtt{Gsum}(t) + \mathtt{Dsum}(t)$$

$$\mathtt{Asum}(t) = \textstyle\sum\ (n\ \mathtt{in}\ A(t))\ C(t,n)$$

$$\mathtt{Gsum}(t) = \textstyle\sum\ (n\ \mathtt{in}\ G(t))\ C(t,n)$$

$$\mathtt{Dsum}(t) = \textstyle\sum\ (n\ \mathtt{in}\ D(t))\ C(t,n)$$

3. Assume the evaluation trees for theories t_1 and t_2 are isomorphic, after the nodes in the sets $D(t_1)$ and $D(t_2)$ are pruned. Corresponding nodes are assumed to call the same functions.
4. Assume the cost $C(t,n)$ of a node n depends only on the version used in theory t of the function called at that node, and is independent of the arguments of the call to that function.
5. Assume the cost of referencing a constant is as small as the cost of evaluating any expression.
6. The costs of nodes in sets $A(t_1)$ and $A(t_2)$ must be equal: Due to the isomorphism of evaluation trees, corresponding nodes in $A(t_1)$ and $A(t_2)$ call the same functions. Since these functions are all distinct from g, and theories t_1 and t_2 agree on definitions of all functions other than g, corresponding nodes must call the same *versions* as well. Since cost depends on the version only, the corresponding nodes have equal costs: $Asum(t_1)=Asum(t_2)$
7. The costs for nodes in set $G(t_1)$ are no greater than the costs of nodes in $G(t_2)$: Every node in $G(t_1)$ references a constant. Due to the isomorphism of evaluation trees, the corresponding nodes in $G(t_2)$ evaluate *Expression[x]*. Since the cost of a constant is as small as the cost of any expression, each node in $G(t_1)$ costs no more than the corresponding node in $G(t_2)$: $Gsum(t_1) \leq Gsum(t_2)$
8. The costs for nodes in set $D(t_1)$ are less than or equal to the costs of nodes in $D(t_2)$, since the set $D(t_1)$ is empty: $Dsum(t_1) \leq Dsum(t_2)$
9. Comparing the sums over the A, G and D sets:

$$(\forall\ f,e)\ \mathtt{Etime}(f,e,t_1) \leq \mathtt{Etime}(f,e,t_2).$$

4.4.6.2. Impact of Argument Abstraction

The efficiency impact theorem for *Argument Abstraction (AA)* is stated in Figure 4-9. The theorem assumes two versions g_1 and g_2 of function g, which both evaluate *Expression[x]*. Each version also uses memoization to store previously computed values. Version g_1 differs from version g_2 by applying the function *Abstract* to the argument x before checking the lookup table. The definition of *Argument Abstraction (AA)* shown in Figure 2-16 could have been used to create version g_1 from version g_2.[38] The theorem further assumes that theories t_1 and t_2 use versions g_1 and g_2 respectively, and agree on all other function versions. The costs of theories t_1 and t_2 are measured in terms of $Acost(f,t)$, i.e., average costs taken over a set of examples. The theorem asserts that theory t_1 has average cost at least as low as theory t_2 on evaluations of any function f.

1. First repeat the analysis of the efficiency impact of *Function Invariance*. Divide the evaluation tree into sets A, G and D. The portions of $Etime(f,e,t)$ associated with nodes in the A sets are the same for t_1 and t_2. By averaging over all examples, the same can be concluded for $Atime(f,t)$. The analysis of *Argument Abstraction* can focus on time costs associated with the sets G and D.
2. Assume that when a miss occurs for both t_1 and t_2, and both must evaluate *Expression[c]*, the

[38] The second form of *Function Invariance (FI)* shown in Figure 2-13 could also have been used to create version g_1 from version g_2. When a single argument of $f(x_1,...,x_n)$ is set to a constant, the effect is equivalent to an abstraction that performs projection of tuples.

```
Given: a. Version g1(x) = Expression[Abstract(x)]
       b. Version g2(x) = Expression[x] (Expression involving "x")
       c. Theories t1 and t2 such that VARIANT(t1,t2,g,g1,g2)
       d. A collection of examples: {e1,...,en}.

          Evaluation of Expression[x]:

                        1. Let L = Lookup(f,x);
                        2. If L ≠ Nil
                              then Return L
                              else 1. Let R = Expression[x];
                                   2. Store(f,x,R);
                                   3. Return R.

Prove: Atime(f,t1) ≤ Atime(f,t2)
```

Figure 4-9: Efficiency Impact of Argument Abstraction

evaluation subtrees are isomorphic. The time cost of evaluating *Expression[x]* is therefore the same whenever this expression is evaluated under both theories t_1 and t_2.

3. Let *NC* be the number of calls to function *g* that occur in processing examples $e_1,...,e_n$. Let $M(t_1)$ and $M(t_2)$ be the numbers of times that a *miss* occurs when looking up the value of *g(x)* in the memo table under theories t_1 and t_2 respectively.

4. Let $ATC(t_1)$ and $ATC(t_2)$ be the average time costs of the subtrees rooted at nodes in the *G* sets for t_1 and t_2. Assume the operations *Abstraction*, *Lookup*, *Store*, *If*, and *Return* all have constant time costs, independent of their arguments.

```
ATC(t1) = Etime(Abstraction)
          + Etime(Lookup) + Etime(If) + Etime(Return)
          + (M(t1)/NC) * [Etime(Expression) + Etime(Store)].

ATC(t2) = Etime(Lookup) + Etime(If) + Etime(Return)
          + (M(t2)/NC) * [Etime(Expression) + Etime(Store)].

Difference = ATC(t2) - ATC(t1)

           = (1/NC) * [M(t2)-M(t1)]
             * {Etime(Expression) + Etime(Store)}
             - Etime(Abstraction)
```

5. The cost comparison depends on the miss rates $M(t_1)$ and $M(t_2)$. Let D_g be the domain of function *g*. Let R_a be the range of the Abstract operation. Assume that calls to *g(x)* are such that *x* is evenly distributed over D_g and *Abstract(x)* is evenly distributed over R_a.

   a. Case A: $NC << R_a << D_g$: The number of calls is so low compared to the sizes of sets R_a and D_g that the lookup hit rate is near zero for both t_1 and t_2. Therefore $M(t_1)=M(t_2)=NC$:

   ```
   ATC(t2) - ATC(t1) = - Etime(Abstraction).
   ```

   Since the lookup hit rate is not changed by abstraction, the time cost of abstraction is wasted. The theorem does not hold. Notice that the lookup tables have size proportional to *NC* for both t_1 and t_2. Abstraction does not make for a smaller lookup table either in this case.

   b. Case B: $R_a << NC << D_g$: The number of calls is large compared to R_a, but small compared to D_g. The lookup hit rate is essentially perfect for t_1 and zero for t_2. Therefore $M(t_1) = 0$ and $M(t_2) = NC$:

   ```
   ATC(t2) - ATC(t1) = Etime(Expression) + Etime(Store)
                       - Etime(Abstraction)
   ```

   Assuming that the cost of *Abstract* is less than the cost of evaluating *Expression[x]*, then abstraction helps and the theorem holds. Notice that the lookup tables have size proportional to R_a for t_1, and to *NC* for t_2. Therefore abstraction diminishes the size of the lookup table in this case.

c. Case C: $R_a << D_g << NC$: The number of calls is large compared to both D_g and R_a. The lookup hit rate is essentially perfect for both t_1 and t_2. Therefore $M(t_1)=M(t_2)=0$:

   ATC(t_2) - ATC(t_1) = - Etime(Abstraction)

   Since the lookup hit rate is not changed by abstraction, the time cost of abstraction is wasted. Notice however that the lookup table has size proportional to R_a in t_1 and D_g in t_2. Therefore abstraction diminishes the size of the lookup table in this case as well.

4.4.6.3. Impact of Equiprobable Random Variables

The efficiency impact of *Equiprobable Random Variables (EP)* is a corollary of the theorem proved for *Function Invariance (FI)*. Consider what happens when *EP* is applied to *F(x)=Prob[λ(e)B(e)|G]* or *F(x)=Exp[λ(e)N(e)|G]*. Each variant of *EP* would set *F(x)* to a constant expression, (e.g., the constant expressions *1/2*, *1/|Range(f)|* or *Average(Range(f))*.) *EP* is thus equivalent to *FI* except that *EP* inserts a constant expression rather than a constant. The result will be at least as efficient as the original version of *F(x)* provided these constant expressions have sufficiently small costs. In practice, the expressions *1/|Range(f)|* and *Average(Range(f))* are memoized so that their average costs become negligible after a sufficient number of evaluations.

