Mechanical Generation of Heuristics for Intractable Theories

Thomas Ellman
Department of Computer Science
Columbia University
New York, New York 10027
ellman@cs.columbia.edu

December 1988
CUCS-400-88

Mechanical Generation of Heuristics

for Intractable Theories

Thomas Ellman
Department of Computer Science
Columbia University
New York, New York 10027
ellman@cs.columbia.edu

10 December 1988

Abstract

A domain independent mechanism for generating heuristics for intractable theories has been implemented in the POLLYANNA program. Heuristics are generated by a process of systematically applying *generic simplifying assumptions* to an initial intractable theory. Such assumptions sacrifice the accuracy of an intractable theory to gain efficiency in return. Truth preserving reformulations are also used to enhance the power of generic simplifying assumptions. This paper describes a framework for generating heuristics using these basic types of knowledge. Examples and representations of each are presented along with an architecture within which they interact to generate heuristic theories. Results from testing this technique in the hearts card game domain are presented as well. This work is part of a system combining analytic and empirical methods for learning heuristics. In the analytic phase, a set of candidate heuristics is generated from an intractable theory. In the empirical phase, heuristic theories are evaluated against teacher-provided training examples. This paper concentrates on the analytic process of generating heuristics.

Table of Contents

List of Figures

1 Introduction

Problems of intractability are pervasive in artificial intelligence. Although complete and correct theories can be formulated for many domains, such theories can be useless in practice if they require excessive computational resources. This difficulty arises in planning, design and game playing domains, among many others. Current machine learning technology is not able to remedy the problem of intractability. Analytic learning methods such as explanation-based learning [Mitchell et al. 86], work only with computationally tractable domain theories. Existing empirical methods are not able to exploit intractable theories as a source of background knowledge. This research uses a combined analytic/empirical approach to the problem of intractability. It employs a strategy of sacrificing the accuracy of an intractable theory, to obtain an efficient, heuristic theory in return. Analytic methods are used to generate candidate heuristics by applying *generic simplifying assumptions* to an initial intractable theory. Empirical methods are used to evaluate candidate heuristics by testing them against teacher-provided training examples.

A program called POLLYANNA has been developed to experiment with this strategy for learning heuristics. The architecture of POLLYANNA is described in Figure 1. In the *heuristic generation* (HG) component, three types of knowledge are combined to generate heuristics. These include an *intractable theory*, *generic simplifying assumptions* (GSAs) and truth-preserving *reformulation knowledge*. Each GSA is described by a schema that can be instantiated in the context of a domain theory to form many different specific simplifying assumptions. By systematically instantiating the GSAs, this module produces a collection of heuristics, each representing a partially complete theory. In the *theory space generation* (TSG) component, individual heuristics are combined into complete heuristic theories. The TSG process also draws upon knowledge of the performance impact of GSAs to build a search space of theories partially ordered according to efficiency. In the *theory space search* (TSS) component, training examples are used to guide a search for heuristic theories meeting specified efficiency and accuracy goals. The HG and TSG modules are considered to be analytic, since they operate without training examples. The TSS module is the empirical component of POLLYANNA.

