Modeling Intelligent Control of Distributed Cooperative Inferencing

Edward M. Williams

AFIT/DS/ENG/97-03

Modeling Intelligent Control of Distributed Cooperative Inferencing

DISSERTATION

Edward Michael Williams
Major, USAF

AFIT/DS/ENG/97-03

AFIT/DS/ENG/97-03

Modeling Intelligent Control of Distributed Cooperative Inferencing

DISSERTATION

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

Edward Michael Williams, BS, MS

Major, USAF

September, 1997

AFIT/DS/ENG/97-03

Modeling Intelligent Control of
Distributed Cooperative Inferencing

Edward Michael Williams, BS, MS

Major, USAF

Approved:

[illegible]	8/15/97
Dr. Eugene Santos Jr.	Date
[illegible]	8/15/97
Dr. Sheila Banks	Date
[illegible]	8/15/97
Dr. Gary Lamont	Date
Mark Oxley	8/15/97
Dr. Mark Oxley	Date
[illegible]	8/15/97
Dr. Meir Pachter (Dean's Representative)	Date

[illegible]

Robert A. Calico, Jr
Dean

Acknowledgements

I would like to thank my advisor, Dr. Eugene Santos, for his insight and guidance throughout this effort. I am also grateful for the generous contributions made by the rest of my committee: Dr. Sheila Banks for her help in making sure this document was finished and submitted in a timely fashion; Dr. Mark Oxley for his crash course in control theory and Dr. Gary Lamont for his advice during my doctoral research and as my advisor for my Master's degree.

For allowing me to be "absent while present" for the past three years, I thank my wife Jennifer and my children Michael, Matthew, Jessica, Kristina, Joshua and the most recent addition: Alyssa. Having a husband and father who is there, but just out of reach is a frustrating experience; your sacrifice is appreciated more than you know. Your unconditional love gave me the strength I needed to keep moving.

Finally, I thank the Lord for giving me the stamina to finish the course, for giving my family and friends the patience to endure my absentmindedness and for providing a loving and supporting church family to keep us going.

Edward Michael Williams

Table of Contents

List of Figures

AFIT/DS/ENG/97-03

Abstract

The ability to harness different problem-solving methods together into a cooperative system has the potential for significantly improving the performance of systems for solving NP-hard problems. The need exists for an intelligent controller that is able to effectively combine radically different problem-solving techniques with *anytime* and *anywhere* properties into a distributed, cooperative environment. This controller requires models of the component algorithms in conjunction with feedback from those algorithms during run-time to manage a dynamic combination of tasks effectively. This research develops a domain-independent method for creating these models as well as a model for the controller itself. These models provide the means for the controller to select the most appropriate algorithms, both initially and during run-time. We overcome the algorithm selection limitations underlying the No Free Lunch Theorems by utilizing the algorithm performance knowledge contained in the algorithm models to aid in the selection process. This methodology is applicable to many NP-hard problems; applicability is only limited by the availability of anytime and anywhere algorithms for that domain. We demonstrate the capabilities of this methodology by applying it to a known NP-hard problem: uncertain inference over Bayesian Networks. Experiments using a collection of randomly generated networks and some common inference algorithms showed very promising results. Future directions for this research could involve the analysis of the impact of the accuracy of the algorithm models on the performance of the controller; the issue is whether the increased model accuracy would cause excessive system overhead, counteracting the potential increase in performance due to better algorithm selection. Research could also be done to allow the controller to learn the performance characteristics of the algorithms, providing another method for improving the selection accuracy.

Modeling Intelligent Control of Distributed Cooperative Inferencing

I. Introduction

Uncertain inference using Bayesian Networks is NP-hard(5, 6, 31); however, it does not change the fact that Bayesian Networks are a common way to represent uncertain knowledge. Example uses include the commercial system *Intellipath* (formerly called *Pathfinder* (18)), a pathological diagnosis system for tissue samples, and power generator real-time monitoring (23, 25). Uncertainty is inevitable in any real-world knowledge base, and it makes the already difficult task of reasoning over the knowledge base even more difficult.

This leaves a difficult task (computationally) without any general-purpose algorithm to efficiently solve any problem instance (as discussed by Wolpert and Macready in their *No Free Lunch Theorems*(38)). However, algorithms are available to solve certain specific subclasses of these problems very efficiently. In order to make uncertain inference computationally feasible it is necessary to maximize the utility of the inferencing process by using the method best suited to the specific network involved. Unfortunately, determining the best algorithm to use for a given network can be just as difficult as solving the inferencing problem itself.

With the increasing use of large networks of personal workstations, the available computing power is no longer limited to a single computer. Harnessing the power of more than one computer (possibly even the entire network) into a cooperative problem solving environment would create a significant resource for working these difficult problems.

This thinking led to the creation of OVERMIND(30), a distributed architecture for inferencing. The OVERMIND architecture allows the use of different inferencing methods in parallel, controlled by a separate task with the capability to start and stop methods as required. The controller evaluates the target network and selects the initial set of inferencing methods to use. During execution, the controller monitors the system's progress, taking actions as needed to increase the overall performance of the system. In this fashion, the controller makes better use of the available resources.

For the controller to make a determination of which inferencing methods to use, a model of both the methods' overall performance for a given class of networks and the methods' run-time performance profiles(41) is required. The model allows the controller to choose appropriate methods to begin the inferencing process as well as decide when to change methods in order to converge more rapidly towards the optimal solution. The models for the different inference algorithms are created (where possible) using analytical methods. The goal is to develop easily calculated approximate models to minimize the computational overhead for the controller. In addition, by specifying the models at a level of abstraction that does not require domain-specific knowledge, the distributed control technique is generalizable to any problem domain requiring optimization, such as scheduling or routing.

1.1 Goals and Scope

This goal of this research is to develop a domain-independent model for the operation of an intelligent controller for a distributed cooperative system. Accomplishing this goal requires three objectives:

1. formulation of models of the algorithms' behavior;
2. development of a method for selecting algorithm combinations given a specific problem instance;
3. development of a method for evaluating algorithm performance during run-time and deciding which algorithm(s) should be used to continue processing.

The decision-making process involves the use of control theory, but this research is limited to the application of existing techniques from that field.

For the purposes of demonstration, we apply the results of this research to the problem domain of uncertain inference. Achieving domain independence broadens the applicability of the system into other domains. It is, however, beyond the scope of this research to develop or implement these models for another problem domain.

1.2 Organization

This document presents a domain independent method for performing the algorithm analysis (Chapter V), then applies it to the algorithm selection (Chapter VI) and run-time control (Chapter VII) processes. First, however, we provide some background on the fundamental concepts needed for this research (Chapter II) and a more detailed description of the research plan (Chapter III). Finally, we look at the performance of our prototype implementation (Chapter VIII) and discuss the future of this research (Chapter IX).

II. Background

Our research builds on two primary foundations: *anytime algorithms* and *control theory*, each of which is a significant research area. Much work has been done in both fields, enough to fill books of their own; the purpose of this chapter is to summarize the significant elements of each area, reviewing the past work done in these (and related) areas and highlighting those concepts underlying our research. In addition, we also present a brief overview of the target problem domain: Uncertain Reasoning using Bayesian Networks.

2.1 Anytime algorithms

Anytime algorithms were first used by Dean and Boddy (2, 7) to provide a means to balance execution time with solution quality in their work on time-dependent planning. In general, anytime algorithms are useful when problems are computationally intractable; they provide a means for evaluating the progress of an algorithm during its execution. Anytime algorithms have four characteristics which differentiate them from ordinary algorithms(11):

Quality Measure It is possible to quantify the quality of a solution generated by the anytime algorithm. This could be measure of the closeness of the solution to the optimal answer, or an indicator of how complete the solution may be, or even a 'best guess'.

Predictability Through empirical, statistical or analytical means, we can estimate the output quality of the anytime algorithm given a particular time and input data. This information can be used by a meta-level controller to determine the usability of a particular algorithm.

Interruptibility An anytime algorithm produces the current status of its solution process when interrupted or periodically throughout its execution. This status can be used to reallocate processing to other algorithms, or simply to provide an interim solution to the problem.

Monotonicity The quality of the output of anytime algorithms never decreases over time; it either remains constant or increases as the algorithm is given more time to work.

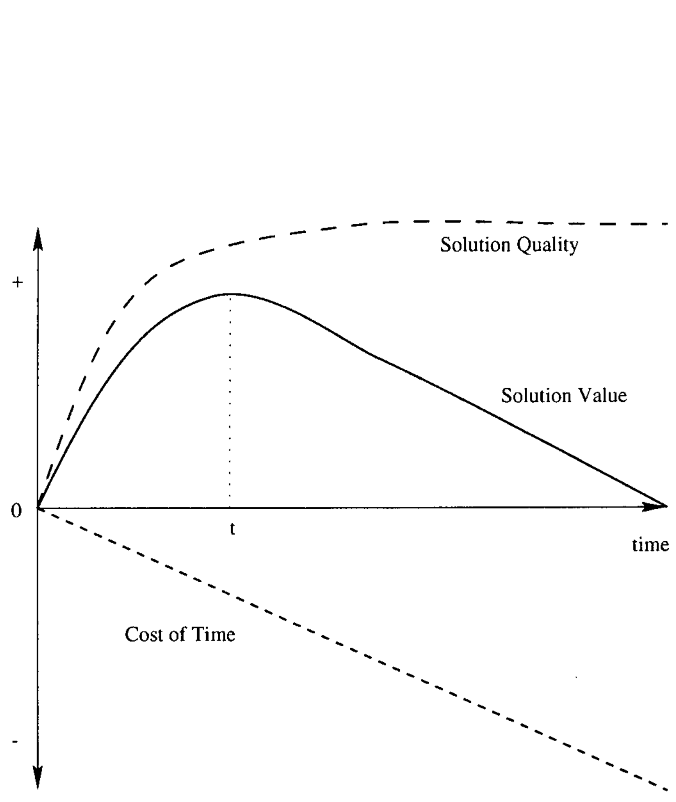


Figure 2.1 Utility of an anytime algorithm

One place where anytime algorithms are particularly useful is where the *value* of a solution involves more than just its quality level. A typical application of anytime algorithms would be in a real-time decision-making environment. Decisions need to be made within time constraints, so we need the best possible answer within the time available. This implies a cost associated with time, such that the longer we wait, the more costly the solution. This cost of time counters the increase in solution quality. If the overall utility is a function of the solution quality and the cost of time, then a typical overall utility performance curve would be as shown in Figure 2.1. As the figure shows, there is a clear point t beyond which the value of the solution starts decreasing. The objective is to determine where this deflection point in the utility curve is located in order to recognize when the system's performance is balanced between solution quality and execution time.

Even if time is not a factor in the value of a solution, it is frequently useful to be able to predict the improvement an algorithm will make if allowed to continue. One method for making this prediction is to create a *performance profile* for the algorithm. This profile characterizes an algorithm's performance over a set of parameters; they can be generated analytically or empirically. Extensive work has been done in empirically characterizing anytime algorithms(9, 11), but little has been done through algorithm analysis.

The logical next step is to use the performance profiles to determine an optimal stopping point. To do this in advance, use the performance profile for the algorithm and optimize the utility of the system:

$$U(t) = U_I(t) - C(t)$$

where t is the stopping time, U_I is the *intrinsic utility* of the algorithm derived from the performance profile, and C is the cost of time. While this could be estimated and determined in advance, it depends heavily on the accuracy of the estimation methods to get a reasonable result. Another approach is to monitor the progress of the anytime algorithm(13); this allows us to identify more precisely the point at which the algorithm is not cost-effective.