4.4.6.4. Impact of Probabilistic Independence

Efficiency comparisons based on applications of *Probabilistic Independence (IN)* were not needed in the theory spaces generated by *POLLYANNA*. Independence assumptions were applied whenever possible, as described in Figure 2-23. Function versions lacking the *IN* assumption were not included in the approximation generator final state. As a result, no pairs of versions differing *only* by an application of an *IN* operator were incorporated into *POLLYANNA*'s theory spaces. If some versions lacking *Probabilistic Independence (IN)* assumptions had been used to generate theory spaces, an appropriate efficiency impact theorem would have been needed.

An efficiency impact theorem for *Probabilistic Independence (IN)* can be envisioned in the following way: Consider the automatic control procedure in Figure 2-23 and the random variable trees in Figures 2-10 and 2-12. Suppose the variable tree is expanded to a depth *d* without using any assumptions of independence. Probabilities and expectation values would be computed by summing over all combinations of values in the ranges of variables at leaf nodes. The resulting theories would have asymptotic complexity $O(R^{b^d})$, where *R* is the range of a typical random variable and *b* is the random variable tree branching factor. In contrast to this, theories that use *IN* assumptions have complexity $O(R^{bd})$, as shown in Section 2.8.2. Asymptotic analysis thus shows that considerable savings result from *IN* assumptions. An efficiency impact theorem would result from applying a similar line of reasoning to the abstract absolute cost model.

4.4.6.5. Implementation of Efficiency Impact Rules

Efficiency impact rules are given a procedural implementation in *POLLYANNA*. The procedures operate by systematically testing all pairs (f_1,f_2) of versions of each function f. After examining the derivation traces for f_1 and f_2 they determine whether the *Primitive-Refinement*(f,f_1,f_2) relation should be asserted to hold. One procedure is used to implement to *Function Invariance (FI)* impact theorem. The *FI* procedure simply tests whether a version f_1 was derived from version f_2 by a operator sequence that includes an *FI* operator. If so, then *Primitive-Refinement*(f,f_1,f_2) is asserted.[39] A second procedure implements the *Equiprobable Random Variables (EP)* theorem. The *EP* procedure first determines the random variables v_1 and v_2 to which *EP* is applied in order to generate each of versions f_1 and f_2. *Primitive-Refinement*(f,f_1,f_2) is asserted whenever v_1 is an ancestor of v_2.

4.4.7. The Significance of Potential Non-Monotonicity

The efficiency impact theorems depend on auxiliary assumptions that limit the circumstances under which the theorems are guaranteed to hold. These assumptions are summarized in Figures 4-10 and 4-11. The two most significant assumptions were used to prove the impact theorems for *Function Invariance (FI)* and *Argument Abstraction (AA)*. They assert that (a) functions have constant costs regardless of arguments and (b) evaluation trees have the same structure before and after application of the generic simplifying assumption. These assumptions can fail to hold in many situations. Costs will fail to be constant when function arguments are lists of varying lengths or numbers of varying sizes. Evaluation tree structure will often vary when theories contain constructs to perform conditional evaluation, e.g., *If-Then-Else*, *Mapcar*, etc. Efficiency measurements obtained from empirical testing will provide evidence of the degree to which these assumptions hold true in the hearts domain. These measurements will be presented in Chapter 5.

- The cost of referencing a constant is as small as the cost of evaluating any other expression.
- The time cost of any node in an evaluation tree depends only on the function version used at that node. It does not depend on the arguments passed to the function.
- The evaluation trees for theories t_1 and t_2 are isomorphic on all examples e, except for subtrees called by function g. Corresponding nodes invoke the same functions.

Figure 4-10: Auxiliary Assumptions for Functional Invariance

POLLYANNA's theory spaces can fail to be monotonic in efficiency when the auxiliary assumptions are violated. The ramifications of non-monotonicity can be seen by examining the table of learning goals shown in Figure 4-1. When the goals involve an absolute cost constraint, monotonicity is used to prune all the descendants of

[39]The *FI* procedure actually depends on both the *FI* theorem and the *AA* theorem. The *Function Invariance (FI) GSA* operator used in *POLLYANNA* allows changing a single argument of a function into a constant. This operator only improves average efficiency of memoized functions, as described by the *AA* theorem.

- The evaluation trees for theories t_1 and t_2 are isomorphic, except for subtrees corresponding to evaluations of function *g*. Whenever *Expression[x]* is evaluated in both t_1 and t_2, the evaluation subtrees are isomorphic. Corresponding nodes invoke the same functions.
- The time cost of any node in an evaluation tree depends only on the function version used at that node. It does not depend on the arguments passed to the function.
- The cost of the *Abstract* operation is no greater than the cost of *Expression[x]* plus the cost of one *Store* operation.
- The operations *Abstract*, *Lookup*, *Store*, *If*, and *Return* all have constant time costs, independent of their arguments.
- The function *g* is called a number of times *N* that is large compared to the range of *Abstract* but small compared to domain of function *g*.
- The calls to *g(x)* are such that *x* is evenly distributed over the domain of *g* and *Abstract(x)* is evenly distributed over the range of *Abstract*.

Figure 4-11: Auxiliary Assumptions for Argument Abstraction

any theory failing to meet this cost bound. Non-monotonicity may cause goal theories to be erroneously pruned. When the goals involve a cost minimization objective, monotonicity is used to order the search. Non-monotonicity can cause the system to erroneously conclude that a non-minimal theory is actually minimal in cost. Monotonicity failures thus can cause errors in the empirical theory space search *TSS* process. Errors resulting from non-monotonicity may not significantly degrade *POLLYANNA*'s capabilities. Most learning goals are achieved only relative to the available candidate theories. Non-monotonicity has the effect of restricting the available candidates. The empirical accuracy and efficiency measurements obtained for each individual theory will still be valid. *POLLYANNA* can be therefore confident about the properties of the theories actually tested and of any goals satisfied in the absolute sense. (See Section 4.3.1).

4.4.8. Local v. Global Comparisons of Efficiency

POLLYANNA's *TSG* module distinguishes between *Refinement* relations between theories and *Primitive-Refinement* relations between functions that represent components of theories. *Refinement* relations involve *global* comparisons of theory efficiency. *Primitive-Refinement* relations involve *local* comparisons of versions of individual functions. The semantic definition of *Primitive-Refinement* provides the link between local and global comparisons. It requires the *TSG* module to analytically infer global properties from local comparisons alone. Both advantages and disadvantages result from this approach.

The principal advantage involves economies of inference. Cost relationships between many pairs of theories result from comparing individual pairs of function versions. If complete theories were compared one by one, the *TSG* process would be more time consuming. An additional advantage concerns the efficiency impact theorems presented and proved above. It sufficed to prove only one theorem for each generic simplifying assumption. The method of inference could be closely tied to the type of assumption under consideration. It was not necessary to consider arbitrary combinations of assumptions.

The disadvantages involve the uncertainty of inferring global cost relationships between theories from local comparisons of function versions. In order to prove the efficiency impact theorems, it was necessary to make auxiliary assumptions that effectively ignore interactions between local modifications and other parts of theories. The so called "simplifying" assumptions are not guaranteed to *simplify* unless the auxiliary assumptions hold.

4.4.9. Equivalence and Redundancy among Candidate Theories

Redundancy is a serious problem for *POLLYANNA*. A great many equivalent theories are contained in the full Cartesian product of function versions taken from the approximation generator final state. If all combinations are passed on to the theory space search module, empirical testing is needlessly expensive since identical candidate theories are tested over and over. Methods for eliminating such redundancy are therefore needed. *TSG* attempts to recognize cases in which candidate theories are equivalent to each other. Results from the empirical phase will show that most, but not all of the redundancy can be eliminated. Such redundancy has been observed in other learning systems as well as *POLLYANNA*. Keller observed semantic redundancy in the space of concept descriptions used by his *MetaLEX* system [Keller 87].

Several possible definitions of equivalence are shown in Figure 4-12. The first definition concerns *semantic* equivalence.[40] Theories t_1 and t_2 considered semantically equivalent with respect to function *f* provided they return the same answers on evaluations of *f(x)* for all arguments. This definition might be modified in case *f(x)* is an evaluation function. Evaluation functions might be considered semantically equivalent even when they return different numerical values for some arguments. A modified semantic equivalence criterion might require only that two evaluation functions yield identical rank orderings among choices. An even weaker condition would only require the two functions to yield identical sets of optimal choices.

POLLYANNA requires an equivalence test that is more selective than semantic equivalence. The criterion of semantic equivalence fails to consider the efficiency levels of candidate theories. Semantically equivalent theories may have different levels of efficiency. In the absence of efficiency objectives, *TSG* could aim to build a candidate theory space omitting duplicate copies of all semantically equivalent theories. In the presence of efficiency goals, pairs of semantically equivalent theories t_1 and t_2 must often be included in the candidate theory space. If analytic methods cannot prove either one of t_1 and t_2 to be more efficient than the other, both theories must be passed on to the empirical testing phase.

[40]The notation *f(t,e)* refers to the value of *f(e)* under theory *t*.

