A Knowledge Based Program For Selecting Problem Solving Paradigms

by

Peng Wu

B.E., Tsinghua University

1982

M.S., Massachusetts Institute of Technology

1988

Submitted to the

Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

August, 1993

Signature of Author ______________________________

Department of Electrical Engineering and Computer Science
September 23, 1993

Certified by ______________________________

Randall Davis
Professor of Computer Science
Thesis Advisor

Accepted by ______________________________

Frederic R. Morgenthaler, Chairman
Committee on Graduate Students

A Knowledge Based Program For Selecting Problem Solving Paradigms

by

Peng Wu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of Doctor of Philosophy
at the Massachusetts Institute of Technology

August 1993

Abstract

Knowledge based systems have been built for a wide variety of application domains, but few areas of AI itself have been explored in this manner. We are building an expert system whose expertise is selecting an appropriate problem solving paradigm (PSP) for a task. It is capable of recommending a total of 1095 different variations of problem solving paradigms.

We use the phrase *problem solving paradigm* for any identifiable problem solving process or its definition at the *task level* in a sense similar to generic tasks. For example, we consider that *model-based diagnosis* is a PSP.

We believe PSP selection has a major impact on problem solving, but has not been adequately addressed to date in AI research. Most problem reformulation methods are targeted to particular PSPs, thus leaving the PSP selection issue virtually untouched. On the other hand, AI research has been accumulating PSPs, usually accompanied by informal statements as to the kind of problems for which they are suitable.

In the thesis, we discuss PSP selection as part of problem solving, describe a representation for PSPs, and document the design and implementation of a knowledge based system for selecting PSPs. We believe systems like this will encourage PSP designers to be more precise about the kind of problems for which their PSPs are designed and will assist others in selecting appropriate paradigms.

We also report work on PSP comparison and on relevance analysis for PSP selection.

Keywords: Artificial Intelligence, Knowledge-based System, Knowledge Representation, Problem Solving, Problem Solving Paradigm Selection, Complexity Analysis, Relevance Analysis.

Thesis Supervisor: Randall Davis
Title: Professor of Computer Science

Acknowledgments

I would like to thank:

Randall Davis, my thesis supervisor, for his guidance, encouragement and many ideas on technical, presentational and research issues, and for his understanding and support at stressful times. Without him, this thesis would not have been possible.

Charles Rich, my thesis committee member, for reading the draft carefully and making comments that improved many parts of the thesis.

Patrick Winston, my thesis committee member, for interest in my work and providing initial comments on the thesis research.

Jeffrey Van Baalen, for explaining his thesis. Understanding of his thesis lead to many key ideas of my thesis.

Walter Hamscher, for interest in my work and generous help at the formative stage of this thesis.

Mark H. Shirley, for interest in my work, generous encouragement and friendship.

Thomas Stahovich, for generous encouragement and friendship.

The MIT AI Lab, for providing an excellent research environment.

My sisters, for constant encouragement and support.

My parents, for constant encouragement, unconditional sacrifice and support.

This thesis describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the author's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under office of Naval Research contract N00014-91-J-4038 and in part by the Digital Equipment Corporation.

Contents

Chapter 1

Introduction

Problem solving underlies many intelligent activities. Problem solving is an intricate process that begins with the perception of a problem and ends with the execution of its solution. We divide the problem solving process into five stages: problem perception, problem solving paradigm selection, representation design, solution construction, and solution execution.

We use the phrase *problem solving paradigm* (PSP) for any identifiable problem solving process or its definition at the *task level* in a sense similar to [Chandrasekaran, 1986]. For example, we consider that `model-based diagnosis` in [Davis and Hamscher, 1988] is a PSP. A PSP is composed of one or more *components*. For example, the model-based diagnosis PSP is composed of three components: `candidate generation`, `candidate verification`, and `candidate discrimination`. Each component has one or more *attributes*. For example, the candidate generation component has one attribute: `method`. Each attribute may have one or more *values*. For example, `up-stream` may be the value of `candidate generation method`. Other examples of PSPs are: `attribute-based classification`, `constraint satisfaction on finite domain`, and `generate and test`.

This thesis reports an expert system whose expertise is selecting an appropriate problem solving paradigm for a task and reports work on PSP comparison and PSP selection oriented relevance analysis.

We believe that PSP selection has a major impact on problem solving: PSP selection may help to define the problem, suggest suitable representations for the facts, and locate heuristics for representation reformulation, identifying irrelevant facts, and constructing solutions.

However, PSP selection has not been adequately addressed to date in AI research. Although problem and representation reformulation may lead to problem solving paradigm change, most of the reformulation methods are targeted to particular PSPs, leaving the PSP selection issue virtually untouched. The disparity between the important role of PSP selection and its current status inspired the work reported here.

Due to the diversity of PSPs and our lack of understanding of them, PSP selection in general is still largely intractable [Newell, 1983]. However, AI research has been accumulating detailed, mechanizable PSPs for use in programs; these PSPs are relatively well defined. We believe that selecting PSPs from among those familiar to AI researchers is within reach.

The PSPs reported in AI literature are usually accompanied by informal or in-direct statements as to the kind of problems for which they are suitable. These statements embody a large amount of knowledge but are difficult to match with real problems. We believe that if we can make these statements precise enough for a PSP selection program, AI PSPs will become more accessible. We believe that building a PSP selection expert system to encode and exploit the knowledge about PSP selection is a reasonable approach.

We believe PSP selection systems will also encourage PSP designers to be more precise in defining the kind of problems for which their PSPs are designed and will assist others in selecting appropriate paradigms.

1.1 A Theory of Problem Solving

Problem solving is a process that begins with the perception of a problem and ends with the execution of its solution. From the perspective of PSP selection, this process can be divided into five stages shown in Figure 1.1.

1) **Perception** Perceive the problem
2) **PSP Selection** Find a problem solving paradigm for the problem
3) **Representation** Represent the problem for that PSP
4) **Solution** Apply the PSP to the representation and find the solution
5) **Execution** Execute the solution

Figure 1.1: Problem Solving Stages

These problem solving stages are "cooperative" in the sense that difficulties in one stage may be alleviated by changes in another. We can achieve overall efficiency of problem solving only by coordinating all the problem solving stages. The impact of the PSP on the overall efficiency of problem solving inspired the work on PSP selection reported in this thesis. In the rest of this section we will illustrate the impact of PSP with two examples, then discuss the role of PSP in problem solving.

1.1.1 Motivating Examples

This section will present two problems. The first is the classic mutilated checker board problem (MCB), that has been traditionally used to illustrate the impact of representation change on problem solving. However, we believe there is also an important change in PSP: a change from constructing a domino cover to proving that there is no cover. The combination of PSP change and representation change has a stronger impact than representation change alone. We will look at the MCB problem here from the perspective of PSP selection.

To help to erase any doubt that PSP change might be merely another name for representation change, we will then present a problem called the Think and Jump problem (T&J). As we will see, the T&J problem is solved adapting the *PSP change* that solved the MCB problem, instead of adapting the well-known *representation change.* The T&J problem will illustrate that an understanding of the involved PSP can help us to discover a suitable representation change, thus demonstrating the utility in making both PSP and its change explicit.

The Mutilated Checker Board Problem

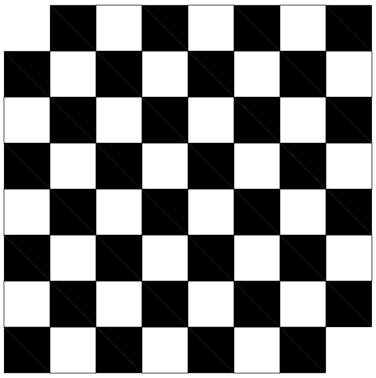

Figure 1.2: Mutilated Checker Board

Figure 1.2 shows a checker board with two diagonal corner squares missing. The MCB problem is to figure out if the mutilated checker board can be exactly covered by dominoes (rectangular tiles) having an area that spans exactly two adjacent squares.

Assume a problem solver is given the MCB problem and is aware of the remaining four problem solving stages: PSP selection, representation design, solution, and execution. Assume that at the PSP selection stage there are only two PSPs suitable for the MCB problem: $\underline{A}$ – try to construct a pattern of dominoes that covers the board exactly, or $\underline{B}$ – try to prove that the objective is impossible by partitioning the squares and looking for some properties of the partition. Assume the problem solver selects $\underline{A}$. In representing the problem for this first PSP, the problem solver decides that the search space for placing dominoes on the board is too large. Assume he then decides to reformulate the problem, using techniques valid for the first PSP such as ignoring the difference between the dominoes, taking advantage of board symmetricity, discovering macro operators, etc. After wondering for a while, the problem solver decides to abandon PSP $\underline{A}$, entirely backs up to the PSP selection stage and selects PSP $\underline{B}$. PSP $\underline{B}$ has two components: partition the squares and find suitable properties. Luckily, one of the known partition methods is to partition the squares according to the board color pattern. This partition dictates that the number of white squares is equal to the number of black squares. At this point the problem solver comes to the solution-finding stage and shows that there is an unequal number of black and white squares, solving the problem. At the execution stage, the problem solver announces the result.

This example is not trying to answer the question about how to select a PSP or where PSPs come from. One possible answer is knowledge engineering, which is the topic of this thesis. The point of this example is that a change in the PSP may have a great impact on representation design. Thus PSP change combined with representation change may have a greater impact on the efficiency of problem solving than representation change alone.

The impact of PSP changes may come from changes in cost functions. For PSP $\underline{A}$, the goal is to show that the given conditions entail that the board can be covered. We may not be able to guarantee the validity of an entailment obtained after homomorphic changes. However, many effective representation changes are homomorphic. Therefore, when homomorphic representation changes are involved after selecting PSP $\underline{A}$, the representation design stage has to carry the burden of guaranteeing the validity of the entailment, or it must pass the burden to later stages. On the other hand, for PSP $\underline{B}$, the goal is to show that the given conditions do *not* entail certain properties. Such non-entailment obtained after homomorphic changes is always valid if we can obtain it at all. Therefore, homomorphic representation changes are cheaper for PSP $\underline{B}$ than PSP $\underline{A}$.

The impact may come from changes in valid representations. For instance, partitioning the board positions according to the color pattern leads to a solution for PSP $\underline{B}$, but it is not a valid representation for PSP $\underline{A}$.

The impact may also come from the heuristics associated with different PSPs which may help later problem solving stages. For instance, the PSP $\underline{B}$ may be associated with a suggestion to partition the board positions in such a way that each operation (placing a domino on the board) involves one board position from each partition, which helps with the representation stage. This kind of heuristic is especially valuable for the representation design stage because of the space of possible representations is enormous.

The Think and Jump Problem

The MCB example has long been used to show the benefit of *noticing* a good representation but it provides no hint on how to *design* a good representation. If the checker board were not colored, we might be still looking for a solution to the MCB problem today. Our next example, the Think and Jump (T&J) problem, on the other hand, will show an example of designing a good representation, by changing the PSP and using heuristics associated with PSPs about representation changes.

Similar to the MCB example, the point of the T&J example will not be about how to select a PSP. The point will be about the role of PSP in problem solving and representation design in order to motivate the main topic of the first section of this chapter—PSP selection.

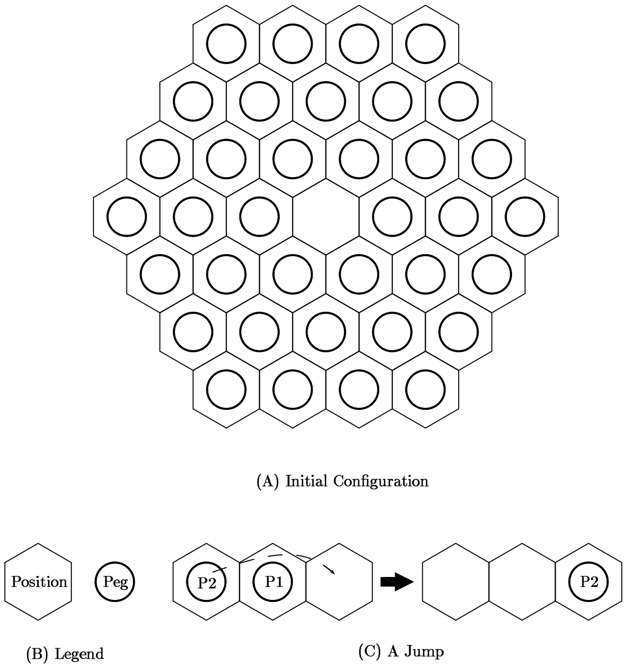

Figure 1.3: Think and Jump

The T&J problem is adapted from the *Think and Jump* game. In this game, the board configuration is a two-dimensional cascade of 37 hexagons (see Figure 1.3, A). Initially, there is a peg on every hexagon except the center one (see Figure 1.3, A and B). There is only one operation: we can remove a peg by having an adjacent peg hop over it (illustrated in Figure 1.3, C). The objective of the game is to end with only one peg on the board by repeating this operation.

The T&J problem is to figure out if the game objective can be achieved. Assume that at the PSP selection stage there are only two PSPs suitable for the T&J problem: $\underline{A}$ – try to achieve the game objective, or $\underline{B}$ – try to prove that the objective is impossible by partitioning the hexagons and looking for some properties of the partition. Assume the problem solver selects $\underline{A}$. In the representation design stage, the problem solver decides that the search space for a successful sequence of jumps is too large and decides to reformulate the problem, using techniques such as ignoring the differences between pegs, taking advantage of board symmetricity, discovering macro operators, etc. After wondering for a while without achieving the game objective, the problem solver decides to abandon PSP $\underline{A}$, backs up to the PSP selection stage, and selects PSP $\underline{B}$. PSP $\underline{B}$ has two components: partition the hexagons and find suitable properties. Luckily, the heuristic associated with PSP $\underline{B}$ leads to a partition in the pattern shown in Figure 1.4, under which each jump involves one hexagon from each partition.

At this point the problem solver comes to the solution finding stage, and notices the following facts:

- Initially, the pegs occupy 12 hexagons of each color - white (w), black (b), and gray (g).

- There are three types of jumps, each of which changes the occupied hexagon as follows:

- Assume there are W W-jumps, B B-jumps, and G G-jumps made in the attempt

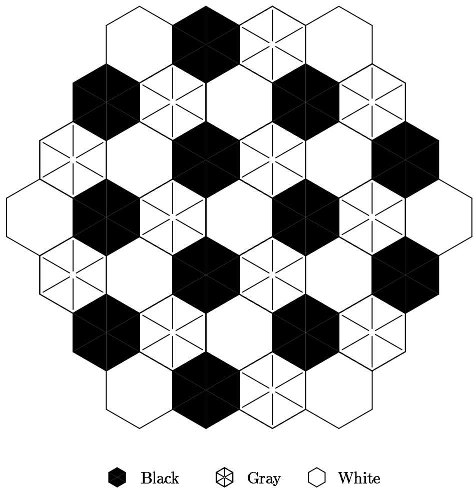

Figure 1.4: Triple-Colored Think and Jump Board

	count change	jump type
W-jump	`- b - g + w`	b over g, g over b
B-jump	`+ b - g - w`	g over w, w over g
G-jump	`- b + g - w`	b over w, w over b

to vacate 35 hexagons (i.e., removing 35 pegs). The hexagon balance would be (Note that the left side of the equation is symmetric about `b`, `g`, and `w`. So we can assume the remaining occupied hexagon is black (b) without losing generality.):

```
 12 (  w + b + g)
+ W (- b - g + w)
+ B (  b - g - w)
+ G (- b + g - w) = b
```

Solving the equation for W, B, and G yields

```
W = G = 11 1/2, B = 12
```

Since it is impossible to make any half jumps, it is proved that the T&J objective is impossible to achieve. The T&J problem is solved.

1.1.2 Problem Solving Paradigm Selection as Part of Problem Solving

The MCB example has long been used to show the benefit of *noticing* a good representation. However, the MCB example does not answer the question about how to *find* a good representation. The T&J example, on the other hand, tries to answer this question directly by demonstrating the role of PSPs in finding a good representation and efficient problem solving. In particular, PSP *B̲* brings us a very specific heuristic about how to change the representation in addition to defining how to use the resulting representation.

By contrast, the only other alternative approach currently available to finding a good representation is representation (or problem) reformulation. Because representation reformulation techniques do not take into account the role of PSPs in problem solving, they are severely hampered by the PSP accidentally chosen for them.

In the rest of this section, we will further motivate PSP selection by discussing the role of PSP selection in problem solving, the role of PSP selection in problem solving research, and the necessity of PSP selection.

The Directional Influence Between Problem Solving Stages

We hypothesize that the influence between different problem solving stages is largely directional: the effect from early stages to late stages is much stronger than vice-versa.

We believe that, when the stages are ordered as in Figure 1.1, decisions at earlier stages help determine the alternatives at later stages. We believe that the difference between sets of alternatives often overwhelms the difference within a set of alternatives.

For example, two alternatives at the problem solving paradigm selection stage for tic-tac-toe are playing by search and by computing best moves. The two PSPs suggest different alternative representations at the representation design stage. There is no need to try all the alternative representations for all the PSPs. The difference between the two paradigms is usually so obvious that we even have a simple heuristic for it: computing a choice it is more efficient than searching for a choice whenever computing is possible. Hence we can select an appropriate paradigm first, then focus on only the set of alternative representations appropriate for the selected paradigm.

The utility of this hypothesis is that we can make selections at earlier stages using relatively inexpensive heuristics which do not depend on accurate comparison between the choices at later stages. Once decisions are made at one stage, we can use more accurate evaluations for only the relevant choices at the next stage, gaining efficiency.

Although heuristics may not always lead to the best solution, we believe we can often find practical solutions efficiently. Therefore, it is a practical strategy to make the selections stage by stage.

The following are some observations about problem solving that are consistent with the directional influence hypothesis:

- Most work on problem solving in AI may be considered to be at the solution stage. At the solution stage, both the problem solving paradigm and representation are given. The improvements available are limited to cutting down duplicated or wasted work. There are no improvements available on necessary work. For instance, suitable search strategies improve problem solving efficiency by reducing backtracking, but will not change the search depth at which a solution is found.

- Work on representation-related issues may be considered to be at the representation design stage. At the representation design stage, the problem solving paradigm is given but one can change the representation. In one aspect, representation directly affects how much necessary, as well as how much duplicated or wasted work, there could be. Therefore, representation (stage 3 in Figure 1.1) has a larger impact than solution finding techniques (stage 4) on problem solving efficiency. For instance, compared with overly detailed representation, say circuits at stateless gate level, suitably abstracted representation, say circuits at flip-flop level, would reduce both the solution depth thus hopefully the search depth and wasted work.

The Automation Frontier

The prevailing wisdom in AI suggests that representation choice is the key to efficient problem solving. Problem perception and problem solving paradigm selection are not seen as important. We conjecture that this is because the problem perception and PSP selection are not significantly understood, thus representation design becomes the first stage of automated problem solving.

But representation choices are constrained by the perception of the problem and the selection of a problem solving paradigm. We thus suggest that selection of the problem solving paradigm plays a role at least as important as that of representation.

Our belief is consistent with the fact that most work on representation design is oriented towards optimization, and the techniques used are PSP specific. For instance, [Amarel, 1968] describes representation reformulations made by ignoring distinctions of individuals, taking the advantage of complement and symmetry, etc. All those representation reformulations are optimizations, that is, they eliminate useless work while leaving the problem perception and PSP intact.

While potentially useful, optimization-oriented representation reformulation suffers from several major problems:

- Representation reformulation trims unnecessary computations but sticks to the original problem solving paradigm and problem formulation. In other words, representation reformulation offers too few promising choices. There are times when either the problem paradigm or the problem perception deserves reconsideration. For instance, when the MCB problem is to be solved by searching for a complete cover, there are no known representation reformulations that will make the problem practically solvable. However, if one changes the PSP to proving the non-existence of a complete cover, there is a representation reformulation that makes this problem solvable.

- The representation space is enormous and representation reformulation often employs little knowledge about how to navigate this space. In a sense, representation reformulation explores too many unpromising choices.

  As an example, consider representation reformulations generated by ignoring individual distinctions. If one has 62 objects (as for example the 62 squares on the MCB) and wants to divide them into 2 sets, there would be 2^{62} possible ways to divide them. Searching through this many alternatives is similar to what would be required to find the well-known solution to the MCB problem.

- Representation reformulation is under the equivalence constraint that when the reformulation is completed, we are solving a problem equivalent to the original one. This constraint carries a heavy mathematical or logical burden. But in reality, people do not solve equivalent problems if the original problem turns out to be too difficult. Therefore, representation reformulation does not address the question of how people formulate and reformulate problems when the equivalence constraint is violated.

- Identifying irrelevance plays an import role in representation reformulation because irrelevance justifies homomorphic reformulations. Many irrelevance analysis techniques developed for representation reformulation, however, do not specify when they are cost-effective thus may reduce the overall problem solving efficiency when applied blindly. For instance, identifying irrelevant assertions is not cost-effective for backward reasoning (see Chapter 4).

We believe we can offer a solution to those problems by extending our access from representation reformulation upward to problem solving paradigm selection.

With the ability to switch to different problem solving paradigms, we can now access promising representations not suitable for the original problem solving paradigm. Because the preferred representations for different problem solving paradigms are often quite different, PSP shifts may bring dramatic representation changes instead of migrations. For example, we may diagnose a circuit by empirical association or by model-based reasoning. Each of these PSPs has its own preferred representation, and it is difficult to reformulate from one to another [Davis, 1989].

By making PSP selection a problem solving stage before representation design, we may sometimes acquire and represent the knowledge relevant to the selected PSP from scratch instead of reformulating the knowledge from a representation designed for other PSPs. This will partially overcome the equivalence constraint mentioned above because we are now not limited to the knowledge contained in some initial representation.

By making the PSP explicit, we may gain guidance associated with the PSP in searching for a suitable representation. For instance, the PSP $\underline{B}$ for the T&J problem may suggest, among other reformulations, partition the hexagons so that each move affects exactly one hexagon from each partition, which eventually leads to a solution. By following the guidance associated with the PSP, we may avoid unpromising or irrelevant reformulations.

By making the PSP explicit, we may also have a better understanding about the relevance of knowledge. Different PSPs respond differently to irrelevant knowledge, thus knowing the selected PSP can help us to identify cost-effective relevance analysis.

In summary, by allowing problem solving paradigm change, we may gain access to dramatically different representations, which may be difficult to obtain by conventional representation reformulation. By making the PSP explicit, we may obtain guidance for the search of a better representation. By making PSP explicit, we may also have a better understanding about what kind of knowledge are relevant and when it is cost-effective to know it.

Although most work on representation reformulation has emphasized representation changes and has failed to make explicit the PSP changes, PSP changes do sometimes accompany representation shifts. For instance, [Van Baalen and Davis, 1988] presents a representation change that implies a PSP change, from theorem-proving to congruent-closure-computing for equality inference.

Analogously, problem perception changes sometimes accompany PSP changes. For instance, the PSP change from 'finding a solution' to 'proving the non-existence of a solution,' as in the MCB problem, is also a problem perception change.

Of course, whether there is a problem perception change depends on the abstraction level of the problem. For instance, if the original MCB problem was defined as "find if we can cover the mutilated checker board exactly with dominoes," one could argue that there is no problem perception change from PSP $\underline{A}$ to PSP $\underline{B}$.

Generally speaking, problem perception is still out of reach without human assistance. We believe, however, that the PSP shift will help us to understand problem perception. Computer-assisted PSP changes may provide some aid for people in problem perception by suggesting different problem perceptions associated with PSPs.

Therefore, including PSP selection in automated problem solving pushes the automation frontier of problem solving toward the limiting point of problem perception and presents us with new research challenges in automated problem solving.

The Ubiquity of Problem Solving Paradigm Selection

In reality, we sometimes collect facts about a problem before deciding on a PSP, especially when we do not know how to solve the problem. For example, we may record faces in some format even though we still do not know how to recognize them, which means we sometimes have a representation before selecting a PSP. This phenomenon raises two questions:

- Will PSP selection have an impact on a representation when the representation is selected first?

  We believe the answer is yes. There may be several applicable PSPs for the representation. Therefore, we still need to select a PSP. The selected PSP will dictate the direction in which the representation can be made more efficient and may demand additional information.

  If none of the applicable PSPs work for the problem, the representation will be forced to change so that more PSPs will be applicable.

- Will the representation have an impact on the PSP selection?

  The answer is yes. Fewer PSPs will be valid candidates since not every PSP works for every representation.

In summary, even if a representation is selected first, we may need to select a PSP for it. Therefore, we cannot substitute PSP selection with representation selection.

- Classification PSP

  Attribute-Based Classification

- Constraint Satisfaction PSPs

  Brelaz K-Coloring Algorithm, Min-Conflict Heuristic, Constraint Satisfaction on Finite Domain, Constraint Satisfaction on Infinite Domain

- Diagnosis PSPs

  Fault-Dictionary-Based Diagnosis, Empirical-Association-Based Diagnosis, Model-Based Diagnosis,

- General PSPs

  Disproof By Partition and Unsolvable Equation, Generate and Test, Order Reasoning, Problem Reduction, Algorithmic, Search

Figure 1.5: Currently Encoded PSPs in the PSP Selection System

1.1.3 A Problem Solving Paradigm Selection System

Considerations in Designing the PSP Selection System

In designing a PSP selection system, we must answer the following questions:

- What is the role of the system?

  Our PSP selection system is intended as the first automated stage in problem solving. It should recommend PSP as well as preferred representations for a given problem. These recommendations might be passed to a representation design system.

- What kinds of PSPs will be known to the system?

  We intend to give the system only the PSPs that have been used in some programs. These PSPs are relatively well understood and are relatively easy to implement, thus are more practical for automated problem solving. Figure 1.5 lists all the PSPs currently encoded in our PSP selection system. Appendix B has a transcript of the knowledge about these PSPs in the system.

- What kind of input does the system take?

  The system interacts with the user, asks questions about the given problem and takes the answers as the input. We avoid using formal problem specifications for several reasons: (1) The user may not know how to write a formal specification. (2) A formal specification may not contain all the information for all applicable PSPs. (3) We would like to focus on the important characteristics that make a PSP suitable for a problem instead of extracting such characteristics from a formal specification.

Sample Interactions with the Problem Solving Paradigm Selection System

Figure 1.6 shows sample interactions during a session with our experiment PSP selection system. In this session, the user (the author) had a digital circuit diagnosis problem in mind. The system asks the user roughly a dozen questions and then makes PSP recommendations. The first few questions are intended to determine the type of the problem. For instance, the first question in Figure 1.6 determines whether the problem is a diagnosis problem. Once the problem is believed to be a diagnosis problem, the next few questions determine which diagnosis paradigms are applicable.

SAMPLE INTERACTIONS:

...
Is the problem to determine why a given object breaks? [Yes/No]
Yes
...
```
Do we know the faulty behavior models of the components?  [Yes/No]
```
Yes
```
What kind of failures do we need to consider?  [Break Components/Abnormal component interactions]
```
Break Components
```
Is the broken subject's behavior repeatable?  [Yes/No]
```
Yes
...
[Typically a dozen or so questions per PSP.]

SAMPLE RECOMMENDATIONS:
```
The problem may be solved by MODEL-BASED-DIAGNOSIS.  There are 3 components to
MODEL-BASED-DIAGNOSIS.  The following are plausible implementations for these
components:

  For CANDIDATE-GENERATION, we have the following plausible alternatives:

Generate candidate by tracing causal path.
LOCALIZE used a similar technique.

  For CANDIDATE-VERIFICATION, we have the following plausible alternatives:

Verify candidates by comparing simulation with faulty components and observations.
SOPHIE IDS used a similar technique.

  For CANDIDATE-DISCRIMINATION, we have the following plausible alternatives:

Discriminate candidates by taking observations affected by nearly half of the candidates.
INTER used a similar technique.
```

Figure 1.6: Sample Interactions with the PSP Selection Program for a Diagnosis Problem

For instance, the second question in the figure determines if a fault-dictionary based diagnosis paradigm is applicable. Finally, there are a few questions to determine the details of applicable PSPs. For instance, the fourth question determines whether the fault is stable so that we may test the subject in case there is more than one plausible diagnosis.

After the questioning, the PSP selection system makes its recommendations. The PSP selection system will make recommendations for each component of the applicable PSPs, as shown in the recommendation part of Figure 1.6.

1.2 Problem Solving Paradigm Comparison

As soon as we have the luxury of selecting PSPs, we must compare them. Complexity analysis is a rich source of heuristics for PSP comparison. However, we must overcome certain problems with some complexity analysis results before we can turn them into practical heuristics:

- Many complexity analysis methods are too idealized to be applied to real problems. For instance, forward and backward branching factors are important characteristics of forward and backward search for many complexity analysis results. However, we often have to improvise to estimate the two factors for a real problem.
- Some complexity analysis methods are too expensive. For instance, complexity analysis methods that must know search depth are often expensive since one often must solve a problem to obtain the search depth.
- Many complexity analysis methods are developed on small problems and thus may miss characteristics important for large problems. For instance, complexity analysis methods for search often use only the branching factor and search depth. Yet for practical problems, the number of initial facts could be as important to the search space as the branching factor.

We believe that PSP selection can fuel the development of better heuristics, heuristics that we know how to apply to real problems, that are inexpensive as compared to the cost of solving the problem, and that reflect the problem more accurately.

In this thesis, we are going to look closely at the complexities of forward and backward reasoning. We will consider factors not reported in AI literature and we will present an extension tree method for comparing the search spaces of forward and backward reasoning. The extension tree method obtains the estimate through an abstract simulation and then calibrates the result with a small portion of the real reasoning process. We hope the ideas about abstract simulation and calibration used in the extension tree method can help to make practical use of other idealized complexity analyses.

1.3 Relevance Analysis

Reducing irrelevant inference may increase problem solving efficiency and change the comparisons between competing PSPs. There are at least two things one can do to make relevance analysis more pragmatic:

- Study when certain relevance analysis techniques are cost effective.

  We believe that the problem solving paradigm is a major factor in determining what kinds of relevance analysis are cost effective. For instance, assume we have a inference problem defined by an IF-THEN type of inference rules and facts. For forward reasoning, it would be useful to know whether a fact is relevant since inference on irrelevant facts will be wasted. For backward reasoning, on the other hand, the relevance of a fact is not as useful since backward reasoning will naturally 'avoid' irrelevant facts.

Therefore, we propose that each relevance analysis technique should specify the kinds of PSPs for which it is cost effective, and each PSP should be associated with a set of relevance analysis techniques that are cost effective for it. This association between relevance analysis techniques and PSPs has been mostly missing in the AI literature. This topic is left as future research and is not covered by this thesis except that we will present a relevance analysis technique.

- Develop cost effective relevance analysis techniques.

  It is our view that the emphasis of relevance analysis research has long been on generally applicable relevance analysis techniques. However, due to the cost associated with such analysis techniques, we have not seen much of their applications on real problems. We believe that cost effective relevance analysis techniques are mostly PSP specific. Therefore, we should start to build a large collection of 'quick and dirty' relevance analysis methods for real world problems.

  This thesis will present one such technique, an object relevance analysis technique. This technique is cost effective for forward chaining inference. When combined with Van Baalen's connection graph analysis technique, they can rule out more irrelevant facts than the sum of the irrelevant facts they can discover individually.

1.4 Implementation Status

The PSP selection system is built on top of an AMORD-like rule-based system [de Kleer *et al.*, 1977]. Currently, the experiment system has the knowledge of 14 PSPs for 3 specific and 6 general distinct problem perceptions.

The system's knowledge base takes about 100K of disk space. Appendix B contains a mechanical translation of the knowledge base. Appendix C contains traces of some interaction sessions with the program.

The extension-tree analysis program is implemented. It produced the results presented in Chapter 3.

The object-relevance and its combination with the connection graph analysis techniques are also implemented. The implementation produced the results presented in Chapter 4.

1.5 Thesis Outline

Problem solving process is like a giant jigsaw puzzle, with many pieces yet to be identified, especially at the problem perception, problem solving paradigm selection, and representation design stages. This thesis identifies three of these pieces: PSP selection, PSP comparison, and PSP specific relevance analysis.

The main body of the thesis has three chapters: Chapter 2 will give a detailed account of the PSP selection expert system and its underlying theories. Chapter 3 will present a theoretical comparison of forward/backward reasoning and breadth-first/depth-first/random search. It will also present the extension-tree method for comparing forward and backward reasoning and adapt it to compare sequential and parallel processing of conjunctive subgoals. Chapter 4 will describe the object-relevance analysis and the connection graph analysis techniques. Chapter 5 comments on related work and speculates on future work. The Appendices contain descriptions of the abstract problem solving paradigms in the PSP selection expert system, the transcript of the knowledge in the system, and sample traces.

Chapter 2

Problem Solving Paradigm Selection

Knowledge-based systems have been built for a wide variety of application domains, but few areas of AI itself have been served in this manner. This chapter will describe an expert system whose expertise is selecting an appropriate problem solving paradigm (PSP) for a task.

We believe PSP selection has a major impact on problem solving but has not been adequately addressed to date in AI research. Most problem reformulation methods are targeted to particular PSPs, thus leaving the PSP selection issue virtually untouched. On the other hand, AI research has been accumulating PSPs that are usually accompanied by informal statements as to the kind of problems for which they are suitable.

In Chapter 1, we have discussed the role of PSP selection in problem solving.

In this chapter, we discuss PSP selection as part of problem solving, we describe a representation for PSPs, and explore the design of a knowledge based system for selecting PSPs. We believe PSP selection systems will encourage PSP designers to be more precise in defining the kind of problems for which their PSPs are designed and will assist others in selecting appropriate paradigms.

2.1 A Theory of Problem Solving Paradigm Selection

This section will present a representation for PSP that supports PSP selection (§2.1.1 and §2.1.6) and discuss the PSP space in terms of this PSP representation (§2.1.2), PSP shift of different kinds (§2.1.3), match between PSPs and problems (§2.1.4), and interactions between PSP selection and other problem solving stages (§2.1.5).

2.1.1 Abridged Representation for Problem Solving Paradigms

Here we describe an abridged PSP representation for the purpose of PSP space analysis. We will describe an extended PSP representation later.