In cases where a single algorithm is not sufficient to solve the problem, multiple algorithms would need to be used; Zilberstein and Russell have researched the use of anytime algorithms as sequential components of a larger system(42). This introduces the problem of time allocation between the components; how much time should be allocated to the earlier components in order to optimize the results of the later components. This process involves compiling a schedule which optimizes the complete system: in the general case, an NP-complete problem all by itself. They have developed a method for performing the compilation offline, with run-time monitoring to fine-tune the schedule.

2.2 Algorithm Combinations

Each algorithm uses different methods to find the optimal solution to a given problem, much as individuals in a group of people take different approaches when given the same problem. We have observed, as have others(19, 20) that cooperation between algorithms can likewise result in a more rapid solution. The current issues with this approach are:

- *What to share between algorithms*: How much and what kind of information is needed to effectively help the other algorithm(s) without inducing too much overhead?(19)

- *How to use the shared information*: How should the algorithm utilize the information imported from the other algorithm(s)?(19)
- *When to share information*: When is sharing the information beneficial, or is it better to just let the algorithms run independently?(21)

The concept underlying information sharing between algorithms is referred to as the *anywhere* property(30). This property refers to an algorithm's ability to accept complete solutions generated elsewhere *and* its ability to incorporate that solution into its own processing. The methods used to accomplish the incorporation of external solutions varies based on the methods used by the algorithm. The solution could be used to place bounds on the problem being worked by the algorithm, it could change the direction of the processing, or it could simply be used to affect the way the algorithm interacts with the rest of the system.

2.3 Control Theory

Control theory in its most basic form is finding a way to provide the appropriate control inputs to the system being controlled, or *plant*, such that the plant's state and output are stable and correct. For very simple plants, a static set of equations relating the plant's current state with the feedback to produce the control signals may be all that is necessary. This type of system has a consistent operating environment such that a single set of equations can handle all possible operating conditions. Many systems do not fit this simple model – the conditions under which the controller must operate vary significantly over the course of the system's operation. A classic example is the different control requirements for an aircraft when flying at subsonic versus supersonic speeds. In cases like these, an *adaptive* controller that can accommodate the changes in operating conditions is needed(1).

Key to the operation of most controllers is a model of the plant. This model provides the expected behavior of the plant so that its actual output can be evaluated. The model provides the relationships between control signals and the plant's state. If the behavior of the plant is well-defined, an adequate model is readily developed; many physical systems fall into this category. It is also possible that a given system's behavior is represented by several distinct or overlapping models depending on external forces or control inputs – it is the job of the adaptive controller to determine which model of behavior is appropriate.

For the controller to be effective, we need to address issues of controllability, stability and optimality. A plant is controllable if the controller is able to provide inputs that change the state from any given state to a state in the optimal region. This factor is important if the controller is to be able to have any effect on the output of the plant. While stability is not a necessary characteristic of a plant, it does make the controller easier to create; a stable plant is one in which a small change in the inputs does not produce a large change in the output. An optimal controller is one which generates controls to the plant which optimize the objective(s) of the controller. Again, this not a necessity for controller design, but provides a level of assurance that the best decisions are being made.

Another perspective on control that is relevant to this research is presented by Dean and Wellman(8) and Hayes-Roth(14, 15, 16, 17): control is simply determining a sequence of actions to take to cause the plant to remain in a desired state; which is the definition of *planning*. The controller uses the plant model to predict what the next action needs to be based on its determination of the current state of the plant. Additionally, the controller can monitor the plant, using the resulting feedback to determine the appropriate course of action.

2.4 Intelligent Control

Barbara Hayes-Roth has published numerous articles and papers on this subject; from her 1985 foundational paper on control of blackboard architectures(14) to her introduction to a special issue of *Artificial Intelligence* on intelligent control(15) in 1993. Her later papers, including (17) and (16) focus on adaptive intelligent control of systems. The following discusses aspects of these papers, focusing on the relevant issues to this research.

These three articles are from an extended sequence that clearly shows the progression of Barbara Hayes-Roth's ideas from early development of an intelligent agent through her current concept of adaptive intelligent systems specialized to fit niches in the real world.

In the 1985 paper, the control problem is clearly stated: *which of its potential actions should an AI system perform at each point in the problem-solving process?* This basic definition is central throughout these and her other papers. The other related fundamental concept presented in the 1985 paper that plays a constant theme in her works is the characterization of intelligent control as a *planning problem* with these behavioral goals:

1. Make explicit control decisions that solve the control problem
2. Decide what actions to perform by reconciling independent decisions about what actions are desirable and what actions are feasible
3. Adopt variable grain size control heuristics
4. Adopt control heuristics that focus on whatever action attributes are useful in the current problem-solving situation
5. Adopt, retain and discard individual control heuristics in response to dynamic problem-solving situations
6. Decide how to integrate multiple control heuristics of varying importance
7. Dynamically plan strategic sequences of actions
8. Reason about the relative priorities of domain and control actions

These goals are somewhat restated in her later papers to describe an *adaptive* intelligent controller.

The emphasis on defining control as a planning task in the 1985 paper seemed to be a significant occurrence at that time, although it is interesting that in her 1993 paper she didn't think she did enough to bring it out and compare her methods with the "classical" planning methods. This definition of control opened up possibilities to make the controller "intelligent" by enabling it to base decisions not only on the immediate feedback from the controlled tasks, but also on tactical and strategic policies. Different sequences of actions could be explored and evaluated. The author obviously recognized the usefulness of this formulation of the control problem – this theme is the basis for much of her later work.

The 1993 paper appears (at least on the surface) to be an introduction to a larger work rather than a work of its own. The author is very retrospective, though the summary of the 1985 paper emphasizes the execution cycle of the controller:

1. Execute the current action
2. Determine the possible next actions
3. Evaluate and select the next action

The *action* could be a single activity or an entire plan itself. This structure allows the controller to adapt the controller by placing restrictions on the possibilities for the next action or by changing the evaluation function. The description of the control process in this fashion is what opens this field up to the methods used in the planning area of research. The controller's activities are mapped into a sequence of actions – but this is just what planning is for: determining the best sequence of actions to accomplish a task.

The extension of the model to accommodate dynamic environments was definitely the next logical step in the model's development, but even so was significant because it gave the controller the opportunity to respond to its environment (something that was not common at the time).

The 1995 paper emphasized the *adaptive intelligent system* extension of the author's research. The focus of this papers is to define a class of intelligent agents that is more general than the current typical agent; the goal is to broaden the scope of an agent's capabilities from its current state (highly restricted single-function agents) to somewhere a little closer to the unrestricted capabilities of humans.

The primary means of broadening used is changing from a single fixed mode of operation to an *adaptive* mode that responds to changing conditions both internal (resources, time) and external (movement, perception) to the agent. This adaptiveness is described in five major behaviors: perception, control, reasoning scope, reasoning methods and meta-control. A goal for the adaptiveness is stated for each of these behaviors.

The architecture presented is intended to provide a superset of that found in simpler agents, not a static specification of what must be implemented. It provides the framework and the interfaces between the components of the architecture, but does not specify what must be in each component. This allows a system to *grow* into this architecture as its capabilities are expanded. The author took this approach because while the architecture does represent a significant step in intelligent agent development, it is an achievable step because it focuses the research on the *adaptiveness* of behavior that gives humans so much flexibility and power.

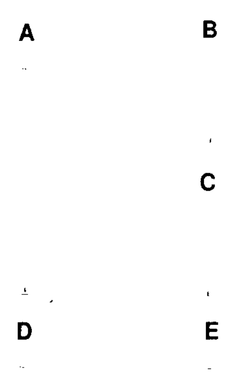
Differences The most significant difference between the intelligent control discussed above and "traditional" AI approaches is that the intelligent controller has the capability to respond to more than immediate (local) data when making a decision. Heuristic searches are only capable of using local information to determine the direction of the search; an intelligent controller could be evaluating several decisions (or even sequences of decisions) in advance.

The other major point brought out in the 1995 paper (actually written in 1992) is adaptability. Adaptability is what gives humans extreme flexibility and the capability to work with very diverse problem areas. By incorporating adaptability, an agent (or other system) can work efficiently over a broader range of problem domains utilizing the most appropriate method for the current task.

Another point related to adaptability is the capability to respond to situations (internal and external) by dynamically determining the appropriate actions. This is definitely in contrast to the traditional expert system mentality which operates in a static, non-real-time environment.

2.5 Bayesian Networks

Bayesian Networks (26) are a compact way of representing probabilistic uncertainty in automated reasoning. Such networks consist of directed acyclic graphs of nodes, each representing a random variable (RV) with a finite domain. Edges represent direct dependency, and the topology as a whole conveys independence assumptions. These allow the joint probability distribution over all the RVs to be fully determined by providing condi-



```
P(A) = .3

P(B) = .8

P(C|B = T) = .1
P(C|B = F) = .1

P(D|A = T, C = T) = .6
P(D|A = T, C = F) = .4
P(D|A = F, C = T) = .4
P(D|A = F, C = F) = .6

P(E|C = T) = .7
P(E|C = F) = .3
```

Figure 2.2 A small example Bayesian Network

tional probabilities of variable states given all their predecessor states (see Figure 2.2), as the product of all the conditional probabilities.

One form of uncertain reasoning on Bayesian Networks is *belief revision*(26). Belief revision is the process of determining the most probable instantiation of the random variables (RVs) in a network given some evidence. More formally, if W is the set of all RVs in the given Bayesian Network and e is the evidence[1], any complete instantiations to all the RVs in W that is consistent with e is called an *explanation* or *interpretation* of e. The problem is to find an explanation w^* such that

$$P(w^*|e) = \max_{w \in W} P(w|e). \tag{2.1}$$

Intuitively, we can think of the non-evidence RVs in W as possible hypotheses for e.

As an example, consider the small Bayesian Network in Figure 2.2; to the right of the network are the associated probabilities. Given the observation that node D is true, the goal of belief revision is to determine the most probable complete instantiation over all the nodes in the network that is consistent with D being true and maximizes the joint probability of that assignment to the network (Equation 2.1). This results in the solution $\{A = F, B = T, C = F, D = T, E = F\}$ which has joint probability 0.21168.

[1]That is, e represents a set of instantiations made on a subset of W.

With a small network such as this, a valid solution method is to simply tabulate all the possible values of the RVs and then calculate the probabilities manually. Once the network gets larger and more complex, this method is obviously unacceptable and more efficient methods must be employed.

Current methods for *approximating* belief revision and updating on Bayesian Networks include best-first search(4), linear programming(29) and genetic algorithms (28, 35). (*Exact* algorithms for these problems abound and are too numerous to cite here, see Pearl(26). However, these algorithms are, as a rule, exponential-time algorithms.) The first two methods are deterministic: they find the optimal solution, given enough time and resources. Solutions based on genetic algorithms are nondeterministic.

While the general problem of reasoning on Bayesian Networks is NP-hard (5, 6, 31), each of the above algorithms exploit *different characteristics* of the problem domain to reduce the time required to generate the solution. That is, each is potentially exponential, but handles certain *classes of networks* efficiently. Attempts have been made to empirically characterize the behavior of different algorithms(22); the results clearly show that the amount of time required to determine the optimal solution is dependent on the characteristics of the network being used. Unfortunately, empirical methods such as this suffer from the large number of potential parameters; it is impractical to completely characterize an algorithm's performance.