Semantic Equivalence:

$$\mathtt{Equivalent}(t_1,t_2,f) \Leftrightarrow (\forall\ e)\ f(t_1,e)=f(t_2,e)$$

Operational Equivalence:

$$\mathtt{Equivalent}(t_1,t_2,f) \Leftrightarrow (\forall\ e)\ f(t_1,e)=f(t_2,e) \wedge \mathtt{Cost}(f,t_1)=\mathtt{Cost}(f,t_2)$$

Syntactic Equivalence:

$$\mathtt{Equivalent}(t_1,t_2,f) \Leftrightarrow t_1(f)=t_2(f) \wedge (\forall\ g \in \mathtt{References}(f,t))\ t_1(g)=t_2(g)$$

Figure 4-12: Types of Equivalence Among Theories

Operational equivalence is the criterion most appropriate in principle for a system like *POLLYANNA*. This definition takes account of the computational costs of using a theory. Theories t_1 and t_2 are said to be operationally equivalent with respect to function *f(x)* provided (1) they are semantically equivalent with respect to function *f(x)* and (2) they incur the same computational costs in the course of evaluating function *f(x)*. The relevant measure of cost will depend on the context. In some contexts, only the average costs need be the same. In other contexts, the costs must be identical for all arguments to function *f(x)*.

POLLYANNA is not able to analytically detect all cases of operational equivalence among candidate theories. The system only detects theories meeting a narrower *syntactic* criterion of equivalence. Two theories are considered to be syntactically equivalent on evaluations of *f(x)* provided that function *f* itself, and all functions referenced directly or indirectly by function *f*, are defined by identical lambda expressions. Syntactically equivalent theories return the same results with the same costs on all evaluations of *f(x)* for all arguments. Syntactic equivalence is a therefore special case of operational equivalence. Since theories can be operationally equivalent even when they fail to be syntactically equivalent, the system does not detect all cases of operational equivalence. Such failure to detect equivalence is not fatal. It simply adds to the costs of empirical testing. The procedure *EQUIVALENT*(t_1,t_2,*f*) is used in *POLLYANNA* to test for syntactic equivalence. (See Figure 4-13.)

The *EQUIVALENT* procedure could be extended to detect some cases of operational equivalence that are missed by the test for syntactic equivalence. Particularly interesting extensions apply to theories that define choice evaluation functions. A technique for detecting *some* cases of operationally equivalent evaluation functions might operate in the following way: The procedure begins by determining when two theories t_1 and t_2 respectively define function *F(x)* to be equal to constants C_1 and C_2. The function *F(x)* will be constant if either *F(x)* directly returns a constant or if every function called by *F(x)* is constant. A simple recursive test could thus detect some but not all constant theories. Redundant evaluation functions are then detected by noticing when rank order does not depend on the particular constant C_1 or C_2 used for function *F(x)*. If function *F(x)* is itself an evaluation function, only one

```
EQUIVALENT(t1,t2,f):
;;Test whether theories t1 and t2 use the same versions for all functions referenced
;;directly or indirectly by function f.
If [For All (g in REFERENCES(f,t1,{}))] t1(g) = t2(g)
    then TRUE
    else FALSE.

REFERENCES(f,t,Found)
;;Find the set of functions referenced directly or indirectly by function f in theory t.
;;Consider that function f implicitly references itself.
1. Found ← Found ∪ {f}.
2. R ← The set of functions referenced directly in version t(f).
3. For All (g in R) do
      If ¬Member(g,Found))
         then  Found ← Found ∪ REFERENCES(g,t,Found)
4. Return Found.
```

Figure 4-13: Equivalence Test Algorithm

constant version is needed and the others are redundant. Furthermore, if *F(x)* is called by an evaluation function of the form *Eval(x)=F(x)+G(x)*, only one constant version of *F(x)* is needed in this case as well. Finally if *F(x)* is called by an evaluation function of the form *Eval(x)=F(x)*G(x)*, only one positive constant version and one negative constant version are needed. The other constant versions of *F(x)* are redundant.

4.4.10. Completeness and Consistency of Theories

Potential problems of completeness and consistency arise in the context of the intractable theory problem. Consider what happens when simplifying assumptions are used to replace an initially intractable theory t_1 with a new theory $t_2=t_1+Assumptions$. In most cases the simplifying assumptions will be inconsistent with the initial theory. For example, the definition *f(x)=Constant* can easily be inconsistent with *f(x)=Expression[x]*. If proof by contradiction is used as a method of inference, inconsistency can be catastrophic. Inconsistency can be avoided if suitable parts of the initial theory are removed when the assumptions are added; however, problems of completeness result when too much is removed.

Consistency and completeness are trivial problems when approximate theories are represented in the formalism used by *POLLYANNA*. (See Section 1.2.2.) A theory is *complete* when *at least* one definition is present for each function referenced in the theory. A theory is *consistent* when *at most* one definition is present for each function referenced. This fortunate situation results from the specific architecture of the approximation generator. (See Figure 2-5.) Theories are approximated when initially exact function versions are *replaced* with approximate versions. Well formed approximate theories thus result from selecting any tuple from the Cartesian product of version sets appearing in the approximation generator final state.

Alternative representations would have presented greater difficulties of completeness and consistency.

Suppose domain theories were represented as sets of predicate calculus clauses in conjunctive normal form (*CNF*). Simplifying assumption could be introduced into such a domain theory by adding new clauses to the initial set of clauses. When such a process is used to make approximations, a difficulty arises when attempting to determine which old clauses should be removed. There is no general method of determining which old clauses to retract since *CNF* consistency is undecidable [Lewis and Papadimitriou 81].

4.4.11. The TSG Algorithm

The theory space generation algorithm is shown in Figure 4-14. The top level routine *TSG* takes three input arguments. *FUNCTIONS* is a list of the names of functions appearing in the approximation generator final state. *PR* is a function implementing the primitive refinement relation, i.e., *PR(f,v)={v'|Primitive-Refinement(f,v,v')}*. *MINIMAL(f)* returns a set containing all versions of *f* that are minimal according to the partial order defined by the primitive refinement relation. The parameter *TOP* is the name of the top level function to be evaluated by theories in the space. *TSG* returns a space of theories for evaluations of the function *TOP*. The space is represented in three parts. *ROOTS* is a set of maximally efficient theories. *SPACE* is the set of all theories in the space. *LINKS* represents the refinement relation among theories in the space.

The function *SUCCESSORS(t,TOP,PR)* carries out the key steps of the generation process. It returns a set of refinements of theory *t*. Refinements are constructed by first finding the set of functions referenced directly or indirectly by *TOP* in theory *t*. For each referenced function *f*, the system finds all the primitive refinements *PR(f,t(f))* of the version *t(f)* of function *f* currently used in theory *t*. Each primitive refinement is substituted for the current version of *f* to form a distinct new theory. Refined theories thus result from replacing versions of referenced functions with their primitive refinements.

The theory space search *(TSG)* algorithm attempts to implement the criterion of syntactic equivalence defined in Figure 4-12. Although *TSG* does not explicitly call the procedure *EQUIVALENT*, it does remove most cases of syntactic equivalence. Since refinements result from modifying only definitions of *referenced functions*, all refinements of theory *t* are guaranteed to be syntactically non-equivalent to theory *t*. As a result, *TSG* generates only a subset of the full Cartesian product of function versions. If this restriction were lifted, refinements would result from modifying the definitions of *any functions*. Some refinements of theory *t* might well be equivalent to theory *t*. The resulting algorithm would generate the entire Cartesian product of function versions.

TSG can sometimes fail to fully implement the syntactic criterion of equivalence. This limitation can occur when the theory space fails to have a unique root. The root will not be unique if *MINIMAL(f)* contains more than one

```
TSG(FUNCTIONS,PR,MINIMAL,TOP)
;;Create a space of theories for evaluating the function TOP.
;;Return the space, the root theories, and the refinement relation.
1. ROOTS ← FIND-ROOTS(FUNCTIONS,MINIMAL).
2. SPACE ← ROOTS.
3. LINKS ← {}.
4. Q ← ROOTS.
5. While non(Empty(Q)) do
     a. t ← Front(Q).
     b. F ← SUCCESSORS(t,TOP,PR).
     c. SPACE ← SPACE ∪ F.
     d. LINKS ← LINKS ∪ {(t,t') | t' ∈ F}.
     e. Q ← Q ∪ F.
6. Return ROOTS, SPACE, LINKS.

FIND-ROOTS(FUNCTIONS,MINIMAL)
;;Find the set of all theories that have MINIMAL versions
;;of each function in the set FUNCTIONS.
1. ROOTS ← {}
2. Let {f1,...,fk} be the members of FUNCTIONS.
3. CP ← Cartesian-Product (i = 1 ... k) MINIMAL(fi).
4. For each tuple (v1,...,vk) in CP do
       a. Let t be a new theory.
       b. For each ni (i = 1 .. k) do t(fi) ← vi.
       c. ROOTS ← ROOTS ∪ {t}.
5. Return ROOTS.

SUCCESSORS(t,TOP,PR)
;;Find all theories that are refinements of t, and are not
;;equivalent to t on evaluations of TOP.
1. R ← {}.
2. For each f in REFERENCES(TOP,t,{}) do
    For each v in PR(f,t(f)) do
      a. Let t' be a copy of theory t.
      b. t'(f) ← v.
      c. R ← R ∪ {t'}.
3. Return R.
```

Figure 4-14: Theory Space Generation Algorithm

element for some function *f*, i.e., if some version set contains two elements that are minimal on the corresponding primitive refinement relation. In such cases, the set *ROOTS* may contain theories that are equivalent under the definition of Figure 4-13. This redundancy could be removed only by introducing unfortunate side effects. Some pairs of equivalent roots can be refined in different ways leading to non-equivalent refinements. Removing redundant theories from *ROOTS* would have the also remove other theories from the space that are reachable in no other way. The redundancy in *ROOTS* can be avoided using the *EQUIVALENT* test during theory space search.