We consider that a PSP is composed of one or more *components*. For example, each model-based diagnosis PSP has three components: candidate generation, candidate verification, and candidate discrimination. A *cluster* of PSPs is defined as a collection of PSPs that share the same component structure. For example, all the

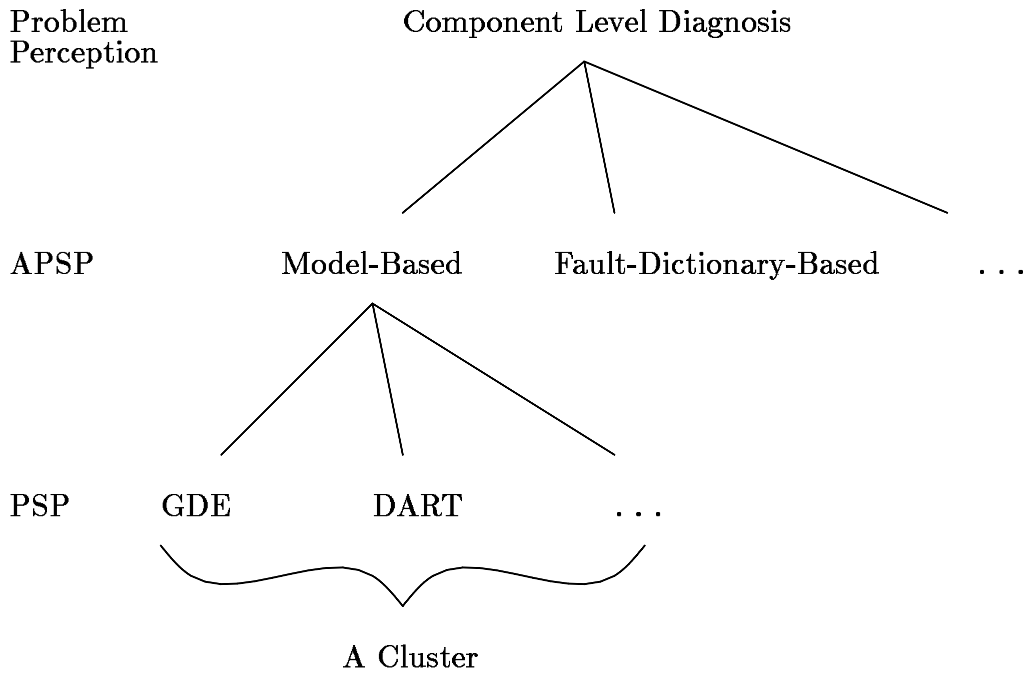

Figure 2.1: Problem Perception, APSP, and PSP

model-based diagnosis PSPs, such as DART and GDE (see [Davis and Hamscher, 1988]), together form a cluster of PSPs (see Figure 2.1). The shared component structure of each cluster defines an *abstract PSP* (APSP). For example, model-based diagnosis defines such an APSP. The components of an APSP correspond to the top level decomposition of a problem. One or several APSPs that we believe can be alternatives to some problem define a *problem perception*. For example, we may define `diagnosis` as a problem perception. We define problem perceptions in terms of APSPs because APSPs form a better defined space than do the problems that we may wish to solve with these APSPs.

We represent component structures for APSPs because we want to represent the structure of APSPs as people understand them. We represent only the top level decomposition because we want to keep the representation for each APSP reasonably simple, in order to emphasize the unique way in which each APSP decomposes a problem. If a subtask represents substantial problem solving activities, it can be hierarchically defined as an APSP.

Each PSP component may have one or more *attributes*. For instance, the components of the model-based diagnosis APSP have one attribute each: `method`, while the control component of search APSP in our implemented PSP selection system has four attributes: `direction`, `bookkeeping`, `ordering`, and `backtracking`. Each attribute ranges over one or more *values*. An assignment to all attributes of a PSP component defines an *implementation* for the PSP component. A combination of component implementations defines a PSP, i.e., an instance of the corresponding APSP. For example, a model-based diagnosis PSP may choose `tracing causal path` as the value for the `method` attribute of the component `candidate generation`, `constraint suspension` as the value of the `method` attribute of the component `candidate verification`, and `testing` for the `method` attribute of the component `candidate discrimination`. The alternative attributes values can be found in Appendix B.

2.1.2 Problem Solving Paradigm Space

Here we are going to discuss the relationship between different PSPs in the PSP space in preparation to discuss PSP changes of different kinds, which will help to define the scope of our PSP selection system.

We consider that the PSP space consists of clusters of PSPs.

As noted above, clusters are PSPs with the same components. For example, all model-based diagnosis methods with different candidate generation strategies are members of the same cluster. Each cluster is identified by an APSP (see Figure 2.1).

Clusters may themselves be grouped together, into what we call problem perceptions. For instance, fault-dictionary-based diagnosis and model-based diagnosis can both locate malfunctioning components of an artifact; thus we may say these two, together with some other APSPs, define a *component level diagnosis* problem perception. Naturally, an APSP may be useful for different problem perceptions. For instance, the fault-dictionary-based diagnosis APSP can be used to diagnose unintended interactions between components.

APSPs that do not share the same problem perception may be either independent or related by a *specificity* relation. An APSP is said to be more *specific* than another if any problem perception solvable by the former is also solvable by the latter. For example, the model-based diagnosis APSP is more specific than the search APSP because a model-based diagnosis problem is solvable by search. We believe the PSP space contains a handful of general APSPs, corresponding to what are usually called weak methods. The rest of the PSP space is populated by mostly unrelated APSPs corresponding to what are usually called strong methods.

Problem representations of different emphases may be associated with PSPs at different abstraction levels. In the description above, the most abstract PSPs are problem perceptions. Each perceived problem may conceptually have an initial representation uncommitted to any APSPs. APSPs are the next most abstract PSPs. Each APSP may associate with a practical initial representation of a problem, containing suitably structured information pertinent to the APSP. APSP instances, or simply PSPs, are the least abstract PSPs. Each APSP instance may associate with heuristics for trimming the initial representation of the problem.

2.1.3 Choices of Problem Solving Paradigms

Here we will discuss PSP changes of different kinds based on the PSP space structure described above. We will discuss what is involved in making each kind of PSP shift and the feasibility of automating it. This will help to define the scoop of our PSP selection system at the end of this section.

Changes between members of the same PSP cluster do not change the problem perception nor the top level decomposition of the problem. To change between members of the same PSP cluster is to select different attribute values for the PSP components, hence it changes the way the subproblems are solved. PSP selection of this nature compares the desirabilities of different attribute value assignments, which depend on relevant problem characteristics. We believe that making this kind of PSP changes without human intervention is a suitable near term goal because the PSPs involved share initial problem representations.

Changes between APSPs that share the same problem perception bring no change to the problem perception, but they change the top level decomposition of the problem. For instance, in our system fault dictionary-based diagnosis has no components, while model-based diagnosis has three. We believe changing APSPs without changing problem perceptions can be done without human intervention in some cases if we can provide an initial representation for the problem perception which carries enough information for all the APSPs involved.

Changes between APSPs that do not share problem perceptions, by definition, changes the problem perception. Switching from a specific APSP to a general APSP usually is trivial and undesirable. For instance, a model-based diagnosis problem can always be considered as a search problem. On the other hand, switching from a general APSP to a specific one is usually desirable but difficult to automate. When and how to switch to a specific APSP is still the topic of some ongoing research. Switching between APSPs not related by the specificity relation usually requires human assistance because it may require information not available to a system, such as the ultimate purpose for solving the problem.

Our PSP selection system is aimed at the coverage instead of the depth in the knowledge involved. Thus it needs to cover the kinds of PSP changes that require problem perception changes. This requires the involvement of the user and relinquishment of an initial formal problem specification.

2.1.4 Match Between Problem Solving Paradigms and Problems

Here we are ready to discuss the action part of PSP selection—matching PSPs to problems.

Problems may be characterized as follows: Each problem may have some requirements that an applicable PSP must satisfy (e.g., a problem may require optimal solutions). Each problem may have some characteristics that an applicable PSP can utilize (e.g., a search problem may have a small forward branching factor).

On the other hand, each PSP and each value of component attributes may require certain problem characteristics to be applicable (e.g., a fault must be non-intermittent if we want to discriminate candidate components by probing). Some problem characteristics may make some PSPs or attribute values more favorable (e.g., a fast probing speed makes probing more favorable for candidate discrimination).

Once problem requirements, problem characteristics, PSP/attribute value requirements, and favored problem characteristics are known, matching between problem and PSP is straightforward. A workable PSP must meet all the problem requirements, and the problem must meet all the PSP/attribute value requirements. A workable PSP will be favorable if the problem has the characteristics favored by the PSP.

2.1.5 Interactions with Other Problem Solving Stages

Since PSP selection is only one stage of problem solving and we need to keep it in perspective, here we will discuss its possible interactions with other problem solving stages. This will also help understand certain considerations of the current PSP selection system.

PSP selection interacts directly with the problem perception stage and representation design stage. The selected PSP dictates the solution finding stage but has almost nothing to do with the solution execution stage.

Perception The perception stage produces the problem which all the remaining stages work on. PSP selection may be closely involved in problem perception by telling the perception stage whether there is a PSP for a perceived problem or how efficient available PSPs are. A PSP selection system may also assist problem perception by suggesting the problem perceptions associated with known APSPs.

Since problem perception depends on human, it is natural for a PSP selection system to interact with human users because several reasons. Firstly, some PSP shift may require problem perception change, which is still beyond automation. Secondly, non-interactive PSP selection requires a formal specification of the problem. All formal specifications will contain only selected information about the problem and thus may limit the choices of PSPs even when problem perception change is not needed. Finally, certain information may be difficult to extract from formal specifications, such as the search depth of a search problem, but it may be easy to obtain from a human who knows more about the problem than the specification.

For an interactive system, there is a basic question about the user: do we presume a naive or sophisticated user? We opt for a sophisticated user. The user shoulders an important part of problem solving. We need the user's intelligence to perceive the problem.

PSP Selection From the PSP selection perspective, there are two kinds of problems: the kind of problems for which we know what PSP to use after reading an AI text book, and the kind of problems for which we do not. For the first kind of problems we may select a PSP before we have a formal representation of the problem. The second kind of problems may require further research to find a PSP. We may have to collect information about the problem, i.e., work out a formal representation of some form, before a PSP can be found.

The PSP selection system reported here currently contains textbook knowledge about PSPs and thus is suitable for the first kind of problems.

Representation The representation design stage receives the selected PSP and the problem, produces an initial representation preferred by the PSP for the problem, then optimizes the representation. The PSP determines the relevance of information. There may be heuristics associated with the PSPs about how to reorganize the information for efficiency.

The representation design stage and later problem solving stages may complain that the given problem is costly to solve with the selected PSP, prompting the PSP selection and problem perception stages to reconsider the problem.

Solution and Execution Finding a solution and executing it is removed from the PSP selection stage. However, difficulties in these two stages may potentially cause the problem solver to backtrack at the PSP stage.

2.1.6 Extended Representation for Problem Solving Paradigms

While the PSP component list describes an important aspect of PSPs, there are three other aspects of PSPs that are equally important for problem solving.

Applicability Conditions We must know when we can use a PSP. We must know when to use certain attribute values for the PSP components.

Preferred Representation We must know how to specify the problem for a PSP.

Procedure We must know how the components of a PSP work together.

Figure 2.2 shows the Model-Based Diagnosis PSP. For simplicity, conditions for the component implementations are omitted from the Applicability Conditions entry, and the component attributes are omitted from the Components entry. The omitted details can be found in the Appendix B.

Brief Description: Diagnosis by comparing observations and predictions based on a structure and behavior model of the artifact, with components as the diagnosis granule.

Applicability Conditions: Model-based diagnosis is suitable for diagnosing artifacts for which (1) we want to locate malfunctioning components, (2) we know the behavior of and the interactions among the components, (3) we usually can infer inputs from outputs of the components as well as infer outputs from inputs, and (4) we do not know the faulty behavior of the components or there are too many kinds of faulty behavior.

Preferred Representation: For using model-based diagnosis we must use a structure and behavior model for the artifact being diagnosed. That is, we need to represent the components of the artifact, the behavior of these components, and the interactions among the components. We may recursively use a structure and behavior model for some of the components.

Components: Candidate generation, candidate verification, candidate discrimination.

Procedure:

- Generate candidates by
  - Simulating the artifact according to the structure and behavior model and the given inputs, assuming all components are normal;
  - Running the artifact 'backward,' i.e., from the model and the given outputs (observations) to infer the inputs, assuming all components are normal; and
  - For each connection of the components in the artifact's model on which simulation results from the two simulations are inconsistent, identifying the components involved in getting the inconsistent values as candidates.
- Verify candidates by eliminating those that are not consistent with all observations.
- Discriminate candidates by
  - Computing the most informative points to probe inside the artifact, carrying out the probe for additional observations while keeping the inputs unchanged, and eliminating the candidates inconsistent with the new observations; or
  - Computing the most informative new inputs, applying the new inputs and making observations, and eliminating the candidates that are inconsistent with the new observations.

Figure 2.2: The Model-Based Diagnosis Paradigm

2.2 A Pilot Problem Solving Paradigm Selection System

We have built an interactive knowledge-based PSP selection system based on the theory described in the previous section. A user may come to the system with a problem in mind, answer some questions asked by the system, and then receive recommendations about what kinds of PSPs may help solve the problem. Figure 2.3

shows segments of the interaction between the system and the user for a diagnosis problem.

We involve the user in the selection process instead of using a formal problem specification because (1) we want to focus on a vocabulary for describing PSPs and on specifying when a PSP is a fit for a given problem, instead of focusing on extracting relevant information from some formal problem specifications, and (2) we want the system to act like a human consultant for users who may not be able to write formal problem specifications for their tasks.

2.2.1 The Process

In the PSP selection process, the system first asks the user questions to identify applicable problem perceptions. It then asks more detailed questions for the relevant APSPs to determine the implementations of the APSP components. We assume the user will then represent his problem for the recommended PSP, implement the PSP and solve the problem.

Figure 2.3 shows sample questions and recommendations generated by the system. The questions and recommendations are canned text. They are based on our study about the key factors in matching problems with PSPs.

SAMPLE INTERACTIONS:
```
Do we know the faulty behavior models of the components?  [Yes/No]
```
Yes
```
What kind of failures do we need to consider?  [Break Components/Abnormal component interactions]
```
Break Components
```
Is the broken subject's behavior repeatable?  [Yes/No]
```
Yes

...

[Typically a dozen or so questions per APSP.]

SAMPLE RECOMMENDATIONS:
```
The problem may be solved by MODEL-BASED-DIAGNOSIS.  There are 3 components to
MODEL-BASED-DIAGNOSIS.  The following are plausible implementations for these
components:

  For CANDIDATE-GENERATION, we have the following plausible alternatives:

Generate candidate by tracing causal path.
(The program: LOCALIZE used a similar technique.)

  For CANDIDATE-VERIFICATION, we have the following plausible alternatives:

Verify candidates by comparing simulation with faulty components and observations.
(The programs: SOPHIE IDS used a similar technique.)

  For CANDIDATE-DISCRIMINATION, we have the following plausible alternatives:

Discriminate candidates by taking observations affected by nearly half of the candidates.
(The program: INTER used a similar technique.)
```

Figure 2.3: Sample Interactions for a Model-Based Diagnosis Problem

2.2.2 The Vocabulary

We choose to represent problem solving paradigms at a level similar to or slightly more general than [Chandrasekaran, 1986]'s task level. At one end of the spectrum of PSPs, we can have a problem solving paradigm highly specific for a particular task and rich in knowledge. For example, we consider the 'Constraint Directed Scheduling' in Appendix A.2 is such a PSP. Toward the other end, as we move toward the other end, the PSPs become more general and relatively poorer in knowledge. For instance, we believe the 'Case Based Reasoning For Solving Problems' in Appendix A.2 is such a PSP. We believe the level around the task level strikes a balance between the amount of the PSP knowledge and the generality of the system for the purpose of PSP selection.

In selecting the vocabulary for our PSP selection system, we tried to follow the guideline that the vocabulary should be problem oriented, e.g., for diagnosis, "finding next candidate diagnosis to verify" is considered to be at the "task level," while "finding next rule to fire" is implementation oriented.

2.2.3 The Knowledge

The knowledge about a PSP pertinent to PSP selection includes the conditions required for the PSP to be applicable, and the conditions required for each attribute value of the components to be applicable. Other knowledge about a PSP is canned text, intended as input to representation design stage. The following is a summary of the PSP representation encoded in our experiment system:

Brief Description: There is a short text associated with each APSP known to the system in order to give people a rough idea about each APSP. For instance, the text for model based diagnosis is `MODEL-BASED-DIAGNOSIS is to Diagnose by comparing observations and predictions based on a structure and behavior model of the artifact, with components as the diagnosis granule.`

Applicability Conditions: There are sets of conditions for recommending each APSP. There are conditions for each attribute value to be applicable in addition to the APSP conditions. For instance, one set of the conditions for model based diagnosis is

```
TYPE of FAILURE is COMPONENTIAL
TYPE of OBJECT is STRUCTURED
TYPE of COMPONENT-BEHAVIOR is KNOWN
TYPE of COMPONENT-FAULTY-BEHAVIOR is not KNOWN
```

Preferred Representation: There is a short text associated with each component about the preferred representation. We associate representation with the components because the components of an APSP may not share the same representation. The preferred representation for the candidate generation component of model based diagnosis is `Structure and behavior model of the artifact`. [1]

Components: There is a list of components for each APSP and a list of attributes for each component. For instance, the list for model based diagnosis is `CANDIDATE-GENERATION, CANDIDATE-VERIFICATION, and CANDIDATE-DISCRIMINATION`.

Procedure: There is a short text associated with each component attribute value explaining the problem solving process. For instance, the text for the `UP-STREAM` value of the `METHOD` attribute of the `CANDIDATE-GENERATION` component of model based diagnosis is `"Generate candidate by tracing causal path."`

Note that in the current knowledge base, there is no information about how the components of a PSP work together–which is left as future work.

A. Applicability Conditions

```
MODEL-BASED-DIAGNOSIS is to Diagnose by comparing observations and predictions based on a structure
and behavior model of the artifact, with components as the diagnosis granule
 which is applicable if

  TYPE of FAILURE is COMPONENTIAL
  TYPE of OBJECT is STRUCTURED
  TYPE of COMPONENT-BEHAVIOR is KNOWN
  TYPE of COMPONENT-FAULTY-BEHAVIOR is not KNOWN

OR
  [other sets of conditions]
```

B. Components

```
Paradigm MODEL-BASED-DIAGNOSIS has 3 components:
CANDIDATE-GENERATION, CANDIDATE-VERIFICATION, and CANDIDATE-DISCRIMINATION
```

C. Component attributes

```
Component CANDIDATE-GENERATION of MODEL-BASED-DIAGNOSIS has 1 attribute: METHOD
Its preferred representation is: Structure and behavior model of the artifact

[other components]
```

D. Attribute Value Conditions

```
Consider UP-STREAM METHOD for MODEL-BASED-DIAGNOSIS CANDIDATE-GENERATION
 i.e., "Generate candidate by tracing causal path." if
  TYPE of COMPONENT-BEHAVIOR is DIRECTIONAL

    The the following will make UP-STREAM more favorable:

  TYPE of COMPONENT-BEHAVIOR is SIMPLE

[other attribute values]
```

Figure 2.4: The Encoded Model-Based Diagnosis Paradigm

Figure 2.4 shows the encoded version of the model-based diagnosis paradigm in the PSP selection system used as examples above. The `favorable` conditions in part D will be explained later when we discuss the matching between a problem and with a PSP. Appendix A.2 shows some classic APSPs, including means ends analysis, constraint directed scheduling, and case based reasoning. Appendix A.1 shows all the

[1] In Figure 2.2, the preferred representation is independent of the components for the sake of presentation simplicity.

APSPs currently known to our PSP selection system while Appendix B shows the encoded version of these APSPs.

The problem conditions for selecting PSPs are verified with the user by asking the user questions. The conditions are collected and turned into questions for the user in the following way:

- If we think a condition can be verified with one question, we will turn the condition directly into a question. For instance, the "TYPE of OBJECT is STRUCTURED" condition, which is the applicable condition of the model-based diagnosis, is turned into the question "Do we know the pathways by which the components affect each other?"

- If we think a condition can be verified with more than one question, we may turn the condition into several questions, introducing intermediate conditions as necessary. For instance, the "TYPE of COMPONENT-FAULTY-BEHAVIOR is not KNOWN" condition is turned in to two questions: "What kind of failures do we need to consider?" and "Do we know the faulty behavior models of the components?" An intermediate condition, the "TYPE of FAILURE is COMPONENTIAL," is introduced when the answer to the first question is "Break components."

- Judgment conditions are turned into questions about facts and thresholds are introduced to convert the facts into judgment. For instance, the condition the "COUNT-CLASS of COMPONENT-FAULT is LARGE" is turned into the question "How many distinctive faults can the object have?" The threshold here is one billion.

- Sometimes several conditions are turned into one multiple choice question. For instance, the question "What kind of failures do we need to consider?" has two answer choices: "Break components" and "Abnormal component interactions."

The questions are arranged in a tree-like order. Usually, general questions are asked before detailed questions. For instance, the question "Is the problem to determine why a given object breaks?" is asked before any other questions about a diagnosis problem.

If a question is relevant only when we get a certain answer to another question, the first question will be asked after obtaining that answer. For instance, the question "Do we know the pathways by which the components affect each other?" is asked only when we know that the object has distinct components. Appendix B contains all the conditions and questions in our PSP selection system.

2.2.4 The Match

A PSP is considered a match for a problem if it meets all the problem requirements, and the problem must meet all the PSP/attribute value requirements.

In our implementation using a rule based system, the necessary conditions for an attribute value come from two sources:

- The necessary conditions for that attribute value (e.g., generating candidates by tracing causal path requires that the components be directional); and

- The negated problem requirements the attribute value cannot satisfy (e.g., beam search does not guarantee optimal solutions, thus the problem must not require optimal solutions).

The first source concerns only the attribute value in question. The second source concerns both the attribute value and all the problems that might use the attribute value. Examining the requirements of all problems in general is impossible. However, for selecting a PSP, we need to examine only some typical problems that match the PSP and examine only the requirements related to PSP selection. The requirements related to PSP selection are usually requirements about the optimality of the solution or the cost of finding a solution. We believe that there are only a few such requirements.

We use favorable conditions to select attribute values. A workable PSP will be more favorable if the problem has the characteristics that make the selected attribute value favorable. Since a PSP is defined by the attribute values of corresponding APSP components, we associate favorable conditions with attribute values only.

The favorable conditions are used for PSP evaluation. This kind of evaluation is crude. However, we believe favorable conditions can encode effective heuristics for PSP selection.

The favorable conditions also come from two sources:

- The problem characteristics that make the attribute value favorable (e.g., small forward branching factors makes search forward favorable), and

- The characteristics of the attribute value that might meet the requirements for some problems but will never disqualify the attribute value for any problem (e.g., exhaustive search guarantees optimal solutions and thus is more favorable for problems that require optimal solutions than those that do not, and optimal solutions are acceptable for any problem).

In Figure 2.4, part D., the necessary condition 'directional component' (for the value UP-STREAM of the METHOD attribute of the CANDIDATE-GENERATION component of the MODEL-BASED-DIAGNOSIS PSP) comes from the attribute value itself, i.e., a condition required by the attribute value instead of an unsatisfied problem requirement. The favorable condition, simple component, also comes from the attribute value itself.

2.2.5 Current Status of the PSP Selection System

The PSP selection system is built on top of an AMORD-like rule-based system [de Kleer *et al.*, 1977]. Currently, the experiment system has the knowledge of 14 APSPs for 3 specific and 6 general distinct problem perceptions, see Figure 1.5.

Problem	APSP	Component	Attribute	Value
Classification	1	1	2	2x2
Constraint Satisf.	4	1,1,1,1	1,1,6,1	1,1,2x3x3x5x2x3,2
Diagnosis	3	1,1,3	1,2,1x1x1	1,2x2,4x5x5
General	6	2,2,1,2,1,2	1x1,1x1,1,2x4,1,1x4	1x1,3x2,2,1:1x3:2:12:2,1,2x3:2:12:2

Table 2.1: Knowledge Base Summary

The model-based diagnosis APSP used extensively in this chapter is one of the three APSPs for diagnosis problems: model-based (three components, one attribute each, of 4, 5, and 5 values, respectively, representing about 100 variations of model-based diagnosis), empirical association (one component, two attributes, 2 values each), and fault dictionary (one component, one attribute, one value). Currently there are 21 questions, and 19 attribute values for APSP component implementations. (Not all 21 questions are asked for every problem, nor are all 19 attribute values made for every problem.)

The system's knowledge base takes about 100K disk space. The diagnosis part is about 35K. Table 2.1 lists the numbers of APSPs, components, attributes, and values. The numbers of components are delimited with comas for different APSPs, the numbers of attributes are delimited with $\times$ for different components, and the numbers of values are delimited with colons. For instance, the last general APSP (search) has two components, the two components have 1 and 4 attributes respectively, the attribute of the first component has 2 values and the attributes of the second component have 3, 2, 12, and 2 values respectively. Except for two of the general APSPs that share one component, APSPs do not share components or attributes in the experiment system. All together, the system may come up with 1095 different PSPs.

Appendix B contains a mechanical translation of the knowledge base.

Chapter 3

Problem Solving Paradigm Comparison: A Detailed Case Study

As soon as we have the luxury of selecting PSPs, we must compare them. In the previous chapter, we suggested that heuristics need not be elaborate to be effective for comparing PSPs. We used this idea in our PSP selection program and encoded some simple heuristics, mainly matches between non-critical problem and PSP characteristics, for comparing competing PSPs.

But sometimes, mostly when comparing at the PSP level in Figure 2.1, we need quantitative methods. Complexity analysis is a rich source of heuristics for PSP comparison. However, we must overcome certain problems with some complexity analysis results before we can turn them into practical heuristics:

- Many complexity analysis methods are too idealized to be applied to real problems. For instance, forward and backward branching factors are important characteristics of forward and backward search for many complexity analysis results. However, we often have to improvise to estimate the factors for a real problem.

- Some complexity analysis methods are too expensive. For instance, complexity analysis methods that must know search depth are often expensive since one often must solve a problem to obtain the search depth.

- Many complexity analysis methods are developed on small problems and thus may miss characteristics important for large problems. For instance, complexity analysis methods for search often use only the branching factor and search depth. Yet for practical problems, the number of initial facts could be as important to the search space as the branching factor.

We believe that PSP selection can fuel the development of better heuristics, heuristics that we know how to apply to real problems, that are inexpensive as compared to the cost of solving the problem, and that reflect the problem more accurately.

In this chapter, we look closely at the complexities of forward and backward reasoning to illustrate the development of practical PSP comparison heuristics. We will consider factors not reported in AI literature. We will present an extension tree method for comparing the search spaces of forward and backward reasoning. The extension tree method obtains the estimate through an abstract simulation and then calibrates the result with a small portion of the real reasoning process. We believe the ideas about abstract simulation and calibration used in the extension tree method can help to make practical use of other idealized complexity analyses.

3.1 Theoretical Comparison of Some Search Strategies

3.1.1 Forward and Backward Search

Traditional search space comparison of forward and backward reasoning uses the branching factor as the criteria [Winston, 1992; Rich and Knight, 1991]. We believe that the search space depends on three factors including the branching factor:

- The *base-line factor*, i.e., the number of initial facts and goals;
- The *depth factor*, i.e., the depth of the solution in the search tree; and
- The branching factor.

Taking the base-line factor into account is important for dealing with real world problems, since they often have a large number of irrelevant facts which may be expensive to identify. The depth factor is important for comparative analysis since there is often a break-even search depth for competing problem solving methods and we need to know which side of the break-even point we are on.

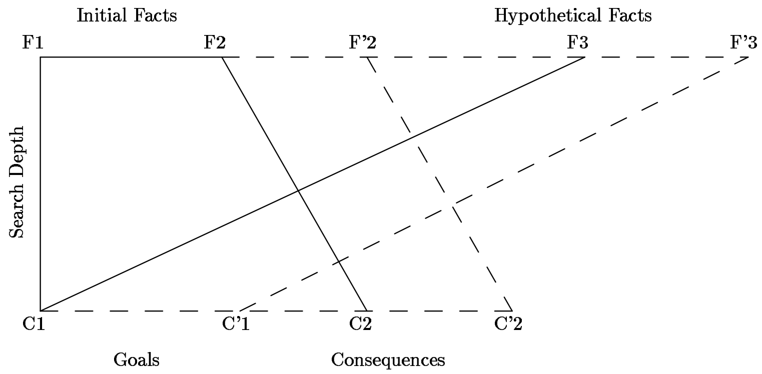

Figure 3.1: Qualitative Model of Forward and Backward Search Space

Figure 3.1 presents a qualitative comparison of forward and backward reasoning search space based on search depth, branching factor, and initial facts and goals. In Figure 3.1, the trapezoid $F_1F_2C_2C_1$ represents the forward search space of a hypothetical problem, where the length of F_1F_2 represents the number of initial facts (longer line for more facts), the slope of F_2C_2 the forward branching factor (steeper means smaller branching factor), the length of F_1C_1 the search depth (the longer the deeper), and the length of C_1C_2 the number of deducible facts (the longer the greater).

The triangle $C_1F_3F_1$ represents the backward search space of the same problem, where C_1 is the single goal, the slope of C_1F_3 represents the backward branching factor, and the length F_1F3 represents the number of subgoals explored at the search depth C_1F_1.

For easier comparison, in Figure 3.2 we flip the triangle in Figure 3.1 so that both forward and backward search spaces grow downward when search depth grows deeper. Figure 3.2 illustrates how four problem characteristics affect search spaces:

Search Depth A deeper search tree favors the direction with a smaller branching factor. Depending on the situation, there could be a threshold search depth at which the search spaces are equal in size.

Facts More facts favors backward reasoning.

Goal More goals favors forward reasoning, assuming all goals must be achieved.

Branching Factor A smaller branching factor reduces search space.

We have made some important assumptions in Figure 3.1 and Figure 3.2:

- The search control strategy assumed to be is breadth first search. However, our analysis above is applicable as long as the search space can be viewed as a tree. For instance, the best-first search may sometimes be viewed as exploring a deeper tree with a smaller branching factor, as illustrated in Figure 3.3.

- The initial facts are assumed to be an equal number of inference steps away from the goals. Irrelevant facts pose no problems for our model, since irrelevant facts tend to increase forward reasoning search space while having no effect on backward reasoning or search depth. Therefore irrelevant facts behave the same way as relevant ones in terms of the analysis result. But we cannot measure the distance between a goal and irrelevant facts.

- Initial facts or goals are assumed to be independent of each other. If there are redundant facts/goals, we may draw the F_1F_2 line so that all relevant facts are not closer to the goals than the line. Therefore, the redundant facts will behave like irrelevant facts in terms of the analysis results.

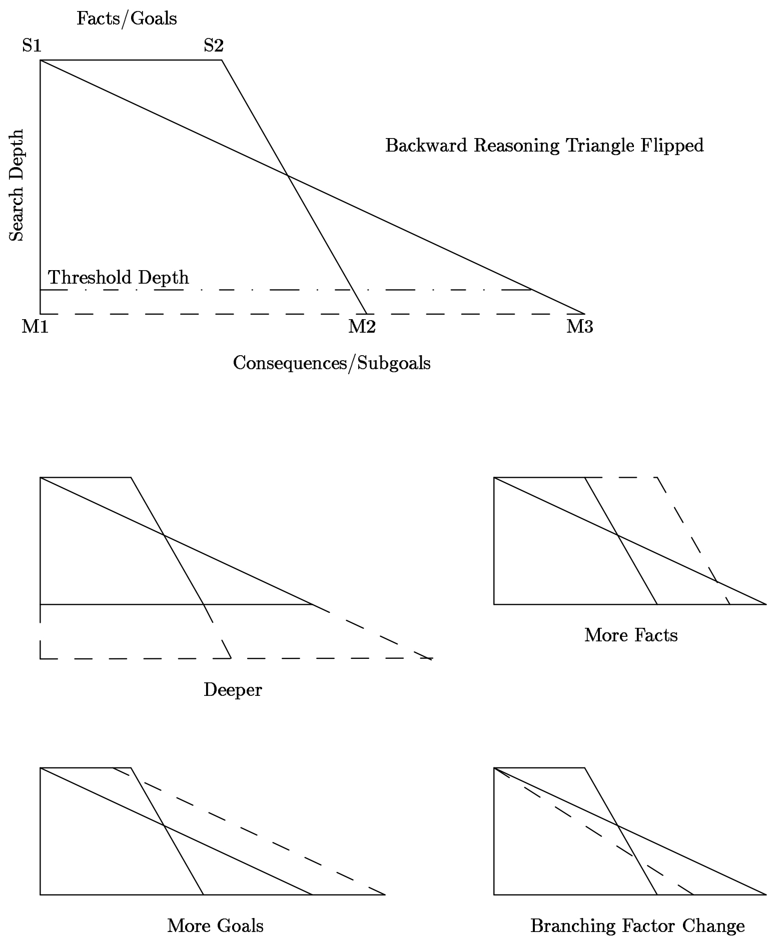

Figure 3.2: Problem Characteristics Determine Search Space

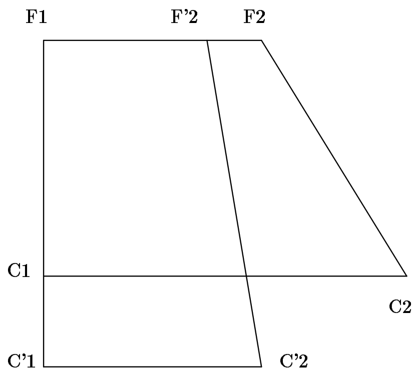

Figure 3.3: Search Space–Breadth First $F_1F_2C_2C_1$ versus Best First $F_1F_2'C_2'C_1'$

3.1.2 Breadth First, Depth First, and Random Search

Breadth versus Depth First

Assume the search space is a perfect tree with a branching factor of B and depth of N. Then a subtree of depth n has a size (in terms of total number of nodes in the tree)

$$T(n) = \frac{B^{n+1} - 1}{B - 1}$$

Further assume that there is exactly one solution at depth $D \leq N$. The knowledge-poor depth first search would explore an average search space of

$$A_d(B, N, D) = P_w(B-1)T(N-1) + 1 + A_d(B, N-1, D-1)$$

where $D \leq N$, $A_d(B, n, 0) = 1$, and P_w is the average ratio of the wrong branches taken before taking the correct one to the total number of wrong branches at each choice point. For knowledge-poor search with truly random choice, $P_w = 1/2$. This formula says that the search space of $A_d(B,N,D)$ consists of three parts: wasting effort on $P_w(B-1)$ wrong subtrees of size $T(N-1)$, examining the top node at the depth 0, and solving the problem with both D and N reduced by 1: $A_d(B,N-1,D-1)$.