2.6 Summary

This research builds on a foundation supported by several different fields, including anytime algorithms, control theory and uncertain inferencing. Each of these fields is quite established, with an abundance of research documented in the literature; yet there has been little research combining these fields. The discussion in this chapter summarized the areas within these established fields that are relevant, and identifies areas which are open for further research.

III. Approach

Recalling the objectives outlined in Chapter I, this research focuses on developing the models necessary to build a controller for a cooperative distributed system. One goal that affects all facets of this process is that the resulting models should be domain independent. If the model is constructed properly, the resulting system will be applicable over many different problem domains. We use uncertain inference over Bayesian Networks as our initial domain. This domain was chosen primarily due to the availability of local expertise in this field and the large variety of algorithms for solving this problem. After some exploratory experimentation, the research proceeded in five phases.

3.1 Phase 1: Architecture Development

Phase 1 established the foundation for the controller's run-time environment. The OVERMIND architecture was created to provide the means to execute, monitor and control the algorithms. A prototype of OVERMIND was developed using available tools.

3.2 Phase 2: Algorithm Modeling

With the development environment in place, the process of analyzing algorithms can start and the initial problem domain of uncertain inference selected. Drawing from the techniques developed in research on anytime algorithms, we explore the effects of sharing the intermediate results produced by the anytime algorithms. The algorithm model is then defined and applied to the algorithms for uncertain reasoning.

3.3 Phase 3: Initial Algorithm Selection

Phase 3 is where the algorithm analysis techniques are utilized to determine which of the many algorithms available will most efficiently produce the best possible answer in the time available? Finding the optimal answer to this question is almost as difficult as actually finding the optimal solution itself; therefore, we develop and implement a selection process based on algorithm analysis instead of empirical methods.

3.4 Phase 4: Run-time Control

In this phase, the focus is on mapping the objectives into a control theory problem. This makes available the existing base of control theory techniques to develop the controller. This effort involves determining the composition of the system *state* and the objective functions for the controller to use.

3.5 Phase 5: Implementation and Evaluation

Throughout the effort, extensive testing is conducted to validate the analytical results, and to gain further insight into the behavior of the system. A set of Bayesian Networks with varying topologies, probability distributions and sizes is used to provide a reasonably diverse data set for testing. For static selection, the objective is to be able to select the algorithm(s) that perform the best; for run-time, the objective is to keep the system configured so that the expected progress is high.

3.6 Summary

This research plan addresses each of the goals and objectives discussed in the previous chapter, and lays out four distinct phases for the development of the overall control model. The first phase necessarily creates the distributed development environment essential for the testing of the concepts developed later. The next phase provides the analysis techniques needed for the algorithm selection and run-time control phases. The implementation and evaluation phase is accomplished concurrently with the other phases to provide the means to experiment with and validate the concepts being developed.

IV. The OVERMIND architecture

The goal for the first phase of the research is to develop an architecture that provides the support necessary for control of a distributed system. Specifically, it needs to:

- provide the capability to start and stop tasks on remote systems;
- provide the means to communicate with tasks on remote systems;
- manage the distribution of tasks across the systems;
- easily allow new algorithms and systems to be added;
- be highly modular with minimal coupling to allow components of the architecture to be substituted as needed.

Figure 4.1 shows the resulting architecture. Each component of the architecture is defined by function and interface, creating a highly modular design. The major components of the architecture are:

- **Library of Tasks (LOTS)** - contains a variety of algorithms suitable for solving the problem.
- **Overseer Task Manager (OVERSEER)** - directs the flow of messages and information between cooperative tasks; initiates and tracks new tasks as requested by the consumer tasks.
- **Intelligent Resource Allocator (IRA)** - manages and allocates available computing resources such as multi-processor systems, workstations, and personal computers.
- **Consumer Tasks** - interface with the outside world, translating requests and queries into instances of the problem domain and choosing and configuring the tasks needed to generate an appropriate response to the requester. Each consumer task is specialized to handle a specific type of external request.
- **Producer Tasks** - are the solution-generating processes in the system. Each producer task is a product of the IRA assigning a host to an algorithm instance from the LOTS combined with a set of parameters from a consumer task.

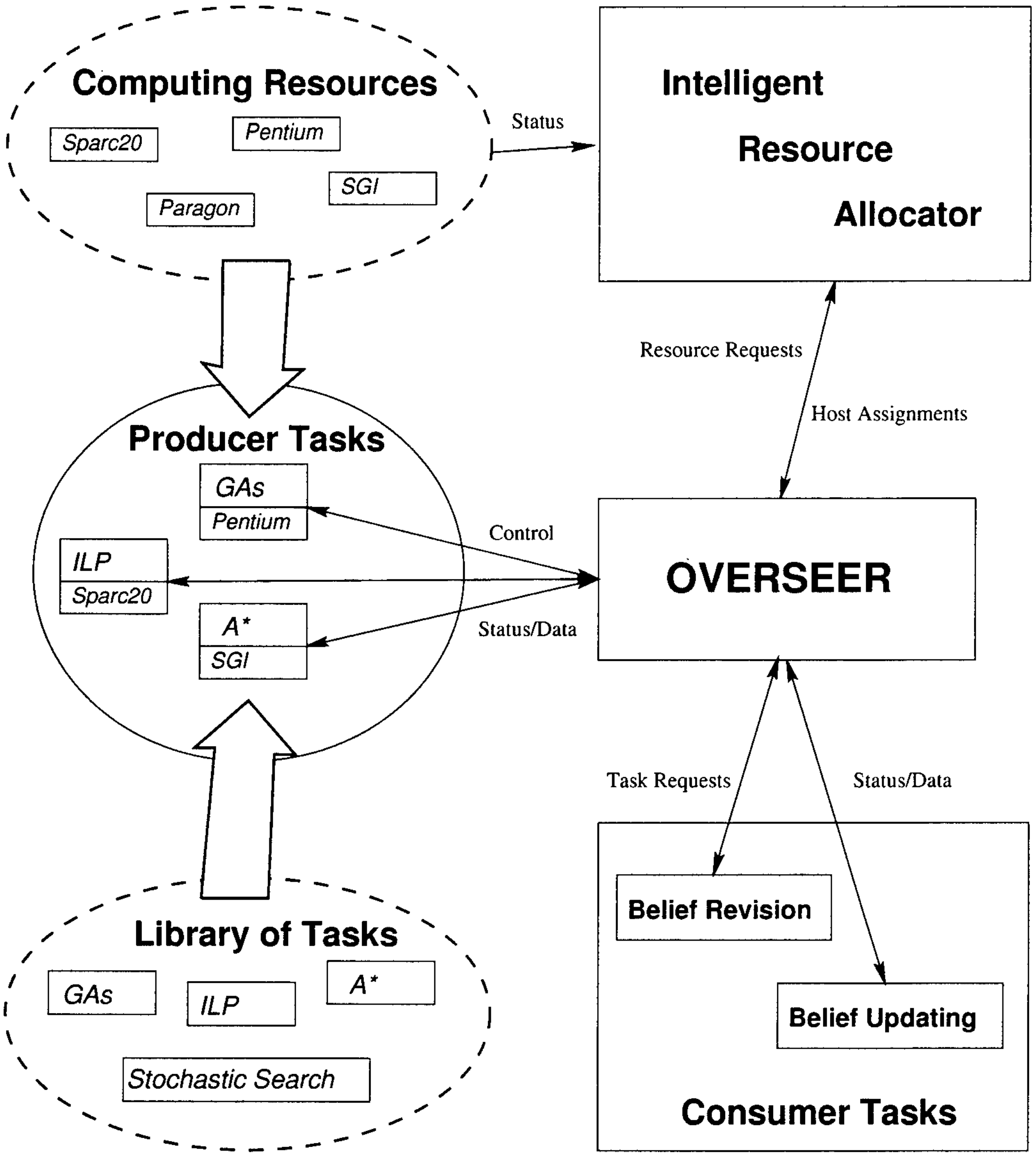


Figure 4.1 The OVERMIND architecture.

As can be seen from Figure 4.1, the OVERMIND architecture is geared towards anytime solutions, anywhere solution sharing and cooperative task management in order to provide fast, near optimal, solutions. Its distributed nature also allows it to effectively utilize diverse networks of computing resources. This modular approach makes it possible to incorporate existing software packages into the OVERMIND framework. Each component of the OVERMIND architecture is described in more detail below.

4.1 Library of Tasks

Key to the success of the OVERMIND architecture is a well-equipped Library of Tasks (LOTS). Most intractable problems have a variety of algorithms available, and each algorithm typically is very efficient for a subset of the entire problem domain. Unfortunately, there are no algorithms which are efficient over the entire problem domain. In terms of problem suitability, it therefore seems natural to choose the best algorithm based on the problem instance being solved. Better yet, if a particular approach is good at starting off a problem or solving some portion of it, we can then take its partial solution and pass it to another approach which itself may be better on this new portion to work on. This naturally leads us in a direction amenable to anytime and anywhere solutions. LOTS needs to contain a variety of algorithms exhibiting both of these important properties, so that solution sharing between methods is possible.

Up to this point, we have been using the term *algorithm* as if it was a single process on a workstation. While this is an obvious first step in populating the library, there is nothing within the OVERMIND architecture that requires this; in fact, the highly modular, loosely coupled structure permits the library tasks to take whatever form is most beneficial. The tasks could be single processes on a workstation, a collection of cooperating processes on one or more workstations, or even a set of tasks running on a multi-processor system.

There are few requirements placed on the algorithms contained in the library in order to ease the task of adding algorithms to the library as well as to reduce the impact of the OVERMIND environment on the algorithm itself. These requirements are:

1. The algorithm must be remotely startable without any command line parameters.

2. The algorithm will receive its data set and any parameters via inter-process communication from the consumer task (through the overseer).
3. The algorithm will send its intermediate solutions to the consumer task.
4. The algorithm will accept (but not depend on) intermediate solutions distributed by the consumer task.
5. The algorithm should have the anytime and anywhere characteristics.

The last requirement is more of a strong recommendation: non-anytime algorithms have been successfully incorporated into the OVERMIND system, but their utility is limited.

4.2 The Overseer

The Overseer Task Manager (OVERSEER) serves as the coordinating member of the OVERMIND architecture. Its primary purpose is to manage the process of allocating computing resources to algorithm instances, resulting in *Producer Tasks.* The OVERSEER also serves as the message center, relaying data, status and results between the components of the system.

The OVERSEER receives requests from *Consumer Tasks* to start an instance of an algorithm, consults with the Intelligent Resource Allocator (IRA) to determine which host or hosts will be used, then initiates the producer task, monitoring it until the task terminates.

In its simplest form, the OVERSEER would simply provide message passing, synchronization and task management services. It effectively insulates the consumer tasks from the details of starting their producer tasks, and ensures that multiple consumer tasks do not interfere with each other.

The OVERSEER could also play a larger role in extending the distributed environment. Under the direction of the IRA, tasks could be relocated as resource loading changes. OVERSEER could also provide a transparent interface between dissimilar hosts, providing whatever protocol or data translation services are needed.