4.4.12. Theory Spaces Generated for Hearts

A theory space for the hearts domain is shown in Figure 4-15. This space was generated using the *TSG* algorithm on the approximation generator output shown in Appendix J. Only selected function versions from the final state were used for the purposes of this example. These included two versions each for the four key functions: *Exp-Win*, *Exp-Trick-Value*, *Exp-Next-Win* and *Exp-Next-Trick-Value*. The primitive refinement relations each include exactly one pair: *[(EWN-0,EWN-1)]*, *[(ETV-0,ETV-1)]*, *[(ENW-0,ENW-1)]* and *[(ENT-0,ENT-1)]*. For all other functions with multiple versions, only the most sophisticated version was used. (The term "Small Hearts Theory Space" shall be hereafter used to refer to this small space of sixteen theories.)

Key: $(v_1,v_2,v_3,v_4) = (EWN\text{-}V_1,ETV\text{-}V_2,EWN\text{-}V_3,EWN\text{-}V_4)$

EWN: Exp-Win
ETV: Exp-Trick-Value
ENW: Exp-Next-Win
ENT: Exp-Next-Trick-Value

Figure 4-15: Small Hearts Theory Space

A highly regular structure is exhibited by the small hearts space in Figure 4-15. The regularity results from two features peculiar to this example. Notice that the space is a lattice. The lattice structure is a consequence of the fact that each primitive refinement relation is itself a lattice, i.e. a small lattice of two versions. This space happens also to include every member of the Cartesian product of version sets. All tuples in the product are non-equivalent, since no function with multiple versions references another function with multiple versions. The Cartesian product is also a binary 4-cube since each function has at most two versions.

A larger hearts theory space results from including all versions of all functions listed in Appendix J. The larger space includes *72* candidate theories. A considerably larger space would have resulted from including the entire Cartesian product of version sets in Appendix L. Eight functions with two versions each would have resulted in a Cartesian product of *256* theories. The equivalence criterion in Figure 4-13 therefore results in a space that is *72*% smaller than the Cartesian product. Results to be presented in Chapter 4 will suggest that several sets of equivalent theories remain among the *72* theories. Stronger tests of equivalence would therefore be useful. (The term "Large Hearts Theory Space" shall be hereafter used to refer to this larger space of seventy-two theories.)

4.4.13. The Size of *POLLYANNA*'s Theory Spaces

The success of empirical learning depends in part on the size of theory spaces generated by *POLLYANNA*. Depending on the number of candidate theories to be tested, empirical learning may be more or less expensive. The size of a theory space depends on the control strategy used in the approximation generator. The size analysis will be carried out for three different strategies. The first assumes an approximation generator using the automatic control procedure *GENERATE* to create the input for theory space generation. (See Section *S-CONTROL* and Figure 2-23.) The resulting theory spaces are extremely large. The second and third analyses are based on proposed modifications to the control procedure. The procedures *GENERATE-A* and *GENERATE-B* are more selective than the original procedure. (See Section *S-RESTRICTED-CONTROL* and Figure 2-24.) They result in much smaller spaces. The role of human control in *POLLYANNA* will be clarified by these analytic results. Theory spaces were generated for the hearts domain using human control in the approximation generator. A hearts theory space will be compared in size to those that would result from using three different automatic control procedures. This analysis serves to support the claim that empirical learning in hearts would be feasible even in the absence of human control (Subsidiary Claim #2 in Section 1.6.2.2).

4.4.13.1. Blind Generation

The size of *POLLYANNA*'s theory spaces can be analyzed by examining the control procedure *GENERATE*. (See Section 2.7 and Figure 2-23.) The total number of theories $S(d)$ can be expressed as a recurrence relation in the depth d to which the random variable tree is expanded. At each level of recursion, *GENERATE* results in several different versions of the function $F(x)$. Two versions result from applying *Equiprobable Random Variables (EP)* before or after testing whether the random variable $\lambda(e)h(e)$ is known in the current state. (*Version#0* and *Version#1*.) Two more versions result each time the definition of $\lambda(e)h(e)$ is unfolded. (*Version#2* and *Version#3*.) If the variable $\lambda(e)h(e)$ can be unfolded N times, a total of $2*N+2$ versions of $F(x)$ are produced at each level of recursion. (*Version#0* and *Version#1* plus N copies each of *Version#2* and *Version#3*.) Each unfolded version of $F(x)$ calls b different functions of the form $G_i(x,v_i)$, where b is the branching factor of the random variable tree. Each $G_i(x,v_i)$ corresponds to $S(d\text{-}1)$ different theories. The total number of theories is therefore $S(d)=2*\{N*[S(d-1)]^b+1\}$. Simple lower and upper bounds can be determined for the recurrence relation.[41] The bounds show that $S(d)$ is at least doubly exponential in the depth d to which *GENERATE* expands the random variable tree. This analysis is summarized in Figure 4-16. The analytic result concerns only the number of *syntactically* distinct theories. The numbers of *semantically* or *operationally* distinct theories will be much lower in some cases. Empirical evidence for this fact will be presented in Chapter 5.

4.4.13.2. Restricted Generation

Smaller theory spaces result from using the restricted control procedures *GENERATE-A* and *GENERATE-B*. (See Section 2.7.5 and Figure 2-24.) The procedure *GENERATE-A* limits redundancy by avoiding versions that test for known values of variables not appearing in the problem state description. Theory spaces resulting from *GENERATE-A* have size $S(d)$ given by the alternate recurrence relation in Figure 4-16. This recurrence was derived by making two simplifying assumptions: It assumes that each random variable $\lambda(e)h(e)$ can only be unfolded one time, i.e. $N=1$. It also assumes that only a small proportion of random variables can appear in the problem state description, so that only two versions (*Version#0* and *Version#2*) are generated for most nodes in the tree. Simple upper and lower bounds can be determined for the alternate recurrence relation. (See Figure 4-16.)[42] The bounds show that $S(d)$ is still at least doubly exponential in the depth d to which the random variable tree is expanded by the *GENERATE-A* procedure.

---

[41]The lower bound results from repeated expansion. The upper bound results from solving the faster growing recurrence: $R(d)=(2N+1)*R(d-1)^b$, with $R(0)=2N+1$.

[42]The lower bound results from repeated expansion. The upper bound results from solving the faster growing recurrence: $R(d)=2*R(d-1)^b$, with $R(0)=1$.

Definitions:

```
S(d) = Size of the theory space.
d    = Depth to which GENERATE unfolds the random variable tree.
b    = Branching factor of the random variable tree.
N    = Number of times each random variable definition can be
       unfolded before reaching an expression computed by a
       primitive operation.
```

Recurrence Relations:

Primary Form:

$$S(d) = 2 * \{ N * [S(d-1)]^{b} + 1\}$$
$$S(0) = 2$$

Upper and Lower Bounds:

$$S(d) \geq 2^{b^{d}} * (2N)^{b^{d-1}+1}$$
$$S(d) \leq (2N+1)^{2(b^{d})}$$

Alternate Form:

$$S(d) = [S(d-1)]^{b} + 1$$
$$S(0) = 1$$

Upper and Lower Bounds:

$$S(d) \geq 2^{b^{d-1}}$$
$$S(d) \leq 2^{2(b^{d-1})}$$

Figure 4-16: Analysis of Theory Space Sizes

The procedure *GENERATE-B* produces only theories representing distinct problem state abstractions. Theory spaces resulting from *GENERATE-B* are somewhat easier to analyze. Let *r(d)* be the number of references to state variables that appear at depth *d* or greater in the random variable tree. Procedure *GENERATE-B* produces two versions for each such variable: (*Version#0* and *Version#1*). It generates only one version for all other variables. The resulting space is bounded by $S(d) \leq 2^{r(d)}$. This upper bound corresponds to the size of the cartesian product of all version sets. The upper bound is tight only when state variables never appear as ancestors/descendants of each other in the random variable tree. The exact size will depend on the layout in the random variable tree of references to state variables.