Expanding A_d and T on the right, we get

$$A_d(B,N,D) = D+1+P_w(B-1)\sum_{d=1}^{D} T(N-d) = D+1+P_w(B-1)\left(\frac{B^{N+1}-B^{N+1-D}}{(B-1)^2} - \frac{D}{B-1}\right)$$

On the other hand, the average search space for breadth first search is

$$A_b(B,N,D) = T(D-1) + 1 + P_w(B^D - 1)$$

where $D > 0$, and $A_b(B,N,0) = 1$.

D	D/N	A_d	A_b	A_d/A_b
0.0	0.0	1.0	1.0	1.0
1.0	0.1	514.5	2.5	205.8
2.0	0.2	772.0	5.5	140.3636
3.0	0.3	901.5	11.5	78.3913
4.0	0.4	967.0	23.5	41.14894
5.0	0.5	1000.5	47.5	21.06316
6.0	0.6	1018.0	95.5	10.65969
7.0	0.7	1027.5	191.5	5.365535
8.0	0.8	1033.0	383.5	2.693611
9.0	0.9	1036.5	767.5	1.350489
10.0	1.0	1039.0	1535.5	.6766526

Table 3.1: Depth first search space A_d versus breadth first A_b, $B = 2, P_w = 1/2$

D	D/N	A_d	A_b	A_d/A_b
0.0	0.0	1.0	1.0	1.0
1.0	0.1	29527.0	3.0	9842.333
2.0	0.2	39370.0	9.0	4374.444
3.0	0.3	42652.0	27.0	1579.704
4.0	0.4	43747.0	81.0	540.0864
5.0	0.5	44113.0	243.0	181.535
6.0	0.6	44236.0	729.0	60.68038
7.0	0.7	44278.0	2187.0	20.246
8.0	0.8	44293.0	6561.0	6.750953
9.0	0.9	44299.0	19683.0	2.250622
10.0	1.0	44302.0	59049.0	.7502583

Table 3.2: Depth first search space A_d versus breadth first A_b, $B = 3, P_w = 1/2$

D	D/N	A_d	A_b	A_d/A_b
0.0	0.0	1.0	1.0	1.0
1.0	0.1	53.25	2.05	25.97561
2.0	0.2	79.9	4.15	19.25301
3.0	0.3	93.75	8.35	11.22754
4.0	0.4	101.2	16.75	6.041791
5.0	0.5	105.45	33.55	3.14307
6.0	0.6	108.1	67.15	1.609829
7.0	0.7	109.95	134.35	.8183848
8.0	0.8	111.4	268.75	.4145116
9.0	0.9	112.65	537.55	.2095619
10.0	1.0	113.8	1075.15	.1058457

Table 3.3: Depth first search space A_d versus breadth first A_b, $B = 2, P_w = 1/20$

D	D/N	A_d	A_b	A_d/A_b
0.0	0.0	1.0	1.0	1.0
1.0	0.1	2954.5	2.1	1406.905
2.0	0.2	3939.7	5.4	729.5741
3.0	0.3	4268.8	15.3	279.0065
4.0	0.4	4379.2	45.0	97.31556
5.0	0.5	4416.7	134.1	32.93587
6.0	0.6	4429.9	401.4	11.03612
7.0	0.7	4435.0	1203.3	3.685698
8.0	0.8	4437.4	3609.0	1.229537
9.0	0.9	4438.9	10826.1	.4100184
10.0	1.0	4440.1	32477.4	.1367135

Table 3.4: Depth first search space A_d versus breadth first A_b, $B = 3, P_w = 1/20$

D	D/N	A_d	A_b	A_d/A_b
0.0	0.0	1.0	1.0	1.0
1.0	0.1	297.25	2.01	147.8856
2.0	0.2	396.67	5.04	78.70437
3.0	0.3	430.48	14.13	30.46568
4.0	0.4	442.42	41.4	10.68647
5.0	0.5	447.07	123.21	3.62852
6.0	0.6	449.29	368.64	1.218777
7.0	0.7	450.7	1104.93	.4078991
8.0	0.8	451.84	3313.8	0.136351
9.0	0.9	452.89	9940.41	.0455605
10.0	1.0	453.91	29820.24	.0152215

Table 3.5: Depth first search space A_d versus breadth first A_b, $B = 3, P_w = 1/200$

D	D/N	A_d	A_b	A_d/A_b
2.0	0.2	3.0	5.444444	.5510204
3.0	0.3	3285.0	15.44444	212.6978
4.0	0.4	4380.0	45.44444	96.38142
5.0	0.5	4746.0	135.4444	35.0402
6.0	0.6	4869.0	405.4444	12.00904
7.0	0.7	4911.0	1215.444	4.040497
8.0	0.8	4926.0	3645.444	1.351276
9.0	0.9	4932.0	10935.44	.4510105
10.0	1.0	4935.0	32805.44	.1504323

$B = 3, P_w = 1/2, N_g = 9$

Table 3.6: Depth first search space A_d versus breadth first A_b

D	D/N	A_d	A_b	A_d/A_b
3.0	0.3	4.0	14.48148	.2762148
4.0	0.4	1099.0	42.48148	25.8701
5.0	0.5	1465.0	126.4815	11.58272
6.0	0.6	1588.0	378.4815	4.195714
7.0	0.7	1630.0	1134.481	1.43678
8.0	0.8	1645.0	3402.481	.4834707
9.0	0.9	1651.0	10206.48	0.16176
10.0	1.0	1654.0	30618.48	.0540197

$B = 3, P_w = 1/2, N_g = 27$

Table 3.7: Depth First Search Space A_d versus Breadth First A_b

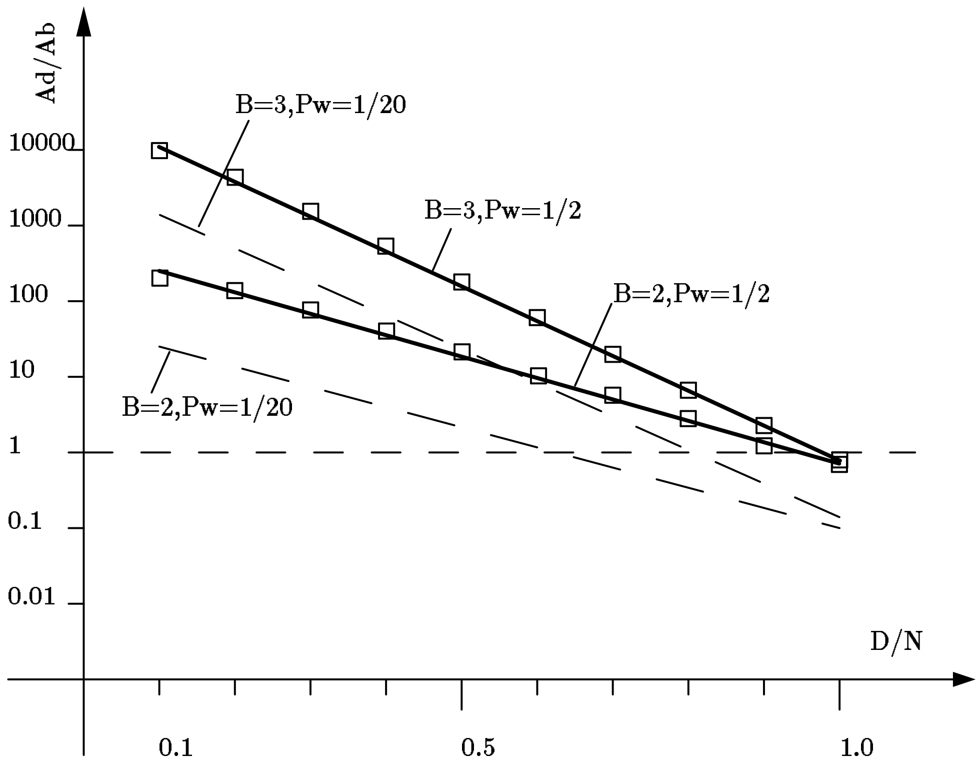

Figure 3.4: Depth versus Breadth First, Knowledge-Poor versus Knowledge-Rich

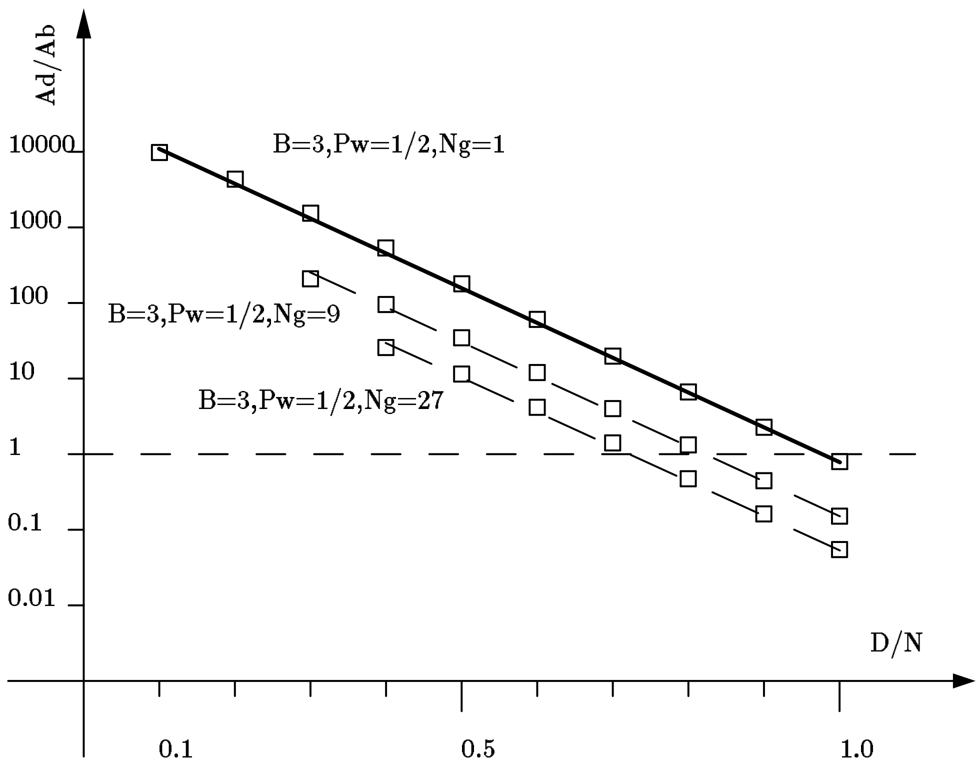

Figure 3.5: Depth versus Breadth First, Different Number of Goals

Tables 3.1 and 3.2 list D/N, A_d, A_b, and A_d/A_b for $D = 0, 1, ..., N$ when $B = 2, 3$ and $N = 10$. The A_d/A_b's are plotted over the D/N axis in bold lines in Figure 3.4. From these two examples we can see that knowledge-poor depth-first search does poorly compared with breadth-first search when $D/N < 0.9$. The larger the branching factor, the larger the differences between these two strategies.

Different heuristics can change different aspects of the search. If a heuristic can estimate the solution depth accurately, i.e., giving us a large D/N ratio so that we can cut the search at a smaller depth without missing the solution, depth-first will have a smaller search space. The search space reduction is exponential in the search depth reduction. Breadth-first search is insensitive to the accuracy of the search depth estimate. When the search cut-off depth is the same as the solution depth, depth-first search has a slightly smaller search space than breadth-first search, as shown in Figure 3.4.

If heuristics can increase the odds of picking up a correct branch at each choice point, the entire depth-first search space will decrease proportionally. Breadth-first search space, on the other hand, will decrease proportionally only at the solution level D of the search tree, because it must first exhaust the first $D-1$ levels. Tables 3.2, 3.4, and 3.5 show A_d/A_bs when P_w is reduced from 1/2 to 1/20 or 1/200 (which reduces the effective branching factor B_e of depth-first search from $B_e = 1 + 2P_w(B_a - 1) = 2$ to $B_e = 1.2$ or 1.02 respectively). Tables 3.1 and 3.3 show A_d/A_bs when $P_w = 1/2$ and 1/20 for $B_a = 2$. The two thin curves in Figure 3.4 are A_d/A_bs for $B = 2$ and 3 when $P_w = 1/20$ (according to Tables 3.3 and 3.4).

When there are $N_g > 1$ solutions randomly distributed at level D and we are interested in finding one solution, the breadth first search space will not change for the first $D-1$ levels and the D-th level will reduce roughly by a factor of N_g. For depth first search, multiple solutions will effectively reduce both the search depth D and the search bound N by $log_B(N_g)$ because the first $log_B(N_g)$ choices are likely to be correct and after that the search reduces to searching for a single solution. Tables 3.2, 3.6, and 3.7 show how breadth first search space compared with depth first when there is 1, 9 and 27 goals at level D. Figure 3.5 shows the curves.

In summary, depth-first search is at a disadvantage, as compared with breadth-first search, because it may explore search spaces beyond the solution depth. There are three factors that may change this picture: the search depth estimation, the choice evaluation, and the number of goals. Breadth-first search is insensitive to the search depth estimation and is affected only at the solution depth by the other two factors. Depth-first search, on the other hand, is affected more favorably by all the three factors. Therefore, if we have heuristics to tighten the search depth bound or make better choices, or there are more than one goal, depth-first search may explore a smaller search space than breadth-first search.

Random Search

Random search picks an unexplored node randomly. Assume the search tree has a constant branching factor B, there are β paths in the search tree, there are o unexplored nodes at the beginning, and solutions are all D steps away from some node. The function

$$O(n, B, o) = o + (B-1)n$$

denotes the number of nodes after processing n nodes, and when $\beta = 1$, i.e., there is exactly one solution path,

$$P_a(n, B, o, D) = (\prod_{i=1}^{n} \frac{O(i-1, B, o) - 1}{O(i-1, B, o)}) \sum_{1 \leq d_1 < \ldots < d_D = n} (\prod_{j=1}^{D} \frac{1}{O(d_j - 1, B, o) - 1})$$

or,

$$P_a(n, B, o, D) = (\prod_{i=1}^{n} \frac{o + (B-1)(i-1) - 1}{o + (B-1)(i-1)}) \sum_{1 \leq d_1 < \ldots < d_D = n} (\prod_{j=1}^{D} \frac{1}{o + (B-1)(d_j - 1) - 1})$$

denotes the probability of hitting the solution when expanding the n-th node, where the first product is the probability of expanding the first N open nodes not on any solution path, while each product in the summation is the probability adjustment if we picked D open nodes on the solution path instead.

For a simple example, let us examine a perfect binary search tree with exactly one solution at one step away from the root, that is, $o = 1, B = 2, \beta = 1$, and $D = 1$. We assume that the tree is infinitively deep, with the solution node being the only leaf node at a finite depth. Then we have

$$P_a(n, 2, 1, 1) = (\prod_{i=1}^{n} \frac{i-1}{i})(\frac{1}{n-1}) = \frac{1}{n}\frac{1}{n-1}$$

The average number of nodes processed before we hit on the solution is

$$A = \sum_{i=2}^{\infty} i P_a(i, 2, 1, 1) = \sum_{j=1}^{\infty} \frac{1}{j} \rightarrow \infty$$

If we process N nodes maximum, that is, stop the search if we fail to hit the solution after processing N open nodes, the overall probability of hitting the solution is

$$P_i(N) = \sum_{n=2}^{N} P_a(n, 2, 1, 1) = 1 - \frac{1}{N}$$

The chance of missing the solution after processing N open nodes is

$$1 - P_i(N) = \frac{1}{N}$$

The average number of nodes expanded will be

$$A(N) = N(1 - P_i(N)) + \sum_{n=2}^{N} n P_a(n) = 1 + \sum_{n=1}^{N-1} \frac{1}{n} \approx 1.577 + ln(N)$$

Table 3.8 has some representative values of P_i and $E(N)$:

N	10	100	1,000	10,000	100,000	1,000,000
$P_i(N)$	0.9	0.99	0.999	0.9999	0.99999	0.999999
$A(N)$	3.83	6.12	8.48	10.79	13.1	15.4
$ln(N) + 1.577$	3.88	6.18	8.48	10.79	13.1	15.4

Table 3.8: Success rate ($P_i(N)$) and average search space $A(N)$

Note that, with the breadth-first search strategy, we will be certain to get the solution if we expand a maximum of 3 and a minimum of 2 open nodes, since the solution node is only one step away from the root. Therefore, the average number of nodes expended will be 2.5.

This result shows that when the tree is much deeper than where the solution is, random search explores a larger space than breadth-first search without guaranteeing a solution.

3.2 Empirical Comparison of Control Strategies –The Extension Tree Method

In this section we present the extension tree method for comparing forward and backward search, using the search space model presented in §3.1.1. We will then adapt this method for comparing sequential and parallel processing of conjunctive subgoals. Note that these two pairs of search strategies are values of different attributes for the search APSP; thus they are examples of the sort of comparisons we must make for a single problem and a single APSP, illustrating the need for a collection of inexpensive comparison techniques.

3.2.1 The Extension Tree Method for Quantitative Comparison of Forward and Backward Reasoning

In this section we present an extension tree method for estimating whether forward reasoning is more efficient than backward reasoning for a given problem. We will first walk through an example to show how the extension tree method works, then summarize the characteristics of it, and finally demonstrate its power with an example.

The Extension Tree Method Illustrated

The extension tree method constructs two kinds of trees, one for forward reasoning and another for backward reasoning. We will use the size of a forward (or backward) extension tree to estimate the number of facts (or goals) derived from each original fact (or goal). The method is outlined below:

- Drop all arguments of predicates
- Build search trees for the abstract problem
  - Build one forward extension tree for each abstract fact
  - Build one backward extension tree for each abstract goal
- Run the problem for two steps forward and calibrate forward search trees
- Analyze the result

The following is an example problem [1] where P is a predicate; INV is a function; A, B, and E are constants; and the small letters represent variables:

Initial Facts: $\forall x\ (P\ (INV\ x)\ x\ E), (P\ E\ A\ B)$

Rule 1: $\forall x\, y\, z\, u\, v\, w\ (P\ x\ y\ u) \wedge (P\ y\ z\ v) \wedge (P\ u\ z\ w) \Rightarrow (P\ x\ v\ w)$

Initial Goal: $(P\ (INV\ (INV\ A))\ E\ B)$

First the problem is abstracted by reducing predicates to their predicate names. For example, both facts and the goal of the example above would be abstracted to P, and the rule becomes $P \wedge P \wedge P \Rightarrow P$:

Initial Facts: P, P

Rule 1: $P \wedge P \wedge P \Rightarrow P$

Initial Goal: P

The forward extension trees are built in the following way: Each forward extension tree will start with an abstracted initial fact as the root. Each leaf node of the extension tree will then be matched with the left hand side of each abstracted rule. For each match we will add a child to the leaf node. For instance, in our example,

[1] This problem is from group theory. The INV is the inverse function; $(P\ x\ y\ z)$ means that the product of x and y is z; and the rule 1 is the association rule for production. A more traditional version of the problem would be:

Initial Facts: $\overline{x} \bullet x = E,\ E \bullet A = B$

Rule 1: $(x \bullet y = u) \wedge (y \bullet z = v) \wedge (u \bullet z = w) \Rightarrow (x \bullet v = w)$

Initial Goal: $\overline{\overline{A}} \bullet E = B$

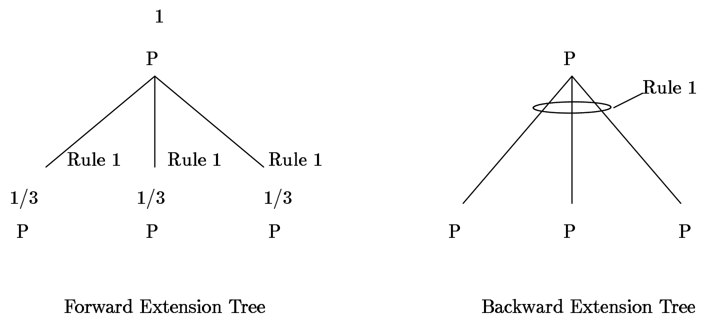

Figure 3.6: Extension Trees

P will match the rule once for each of the three conjuncts at the left hand side and create three children. Each child is simply the right hand side of the abstracted rule. The newly attached leaf nodes will be matched against the rules and the tree grows until a leaf node is equal to one of its ancestors (i.e., a repetition cut-off).[2] Figure 3.6 shows the forward extension tree for this example, which has two levels and four nodes. Note that we only need to generate one extension tree for duplicated abstract facts.

In the forward extension tree, each link can be viewed as one of the N contributions needed for the corresponding rule to trigger, where N is the number of antecedents of the rule. Since separately matched antecedents may not be compatible, we introduce a compatibility constant C, $0 \leq C \leq 1$, and assign C^{N-1}/N to the child as its *relative weight*. The *weight* of each node, therefore, is the product of its relative weight and

[2]Or until it has reached a predetermined depth (depth cut-off).
The depth cut-off is introduced to prevent this method from generating arbitrarily large extension trees against our desire for a fast method. When depth cut-off actually happens, it will reduce the "sampling size" of the extension tree. The reduced sampling size may reduce the accuracy of the estimate when the forward or backward branching factors near the initial facts are different from that near the initial goals.

the weight of its parent. The root weighs 1. In Figure 3.6 we have labeled the weights for each node in the forward extension tree for $C = 1$.

Each problem has its own compatibility constant C. The C is so calibrated that the weighted sizes of the first few levels of the forward extension trees are close[3] to the sizes of the first two levels of the real search tree.

The backward reasoning extension trees are built in the same way as conventional backward reasoning. A backward reasoning extension tree starts with an abstract goal as the root. For instance, in the example above, the root is P. Then each leaf node in the tree will be matched against the righthand side of the rules of the problem. Each match will create one child for each of the conjuncts on the left hand side of the matched rule. The backward extension tree is also subject to cutoff. Figure 3.6 also shows the backward extension tree, which has two levels and four nodes. In backward extension trees, each node counts as 1, representing a plausible subgoal. We have not used a compatibility constant for backward extension trees so far.

For each problem we build one extension tree for each different abstract fact or goal. The trees will be weighted according to how many original facts or goals they represent. For our example, the forward extension tree in Figure 3.6 will be counted twice, while the backward extension tree will be counted once in the final tally, because the forward extension tree represents two initial facts and the backward

[3]Since it does not matter whether we select forward or backward reasoning when the search spaces are similar in sizes, we need no fancy definition for "close." We can define "close" as the sum of the difference between the weighted sizes of the extension tree and the real search tree at these levels being zero, for instance.

extension tree represents one goal.

Characteristics of the Extension Tree Method

An extension tree is an *abstract simulation* of the real inference process, using abstract facts and goals. The base-line factor is resolved by counting each extension tree the same number of times as there are corresponding initial facts or goals.

The branching factor is resolved by calibrating the compatibility factor.

The depth factor is resolved by qualifying the comparison with applicable search depth ranges.[4]

In summary, the construction of the extension trees requires the problem be represented as facts and inference rules. It assumes breadth-first search strategy for both forward and backward reasoning. For reasons similar to those presented in the previous section, the accuracy of the extension tree method may deteriorate. However, with (1) the first few levels calibrated and (2) using forward and backward extension trees together[5], we are confident that we will get "good estimates."

The cost of building extension trees is expected to be less than that of solving the problem because each abstract fact or goal may replace many real ones, and there is no unification cost for the variables. Thus we achieve the "inexpensive" requirement by abstract simulation.

[4]If we have to select a search direction without knowing the search depth, we may assume the search depth is about half that of the extension trees. But the assumed search depth is not a part of the estimate. Estimating search depth may well be another research project by itself.

[5]The "cancel-out" heuristic has it that the forward and backward extension tree will deviate from the real search space in the same way, thus leaving the comparison valid.

An Example: The Families Problem

The families problem used in our experiment has 42 domain inference rules, 77 initial domain facts (including 72 inequality assertions among family members) and one goal. Samples of the facts and rules as well as the goal are listed below. [6]

Facts: $(MARRIED\ P\ N)$

$(GRANDCHILD\ Q\ K\ S)$

$(NIECE\ M\ L\ O)$

$(UNCLE\ S\ L\ M)$

$(GRANDFATHER\ O\ L\ N)$

$(\neq\ K\ L)$

$\ldots$

Rules: $\forall\, x\, y\, z\ (CHILD\ x\ y) \wedge (CHILD\ y\ z) \Rightarrow (GRANDCHILD\ x\ y\ z)$

$\forall\, x\, y\, z\ (FEMALE\ z) \wedge (CHILD\ y\ z) \wedge (SIBLING\ x\ y) \Rightarrow (NIECE\ x\ y\ z)$

$\ldots$

Goal: $\exists x\ (SIBLING\ S\ x)$

The experimental results are listed in Table 3.9. Each table entry for forward/backward reasoning is the number of facts/goals explored up to that depth. Each entry for extension trees is the total (weighted) non-root nodes up to that level.

[6] This example is adapted from [Van Baalen, 1989]. The original problem involves equality reasoning. Thus the backward reasoning has a large branching factor. In the version here, we get rid of the need for equality reasoning by making the anonymous person in ordinary family relations explicit. For example, we make the middle generation person explicit in the grandfather relation as in the $GRANDCHILD$ relation: $\forall\, x\, y\, z\ (CHILD\ x\ y) \wedge (CHILD\ y\ z) \Rightarrow (GRANDCHILD\ x\ y\ z)$. Therefore, this adapted version is expected to have similar forward and backward branching factors.

Depth	1	2	3	4	5	6	7	8	9	10
Forward Extension Tree	15	31	45	51	56	57	58	58	58	
Forward Reasoning	12	30	51	56						
Backward Extension Tree	9	34	97	226	427	598	672	678		
Backward Reasoning	6	32	64	101	107	108	108			

Table 3.9: Weighted Extension Tree Sizes and Search Space Sizes

The compatibility constant used in the forward extension tree is 1/3, which is selected to bring the weighted forward extension tree size at depth 2 close to the actual search space size.

The maximum depths of forward and backward extension trees are 9 and 8, respectively. The forward reasoning inferred ($SIBLING\ S\ O$) at search depth 2 after firing 17 domain rules. The forward reasoning was exhausted at depth 4 after firing 56 domain rules. The backward reasoning generated all necessary subgoals at search depth 2 after firing 25 domain rules. The backward reasoning was exhausted at depth 7 after firing 108 domain rules.

Backward reasoning explores a smaller search space than forward reasoning when the search depth is no deeper than 2. If the search depth is deeper, forward reasoning beats backward reasoning. The extension tree estimates are consistent with the experiment results.

When we added more family members to the above problem while leaving other things mostly unchanged, the forward search space increased while the backward search space remained unchanged. The extension tree method predicted this.

It is interesting that the extension trees correctly indicated that forward reasoning has a tendency to stop earlier than backward reasoning in this experiment. Although it worked for this example, it seems that the backward extension tree may also require calibration at a depth deeper than a few levels.

3.2.2 Using the Extension Tree for Comparing Sequential and Parallel Conjunctive Subgoal Processing

The extension trees can be either used as direct estimates of the search space, or used as representative samples of the search space. In the previous section, we have used extension trees as direct estimates of forward and backward search space. In this section, we are going to use extension trees as representative samples of the search space. In particular, we will use extension trees to compare sequential and parallel conjunctive subgoal processing. In this example, only backward extension trees will be involved.

Sequential versus Parallel Conjunctive Subgoal Processing

It is well known that processing conjunctive subgoals in sequence requires less work than processing them in parallel. It is believed that there are two situations when sequential processing will do better ([de Kleer *et al.*, 1979], pages 103-104):

- If subgoals are pursued in sequence, the work on the subgoals pursued later will be saved when one subgoal fails.

- In sequential processing, solutions to conjunctive subgoals solved earlier will restrict the solutions to subgoals solved later.

However, our study shows that there are exceptions to these.

- For the first situation above, we need to watch out for subgoals that require an arbitrary amount of work to conclude failure.
- For the second situation, we believe
  - Sequential processing of conjunctive subgoals might search a deeper but less branchy tree than parallel processing; thus, sequential conjunctive processing is not always better.
  - Sequential processing of conjunctive subgoals might search several different restricted versions of a goal, thus duplicating some search efforts. This factor is significant when we want to find all alternative solutions to the top level goal.

In the rest of this section we will first present a search space model, which will lead to a surprising prediction: Sequential processing of conjunctive subgoals may often explore a larger search space than parallel processing. We will then show how to use the extension tree method to apply the analysis results to real problems.

The Search Spaces for Sequential and Parallel Conjunctive Subgoal Processing

We assume a hypothetical problem, where the branching factors are constants (that is, every subgoal matches the same number of different rules' consequent parts) and every rule has the same number of antecedents. We also assume that all the first level subgoals of the hypothetical problem will be first achieved at the same depth of the search tree. We further assume that the control strategy for disjunctive subgoals is breadth first, that the search continues for alternative solutions after a subgoal is achieved, and the search stops when the first solution to the top level goal is achieved.

Let

- $H > 0$ be the depth of the inference tree for parallel conjunctive subgoal processing,
- $K > 0$ be the number of rules that match every subgoal, and
- $N > 0$ be the number of antecedents per rule, see Figure 3.7.

Then, the size of the tree for parallel conjunctive subgoal processing is $(KN)^H$, since each goal generates KN subgoals (see Figure 3.7, sequential processing).

On the other hand, the size of the tree for sequential conjunctive subgoal processing is at least K^{NH}, since it takes at least H steps to achieve each of the N top level subgoals, and the search for alternative solutions to the first subgoal continues until the last moment (see Figure 3.7, parallel processing).

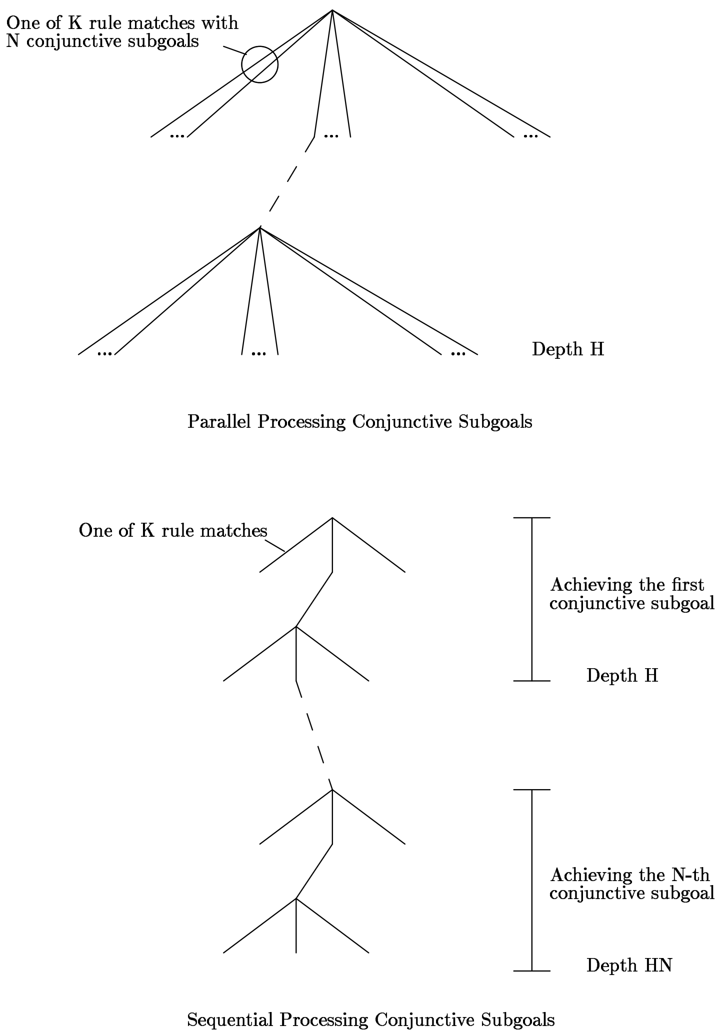

Figure 3.7: Search Space, Parallel vs Sequential Subgoaling

Therefore, under the assumptions mentioned above, sequential processing of conjunctive subgoals will be exploring a smaller search space than parallel processing only when $K^N < KN$.

This means that sequential processing of conjunctive subgoals is very likely to explore a larger search space than parallel processing, counter to the prevailing believe that sequential processing explores a smaller search space than parallel processing. The reason for this some what surprising result is analyzed in the beginning of this section. The result is also confirmed by both a variation of the extension tree method and experiment on a small problem to be presented next.

Using the Extension Tree Method to Compare Sequential and Parallel Conjunctive Subgoal Processing

As typical with complexity analysis, the above result cannot be directly applied to real problems when the parameters are neither available nor ideally constant. In this section we will demonstrate that the extension trees can be used as a representative sample of the search space for the purpose of computing average parameters.

Figure 3.8 is a group theory problem reduced from [Wos, 1988]. Figure 3.9 shows the backward extension tree for it, with the root on the left. The extension tree is similar to the search tree that parallel conjunctive subgoal processing would explore. In the extension tree, conjunctive subgoals have the links leading into them connected by an arc. Obviously, this problem does not have a constant N and K. We opt to use the following method to estimate KN and K^N by their average effective node

$$
\begin{gathered}
\neg(O\ (INV\ x)) \Rightarrow \neg(O\ x) \\
(O\ x) \Rightarrow (O\ (INV\ x)) \\
\neg(O\ x) \Rightarrow \neg(O\ (INV\ x)) \\
(O\ (INV\ x)) \Rightarrow (O\ x) \\
\neg(O\ z) \wedge (P\ x\ y\ z) \wedge (O\ y) \Rightarrow \neg(O\ x) \\
\neg(O\ z) \wedge (P\ x\ y\ z) \wedge (O\ x) \Rightarrow \neg(O\ y) \\
\neg(O\ z) \wedge (O\ y) \wedge (O\ x) \Rightarrow \neg(P\ x\ y\ z) \\
(P\ x\ y\ z) \wedge (O\ y) \wedge (O\ x) \Rightarrow (O\ z) \\
(P\ A\ C\ D) \\
(O\ B) \\
(O\ C) \\
(P\ (INV\ B)\ C\ (INV\ A)) \\
goal:\ (O\ D)
\end{gathered}
$$

Figure 3.8: A Deduction Problem SIT

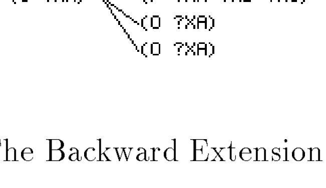

Figure 3.9: The Backward Extension Tree of SIT

values:

- For each node which has k matching rules each of which has n_i, $i = 1, \ldots, k$, conjunctive antecedents, its effective node KN and K^N are $\frac{1}{k}\sum_{i=1}^{k} kn_i$ and $\frac{1}{k}\sum_{i=1}^{k} k^{n_i}$, respectively.