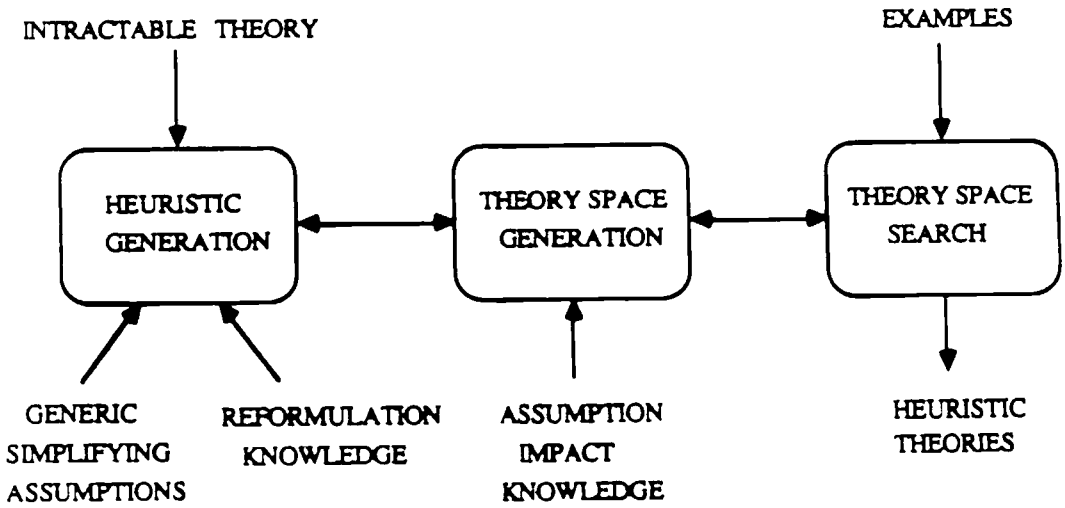

Figure 1: POLLYANNA Architecture

This paper concentrates on the heuristic generation component of POLLYANNA. Particular emphasis is placed on demonstrating that a large set of heuristic theories can be created through systematic application of a few *generic* simplifying assumptions. The interaction of assumptions and reformulation knowledge is also emphasized. Techniques for building and searching a heuristic theory space were discussed in [Ellman 88]. The results presented there were based on complete implementations of the TSG and TSS modules only. Hand coded "outputs" of the heuristic generation process were used to demonstrate the other modules. This paper reports on a complete implementation of the heuristic generator.

This research is similar in spirit to other efforts at mechanical generation of heuristics including [Gashnig 79; Lenat 83; Pearl 84; Kibler 85] and [Unruh et al. 87]. The work reported here is distinguished by using rather different operations to simplify intractable theories. It also applies to domain theories expressed in a different algebraic form. In addition, it differs from the research cited above in the fact of combining both empirical and analytic learning methods. Approximations similar to the simplifying assumptions used in POLLYANNA are discussed in [Keller 87; Mostow and Fawcett 87; Bennett 87]. Another system using both analytic and empirical methods to learn heuristics is reported in [Keller 87].

2 Representation of an Intractable Theory

POLLYANNA has been developed using the card game *hearts*[1] as a testbed domain. A portion of an intractable hearts theory is shown in Figure 2. The theory 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 *ExpGameScore* over all the *LegalChoices*. The evaluation function *ExpGameScore* computes the conditional expectation value of the random variable *GameScore*, assuming that a given card *c* is chosen for play.[2]

The random variable *GameScore* is expressed as a function of the random variables *TrickScore*, *Win* and *TrickValue*. These in turn are written as functions of other random variables. The structure of relationships among random variables is summarized in Figure 9. Each random variable is ultimately a function of the initial game deal *d*, which represents the underlying probabilistic event space. Given a description of the deal *d*, the theory is capable in principle of computing the values of all random variables, and predicting the course of the game. In practice, the theory is intractable for several reasons. To evaluate a card choice using *ExpGameScore*, one must must average *GameScore(p,t,d)* over a large set of deals. In addition, for each deal *d*, even the task of computing the function *GameScore(p,t,d)* is hopelessly intractable.

---

[1]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 card in the same suit as the leader's card will take the trick and become the leader for the next trick. In the simplest version of the game, each player receives one point for every heart card and thirteen points for the queen of spades, if that card is played in a trick that he takes. The game objective is to minimize the number of points in one's score. Complete rules are found in [Andrews 83].

[2]The notation *Exp[λ(e)N(e)]* signifies the expectation value of some numeric function *N(e)*, viewed as a function of an event space. Likewise *Prob[λ(e)B(e)]* is the probability that a boolean function *B(e)* is true. Notice that random variables are represented as λ-Expressions. Both *Prob* and *Exp* can take a boolean function as a second argument indicating the "givens" of a conditional expectation or probability.

```
Choice(p,t,hand,state) =
 =  Minimize( λ(c) ExpGameScore(p,t,hand,state,c),
              LegalChoices(p,t,hand,state) )

ExpGameScore(p,t,hand,state,c) =
 = Exp[λ(d) GameScore(p,d) | NextState(p,t,hand,state,c)]

GameScore(p,d) = Σ (t in TRICKS) TrickScore(p,t,d)

TrickScore(p,t,d) = If Win(p,t,d) Then TrickValue(t,d) Else 0

TrickValue(t,d) = Σ (p in PLAYERS) PointValue(Card(p,t,d))

Win(p,t,d) =
  = Defeats(Card(p,t,d),Card(Left(p),t,d),LeadSuit(t,d))
    ∧ Defeats(Card(p,t,d),Card(Right(p),t,d),LeadSuit(t,d))
    ∧ Defeats(Card(p,t,d),Card(Across(p),t,d),LeadSuit(t,d))

Card(p,t,d) = Choice(p,t,Hand(p,t,d),GameState(p,t,d))
```