4.4.13.3. The Size of Hearts Theory Spaces

Hearts theory spaces can be compared to those that would have resulted from the automatic control procedures *GENERATE*, *GENERATE-A*, and *GENERATE-B*. Consider using the original procedure *GENERATE* without any additional control information. The resulting theory space has size given by the primary recurrence for $S(d)$ in Figure 4-16. The hearts tree has an average branching factor $b \approx 2$, which can be determined by examining the function definitions in Appendix C. Most random variable definitions can be unfolded only once, so that $N \approx 1$. The approximation generator output in Appendix J required expanding the hearts random variable tree to depth $d=7$. The procedure *GENERATE* would have to expand the tree to this same depth in order to produce all the same approximations that resulted from human control. Using these values in the recurrence relation, the space produced by the original version of *GENERATE* would have size $S \approx 10^{81}$. Exhaustive empirical testing would not be feasible using this theory space. A smaller hearts theory space would result from using procedure *GENERATE-A* to avoid some redundancy limiting insertion of tests for known values. The resulting theory space has size given by the alternate recurrence relation in Figure 4-16, which leads to a value of $S \approx 6*10^{22}$. Exhaustive empirical testing would not be practical using this theory space either.

A much smaller theory space would result from using procedure *GENERATE-B* to produce only candidates that represent distinct problem state abstractions. The size of this space is bounded from above by $S(d) \leq 2^{r(d)}$, where $r(d)$ is the number of references to state variables that appear at depth d or greater in the random variable tree. A total of 35 references to state variables occur at depth 7 or higher in the hearts random variable tree. The space produced using *GENERATE-B* therefore has size at most $S=2^d=2^{35} \approx 32*10^9$. This represents only an upper bound which is not very tight since many of the 35 references appear as ancestors or descendants of each other. Performing a hand simulation of *GENERATE-B* to account for the layout of references in the random variable tree, the state abstraction space is determined to have size $S=52{,}521{,}875$. This candidate space would still be about one million times larger than the hearts theory space of size 72 that resulted from using human control in the approximation generator. The state abstraction space would not include all the candidates generated using human control. Nevertheless, very similar candidates would be generated by the automatic method. (See Section 3.4.5.) The feasibility of exhaustive empirical testing for this hearts theory space is discussed in Section 5.5.

4.5. Theory Space Search (*TSS*)

Theory space search (*TSS*) is guided by the learning goals shown in Figure 4-1. These goals are handled by four different algorithms: *TSS-C, TSS-E, TSS-G, TSS-K*. Some major features are shared by these algorithms. They all commence search at the theory space root.[43] In so doing, they implement *POLLYANNA*'s *optimistic* philosophy of beginning with the most efficient theory in the space. They move to less efficient theories only when the simpler ones fail empirical testing. Each algorithm also exploits the monotonic organization of the theory space, described in Figure 4-2.

The *TSS* algorithms are specifically designed for goal types *C, E, G,* and *K* (Figure 4-1.) They also handle the strictly simpler goals as well. For example, any algorithm for goal type *C* can be used for goal types *A* and *B* as well. One simply sets cost bound to infinity for goal *A* and sets the error rate bound to one for goal *B*. A similar device enables any algorithm for goal *K* to handle goal types *H, I*, and *J* as well. The remaining goal types, *D* and *F*, can be handled trivially by returning the initial intractable theory or the theory space root.

Each algorithm takes two parameters as input. One parameter is the theory space root. The second parameter is a set of training examples. The examples are used to empirically measure the accuracy and efficiency of theories in the space. The empirical measurements are performed in batch mode. All examples must therefore be available at the outset. Each algorithm makes use of the procedure: *EVALUATE(theory,examples)*. This procedure actually carries out the empirical measurements by testing *theory* against each member of *examples*. It returns a tuple of the form: *(theory, error-rate, cost)*. The return value *error-rate* is a number in the interval *[0,1]* representing the accuracy of the theory with respect to the training examples. The return value *cost* is a positive number representing a measure of the efficiency of the theory in the course of processing the example set. The operation of the procedure *EVALUATE* will be discussed in Section 4.5.2. The *TSS* algorithms do not depend on the details of the procedure *EVALUATE*.

4.5.1. Search Algorithms

[43]Trivial modifications of these algorithms are required when the root is not unique.

4.5.1.1. TSS-C: Best Accuracy First Search

The algorithm *TSS-C*, shown in Figure 4-17, is designed to handle goal type *C*, (Figure 4-1). This algorithm is appropriate whenever the context requires a theory satisfying both an efficiency threshold and an accuracy threshold. Algorithm *TSS-C* starts searching at the theory space root. It uses a *best first* heuristic to control the order in which nodes are expanded. At each iteration of the node expansion loop, a minimal error-rate theory is chosen to be expanded next. Since all theories in the queue will satisfy the cost bound, *best accuracy first* is a natural heuristic for choosing among them. This heuristic attempts to minimize the number of calls to *EVALUATE* required to find a theory meeting the accuracy bound. The algorithm also uses a pruning condition to limit the number of theories that must be empirically tested. It expands a theory *t* only if $Cost(t) \leq K_2$, i.e., if theory *t* satisfies the efficiency threshold. Any theory not meeting the efficiency threshold is pruned along with its descendants. Due to the monotonicity condition, the descendants are guaranteed to also violate the efficiency threshold constraint.

```
TSS-C(root,examples,K1,K2):
;;Goal: Error-Rate(t) ≤ K1 and Cost(t) ≤ K2
;;Best first search from root. Search queue prioritized by error rate.
;;Cost bound gives pruning condition. Terminate upon meeting error rate condition.
1. T ← EVALUATE(root,examples).
2. If Cost(T)≤K2 ∧ ERROR-RATE(T)≤K1
      then Return(T)
3. If Cost(T)≤K2
      then Q ← Insert(T,Empty-Queue)
      else Return(Fail).
4. Repeat
      a. If Empty(Q) then Return(Fail).
      b. T ← Min-Error-Rate(Q).
      c. Q ← Q - {T}
      d. For (t in SUCCESSORS(T)) do
             1. T ← EVALUATE(t,examples).
             2. If ERROR-RATE(T)≤K1 ∧ COST(T)≤K2
                   then Return(T)
                   else If COST(T)≤K2
                           then Q ← Insert(T,Q)
   Forever
```

Figure 4-17: Search Given Error and Cost Bounds

4.5.1.2. TSS-E: Efficiency Bounded Depth First Search

The algorithm *TSS-E*, shown in Figure 4-18, is designed to handle goal type *E*, (Figure 4-1). This algorithm is useful when the context requires a theory meeting an absolute efficiency threshold, and prefers optimal accuracy only as a secondary requirement. It systematically checks all theories meeting the efficiency bound by doing a depth first search starting from the theory space root. Search depth is limited by a test that prunes any theory failing to meet the efficiency threshold, along with its descendants. The monotonicity condition guarantees that the descendants also fail to meet the efficiency bound. During the depth first search, the algorithm keeps track of a theory with globally optimal accuracy. It returns this theory as the final answer.

```
TSS-E(theory,examples,K):
;;Goal: Minimize Error-Rate(t) subject to Cost(t) ≤ K
;;Depth first search from root.
;;Cutoff determined by cost bound.
;;Keep track of globally most accurate theory.
;;Assume Error-Rate(Best) initialized to one.
;;Assume Visited[t] initialized to false for all t.
;;Begin Searching with the Call: TSS-E(root,examples,K).
1. T ← EVALUATE(theory,examples).
2. Visited[theory] ← True.
3. If COST(T) ≤ K
      then Begin
           a. If ERROR-RATE(T) < ERROR-RATE(Best)
                 then Best ← T.
           b. For (t in SUCCESSORS(theory)) do
                  If Not(Visited[t]) then TSS-E(t,examples).
           End.
```

Figure 4-18: Search to Minimize Error Given Cost Bound

4.5.1.3. TSS-G: Best Efficiency First Search

The algorithm *TSS-G*, shown in Figure 4-19, is designed to handle goal type *G*, (Figure 4-1). This algorithm is useful when the context requires a theory meeting an absolute accuracy threshold, and prefers optimal efficiency only as a secondary requirement. For this purpose, the algorithm uses a best first search starting at the theory space root. During each iteration of the node expansion loop, it chooses a minimal cost theory to expand next. The algorithm terminates upon removing a theory from the queue and discovering it to satisfy the accuracy threshold. All theories remaining in the queue will have cost equal to or greater than the one just removed. The descendants of theories still in the queue will have greater or equal cost as well, due to the monotonicity condition. Therefore the theory just removed is a minimal cost theory meeting the accuracy threshold.

```
TSS-G(root,examples,K):
;;Goal: Minimize Cost(t) subject to Error-Rate(t) ≤ K
;;Best first search from root.
;;Search queue prioritized by cost.
;;Terminate upon expanding theory meeting error bound.
1. Answer ← Nil.
2. Q ← Insert(EVALUATE(root,examples),Empty-Queue).
3. Repeat
     a. If Empty(Q) then Return(Fail).
     b. T ← Min-Cost(Q).
     c. Q ← Q - {T}.
     d. If ERROR-RATE(T) ≤ K1
           then Answer ← T
           else For (t in SUCCESSORS(T)) do
                    Q ← Insert(EVALUATE(t,examples),Q).
   Until Answer.
4. Return(Answer).
```