4.3 Intelligent Resource Allocator

The Intelligent Resource Allocator (IRA) serves to maximize processor use by coordinating requests for resources from OVERSEER and even possibly the producer and consumer tasks themselves. In the environment OVERMIND is intended to operate (a heterogenous network of workstations and computers), IRA would need to be aware of the type and configuration of each host to be used. To perform its duties efficiently, IRA also needs to be aware of the resource requirements of the members of the LOTS: As an example, GAs tend to be quite CPU intensive but require very little memory, the stochastic sampling requires very little CPU, but uses quite a bit of memory. Simply identifying a Sparc2 from a Sparc20 can improve efficiency greatly.

For the most part, we can predict relatively easily the resource requirements for our current collection of tasks. However, once we begin introducing highly specialized tasks and sophisticated variants of our existing tasks, we must move towards a model-theoretic approach of resources and requirements management.

4.4 Consumer Tasks

The consumer tasks provide the external interface to the OVERMIND system. These tasks would accept queries or other requests from the user interface or other external entity and convert them into a Bayesian network to inference over. Each of the consumer tasks would be specialized to respond to a particular type of request; either directly or through the use of other consumer tasks.

Based on the request, the consumer task would determine what is needed to provide an answer to the requester – which algorithms to use, how to construct an appropriate problem to represent the request and what the termination conditions need to be. The consumer task selects algorithms from the Library of Tasks (LOTS); since the algorithm instances contained in the library could be configured through the use of parameters, it is entirely possible that several instances of the same algorithm could be used – with the parameters chosen to produce the desired effect. Through the OVERSEER, the required algorithms are invoked, and the consumer task uses the output of the algorithms to generate the response back to the requester.

As will be discussed later, consumer tasks could also implement some form of meta-level reasoning to dynamically allocate computing resources to tasks based on their performance, either actual or expected. Each consumer task will essentially reason over the pool of available computational resources, tailored to the specific problem instance it is designed to process. A decision-theoretic framework could be used, in which a utility function over outcomes is given.

Another possibility inherent in the structure of OVERMIND is the capability for the consumer task to control the algorithm instances directly through the use of run-time parameter changes. If an algorithm is not performing well, a new set of parameters could be sent to modify its behavior and (hopefully) improve its performance.

4.5 Producer Tasks

Unlike the other components of the OVERMIND system, the producer tasks do not exist until they are needed at run-time. Each producer task is a combination of an algorithm instance from the LOTS, a set of parameters from a consumer task and a host allocated by the IRA. The producer task is created dynamically by the OVERSEER.

The purpose of the producer task is to generate solutions to the problem set up by a consumer task. Ideally, it will produce solutions incrementally in anytime fashion, as well as accept intermediate solutions – the anywhere capability.

4.6 Prototype Implementation

Our prototype implementation of the OVERMIND system was developed using C and C++ on a network of Sun and Linux workstations. The first two requirements from the list at the beginning of this chapter were satisfied through the use of the Parallel Virtual Machine (PVM) communications library(10). The PVM library provides the capability of remote spawning and killing of tasks, as well as a set of communications primitives to allow inter-task communications. Inherent in PVM's design is the ability to transparently interconnect different hardware platforms; this was ideal for OVERMIND's target environment of a heterogenous network of workstations.

The Library of Tasks was populated with algorithms from the initial problem domain of uncertain reasoning; specifically, belief revision over Bayesian Networks. Two variants of heuristic search were used, in addition to a Genetic Algorithm and a Hybrid Stochastic Search. One non-anytime algorithm was also used: the Barrier Computation algorithm described in Appendix A. These algorithms were chosen because of the variety of methodologies and the availability of existing implementations.

4.7 Summary

The OVERMIND architecture provides a flexible environment for experimenting with distributed problem solving. Its modular design allows the independent development of the components, as well as alternate implementations of a given component without affecting the other components in the system. OVERMIND's capability to transparently utilize multiple hardware platforms creates an ideal test bed for experimenting with control methods for a distributed system.

V. Algorithm Modeling

The second phase of this research effort addresses the need to develop analytical models of algorithms used on the target problem. Algorithm analysis is an old, established research field; but the primary goal of traditional analysis is to determine algorithm complexity(39, 40) or to predict the algorithm's performance over the entire execution time(34, 36). This is not what is needed for algorithm selection and control; therefore a different approach to algorithm analysis is indicated.

We have two motivations for this analysis: algorithm selection and run-time control. Initial algorithm selection requires the ability to predict algorithm performance over an initial short time interval. This time interval is dependent on the time required for the algorithms to initialize and begin producing results. Run-time control also utilizes the expected performance of the algorithms; but in order to provide a stable system, it also needs to address additional run-time performance factors.

Both of these uses for our analysis have a characteristic which makes our analysis task much simpler: they both only need to evaluate the algorithm's performance potential over a short interval. This concept is discussed further in the Chapters VI and VII. The important point here, however, is that we can take advantage of approximations of the algorithm's behavior to simplify the analysis and reduce the complexity of the resulting models.

Our algorithm model consists of six factors which we have identified as being useful in predicting algorithm performance: *Time Complexity, Space Complexity, Resource Cost, Restart Cost, Level of Improvement* and *Probability of Improvement.* The model also contains one factor (*Interaction Rate*), which is calculated at run-time for each algorithm. These factors capture the expected performance of an algorithm (or combination of algorithms) in a domain-independent manner: when applied to an algorithm, the factors are calculated using characteristics of the problem domain, but the value of each factor is a unitless scalar. It is at this point where the domain independence is established.

The rest of this chapter first describes the analysis needed to support the algorithm model, then provides a more detailed description of the factors contained in the model.

5.1 Analysis Methods

This section presents a method for analyzing algorithms; the goal is to produce measures which can be used to predict the algorithm's performance. To achieve domain independence, these measures must be scalar values: they cannot be stated in units which are tied to a particular problem domain. For this purpose, we use the term *solution quality* to refer to the output of the process, with the goal to produce the solution with the highest quality value. How the quality of a particular solution is determined depends on the domain: it could be related to the progress of the algorithm (how far until it is done) or possibly a value which is calculated directly from the solution itself. The important concept is that the solution quality can be used to compare solutions, with our goal to obtain the highest quality solution(s).

For the purposes of predicting progress, there are two measures which capture the potential behavior of an algorithm:

Probability of Improvement How likely is the algorithm to produce a solution with a higher quality value than the current solution?

Level of Improvement What is the expected relative change in the output value of the algorithm?

These concepts are developed more fully and applied to different algorithms in the domain of uncertain reasoning in the following sections.

5.2 Algorithm Analysis

One factor needed by the run-time controller is the likelihood that an algorithm will produce a better solution if allowed to continue from a known point. This likelihood is called the *probability of improvement*:

$$P_{imp}(S_0) = \sum_{S_1 \in \mathcal{S}} P_{select}(S_1|S_0) P\left[Q(S_1) > Q(S_0)\right] \tag{5.1}$$

where S_i is the current anytime solution available and $\mathcal{S}$ is the set of all possible solutions. This probability encompasses both the characteristics of the solution landscape

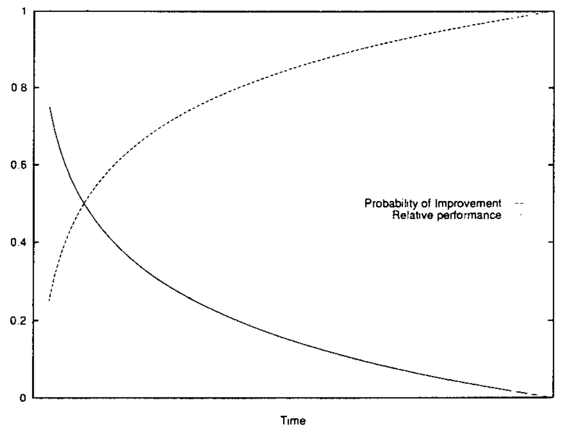


Figure 5.1 The relationship between algorithm performance and probability of improvement

(through the solution evaluation function $Q(S_1)$) and the actions of the algorithm itself ($P_{select}(S_1|S_0)$). The probability of improvement typically has an inverse relationship with overall algorithm performance as shown in Figure 5.1: over time, as the algorithm's solution quality improves there is less room for improvement, and the probability of improvement decreases. The algorithm with the highest probability of improvement is most likely to be the algorithm with the steepest performance profile and this is the algorithm most likely to rapidly produce a high quality solution.

The anytime nature of the algorithms allows us to treat each one as a state machine, and its operation can be modeled as the relation that produces the next state (S_1) given the current state (S_0). Thus, the algorithm's execution produces a sequence of states. In general, for each state in the sequence there is a set of possible states that can be transitioned into. In the first term of the summation, the probability of selecting these states is non-zero. P_{imp} is then the likelihood that the algorithm will select a solution with a joint probability higher than the joint probability of the initial state S_0. For a simple random search (equal probability of selection for all solutions) the probability of selection is $1/N$ ($N = |S|$), and P_{imp} is simply the likelihood that a solution exists with a higher joint probability. This model is not necessarily exact, but reflects the nature of the algorithm's response to interesting characteristics in the data as well as its response to the other algorithms in the cooperative system.

The other interesting characteristic of anytime algorithms is the *level of improvement* produced by the algorithm. This is the expected magnitude of the increase in solution quality that the algorithm will make in the next increment:

$$E_{imp} = \frac{Q(S_1)}{Q(S_0)} \tag{5.2}$$

With anytime algorithms the level of improvement is always greater than or equal to 1.0 because the output quality is required to be non-decreasing.

The rest of this section analyzes four different inference algorithms for performing belief revision on Bayesian Networks. For this problem domain, the joint probability of the solution is used as the solution quality. The algorithms considered here are an A* search(3, 4), a simple Genetic Algorithm (GA)(24), a hybrid stochastic simulation with local optimization and a Barrier Computation algorithm. All of these algorithms are both anytime and anywhere except for the Barrier algorithm: it is included to demonstrate the extent to which this analysis can apply to non-anytime algorithms.

5.2.1 A Search.* Traditional best-first algorithms with additive evaluation functions, such as A^*, begin with an initial state and perform heuristic (possibly exhaustive) search for the optimal goal. The two major components of this type of search are the next-state generator and the heuristic evaluation function. Earlier work experimented with different admissible heuristics for this problem: current cost(4) and shared cost(3). Both are guaranteed to provide the correct optimum, given unlimited time. Another type of heuristic for the A* search is *monotonic*; this heuristic is guaranteed to select the best path through the search tree. If such an heuristic can be found, it will produce the best possible performance for that problem domain; unfortunately, such heuristics are difficult (if not impossible) to develop. In the context of anytime processing within the OVERMIND system, the assumption of limited time is made due to the intractable nature of the problem domain; thus this guarantee does not directly apply.

The A* contained in our system uses an implementation of the shared-cost heuristic(3). The heuristic was developed for weighted AND-OR DAGs (WAODAGs); a mapping of the heuristic to Bayesian Networks is straightforward(32). The next-state generator expands one RV based on a topologically ordered list of RVs. Due to the nature of the heuristic, this ordering is from the leaves of the network to the root.

This A* algorithm was adapted for use in this system by adding the anytime and anywhere characteristics. The first of these is by far the easier; to provide an anytime answer, simply pick the current best partial instantiation and (if necessary) complete the instantiation; for example, by random or arbitrary instantiation of the unassigned variables. The completion could also be achieved by selecting local optima for the remaining variable instantiations, for example by using the efficient polytree belief revision algorithm (26) or by any other fast scheme. In this implementation, the solution is completed by using the first defined state in each uninstantiated RV.