- For the whole problem, we take the average effective node KN and K^N over the entire extension tree.

For our example SIT, the average node effective KN and K^N are roughly 5 and 10. That is, parallel processing of conjunctive subgoals would explore a smaller search space than sequential processing by our estimation. In reality, parallel conjunctive subgoal processing confirmed the goal after firing 29 domain rules backward while sequential processing triggered 95 domain rules.

Chapter 4

Relevance Analysis

Reducing irrelevant inference can increase problem solving efficiency, and may change the comparison between competing PSPs. For example, assume we are considering whether we should reason forward or backward for an inference problem that gives information about ten thousands of families but the query involves only one family. Assume this problem involves equality reasoning, thus has a large backward branching factor. On the other hand, this problem has many initial facts thus forward reasoning has a large baseline factor. None of the two search direction is attractive. For this kind of problems, we can either try to get rid of the equality reasoning to make the backward inference less branchy, or we can try to reduce the number of the initial facts by separating the family in question from the rest, which is the subject of this chapter.

Reducing irrelevant inference can increase problem solving efficiency provided that the effort in reducing irrelevant inference is less than the reduced inference. The goal of relevance analysis is thus to achieve overall problem solving efficiency, rather than simply reducing irrelevant inference to a minimum.

There are at least two things one can do to make relevance analysis more efficient:

- We can study when certain relevance analysis techniques are cost effective.

  We believe that problem solving paradigm is a major factor in determining what kinds of relevance analysis are cost effective. For instance, assume we have an inference problem defined by IF-THEN inference rules and facts. For forward reasoning, it would be useful to know whether a fact is relevant since inference on irrelevant facts will be wasted. For backward reasoning, on the other hand, the relevance of a fact is not as useful since backward reasoning will naturally "avoid" irrelevant facts.

  Therefore, we propose that each relevance analysis technique should specify the kinds of PSPs for which it is cost effective, and each PSP should be associated with a set of relevance analysis techniques that are cost effective for it. This association between relevance analysis techniques and PSPs has been mostly missing in the AI literature. This topic is left as future research and is not covered by this thesis except that we will present a relevance analysis technique.

- Develop cost effective relevance analysis techniques.

  It is our view that the emphasis of relevance analysis research has long been on generally applicable relevance analysis techniques. However, due to the cost associated with such analysis techniques, we have not seen much of their applications on real problems. We believe that cost effective relevance analysis techniques are mostly PSP specific. Therefore, we should start to build a large collection of "quick and dirty" relevance analysis methods for real world problems.

This chapter will present two such techniques, an object relevance analysis technique and Van Baalen's connection graph analysis technique, that when combined are cost effective for forward chaining inference.

4.1 Connection Graph Analysis

[Van Baalen, 1989] reports a connection graph relevance analysis technique, which is a dual of the object relevance analysis reported here. The object relevance analysis ignores the difference between different relations and performs a simplified backward chaining inference in order to determine relevant objects. The connection graph analysis ignores the difference between different objects and performs a simplified backward chaining inference in order to determine relevant relations.

We believe there are several ideas behind the connection graph analysis. The first is similar to backward chaining, that only the inference related to the query is necessary. A relation or an inference rule is irrelevant to the query if it will not contribute to the query. The second is that a rule is irrelevant if there is no fact to trigger it, or a fact is irrelevant if there is no rule to trigger on it. The third is that by ignoring the difference of objects, the connection graph analysis will cost less than solving the problem although it may not always find every irrelevant rule or fact.

4.2 Object Relevance Analysis

4.2.1 Motivation–The Families Problem

Here is a simple inference problem that one can find in LSAT or GRE tests (used by [Van Baalen, 1989]):

> M, N, O, P, Q, R, and S are all members of the same family. N is married to P. S is the grandchild of Q. O is the niece of M. The mother of S is the only sister of M. R is Q's only child. M has no brothers. N is the grandfather of O. Who are the siblings of S?

This problem is easy to formulate as a logic deduction problem. Note that the statement "the mother of S is the only sister of M" mentions an anonymous person–the mother of S. To solve a problem with such anonymous objects one needs to use equality axioms in the deduction. Unfortunately, using equality axioms means that the backward branching factor is large, because there are many ways that one can conclude that two things are equal. Therefore, this problem should be solved by forward reasoning from the facts.

Now imagine that the statements about this family are mixed with those of tens of thousands of other families, as one would expect in a realistic knowledge base (hereafter called the ten-thousand family problem). Clearly, forward reasoning from the facts is undesirable because we would waste our energy on tens of thousands of irrelevant families for the question about the siblings of S. Due to the large branching factor caused by the equality axioms, backward reasoning is also undesirable.

The dilemma can be solved if we can separate the family in question from others, then reason forward on the members of the single family. Object relevance analysis is designed to do exactly this. The idea is to collect all the family members that are related to S, the person mentioned in the question, through a chain of family relation. In more technical terms, the idea is to treat all family relations as a single equivalence relation, then compute the equivalence class that contains the objects mentioned in the query. The resulting equivalence class contains the objects relevant to the query while all others are irrelevant.

Note that under this definition, a "relevant object" may not in fact be needed for answering the query. The design goal of object-relevance analysis is correctness and efficiency; that is, we want first to be sure to include every object needed for answering the query, and only then aim for speed of analysis, instead of squeezing out every possible irrelevant object.

The rest of the section will present the basic object relevance analysis algorithm, the handling of statements without universal quantifiers, the single family example, the handling of statements with universal quantifiers, and a small family problem with domain inference rules.

4.2.2 The Basic Algorithm

The object relevance analysis is suitable for logic inference problem. The basic algorithm processes only assertions:

- Conceptually replace all the relations in the problem with an equivalence relation, say "object-relevant." All arguments to an "object-relevant" relation are pair-wise considered as "object-relevant."
- Collect all the objects mentioned in the query into a set "relevant-objects."
- Add into that set all the objects that have an explicit "object-relevant" relation with an object already in the relevant-object set (processing one object-relevant statement at a time).
- Repeat the above step until no more new objects can be added.
- Return the relevant-objects set.

This algorithm computes the equivalence set under the object-relevant relation for the query-mentioned objects. This algorithm is $O(e)$ in complexity, where $e \leq O(n^2)$ is the number of explicit object-relevant relations among the objects of the resulting relevant-object set and n is the size of the set. For the ten-thousand family problem, n is the size of one family. Therefore, this algorithm is inexpensive but effective, the kind of algorithms we are looking for.

4.2.3 Statements without Universal Quantifiers

Quantifier- and connector-free statements are processed by the basic algorithm. Problem statements with existential quantifiers or connectors need to be reformulated into quantifier- and connector-free statements, by the following process:

- Rewrite statements with existential quantifiers using Skolem functions. The application of Skolem functions are treated as objects.

- Rewrite statements without quantifiers as individual predicates by simply breaking at each connective. For example, both $A \wedge B$ and $A \vee B$ will be rewritten as A and B. Breaking the $\vee$ connector this way may lead to false relevant objects, but will not miss relevant ones.

- Ignore negative statements without quantifiers. Such statements are not considered as relating the objects in their arguments. Here we assume all relations are "normal" by default, where a normal relation is a relation that implies relatedness between its arguments. An intuitive example of abnormal relation could be something like ($unrelated\ A\ B$), whose negation means relatedness between its arguments. We hypothesize that under certain circumstances, the normality of clauses may be recognized by examining the domain inference rules and seeing how a clause is used. We leave this recognition for future research.

4.2.4 The Single Family Problem

For the simple inference problem (hereafter the single family problem), we have the following statements after Skolemization:

```
(married N P)
(grandchild Q S)
(niece M O)
(mother S mother(S)) and (sister M mother(S)) <Skolemized>
(child Q R)
not (brother M non-exist(M)) <ignored>
(grandfather O N)

Query: (sibling S x)
```

After breaking connectors and ignoring negative statements, we have:

```
(married N P)
(grandchild Q S)
(niece M O)
(mother S mother(S))
(sister M mother(S))
(child Q R)
(grandfather O N)
```

Below is a trace of the object relevance analysis algorithm for the single family problem. Note that all family relations, that is, sibling, grandchild, mother, sister, niece, grandfather, child, and married, are aliases of the object-relevant relation.

```
{S}                       query: (sibling S x) [(grandchild Q S) (mother S mother(S))]
{S Q}                     (grandchild Q S)     [(mother S mother(S)) (child Q R)]
{S Q mother(S)}           (mother S mother(S)) [(child Q R) (sister M mother(S))]
{S Q mother(S) M}         (sister M mother(S)) [(child Q R) (niece M O)]
{S Q mother(S) M O}       (niece M O)          [(child Q R) (grandfather O N)]
{S Q mother(S) M O N}     (grandfather O N)    [(child Q R) (married N P)]
{S Q mother(S) M O N R}   (child Q R)          [(married N P)]
{S Q mother(S) M O N R P} (married N P)        []
```

The relevant-object set is initially {`S`}, since S is the only object mentioned in the query. Each line starts with the current relevant-object set after each of the object-relevant statements is processed, followed by the statement itself, and the set of the object relevant statements to be processed in square brackets. The final relevant-object set is {`S Q mother(S) M O N R P`}, containing all the family members as expected.

If we add people from other families to the problem, the basic algorithm will separate them out. For example, if we add:

T is the brother of U, U is the uncle of V, ..., we will have additional object-relevant statements as the following:

```
(brother U T)
(uncle V U)
...
```

The final relevant-object set, however, will remain unchanged since T, U, and V are not connected to any member of the relevant-objects set through an object-relevant relation.

4.2.5 Statements with Universal Quantifiers

Universally quantified statements are domain inference rules. For the object-relevance analysis to be valid, every domain inference should be mapped to object-relevance inferences. The object-relevance inference is more efficient than the actual inference because it collapses all domain relations into a single object-relevant relation, and thus dramatically simplifies the domain inference rules.

Since not all relations in the problem are equivalent relations, forcing them to be equivalent relations may introduce inconsistency. Because ignoring differences between relations loosens the constraints on what can be derived, we will only derive more facts than the original problem. Because our goal is to preserve object relevance conservatively, such inconsistency causes no problem.

Statements with universal quantifiers may be classified into three types for the purpose of object relevance analysis:

- Those that render all objects relevant. We say such statements establish "wild" relevance. When such statements exist, the object relevance analysis will not rule out any object as irrelevant.
- Those that render some constant objects relevant. We say such statements establish "restrained" relevance. Object relevance analysis should take so established relevance into account.
- Those that are subsumed by the equality inference rules. Such statements may be ignored by the object relevance analysis.

Atomic clauses are easy to classify. An atomic clause with universally quantified variables will cause our object-relevance analysis to consider every object relevant since the variable can be bound to any object. If such clauses exist, the object-relevance analysis will not find any irrelevant objects.

Clauses with a negated atomic clause as their only literal can be ignored since such clauses do not establish relevance between objects. Here we assume the relations involved are normal.

Non-atomic clauses are more complicated to classify. Below is our algorithm for classifying statements with universally quantified variables:

Without loosing generosity, we assume all relations are binary and every clause can be rewritten as

$(R\ x_1\ y_1) \wedge ... \wedge (R\ x_n\ y_n) \Rightarrow (R\ u_1\ v_1) \vee ... \vee (R\ u_m\ v_m)$

where R is short-hand for "object-relevant."

- Compute the equivalence classes among $x_1, ..., x_n, y_1, ..., y_n$ according to the n R equivalence relation on the left-hand side of the "implication." These are called left-hand equivalence classes. (A non-binary R will be treated in a similar way, that is, all its arguments will be put in the same equivalence classes.)

- Compute the equivalence classes among $u_1, ..., u_m, v_1, ..., v_m$ according to the m R equivalence relation on the right-hand side of the "implication." These are called right-hand equivalence classes.

- Check if any one of of the right-hand equivalence classes has members (1) belonging to two or more different left-hand equivalence classes and at least one of the involved left-hand equivalence classes consists of only variables, or (2) belonging to no left-hand equivalence classes and at least one of such members of the right-hand equivalence class is a variable. If, and only if, either of the two situations is true, the clause establishes "wild" relevances, that is, this clause may cause every object to be object-relevant to every other.

- If every right-hand equivalence class is included by a left-hand equivalence class, this clause is subsumed by the properties of the object-relevant relation and can be ignored.

- In all other cases the clause establishes "restrained" relevance and should be turned into corresponding object-relevant assertions.

If the problem has abnormal relations, the literals of abnormal relations should change sides so that they will take negative forms.

The standard clause form used in this algorithm reflects one of the several ways to use a clause as an inference rule, that is, when all the left-hand literals are consistently bound we can conclude the right-hand conjunctions. This is the most productive inference we can make in terms of propagating the object-relevant relation using that clause. If any literal changes sides in the "implication," it will become negative, which will not propagate the object-relevant relation. Therefore, it will be adequate to use the standard form in the algorithm.

Examples of different types of statements can be found in §4.2.6, where we will classify statements using the algorithm above.

Summary—Dealing with Universal Quantifiers

Statements with universal quantifiers are domain inference rules. Domain inference rules can be classified into three types: Those that establish "wild" relevance, those that establish "restrained" relevance, and those that are subsumed by the properties of the object-relevant relation. The first type will render object-relevance analysis ineffective. In our experiments we have not encountered this type of statements. We hypothesize that such statements are rare in practical problems.

The complexity of statement classification depends on the domain inference rules. Because the size of the rule set is relatively small for problems such as the ten-thousand family problem, the classification is inexpensive. Therefore, the overall cost of object relevant analysis is small compared with the cost of actually solving the problem.

4.2.6 The Small Family Problem With Domain Inference Rules

In this subsection, we illustrate the clause pre-processing with a small families problem. This problem is adapted from [Van Baalen, 1989].

Here is the problem in English:

> P, Q, R, and S are all members of the same family. Q's grandchild is S. R is the only child of P. Q is married to P. Who are the parents of S?

The problem in predicate calculus:

$$
\begin{aligned}
&family\text{-}member(P), family\text{-}member(Q),\\
&family\text{-}member(R), family\text{-}member(S)\\
&grandchild(Q,S)\\
&\forall x\ child(P,x) \Leftrightarrow x = R\\
&married(Q,P)\\
&\forall x \forall y[grandchild(x,y) \Leftrightarrow \exists z(child(x,z) \wedge child(z,y))]\\
&\forall x \forall y[child(x,y) \Leftrightarrow parent(y,x)]\\
&\forall x \forall y \forall c[child(x,c) \wedge child(y,c) \wedge x \neq y \Rightarrow married(x,y)]\\
&\forall x \forall y \forall c[married(x,y) \wedge child(x,c) \Rightarrow child(y,c)]\\
&\forall x \neg married(x,x)\\
&\forall x \forall y[married(x,y) \Leftrightarrow married(y,x)]\\
&\forall x \forall y \forall z[married(x,y) \wedge married(y,z) \Rightarrow \neg married(x,z)]\\
&Query: parent(S,x)
\end{aligned}
$$

After Skolemization (gcc is a skolem function, lower-case arguments are variables, otherwise they are constants) and omitting universal quantifiers:

$$
\begin{aligned}
&family\text{-}member(P), family\text{-}member(Q),\\
&family\text{-}member(R), family\text{-}member(S)\\
&grandchild(Q,S)\\
&child(P,x) \Leftrightarrow x = R\\
&married(Q,P)\\
&grandchild(x,y) \Leftrightarrow (child(x, gcc(x,y)) \wedge child(gcc(x,y),y))\\
&child(x,y) \Leftrightarrow parent(y,x)\\
&child(x,c) \wedge child(y,c) \wedge x \neq y \Rightarrow married(x,y)\\
&married(x,y) \wedge child(x,c) \Rightarrow child(y,c)\\
&\neg married(x,x)\\
&married(x,y) \Leftrightarrow married(y,x)\\
&married(x,y) \wedge married(y,z) \Rightarrow \neg married(x,z)\\
&Query:\ parent(S,x)
\end{aligned}
$$

After replacing every predicate with *object-relevant*:

$$
\begin{aligned}
&\textit{object-relevant}(P), \textit{object-relevant}(Q),\\
&\textit{object-relevant}(R), \textit{object-relevant}(S)\\
&\textit{object-relevant}(Q, S)\\
&\textit{object-relevant}(P, x) \Leftrightarrow \textit{object-relevant}(x, R)\\
&\textit{object-relevant}(Q, P)\\
&\textit{object-relevant}(x, y) \Leftrightarrow\\
&\quad (\textit{object-relevant}(x, gcc(x, y)) \wedge \textit{object-relevant}(gcc(x, y), y))\\
&\textit{object-relevant}(x, y) \Leftrightarrow \textit{object-relevant}(y, x)\\
&\textit{object-relevant}(x, c) \wedge \textit{object-relevant}(y, c) \wedge \neg\textit{object-relevant}(x, y)\\
&\quad \Rightarrow \textit{object-relevant}(x, y)\\
&\textit{object-relevant}(x, y) \wedge \textit{object-relevant}(x, c) \Rightarrow \textit{object-relevant}(y, c)\\
&\neg\textit{object-relevant}(x, x)\\
&\textit{object-relevant}(x, y) \Leftrightarrow \textit{object-relevant}(y, x)\\
&\textit{object-relevant}(x, y) \wedge \textit{object-relevant}(y, z) \Rightarrow \neg\textit{object-relevant}(x, z)\\
&\textit{Query}: \ \textit{object-relevant}(S, x)
\end{aligned}
$$

Statement classification:

- The first four unary *object-relevant* clauses are ignored since they do not propagate object-relevance between objects.

- $\textit{object-relevant}(P, x) \Leftrightarrow \textit{object-relevant}(x, R)$

  This rule can be rewritten as two implications:

$$
\begin{aligned}
&\textit{object-relevant}(P, x) \Rightarrow \textit{object-relevant}(x, R)\\
&\textit{object-relevant}(P, x) \Leftarrow \textit{object-relevant}(x, R)
\end{aligned}
$$

  For the first implication, we have one equivalence class on the left-hand side, namely, $\{P, x\}$. Since the pair on the right-hand side, $\{x, R\}$, neither crosses two equivalence classes nor is contained by a single one, we have a rule establishing "restrained" relevance. This rule defines an equivalence class $\{P, x, R\}$ and we need to replace this rule with assertion $object\text{-}relevant(P, R)$. The variable x in the equivalence class is ignored. If the variable can ever be bound to some object, we will know that it is related to P or R from the source that defined the binding; therefore, ignoring it here will not cause a problem.

The second implication is similar.

- $object\text{-}relevant(x, y) \Leftrightarrow$
  $(object\text{-}relevant(x, gcc(x, y)) \wedge object\text{-}relevant(gcc(x, y), y))$

  This rule can be rewritten as three implications in the standard form:

$$
\begin{aligned}
&object\text{-}relevant(x, y) \Rightarrow object\text{-}relevant(x, gcc(x, y))\\
&object\text{-}relevant(x, y) \Rightarrow object\text{-}relevant(gcc(x, y), y)\\
&object\text{-}relevant(x, gcc(x, y)) \wedge object\text{-}relevant(gcc(x, y), y)) \Rightarrow\\
&\quad object\text{-}relevant(x, y)
\end{aligned}
$$

  The third implication is subsumed by the property of *object-relevant* and thus is ignored. The first two establishes "restrained" relevance since we consider Skolem functions as constants. The object-relevant equivalence classes involved are $\{x, gcc(x, y)\}$ and $\{gcc(x, y), y\}$, which contain only one constant. Thus there are no object-relevant assertions to be made.

- $object\text{-}relevant(x, y) \Leftrightarrow object\text{-}relevant(y, x)$

  This rule is subsumed by the property of *object-relevant*: it is symmetric. We do not have to recognize this subsumption; our classification algorithm will decide to ignore it because the right-hand equivalence class is included in a left-hand equivalence class.

- $object\text{-}relevant(x, c) \wedge object\text{-}relevant(y, c) \wedge \neg object\text{-}relevant(x, y) \Rightarrow object\text{-}relevant(x, y)$

  This rule can be rewritten as

  $object\text{-}relevant(x, c) \wedge object\text{-}relevant(y, c) \Rightarrow$
  $object\text{-}relevant(x, y) \vee object\text{-}relevant(x, y)$

  which will be duly ignored.

- $\neg object\text{-}relevant(x, x)$

  This rule is inconsistent with the property of *object-relevant*. According to our classification algorithm, it will be rewritten as

  $object\text{-}relevant(x, x) \Rightarrow$

  and will be ignored. Our statement classification algorithm also works for atomic clauses and this is an example.

- $object\text{-}relevant(x, y) \wedge object\text{-}relevant(y, z) \Rightarrow \neg object\text{-}relevant(x, z)$

  This rule is inconsistent with the transitivity of *object-relevant*. Similarly to the previous rule, it will be rewritten as

  $object\text{-}relevant(x, y) \wedge object\text{-}relevant(y, z) \wedge object\text{-}relevant(x, z) \Rightarrow$

and will be ignored. Again, we do not have to recognize this rule as an inconsistency.

So the statement classification leaves with us with $object\text{-}relevant(P, R)$. Since S is mentioned in the query and we also have $object\text{-}relevant(Q, S)$ and $object\text{-}relevant(Q, P)$, all four people, P, Q, R, and S are relevant.

4.3 Combining Connection Graph and Object Relevance Analysis

The effectiveness of both object relevance analysis and the connection graph analysis depends on the initial formulation of the problem. The addition of certain irrelevant facts or rules may render these techniques totally ineffective. For instance, if we add the following irrelevant facts to the single family problem (see §4.2.4), the object relevance analysis will not be able to exclude T, U, and V:

```
(haircolor P Red)
(haircolor T Red)
```

Therefore, the more irrelevant facts we know, the more irrelevant facts we can discover. One possible way to get around this dilemma is to use multiple relevance analysis techniques to help each other. For instance, the connection graph analysis is such a technique.

4.3.1 The Mini-Family Problem: Connection Graph Analysis

The connection graph analysis is reported in [Van Baalen, 1989]. The algorithm is listed here for the benefit of the reader:

- Represent the problem in clause form. Ignore equality literals or any other literals that will be interpreted non-declaratively by removing them.
- Ignore the arguments of relations and remove duplicated rules.
- Connect literals from different rules or facts that are negation to each other.
- Remove all the rules and facts that do not connect to the query. These rules or facts will not contribute to the query.
- Remove the rules that contain unconnected literals. Remove facts that become unconnected due to the rule removal. These rules will not have the facts to trigger them.
- Repeat the above until no more rules can be removed.
- All the removed rules and facts are irrelevant to the query.

We will use the mini-family problem to illustrate how the connection graph analysis work:

> M has only one brother, N. M has no sisters. All M's children have red hair. Name all M's siblings.

The predicate form is:

$\forall x(brother\ M\ x) \Rightarrow (=\ x\ N)$
$\forall x \neg(sister\ M\ x)$
$\forall x(child\ M\ x) \Rightarrow (haircolor\ x\ Red)$
$\forall x, y(sibling\ x\ y) \iff (brother\ x\ y) \vee (sister\ x\ y)$
Query: $(sibling\ M\ x)$

The clause form is (the $\iff$ statement is translated into three clauses):

$\neg(brother\ M\ x) \vee (=\ x\ N)$
$\neg(sister\ M\ x)$
$\neg(child\ M\ x) \vee (haircolor\ x\ Red)$
$\neg(sibling\ x\ y) \vee (brother\ x\ y) \vee (sister\ x\ y)$
$\neg(sister\ x\ y) \vee (sibling\ x\ y)$
$\neg(brother\ x\ y) \vee (sibling\ x\ y)$
Query: $(sibling\ M\ x)$

After ignoring the difference between objects and equality relations:

$\neg brother$
$\neg sister$
$\neg child \vee haircolor$
$\neg sibling \vee brother \vee sister$
$\neg sister \vee sibling$
$\neg brother \vee sibling$
Query: $sibling$

Figure 4.1 shows the final connection graph. The $\neg child \vee haircolor$ clause is not connected to the query and thus is irrelevant. Both *child* and *haircolor* are also unconnected literals. There are no other unconnected literals.

The connection algorithm may be improved by connecting only those literals that can connect to the query. This might be a good idea if there are many unconnected rules.

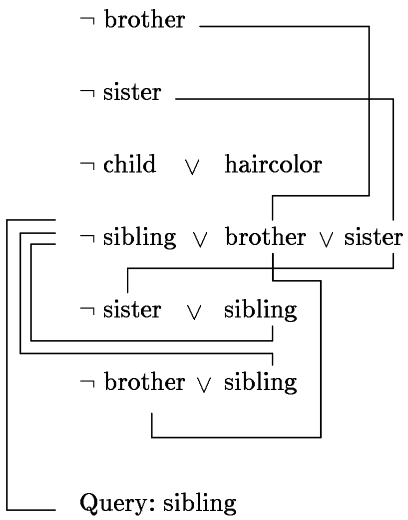

Figure 4.1: A Connection Graph

4.3.2 The Three Family Problem: Connection Graph and Object Relevance Analysis Combined

Here we are going to demonstrate with a three family problem how the object relevance analysis and the connection graph analysis can help each other.

This three family problem has members of three families: A, B, and C. Members of the same family will be subscripted, e.g., A_{10}, B_1, C_5, etc. For simplicity, we are going to give the Skolemized predicate form of the problem directly:

$(grandchild\ A1\ A3)$
$(child\ A10\ x) \Rightarrow x = A2$
$(married\ A1\ A10)$
$(child\ A10\ x) \Rightarrow (haircolor\ x\ Red)$
$(haircolor\ x\ Red) \Rightarrow (toe\text{-}type\ x\ Pigeon)$
$(haircolor\ x\ Red) \Rightarrow (child\ A1\ x)$
$(toe\text{-}type\ A2\ normal)$
$(toe\text{-}type\ B2\ normal)$
$(grandchild\ B1\ B3)$

$(child\ B10\ x) \Rightarrow x = B2$
$(married\ B1\ B10)$
$(grandchild\ C1\ C3)$
$(child\ C10\ x) \Rightarrow x = C2$
$(married\ C1\ C10)$

$(child\ x\ c) \wedge (child\ y\ c) \wedge x \neq y \Rightarrow (married\ x\ y)$
$(married\ x\ y) \wedge (child\ x\ c) \Rightarrow (child\ y\ c)$
$(grandchild\ x\ y) \iff (child\ x\ (mgc\ x\ y)) \wedge (child\ (mgc\ x\ y)\ y)))$
$(child\ x\ y) \Rightarrow (parent\ y\ x)$

Query: $(parent\ B3\ x)$

The upper part of Figure 4.2 is the connection graph for this problem, the lower part is the object relevant sets. Each text line in the connection graph part represents an abstract rule (i.e., ignoring the arguments). The solid lines in the connection graph are the consolidated connection between positive and negative literals, representing possible unifications in a inference process. The solid lines in the object relevant sets indicate object relevant relations; the objects connected through these lines are in the same object relevant sets. The query is connected to the connection graph or the object relevant sets through the dashed arrows.

Assume we perform the object relevance analysis first. This will identify the objects in circle $O1$ as irrelevant, that is, the family C is ruled irrelevant. The assertions involving objects in $O1$ are subsequently removed as in Figure 4.3.

Then we perform the connection graph analysis. This will identify the relation *toe-type* as irrelevant as pointed by thick arrows $C1$ as in Figure 4.3. So the rules and facts that involve *toe-type* are removed. The removal of *toe-type* cuts the object relevant relation at the places around the objet "normal." The removal of the rule

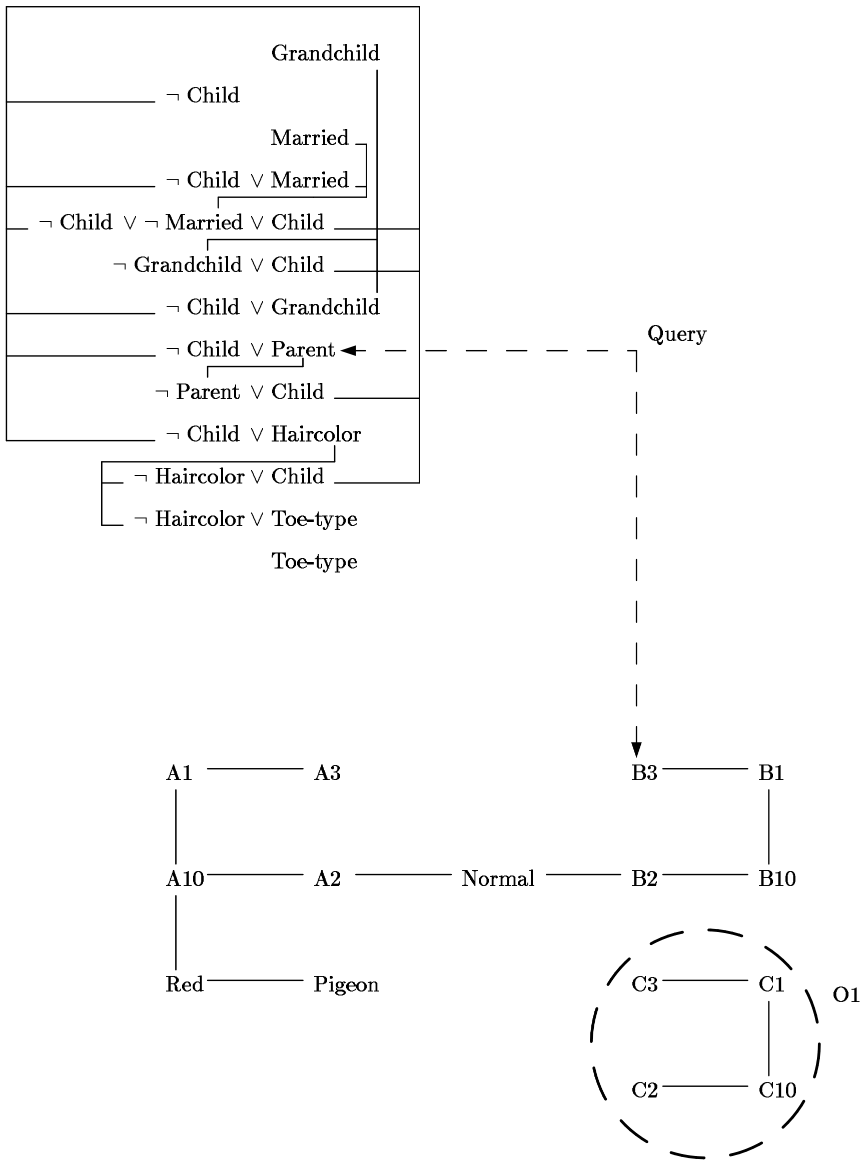

Figure 4.2: The Object Relevance Analysis

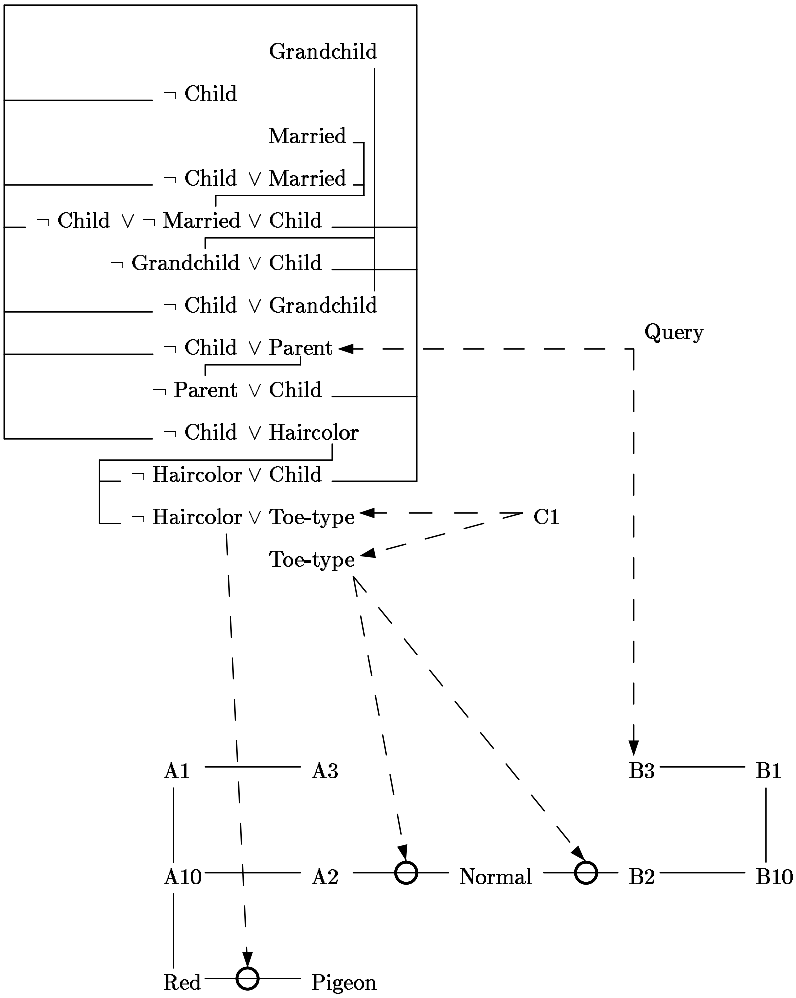

Figure 4.3: The Connection Graph Analysis

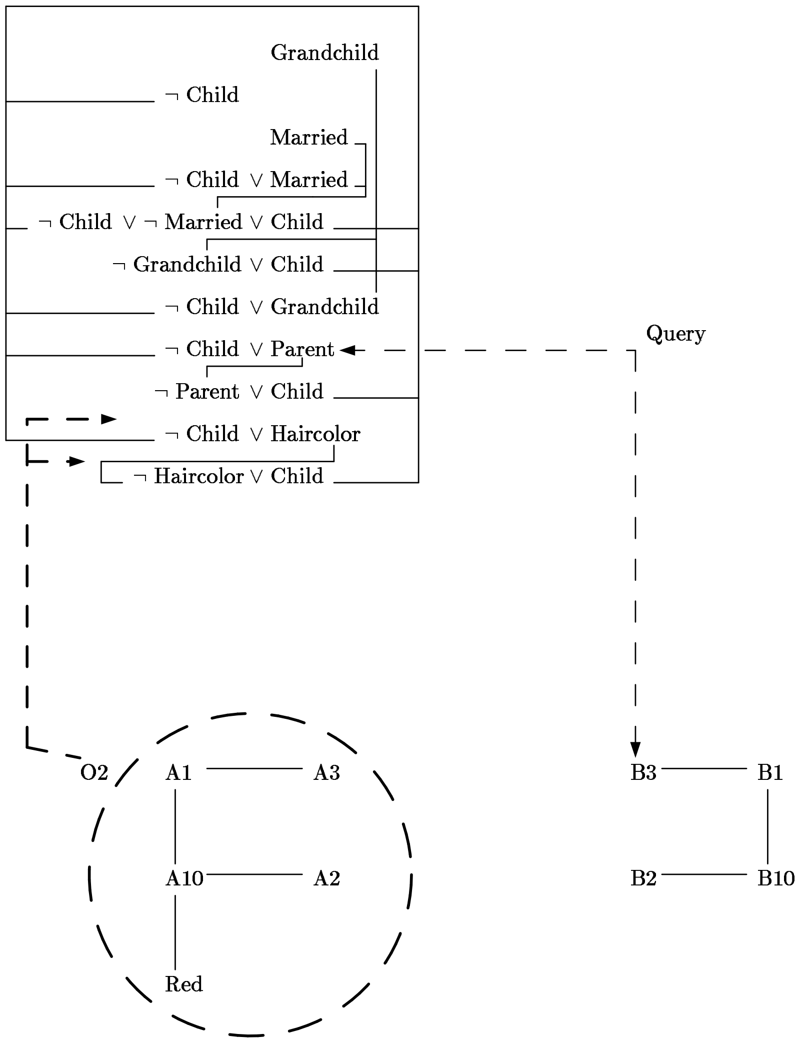

Figure 4.4: The Object Relevance Analysis—Round 2

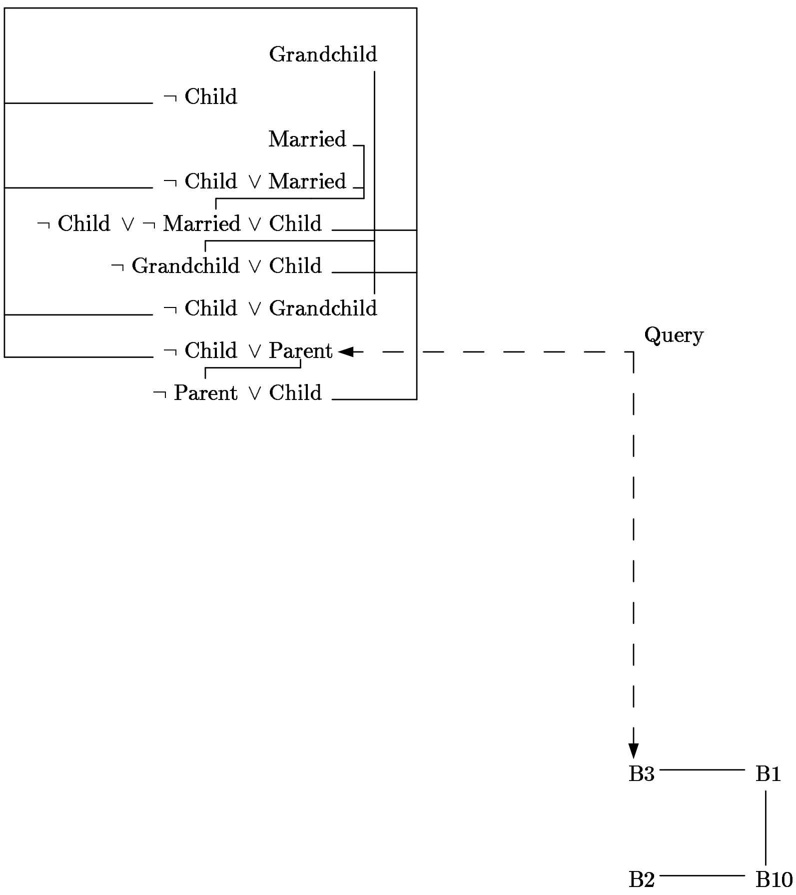

Figure 4.5: The Relevance Analyses—The Final Result

(*haircolor x Red*) $\Rightarrow$ (*toe-type x Pigeon*) eliminates the object relevance relation between *Red* and *Pigeon*. Figure 4.4 shows the remaining facts and rules after the connection graph analysis.