Figure 4-19: Search to Minimize Cost Given Error Bound

4.5.1.4. TSS-K: Efficiency Bounded Depth First Search

The algorithm *TSS-K*, shown in Figure 4-20, is designed to handle goal type *K* (Figure 4-1). This algorithm is useful when the context requires a theory satisfying both an absolute cost bound and an absolute accuracy bound. Unlike algorithm *TSS-C* which simply finds any theory meeting these bounds, algorithm *TSS-K* finds all the theories that (a) satisfy these bounds and (b) are Pareto optimal over the set of theories meeting these bounds. The algorithm begins by finding all theories meeting both the efficiency bound and the accuracy bound. It does so by performing a depth first search starting from the theory space root. The search depth is limited by a test using the efficiency threshold as a pruning condition. During the search process, the algorithm maintains the list of all visited theories that meet the accuracy bound. Upon completing the depth first search, the algorithm has found all theories meeting both thresholds. Notice that this result depends on the theory space monotonicity condition. The procedure *Pareto-Optimize* is then used to examine the visited theories to select the ones that are Pareto optimal.

Algorithm *TSS-K* actually produces results that meet slightly stronger conditions than those described by goal type *K* in Figure 4-1. Goal *K* only requires the resulting theories to be Pareto optimal over the set of theories meeting specified accuracy and efficiency bounds. In some cases, the theories returned by the procedure *Pareto-Optimize* will be Pareto optimal over the entire theory space. Suppose theory *t* satisfies goal *K* and is therefore known to be Pareto optimal over all the theories meeting the specified cost and error rate bounds. Suppose further that *Cost(t)* is less than the cost of the most efficient theory remaining in the search queue upon termination of the algorithm. In that case, theory *t* is strictly more efficient than all theories remaining in the queue. It is also strictly more efficient than all their descendants as a result of the monotonicity condition. Theory *t* is therefore Pareto optimal over the entire theory space.

4.5.1.5. Search Algorithms Implemented in *POLLYANNA*

The following search algorithms and procedures have been implemented in POLLYANNA: *TSS-G, Evaluate, Pareto-Optimize* and *Dominates*. The procedures *TSS-G* and *Pareto-Optimize* were used to collect the empirical data that will be presented in Chapter 5. Implementation of these algorithms required only a straightforward translation of the pseudocode into LISP. The procedure *Evaluate* is not so straightforward. This procedure is used to empirically measure accuracy and efficiency. It will be discussed in Section 4.5.2.3. Implementation of the remaining algorithms, *TSS-C*, *TSS-E* and *TSS-K*, would require no more than a straightforward translation from pseudocode into LISP.

```
TSS-K(theory,examples,K1,K2):
;;Goal: Find all theories Pareto optimal on Cost and Error Rate subject to constraints:
;;      Cost(t) ≤ K1 and Error-Rate(t) ≤ K2.
;;Depth first search root.
;;Cutoff determined by cost bound.
;;Keep track of Pareto optimal theories.
;;Assume Candidates initialized to the empty set.
;;Assume Visited[t] initialized to false for all t.
;;Begin Searching with Call TSS-K(root,examples,K1,K2).
;;Then Call Pareto-Optimize(Candidates).
1. T ← EVALUATE(theory,examples).
2. Visited[theory] ← True.
3. If COST(T) ≤ K1
      Then Begin
           a. If Error-Rate(T) ≤ K2
                 then Candidates ← Candidates ∪ {T}.
           b. For (t in SUCCESSORS(theory)) do
                  If Not(Visited[t]) then TSS-K(t,examples).
           End.

PARETO-OPTIMIZE(Candidates)
;;Find the members of Candidates that are Pareto optimal on measures of Error-Rate(t) and Cost(t).
1. Optimals ← {}.
2. For (T in Candidates) do
       If Not(Exists (T' in Candidates)) Dominates(T,T')
          then Optimals ← Optimals ∪ {T}.

DOMINATES(t1,t2)
;;Test if t1 is superior to t2 on measures of cost and accuracy.
;;t1 dominates t2 if t1 is equal or better on each measure and is strictly better on at least one measure.
If {COST(t1) ≤ COST(t2) AND ERROR-RATE(t1) ≤ ERROR-RATE(t2)}
              AND
   {COST(t1) < COST(t2) OR ERROR-RATE(t1) < ERROR-RATE(t2)}
   then True
   else False.
```

Figure 4-20: Search to Pareto Optimize Given Cost Bound

4.5.2. Empirical Evaluation of Theories

Empirical evaluation is performed by the procedure *Evaluate(theory,examples)*, described in Section This procedure takes a theory and a set of training examples as input. It measures accuracy and efficiency by testing the theory against the training examples. Accuracy and efficiency can each be measured in a variety of ways. The appropriate measurement methods must ultimately be determined by context. (See Section 4.3.2). The specific measurement techniques used in the hearts domain implementation will be discussed in following sections. Representation of training examples will be discussed in Section 4.5.2.1. Methods for making accuracy and efficiency measurements will be described in Sections 4.5.2.4 and 4.5.2.5.

4.5.2.1. Representation of Training Examples in Hearts

Training examples from the hearts domain are represented as shown in Figure 4-21. Each example is a pair of the form: *(situation, action)*. The variable *situation* represents a state of the hearts game. The variable *action* describes the correct card choice or choices in the game state *situation*. Examples sets are prepared in two different forms. In one form *(Form#1)* the variable *action* includes two components that define a partition of the legal card choices. The variable *goods* represents the set of legal card choices that the teacher considers optimal for the given situation. The variable *bads* represents the set of the legal card choices the teacher considers to be suboptimal. In the alternate form of training example *(Form#2)* the variable *action* includes only one component called *choice*. The variable *choice* is a single card that the teacher considers to be optimal.

```
Situation:
   t       = Trick Number.
   p       = Player to Play Next.
   legals  = List of Legal Card Choices.
   public  = Description of Game History up to the Current State.
   private = Initial Hand for Player p.

Action:
   Form#1:

      Goods:     List of Optimal Legal Card Choices.
      Bads:      List of Suboptimal Legal Card Choices.

   Form#2:

      Choice:    One of the Optimal Legal Card Choices.
```

Figure 4-21: Training Example Representation

Greater demands are placed on the teacher when he is required to supply training examples in *Form#1*. For each game situation, the teacher must indicate *all* the choices that he would consider to be correct. This requires an especially cooperative teacher. When training examples are supplied in *Form#2*, the teacher need not be so cooperative. Examples in *Form#2* indicate only *one* of the choices that the teacher considers to be correct. Such examples could be obtained by a student watching the teacher in the course of normal card playing behavior. Most of the training example sets tested in *POLLYANNA* have used *Form#1*. A few tests have been run using *Form#2*, in order to determine whether the results depend on having an especially cooperative teacher. These tests will be described in Chapter 5.

The variable *situation* includes several distinct components, as shown in Figure 4-21. These include the trick number *t*, the player *p* who is choosing a card, and the list *legals* of the cards that he is legally entitled to play.[44] The

[44] The list of legal choices could easily be computed by each candidate theory. This information is given for free since learning about legality is not the hard part of the hearts game. The more difficult part involves learning which legal cards are optimal.

variable *situation* also includes the additional components *public* and *private* described in Figure 4-22. These components are represented in a form that allows them to be used as *givens* in evaluating expressions of the form: *Exp[λ(e)f(e)|Given]* or *Prob[λ(e)f(e)=v|Given]*. The *public* component represents a history of the publicly visible events that have occurred so far during the game. It encodes information that is known to all four players of the game. This information is represented as a list of values of three random variables: (1) The variable *λ(d)Card(p,t,d)* equal to the card choice of player *p* in trick *t*; (2) The variable *λ(d)Leader(t,d)* equal to the leader of trick *t*; (3) The boolean variable *λ(d)Unplayed(c,p,t,d)* which indicates whether player *p* has played card *c* by the beginning of trick *t*.[45] The *private* component represents information that is only available to the player *p* who is choosing a card in the current game state. This component encodes the player's hand as it appears at the start of the game. It is represented as a list of values of the random variable *λ(d)Hand(c,d)*. This variable is equal to the name of the player holding card *c* when the cards are dealt.

Public Information:

Evaluations of Card: $\lambda(d)\mathbf{Card}(p_1,t_1,d) = c_1$
$\lambda(d)\mathbf{Card}(p_2,t_2,d) = c_2$
$\lambda(d)\mathbf{Card}(p_3,t_3,d) = c_3$
...