Allowing an intermediate solution to help a best-first search is more difficult. In order to use the solutions as starting points for continuing the search, they would have to be compatible with partial instantiations already produced by the search algorithm, preferably such that the optimal solution be reachable from these solutions; but here, this situation cannot be guaranteed. Even if the incoming solutions cannot be used as "jumping-off points" by the best-first algorithm, the value of the solution can be used to prune the search space: since we know that a solution exists with a particular probability, any partial solutions with lower heuristic values can be removed from further consideration. This is where a highly accurate heuristic would benefit most – as the incoming solutions approach the optimal value, the open queue of the best-first search would rapidly reduce to the optimal solution. The incoming solution can also be used to help with the generation of the anytime solution; instead of using an arbitrary value to fill the uninstantiated RVs, the values from the received solution can be used as a template. This allows the generated solutions to benefit from the optimization that has already occurred in the rest of the system.

Determining the probability of improvement based on a given intermediate solution is also difficult. Lacking any knowledge about the source of our solution, the range of possible solutions that could be produced by the next iteration of the search algorithm cannot be narrowed; thus the probability of selection is $1/N$ for all solutions. This gives:

$$P_{imp}(S_i) = \sum_{S \in \mathcal{S}} \frac{P\left[P(S) > P(S_i)\right]}{N} \tag{5.3}$$

This reduces to the percentage of solutions with probabilities higher than our given solution, which is equivalent to the probability distribution function,

$$P_{imp}(S_i) = P(X > c) \tag{5.4}$$

where X is a discrete random variable with an underlying probability density function $P(S)$, the joint probability distribution of our network. The constant c is the joint probability of our given solution $P(S_i)$. If the probability distribution of our network is known, the probability of improvement can be calculated directly; unfortunately, this is rarely the case.

If the given solution came from an earlier iteration of this search, the probability of improvement can be estimated by using information about the A* heuristic. The A* search produces a series of partial solutions such that the value of the solution (cost plus heuristic) is monotonically non-increasing. Due to the bottom-up expansion of the next-state generation method, the RVs in this partial solution fall into two groups: I represents those RVs that are instantiated and also have all of their parent RVs instantiated; F covers those RVs on the 'fringe' which are instantiated but do not have all of their parent RVs instantiated. A third group of RVs in the network consists of those that are uninstantiated; these are represented by U.

Using these categories, the probability of improvement from one iteration of the search to the next can be determined by comparing the probability of the given solution $(P(S_0))$ with the probability of the next generated solution$(P(S_1))$. Using the chain rule to expand these probabilities for each solution:

$$P(S_i) = P(I_i|\pi(I_i))P(F_i|\pi(F_i))P(U_i|\pi(U_i)), \tag{5.5}$$

where $\pi()$ is the set of instantiations to the parents of the RVs. Note that the last term involving U is dependent solely on the method used to complete the solution. Because the two solutions are completed with the same value, the effects of the completed part of the solution on the joint probability are limited to the dependencies of the RVs in F on the completed RVs. If the *cost* of the partial solution is defined as the marginal probability of those RVs that have complete parental instantiations (the set I) and $h()$ represents the heuristic covering the RVs not in I, the monotonically non-increasing attribute of the cost and heuristic can be stated:

$$P(I_1|\pi(I_1))h(F_1 \cup U_1) \leq P(I_0|\pi(I_0))h(F_0 \cup U_0). \tag{5.6}$$

By moving the probabilities to the left side of the inequality, the following relation between the cost and heuristic of the two solutions becomes:

$$\frac{P(I_1|\pi(I_1))}{P(I_0|\pi(I_0))} \leq \frac{h(F_0 \cup U_0)}{h(F_1 \cup U_1)}. \tag{5.7}$$

Equation 5.4 only represents the relationship between two samples out of a (possibly unknown) joint probability distribution. Equation 5.7 defines the probability distribution the samples are drawn from in terms of the semantics used in the computation of the heuristic. Making the proper substitutions results in:

$$P_{imp} \leq P\left[\frac{P(F_1|\pi(F_1))P(U_1|\pi(U_1))}{h(F_1 \cup U_1)} \geq \frac{P(F_0|\pi(F_0))P(U_0|\pi(U_0))}{h(F_0 \cup U_0)}\right]. \tag{5.8}$$

This indicates that the likelihood of improvement is bounded by the relative proximity of the completed portion of the anytime solution to the value of the heuristic representing those same RVs. Since the solutions are completed with the same values, the significant part of the numerator is the marginal probability of the 'fringe' RVs. The accuracy of the heuristic and the variance in the marginal probabilities are the driving factors in this calculation.

The level of improvement E_{imp} can be determined by looking at the way the search progresses through its search space. The search processes the search space by selecting the most promising candidate from its open queue and creates new candidates by making all possible assignments to a selected random variable. We refer to this action as reaching a new *level* when the maximum number of assigned random variables for all candidates is increased. The search reaches a new level only when the value of all candidates at the previous level is less than the value of the best candidate at the current level. This is because each time a candidate is expanded, the value of the resulting candidates is less than the value of the original candidate, and none of the new candidates will be expanded until all the others with values higher than its new value have been expanded.

This process impacts the quality of the generated anytime solutions because each solution is completed in order for the other algorithms to be able to use it. If we assume (for the moment) that a new level is reached each time a new candidate is expanded, the original formula for the level of improvement can be expanded by substituting the expansion of the solution probabilities from Equation 5.5 and reducing. The result is:

$$E_{imp}(S_0) = \frac{P(I_1|\pi(I_1))P(F_1|\pi(F_1))}{P(I_0|\pi(I_0))P(F_0|\pi(F_0))}. \tag{5.9}$$

The difference between the two solutions is only one random variable that we call r; this variable gets moved from the fringe set (F) to the instantiated set (I). This reduces the equation to

$$E_{imp}(S_0) = \frac{P(r_1|\pi(r))}{P(r_0|\pi(r))}. \tag{5.10}$$

This result is valid for those cases where the heuristic is extremely accurate and can select the correct candidate each iteration, resulting in a depth-first descent to the optimal solution. In most cases, the heuristic is not that accurate. In those cases where the heuristic is very poor, we could get a breadth-first expansion; the search would only reach a new level when *all* candidates at the previous level had been expanded. The frequency of this occurrence is the logarithm of the number of candidates expanded, with the base of the logarithm being the number of states in the random variables.

The actual rate of descent through the search space depends on the variance in the values in the conditional probability tables and the nature of the heuristic used: higher variances typically result in faster descents, while smaller variances result in more breadth-first operations.

5.2.2 Genetic Algorithms. The class of algorithms based on simple Genetic Algorithms (GA) is a nondeterministic approach to reasoning with Bayesian Networks. The process of belief revision is accomplished when genetic algorithms take a small sample from the space of possible solutions (called the population) and use it to generate other (possibly better) solutions. The method of generating new solutions is modeled after natural genetic evolution.

Each population is subjected to three basic operations (*selection*, *crossover* and *mutation*) during the course of one generation; the results of the operations determine the composition of the population for the next generation. The three operations are probabilistic in nature; this allows the GA to explore more of the search space than a deterministic algorithm. Unfortunately, there is no known method for determining if the optimal solution has been found.

Incorporating anytime and anywhere characteristics into the GA process is straightforward. Since the population contains complete solutions, intermediate solutions can be generated by selecting a high-probability solution. Solutions received from other tasks are easily utilized by incorporating them into the population between generations. Including the external solutions does not disrupt the GA's processing; it has the same effect as a mutation operation.

The two issues that must be addressed when mapping a problem domain into a problem that is solvable by GAs are:

- How to represent a solution to the problem as a *gene* containing a set of *chromosomes* that can be genetically manipulated
- How to evaluate the fitness of a solution

For the belief revision problem domain, the *gene* represents a complete instantiation to the random variables in the network; each *chromosome* represents the instantiation of a single random variable. The genetic operations manipulate each gene by changing the values of the *chromosomes*; if an invalid instantiation is computed as the result of an operation, a randomly generated valid instantiation is substituted. The fitness evaluation is simply our solution quality calculation; in this case, it is the joint probability of the solution.

The *selection* operation is the standard "roulette wheel" selection approach, based on the ranking of the individuals within the population instead of the absolute performance value. With the wide range of performance values typical to this problem domain, a strictly performance-based selection disproportionately favors the highest probability solution; this causes premature convergence of the population onto a localized optimum. The *crossover* operation performs a two-point crossover: two selected genes are broken in two randomly selected places and the middle sections are exchanged to form the new members of the population. *Mutation* randomly selects a chromosome to modify and then randomly chooses a new value for that chromosome.

The result of this genetic manipulation is that the population tends to *converge* towards a localized optimum in the solution landscape; the convergence is exhibited by the population containing a large number of the same solution. If the *mutation* operation is disabled, this convergence typically occurs quite rapidly; unfortunately it is not possible to determine if this local optimum is actually the global optimum. The *mutation* operator helps the GA find other (better) local optimum by forcing some members of the population to lie outside of the current local optimum.

The probability of improvement is dominated by the probability of selection. The probability that a given solution is better than the initial solution is just the probability distribution itself. If the GA is viewed as a set of separate selection operations with identical characteristics, the probability of improvement for the whole operation is

$$P_{imp} = 1 - (1 - P'_{imp})^M$$

where P'_{imp} is the probability of improvement for each member of the population of size M. This obviously is a gross oversimplification, but it does show the significant potential performance for the GA in the early stages where P'_{imp} is quite large.

Determining P_{imp} for the GA requires knowledge of the probability of selecting a given solution, something that in general cannot currently be calculated; but for our problem domain it is clearly dependent on several factors including the following: the probability distribution of the Bayesian Network, the function used to evaluate the members of the population and the number of RVs that are different from the current member of the population. In a simple case where the evaluation function is the joint probability of the solution and the landscape has only a single optimum, the individual probability of selection can be approximated by $P(S)$, giving us

$$P_{imp}(S_0) = \sum_{S \in \mathcal{S}} P(S_1)P(P(S_1) > P(S_0)) \tag{5.11}$$

which is the sum of solutions whose values are greater than the value of the initial solution; a value that can be estimated from the probability distribution of the Bayesian Network. As the members of the population evolve towards their local optima, the level rises and the probability of finding a higher probability solution decreases. Empirically, it has been shown that GAs can find good solutions quickly(28); but again, we do not know when we have the optimal answer.

The level of improvement for the GA is determined by its rate of convergence; this is determined by characteristics of the solution landscape. The *crossover* operator moves the population in small steps "uphill" (towards the closest local optimum); the steeper the slope, the faster the population as a whole converges. If the landscape is level, the *crossover* and *selection* operators have no direction in which to move the population, and convergence does not occur. For Bayesian Networks, the topology of the landscape is a function of the distribution of the conditional probabilities for the random variables; highly skewed networks produce landscapes with steep slopes.