In the next round of the object relevance analysis, members of family A are ruled as irrelevant as indicated by circle *O*2 in Figure 4.4. Removing members of family A and related objects lead to the removal of the two rules pointed by the arrows starting from *O*2 because they mention members of family A.

The next round of the connection graph analysis does not find any new irrelevant rules and the process halts, after removing more irrelevant facts and rules than could either technique by itself. Figure 4.5 shows the final remaining facts and rules.

Chapter 5

Related and Future Work

5.1 Related Work

5.1.1 On PSP Selection

Most work on problem solving in AI may be considered to be finding new PSPs or varieties of existing ones.

Work on representation and problem reformulation, such as [Amarel, 1968], [Korf, 1980], [Korf, 1985], and [Van Baalen, 1989], may be considered to be at the representation design stage. Most of the representation or problem reformulations involved are incremental changes based on syntactic properties of an initial representation. The incremental approach is sometime called reformulation from first principle and is encouraged by the fact that the elusive "leaps of insight" sometimes demonstrated by humans can be modeled by gradual refinement of an initial representation. However, because there is unlikely a general principle for problem and representation reformulation [Newell, 1966], problem and representation reformulation techniques will likely have only narrow applications as they have been. Because of the huge search space that the incremental approach mandates, most results obtained in this line are still "possibility demonstrations." In representation and problem reformulation research, the PSPs involved in using the representation or solving the problem are often overlooked. On the other hand, our PSP selection system takes a knowledge-based approach. We associate PSPs directly with preferred representation. Thus it is possible to introduce representation or problem perception jumps by switching PSPs. Our PSP selection system interacts with a user, thus we may get help from humans who understand the purpose and semantics of the problem.

Work on multiple representations of the same artifact requires a relatively information-complete model of the artifact in question. Simpler models will be used according to the task at hand. Simpler models are either automatically generated as in [Weld, 1989] or hand-coded as in [de Kleer, 1977]. The selection of different models is similar to PSP selection. The run-time selection of models ensures their ability to handle a wide range of tasks. However, the requirement of an information-complete model is costly for large problems. In contrast, our PSP selection system is oriented to a single problem and advocates the strategy of building a model most suitable for the task at hand. In theory, though, our PSP selection system may include the PSP that selects different models of the same artifact for different tasks.

[Lowry, 1987a] presents a formal framework for some of the existing reformulation methods. Lowry's method works in the problem space while our system works in the PSP space, which is a better defined space than the problem space. Lowry's system reformulates problems by removing superfluous distinctions in the initial problem specifications. In comparison, our system does not even require initial specifications and encourages the user to consider different problem perceptions. Lowry's method is rigorous but may suffer from the complexity and diversity of real world problems. Our system is not as formal, but we believe our system is more accessible to the user and covers more diverse PSPs.

[Lowry, 1987b] represents "Generic Designs" as parameterized theories. The generic designs are similar to our APSPs without the component structure.

In a future version of the PSP selection system, program segments may be associated with the PSP and preferred representation recommendations. The user, then, formally specifies the problem in the preferred representation. In a sense, the PSP selection system is also a computer-aided formal specification system as well as an automatic programming system.

Work on analogical problem solving such as [Carbonell, 1985] tries to use past experience as PSPs. This line of research emphasizes the amount of knowledge extracted from past experience instead of the range of application. We believe past experience is a rich source of PSPs. As the MCB and T&J examples show, sometimes copying top-level problem decomposition of past experience can be very effective. Because of the absence of details, top-level problem decomposition is a coarse image of past experience, simple to represent and flexible to apply. PSP components are intended as representations of top-level problem decomposition.

[Chandrasekaran, 1986] identified several APSPs, or 'generic tasks' as he calls them, such as, in his terminology, 'hierarchical classification,' 'knowledge-directed information passing,' 'abductive assembly,' 'hierarchical design by plan selection and refinement,' and 'state abstraction.' All the generic tasks are intended as building blocks for an expert system design instead of problem solving in general. [Chandrasekaran, 1986] did not address the issue of how to computerize the selection or combination of those building blocks. Our definition of PSPs is at a similar level as that of his generic tasks.

Research on human problem solving, such as [Kaplan and Simon, 1990], will continue to reveal PSPs that people use. We believe that human PSPs are less well defined then computerized PSPs and need additional work before they can be encoded effectively.

5.1.2 On PSP Comparison

Jeff Van Baleen used a connection graph method for relevance analysis([Van Baalen, 1989]). His abstraction from a given problem to the connection graph is similar to our abstraction from search space to extension

The extension tree method is an attempt to apply complexity analysis results to real problems. While the extension tree methods, especially the abstract simulation and calibration idea may be applicable to other complexity analysis results, we believe in general that we need to study individual complexity analysis results in order to apply them to real problems.

5.1.3 On Relevance Analysis

Jeff Van Baleen used a connection graph method for relevance analysis [Van Baalen, 1989]. The object relevance analysis is a dual of his connection graph analysis in terms of the treatment of predicates and their arguments: The connection graph analysis ignores the arguments of predicates while the object relevance analysis ignores the predicates.

[Subramanian and Genesereth, 1987] presents a theory for deriving additional irrelevant facts from known irrelevant ones. In comparison, our object relevance and connection graph methods derives irrelevant facts from known relevant ones, thus is less demanding on the user. In addition, we specify the PSPs for which our relevance analysis is cost-effective while [Subramanian and Genesereth, 1987] does not.

5.2 Future Work

In the future, we would like to try our PSP representation on more PSPs, to explore the limitations of the current representation and to add more knowledge to the system. We also would like to explore more elaborate component specifications such as I/O specifications. Elaborate structure and component specifications provide a necessary foundation for synthesizing APSP by combining components in novel ways.

We would like to extract PSPs from past experience in the way we extracted the PSP that solved the T&J problem. We would like to focus on the top level task decomposition structure of each past experience. We may develop hierarchical abstractions for each past experience.

We would like to explore quick PSP evaluation methods for comparing recommendations. We would like to use the extension tree method to develop practical comparative analysis for other search methods.

Making the PSP explicit makes it possible to select applicable problem reformulation methods. We would like to represent and apply problem reformulation methods in a way similar to PSPs so that we can help with the representation design stage of problem solving.

Eventually, we would like to see an AI consultant system with the capability of producing prototype programs based on the recommendations (in a way similar to [McDermott, 1988]).

We would also like to try our object relevance analysis and connection graph analysis on real problems, such as large data base. Also we would like to work on atomic clauses with variables which make object relevance analysis ineffective. In particular, we would try the following ideas:

- Determine if such clauses are irrelevant through other means, e.g., the connection graph analysis. If so, they can be removed.

- Determine if such clauses are superfluous and can be removed from the problem formulation. For instance, if we have a "same-species" relation that holds for every pair of family member and non-member, we may ask the user if it is possible to remove it from the problem formulation.

- Perform partial inference to "use up" such clauses so that the variables are bound as the problem dictates and the clauses themselves can be removed.

  For example, if we have $(P\ A\ x) \wedge (\neg(P\ x\ y) \vee (Q\ x\ y))$, where $(P\ A\ x)$ is an atomic clause with a variable x and $(\neg(P\ x\ y) \vee (Q\ x\ y))$ is another clause that resolves with the atomic clause, the result of the partial inference about $(P\ A\ x)$ will be $(\neg(P\ x\ y) \vee (Q\ x\ y)) \wedge (Q\ A\ x)$.

  As the example illustrates, the partial inference may introduce a new "trouble maker" and the process recurs. We hypothesize that for practical problems such a process is much simpler than the inference required to solve the original problem, and we would like to find theoretical or experimental support for this hypothesis.

We would also like to detect "abnormal" relations, whose arguments "relevant" to each other only when the relations are negated.

Appendix A

A Collection of Problem Solving Paradigms

This Appendix contains short descriptions of APSPs. Those in the first section are implemented in our experiment program. Those in the second section are not currently implemented.

A.1 Implemented Problem Solving Paradigms

A.1.1 Classification

Attribute-Based Classification

Brief Description: Attribute-based classification classifies objects according to their attributes.

Applicability Conditions: Attribute-based classification is suitable for determining the type of a given object where the attributes of the object are sufficient to do so.

Preferred Representations: Attribute-Based classification may represent type either as a set of attribute templates or a set of discrimination rules on attribute values.

Subtasks: Look up type according to attribute values.

Procedure:

- Look up type according to attribute values by
  - Matching the attribute values with type templates, or
  - Running the attribute values through the discrimination rules

  according to the type representation. The type found could either match the attribute exactly or approximately, depending on the problem.

A.1.2 Constraint Satisfaction

Brelaz K-Coloring Algorithm

Brief Description: Brelaz K-Coloring algorithm assigns values to variables such that there are K different variable values and certain pairs of variables are assigned with different values. This algorithm uses two strong heuristics (forms of most-constrained first) and will optimally solve almost all random K-coloring problems without backtracking if solutions exist.

Applicability Conditions: Brelaz algorithm is suitable for variable assignment problems that require certain pairs of variables to have pair-wise different values and each of the variables can choose from the same set of K different values.

Preferred Representations: Per Brelaz algorithm, see [Brelaz, 1988].

Subtasks: Per Brelaz algorithm.

Procedure: Brelaz algorithm.

Min-Conflict Heuristic

Brief Description: The min-conflict heuristic is effective for the large-N-queens problem, which is to place N queens on a NxN chess board for some large N in such a way that the queens are not attacking each other, that is, on each column (N), row, or diagonal, there is only one queen. When the N is much larger than 10, it will be efficient to place the N queens randomly on the board then try to repair incrementally the placement using the min-conflict heuristic ([Minton *et al.*, 1990]), i.e., relocate one queen at a time that minimizes the number of violated constraints.

Applicability Conditions: The min-heuristic is proven effective for the N-queens problem with a large N. It may also be suitable for constraint satisfaction problems (on small finite variable domains) that have many solutions, that do not require the optimality of the solution, and that have a random assignment of variables that will become a solution after changing a small fraction of the variables.

Preferred Representations: Constraints of suitable form.

Subtasks: Randomly assign variables to obtain a near solution. Count violated constraints. Search for reassignments in order to obtain a solution.

Procedure:

- Randomly assign values to all the variables.
- Until a solution is found,
  - Count violated constraints for every possible single variable reassignment,
  - Select a variable reassignment that does not lead to an old overall assignments and minimizes violated constraints, and
  - Reassign the variable.

Constraint Satisfaction on Finite Domain

Brief Description: Find consistent assignments to variables through enumeration or search that satisfy given constraints. The variables may take finitely many values.

Applicability Conditions: Constraint satisfaction on finite domain is suitable for problems that need to find assignments to variables consistent with given constraints, where each variable has a finite number of values.

Preferred Representations: Constraints of various formats.

Subtasks: Solution testing (constraint verification, either partial or complete assignment), invalid variable value detection (either before any assignment is made or while the assignment is being made), backtracking (chronological, dependency directed, or repairing), variable ordering (least-alternative, most-participation, stable-set-last, cycle-cut-set-first, unspecific), value ordering (most-options-first, unspecific), dependency recording (retroactive, interactive, batch).

Procedure:

- Start with an empty partial assignment. Until a complete assignment is found,
  - Eliminate variable values inconsistent with the partial assignment and the constraints.
  - Select a variable to be assigned next, and
  - Select a value for the selected variable. If no value is consistent with the partial assignment and the constraints, find alternative assignments for some variables assigned already.

Constraint Satisfaction on Infinite Domain

Brief Description: Find consistent assignments to variables through equation solving or numerical methods that satisfy given constraints for variables that can take infinitely many values.

Applicability Conditions: Constraint satisfaction on infinite domain is suitable for problems that need to find assignments to variables consistent with given constraints, where each variable may have an infinite number of values.

Preferred Representations: Mathematic equations.

Subtasks: Solving numerical equations.

Procedure: Equation solving algorithms or relaxation methods.

A.1.3 Diagnosis

Fault-Dictionary-Based Diagnosis

Brief Description: Fault-dictionary-based diagnosis looks up the symptoms in a fault-dictionary.

Applicability Conditions: Fault-dictionary-based diagnosis is suitable for diagnosis problems for which we can build a practical fault dictionary. There must be a reasonable number of faults, a reasonable number of symptoms, and a small number of faults associated with each symptom.

Preferred Representations: Symptom indexed fault dictionary.

Subtasks: Fault dictionary lookup.

Procedure: Use dictionary lookup algorithms.

Empirical-Association-Based Diagnosis

See also the MYCIN, three-phased heuristic classification in Section "Familiar Problem Solving Paradigms." Empirical-association-based diagnosis is the second phase of the three-phased heuristic classification.

Brief Description: Empirical-association-based diagnosis hypothesizes diagnoses using heuristic association between observations and diagnoses.

Applicability Conditions: Empirical-association-based diagnosis is suitable for diagnosis problems for which we know the empirical-association between observations and diagnoses. (A diagnosis problem is to find the causes for given symptoms/observations.)

Preferred Representations: Association rules based on either categorical association or probability association.

Subtasks: Hypothesize diagnoses according to the observations.

Procedure: Hypothesize diagnoses according to the observations.

Model-Based Diagnosis

Brief Description: Model-based diagnosis compares observations and predictions based on a structure and behavior model of the artifact, with components as the diagnosis granule.

Applicability Conditions: Model-based diagnosis is suitable for diagnosing artifacts for which (1) we want to locate malfunctioning components, (2) we know the behavior of and the interactions among the components, (3) we usually can infer inputs from outputs of the components as well as infer outputs from inputs, and (4) we do not know the faulty behavior of the components or there are too many kinds of faulty behavior.

Preferred Representation: For using model-based diagnosis we must use a structure and behavior model for the artifact being diagnosed. That is, we need to represent the components of the artifact, the behavior of these components, and the interactions among the components. We may recursively use a structure and behavior model for some of the components. The three subtasks may use the same artifact representation.

Subtasks: Candidate generation, candidate verification, candidate discrimination.

Procedure:

- Generate candidates by

- Simulating the artifact forward according to the structure and behavior model and the given inputs, assuming all components are normal;
- Simulating the artifact backward according to the model and the given outputs, i.e., observations, assuming all components are normal; and
- For each connection of the components in the artifact's model on which simulation results from the two simulations are inconsistent, identifying the components involved in getting the inconsistent values as candidates.

- Verify candidates by eliminating those that are not consistent with all observations.
- Discriminate candidates by
  - Computing the most informative points to probe inside the artifact, carrying out the probe for additional observations while keeping the inputs unchanged, and eliminating the candidates inconsistent with the new observations; or
  - Computing the most informative new inputs, applying the new inputs and making observations, and eliminating the candidates that are inconsistent with the new observations.

A.1.4 General PSPs

Disproof by Partition and Unsolvable Equation

Brief Description: Disproof by partition and unsolvable equation attempts to prove that a set of conditions is inconsistent by partitioning the objects involved, establishing equations on the number of members in each partition derived from the given set of conditions, and then proving that the equations are unsolvable. The mutilated checker board problem may be solved this way.

Applicability Conditions: Disproof by partition and unsolvable equation is worth trying on problems (1) that have conditions which are questionable as to their consistency, and (2) that have operations engaging a certain number of resources.

Preferred Representations: Equations.

Subtasks: Find a partition. Establish equations related to the partition. Prove that the equations are unsolvable.

Procedure: Until the problem is solved,

- Generate a partition, and
- Generate a set of equations related to the partition.
- Try to prove that the set of equations is not solvable.

Generate and Test

Brief Description: Generate-and-test attempts to solve a problem by generating potential solutions and testing them one by one.

Applicability Conditions: Generate-and-test is suitable for problems for which (1) some conditions that can be verified with only complete solutions, (2) we know how to generate potential solutions, and (3) we know how to test if a solution is valid.

Preferred Representations: Problem specific.

Subtasks: Generate potential solutions (enumerate, debug non-solutions, or opportunistically generate). Test solutions (pass-fail, diagnostic).

Procedure: Until a solution is found or there are no more potential solutions,

- Generate a potential solution, and
- Test whether it is valid.

Order Reasoning

Brief Description: Order reasoning solves order related problems by building the total order chain or partial order tree for the given problems and computing solutions using the order.

Applicability Conditions: Order reasoning is suitable for problems that involve a total or partial order, which usually have only one relation and the only relation is transitive.

Preferred Representations: Partial order tree or total order chain.

Subtasks: Construct the order. Reason about the order.

Procedure:

- Construct the order.
- Reason about the order for the given problem.

Problem Reduction

Brief Description: Problem reduction solves a problem by recursively decomposing it into subproblems until it becomes a primitive problem which we know how to solve directly.

Applicability Conditions: Problem reduction is suitable for problems (1) which we can decompose recursively into primitive problems, and (2) for which we know a library of primitive problems and their solutions.

Preferred Representations: A primitive problem library (primitive problems and solutions), a decomposition library (ways of decomposing problems), and a transform library (ways of changing problems into equivalent ones).

Subtasks: Decompose the problem. Search the decomposing space.

Procedure:

- If there is a solution in the primitive problem library for the problem, use that solution,
- Select an applicable decomposing method and solve every resulting subproblem recursively, or
- Select an applicable transform method and solve the resulting problem recursively.

Algorithmic

Brief Description: Algorithmic methods solve problems by computing or constructing solutions without search, i.e., without trying alternatives and backtracking from wrong choices.

Applicability Conditions: Algorithmic methods are suitable for problems that have known algorithmic methods.

Preferred Representations: Per algorithm.

Subtasks: Per algorithm.

Procedure: Per algorithm.

Search

Brief Description: Search solves a problem by trying alternatives at each choice point and backtracking when a choice turns out to be wrong later.

Applicability Conditions: Search is suitable for problems (1) which need to make a sequence of choices, and (2) about which we do not know how to make the right choice every time.

Preferred Representations: State space or goal space for the search space. Procedural representation for search control.

Subtasks: Generate the search space (state space, goal space). Select alternatives at each choice point (direction: forward, backward, or bidirectional; bookkeeping: tree, or graph; ordering: depth-first, breadth-first, random, best-first, A*, hill-climbing, branch-and-bound, beam, tapered, minimax, alpha-beta, or progressive-deepening; backtracking: chronological, or dependency directed).

Procedure: Until a solution is found, or there is no more alternatives,

- For each choice point, order the alternatives, and
- For each alternative, search recursively for the subsequent situation resulting from that alternative when further search is necessary.

A.2 Unimplemented Problem Solving Paradigms

This section contains some PSPs that we have examined but not encoded.

A.2.1 Means Ends Analysis

Brief Description: Means-ends analysis extracts the difference between the current state and the goal state and applies operators that reduce the difference until the goal state is achieved.

Applicability Conditions: Means-ends analysis can be used for problems that need to find a sequence of operators that change an initial state to a goal state. For means-ends analysis to be effective, we should be able to specify the difference between any state and a goal state and know what kind of differences each operator can reduce.

Preferred representation: The preferred representation for means-ends analysis is the state space. A state should carry information sufficient to predict the future. The problem should specify an initial state and a goal state. There should be operators that change states. The operators should define when they can be applied and how they change states. Preferably, the problem should define and represent the favorableness of operators. In addition to the characteristics common to state space, the problem should define and represent differences between states. Each operator should be associated with the differences it can reduce.

Subtasks: Determine the difference between states. Determine the operators that reduce the differences between states. Select operators. Determine if the prerequisites of an operator are satisfied. Make subgoals to meet prerequisites of operators. Apply operators when prerequisites are met. Determine if a state is promising. Keep track of the relation between goals.

Procedure:

- Until the goal state is achieved or no more operators are available,
  - If the current state becomes unpromising, go back to a previous state and try alternative operators,
  - Extract the difference between the current state and the goal state,
  - Select a promising operator that could reduce the difference,
  - If the operator is not applicable, set a subgoal to change the current state first so that the operator becomes applicable,
  - Apply the operator to the current state, and
- If the goal state is reached, announce success; otherwise announce failure.

A.2.2 MYCIN, Three-Phased Heuristic Classification

Brief Description: MYCIN-like three-phased heuristic classification abstracts observations, hypothesizes diagnoses using heuristic association between observations and diagnoses, then refines the diagnoses with further observations.

Applicability Conditions: MYCIN-like three-phased heuristic classification can be used for reasoning causes from effects when we know heuristic associations from effects to causes. It would be desirable if the associations are between abstracted effects and causes and we know the taxonomies of the effects and causes.

Preferred representation: The preferred representation for MYCIN-like three-phased heuristic classification has three parts: taxonomy of effects, symbolic or qualitative associations between effects and causes, and taxonomy of causes.

Subtasks: There are three subtasks using each of the three parts of representation: effect abstraction, cause hypothesizing, and cause refinement.

Procedure:

- Abstract effects to a level where effect-cause association is known.
- Hypothesize causes according to suitably abstracted effects.
- Refine causes according to effects, making new observations if necessary.

A.2.3 Diagnosis as Assembly

Brief Description: Diagnosis as assembly uses a numerical causal model to predict the aggregated disorders caused by multiple malfunctions. When the observed disorders are not completely accounted for, use the causal model to suggest additional malfunctions until all observed disorders are accounted for.

Applicability Conditions: Diagnosis as assembly is suitable for diagnosing artifacts that may have multiple malfunctions, each of which may contribute to different disorders in different ways. The problem domain must have an adequate numerical theory for computing the combined disorders of multiple malfunctions. A set of malfunction is acceptable as a diagnosis.

Preferred representations: Causal model of the domain indicating causal links between malfunctions and disorders. An adequate causal model for computing the combined disorders of multiple malfunctions.

Subtasks: Select a disorder not yet completely accounted for. Suggest a malfunction for a disorder according to the causal model. Compute disorders according to the malfunctions.

Procedure:

- Until all orders are accounted for,
  - Select a observed disorder not yet completely accounted for by the predicted disorder,
  - Suggest a malfunction for the disorder according to the causal model and the current malfunction set, add the suggested malfunction to the malfunction set,
  - Compute the predicted disorders according to the current malfunction set, or
- Present the final malfunction set as the diagnosis.

A.2.4 Constraint Directed Scheduling

This PSP is near the domain specific end, that is, has domain specific aspects (e.g., orders, equipment, timing, etc).

Domain task: Arrange the use of equipment to fulfill orders.

Brief Description: Machine shop scheduling by satisfying critical constraints first and incrementally satisfying less critical constraints.

Applicability Conditions: Constraint directed scheduling is suitable for scheduling problems that involve temporal constraints which allow few alternative partial orderings among subtasks, resources constraints which allow many alternative resource assignments, and preferences over resource assignments.

Preferred representation: Orders as ordering constraints on tasks. Timing constraint on orders. Equipment capacities as resources for tasks. Non-capacity constraints on equipment. Preferences as soft constraints.

Subtasks: Process planning to generate the temporal partial ordering of domain tasks. Incremental scheduling of resources for the domain tasks.

Procedure:

- Generate a subtask partial order according to the temporal constraints of the orders.

- Incrementally construct resource assignments according to the resource constraints, using beam search to find a good assignment according to the given preferences.

A.2.5 Case Based Reasoning For Solving Problems

This PSP is far removed from the domain specific end.

Brief Description: Case based reasoning for solving problems is to adapt solutions to similar problems.

Suitable Problems: Case based reasoning for solving problems is suitable for problems for which there are enough cases for solving them. These problems should be difficult to solve if solutions are constructed from scratch.

Preferred Representation: Cases. Case modification methods associated with the differences between cases and given problems.

Subtasks: Case retrieval. Case adaptation. (Optional: Remembering solved problems as cases.)

Procedure:

Until a solution is found for the problem or there are no more promising cases,

- Retrieve similar cases:
  - Compute an index for the given problem, and

  - Compare the given problem with cases using indexes. Select one that is similar enough.

- Adapt cases for the given problem:

  - Compute the differences between the given problem and a case,
  - Select a modification rule and modify the case,
  - Repeat until a solution is found or the case becomes unpromising, and

If a solution is found, announce success. Otherwise, announce failure.

Appendix B

Transcript of the Knowledge in the PSP Selection Program

This appendix contains a mechanical translation of the entire knowledge base of the PSP selection expert system. Each section contains knowledge about one problem perception. Each subsection contains the definition for one APSP. The questions for one problem perception are usually defined in the last APSP of that problem perception.