Figure 2: Definitions of Some Hearts Functions

3 An Architecture for Generation of Heuristics

As shown in Figure 1, three types of knowledge are used to generate heuristics in POLLYANNA. The generic simplifying assumptions and reformulation knowledge are intended to be domain independent. The intractable theory will naturally vary depending on the domain under study. The internal architecture of POLLYANNA's heuristic generator can be understood in terms of a problem space model. The domain theory corresponds to a problem state. Both generic simplifying assumptions and reformulation knowledge are implemented as operators that modify problem states. As illustrated in Figure 3, each state is described by a table, *Versions*, that associates one or more lambda expressions with each function name. In the initial state, each name is associated with only one definition, i.e. the true definition of the function. In subsequent states, each function name may be associated with several versions. Each operator has the effect of creating a new version of one or more functions. Most of the operators are represented as simple declarative rules for replacing one algebraic expression by another.

```
Versions: Functions → {Set of λ-Expressions}

Functions:     Versions:
f              f0, f1, f2
g              g0, g1
h              h0, h1, h2, h3

Initial State: One λ-Expression for each function.
Later State:   Many λ-Expressions for each function.
```

Figure 3: Heuristic Generation Problem State

In the current implementation, a human user is required to make search control decisions. The user chooses both an operator and an expression to which the operator should be applied. In some cases, the user also supplies a parameter for the operator. The system actually carries out each operator application. For this reason, the heuristic generator is

"mechanical" but not "automatic". Complete automatization appears possible and work on an automatic control strategy is currently in progress.

4 Generic Simplifying Assumptions

Descriptions of some generic simplifying assumptions are shown in Figure 4. These are all considered to be *assumptions* because they are not true in general. Although they introduce errors into a theory, one hopes that the errors will be small, infrequent or at least worth the resulting gain in efficiency. Each GSA is considered *generic* for two reasons. In a given domain, each GSA can be instantiated in more than one way. Furthermore, each GSA is formulated in a manner that purports to apply to a large class of domain theories. The GSAs in Figure 4 are associated with rules describing their impact on the efficiency of domain theories. The assumption impact rules are used by the theory space generator to partially order theories according to efficiency, as indicated in Figure 1. Example applications of these GSAs in the hearts domain will be described in a later section on results.

```
Argument Abstraction (AA):

   (For All x) F(x) = F(Abstract(x))

   (For All x) F(x) = F(Project(x))

   (For All x) F(x) = F(Constant)

Equiprobable Random Variables (EP):

   Prob[λ(e) v=f(e) ] = 1 / |Range(f)|

   Exp[λ(e) f(e) ] = Average(Range(f))

Probabilistic Independence (IN):

   Prob[λ(e) p(e)∧q(e) ] = Prob[λ(e)p(e)] * Prob[λ(e)q(e)]

   Exp[λ(e) f(e)*g(e) ] = Exp[λ(e)f(e)] * Exp[λ(e)g(e)]
```

Figure 4: Generic Simplifying Assumptions

One type of GSA is called *Argument Abstraction* (AA). In its most general form, argument abstraction uses a many to one function *Abstract* to replace the argument of a function to be evaluated. One special case occurs when *Abstract* simply returns a constant. Another occurs when *Abstract* performs projection, i.e., it sets one component of an n-tuple to a constant. All of the AA GSAs improve efficiency by avoiding some repeated evaluations of functions. If one has already evaluated $F(x)$ and wants to evaluate $F(y)$ such that $Abstract(x) = Abstract(x)$, then $F(y)$ need not be evaluated if $F(x)$ was remembered. In POLLYANNA this efficiency is achieved by using memo functions to store prior evaluations in hash tables. AA applies in principle to any theory expressed in terms of functions.