Given two *genes*, G_1 and G_2, from the GA's population selected for crossover, the joint probability of the resulting genes can be calculated. The probability of the initial genes are

$$P(G_1) = \prod_j P(x_{1_j}|\pi(x_{1_j})) \tag{5.12}$$

$$P(G_2) = \prod_j P(x_{2_j}|\pi(x_{2_j})) \tag{5.13}$$

The crossover operator changes the values of one or more RVs; let the set of changed RVs be I and the set containing the RVs who have the members of I as their parents be S; let the set R be the remainder of the RVs. The probability of the original first gene is then

$$P(G_1) = \left[\prod_{i\in R} P(x_{1_i}|\pi(x_{1_i}))\right]\left[\prod_{j\in I} P(x_{1_j}|\pi(x_{1_j}))\right]\left[\prod_{k\in S} P(x_{1_k}|\pi(x_{1_k}))\right] \tag{5.14}$$

After crossover the RVs in I have new values, with the corresponding change in the conditional probabilities used to calculate the joint probability; this is the second product in the above equation. The conditional probability for the RVs in S also change due to the different values of their parents. Thus, the joint probability of the new first gene is

$$P(G_1') = \left[\prod_{i\in R} P(x_{1_i}|\pi(x_{1_i}))\right]\left[\prod_{j\in I} P(x_{1_j}'|\pi(x_{1_j}))\right]\left[\prod_{k\in S} P(x_{1_k}|\pi'(x_{1_k}))\right] \tag{5.15}$$

Notice that the first product term remains the same; so the level of improvement, which is the ratio between the original and new joint probabilities, reduces to

$$E_{imp}(S_0) = \frac{P(G_1')}{P(G_1)} = \frac{\left[\prod_{j\in I} P(x_{1_j}'|\pi(x_{1_j}))\right]\left[\prod_{k\in S} P(x_{1_k}|\pi'(x_{1_k}))\right]}{\left[\prod_{j\in I} P(x_{1_j}|\pi(x_{1_j}))\right]\left[\prod_{k\in S} P(x_{1_k}|\pi(x_{1_k}))\right]} \tag{5.16}$$

This ratio represents the slope of the landscape; a ratio greater than one results in an increase in probability. The terms in this ratio are values pulled from the distributions of the conditional probability tables; the value of the ratio is statistically most likely to be within the interval

$$\left(\frac{\overline{x}-\sigma}{\overline{x}+\sigma}, \frac{\overline{x}+\sigma}{\overline{x}-\sigma}\right) \tag{5.17}$$

where σ and $\overline{x}$ are parameters of the conditional probability distribution. This clearly shows that the slope of the landscape is directly related to the parameters of the underlying distribution of probabilities in the Bayesian Network.

5.2.3 Hybrid Stochastic Simulation (HySS). The stochastic simulation algorithm uses a combination of random search and local optimization to perform belief revision. This method differs from an ordinary random search in two ways. First, a set of solutions is maintained as for the genetic algorithms; the members of the set do not interact, but the overall quality of the solutions in the set is used to steer the search towards better solutions. The second difference is in the method for selecting the members of the set: instead of choosing a random solution each time, each member's joint probability is used to determine whether that solution is optimized or replaced. If the solution is to be optimized, it is analyzed to determine the change to a single random variable instantiation that produces the largest increase in joint probability; this new solution replaces the original in the solution set.

This algorithm has three parameters:

Stability determines, in combination with a solution's joint probability, the likelihood that a solution S is retained in the the population or replaced with a new random solution.

$$P_{retained} = \frac{stability * P(S)}{P(S')}$$

where S' is the highest probability solution found in the run.

Regression controls how far the search is allowed to regress for each iteration. Regression is measured by comparing the probability mass contained in the current population with that measured from the previous population. If this ratio is less than the regression parameter, the current population is discarded and a new population is generated.

Population Size is the size of the population that is retained each iteration.

The *stability* parameter has the largest impact on the performance of the simulation during each iteration. A low value (close to 0) turns the process into a pure random search, while a high value results in a pure local optimization of the solutions in the original population.

The random nature of this search does not allow it to recognize the optimal solution; but as with Genetic Algorithms, the Stochastic Simulation is easily adaptable to anytime and anywhere operation. The best solution in the population is easily used as an anytime solution, and the received solution can be placed directly into the current population since it is as good or better than any existing member. Note, however, that the best solution is not guaranteed to survive until a better one is found; it could be selected for replacement at any time (this is unlikely, but possible). We use a best-so-far filter to ensure that the value of the distributed anytime solution does not decrease in value.

To analyze the performance of the algorithm, we need to first look at the algorithm's behavior on an individual member of the population; then we can determine the aggregate behavior for the entire population. For a given member S of the population, the probability that the solution S' generated from S by the next iteration of the algorithm will be better than the previous solution can be stated as:

$$P_{imp_x} = P_{retained}P_{opt} + (1 - P_{retained})P_{random} \tag{5.18}$$

P_{random} is the probability that a new solution will be chosen randomly from the solution space with a solution quality greater than the original solution:

$$P_{random} = P_{select}(S')P(P(S') > P(S)) \tag{5.19}$$

and P_{opt} is the probability that the solution can be improved in the local optimization step:

$$P_{opt} = 1 - \prod_{r \in R} \left[P\left(P(r'|\pi(r')) \prod_{C} P(c|\pi'(c)) > P(r|\pi(r)) \prod_{C} P(c|\pi'(c)) \right) \right] \tag{5.20}$$

where R is the set of all RVs and C is the set of children of the RV being considered r. In other words, it is the probability that there exists an RV r' with a marginal probability greater than the original solution for at least one RV. The left side of the inequality is the marginal probability of an RV r_i and its children r_j; the right side is the marginal probability of the same subset of RVs in the solution S. Since only one RV is modified in the local optimization, the probability that improvement can occur for one RV is independent of any other RV; so each RV can be treated as a separate event and the probability of the whole is the probability that at least one RV will improve.

This probability is dependent only on the position of each marginal probability within its distribution. Initially, the probability is high since the initial population is randomly generated; as the solutions are improved, the probability decreases until one of the local optima is reached. Due to the structure of the Bayesian Network, a change in value of one RV may have an effect on its neighbors in the network; therefore, each RV may need to be changed more than once to reach the local maxima. It is worth noting that according to Equation 5.18, the only time there is no possibility of improvement ($P_{imp_x} = 0$) is when S is the optimal solution. For any other solution in the population there is always the possibility of improvement; either from local optimization or from random selection.

When combining the individual member probabilities into a composite probability for the whole population, we need to consider the fact that the members are starting out at different solution quality levels and that the anytime solution for this algorithm is the best solution in the population. This means that a significant improvement in the quality of one member may still not improve the overall solution. Equation 5.18 needs to be restated to reflect the probability that the change to the member will improve the *overall* solution quality:

$$\begin{aligned} P_{imp_x}(S_i) &= P_{imp_x} P(P(S') > P(S_i)) \\ &= P_{retained} P_{opt} P(P(S') > P(S_i)) + \\ &\quad (1 - P_{retained}) P_{select}(S') P(P(S') > P(S_i)) \end{aligned} \tag{5.21}$$

where S_i is the best member in the population: the anytime solution. The added term reflects the likelihood that the solution produced by manipulating the member will be better than the global best solution. With everything relative to the same solution, the probability of improvement for the entire population is simply

$$P_{imp}(S_i) = \max(P_{imp_x}(S_i)) \tag{5.22}$$

over the individuals x in the population since each member is manipulated independently. In a cooperative environment, the incoming anytime solutions are placed directly into the population; thus Equation 5.22 also gives us the probability that the HySS will improve the overall system solution quality.

Unlike the probability of improvement; the level of improvement E_{imp} is not dependent on the parameters given to the HySS algorithm; instead, (as with the GA) it is dependent primarily on the parameters of the probability distributions for the conditional probability tables. The HySS algorithm always makes the one-random variable change which maximizes the improvement; this gives us an easily determined relationship:

$$E_{imp} = \frac{\overline{x} + \sigma}{\overline{x}} \tag{5.23}$$

5.2.4 Barrier Computation. The Barrier algorithm presents a significant difficulty to this method of analysis since it is not an anytime (or anywhere) algorithm. Initially, it would seem that this problem would make the algorithm unusable; but if we keep the intent of the analysis in mind, we can generate some usable results.

The purpose of the probability of improvement is to allow the controller to determine the algorithm that is most likely to contribute to the system's progress. Technically speaking, the probability of improvement for the Barrier algorithm is 1.0, since it will produce the optimal answer when it finishes; the problem is the time frame involved. For anything larger than a toy network, the Barrier algorithm takes more time to solve the network than most anytime algorithms take to produce a single solution, so we need a way to take the time into account in generating the probability of improvement. This is where the Barrier algorithm helps us out: it is fairly simple to get a close approximation of the time required for the algorithm to solve a particular network. One way would be to spread the probability of improvement for the entire solution process uniformly over the solution time. Given a time interval Δt, the probability can be calculated as:

$$P_{imp}(S) = \frac{\Delta t}{t_s} \qquad (5.24)$$

where t_s is the time for the Barrier algorithm to complete. This provides an indication of how close this algorithm will get to the solution in the specified interval.

Level of improvement is another difficult concept for this algorithm: We will end up with the best solution, but we do not know what the value of that solution is ahead of time. Since this factor is used to find the algorithm which has the most impact on the system's output quality, it has to be 1.0 (indicating no improvement) unless the algorithm will complete; then some arbitrarily large value can be specified to signal that it will finish.

5.3 Algorithm Interaction

With the analysis method for the different algorithms established, it's time to look at what the impact when the algorithms are used cooperatively. Intuitively, the combination should perform at least as well as the algorithms run individually (assuming no resource limitations); but the open issue is whether to expect an improvement over the individual algorithm performance. Our initial experiments confirmed this expectation in some, but not all cases. These initial results initiated the effort to determine which properties of the algorithms combine to produce better results as well as what causes the poor results. The goal is to be able to select good initial algorithm combinations in addition to recognizing algorithm combinations that do not work well.

The OVERMIND system provides the mechanism for the anytime solutions to be shared between the cooperating algorithms, allowing each algorithm to receive and utilize the current best solution. In general, the solutions received from the other algorithms can affect the execution of the algorithm itself or they can affect the way its anytime solutions are generated. The next step is to determine specifically how each class of anytime algorithms reacts to the shared information. Since only anytime and anywhere algorithms have the capability to produce and receive solutions, the Barrier algorithm has no capability to interact with the other algorithms and is not discussed here.

5.3.1 Exact Algorithms. The A* search uses the incoming solutions to set a lower bound on the search process. For the A* this unfortunately does not significantly reduce the run time since the search works by reducing the upper bound. It can, however, reduce the space requirement by eliminating segments of the search space that fall below the value of the received solution.

The greatest impact of the incoming solution on this algorithm is in the generation of anytime solutions. The use of the received solution as a template to fill in the incomplete parts of the partial solution instead of using a random or arbitrary template can significantly improve the quality of the completed solutions. Most exact algorithms expand their search progressively from either the roots or the leaves of the network; using the sets of RVs defined in Section 5.2.1, the difference between completing the solution arbitrarily and with the received solution is limited to the RVs in the F and U sets. Thus, the solution completed with the incoming solution is better than the solution completed with the arbitrary solution when

$$P(U_1|\pi(U_1))P(F_1|\pi(F_1)) > P(U_0|\pi(U_0))P(F_0|\pi(F_0)) \tag{5.25}$$

Once the quality of the incoming solution is higher than the arbitrary solution, the resulting completed solutions will be better.