B.1 Classification

B.1.1 Attribute-Based Classification

```
Loading PC:LISP;DB-CLASSIFICATION.LISP.NEWEST into package QUESTION

******* Definition for Attribute Based Classification *******

ATTRIBUTE-BASED-CLASSIFICATION is to Classify according to attributes
 which is applicable if

  TYPE of PROBLEM is ATTRIBUTE-BASED-CLASSIFICATION


Paradigm ATTRIBUTE-BASED-CLASSIFICATION has 1 component: CLASSIFICATION
```

```
Component CLASSIFICATION of ATTRIBUTE-BASED-CLASSIFICATION has 2 attributes:
TYPE-REPRESENTATION and MATCHER
Its preferred representation is: Implementation specific

Consider TEMPLATE TYPE-REPRESENTATION for ATTRIBUTE-BASED-CLASSIFICATION CLASSIFICATION
 i.e., "Represent types as templates or examples to match with given objects" if
  ATTRIBUTE-BASED-CLASSIFICATION is applicable

    The the following will make TEMPLATE more favorable:

  COUNT-CLASS of ATTRIBUTE is SMALL
  COST-CLASS of ATTRIBUTE is SMALL
  COUNT-CLASS of OBJECT-TYPE is SMALL

Consider DISCRIMINATION-RULES TYPE-REPRESENTATION for ATTRIBUTE-BASED-CLASSIFICATION CLASSIFICATION
 i.e., "Represent types as discrimination rules" if
  ATTRIBUTE-BASED-CLASSIFICATION is applicable

    The the following will make DISCRIMINATION-RULES more favorable:

  COUNT-CLASS of ATTRIBUTE is not SMALL
  COST-CLASS of ATTRIBUTE is not SMALL
  COUNT-CLASS of OBJECT-TYPE is not SMALL

Consider EXACT MATCHER for ATTRIBUTE-BASED-CLASSIFICATION CLASSIFICATION
 i.e., "Object matches type specification exactly" if
  ATTRIBUTE-BASED-CLASSIFICATION is applicable

    The the following will make EXACT more favorable:

  TYPE of ATTRIBUTE is EXACT

Consider APPROXIMATE MATCHER for ATTRIBUTE-BASED-CLASSIFICATION CLASSIFICATION
 i.e., "Object matches type specification approximately" if
  ATTRIBUTE-BASED-CLASSIFICATION is applicable

    The the following will make APPROXIMATE more favorable:

  TYPE of ATTRIBUTE is not EXACT

******* Questions for Attribute-Based Classification *******

Ask "Is the problem to determine the type of a given object?" to begin with

    The expected answers are:

  "Yes" means "This is a classification problem because it is to determine the type of an object."
        TYPE of PROBLEM is CLASSIFICATION
  "No" means "nothing"

Ask "Does the object have attributes relevant to and sufficient for the decision?" when
  TYPE of PROBLEM is CLASSIFICATION

    The expected answers are:

  "Yes" means "This is a attribute-based classification problem because the attributes of
the object are sufficient for determining the type."
        TYPE of PROBLEM is ATTRIBUTE-BASED-CLASSIFICATION
  "No" means "nothing"

Ask "How many types are there?" when
  TYPE of PROBLEM is ATTRIBUTE-BASED-CLASSIFICATION

    The expected answers are:

  "(> OBJECT-TYPE-COUNT 10)" means "This problem has a large number of types."
        COUNT-CLASS of OBJECT-TYPE is not SMALL
  "(<= OBJECT-TYPE-COUNT 10)" means "This problem has a small number of types."
        COUNT-CLASS of OBJECT-TYPE is SMALL

Ask "How many attributes are there?" when
```

```
  TYPE of PROBLEM is ATTRIBUTE-BASED-CLASSIFICATION

    The expected answers are:

  "(> ATTRIBUTE-COUNT 10)" means "This problem has a large number of attributes."
        COUNT-CLASS of ATTRIBUTE is not SMALL
  "(<= ATTRIBUTE-COUNT 10)" means "This problem has a small number of attributes."
        COUNT-CLASS of ATTRIBUTE is SMALL

Ask "Will the object match the type definition exactly?" when
  TYPE of PROBLEM is ATTRIBUTE-BASED-CLASSIFICATION

    The expected answers are:

  "Yes" means "Objects match type definition exactly."
        TYPE of ATTRIBUTE is EXACT
  "No" means "Objects match type definition approximately."
        TYPE of ATTRIBUTE is not EXACT

Ask "Will the attributes expensive to measure?" when
  TYPE of PROBLEM is ATTRIBUTE-BASED-CLASSIFICATION

    The expected answers are:

  "Yes" means "The attributes are expensive to measure."
        COST-CLASS of ATTRIBUTE is not SMALL
  "No" means "The attributes are inexpensive to measure."
        COST-CLASS of ATTRIBUTE is SMALL
```

B.2 Constraint Satisfaction

B.2.1 Brelaz K-Coloring Algorithm

```
Loading PC:LISP;DB-CONSTRAINT.LISP.NEWEST into package QUESTION

******* Definitions for K-Colorability *******

K-COLORABILITY is to k-color a graph using Brelaz's algorithm
 which is applicable if

  TYPE of PROBLEM is K-COLORABILITY


Paradigm K-COLORABILITY has 1 component: K-COLORABILITY

Component K-COLORABILITY of K-COLORABILITY has 1 attribute: METHOD
Its preferred representation is: Graph

Consider BRELAZ88 METHOD for K-COLORABILITY K-COLORABILITY
 i.e., "k-color a graph using Brelaz's algorithm" if
  K-COLORABILITY is applicable
```

B.2.2 Min-Conflict Heuristic

```
******* Definitions for N-Queens Problem with a Large N *******

LARGE-N-QUEENS is to Solve the n-queens problem using the min-conflict heuristic guided repair
 which is applicable if

  TYPE of PROBLEM is LARGE-N-QUEENS


Paradigm LARGE-N-QUEENS has 1 component: LARGE-N-QUEENS

Component LARGE-N-QUEENS of LARGE-N-QUEENS has 1 attribute: METHOD
Its preferred representation is: Position vector of the queens
```

```
Consider MINTON90 METHOD for LARGE-N-QUEENS LARGE-N-QUEENS
 i.e., "Solve the n-queens problem using the min-conflict heuristic guided repair" if
  LARGE-N-QUEENS is applicable
```

B.2.3 Constraint Satisfaction on Finite Domain

```
******* Definitions for Constraint Satisfaction on Finite Discrete Variable Domains *******

CONSTRAINT-SATISFACTION-FINITE-DOMAIN is to Satisfy constraints
by searching through plausible assignments
 which is applicable if

  TYPE of PROBLEM is CONSTRAINT-SATISFACTION-FINITE-DOMAIN


Paradigm CONSTRAINT-SATISFACTION-FINITE-DOMAIN has 1 component: CONSTRAINT-SATISFACTION

Component CONSTRAINT-SATISFACTION of CONSTRAINT-SATISFACTION-FINITE-DOMAIN has 6 attributes:
SOLUTION-TESTING, FILTERING, BACKTRACKING, VARIABLE-ORDERING, VALUE-ORDERING, and ENVIRONMENT
Its preferred representation is: Variables and their domains

Consider COMPLETE-SOLUTION SOLUTION-TESTING for
CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "Testing only complete solutions in a generate and test fashion" if
  CONSTRAINT-SATISFACTION-FINITE-DOMAIN is applicable

    The the following will make COMPLETE-SOLUTION more favorable:

  SIZE-CLASS of SEARCH-SPACE is SMALL
  ARITY-CLASS of CONSTRAINT is not SMALL

Consider PARTIAL-SOLUTION SOLUTION-TESTING for
CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "Testing partial solutions incrementally in a backtracking fashion" if
  CONSTRAINT-SATISFACTION-FINITE-DOMAIN is applicable

    The the following will make PARTIAL-SOLUTION more favorable:

  SIZE-CLASS of SEARCH-SPACE is SMALL
  ARITY-CLASS of CONSTRAINT is SMALL

Consider STATIC FILTERING for CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "Constraint propagation to reduce the effective domain of variables" if
  CONSTRAINT-SATISFACTION-FINITE-DOMAIN is applicable

    The the following will make STATIC more favorable:

  STRICTNESS of CONSTRAINT is HIGH
  ARITY-CLASS of CONSTRAINT is SMALL
  COUNT-CLASS of SOLUTION is SMALL

Consider DYNAMIC FILTERING for CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "Reduce the effective domain of unassigned variables while making assignments incrementally" if
  CONSTRAINT-SATISFACTION-FINITE-DOMAIN is applicable

    The the following will make DYNAMIC more favorable:

  DENSITY of CONSTRAINT is HIGH
  COUNT-CLASS of SOLUTION is SMALL

Consider NONE FILTERING for CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "Reduce the effective domain of unassigned variables while making assignments incrementally" if
  CONSTRAINT-SATISFACTION-FINITE-DOMAIN is applicable

    The the following will make NONE more favorable:

  SIZE-CLASS of SEARCH-SPACE is SMALL
  COUNT-CLASS of SOLUTION is not SMALL
```

```
Consider CHRONOLOGICAL BACKTRACKING for CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "Retract assignments in chronological order when constraints are violated" if
  CONSTRAINT-SATISFACTION-FINITE-DOMAIN is applicable

    The the following will make CHRONOLOGICAL more favorable:

  SIZE-CLASS of SEARCH-SPACE is SMALL

Consider DEPENDENCY BACKTRACKING for CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "Retract assignments according to dependency when constraints are violated" if
  CONSTRAINT-SATISFACTION-FINITE-DOMAIN is applicable

    The the following will make DEPENDENCY more favorable:

  DETECTION of WRONG-CHOICE is DELAYED

Consider REPAIR BACKTRACKING for CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "Repair assignments using min-conflict heuristic when constraints are violated" if
  CONSTRAINT-SATISFACTION-FINITE-DOMAIN is applicable

    The the following will make REPAIR more favorable:

  COUNT-CLASS of SOLUTION is not SMALL
  RATIO-CLASS of VARIABLE-PARTICIPATION-TO-DOMAIN is not SMALL

Consider LEAST-ALTERNATIVE VARIABLE-ORDERING for
CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "Assign variables with the least number of valid assignments first
 according the current partial assignment" if
  CONSTRAINT-SATISFACTION-FINITE-DOMAIN is applicable

    The the following will make LEAST-ALTERNATIVE more favorable:

  DENSITY of CONSTRAINT is HIGH
  SIZE-CLASS of SEARCH-SPACE is not SMALL

Consider MOST-PARTICIPATION VARIABLE-ORDERING for
CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "Assign variables which involve in the most constraints first" if
  CONSTRAINT-SATISFACTION-FINITE-DOMAIN is applicable

    The the following will make MOST-PARTICIPATION more favorable:

  SIZE-CLASS of SEARCH-SPACE is not SMALL

Consider STABLE-SET-LAST VARIABLE-ORDERING for
CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "Assign variables which involves in a stable set last" if
  CONSTRAINT-SATISFACTION-FINITE-DOMAIN is applicable

    The the following will make STABLE-SET-LAST more favorable:

  SIZE-CLASS of SEARCH-SPACE is not SMALL

Consider CYCLE-CUT-SET-FIRST VARIABLE-ORDERING for
CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "Assign variables which involves in a cycle cut set last" if
  CONSTRAINT-SATISFACTION-FINITE-DOMAIN is applicable

    The the following will make CYCLE-CUT-SET-FIRST more favorable:

  SIZE-CLASS of SEARCH-SPACE is not SMALL

Consider UNSPECIFIC VARIABLE-ORDERING for
CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "Assign variables in an arbitrary order" if
  CONSTRAINT-SATISFACTION-FINITE-DOMAIN is applicable

    The the following will make UNSPECIFIC more favorable:

  SIZE-CLASS of SEARCH-SPACE is SMALL
```

```
Consider MOST-OPTIONS-FIRST VALUE-ORDERING for
CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "Assign values leaving the most options for unassigned variables first" if
  CONSTRAINT-SATISFACTION-FINITE-DOMAIN is applicable

    The the following will make MOST-OPTIONS-FIRST more favorable:

  SIZE-CLASS of SEARCH-SPACE is not SMALL

Consider UNSPECIFIC VALUE-ORDERING for CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "Assign values in an arbitrary order" if
  CONSTRAINT-SATISFACTION-FINITE-DOMAIN is applicable

    The the following will make UNSPECIFIC more favorable:

  SIZE-CLASS of SEARCH-SPACE is SMALL

Consider RETROACTIVE ENVIRONMENT for CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "The problem is to be solved interactively, and it needs the ability to
retract previous results" if
  CONSTRAINT-SATISFACTION-FINITE-DOMAIN is applicable

    The the following will make RETROACTIVE more favorable:

  TYPE of INPUT is not MONOTONIC
  TYPE of RESULT-OUTPUT is CONTINUOUS

Consider INTERACTIVE ENVIRONMENT for CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "The problem is to be solved interactively, and it needs to display
the results as each piece of input is taken into consideration" if
  TYPE of INPUT is MONOTONIC

    The the following will make INTERACTIVE more favorable:

  TYPE of RESULT-OUTPUT is CONTINUOUS

Consider BATCH ENVIRONMENT for CONSTRAINT-SATISFACTION-FINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "The problem may be solved without interaction with the user" if
  TYPE of RESULT-OUTPUT is not CONTINUOUS
```

B.2.4 Constraint Satisfaction on Infinite Domain

```
******* Definitions for Constraint Satisfaction on Infinite or Continuous Variable Domains *******

CONSTRAINT-SATISFACTION-INFINITE-DOMAIN is to Satisfy constraints by solving numerical equations
 which is applicable if

  TYPE of PROBLEM is CONSTRAINT-SATISFACTION-INFINITE-DOMAIN


Paradigm CONSTRAINT-SATISFACTION-INFINITE-DOMAIN has 1 component: CONSTRAINT-SATISFACTION

Component CONSTRAINT-SATISFACTION of CONSTRAINT-SATISFACTION-INFINITE-DOMAIN has 1 attribute: METHOD
Its preferred representation is: Solvable equations on variables

Consider SOLUTION-PROPAGATION METHOD for CONSTRAINT-SATISFACTION-INFINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "Solve the problem by finding correct assignments to some variables,
and using them to determine other variables." if
  CONSTRAINT-SATISFACTION-INFINITE-DOMAIN is applicable

    The the following will make SOLUTION-PROPAGATION more favorable:

  SOLVING-ORDER of VARIABLE is DYNAMIC
  SOLVING-ORDER of VARIABLE is STATIC

Consider RELAXATION METHOD for CONSTRAINT-SATISFACTION-INFINITE-DOMAIN CONSTRAINT-SATISFACTION
 i.e., "Solve the problem by iterative approximation using relaxation procedure
if there is one for this problem." if
  CONSTRAINT-SATISFACTION-INFINITE-DOMAIN is applicable
```

```
******* Questions for constraint-satisfaction problems *******

Ask "Is the problem to determine variable values which constrain each other?" to begin with

    The expected answers are:

  "Yes" means "This is a variable assignment problem."
        TYPE of PROBLEM is ASSIGNMENT
  "No" means "nothing"

Ask "Can we consider the problem as" when
  TYPE of PROBLEM is ASSIGNMENT

    The expected answers are:

  "Assigning different values for pairs of variables" means "This is a coloring problem."
        TYPE of PROBLEM is K-COLORABILITY
  "Placing N queens on a NxN board for N>10 without attacking each other" means
        "This is a n-queens problem with a large n."
        TYPE of PROBLEM is LARGE-N-QUEENS
  "Other assignment problem" means "This is a generic constraint satisfaction problem"
        TYPE of PROBLEM is GENERIC-ASSIGNMENT

Ask "Does each variable have a finite number of possible values?" when
  TYPE of PROBLEM is GENERIC-ASSIGNMENT

    The expected answers are:

  "Yes" means "This is a constraint satisfaction problem on finite discrete domains."
        TYPE of PROBLEM is FINITE-VARIABLE-DOMAIN
  "No" means "This is a constraint satisfaction problem on infinite or continuous domains."
        TYPE of PROBLEM is INFINITE-VARIABLE-DOMAIN

Ask "Is there an order of variables by which we can determine the variable values one by one?" when
  TYPE of PROBLEM is GENERIC-ASSIGNMENT

    The expected answers are:

  "Yes, regardless the actual variable values" means
        "The variables can be determined one by one by following a predetermineable order."
        SOLVING-ORDER of VARIABLE is STATIC
  "Yes, depending on the partial assignment" means
        "The variables can be determined one by one by following an order."
        SOLVING-ORDER of VARIABLE is DYNAMIC
  "No/Unknown" means "We cannot determine the variables one by one."
        TYPE of PROBLEM is CONSTRAINT-SATISFACTION

Ask "Do we want to see the effect immediately when each piece of input is taken into effect?" when
  TYPE of PROBLEM is CONSTRAINT-SATISFACTION-FINITE-DOMAIN

    The expected answers are:

  "Yes" means "We need to display the result incrementally"
        TYPE of RESULT-OUTPUT is CONTINUOUS
  "No" means "We may present only the final result"
        TYPE of RESULT-OUTPUT is not CONTINUOUS

Ask "Could the input change and invalidate previous result?" when
  TYPE of PROBLEM is CONSTRAINT-SATISFACTION-FINITE-DOMAIN

    The expected answers are:

  "Yes" means "Input could change and we may need to retract previous result"
        TYPE of INPUT is not MONOTONIC
  "No" means "Input will not change and we do not need to retract previous result"
        TYPE of INPUT is MONOTONIC

Ask "On the average, how many possible values each variable can have?" when
  TYPE of PROBLEM is CONSTRAINT-SATISFACTION-FINITE-DOMAIN

    The expected answers are:
```

```
  "(> VARIABLE-DOMAIN-SIZE 10)" means "Variables have large domain"
        SIZE-CLASS of VARIABLE-DOMAIN is not SMALL
  "(<= VARIABLE-DOMAIN-SIZE 10)" means "Variables have small domain"
        SIZE-CLASS of VARIABLE-DOMAIN is SMALL

Ask "How many variables are there?" when
  TYPE of PROBLEM is CONSTRAINT-SATISFACTION-FINITE-DOMAIN

    The expected answers are:

  "(> VARIABLE-COUNT 10)" means "The problem has lots of variables"
        COUNT-CLASS of VARIABLE is not SMALL
  "(<= VARIABLE-COUNT 10)" means "The problem has few variables"
        COUNT-CLASS of VARIABLE is SMALL

Ask "On the average, how many constraints does each variable participate in?" when
  TYPE of PROBLEM is CONSTRAINT-SATISFACTION-FINITE-DOMAIN

    The expected answers are:

  "(> VARIABLE-PARTICIPATION-SIZE 10)" means "Each variable participates in many constraints"
        SIZE-CLASS of VARIABLE-PARTICIPATION is not SMALL
  "(<= VARIABLE-PARTICIPATION-SIZE 10)" means "Each variable participates in few constraints"
        SIZE-CLASS of VARIABLE-PARTICIPATION is SMALL

Ask "By the best estimate, how many solutions are there?" when
  TYPE of PROBLEM is CONSTRAINT-SATISFACTION-FINITE-DOMAIN

    The expected answers are:

  "(> SOLUTION-COUNT 10)" means "The problem has many solutions"
        COUNT-CLASS of SOLUTION is not SMALL
  "(<= SOLUTION-COUNT 10)" means "The problem has few solutions"
        COUNT-CLASS of SOLUTION is SMALL

Ask "What is the maximum number of variables a constraint can relate?" when
  TYPE of PROBLEM is CONSTRAINT-SATISFACTION-FINITE-DOMAIN

    The expected answers are:

  "(< CONSTRAINT-ARITY 4)" means "Constraints have small arities"
        ARITY-CLASS of CONSTRAINT is SMALL
  "(> CONSTRAINT-ARITY 3)" means "Constraints have large arities"
        ARITY-CLASS of CONSTRAINT is not SMALL

Ask "On the average, how many ways are there to satisfy a constraint
when holding one of the involved variables constant?" when
  TYPE of PROBLEM is CONSTRAINT-SATISFACTION-FINITE-DOMAIN

    The expected answers are:

  "(> CONSTRAINT-RANGE 10)" means
        "Each constraint can be satisfied in many ways when holding one involved variable constraint"
        RANGE-CLASS of CONSTRAINT is not SMALL
  "(<= CONSTRAINT-RANGE 10)" means
        "Each constraint can be satisfied in few ways when holding one involved variable constraint"
        RANGE-CLASS of CONSTRAINT is SMALL

Ask "Are there groups of variables each of which as a group has a small number of
valid assignments but we do not know them before hand?" when
  TYPE of PROBLEM is CONSTRAINT-SATISFACTION-FINITE-DOMAIN

    The expected answers are:

  "Yes" means "Wrong assignments may not be detected immediately"
        DETECTION of WRONG-CHOICE is DELAYED
  "No/Unknown" means "Wrong assignments will be detected immediately"
        DETECTION of WRONG-CHOICE is not DELAYED

******* Some Inference Rules for Constraint Satisfaction Problems *******
```

```
CONSTRAINT-SATISFACTION-ON-FINITE-DISCRETE-DOMAINS:
  So called constraint satisfaction problems usually have finite discrete variable domains

CONSTRAINT-SATISFACTION-ON-INFINITE-DOMAINS:
  Constraint satisfaction problems with infinite variable domains need
techniques different from problems with finite discrete variable domains

SEARCH-SPACE-IS-LARGE-WHEN-LARGER-THAN-8-QUEENS:
  The search space is large compared with 8 queens problem.

CONSTRAINTS-ARE-DENSE-IF-MORE-THAN-HALF-COMPLETE:
  The constraints are dense if more than half complete

STRICT-CONSTRAINTS-ELIMINATE-HALF-OF-THE-POSSIBILITIES:
  The constraints are strict if eliminating more than half possibilities

PARTICIPATION-TO-DOMAIN-RATIO-IS-SMALL-WHEN-LESS-THAN-HALF:
  The participation to domain size ratio is small if less than 0.5.
```

B.3 Diagnosis

B.3.1 Fault-Dictionary-Based Diagnosis

```
Loading PC:LISP;DB-DIAGNOSIS.LISP.NEWEST into package QUESTION

******* Definitions for Fault Dictionary Diagnosis *******

FAULT-DICTIONARY-DIAGNOSIS is to Look up diagnosis in a fault dictionary
 which is applicable if

  TYPE of OBJECT is STRUCTURED
  TYPE of FAILURE is STRUCTURAL
  COUNT-CLASS of STRUCTURE-FAULT is not LARGE

OR

  TYPE of OBJECT is COMPOSITE
  TYPE of FAILURE is COMPONENTIAL
  COUNT-CLASS of COMPONENT-FAULT is not LARGE


Paradigm FAULT-DICTIONARY-DIAGNOSIS has 1 component: DIAGNOSIS

Component DIAGNOSIS of FAULT-DICTIONARY-DIAGNOSIS has 1 attribute: METHOD
Its preferred representation is: Diagnoses indexed by fault

Consider LOOKUP METHOD for FAULT-DICTIONARY-DIAGNOSIS DIAGNOSIS
 i.e., "Diagnose the object by looking up the faults in a fault dictionary" if
  FAULT-DICTIONARY-DIAGNOSIS is applicable
```

B.3.2 Empirical-Association-Based Diagnosis

```
******* The definition for Empirical Association Diagnosis *******

EMPIRICAL-ASSOCIATION-DIAGNOSIS is to Diagnose by empirically associating diagnosis with symptoms
 which is applicable if

  TYPE of OBSERVATION-FAILURE-ASSOCIATION is KNOWN


Paradigm EMPIRICAL-ASSOCIATION-DIAGNOSIS has 1 component: EMPIRICAL-ASSOCIATION-INFERENCE

Component EMPIRICAL-ASSOCIATION-INFERENCE of EMPIRICAL-ASSOCIATION-DIAGNOSIS has 2 attributes:
CERTAINTY and MAPPING
Its preferred representation is: Empirically associate diagnosis with symptoms

Consider CATEGORICAL CERTAINTY for
```

```
EMPIRICAL-ASSOCIATION-DIAGNOSIS EMPIRICAL-ASSOCIATION-INFERENCE
 i.e., "Diagnose the object by categorical empirical association" if
  TYPE of OBSERVATION-FAILURE-ASSOCIATION is CATEGORICAL

Consider PROBABILISTIC CERTAINTY for EMPIRICAL-ASSOCIATION-DIAGNOSIS EMPIRICAL-ASSOCIATION-INFERENCE
 i.e., "Diagnose the object by probabilistic empirical association" if
  TYPE of OBSERVATION-FAILURE-ASSOCIATION is PROBABILISTIC

Consider PREDICTIVE MAPPING for EMPIRICAL-ASSOCIATION-DIAGNOSIS EMPIRICAL-ASSOCIATION-INFERENCE
 i.e., "Diagnose the object by predictive empirical association" if
  TYPE of OBSERVATION-FAILURE-ASSOCIATION is PREDICTIVE

Consider DIAGNOSTIC MAPPING for EMPIRICAL-ASSOCIATION-DIAGNOSIS EMPIRICAL-ASSOCIATION-INFERENCE
 i.e., "Diagnose the object by diagnostic empirical association" if
  TYPE of OBSERVATION-FAILURE-ASSOCIATION is DIAGNOSTIC
```

B.3.3 Model-Based Diagnosis

```
******* The definition for Model Based Diagnosis *******

MODEL-BASED-DIAGNOSIS is to Diagnose by comparing observations and predictions based on a structure
and behavior model of the artifact, with components as the diagnosis granule
 which is applicable if

  TYPE of OBJECT is STRUCTURED
  TYPE of COMPONENT-BEHAVIOR is KNOWN
  TYPE of FAILURE is COMPONENTIAL
  TYPE of COMPONENT-FAULTY-BEHAVIOR is not KNOWN

OR

  TYPE of OBJECT is STRUCTURED
  TYPE of COMPONENT-BEHAVIOR is KNOWN
  TYPE of FAILURE is COMPONENTIAL
  TYPE of COMPONENT-FAULTY-BEHAVIOR is KNOWN
  COUNT-CLASS of COMPONENT-FAULT is LARGE

OR

  TYPE of OBJECT is STRUCTURED
  TYPE of COMPONENT-BEHAVIOR is KNOWN
  TYPE of FAILURE is STRUCTURAL
  TYPE of STRUCTURE-FAULTY-BEHAVIOR is not KNOWN

OR

  TYPE of OBJECT is STRUCTURED
  TYPE of COMPONENT-BEHAVIOR is KNOWN
  TYPE of FAILURE is STRUCTURAL
  TYPE of STRUCTURE-FAULTY-BEHAVIOR is KNOWN
  COUNT-CLASS of STRUCTURE-FAULT is LARGE


Paradigm MODEL-BASED-DIAGNOSIS has 3 components:
CANDIDATE-GENERATION, CANDIDATE-VERIFICATION, and CANDIDATE-DISCRIMINATION

Component CANDIDATE-GENERATION of MODEL-BASED-DIAGNOSIS has 1 attribute: METHOD
Its preferred representation is: Structure and behavior model of the artifact

Component CANDIDATE-VERIFICATION of MODEL-BASED-DIAGNOSIS has 1 attribute: METHOD
Its preferred representation is: Structure and behavior model of the artifact

Component CANDIDATE-DISCRIMINATION of MODEL-BASED-DIAGNOSIS has 1 attribute: METHOD
Its preferred representation is: Structure and behavior model of the artifact

******* Definitions for how to Generate Candidate *******

Consider UP-STREAM METHOD for MODEL-BASED-DIAGNOSIS CANDIDATE-GENERATION
 i.e., "Generate candidate by tracing causal path." if
  TYPE of COMPONENT-BEHAVIOR is DIRECTIONAL
```

```
    The the following will make UP-STREAM more favorable:

  TYPE of COMPONENT-BEHAVIOR is SIMPLE

Consider UP-STREAM-APPROXIMATE METHOD for MODEL-BASED-DIAGNOSIS CANDIDATE-GENERATION
 i.e., "Generate candidate by tracing only intended causal paths." if
  MODEL-BASED-DIAGNOSIS is applicable

    The the following will make UP-STREAM-APPROXIMATE more favorable:

  TYPE of COMPONENT-BEHAVIOR is SIMPLE
  TYPE of COMPONENT-BEHAVIOR is not DIRECTIONAL

Consider UP-STREAM-WITH-COMPONENT-BEHAVIOR-CONSTRAINT METHOD for
MODEL-BASED-DIAGNOSIS CANDIDATE-GENERATION
 i.e., "Generate candidate by tracing causal path considering components' behaviors." if
  TYPE of COMPONENT-BEHAVIOR is DIRECTIONAL

    The the following will make UP-STREAM-WITH-COMPONENT-BEHAVIOR-CONSTRAINT more favorable:

  TYPE of COMPONENT-BEHAVIOR is not SIMPLE

Consider CONFLICT METHOD for MODEL-BASED-DIAGNOSIS CANDIDATE-GENERATION
 i.e., "Generate candidate by finding conflicting assumptions." if
  MODEL-BASED-DIAGNOSIS is applicable

    The the following will make CONFLICT more favorable:

  TYPE of COMPONENT-BEHAVIOR is not DIRECTIONAL

******* Definitions for how to Test Candidate *******

Consider CORROBORATION METHOD for MODEL-BASED-DIAGNOSIS CANDIDATE-VERIFICATION
 i.e., "Exonerate candidates by corroborations, i.e., the matching of observations and predictions." if
  TYPE of COMPONENT-BEHAVIOR is SIMPLE
  TYPE of COMPONENT-BEHAVIOR is DIRECTIONAL

Consider FAULT-ENVISIONMENT METHOD for MODEL-BASED-DIAGNOSIS CANDIDATE-VERIFICATION
 i.e., "Verify candidates by comparing simulation with faulty components and observations." if
  TYPE of COMPONENT-FAULTY-BEHAVIOR is KNOWN

Consider DISCREPANCY-COVERING METHOD for MODEL-BASED-DIAGNOSIS CANDIDATE-VERIFICATION
 i.e., "Verify candidates by covering all discrepancies between observations and predictions." if
  MODEL-BASED-DIAGNOSIS is applicable

    The the following will make DISCREPANCY-COVERING more favorable:

  TYPE of FAILURE is not SINGLE-POINT

Consider CONSTRAINT-SUSPENSION METHOD for MODEL-BASED-DIAGNOSIS CANDIDATE-VERIFICATION
 i.e., "Verify candidates by constraint suspension." if
  TYPE of COMPONENT-BEHAVIOR is REVERSIBLE

    The the following will make CONSTRAINT-SUSPENSION more favorable:

  TYPE of FAILURE is SINGLE-POINT

Consider DISCREPANCY-CORROBORATE-COVERING METHOD for MODEL-BASED-DIAGNOSIS CANDIDATE-VERIFICATION
 i.e., "Verify candidates by covering all conflict based on both discrepancies and corroborations." if
  MODEL-BASED-DIAGNOSIS is applicable

    The the following will make DISCREPANCY-CORROBORATE-COVERING more favorable:

  TYPE of COMPONENT-BEHAVIOR is REVERSIBLE

******* Candidate Discrimination: Probe *******

Consider STRUCTURE-GUIDED-PROBE METHOD for MODEL-BASED-DIAGNOSIS CANDIDATE-DISCRIMINATION
 i.e., "Discriminate candidates by probing upstream from a discrepancy until an expected value is found.
The component with correct inputs but faulty outputs is considered as a faulty component.
Good for both single- and multiple-point-failure.  However, this method as described here and
```

```
as in Breuer76 locates only failed components thus must be modified for structure failure, e.g.,
short circuit.  I think the modification is plausible." if
  TYPE of FAILURE is STABLE
  TYPE of CANDIDATE-DISCRIMINATION is PROBING

    The the following will make STRUCTURE-GUIDED-PROBE more favorable:

  TYPE of FAILURE is COMPONENTIAL
  SPEED of PROBE is FAST

Consider PSEUDO-BINARY-PROBE METHOD for MODEL-BASED-DIAGNOSIS CANDIDATE-DISCRIMINATION
 i.e., "Discriminate candidates by taking observations affected by nearly half of the candidates." if
  TYPE of FAILURE is STABLE
  TYPE of CANDIDATE-DISCRIMINATION is PROBING

    The the following will make PSEUDO-BINARY-PROBE more favorable:

  SPEED of PROBE is FAST

Consider STATISTICS-AND-COST-GUIDED-PROBE METHOD for MODEL-BASED-DIAGNOSIS CANDIDATE-DISCRIMINATION
 i.e., "Discriminate candidates by probing at the point with the best probing-cost to
information-gain ratio." if
  TYPE of CANDIDATE-DISCRIMINATION is PROBING
  COST of PROBE is KNOWN
  OUTCOME of PROBE is KNOWN
  PROBABILITY of FAILURE is KNOWN

    The the following will make STATISTICS-AND-COST-GUIDED-PROBE more favorable:

  SPEED of PROBE is FAST

******* Candidate Discrimination: Test *******

Consider EXHAUSTIVE-TEST-GENERATION METHOD for MODEL-BASED-DIAGNOSIS CANDIDATE-DISCRIMINATION
 i.e., "Discriminate candidates by applying tests.  The tests are enumerated
and only the ones that can reduce the number of candidates are applied.
This strategy works well when the test space, i.e., the total number of
possible test patterns, is small.  Note that the test enumeration may
use domain knowledge to reduce the number of tests generated." if
  TYPE of CANDIDATE-DISCRIMINATION is TESTING

    The the following will make EXHAUSTIVE-TEST-GENERATION more favorable:

  SPEED of RANDOM-TEST-GENERATION is FAST

Consider FOCUSED-TEST-GENERATION METHOD for MODEL-BASED-DIAGNOSIS CANDIDATE-DISCRIMINATION
 i.e., "Discriminate candidates by applying tests.  The tests are generated
for focus candidates.  Each test tries to use no candidate components other
than the focus component.  This strategy works well when the test space is
large, i.e., their is a large number of possible test patterns, which is
usually the case for circuits with states.  In test generation jargon, this
strategy is a combination of line-justification to control the inputs of the
focus candidate and path sensitization to observe the outputs of the focus." if
  TYPE of CANDIDATE-DISCRIMINATION is TESTING

    The the following will make FOCUSED-TEST-GENERATION more favorable:

  SPEED of FOCUSED-TEST-GENERATION is FAST

******* Definitions for Questions
  --the Top question and the Empirical Association Diagnosis Track *******

Ask "Is the problem to determine why a given object breaks?" to begin with

    The expected answers are:

  "Yes" means "This is a diagnosis problem because it is to determine why a given object breaks."
        TYPE of PROBLEM is DIAGNOSTIC
  "No" means "nothing"

Ask "Is there any association between observations and failures?" when
  TYPE of PROBLEM is DIAGNOSTIC
```

```
    The expected answers are:

  "Yes" means "We know how observations are associated with failures."
        TYPE of OBSERVATION-FAILURE-ASSOCIATION is KNOWN
  "No" means "nothing"

Ask "What kind of association between observations and failures is it?" when
  TYPE of OBSERVATION-FAILURE-ASSOCIATION is KNOWN

    The expected answers are:

  "Categorical" means "The association between observations and failures is categorical."
        TYPE of OBSERVATION-FAILURE-ASSOCIATION is CATEGORICAL
  "Probabilistic" means "The association between observations and failures is probabilistic."
        TYPE of OBSERVATION-FAILURE-ASSOCIATION is PROBABILISTIC

Ask "On the average, how many failures can cause the same observation?" when
  TYPE of OBSERVATION-FAILURE-ASSOCIATION is KNOWN

    The expected answers are:

  "(< FAILURE-PER-SYMPTOM-COUNT 3)" means
        "The map from observation to failures is productive, i.e., each symptom is associated
        with a small number of failures."
        TYPE of OBSERVATION-FAILURE-ASSOCIATION is DIAGNOSTIC
  "(>= FAILURE-PER-SYMPTOM-COUNT 3)" means
        "We know how to predict observations given failures but the map from failures to observations
        loses information so bad that it is unproductive to map observations to failures."
        TYPE of OBSERVATION-FAILURE-ASSOCIATION is PREDICTIVE

******* Questions: the Model Based Diagnosis Track *******

Ask "Is the broken object's behavior repeatable?" when
  TYPE of PROBLEM is DIAGNOSTIC

    The expected answers are:

  "Yes" means "The fault is repeatable."
        TYPE of FAILURE is STABLE
  "No" means "The fault is not repeatable."
        TYPE of FAILURE is not STABLE

Ask "Is the object broken because of a single point of failure?" when
  TYPE of PROBLEM is DIAGNOSTIC

    The expected answers are:

  "Yes" means "We may assume that there is only a single point of failure."
        TYPE of FAILURE is SINGLE-POINT
  "No" means "We may assume that there are more than one point of failure."
        TYPE of FAILURE is not SINGLE-POINT

Ask "Does the object have distinct components?" when
  TYPE of PROBLEM is DIAGNOSTIC

    The expected answers are:

  "Yes" means "The object of the diagnosis may be divided into distinct components."
        TYPE of OBJECT is COMPOSITE
  "No" means "The object of the diagnosis may be treated as a whole."
        TYPE of OBJECT is not COMPOSITE

Ask "Do we know the pathways by which the components affect each other?" when
  TYPE of OBJECT is COMPOSITE

    The expected answers are:

  "Yes" means "We know the structure of the object.
        That is, we know how the components relate to each other."
        TYPE of OBJECT is STRUCTURED
  "No" means "We may do without the knowledge about the structure."
```

```
      TYPE of OBJECT is not STRUCTURED

Ask "Do the components pass information in one direction?" when
  TYPE of OBJECT is STRUCTURED

    The expected answers are:

  "Yes" means "The components pass information in one direction.
        We may use this property in causal analysis."
        TYPE of COMPONENT-BEHAVIOR is DIRECTIONAL
  "No" means "The components pass information in more than one direction."
        TYPE of COMPONENT-BEHAVIOR is not DIRECTIONAL

Ask "Do the components always pass information through?" when
  TYPE of OBJECT is STRUCTURED

    The expected answers are:

  "Yes" means "The components alway pass information.  We may use this property in causal analysis."
        TYPE of COMPONENT-BEHAVIOR is SIMPLE
  "No" means "nothing"

Ask "What kind of failures we need to consider?" when
  TYPE of OBJECT is COMPOSITE

    The expected answers are:

  "Break Components" means "The goal of the diagnosis is to find broken components."
        TYPE of FAILURE is COMPONENTIAL
  "Abnormal component interactions" means
        "The goal of the diagnosis is to abnormal interactions between components."
        TYPE of FAILURE is STRUCTURAL

Ask "Do we know the component behavior models?" when
  TYPE of OBJECT is COMPOSITE

    The expected answers are:

  "Yes" means "We know the behavior model of components with which
        we may predict the behavior of the object."
        TYPE of COMPONENT-BEHAVIOR is KNOWN
  "No" means "nothing"

Ask "Do we know how to compute an input of a component given the output and other inputs?" when
  TYPE of COMPONENT-BEHAVIOR is KNOWN

    The expected answers are:

  "Yes" means "We know the behavior model of components with which
        we may compute the inputs from their outputs."
        TYPE of COMPONENT-BEHAVIOR is REVERSIBLE
  "No" means "nothing"

Ask "Do we know the faulty behavior models of the components?" when
  TYPE of FAILURE is COMPONENTIAL

    The expected answers are:

  "Yes" means "We know the faulty behavior model of the components with which
        we can predict the faulty behavior of the object."
        TYPE of COMPONENT-FAULTY-BEHAVIOR is KNOWN
  "No" means "We do not know the faulty behavior of the component before hand."
        TYPE of COMPONENT-FAULTY-BEHAVIOR is not KNOWN

Ask "How many distinct faults the object can have?" when
  TYPE of COMPONENT-FAULTY-BEHAVIOR is KNOWN
  TYPE of FAILURE is SINGLE-POINT

    The expected answers are:

  "(> NUMBER-OF-FAULTS 1000000000)" means
        "There are too many component faults to put in a practical fault dictionary."
```

```
        COUNT-CLASS of COMPONENT-FAULT is LARGE
  "(<= NUMBER-OF-FAULTS 1000000000)" means "We may build a practical component fault dictionary."
        COUNT-CLASS of COMPONENT-FAULT is not LARGE

Ask "Do we know the behavior models of the abnormal interactions between components?" when
  TYPE of FAILURE is STRUCTURAL

    The expected answers are:

  "Yes" means "We know the faulty behavior model of the abnormal interactions
with which we can predict the faulty behavior of the object."
        TYPE of STRUCTURE-FAULTY-BEHAVIOR is KNOWN
  "No" means "We do not know how to model the abnormal interactions."
        TYPE of STRUCTURE-FAULTY-BEHAVIOR is not KNOWN

Ask "How many structure faults are there?" when
  TYPE of STRUCTURE-FAULTY-BEHAVIOR is KNOWN
  TYPE of FAILURE is SINGLE-POINT

    The expected answers are:

  "(> NUMBER-OF-STRUCTURE-FAULTS 1000000000)" means
        "There are too many structure faults to put in a practical fault dictionary."
        COUNT-CLASS of STRUCTURE-FAULT is LARGE
  "(<= NUMBER-OF-STRUCTURE-FAULTS 1000000000)" means
        "We may build a practical structure fault dictionary."
        COUNT-CLASS of STRUCTURE-FAULT is not LARGE

Ask "Is it possible to apply new inputs and to measure new outputs?" when
  TYPE of FAILURE is STABLE

    The expected answers are:

  "Yes" means "We may discriminate failures with tests."
        TYPE of CANDIDATE-DISCRIMINATION is TESTING
  "No" means "nothing"

Ask "Is at least one in ten randomly generated test informative?" when
  TYPE of CANDIDATE-DISCRIMINATION is TESTING

    The expected answers are:

  "Yes" means "We may discriminate failures with randomly generated tests."
        SPEED of RANDOM-TEST-GENERATION is FAST
  "No" means "We may discriminate failures with randomly generated tests only if we can afford the time."
        SPEED of RANDOM-TEST-GENERATION is not FAST
  "Unknown" means "nothing"

Ask "Can we generate an informative test in a few minutes?" when
  TYPE of CANDIDATE-DISCRIMINATION is TESTING

    The expected answers are:

  "Yes" means "We may discriminate failures with randomly generated tests."
        SPEED of FOCUSED-TEST-GENERATION is FAST
  "No" means "We may discriminate failures with randomly generated tests only if we can afford the time."
        SPEED of FOCUSED-TEST-GENERATION is not FAST

Ask "Is it possible to take internal observations?" when
  TYPE of FAILURE is STABLE

    The expected answers are:

  "Yes" means "We may discriminate failures with probes"
        TYPE of CANDIDATE-DISCRIMINATION is PROBING
  "No" means "nothing"

Ask "Can we make one probe in a few minute?" when
  TYPE of CANDIDATE-DISCRIMINATION is PROBING

    The expected answers are:
```

```
  "Yes" means "We may discriminate failures with probes"
        SPEED of PROBE is FAST
  "No" means "We may discriminate failures with probes if we have the resources"
        SPEED of PROBE is not FAST

Ask "Do we know the cost of probing?" when
  TYPE of CANDIDATE-DISCRIMINATION is PROBING

    The expected answers are:

  "Yes" means "We know the cost of probes"
        COST of PROBE is KNOWN
  "No" means "We do not know the cost of probes"
        COST of PROBE is not KNOWN

Ask "Do we know the possible outcomes of each probe?" when
  TYPE of CANDIDATE-DISCRIMINATION is PROBING

    The expected answers are:

  "Yes" means "We know the outcome of each probe."
        OUTCOME of PROBE is KNOWN
  "No" means "We do not know the outcomes of probes"
        OUTCOME of PROBE is not KNOWN

Ask "Do we know the probability of each possible failure?" when
  TYPE of CANDIDATE-DISCRIMINATION is PROBING

    The expected answers are:

  "Yes" means "We know the probability of each failure."
        PROBABILITY of FAILURE is KNOWN
  "No" means "We do not know the probability of each failure."
        PROBABILITY of FAILURE is not KNOWN
```