Evaluations of Leader: $\lambda(d)\mathbf{Leader}(t_1,d) = p_1$
$\lambda(d)\mathbf{Leader}(t_2,d) = p_2$
$\lambda(d)\mathbf{Leader}(t_3,d) = p_3$
...

Evaluations of Unplayed: $\lambda(d)\mathbf{Unplayed}(c_1,p_1,t_1,d) = tv_1$
$\lambda(d)\mathbf{Unplayed}(c_2,p_2,t_2,d) = tv_2$
$\lambda(d)\mathbf{Unplayed}(c_3,p_3,t_3,d) = tv_3$
...

Private Information:

Evaluations of Hand: $\lambda(d)\mathbf{Hand}(c_1,d) = p$
$\lambda(d)\mathbf{Hand}(c_2,d) = p$
$\lambda(d)\mathbf{Hand}(c_3,d) = p$
...

Figure 4-22: Public and Private Information

[45]The variables *Leader* and *Unplayed* could actually be computed from the variable *Card*. The computation would be prohibitively expensive in the representation used here. By providing them for free in the game state description, the computation is simplified.

4.5.2.2. Generation of Training Examples in Hearts

Hearts training examples were generated by the computer program outlined in Figure 4-23. This program is based on a strategy developed by a student with considerable experience playing the game of hearts.[46] The procedure *GENERATE-EXAMPLES* creates the initial game state by randomly dealing the cards. It then proceeds to simulate an actual game consisting of *13* tricks in which each of *4* players plays a card. At each turn during the game, the procedure *SELECT-CARDS* is used to divide the legal choices into the sets *goods* and *bads*. *GENERATE-EXAMPLES* then randomly selects a card *choice* from the set *goods* to update the game state. Each training set consist of *52* examples corresponding to *52* successive situations in a single game. Examples can be represented in either *Form#1* or *Form#2* (Figure 4-21), since the program determines the entire set *goods* along with a random *choice* drawn from this set.

The card playing strategy is implemented by the procedure *SELECT-CARDS*. This procedure generates the *action* components of examples as shown in Figure 4-23. A family of similar strategies is described. All the strategies have the same overall structure. They begin by finding the set *safe-cards* of all cards that are guaranteed to result in taking zero points in the current trick. A card *c* is guaranteed to take zero points if: (a) Card *c* is defeated by some card already on the table or (b) Both card *c* and three cards already on the table have point value zero. If the set *safe-cards* is not empty, then at least one card guarantees zero points on the current trick. In this case the strategies proceed to find a member of *safe-cards* that minimizes the expected number of points to be taken on future tricks. In cases when the set *safe-cards* is empty, no card guarantees taking zero points in the current trick. In this case, the strategies proceed to find a member of *legal-cards* that minimizes the expected number of points to be taken in the current trick.

Three heuristic measures are used to predict the expected number of points to be taken by a card *c* in the current trick or in future tricks. These measures are described in Figure 4-23. Roughly speaking, a card will avoid taking points in the current trick if (a) It has low *point-value* (so the trick has low value) (b) It has high *relative-depth* (so lots of cards can defeat it) or (c) It has value zero for *shown-out*, (so no opponent is likely to be void in *suit(c)* and able to play a point card). The conditions for avoiding future trick points are exactly the opposite. A card that maximizes expected points taken in the current trick will minimize expected points taken in future tricks, since a card played in the current trick will not be available for play in future tricks.

The heuristic measures are applied by a successive minimization procedure. Given measures $m_1 ... m_n$, the

[46] Thanks to Marci Pace.

```
GENERATE-EXAMPLES:

1. Generate the initial SITUATION by randomly dealing
   13 cards to each of 4 players.
2. For each of 13 tricks do
     For each of 4 players do
       a. Call SELECT-CARDS to divide the legal choices into the sets
          GOODS and BADS.
       b. Update the SITUATION by randomly selecting a CHOICE from the
          set GOODS.
       c. Record the training example in Form#1 or Form#2.


SELECT-CARDS:

1. Safe-Cards ← Set of Legal Cards Guaranteed to Take Zero Points
                 in the Current Trick.
2. If Safe-Cards = {}
      Then Begin
           ;;Optimize Legal-Cards for the current trick.
           Successively Minimize Legal-Cards on Some Ordered
           Subset of Measures: +Point-Value, -Relative-Depth,
           +Shown-Out.
           End
      Else Begin
           ;;Optimize Safe-Cards for future tricks.
           Successively Minimize Safe-Cards on Some Ordered
           Subset of Measures: -Point-Value, +Relative-Depth,
           -Shown-Out.
           End.

Point-Value(c):      The point value of card c.
Relative-Depth(c):   The number of cards out in Suit(c) of greater
                     rank than c.
Shown-Out(c):        Equals one if someone has shown himself to be
                     void in Suit(c) by failing to play a card in
                     Suit(c) when Suit(c) was the lead suit,
                     otherwise zero.
```

Figure 4-23: Generation of a Training Set

minimization procedure first selects the cards that are minimal on m_1. From these it selects those that are minimal on m_2, and so on until all the measures are used. Thus measure m_i is considered more important than m_j if $i < j$. Measures with higher numbers are used only to break ties after measures with lower numbers are used. Various strategies result from selecting different measures and ordering them in different ways. Figure 4-23 actually describes a total of *196* distinct strategies, only some of which were actually used to generate test examples. The specific strategies used to generate example sets will be described in Chapter 5.

4.5.2.3. Invocation of Candidate Theories

Candidate theories are tested against training examples in the following manner: Given the variable *situation*, a candidate theory *t* is used to partition the set of *legals* into the two sets *t-goods* and *t-bads*. The set *t-goods* represents all the legal cards that are considered optimal under theory *t*. The set *t-bads* represents all the cards considered suboptimal under theory *t*. In order to generate the sets *t-goods* and *t-bads*, the system calls the evaluation function *Exp-Game-Score(t,c,p,tr,private,public),* for each card *c* among the *legals*. To arrange that this function be evaluated under theory *t*, the evaluation function takes *t* as its first argument. The parameter *t* actually refers to the generic function implementing theory *t*, as described in Section 2.3.5. By examining the parameter *t*, the generic function mechanism finds the appropriate definitions for all functions called during evaluation of *Exp-Game-Score*. After evaluating *Exp-Game-Score* for each legal card, the cards that have minimal values are placed in the set *t-goods*. The others are placed in the set *t-bads*. The accuracy of theory *t* is measured by comparing *t-goods* and *t-bads* to the *action* component of the training example, i.e., comparing *t-goods* and *t-bads* to *goods* and *bads* or *choice*. The efficiency of theory *t* is determined by measuring resources consumed while evaluating *Exp-Game-Score*.

4.5.2.4. Measuring Accuracy of Candidate Theories

POLLYANNA uses the two accuracy measures defined in Figure 4-24. One measure is defined for each of the two training example formats described in Figure 4-21. When examples are supplied in *Form#1* the system computes the accuracy measure A_1. This quantity is defined in terms of the set *t-goods* of choices considered optimal by a candidate theory and the set *goods* of choices considered optimal by the teacher. The accuracy measure A_1 is equal to the fraction of choices in the set *t-goods* that also appear in the set *goods*. When examples are supplied in *Form#2*, the system computes the accuracy measure A_2. This quantity is defined in terms of the set *t-goods* and the teacher supplied variable *choice*. The accuracy measure A_2 is simply the fraction choices in *t-goods* that are equal to the teacher's *choice*.

The accuracy measures can be motivated by considering how a candidate theory would be used to actually play hearts in practice. In a given card playing situation, a candidate theory would compute the set *t-goods* of cards considered optimal. This set may contain one or more card choices. To determine an actual choice, the system would select randomly from *t-goods*. The measures A_1 and A_2 can be interpreted in terms of this random selection. Quantity A_1 is the probability that the random choice would be contained in the set *goods* of choices considered optimal by the teacher. Quantity A_2 is the probability that the random choice would be equal to the teacher's *choice*. Each measure thus represents the probability that a candidate theory leads to a correct choice.

```
Examples in Form#1:

     A1 = |T-Goods ∩ Goods| / |T-Goods|

     E1 = 1 - A1

Examples in Form#2:

     A2 = 1 - |T-Goods - {Choice}|/|T-Goods|

        = If Member(Choice,T-Goods)
             Then 1 / |T-Goods|
             Else 0

     E2 = 1 - A2
```

Figure 4-24: Definitions of Accuracy and Error Rate

Errors of inclusion are treated somewhat differently from errors of exclusion by the accuracy measures in Figure 4-24. Both A_1 and A_2 penalize a theory for choices falsely considered to be optimal (false positives); however, they do not penalize for choices falsely considered to be non-optimal (false negatives). The distinction makes sense when considering the context in which these theories would be used. So long as a theory makes *one* of the optimal card playing choices, it does not matter if some truly optimal choices were considered non-optimal. False negatives may indicate some flaws in the underlying semantics of an approximate theory; however, the flaws do not matter in the context of playing a card game.