Two other GSAs are used to simplify expressions involving probabilities and expectation values. *Equiprobable Random Variables* (EP) assumes that a random variable is equally likely to manifest any value in its range. This GSA is useful when the distribution of a variable is known but is difficult to compute. *Probabilistic Independence* (IN) is useful for

two reasons. To begin with, this GSA improves efficiency by removing a conjunctive probability that leads to a double summation. The conjunctive probability is replaced by a product of individual probabilities, each requiring a single summation. Probabilistic independence is useful also for enabling EP to be subsequently applied to individual random variables. Both EP and IN apply in principle to any theory formulated in terms of probabilities and expectation values. EP also requires that random variables have finite ranges.

The generic simplifying assumptions in Figure 4 were developed through a process that began by investigating heuristics for the hearts game. Sample heuristics were obtained by analyzing protocols of hearts games played by humans. Some additional heuristics were obtained from a book on hearts [Andrews 83]. After collecting sample heuristics, efforts were made to "derive" the heuristics from the rules of the game. For each heuristic H, an attempt was made to formally prove H using facts from the initial theory, specific simplifying assumptions and reformulations. For example, one naive heuristic ignores all future tricks and chooses cards to optimize the current trick only. This heuristic can be derived from the initial theory by inserting constants to replace evaluation function sub-expressions that correspond to future tricks of the game. The AA GSA was formulated as a general version of this specific assumption. Likewise, the EP GSA was developed by noticing that some heuristics could be derived under the assumption that one's opponents are equally likely to play any card in the deck. Thus the GSAs in Figure 4 were formulated as domain independent versions of specific assumptions found to be useful in hearts.

5 Reformulation Knowledge

A summary of the reformulation knowledge used in POLLYANNA is found in Figure 5. Unlike the generic simplifying assumptions described above, the reformulation operators all preserve the truth of the expressions they modify. These operations are useful for two reasons. Some reformulations can directly improve the efficiency of a domain theory. For example, memo functions can lead directly to efficiency gains, especially when they follow applications of argument abstraction, as described above. Compile time evaluation of constant expressions can also improve efficiency.

- Identities of Set Theory, Algebra and Probability Theory.
- Fold and Unfold Function Definitions.
- Insertion of Memo Functions.
- Evaluation of Constant Expressions.

Figure 5: Reformulation Knowledge

Some reformulations are useful for creating new opportunities to apply generic simplifying assumptions. Two reformulations of this type are shown in Figure 6. These equations respectively describe how to compute probabilities or expectation values of composite functions. They have the effect of replacing a probability or expectation value of one random variable, f, with an expression involving a probability or expectation of some other random variable, g. By generating an expression involving a probability or expectation of g, these reformulations enable a subsequent application of an EP GSA operator to assume all values of the variable g are equally likely. Similar reformulations create new opportunities to apply an IN GSA operator, to assume probabilistic independence.

Interesting interactions also occur between GSAs and the operations of folding and

Probability of Composite Function:

```
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 Composite Function:

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

Figure 6: Probabilities and Expectations of Composite Functions

unfolding function definitions, described in Figure 7.[3] For example, suppose one unfolds a definition of *f(x)* in an expression *E* containing a call to *f(x)*. Subsequent GSA operations applied to versions of *f* will have no effect on the expression *E*. Thus by unfolding a function definition, one can limit the scope of subsequent GSA applications. The opposite effect occurs using the fold operation. Suppose one uses the fold operator to replace an expression *E1* with *f(E2)*, where *E1* is an expression matching the body of *f*. Subsequent GSA operations applied to versions of *f* will impact the expression *f(E2)*. Thus by folding a function definition, one can broaden the scope of subsequent GSA applications. By selectively applying the fold and unfold operators, one can arrange to use a "quick and dirty" 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.

Unfolding a Function Definition:

```
If function F = λ (x1 ... xN) Mac[x1,...,xN] then replace
F(e1,...,eN) with the result of expanding Mac[e1,...,eN].
```

Folding a Function Definition:

```
If expression E contains sub-expressions e1,...,eN and has
the form: Mac[e1,...,eN], then replace E with F(e1,...,eN)
and define F = λ (x1,...,xN) Mac[x1,...,xN].
```

Figure 7: Folding and Unfolding Function Definitions

6 Results

A portion of the final state obtained from one heuristic generation run is shown in Figure 8. This state results from using the heuristic generator to process the hearts theory described in Figure 2. The final state lists multiple versions for some functions. All these definitions are equivalent to the actual lambda expressions generated in LISP notation by POLLYANNA. For clarity and compactness, they have been translated by the author into ordinary function notation. Not all the functions appearing the final state are shown in Figure 8. For those functions shown, only some of their versions are described. The others were omitted for the purpose of brevity. The final state defines some functions that do not actually appear in the initial theory, e.g., *ExpTrickScore*, *PTrickValue* and *PWin*. These names were chosen by the human user upon applying the fold operation to create new function definitions. Typical final states have included about 20-30 function names with about 3-5 versions each. The results depend on search control decisions made by the human user.

---

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

```
ExpGameScore(p,t,hand,state,c):

 Version 1: = Average(Range(GameScore))

 Version 2: = Σ (t' in TRICKS)
                ExpTrickScore(p,t,hand,state,c,t')

ExpTrickScore(p,t,hand,state,c,t'):

 Version 1: = Average(Range(TrickScore))

 Version 2: = Σ (v1 in Range(Win))
                Σ (v2 in Range(TrickValue))
                  (If v1 Then v2 Else 0) *
                  * PWin(p,t,hand,state,c,t',v1) *
                  * PTrickValue(p,t,hand,state,c,t',v2)

PWin(p,t,hand,state,c,t',v1):

 Version 1: = 1 / |Range(Win)|

 Version 2: = ...Average over Range(Defeats)...

PTrickValue(p,t,hand,state,c,t',v2):