B.4 General PSPs

B.4.1 Disproof by Partition and Unsolvable Equation

```
Loading PC:LISP;DB-DISPROOF.LISP.NEWEST into package QUESTION

******* Definitions for Disproof by Partition and Unsolvable Equation *******

DISPROOF-BY-PARTITION-AND-UNSOLVABLE-EQUATION is to Disprove by partitioning resources
in order to create unsolvable equations
 which is applicable if

  TYPE of PROBLEM is DISPROOF-BY-PARTITION-AND-UNSOLVABLE-EQUATION


Paradigm DISPROOF-BY-PARTITION-AND-UNSOLVABLE-EQUATION has 2 components:
PARTITION-GENERATOR and EQUATION-GENERATOR

Component PARTITION-GENERATOR of DISPROOF-BY-PARTITION-AND-UNSOLVABLE-EQUATION has 1 attribute: METHOD
Its preferred representation is: Partition heuristics or cases

Component EQUATION-GENERATOR of DISPROOF-BY-PARTITION-AND-UNSOLVABLE-EQUATION has 1 attribute: METHOD
Its preferred representation is: Procedural

Consider OPERATOR-PARTITION METHOD for DISPROOF-BY-PARTITION-AND-UNSOLVABLE-EQUATION PARTITION-GENERATOR
 i.e., "Partition resource according to operators" if
  DISPROOF-BY-PARTITION-AND-UNSOLVABLE-EQUATION is applicable

    The the following will make OPERATOR-PARTITION more favorable:

  SIZE-CLASS of OPERATOR-DOMAIN is CONSTANT
  SIZE-CLASS of OPERATOR-DOMAIN is SMALL
  TYPE of PARTITION is PERFECT
```

```
Consider OPERATOR-BALANCE METHOD for DISPROOF-BY-PARTITION-AND-UNSOLVABLE-EQUATION EQUATION-GENERATOR
 i.e., "Balance resources for each partition, solve for the number of operations" if
  DISPROOF-BY-PARTITION-AND-UNSOLVABLE-EQUATION is applicable

    The the following will make OPERATOR-BALANCE more favorable:

  TYPE of PARTITION is PERFECT

Ask "Is the problem to prove that a fixed number of resources
can not be consumed by a sequence of operations?" to begin with

    The expected answers are:

  "Yes" means "This may be a disproof problem based on impossible resource consumption."
        TYPE of PROBLEM is CONSUMPTION-DISPROOF
  "No" means "nothing"

Ask "Can we partition the resources some how?" when
  TYPE of PROBLEM is CONSUMPTION-DISPROOF

    The expected answers are:

  "Yes" means "The resources can be divided into partitions and the problem may
be solved by unsolvable equations about the number of operations needed to
to balance the resources in each partition."
        TYPE of PROBLEM is DISPROOF-BY-PARTITION-AND-UNSOLVABLE-EQUATION
  "No" means "nothing"

Ask "Can we partition the resources in such a way that each operation
will affect exactly one resource from each partition?" when
  TYPE of PROBLEM is DISPROOF-BY-PARTITION-AND-UNSOLVABLE-EQUATION

    The expected answers are:

  "Yes" means "The partition is perfect in the sense that all operations affect the same
number of resources, the resources can be divided into that many partitions,
and each operation will affect exactly one resource from each partition.
Assume there are N partitions, there will be at most N! types of operations,
there will be N simultaneous equations to balance the resources in each of the
partition, and the problem will be solved if the solutions to the number of
operations will be fractional and the problem does not allow incomplete
operations."
        TYPE of PARTITION is PERFECT
        SIZE-CLASS of OPERATOR-DOMAIN is CONSTANT
  "No" means "The partition is imperfect in the sense that not all operations affect
the same number of resources, or the resources can not be divided into
partitions in such a way that and each operation will affect exactly one
resource from each partition.  Assume there are N partitions, there will be at
most N!  types of operations, there will be N simultaneous equations to
balance the resources in each of the partition, and the problem will be solved
if the solutions to the number of operations will be fractional and the
problem does not allow incomplete operations.  Since the partition is
imperfect, the equations may not be sufficiently constrained to establish the
disproof."
        TYPE of PARTITION is not PERFECT

Ask "How many resources an operation can affect?" when
  TYPE of PROBLEM is DISPROOF-BY-PARTITION-AND-UNSOLVABLE-EQUATION

    The expected answers are:

  "(< OPERATION-DOMAIN-SIZE 5)" means "The operations affects a small number of resources."
        SIZE-CLASS of OPERATOR-DOMAIN is SMALL
  "(>= OPERATION-DOMAIN-SIZE 5)" means "The operations affects a large number of resources."
        SIZE-CLASS of OPERATOR-DOMAIN is not SMALL
```

B.4.2 Generate and Test

```
Loading PC:LISP;DB-GAT.LISP.NEWEST into package QUESTION
```

```
******* Definitions for Generate and Test *******

GENERATE-AND-TEST is to Generate potential solutions then test each for validity
 which is applicable if

  TYPE of PROBLEM is GENERATE-AND-TEST


Paradigm GENERATE-AND-TEST has 2 components: GENERATOR and TESTER

Component GENERATOR of GENERATE-AND-TEST has 1 attribute: METHOD
Its preferred representation is: Procedural or declarative

Component TESTER of GENERATE-AND-TEST has 1 attribute: METHOD
Its preferred representation is: Procedural or declarative

Consider ENUMERATOR METHOD for GENERATE-AND-TEST GENERATOR
 i.e., "Generate potential solutions by enumeration" if
  TYPE of GENERATOR is COMPLETE

    The the following will make ENUMERATOR more favorable:

  REQUIREMENT of SOLUTION is OPTIMAL

Consider DEBUGGER METHOD for GENERATE-AND-TEST GENERATOR
 i.e., "Generate potential solutions by modifying previous solutions" if
  REQUIREMENT of SOLUTION is not OPTIMAL
  TYPE of TESTER is DIAGNOSTIC
  TYPE of GENERATOR is DEBUGGING

Consider HITTING METHOD for GENERATE-AND-TEST GENERATOR
 i.e., "Generate potential solutions by incomplete and hopefully more efficient enumeration" if
  REQUIREMENT of SOLUTION is not OPTIMAL

Consider PASS-FAIL METHOD for GENERATE-AND-TEST TESTER
 i.e., "Test solutions for validity" if
  TYPE of TESTER is PASS-FAIL

Consider DIAGNOSTIC METHOD for GENERATE-AND-TEST TESTER
 i.e., "Test solutions for validity and failure reason" if
  TYPE of TESTER is DIAGNOSTIC

    The the following will make DIAGNOSTIC more favorable:

  TYPE of GENERATOR is DEBUGGING

******* Questions for Generate and Test *******

Ask "Does the solutions to the problem have some conditions
that are based on complete solutions?" to begin with

    The expected answers are:

  "Yes" means "The problem has some conditions that can be verified only against complete solutions."
        TYPE of PROBLEM is AGGREGATELY-CONSTRAINED
  "No" means "nothing"

Ask "Is there a way to enumerate some potential solutions that include at least one real solution?" when
  TYPE of PROBLEM is AGGREGATELY-CONSTRAINED

    The expected answers are:

  "Yes" means "There is a solution generator that guarantees at least one solution."
        TYPE of GENERATOR is HITTING
  "No" means "nothing"

Ask "Is there a way to enumerate all potential solutions without missing any real solutions?" when
  TYPE of GENERATOR is HITTING

    The expected answers are:
```

```
  "Yes" means "There is a complete solution generator for this problem."
        TYPE of GENERATOR is COMPLETE
  "No" means "There is no know complete solution generator for this problem."
        TYPE of GENERATOR is not COMPLETE

Ask "Is there a way to verify solutions?" when
  TYPE of GENERATOR is HITTING

    The expected answers are:

  "Yes" means "There is tester for the solutions to this problem."
        TYPE of TESTER is PASS-FAIL
  "No" means "nothing"

Ask "If a potential solution fails to meet the requirement, do we know the reason?" when
  TYPE of TESTER is PASS-FAIL

    The expected answers are:

  "Yes" means "There is diagnostic tester for the solutions to this problem."
        TYPE of TESTER is DIAGNOSTIC
  "No" means "nothing"

Ask "Can we fix a solution that fails to meet the requirement given the reason of failure?" when
  TYPE of TESTER is DIAGNOSTIC

    The expected answers are:

  "Yes" means "We can generate new potential solutions by fixing old ones."
        TYPE of GENERATOR is DEBUGGING
  "No" means "nothing"

Ask "Do we need all the solutions or an optimal solution?" when
  TYPE of GENERATOR is HITTING

    The expected answers are:

  "Yes" means "We want every solution or the optimal solution."
        REQUIREMENT of SOLUTION is OPTIMAL
  "No" means "We will be satisfied with any solutions."
        REQUIREMENT of SOLUTION is not OPTIMAL

******* Inference Rules *******

GENERATE-AND-TEST-NEEDS-GENERATOR-AND-TESTER:
  We may try generate and test if we have a generator and a tester for the solutions
```

B.4.3 Order Reasoning

```
Loading PC:LISP;DB-ORDER.LISP.NEWEST into package QUESTION

******* Definition about Ordering Problem. *******

ORDERING is to Inference orders between objects
 which is applicable if

  TYPE of PROBLEM is ORDERING


Paradigm ORDERING has 1 component: ORDER

Component ORDER of ORDERING has 1 attribute: METHOD
Its preferred representation is: Partial or total order

Consider PARTIAL-ORDER METHOD for ORDERING ORDER
 i.e., "Represent the facts in a partial order trees and answer questions
 by traversing the trees." if
  ORDERING is applicable

    The the following will make PARTIAL-ORDER more favorable:
```

```
  TYPE of ORDER is not TOTAL

Consider TOTAL-ORDER METHOD for ORDERING ORDER
 i.e., "Represent the facts in a partial order trees and answer questions by traversing the trees.
Turn the trees into a linear chain when the total order is completely defined." if
  TYPE of ORDER is TOTAL

Ask "Does the problem have only one relation and is the only relation transitive?" to begin with

    The expected answers are:

  "Yes" means "This is a ordering problem because its only relation is an order relation."
        TYPE of PROBLEM is ORDERING
  "No" means "nothing"

Ask "Is the order total?" when
  TYPE of PROBLEM is ORDERING

    The expected answers are:

  "Yes" means "This is a ordering problem because its only relation is a total order relation."
        TYPE of ORDER is TOTAL
  "No/Unknown" means "This is a ordering problem because its only relation is a partial order relation."
        TYPE of ORDER is not TOTAL
```

B.4.4 Problem Reduction

```
Loading PC:LISP;DB-REDUCTION.LISP.NEWEST into package QUESTION

PROBLEM-REDUCTION is to Solve problems by dividing them into small problems
until solutions are directly known
 which is applicable if

  TYPE of PROBLEM is REDUCTIONAL


Paradigm PROBLEM-REDUCTION has 2 components: REDUCTION and SEARCH-CONTROL

Component REDUCTION of PROBLEM-REDUCTION has 2 attributes: DIVIDE and TRANSFORM
Its preferred representation is: Problem decomposition and transformation tables or rules

Component SEARCH-CONTROL of PROBLEM-REDUCTION has 4 attributes:
DIRECTION, BOOKKEEPING, ORDERING, and BACKTRACKING
Its preferred representation is: Procedural

Consider DIVIDE DIVIDE for PROBLEM-REDUCTION REDUCTION
 i.e., "Divide the problem recursively into primitive problems according to a decomposition library" if
  PROBLEM-REDUCTION is applicable

Consider TRANSFORM TRANSFORM for PROBLEM-REDUCTION REDUCTION
 i.e., "Transform problems according to a transform library" if
  TYPE of PROBLEM is TRANSFORMABLE

Ask "Can we convert the problem into a collection of primitive problems?" to begin with

    The expected answers are:

  "Yes" means "This problem can be divided into primitive problems."
        TYPE of PROBLEM is DIVIDABLE
  "No" means "nothing"

Ask "Do we know the solutions to the primitive problems?" when
  TYPE of PROBLEM is DIVIDABLE

    The expected answers are:

  "Yes" means "This problem can be solved by dividing into primitive problems."
        TYPE of PROBLEM is REDUCTIONAL
  "No" means "nothing"
```

```
Ask "Is there a way to make the right reduction or transform every time?" when
  TYPE of PROBLEM is REDUCTIONAL

    The expected answers are:

  "Yes" means "It is always possible to make the right choice, no search necessary."
        TYPE of PROBLEM is REDUCTIONAL-ALGORITHMIC
  "No" means "Need to search since it is not possible to make the right choice every time."
        TYPE of PROBLEM is REDUCTIONAL-SEARCH

Ask "On the average, how many different ways are there to divide a problem?" when
  TYPE of PROBLEM is REDUCTIONAL

    The expected answers are:

  "(< REDUCTION-METHOD-COUNT 3)" means "There is a small number of reduction methods for each problem."
        COUNT-CLASS of REDUCTION-METHOD is SMALL
  "(>= REDUCTION-METHOD-COUNT 3)" means "There is a large number of reduction methods for each problem."
        COUNT-CLASS of REDUCTION-METHOD is not SMALL

Ask "On the average, how many subproblems each reduction method can generate from one problem?" when
  TYPE of PROBLEM is REDUCTIONAL

    The expected answers are:

  "(< REDUCTION-SPLIT-FACTOR 3)" means
        "Each problem is reduced into a small number of subproblems in one step of reduction."
        SIZE-CLASS of REDUCTION-SPLIT-FACTOR is SMALL
  "(>= REDUCTION-SPLIT-FACTOR 3)" means
        "Each problem is reduced into a large number of subproblems in one step of reduction."
        SIZE-CLASS of REDUCTION-SPLIT-FACTOR is not SMALL

Ask "Can we transform one problem to another?" when
  TYPE of PROBLEM is REDUCTIONAL

    The expected answers are:

  "Yes" means "We can transform one problem to another."
        TYPE of PROBLEM is TRANSFORMABLE
  "No" means "We can not transform one problem to another."
        TYPE of PROBLEM is not TRANSFORMABLE
        SIZE of TRANSFORM-FACTOR is 1

Ask "On the average, how many transforms each problem has?" when
  TYPE of PROBLEM is TRANSFORMABLE

    The expected answers are:

  "(< TRANSFORM-FACTOR 3)" means "Each problem has a small number of transforms."
        SIZE-CLASS of TRANSFORM-FACTOR is SMALL
  "(>= TRANSFORM-FACTOR 3)" means "Each problem has a large number of transforms."
        SIZE-CLASS of TRANSFORM-FACTOR is not SMALL

BACKWARD-BRANCHING-FACTOR-FOR-PROBLEM-REDUCTION:
  Compute backward branching factor from reduction method count, split factor, and transform factor

IMPLIED-FACTS-ABOUT-SEARCH-BY-PROBLEM-REDUCTION:
  Assert known facts for problem reduction pertaining to search

ALGORITHMIC-PROBLEM-REDUCTION:
  There is an algorithm for ordering the reduction and transformation

ATTRIBUTE-CONVERSION-SEARCH-TO-PROBLEM-REDUCTION:
  Use some search paradigm components for problem reduction
```

B.4.5 Algorithmic

```
Loading PC:LISP;DB-SEARCH.LISP.NEWEST into package QUESTION
```

```
******* Definitions for Algorithmic *******

ALGORITHMIC is to Compute solutions without explicit retracting choices
 which is applicable if

  TYPE of PROBLEM is ALGORITHMIC


Paradigm ALGORITHMIC has 1 component: DETERMINE

Component DETERMINE of ALGORITHMIC has 1 attribute: METHOD
Its preferred representation is: Problem specific

Consider CALCULATE METHOD for ALGORITHMIC DETERMINE
 i.e., "Calculate each choice without search" if
  ALGORITHMIC is applicable
```

B.4.6 Search

```
******* Definitions for Search *******

SEARCH-PARADIGM is to Search solutions by making and retracting choices
 which is applicable if

  TYPE of PROBLEM is SEARCH-PARADIGM


Paradigm SEARCH-PARADIGM has 2 components: SPACE and CONTROL

Component SPACE of SEARCH-PARADIGM has 1 attribute: METHOD
Its preferred representation is: State or goal space

Component CONTROL of SEARCH-PARADIGM has 4 attributes: DIRECTION, BOOKKEEPING, ORDERING, and BACKTRACKING
Its preferred representation is: Procedural

******* Definitions for Search Space *******

Consider STATE METHOD for SEARCH SPACE
 i.e., "This is a state space search problem" if
  TYPE of SEARCH-SPACE is STATE

Consider GOAL METHOD for SEARCH SPACE
 i.e., "This is a goal space search problem" if
  TYPE of SEARCH-SPACE is GOAL

******* Definitions for the Direction of Search Control *******

Consider FORWARD DIRECTION for SEARCH CONTROL
 i.e., "Search forward starting from known facts" if
  SCOPE of FACT is ALL-KNOWN
  SIZE-CLASS of MEMORY is FORWARD-POSSIBLE
  SPECIFICATION of GOAL is not EXPLICIT

Consider FORWARD DIRECTION for SEARCH CONTROL
 i.e., "Search forward starting from known facts" if
  SCOPE of FACT is ALL-KNOWN
  SIZE-CLASS of MEMORY is FORWARD-POSSIBLE
  COUNT-CLASS of GOAL is not SMALL

Consider FORWARD DIRECTION for SEARCH CONTROL
 i.e., "Search forward starting from known facts" if
  SCOPE of FACT is ALL-KNOWN
  SIZE-CLASS of MEMORY is FORWARD-POSSIBLE
  SPECIFICATION of GOAL is EXPLICIT
  COUNT-CLASS of GOAL is SMALL
  SIZE-CLASS of BRANCHING-FACTOR is FAVOR-FORWARD

Consider BACKWARD DIRECTION for SEARCH CONTROL
 i.e., "Search backward starting from goals" if
  COUNT-CLASS of GOAL is SMALL
```

```
  SIZE-CLASS of MEMORY is BACKWARD-POSSIBLE
  SIZE-CLASS of BRANCHING-FACTOR is FAVOR-BACKWARD

Consider BACKWARD DIRECTION for SEARCH CONTROL
 i.e., "Search backward starting from goals" if
  COUNT-CLASS of GOAL is SMALL
  SIZE-CLASS of MEMORY is BACKWARD-POSSIBLE
  SCOPE of FACT is not ALL-KNOWN
  COST-CLASS of FACT is not CHEAP

Consider BIDIRECTION DIRECTION for SEARCH CONTROL
 i.e., "Search forward and backward, starting from both facts and goals" if
  SCOPE of FACT is ALL-KNOWN
  COUNT-CLASS of GOAL is SMALL
  SIZE-CLASS of MEMORY is HALF-DEPTH-SUFFICIENT

******* Definitions for the Bookkeeping of Search Control *******

Consider TREE BOOKKEEPING for SEARCH CONTROL
 i.e., "Consider the search space as a tree and do not check for duplicated states" if
  TYPE of BOOKKEEPING is TREE

Consider GRAPH BOOKKEEPING for SEARCH CONTROL
 i.e., "Consider the search space is a graph and check for duplicated states" if
  TYPE of BOOKKEEPING is GRAPH

******* Definitions for the Ordering of Search Control *******

Consider DEPTH-FIRST ORDERING for SEARCH CONTROL
 i.e., "Search depth first" if
  DEPTH-CLASS of SEARCH is BOUND

    The the following will make DEPTH-FIRST more favorable:

  REQUIREMENT of SOLUTION is OPTIMAL
  DEPTH-CLASS of SOLUTION is NEAR-SEARCH-BOUND
  SIZE-CLASS of MEMORY is DEPTH-BARELY-SUFFICIENT

Consider BREADTH-FIRST ORDERING for SEARCH CONTROL
 i.e., "Search breadth first" if
  SIZE-CLASS of MEMORY is BREADTH-SUFFICIENT

    The the following will make BREADTH-FIRST more favorable:

  REQUIREMENT of SOLUTION is OPTIMAL
  DEPTH-CLASS of SEARCH is not BOUND

Consider RANDOM ORDERING for SEARCH CONTROL
 i.e., "Search randomly" if
  DEPTH-CLASS of SEARCH is not BOUND
  SIZE-CLASS of MEMORY is UNKNOWN

    The the following will make RANDOM more favorable:

  REQUIREMENT of SOLUTION is OPTIMAL

Consider BEST-FIRST ORDERING for SEARCH CONTROL
 i.e., "Search best first" if
  REQUIREMENT of SOLUTION is not OPTIMAL
  MEASURE-CLASS of PARTIAL-SOLUTION is APPROXIMATE

Consider A* ORDERING for SEARCH CONTROL
 i.e., "A* search" if
  MEASURE-CLASS of PARTIAL-SOLUTION is ACCURATE
  MEASURE-CLASS of PARTIAL-TASK is UNDERESTIMATE

    The the following will make A* more favorable:

  REQUIREMENT of SOLUTION is OPTIMAL

Consider HILL-CLIMBING ORDERING for SEARCH CONTROL
 i.e., "Hill-climbing search" if
```

```
  REQUIREMENT of SOLUTION is not OPTIMAL
  TYPE of PROBLEM is IMPROVEMENT

Consider BRANCH-AND-BOUND ORDERING for SEARCH CONTROL
 i.e., "Branch-and-bound search" if
  MEASURE-CLASS of PARTIAL-SOLUTION is ACCURATE

    The the following will make BRANCH-AND-BOUND more favorable:

  REQUIREMENT of SOLUTION is OPTIMAL

Consider BEAM ORDERING for SEARCH CONTROL
 i.e., "Beam search" if
  REQUIREMENT of SOLUTION is not OPTIMAL
  MEASURE-CLASS of PARTIAL-SOLUTION is APPROXIMATE

    The the following will make BEAM more favorable:

  DEPTH-CLASS of SEARCH is not SMALL

Consider TAPERED ORDERING for SEARCH CONTROL
 i.e., "Tapered search" if
  REQUIREMENT of SOLUTION is not OPTIMAL
  MEASURE-CLASS of PARTIAL-SOLUTION is APPROXIMATE
  DEPTH-CLASS of SEARCH is not SMALL

    The the following will make TAPERED more favorable:

  TYPE of PROBLEM is REALTIME
  SIZE-CLASS of BRANCHING-FACTOR is SMALL

Consider MINIMAX ORDERING for SEARCH CONTROL
 i.e., "Minimax search" if
  TYPE of PROBLEM is ALTERNATE-ADVERSARY
  REQUIREMENT of SOLUTION is not OPTIMAL
  MEASURE-CLASS of PARTIAL-SOLUTION is APPROXIMATE

    The the following will make MINIMAX more favorable:

  DEPTH-CLASS of SEARCH is not SMALL
  SIZE-CLASS of BRANCHING-FACTOR-FORWARD is not SMALL

Consider ALPHA-BETA-PRUNING ORDERING for SEARCH CONTROL
 i.e., "Alpha-beta-pruning search" if
  TYPE of PROBLEM is ALTERNATE-ADVERSARY
  REQUIREMENT of SOLUTION is not OPTIMAL
  MEASURE-CLASS of PARTIAL-SOLUTION is APPROXIMATE

    The the following will make ALPHA-BETA-PRUNING more favorable:

  DEPTH-CLASS of SEARCH is not SMALL
  SIZE-CLASS of BRANCHING-FACTOR-FORWARD is not SMALL

Consider PROGRESSIVE-DEEPENING ORDERING for SEARCH CONTROL
 i.e., "Progressive-deepening search" if
  TYPE of PROBLEM is ALTERNATE-ADVERSARY
  REQUIREMENT of SOLUTION is not OPTIMAL
  MEASURE-CLASS of PARTIAL-SOLUTION is APPROXIMATE

    The the following will make PROGRESSIVE-DEEPENING more favorable:

  SIZE-CLASS of BRANCHING-FACTOR-FORWARD is not SMALL
  TYPE of PROBLEM is REALTIME
  DEPTH-CLASS of SEARCH is not SMALL

******* Definitions for the Backtracking of Search Control *******

Consider CHRONOLOGICAL BACKTRACKING for SEARCH CONTROL
 i.e., "Search with chronological backtracking" if
  DETECTION of WRONG-CHOICE is not DELAYED

Consider DEPENDENCY BACKTRACKING for SEARCH CONTROL
```

```
 i.e., "Search with dependency directed backtracking" if
  DETECTION of WRONG-CHOICE is DELAYED

******* Questions for Search *******

Ask "Is the problem to make a sequence of choices?" to begin with

    The expected answers are:

  "Yes" means "This is a choice-selection problem because it is to make a sequence of choices."
        TYPE of PROBLEM is SELECTION
  "No" means "nothing"

Ask "Is there a way to make the right choice every time?" when
  TYPE of PROBLEM is SELECTION

    The expected answers are:

  "Yes" means "It is always possible to make the right choice, no search necessary."
        TYPE of PROBLEM is ALGORITHMIC
  "No" means "Need to search since it is not possible to make the right choice every time."
        TYPE of PROBLEM is SEARCH-PARADIGM

Ask "What is the choices about?" when
  TYPE of PROBLEM is SEARCH

    The expected answers are:

  "Improving the quality of something." means
        "It is a hill climbing problem because it is to improve the quality of something.
        This is a state space search problem because it is to make a sequence of action choices."
        TYPE of PROBLEM is IMPROVEMENT
        TYPE of SEARCH-SPACE is STATE
  "Changing something in the world" means
        "This is a state space search problem because it is to make a sequence of action choices."
        TYPE of SEARCH-SPACE is STATE
  "Dividing a task into smaller pieces or accomplishing the task" means
        "This is a goal space search problem because it is to make a sequence of goal reduction choices."
        TYPE of SEARCH-SPACE is GOAL

Ask "How are the goals specified?" when
  TYPE of SEARCH-SPACE is STATE

    The expected answers are:

  "By enumerating the goals directly" means "Goal states are explicitly enumerated."
        SPECIFICATION of GOAL is EXPLICIT
  "By specifying the properties that a goal must have" means
        "Goal states are implicitly specified by the properties that a goal must have."
        SPECIFICATION of GOAL is not EXPLICIT

Ask "How many goals are specified?" when
  SPECIFICATION of GOAL is EXPLICIT

    The expected answers are:

  "(< GOAL-COUNT 10)" means "There is a small number of goals."
        COUNT-CLASS of GOAL is SMALL
  "(>= GOAL-COUNT 10)" means "There is a large number of goals."
        COUNT-CLASS of GOAL is not SMALL

Ask "How many goals are there by estimate?" when
  SPECIFICATION of GOAL is not EXPLICIT

    The expected answers are:

  "(< GOAL-COUNT 10)" means "There is a small number of goals."
        COUNT-CLASS of GOAL is SMALL
  "(>= GOAL-COUNT 10)" means "There is a large number of goals."
        COUNT-CLASS of GOAL is not SMALL

Ask "How many facts are there initially?" when
```

```
  TYPE of SEARCH-SPACE is GOAL

    The expected answers are:

  "(< FACT-COUNT 10)" means "There is a small number of initial facts."
        COUNT-CLASS of FACT is SMALL
  "(>= FACT-COUNT 10)" means "There is a large number of initial facts."
        COUNT-CLASS of FACT is not SMALL

Ask "Do we know all the facts needed to solve the problem?" when
  TYPE of SEARCH-SPACE is GOAL

    The expected answers are:

  "Yes" means "All the necessary facts are known."
        SCOPE of FACT is ALL-KNOWN
  "No" means "Need additional facts to solve the problem."
        SCOPE of FACT is not ALL-KNOWN

Ask "Is it expensive to obtain new facts?" when
  SCOPE of FACT is not ALL-KNOWN

    The expected answers are:

  "Yes" means "New facts are expensive to obtain."
        COST-CLASS of FACT is not CHEAP
  "No" means "New facts are inexpensive to obtain."
        COST-CLASS of FACT is CHEAP

Ask "On the average, how many new facts can each fact derive?" when
  TYPE of SEARCH-SPACE is GOAL

    The expected answers are:

  "(< BRANCHING-FACTOR-FORWARD 3)" means "The forward branching factor is small."
        SIZE-CLASS of BRANCHING-FACTOR-FORWARD is SMALL
  "(>= BRANCHING-FACTOR-FORWARD 3)" means "The forward branching factor is large."
        SIZE-CLASS of BRANCHING-FACTOR-FORWARD is not SMALL

Ask "On the average, how many new goals can each goal be divided up?" when
  TYPE of SEARCH-SPACE is GOAL

    The expected answers are:

  "(< BRANCHING-FACTOR-BACKWARD 3)" means "The backward branching factor is small."
        SIZE-CLASS of BRANCHING-FACTOR-BACKWARD is SMALL
  "(>= BRANCHING-FACTOR-BACKWARD 3)" means "The backward branching factor is large."
        SIZE-CLASS of BRANCHING-FACTOR-BACKWARD is not SMALL

Ask "On the average, how many possible changes are there each time when making a choice?" when
  TYPE of SEARCH-SPACE is STATE

    The expected answers are:

  "(< BRANCHING-FACTOR-FORWARD 3)" means "The forward branching factor is small."
        SIZE-CLASS of BRANCHING-FACTOR-FORWARD is SMALL
  "(>= BRANCHING-FACTOR-FORWARD 3)" means "The forward branching factor is large."
        SIZE-CLASS of BRANCHING-FACTOR-FORWARD is not SMALL

Ask "On the average, how many changes could possibly lead to a situation?" when
  TYPE of SEARCH-SPACE is STATE

    The expected answers are:

  "(< BRANCHING-FACTOR-BACKWARD 3)" means "The backward branching factor is small."
        SIZE-CLASS of BRANCHING-FACTOR-BACKWARD is SMALL
  "(>= BRANCHING-FACTOR-BACKWARD 3)" means "The backward branching factor is large."
        SIZE-CLASS of BRANCHING-FACTOR-BACKWARD is not SMALL

Ask "Do we have any idea about the maximum number of steps it will take to solve the problem?" when
  TYPE of PROBLEM is SEARCH
```

```
    The expected answers are:

  "Yes" means "The search has a known depth bound."
        DEPTH-CLASS of SEARCH is BOUND
  "No" means "The search has no known depth bound."
        DEPTH-CLASS of SEARCH is not BOUND

Ask "How many steps it will take to solve the problem by the best estimate?" when
  DEPTH-CLASS of SEARCH is BOUND

    The expected answers are:

  "(< SEARCH-DEPTH 10)" means "The search may be shallow."
        DEPTH-CLASS of SEARCH is SMALL
  "(>= SEARCH-DEPTH 10)" means "The search may be deep."
        DEPTH-CLASS of SEARCH is not SMALL

Ask "Are there many situations that will lead to themselves after a number of changes?" when
  TYPE of SEARCH-SPACE is STATE

    The expected answers are:

  "Yes" means "The search space has many loops when represented as a graph."
        TYPE of BOOKKEEPING is GRAPH
  "No" means "The search space has few loops when represented as a graph, tree may be used."
        TYPE of BOOKKEEPING is TREE

Ask "Are there many tasks that lead to themselves again after dividing them into new tasks
several times?" when
  TYPE of SEARCH-SPACE is GOAL

    The expected answers are:

  "Yes" means "The search space has many loops when represented as a graph."
        TYPE of BOOKKEEPING is GRAPH
  "No" means "The search space has few loops when represented as a graph, tree may be used."
        TYPE of BOOKKEEPING is TREE

Ask "Do we have some way to measure how expensive a partial solution is?" when
  TYPE of PROBLEM is SEARCH

    The expected answers are:

  "Yes, accurately" means "There is an accurate way to measure the cost of a partial solution."
        MEASURE-CLASS of PARTIAL-SOLUTION is ACCURATE
  "Yes, approximately" means "There is an approximate way to measure the cost of a partial solution."
        MEASURE-CLASS of PARTIAL-SOLUTION is APPROXIMATE
  "No" means "There is no way to estimates partial solution costs."
        MEASURE-CLASS of PARTIAL-SOLUTION is UNKNOWN

Ask "Do we have some way to estimate the costs of competing choices?" when
  TYPE of PROBLEM is SEARCH

    The expected answers are:

  "Yes, underestimate" means
        "There is an estimate that always underestimates the cost thus may be used for A*."
        MEASURE-CLASS of PARTIAL-TASK is UNDERESTIMATE
  "Yes, may overestimate" means "All estimates may overestimate the cost, not suitable for A*."
        MEASURE-CLASS of PARTIAL-TASK is not UNDERESTIMATE

Ask "Do we need an optimal solution to the problem?" when
  TYPE of PROBLEM is SEARCH

    The expected answers are:

  "Yes" means "The problem needs an optimal solution."
        REQUIREMENT of SOLUTION is OPTIMAL
  "No" means "The problem does not need an optimal solution."
        REQUIREMENT of SOLUTION is not OPTIMAL

Ask "Are the choices made by two adversaries taking turns?" when
```

```
  TYPE of PROBLEM is SEARCH

    The expected answers are:

  "Yes" means "The problem involves adversaries, e.g., two-person game."
        TYPE of PROBLEM is ALTERNATE-ADVERSARY
  "No" means "The problem does not involve adversaries."
        TYPE of PROBLEM is not ALTERNATE-ADVERSARY

Ask "Is there a time limit for making a choice?" when
  TYPE of PROBLEM is SEARCH

    The expected answers are:

  "Yes" means "There is a time limit for making a choice."
        TYPE of PROBLEM is REALTIME
  "No" means "There is no time limit for making a choice."
        TYPE of PROBLEM is not REALTIME

Ask "Can we detect wrong choices in time?" when
  TYPE of PROBLEM is SEARCH

    The expected answers are:

  "Yes" means "Dependency directed backtracking is unnecessary."
        DETECTION of WRONG-CHOICE is not DELAYED
  "No" means "We may need dependency directed backtracking."
        DETECTION of WRONG-CHOICE is DELAYED

******* Some inference rules for Search *******

ALL-FACTS-KNOWN-IN-STATE-SPACE-SEARCH:
  Assume all the facts are known in state space search.

ONE-INITIAL-FACT-FOR-STATE-SPACE-SEARCH:
  Assume the start state is the only initial fact in state space search.

FORWARD-POSSIBLE-1:
  State space: forward search is possible if depth first search is.

FORWARD-POSSIBLE-2:
  Goal space: forward search is possible if breadth first search is.

BACKWARD-POSSIBLE:
  Backward search is possible if depth first search is, both space types.

FORWARD-MEMORY-REQUIREMENT:
  Check if memory is sufficient for forward reasoning.

FORWARD-REQUIREMENT-ORDER-1:
  Full depth breadth-first search forward requires more memory than half-depth search.

FORWARD-REQUIREMENT-ORDER-2:
  Half depth breadth-first search forward requires more memory than depth-first search.

FORWARD-REQUIREMENT-GENERALIZATION-0:
  It is breadth-first memory sufficient if it is so for forward search.

FORWARD-REQUIREMENT-GENERALIZATION-1:
  It is Half-depth memory sufficient if half depth breadth-first search forward sufficient.

FORWARD-REQUIREMENT-GENERALIZATION-2:
  Memory is barely sufficient for depth first search if it is so in forward direction.

BACKWARD-REQUIREMENT-GENERALIZATION-0:
  It is breadth-first memory sufficient if it is so for backward search.

BACKWARD-REQUIREMENT-GENERALIZATION-1:
  It is Half-depth memory sufficient if half depth breadth-first search backward sufficient.

BACKWARD-REQUIREMENT-GENERALIZATION-2:
  Memory is barely sufficient for depth first search if it is so in backward direction.
```