5.3.2 Approximate Algorithms. The class of approximate algorithms typically work with complete solutions, and the GA and Stochastic Simulation are no exception. Incoming solutions are used directly by the algorithm to improve the algorithm's performance. Both approximate algorithms used here incorporate the incoming solution directly into the pool of solutions being manipulated. The presence of the higher probability solutions improves the algorithm performance if the new solutions are significantly better than the rest of the solutions in the pool.

Approximate algorithms by themselves are usually not extremely effective at finding optimal solutions, but they can be quite good at performing local optimization; this can be quite useful when used in combination with other (exact) algorithms.

5.3.3 Combinations. A system composed of combinations of dissimilar algorithms yields performance that is as good as or better than any of the component algorithms individually. With the different algorithms executing independently with no cooperation, the probability of improvement for the system at any given time is

$$P_{imp_{total}}(S_i) = 1 - \prod_j (1 - P_{imp_j}(S_i)), \tag{5.26}$$

which is the probability that at least one algorithm will improve. Unless the probability of improvement for all algorithms is zero, we always gain something by utilizing multiple algorithms. This confirms our initial intuition. We now show that Equation 5.26 also holds for systems of cooperating algorithms:

Theorem 1 *The probability of improvement for a system composed of multiple unique cooperating algorithms is as good or better than any of its component algorithms.*

Proof: For Equation 5.26 to be valid, $P_{imp_j}(S_i)$ for each algorithm must be independent:

- $P_{imp_j}(S_i)$ *is defined as a conditional probability*
- $P_{imp_j}(S_i)$ *is only conditioned on the anytime solution* S_i
- $P_{imp_j}(S_i)$ *is therefore* conditionally independent *for all algorithms since the value of* S_i *is given.*

Since $P_{imp_j}(S_i)$ is independent for all algorithms given S_i, Equation 5.26 still holds.

■

The anytime solutions being shared among the algorithms are a historical quantity: the probability of improvement is a prediction of the future performance based solely on the characteristics of the algorithm and the problem. This higher overall probability of improvement manifests itself as a steeper slope on the system's performance profile compared to the performance profiles of its components. This confirms our initial hypothesis and explains the performance improvements we observed, but the situations that did not result in a performance improvement still need further explanation.

The analysis of the algorithms in Section 5.2 indicated a similarity in performance characteristics between some algorithms. The GA and Stochastic Simulation have similar probability of improvement equations, which is not surprising due to the similar nature of the manipulations performed. While combining multiple instances of algorithms from the same class improves the performance by increasing the number of samples taken, the improvement due to the information sharing can only take place if the information being shared is not already known to the receiving algorithm. For example, if two identical GAs were being used, as soon as the first high-probability solution was shared, both populations would tend to converge to the same local optimum because the subsequent solutions would be grown from the first shared solution; the two instances of the GA are now virtually identical, resulting in an effective contribution to the system of only one algorithm. The overall system probability of improvement is unchanged, as is the system performance profile.

Performance improvement due to sharing information mostly occurs when the algorithm's performance characteristic is significantly different from the rest of the collection. One algorithm with a steep performance curve can boost the performance of a slower performer; or if the performance curves for two algorithms intersect, they alternate the roles of leader and follower. Obviously, algorithms from different classes have significantly different performance profiles; but in some cases, it is possible to significantly affect the performance of an algorithm through its parameters. In such cases, it may be beneficial to use two similar algorithms, but with different parameters.

As an example, consider the combination of an A* search and a Stochastic Simulation. If the target Bayesian Network's conditional probabilities exhibited a normal probability distribution with a moderate variance, the A* would initially produce low quality solutions because the heuristic and the unassigned RVs dominate both terms in Equation 5.8, resulting in a low probability of improvement. On the other hand, the Stochastic Simulation would initially show significant improvement in solution quality: both parts of Equation 5.22 are large because the typically low quality initial population leaves a lot of room for improvement. When the Stochastic Simulation generates its first anytime solution, it is passed to the A*, which splices it into its current partial solution. With the

A*'s optimal search, it is highly likely that the completed part of the solution is better than the same part of the solution generated by the Stochastic Simulation. The splice then generates a better solution, which is then passed back to the Stochastic Simulation for further improvement. Once the optimal solution is reached, the A* processes its open queue; but with its lower bound set to the optimal answer, many of the remaining branches are discarded until the only item left is the optimal answer. A highly accurate heuristic would benefit greatly here: once the optimal answer is found, fewer branches of the search tree would need to be explored to confirm that the optimal answer was found.

This analysis explains the mixed results of our initial experiments(37); all the combinations that showed an increase in performance were combinations containing at least one exact algorithm and at least one approximate algorithm. It was also frequently the case that such combinations, while not performing significantly better than the approximate algorithm by itself, confirmed the answer produced by the approximate algorithm.

5.4 Algorithm Model

For an algorithm model to be useful for performance prediction, it must capture the characteristics of that algorithm's behavior on the different instances of the problem domain. The analysis in the previous sections develops the concepts of *probability of improvement* and *level of improvement*, which together indicate whether the algorithm is likely to improve and how much it is likely to improve. The process of performing the analysis also produces as a side effect the characteristics of the problem domain which are parameters to the algorithm's behavior. These characteristics are represented below by the symbol $\mathcal{N}$; its presence indicates that each of the factors within the model is calculated using domain-dependent information, but that information is contained in the implementation of the model, not in the specification.

The two performance-related factors already mentioned could be used alone to select and evaluate algorithms and combinations thereof, but there are other factors that can be important in evaluating algorithms. One such factor is the time and space complexity of the algorithm on that particular problem instance. Traditional complexity analysis gives us a complexity value dependent on some characteristic of the problem, typically the dimensionality or size of the problem; but some algorithms also respond to semantic characteristics such as graph topology. Our assumption that the problem is large or complex enough to make it intractable somewhat reduce the need for complexity analysis, but it can still provide an indication of how far the algorithm might be able to progress towards the optimal solution.

In a distributed environment, one obvious approach is to skip the selection process and simply run all of the algorithms together at the same time, assuming that we have enough systems. While this approach has its merits in simplicity and could be utilized if the number of algorithms available were small and the systems were dedicated to this purpose, neither of these assumptions are likely to be true for a realistic scenario. For this reason, we want to make reasonable use of the resources available to us, and the number of systems being used must also be a factor in the selection and evaluation process.

In a dynamically configured system, one issue that needs to be addressed is the concept of thrashing: the continual, repetitive switching among a small set of configurations. This wastes system resources in sever ways, but the important one to us is that the algorithm cannot contribute to the progress of the system when it is initializing. This dead time during algorithm initialization should therefore be a factor for run-time evaluation.

While the *No Free Lunch* theorems(38) clearly state that past history of an algorithm's performance on a given problem (or even problem domain) has no bearing on its potential future performance, it still needs to be a factor in the run-time evaluation. The premise under which we operate is that the problem is intractable and that the algorithms we use are anytime; the focus is on short-term performance. If an algorithm is not producing solutions in the short term which advance the output solution quality, they are wasting resources and should be candidates for removal from the current configuration if their performance factors do not warrant their presence in the system. This does not necessarily mean they will never get used again, but merely that they are not effective at this time.

Following is a more precise definition of the seven model factors. The model factors are parameterized on these quantities:

- **s**: The current highest quality solution produced by the system. For initial algorithm selection, this is some initial solution appropriate to the problem.
- **t**: The total running time of the algorithm(s) being evaluated. For initial algorithm selection, none of the algorithms has yet run, so it is set to zero.
- **Δt**: The time interval over which we want to evaluate the algorithms.

Again, notice the lack of domain-dependent information. It is at this level in the model hierarchy where we declare the domain independence: below is the domain-dependent application of these model factors, but above is our domain-independent control model.

5.4.1 Factor Description.

Level of Improvement ($\mathcal{C}(s, t, \Delta t)$) *How rapidly will the algorithm approach the solution?* To get the system off to a fast start, we prefer to use algorithms that progress rapidly towards the optimal solution right from the start. This factor is an extension of the *level of improvement* discussed earlier; the level of improvement analysis is applied over the time interval $[t, t + \Delta t]$ instead of a single iteration of the algorithm:

$$\mathcal{C} = f(\mathcal{N}, s, t, \Delta t)$$

where s is the current solution at time t and the function f is the algorithm-specific method for applying E_{imp} over the time interval. Ideally, this approximation should be more accurate for smaller values of Δt, with the error increasing as Δt increases.

Probability of Improvement ($\mathcal{I}(s, t, \Delta t)$) *How likely is this algorithm to produce an improved solution in the given interval?* We want to select algorithms which are more likely to produce some improvement over the current solution quality in the system. This concept was discussed earlier, and is now applied over the time interval $[t, t + \Delta t]$ as with the level of improvement:

$$\mathcal{I} = g(\mathcal{N}, s, t, \Delta t)$$

where g models the effect of the time interval on the specific algorithm involved.

Time Complexity ($\mathcal{T}$) *How long will it take this algorithm to solve this problem?* For obvious reasons, we prefer to use algorithms with a lower time complexity to solve our problem; this factor quantifies the time complexity by specifying the constant that is implicit in traditional worst-case complexity analysis of algorithms:

$$\mathcal{T} = C_{time}O(\mathcal{N})$$

$\mathcal{N}$ represents the characteristics of the given problem instance within the problem domain. The constant C_{time} is dependent on the specific implementation of the algorithm and the speed of the hardware used; it could be determined empirically, but it also could be learned as the system is used. In general, this factor should be an estimate of the time required to completely solve the problem.

Space Complexity ($\mathcal{S}$) *How much space will this algorithm need?* It is also preferable to use algorithms with smaller space complexity. We similarly quantify the space complexity:

$$\mathcal{S} = C_{space}O(\mathcal{N})$$

This constant is also dependent on the specific implementation of the algorithm, but is less dependent on the hardware used. As with time complexity, this factor should represent the space requirements for completely solving the problem.

Resource Usage ($\mathcal{R}$) *How much of the overall system resources will be used by this algorithm?* Typically, we also want to minimize the amount of system resources used to solve the problem:

$$\mathcal{R} = C_{res}g(CPUs)$$

The scope of this factor can be as broad as the number of systems or processors used, or as specific as the amount of memory and disk space needed. Whatever is used as the measure, the quantification should represent the resource impact of using this algorithm.

Reconfiguration Cost ($\mathcal{G}(\mathbf{u})$) *How much does it cost to start this algorithm?* It takes time and effort to reconfigure a system, so we want to avoid changing the system unless the difference is significant. This factor represents the amount of time required to start the algorithm, plus an additional penalty value if desired; if the algorithm is currently running, the value is a non-zero constant.

Interaction Frequency ($\mathcal{F}(\mathbf{u})$) *How frequently has this algorithm contributed to the progress of the system?* We want algorithms that actively participate and share solutions. This value is a statistic gathered at runtime and represents the percentage of the total generated solutions which advanced the solution quality of the system. For algorithms which have not started, we set the value to an optimistic 100%.

Note that only one of these criteria is historical; our models rely primarily on predictions of future behavior. Past performance is not necessarily useful in predicting future behavior[1], especially when multiple algorithms are used. The only other historical information used is the current solution quality; however, since it is the current solution for the entire system, it is not necessarily a measure of past performance when evaluating any specific algorithm.

5.4.2 Algorithm Combinations. Each of the model factors are evaluated for each individual algorithm for the given problem instance $\mathcal{N}$; however we also want to evaluate different combinations of algorithms. There are two approaches we can take: analyze each possible combination of algorithms, producing a separate model for each combination; or combine the results of the models for the individual algorithms, producing a composite result.