 Version 1: = 1 / |Range(TrickValue)|

 Version 2: = ...Average over Range(PointValue)...
```

Figure 8: Example Final State of Heuristic Generator

In order to understand the process used to generate this final state, it helps to consider the diagram in Figure 9. This tree indicates the functional relationships among some random variables in the hearts theory. The exact functional relationships are given in Figure 2. In the initial theory, the evaluation function *ExpGameScore* is written as as an expectation of the top level variable *GameScore*. Truth preserving reformulations allow one to express *ExpGameScore* in terms of probabilities and expectations of variables at lower points in the tree. This process generates various expressions of the form $Prob[\lambda(d)v=f(d)]$ or $Exp[\lambda(d)f(d)]$, where $f(d)$ is some random variable in the tree. Each such expression can be processed in one of three ways. In some cases, it can be directly evaluated since the value of f(d) is already known in the current game state. For example, the value of *Card(p,t,d)* might be known for the players who already played cards in the current trick. If the value is not known, one may apply a GSA to convert the expression into a simpler form. As a third alternative, one can unfold the definition of $f(d)$ and then use the reformulations in Figure 6 to write the expression using variables at lower points in the random variable tree.

Depending on where and when GSA operations are applied, theories with varying costs and error rates can be generated. When a GSA is used to avoid unfolding the definition of some random variable, the subtree under that variable is effectively "pruned". Any known values of variables in the subtree 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 game state. If GSAs are applied at deeper levels, the theory becomes more expensive to evaluate. In return for this computational expense, the theory is able to exploit more information from the current game state.

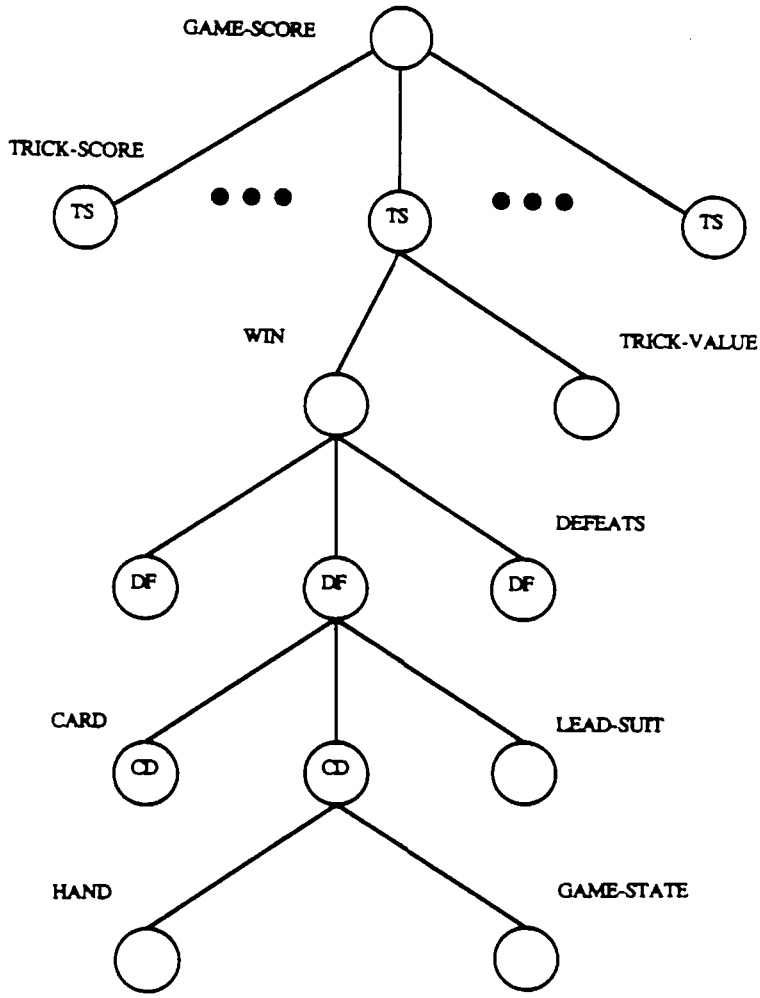

Figure 9: Random Variable Tree

Heuristic theories can be compared in terms of which sub-trees of Figure 9 they do and do not expand. For example, suppose theory *t1* uses Version #1 of *PWin* and Version #2 of *PTrickValue*. Suppose further that theory *t2* uses Version #2 of *PWin* and Version #1 of *PTrickValue*. Theory *t2* thus puts more effort into determining who wins a trick and less effort into determining the trick value. Theory *t1* does the opposite. Each theory thus implicitly takes a position regarding which random variables are most important.

A verbal summary of some heuristics generated by POLLYANNA is shown in Figure 10. A total of 6 different heuristic theories are described by this figure: (1a, 1b, 2aa, 2ab, 2ba, 2bb). The first two heuristics (1a and 1b) were obtained by using AA to ignore terms in the evaluation function that correspond to future tricks. These heuristics optimize the current trick score. Heuristic (1a) results from using EP to ignore the variable *TrickValue* and thus chooses cards of lowest rank to minimize the odds of winning the current trick. Heuristic (1b) results from using EP to ignore the variable *Win* and thus chooses cards of lowest point value to minimize the expected trick value. All the other heuristics (2aa, 2ab, 2ba and 2bb) involve limited consideration of future tricks. These result from combining either (1a) or (1b) with one of two different ways of analyzing the future: (2c and 2d). Both (2c) and (2d) analyze the future using EP to assume that opponents are equally likely to play any card in the deck in future tricks. Heuristic (2c) results from focusing on the variable *Win* in future tricks, while ignoring the variable *TrickValue*. It suggests getting rid of high rank cards to minimize the odds of winning future tricks. Heuristic (2d) takes the opposite

approach of considering *TrickValue* while ignoring the variable *Win* in future tricks. It suggests getting rid of high point value cards to minimize the expected trick score in future tricks. Of course these heuristics are not represented as verbal rules in POLLYANNA. The rules were written by the author to characterize the behavior of function definitions appearing in the final state of the heuristic generator. The functions implementing these heuristics are similar to those shown in Figure 8, although they are somewhat more complicated.

The results described in Figure 10 indicate that POLLYANNA is capable of generating heuristics that perform on the level of a human novice. Some strategies more complex than these have been generated as well; however, efficient heuristics performing on the level of human experts have not been generated. Difficulties arise because the cost of evaluation blows up when the tree of Figure 9 is expanded beyond a certain point. Efficient theories describing advanced heuristics appear to require more powerful reformulations and/or generic simplifying assumptions.

```
1. If some legal card choices risk winning the current
   trick then:
    a. Play a card of minimal rank.
    b. Play a card of minimal point-value.
   Otherwise play any legal card.