BACKWARD-MEMORY-REQUIREMENT:
  Check if memory is sufficient for backward reasoning.

BACKWARD-REQUIREMENT-ORDER-1:
  Full depth breadth-first search backward requires more memory than half-depth search.

BACKWARD-REQUIREMENT-ORDER-2:
  Half depth breadth-first search backward requires more memory than depth-first search.

BRANCHING-FACTOR-COMPARISON:
  Compare forward and backward branching factors when known.

BRANCHING-FACTOR-GENERALIZATION-1:
  The branching factor of the problem is small if the forward branching factor is.

BRANCHING-FACTOR-GENERALIZATION-2:
  The branching factor of the problem is small if the backward branching factor is.

BRANCHING-FACTOR-GENERALIZATION-3:
  The branching factor of the problem is large if both backward and backward branching factors are.

ACCURATE-IS-BETTER-THAN-APPROXIMATE:
  Accurate measurement can be used where approximate one is required.

SEARCH-PARADIGM-IS-THE-SAME-AS-SEARCH-1:
  Search paradigm is the same as search.

SEARCH-PARADIGM-IS-THE-SAME-AS-SEARCH-2:
  Search paradigm component attributes are the same as search component attributes.

DEFAULT-MEMORY-SIZE-FOR-SEARCH:
  Assume 1 giga bytes memory for search

Appendix C

Selected Interactions with the PSP Selection Program

This appendix has traces of sample interaction sessions with the PSP selection expert system. There is one problem for each session. The problem is defined by the answers that the user gives to the PSP selection system. The initial prompt of the system is `->`, the command to invoke a session is `Ask`. Comments inserted when editing the traces for this appendix are in angle brackets `<>`. Questions are ended with a question mark, followed by expected answers. The selected answer is usually on the line next to the question.

C.1 Digital Circuit Diagnosis Problem

```
-> Ask <Start a session>

Is the problem to make a sequence of choices? "Yes" "No"
"No"

Can we convert the problem into a collection of primitive problems? "Yes" "No"
"No"
```

Does the solutions to the problem have some conditions that are based on complete solutions? "Yes" "No"
"No"

<Below are for the diagnosis problem. Above are questions for other problem types.>

Is the problem to determine why a given object breaks? "Yes" "No"
"Yes"

Does the object have distinct components? "Yes" "No"
"Yes"

Do we know the component behavior models? "Yes" "No"
"Yes"

Do we know how to compute an input of a component given the output and other inputs? "Yes" "No"
"Yes"

What kind of failures do we need to consider? "Break Components" "Abnormal component interactions"
"Break Components"

Do we know the faulty behavior models of the components? "Yes" "No"
"No"

Do we know the pathways by which the components affect each other? "Yes" "No"
"Yes"

Do the components always pass information through? "Yes" "No"
"No"

Do the components pass information in one direction? "Yes" "No"
"Yes"

Is the object broken because of a single point of failure? "Yes" "No"
"Yes"

Is the broken object's behavior repeatable? "Yes" "No"
"Yes"

Is it possible to take internal observations? "Yes" "No"
"No"

Is it possible to apply new inputs and to measure new outputs? "Yes" "No"
"Yes"

Can we generate an informative test in a few minutes? "Yes" "No"
"Yes"

Is at least one in ten randomly generated test informative? "Yes" "No" "Unknown"
"No"

Is there any association between observations and failures? "Yes" "No"
"Yes"

On the average, how many failures can cause the same observation? (a number) 3

What kind of association between observations and failures is it? "Categorical" "Probabilistic"
"Categorical"

<Above are questions about the diagnosis problem type. Below about others.>

Is the problem to prove that a fixed number of resources can not be consumed
by a sequence of operations? "Yes" "No"
"No"

Is the problem to determine variable values which constrain each other? "Yes" "No"
"No"

Is the problem to determine the type of a given object? "Yes" "No"
"No"

Does the problem have only one relation and is the only relation transitive? "Yes" "No"

```
"No"

<Questions end.  Recommendations begin.>

The problem may be solved by MODEL-BASED-DIAGNOSIS.
There are 3 components to MODEL-BASED-DIAGNOSIS.
The following are plausible implementations for these components:


     For CANDIDATE-GENERATION, there is 1 attribute to select -- METHOD


  The following are plausible METHOD alternatives:
Is the TYPE of COMPONENT-BEHAVIOR SIMPLE? (Yes or No) Yes <Unprepared query.>

Generate candidate by tracing causal path.
LOCALIZE used a simiar technique.         <Known systems as examples.>
This choice is rated as 0.5               <Rating of this recommendation.>
((TYPE COMPONENT-BEHAVIOR SIMPLE))        <Reason for the rating.>

Generate candidate by tracing only intended causal paths.
IDS used a simiar technique.
This choice is rated as 0.25
((TYPE COMPONENT-BEHAVIOR SIMPLE)
 (NOT (TYPE COMPONENT-BEHAVIOR DIRECTIONAL)))

Generate candidate by tracing causal path considering components' behaviors.
DART HT LOX used a simiar technique.
This choice is rated as 0.0
((NOT (TYPE COMPONENT-BEHAVIOR SIMPLE)))

Generate candidate by finding conflicting assumptions.
INTER WATSON SOPHIE GDE used a simiar technique.
This choice is rated as 0.0
((NOT (TYPE COMPONENT-BEHAVIOR DIRECTIONAL)))


     For CANDIDATE-TEST, there is 1 attribute to select -- METHOD


  The following are plausible METHOD alternatives:

Exonerate candidates by corroborations, i.e., the matching of observations and predictions.
LOCALIZE INTER SOPHIE used a simiar technique.
This choice is rated as 0.25
()

Verify candidates by covering all discrepancies between observations and predictions.
DART Ginsberg84 used a simiar technique.
This choice is rated as 0.0
((NOT (TYPE FAILURE SINGLE-POINT)))

Verify candidates by constraint suspension.
HT LOX used a simiar technique.
This choice is rated as 0.5
((TYPE FAILURE SINGLE-POINT))

Verify candidates by covering all conflict based on both discrepancies and corroborations.
GDE used a simiar technique.
This choice is rated as 0.5
((TYPE COMPONENT-BEHAVIOR REVERSIBLE))


     For CANDIDATE-DISCRIMINATION, there is 1 attribute to select -- METHOD


  The following are plausible METHOD alternatives:

Discriminate candidates by applying tests.  The tests are enumerated
and only the ones that can reduce the number of candidates are applied.
This strategy works well when the test space, i.e., the total number of
possible test patterns, is small.  Note that the test enumeration may
```

use domain knowledge to reduce the number of tests generated.
DART used a simiar technique.
This choice is rated as 0.0
((SPEED RANDOM-TEST-GENERATION FAST))

Discriminate candidates by applying tests. The tests are generated
for focus candidates. Each test tries to use no candidate components other
than the focus component. This strategy works well when the test space is
large, i.e., their is a large number of possible test patterns, which is
usually the case for circuits with states. In test generation jargon, this
strategy is a combination of line-justification to control the inputs of the
focus candidate and path sensitization to observe the outputs of the focus.
Shirley83 used a simiar technique.
This choice is rated as 0.5
((SPEED FOCUSED-TEST-GENERATION FAST))

<Second APSP recommendation for the same problem.>
The problem may be solved by EMPIRICAL-ASSOCIATION-DIAGNOSIS.
There is 1 component to EMPIRICAL-ASSOCIATION-DIAGNOSIS.
The following are plausible implementations for these components:

For EMPIRICAL-ASSOCIATION-INFERENCE, there are 2 attributes to select -- CERTAINTY and MAPPING

The following are plausible CERTAINTY alternatives:

Diagnose the object by categorical empirical association
This choice is rated as 0.25
()

The following are plausible MAPPING alternatives:

Diagnose the object by predictive empirical association
DENDRAL used a simiar technique.
This choice is rated as 0.25
()

C.2 The Think and Jump Problem

-> Ask

Is the problem to make a sequence of choices? "Yes" "No"
"No"

Can we convert the problem into a collection of primitive problems? "Yes" "No"
"No"

Does the solutions to the problem have some conditions that are based on complete solutions? "Yes" "No"
"No"

Is the problem to determine why a given object breaks? "Yes" "No"
"No"

<Questions relevant to the T&J problem begin.>

Is the problem to prove that a fixed number of resources can not be consumed
by a sequence of operations? "Yes" "No"
"Yes"

Can we partition the resources some how? "Yes" "No"
"Yes"

How many resources an operation can affect? (a number) 3

Can we partition the resources in such a way that each operation
will affect exactly one resource from each partition? "Yes" "No"

```
"Yes"

<Questions irrelevant to the T&J problem begin.>

Is the problem to determine variable values which constrain each other? "Yes" "No"
"No"

Is the problem to determine the type of a given object? "Yes" "No"
"No"

Does the problem have only one relation and is the only relation transitive? "Yes" "No"
"No"

<Recommendations beging.>

The problem may be solved by DISPROOF-BY-PARTITION-AND-UNSOLVABLE-EQUATION.
There are 2 components to DISPROOF-BY-PARTITION-AND-UNSOLVABLE-EQUATION.
The following are plausible implementations for these components:


     For PARTITION-GENERATOR, there is 1 attribute to select -- METHOD


  The following are plausible METHOD alternatives:

Partition resource according to operators
Mutilated Checker Board Think and Jump used a simiar technique.
This choice is rated as 1.0
((SIZE-CLASS OPERATOR-DOMAIN CONSTANT)
 (SIZE-CLASS OPERATOR-DOMAIN SMALL)
 (TYPE PARTITION PERFECT))


     For EQUATION-GENERATOR, there is 1 attribute to select -- METHOD


  The following are plausible METHOD alternatives:

Balance resources for each partition, solve for the number of operations
Mutilated Checker Board Think and Jump used a simiar technique.
This choice is rated as 0.5
((TYPE PARTITION PERFECT))
```

C.3 The 8 Queens Problem as a Search Problem

The problem for the following trace is the 8-queens problem. The knowledge base has knowledge about search and diagnosis related APSPs. This session was used to test the knowledge about the search APSPs.

```
-> Ask

Is the problem to determine why a given object breaks? "Yes" "No"
"No"

<Questions about search APSPs begin.>

Is the problem to make a sequence of choices? "Yes" "No"
"Yes"

Is there a way to make the right choice every time? "Yes" "No"
"No"

Can we detect wrong choices in time? "Yes" "No"
"No"

Is there a time limit for making a choice? "Yes" "No"
"No"
```

```
Are the choices made by two adversaries taking turns? "Yes" "No"
"No"

Do we need an optimal solution to the problem? "Yes" "No"
"No"

Do we have some way to estimate the costs of competing choices? "Yes, underestimate"
                                                                "Yes, may overestimate"
"Yes, underestimate"

Do we have some way to measure how expensive a partial solution is?
"Yes, accurately" "Yes, approximately" "No"
"Yes, accurately"

Do we have any idea about the maximum number of steps it will take to solve the problem? "Yes" "No"
"Yes"

How many steps it will take to solve the problem by the best estimate? (a number) 8


What is the choices about? "Improving the quality of something."
                           "Changing something in the world"
                           "Dividing a task into smaller pieces or accomplishing the task"
"Changing something in the world"

Are there many situations that will lead to themselves after a number of changes? "Yes" "No"
"No"

On the average, how many changes could possibly lead to a situation? (a number) 1


On the average, how many possible changes are there each time when making a choice? (a number) 4


How are the goals specified? "By enumerating the goals directly"
                             "By specifying the properties that a goal must have"
"By specifying the properties that a goal must have"

How many goals are there by estimate? (a number) 10


<Recommendations begin.>

The problem may be solved by SEARCH.  There are 2 components to SEARCH.
The following are plausible implementations for these components:


     For SPACE, there are 1 attribute to select:


  For METHOD, the following are plausible alternatives:

This is a state space search problem
This choice is rated as 0.25
()


     For CONTROL, there are 4 attributes to select:


  For DIRECTION, the following are plausible alternatives:

Search forward starting from known facts
JVB's Families Problem used a simiar technique.
This choice is rated as 0.25
()


  For BOOKKEEPING, the following are plausible alternatives:

Consider the search space as a tree and do not check for duplicated states
```

```
This choice is rated as 0.25
()


  For ORDERING, the following are plausible alternatives:
Is the DEPTH of SOLUTION NEAR-SEARCH-BOUND? (Yes or No) Yes
Is the SIZE of MEMORY DEPTH-BARELY-SUFFICIENT? (Yes or No) Yes


Search depth first
This choice is rated as 0.5
((REQUIREMENT SOLUTION OPTIMAL)
 (DEPTH SOLUTION NEAR-SEARCH-BOUND)
 (SIZE MEMORY DEPTH-BARELY-SUFFICIENT))


Search breadth first
This choice is rated as -0.25
((REQUIREMENT SOLUTION OPTIMAL)
 (NOT (DEPTH SEARCH BOUND)))

Search best first
This choice is rated as 0.25
()

A* search
This choice is rated as 0.0
((REQUIREMENT SOLUTION OPTIMAL))

Branch-and-bound search
This choice is rated as 0.0
((REQUIREMENT SOLUTION OPTIMAL))

Beam search
This choice is rated as 0.0
((NOT (DEPTH SEARCH SMALL)))


  For BACKTRACKING, the following are plausible alternatives:

Search with dependency backtracking
This choice is rated as 0.25
()
```

C.4 The 8 Queens Problem as a Constraint Satisfaction Problem

The problem for the following trace is the 8-queens problem. The knowledge base has knowledge about constraint satisfaction APSPs. This session was used to test the knowledge about the constraint satisfaction APSPs.

```
-> Ask

Is the problem to determine variable values which constrain each other? "Yes" "No"
"Yes"

Can we consider the problem as "Assigning different values for pairs of variables"
                               "Placing N queens on a NxN board for N>10
                                without attacking each other"
                               "Other assignment problem"
"Other assignment problem"

Does each variable have a finite number of possible values? "Yes" "No"
"Yes"

Is there an order of variables by which we can determine the variable values one by one?
```

```
                                "Yes, regardless the actual variable values"
                                "Yes, depending on the partial assignment"
                                "No/Unknown"
"No/Unknown"

Do we want to see the effect immediately when each piece of input is taken into effect? "Yes" "No"
"No"

Could the input change and invalidate previous result? "Yes" "No"
"No"

On the average, how many possible values each variable can have? (a number) 8


How many variables are there? (a number) 8


On the average, how many constraints does each variable participate in? (a number) 7


By the best estimate, how many solutions are there? (a number) 4


What is the maximum number of variables a constraint can relate? (a number) 2


On the average, how many ways are there to satisfy a constraint
when holding one of the involved variables constant? (a number) 5


Are there groups of variables each of which as a group has a small number of
valid assignments but we do not know them before hand? "Yes" "No/Unknown"
"Yes"



The problem may be solved by CONSTRAINT-SATISFACTION-FINITE-DOMAIN.
There is 1 component to CONSTRAINT-SATISFACTION-FINITE-DOMAIN.
The following are plausible implementations for these components:


     For CONSTRAINT-SATISFACTION, there are 6 attributes to select --
     SOLUTION-TESTING, FILTERING, BACKTRACKING, VARIABLE-ORDERING,
     VALUE-ORDERING, and ENVIRONMENT


  The following are plausible SOLUTION-TESTING alternatives:

Testing partial solutions incrementally in a backtracking fashion
kumar87 used a simiar technique.
This choice is rated as 0.75
((SIZE-CLASS SEARCH-SPACE SMALL)
 (ARITY-CLASS CONSTRAINT SMALL))

Testing only complete solutions in a generate and test fashion
This choice is rated as 0.25
((SIZE-CLASS SEARCH-SPACE SMALL)
 (NOT (ARITY-CLASS CONSTRAINT SMALL)))


  The following are plausible FILTERING alternatives:

Reduce the effective domain of unassigned variables while making assignments incrementally
forechecking, n-queens, mcgreger79 used a simiar technique.
This choice is rated as 0.25
((SIZE-CLASS SEARCH-SPACE SMALL)
 (NOT (COUNT-CLASS SOLUTION SMALL)))

Reduce the effective domain of unassigned variables while making assignments incrementally
forechecking, n-queens, mcgreger79 used a simiar technique.
This choice is rated as 0.75
((DENSITY CONSTRAINT HIGH)
```

```
 (COUNT-CLASS SOLUTION SMALL))

Constraint propagation to reduce the effective domain of variables
waltz line drawing labeling algorithm arc consistency kumar92 used a simiar technique.
This choice is rated as 0.5
((STRICTNESS CONSTRAINT HIGH)
 (ARITY-CLASS CONSTRAINT SMALL)
 (COUNT-CLASS SOLUTION SMALL))


  The following are plausible BACKTRACKING alternatives:

Repair assignments using min-conflict heuristic when constraints are violated
minton90 used a simiar technique.
This choice is rated as 0.25
((NOT (COUNT-CLASS SOLUTION SMALL))
 (NOT (RATIO-CLASS VARIABLE-PARTICIPATION-TO-DOMAIN SMALL)))

Retract assignments according to dependency when constraints are violated
This choice is rated as 0.5
((DETECTION WRONG-CHOICE DELAYED))

Retract assignments in chronological order when constraints are violated
This choice is rated as 0.5
((SIZE-CLASS SEARCH-SPACE SMALL))


  The following are plausible VARIABLE-ORDERING alternatives:

Assign variables in an arbitrary order
This choice is rated as 0.5
((SIZE-CLASS SEARCH-SPACE SMALL))

Assign variables which involves in a cycle cut set last
dechter86 used a simiar technique.
This choice is rated as 0.0
((NOT (SIZE-CLASS SEARCH-SPACE SMALL)))

Assign variables which involves in a stable set last
freuder85 used a simiar technique.
This choice is rated as 0.0
((NOT (SIZE-CLASS SEARCH-SPACE SMALL)))

Assign variables which involve in the most constraints first
This choice is rated as 0.0
((NOT (SIZE-CLASS SEARCH-SPACE SMALL)))

Assign variables with the least number of valid assignments first
according the current partial assignment
forechecking, n-queens, mcgreger79 bitner75 used a simiar technique.
This choice is rated as 0.0
((DENSITY CONSTRAINT HIGH)
 (NOT (SIZE-CLASS SEARCH-SPACE SMALL)))


  The following are plausible VALUE-ORDERING alternatives:

Assign values in an arbitrary order
This choice is rated as 0.5
((SIZE-CLASS SEARCH-SPACE SMALL))

Assign values leaving the most options for unassigned variables first
haralick80 dechter88 used a simiar technique.
This choice is rated as 0.0
((NOT (SIZE-CLASS SEARCH-SPACE SMALL)))


  The following are plausible ENVIRONMENT alternatives:

The problem may be solved without interaction with the user
This choice is rated as 0.25
()
```

```
The problem is to be solved interactively, and it needs to display
the results as each piece of input is taken into consideration
This choice is rated as 0.0
((TYPE RESULT-OUTPUT CONTINUOUS))

The problem is to be solved interactively, and it needs the ability to
retract previous results
This choice is rated as -0.25
((NOT (TYPE INPUT MONOTONIC))
 (TYPE RESULT-OUTPUT CONTINUOUS))
```

Bibliography

[Amarel, 1968] Saul Amarel. On representations of problems of reasoning about actions. In *Machine Intelligence 3*, chapter 10. 1968.

[Barr *et al.*, 1981 1989] Avron Barr, Paul R. cohen, and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence*, volume 1-4. Vols 1-3: William Kaufmann, Inc. Vol 4: Addison-Wesley, 1981-1989.

[Brelaz, 1988] D. Brelaz. Almost all k-colorable graphs are easy to color. *Journal of Algorithms*, 9:63–82, 1988.

[Bundy, 1990] Alan Bundy. *Catalogue of Artificial Intelligence Techniques, Third, Revised Edition.* Springer-Verlag, 1990.

[Carbonell, 1985] Jaime G. Carbonell. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. Technical Report CMU-CS-85-115, Carnegie-Mellon University, March 1985.

[Chandrasekaran, 1986] B. Chandrasekaran. Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design. *IEEE Expert*, 1(3):23–30, 1986.

[Davis and Hamscher, 1988] Randall Davis and Walter C. Hamscher. Model-based reasoning: Troubleshooting. In H. E. Shrobe, editor, *Exploring Artificial Intelligence.* MKP, 1988. Also MIT A.I. Memo 1059.

[Davis, 1989] Randall Davis. Form and content in model-based reasoning. In *Proceedings IJCAI-89 Workshop on Model-Based Reasoning*, August 1989.

[de Kleer *et al.*, 1977] Johan de Kleer, Jon Doyle, Charles Rich, Guy L. Steele, Jr., and Gerald J. Sussman. AMORD — A Deductive Procedure System. MIT AI Lab Working Paper 151, August 1977.

[de Kleer *et al.*, 1979] Johan de Kleer, Jon Doyle, Guy L. Steele, and Gerald J. Susman. Explicit control of reasoning. In Patrick Hery Winston and Richard Henry Brown, editors, *Artificial Intelligence: An MIT Perspective*, pages 95–116. MIT Press, 1979.

[de Kleer, 1977] Johan de Kleer. Multiple representations of knowledge in a mechanics problem-solver. In *Proceedings of IJCAI-77*, pages 299–304. IJCAI, 1977.

[Gentner, 1987] Dedre Gentner. Mechanisms of analogical learning. Technical Report UILU-ENG-87-1770, University of Illinois at Urbana-Champaign, October 1987.

[Hall, 1989] Roges P. Hall. Computational approaches to analogical reasoning: A comparative analysis. *Artificial Intelligence*, 1(39):39–120, May 1989. Received Nov 1986, Revised Dec 1987.

[Kaplan and Simon, 1990] Craig A. Kaplan and Herbert A. Simon. In search of insight. *Cognitive Psychology*, (22):374–419, 1990.

[Korf, 1980] Richard E. Korf. Toward a model of representation changes. *Artificial Intelligence*, 14(1):41–78, 1980.

[Korf, 1985] Richard E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77, 1985.

[Korf, 1988] R. E. Korf. Search: A survey of recent results. In H. E. Shrobe, editor, *Exploring Artificial Intelligence*, pages 197–237. Morgan Kaufmann, 1988.

[Kowalski, 1979] R. Kowalski. *Logic for problem solving.* American Elsevier, New York, 1979.

[Lenat and Guha, 1989] Douglas B. Lenat and R. V. Guha. *Building Large Knowledge-Based Systems – Representation and Inference in the Cyc Project.* Addison-Wesley, Reading, Massachusetts, 1989.

[Loveland and Stickel, 1973] D. Loveland and M. Stickel. A hole in goal trees: Some guidance from resolution theory. In *IJCAI 3*, pages 153–161. IJCAI, 1973.

[Lowry, 1987a] Michael R. Lowry. The abstraction/implemenation model of problem reformulation. In *IJCAI-87*, pages 1004–1010, 1987.

[Lowry, 1987b] Michael R. Lowry. Algorithm synthesis through problem reformulation. In *AAAI-87*, pages 432–436, 1987.

[McDermott, 1988] John McDermott. Preliminary steps toward a taxonomy of problem-solving methods. In Sandra Marcus, editor, *Automating Knowledge Acquisition for Expert Systems*, chapter 8. Kluwer Academic Publishers, 1988.

[Minton *et al.*, 1990] S. Minton, M. Johnston, A. Phillips, and P. Laird. Solving large-scale constraint-satisfaction and scheduling problems using a heuristic repair method. In *Proceedings to the Eighth National Conference on Artificial Intelligence (AAAI90)*, pages 17–24. AAAI, 1990.

[Nayak *et al.*, 1991] P. Pandurang Nayak, Leo Joskowicz, and Sanjaya Addanki. Automated model selectioin using context-dependent behaviors. In *Proceedings of the Fifth International Workshop on Qualitative Reasoning about Physical Systems*, 1991.

[Newell, 1966] Allen Newell. On the representations of problems. In *Annual research review*. 1966.

[Newell, 1983] Allen Newell. The heuristic of George Polya and its relation to artificial intelligence. In Rudolf Groner, Marina Groner, and Walter F. Bischof, editors, *Methods of Heuristics*, chapter 12. Lawrence Erlbaum Associates, Publishers, 1983.

[Quinlan, 1983] J. Ross Quinlan. Learning efficient classification procedures and their application to chess end games. In R. S. Michalski, J. G. Carbonell, and T. M. Michell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 463–482. Tioga, 1983.

[Rich and Knight, 1991] Elaine Rich and Kevin Knight. *Artificial Intelligence, Second Edition*. McGraw-Hill Book Company, 1991.

[Riddle, 1986] Patricia J. Riddle. A Proposal for Automating Shifts of Representation. Technical Report LCSR-TR-84, Rutgers University, February 1986.

[Riesbeck and Roger C, 1989] Christopher K. Riesbeck and Schank Roger C. *Inside Case-Based Reasoning*. LEA, 1989.

[Smith and Genesereth, 1985] David E. Smith and Michael R. Genesereth. Ordering conjunctive queries. *AIJ*, (26):171–215, 1985.

[Smith *et al.*, 1986] David E. Smith, Michael R. Genesereth, and Matthew L. Ginsberg. Controlling recursive inference. *Artificial Intelligence*, (30):343–389, 1986.

[Smith, 1989] David E. Smith. Controlling backward inference. *Artificial Intelligence*, (39):145–208, 1989.

[Stallman and Sussman, 1977] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[Steier and Newell, 1988] David Steier and Allen Newell. Integrating multiple sources of knowledge into designer-soar, an automatic algorithm designer. In *Proceedings to the Seventh National Conference on Artificial Intelligence (AAAI88)*. AAAI, August 1988.

[Stuart C. Shapiro *et al.*, 1990] Editor-in-Chief Stuart C. Shapiro, Managing Editor David Eckroth, and Developmental Editor George A. Vallasi, editors. *Encyclopedia of Artificial Intelligence*. Wiley-Interscience Publication, John Wiley & Sons, 1990.

[Subramanian and Genesereth, 1987] Devika Subramanian and Michael R. Genesereth. The relevance of irrelevance. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 416–422. IJCAII, August 1987.

[Subramanian, 1989] Devika Subramanian. A theory of justified reformulations. Technical Report STAN-CS-89-1260, Dept. of Computer Science, Stanford University, March 1989. Ph.D. Thesis.

[Szolovits and Pauker, 1978] Peter Szolovits and Stephen G. Pauker. Categorical and probabilistic reasoning in medical diagnosis. *Artificial Intelligence*, 11(1,2):115–144, August 1978.

[Treitel and Genesereth, 1987] Richard Treitel and Michael R. Genesereth. Choosing directions for rules. *Jaurnal of Automated Reasoning*, (2):395–431, 1987.

[Ulrich, 1988] Karl T. Ulrich. *Computation and Pre-Parametric Design.* PhD thesis, MIT, July 1988. Also: MIT AI Lab TR1043.

[Van Baalen and Davis, 1988] Jeffrey Van Baalen and Randall Davis. Overview of an Approach to Representation Design. In *Proc. of AAAI-88.* AAAI, August 1988.

[Van Baalen, 1989] Jeffrey Van Baalen. *Steps Towards a Theory of Representation Design.* PhD thesis, Massachusetts Institute of Technology, January 1989. Also available as MIT AI-TR 1128.

[Weber, 1983] Karl Weber. *How to Prepare for the New Law School Admission Test.* Harcourt Brace Jovanovich, Publishers, 1983.

[Weld, 1989] Daniel S. Weld. Automated model switching: Discrepancy driven selection of approximation reformulations. Technical Report Technical report 89-08-01, Department of Computer Science and Engineering, University of Washington, October 1989.

[Weld, 1992] Daniel S. Weld. Reasoning about model accuracy. *AIJ*, (56):255–300, 1992.

[Williams, 1985] Brain C. Williams. Circumscribing circumscription: A guide to relevance and incompleteness. Technical Report A.I. Memo No. 868, MIT, October 1985.

[Winston, 1992] Patrick Henry Winston. *Artificial Intelligence, Third Edition.* Addison-Wesley, 1992.

[Wos, 1988] Larry Wos. *AUTOMATED REASONING: 33 BASIC RESEARCH PROBLEMS.* Prentice Hall, Englewood Cliffs, Nes Jersey 07632, 1988.

[Wu, 1990] Thomas D. Wu. Efficient diagnosis of multiple disorders based on a symptom clustering approarch. In *Proceedings to the Eighth National Conference on Artificial Intelligence (AAAI90)*, pages 357–364. AAAI, July 1990.