The measures A_1 and A_2 appear suited to a range of contexts going beyond the game of hearts. They apply as well to any decision problem which requires selecting a subset of optimal choices from a given set of legal choices. These measures should be useful for determining the accuracy of evaluation functions used in other multi-person games. They should also be useful for determining the accuracy of evaluation functions used in ordinary (one person) search control decisions. For all such decision problems, false negatives are irrelevant. Measures of false positives such as A_1 and A_2 would be more appropriate.

A simple relationship exists between the average values of the two accuracy measures A_1 and A_2 shown in Figure 4-24. Let $<A_1>$ and $<A_2>$ represent the average values of these quantities. Considering just a single problem state: $<A_2>=A_1/|Goods|$. Averaging over all game states: $<A_2>=<A_1>/<|Goods|>$, provided one makes the assumption that A_1 and $1/|Goods|$ are statistically independent. Under this assumption, average values of A_1 and A_2 thus differ by a factor equal to $<|Goods|>$, i.e., the number of choices the teacher considers optimal averaged over all problem states. As a result of this relationship, certain types of learning results are independent of whether the examples are provided in *Form#1* or *Form#2*. Given a sufficiently large number of examples, any of the learning goals *(B,D,E,F,H,J)* listed in Figure 4-1 will yield the same results regardless of whether examples have *Form#1* or

Form#2. The remaining goals *(A,C,G,I,K)* will yield the same results only after the error rate bounds are adjusted by a factor of <|*Goods*|>. Despite these invariance properties, the example format may influence the number of examples required for learning to converge. One would expect *Form#1* to require fewer examples than *Form#2*, since *Form#1* avoids the noisy process of randomly selecting *choice* from *goods*.

The accuracy measures A_1 and A_2 suffer from some disadvantages that may be important in the context of hearts. To begin with, all false positives are treated equally by measures A_1 and A_2. This ignores the possibility that some sub-optimal choices may be nearly optimal, while others are badly sub-optimal. These measures also treat all examples with equal emphasis. In practice some examples are more important than others. In some card playing situations, the correct choice is critical. In others, the correct choice may not greatly matter.

Alternate measures of accuracy are suggested by the foregoing observations. Suppose the teacher supplies estimates of the correct values of *Exp-Game-Score* for each card choice. The system might then measure accuracy in terms of the average difference in *Exp-Game-Score* as computed by a candidate theory and the estimate value supplied by the teacher. Unlike the measures A_1 and A_2, this alternate measure *would* be sensitive to the difference between barely sub-optimal choices and badly sum-optimal choices. Another measure would be more sensitive to the distinction between critical and non-critical examples. Suppose the teacher plays one or more entire games against each candidate theory. Accuracy could then be measured in terms of the average final game score obtained by candidate theories. Errors on individual examples would only be measured indirectly in terms of their impact on the final game score. This would give more appropriate weight to critical and non-critical examples. Both of these alternatives place serious demands on the teacher. In one case he must know or guess evaluations of card choices. In the other case, he must play one or more games against each candidate in the explored regions of the theory space. Games against each candidate must be played separately because they would not all follow the same course.

4.5.2.5. Measuring Efficiency of Candidate Theories

Efficiency is measured in *POLLYANNA* using a mechanism that reflects the *abstract absolute cost* model. (See Section 4.4.5). The model is implemented by deciding on a list of primitive operations to be counted during execution of candidate theories. Efficiency is then measured in terms of an abstract *Clock*. This data structure records the number of applications of each primitive operation that occur during execution of a candidate theory. The *Clock* is updated continuously during theory execution by demons associated with primitive operations. The abstract *Clock* used in *POLLYANNA* is defined in Figure 4-25. This data structure actually counts operations by category, rather than by individual operation.

A tradeoff between accuracy and efficiency is involved in the choice of operations to be counted. One

```
Clock: Categories: → Integers

Categories:     Included Operations:

  CALLS:        <Function Invocations>.
  LISTOPS:      Cons, Car, Cdr.
  ARITHMETICS:  +, -, *, /.
  COMPARISONS:  Equal, If.
  KNOWNS:       <Tests of whether a random variable is known>.
  EVALUATES:    <Evaluations of random variables>.
  LOOKUPS:      <Hash table lookups>.
```

Figure 4-25: The Abstract System Clock

approach would count very low level operations, e.g., *Cons, Car, Cdr, Equal, Atom, +, -, *, And, /.* Sufficiently primitive operations are likely to each require constant time for all arguments.[47] A count of low level operations is likely to reflect real time elapsed. The cost of measuring efficiency can unfortunately be rather high when each low level operation invokes a demon to update the *Clock*. The opposite situation results from counting higher level operations. The overhead is lower; however, the high level operations are less likely to take constant time. The measurements are less likely to reflect real time elapsed.

POLLYANNA measures time efficiency in a manner that represents a compromise between high and low level operation counts. The timing demons are associated with high level operations. High level demons are invoked less often than demons associated with lower level operations. The timing mechanism is therefore more efficient. Nevertheless, the high level demons do not actually count high level operations. Instead they estimate the number of lower level operations that correspond to each higher level operation. These estimates are functionally dependent on the arguments to the high level operations. They reflect the fact that high level operations do not require constant time. This approach leads to more accurate efficiency measurements than would result if the high level operations were themselves to be counted.

The abstract *Clock* defined in Figure 4-25 is not directly usable by the theory space search *TSS* module of *POLLYANNA*. The *TSS* algorithms require a numerical scalar value in order to control the search process. The clock vector must therefore be converted into a single number. The conversion is achieved by defining a table assigning weights to categories of operations. The weights are intended to reflect the relative costs of performing each type of operation. A numerical scalar results from computing the weighted sum of components of the clock vector. The experimental tests of *POLLYANNA*'s theory space search module were carried out by setting weights of all categories equal to one. Empirical results can depend on the weights actually used. The sensitivity of *POLLYANNA*'s results to the choice of weights will be examined in Chapter 5.

[47]This is probably true of *car* and *cdr*, but not *equal* or +, which depend on the length of lists or the number of digits in an integer.

POLLYANNA measures time efficiency only. Space consumption is not measured. This reflects a judgment that time is the critical resource for the hearts domain. If space were considered more important in other contexts, *POLLYANNA*'s efficiency measurement apparatus could be changed. Demons would be associated with the creation and destruction of data structures. The demons would track the current number of instances of each data type by counting creation and destruction operations. Measurements of space consumption could be implemented without modifying any of the theory space search algorithms.

4.6. Summary of Empirical Testing Techniques

- **Learning Goals and the Performance Context:** Learning in *POLLYANNA* is guided by accuracy and efficiency goals. A variety of goal types are handled. Goals can require optimizing accuracy and/or efficiency, satisficing accuracy and/or efficiency or both. No single goal type is appropriate for all situations. Appropriate learning goals depend ultimately on the intended performance context. Accuracy will be a priority in some contexts. Efficiency will be a priority in other contexts. Learning goals can specify a balance between accuracy and efficiency that is appropriate to the intended performance context.
- **Theory Space Generation as an Analytic Reasoning Process:** Analytic reasoning techniques are used in *POLLYANNA* to generate efficiently searchable theory spaces. Analytic techniques reason about efficiency of theories to generate spaces monotonic in computational efficiency. Monotonicity facilitates search by enabling portions of the search space to be pruned. Monotonicity is implemented by rules describing the efficiency impact of generic simplifying assumptions. Efficiency impact rules compare the relative efficiency levels of approximate theories. Theorems asserting the correctness of such rules are provable, given assumptions about the interaction of functions appearing in approximate theories. Analytic techniques also reason about equivalence of theories, attempting to avoid generating a space containing redundant, equivalent candidates. Search is therefore facilitated by reducing the size of the candidate theory space.
- **The Sizes of Theory Spaces:** Upper bounds on the sizes of theory spaces can be derived by analyzing the proposed automatic control procedures *GENERATE*, *GENERATE-A* and *GENERATE-B*. The spaces generated by these procedures will depend on the branching factor of the tree of random variables appearing in the initial intractable theory, and the depth to which these procedures expand the tree. Estimates of the sizes of hearts theory spaces that would result from these procedures are derived by examining the structure of the random variable tree appearing in the hearts domain theory.
- **Empirical Learning as a Theory Space Search Process:** Empirical learning is treated as a search process in *POLLYANNA*. Appropriate theory space search algorithms depend on the accuracy and efficiency goals of learning. Four different search algorithms are needed to handle all types of learning goals. The four algorithms differ on the type of search control or pruning condition that is used. All of *POLLYANNA*'s search algorithms are guided by empirical tests that measure the accuracy and efficiency of candidate theories.