2. If some legal card choices risk winning the current
   trick then:
    a. Play a card of minimal rank.
    b. Play a card of minimal point-value.
   Otherwise,
    c. Play a card of maximal rank.
    d. Play a card of maximal point-value.
```

Figure 10: Summary of Heuristics Generated by POLLYANNA

Even the novice level results are significant considering the manner in which they were generated. For a machine that cannot use the initial hearts theory for anything other than direct evaluation, the heuristics in Figure 10 are not obvious. Given the availability of generic simplifying assumptions and reformulation knowledge, these heuristics *can* be extracted from the intractable theory. The results are significant also because the operations used are potentially domain independent. Work in progress is attempting to apply the GSAs of Figure 4 to a stochastic scheduling domain, among others. This work is intended to demonstrate that the generic simplifying assumptions apply to many domains.

After generating the function versions described in Figure 8, POLLYANNA passes these definitions along to the theory space generation module. Various complete theories are then formed by taking one version of each function. The theory space is actually a subet of the cartesian product of the version sets. Many equivalent theories are contained in the complete cartesian product. The theory space generator attempts to remove most of this redundancy. The empirical theory space search module then uses training examples to search for theories meeting specified cost and accuracy goals. A previous paper reported empirical results based on hand coded output from the heuristic generator [Ellman 88]. Those measurements indicated wide variation in levels of accuracy and efficiency of heuristic theories. Empirical tests using mechanically generated heuristics are currently in progress. These tests involve experimenting with various subsets of GSAs and reformulation operators. They are aimed at determining the contribution of each toward improving efficiency with minimal loss of accuracy.

7 Conclusion

The power of generic simplifying assumptions has been demonstrated by results from the hearts domain. These results show that each GSA can be instantiated in many different ways, each of which has a different impact on the domain theory. A small set of GSAs can lead to a large class of heuristics when they are systematically applied to different parts of the theory. Both good and bad heuristics are generated by this process. For this reason, generic simplifying assumptions should be considered a *weak* theory of heuristics. Empirical learning is needed to separate the useful heuristics from the useless ones. The integrated analytic/empirical architecture of POLLYANNA is thus a natural consequence of generating heuristics from generic simplifying assumptions.

8 Acknowledgments

Thanks to Michael Lebowitz and Jack Mostow for many useful discussions about the work reported in this paper. This research was supported in part by the Defense Advanced Research Projects Agency under contract N00039-84-C-0165.

References

[Andrews 83] Andrews, J., D. *Win at Hearts.* Dover Publications, New York, NY, 1983.

[Bennett 87] Bennett, S. W. Approximation in Mathematical Domains. Proceedings of the Tenth International Joint Conference on Artificial Intelligence, 1987.

[Ellman 88] Ellman, T. Approximate Theory Formation: An Explanation-Based Approach. Proceedings of the Seventh National Conference on Artificial Intelligence, 1988.

[Gashnig 79] Gashnig, J. A Problem Similarity Approach to Devising Heuristics: First Results. Proceedings of the Sixth International Joint Conference on Artificial Intelligence, 1979.

[Keller 87] Keller, R. M. The Role of Explicit Contextual Knowledge in Learning Concepts to Improve Performance. Technical Report ML-TR-7, Department of Computer Science, Rutgers University, New Brunswick, NJ, 1987. PhD Thesis.

[Kibler 85] Kibler, D. Generation of Heuristics by Transforming the Problem Representation. Technical Report ICS TR-85-20, Department of Computer Science, University of California, Irvine , CA, 1985.

[Lenat 83] Lenat, D. B. "EURISKO: A program that learns new heuristics and domain concepts." *Artificial Intelligence 21*, 1, 2, 1983, pp. 61 - 98.

[Mitchell et al. 86] Mitchell, T. M., Keller, R. M. and Kedar-Cabelli, S. T. "Explanation-Based Learning: A Unifying View." *Machine Learning 1*, 1, 1986, pp. 47 - 80.

[Mostow and Fawcett 87] Mostow, J. and Fawcett, T. Approximating Intractable Theories: A Problem Space Model. Technical Report ML-TR-16, Rutgers University, Department of Computer Science, New Brunswick, NJ, 1987.

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

[Unruh et al. 87] Unruh, A., Rosenbloom, P. and Laird, J. E. Dynamic Abstraction Problem Solving in Soar. Proceedings of the AOG/AAAIC Joint Conference, 1987.