A simple method for calculating a combined utility value uses the extreme values for the different factors. The convergence rate of the combined system is driven by the performance of the best algorithm, so we must use the highest level of improvement; but the algorithm with the worst time and space complexity still drives the complexity of the overall system. We cannot, however, do this with the resources, since the sum of the resources is used for the combined system. This gives us a more general definition of our

[1]This concept is a major component of the No Free Lunch theorems(38).

factors:

$$\begin{aligned}
\mathcal{C}' &= \max_i \mathcal{C}_i \\
\mathcal{I}' &= \max_i \mathcal{I}_i \\
\mathcal{T}' &= \max_i \mathcal{T}_i \\
\mathcal{S}' &= \max_i \mathcal{S}_i \\
\mathcal{G}' &= \max_i \mathcal{G}_i \\
\mathcal{F}' &= \min_i \mathcal{F}_i \\
\mathcal{R}' &= \sum_i \mathcal{R}_i
\end{aligned}$$

This method of producing a composite utility is useful if the algorithms do not interact with each other, or if the effects of the interaction are minimal.

If the algorithm interaction (through the sharing of anytime solutions) has a significant impact on performance, the simplified model presented above is not effective in recognizing which combinations benefit from such interaction. We need to replace the level of improvement and probability of improvement factors with new terms reflecting the performance characteristics of all algorithms in the combination, this time accounting for the interaction.

Unfortunately, the calculation of this new term is not as simple as for a single algorithm. Since the interaction between the component algorithms occurs throughout the time interval t, a simple combination of the convergence factors for the individual algorithms is not adequate. We can, however, simulate the interaction by iteratively calculating the convergence factors at approximately the same rate that the algorithms would actually exchange solutions; this rate is determined by obtaining for each algorithm the length of time between production of solutions. At each iteration, each algorithm model would account for the received solution in its prediction for the next iteration as shown in the algorithm in Figure 5.2. The best score received from any algorithm is retained for the next iteration.

```
function $C'(c, s_0, t_0, \Delta t)$
    let $t_e \equiv$ algorithm exchange time

    $\hat{t_e} = \max\{t_{e_i} | i \in c\}$
    $s' = s_0$
    $t' = t_0$
    while $t' < (t_0 + \Delta t)$ do
        $s' = \max\{C_i(s', t', \hat{t_e}) | i \in c\}$
        $t' = t' + \hat{t_e}$

    return $s'$
```

Figure 5.2 Algorithm for calculating convergence factor for combinations with multiple algorithms. c represents the combination and i refers to the individual algorithms within the combination.

The probability of improvement factor combines in a much more straight forward manner when combining algorithms. The probability of improvement for the algorithm combination is simply the probability that at least one of the algorithms will improve:

$$\mathcal{I}'(s, t, \Delta t) = 1 - \prod_i (1 - \mathcal{I}_i(\mathbf{x}, t, \Delta t))$$

for all algorithms in the combination. This is possible due to the fact that the probability of improvement for each algorithm is independent of the others given a particular state.

5.4.3 Model Application. The goal when applying the model's performance factors is to provide an approximate answer quickly; therefore, we need to use equations with very small computational complexity. The approximation should be reasonably accurate over short intervals, possibly at the expense of long-term accuracy. Constant time expressions are obviously best, but are not always usable; however, the complexity should be kept to a minimum whenever possible. For similar reasons, we also need to use characteristics of the network that are easily determined.

When constructing the model factors, it is important to capture the features of the algorithm that make it useful. For example, Genetic Algorithms are generally insensitive to network topology, but extremely sensitive to the probability distribution of the conditional probability tables within the Bayesian Network. The Barrier algorithm, on the other hand, is very sensitive to topology, but is not affected at all by the probability distribution of the individual random variables.

For approximate models, the time complexity is problematic: approximate algorithms have no termination condition, so the time complexity is by definition infinite. We do not want to eliminate approximate algorithms, since they have proven in practice to be quite effective in finding high-quality solutions. Our model can address this problem by using the time complexity of *one iteration* of the approximate algorithm in place of the complexity of the entire algorithm run. This still emphasizes lower cost algorithms without eliminating an entire class of effective solution methods.

Care also needs to be taken to ensure that the models are constructed such that comparisons between running algorithms and non-running algorithms produce the desired result. New algorithms will be added to the system only when the running algorithms have stopped contributing or if the non-running algorithms can make a significant contribution to the system performance.

Examples of the application of the algorithm model to the algorithms used for belief revision on Bayesian Networks can be found in Appendix B.

5.5 Summary

The algorithm model developed in this chapter captures the performance of the algorithm, parameterized by the characteristics of the problem domain. This model is developed analytically to avoid the extensive amount of time required to do empirical profiling; yet due to its approximate nature, it can be extremely simple to evaluate. The model focuses primarily on predictive modeling. While the No Free Lunch theorems conclude that past performance is not useful in algorithm selection or evaluation in the long term, our limited use of historical information allows us to evaluate short-term efficiency of algorithm combinations. This model forms the foundation for the overall control model, providing the means for the algorithm selection and run-time control processes to evaluate the candidate combinations.

VI. Initial Algorithm Selection

The goal of the initial algorithm selection phase (Phase 3 from the research plan) is not to select the optimal set of algorithms to solve the entire problem. As discussed before, such a task would be difficult, if not impossible. We also face the added difficulty that it is not always possible to solve the problem in the time we have available. With this in mind, we want to use models of the algorithms which are easy to compute; all we need to do is select a reasonably good set of algorithms to get the system started, and then the run-time controller will take over to finish the job. Obviously, the more accurate the models are, the better the system performance will be at first; but as we will see later, this is not essential to obtaining a good result from the dynamic phase of the system.

6.1 No Free Lunch

In a recently published paper(38), Wolpert and Macready present a set of theorems called the *No Free Lunch Theorems* concerning algorithm performance. To summarize, their conclusions fall mostly into two categories:

- There is no benefit in choosing one particular algorithm over another in the general case. The assumption in this case is that the algorithm does not know about or take advantage of any domain-specific knowledge.
- Given a specific algorithm, its past performance on problems within the problem domain are not useful in predicting performance on other problems in the same domain. (This concept was discussed in the previous chapter.)
- Without future performance knowledge about the algorithms involved, no algorithm selection method will perform better than any other method.

In many ways, these conclusions are common sense, but they serve to focus attention on what information is actually required for the algorithm selection process to be successful: knowledge about the algorithm.

The algorithm models developed in Chapter V capture the essential characteristics of the algorithm's behavior over the problem domain. This gives the selection algorithm what it needs to successfully make its decisions.

6.2 Selection Process

One fact that can simplify the static algorithm models is that we are only interested in the early stages of the algorithm's behavior. We only need to get the solution process started; the dynamic controller will take over after a small initial interval Δt. Since we are are using approximate models of the algorithms, we can expect their accuracy to decrease as Δt increases; this is not a problem because Δt will typically be small so that the controller can take over.

Only four of the model factors developed in the previous chapter are useful in static selection: *Time Complexity*, *Space Complexity*, *Level of Improvement* and *Resource Usage.* These factors are calculated prior to execution, so there can be no dependence on run-time system characteristics, but rather on the characteristics of the problem being solved and the algorithms themselves.

The selection process has two objectives:

1. maximize system performance by selecting algorithms which will produce the highest quality solution at the end of the time interval Δt;
2. minimize the system resources needed, specifically the number of processors used and memory requirements.

The first objective is measured primarily by the level of improvement factor, but the time complexity factor can also be used as an indicator of how complete the algorithm will be in the interval. The resource objective is indicated by the space complexity and resource usage factors.

For obvious reasons, these objectives are typically conflicting. The decision-making process must accommodate this conflict when determining the best algorithm to use. The technique known as *multicriteria optimization*(33) can be used in this case to determine which algorithm combination to use. The optimization is done as a two-step process: first, determine the *Pareto-optimal* set of configurations; then use a linear weighted sum to make the final selection.

The multicriteria optimization problem can be stated as finding the best combination $\hat{c}$ from the set of all combinations C such that:

$$\begin{aligned} \hat{c} &= \arg\max_{c \in C} F(c) \qquad \text{where} \\ F(c) &\equiv \langle F_1(c), F_2(c), F_3(c), F_4(c) \rangle = \langle \mathcal{C}'_c, \mathcal{T}_c^{-1}, \mathcal{S}_c^{-1}, \mathcal{R}_c^{-1} \rangle. \end{aligned} \tag{6.1}$$

Notice that the reciprocal of the time, space and resource factors is being maximized since these are minimization objectives. Unfortunately, as is typical with this type of problem, the same value of c rarely maximizes all four objectives at the same time, so the traditional approach is to first find the Pareto-optimal set of combinations. This process eliminates the obviously inferior combinations by subjecting each combination's vector of factors to a partial ordering $\prec$, defined as:

$$\begin{aligned} F(c_0) < F(c_1) &\quad \text{if and only if} \quad F_i(c_0) < F_i(c_1) \quad \forall i \in R \\ F(c_0) \leq F(c_1) &\quad \text{if and only if} \quad F_i(c_0) \leq F_i(c_1) \quad \forall i \in R \\ F(c_0) = F(c_1) &\quad \text{if and only if} \quad F_i(c_0) = F_i(c_1) \quad \forall i \in R \\ F(c_0) \prec F(c_1) &\quad \text{if and only if} \quad F(c_0) \leq F(c_1) \quad \text{and} \quad F(c_0) \neq F(c_1) \end{aligned} \tag{6.2}$$

where R is the dimensionality of our vector, in this case four. With this partial ordering, we can define our Pareto-optimal set P as:

$$P = \{\hat{c} \in C | F(c) \not\prec F(\hat{c}) \;\; \forall c \in C\} \tag{6.3}$$

It is possible (in fact, highly likely) for the set P to contain more than one combination; we then use a linear weighted sum to determine the combination $\hat{c}$ out of P to use:

$$\begin{aligned} &\text{Given:} \quad s, t, \Delta t \\ &\qquad \hat{c} = \arg\max_{c \in P} U(s, c, t, \Delta t) \end{aligned} \tag{6.4}$$

where the utility function U is:

$$U(s,c,t,\Delta t) = M_{conv}C_c'(t,\Delta t,s) + M_{time}\frac{1}{\mathcal{T}_c} + M_{space}\frac{1}{\mathcal{S}_c} + M_{res}\frac{1}{\mathcal{R}_c} \tag{6.5}$$

The factors used to calculate the utility of a given combination c are specific to that combination, hence the subscripts. The coefficients (M) allow the user of the system to specify a preference as to which of the factors to favor in this final determination. The combination with the highest utility is the best compromise (subject to the user's preferences) between performance, complexity and resource usage. Given equal utility values for the best combinations, some arbitrary method may be used since those combinations should have equivalent performance.

6.3 Summary

This selection method provides the means to choose an initial algorithm combination for the system, balancing the conflicting objectives of performance versus resource efficiency. In general, the process of selecting an algorithm without any domain knowledge does not provide any benefit. However, the use of the algorithm models presented earlier gives the selection process the added information it needs to successfully make its decisions. The use of multicriteria optimization allows the system to select the best combination subject to the priorities set up by the user.

VII. Run-time Control

The final phase of the research addresses the issues of managing the cooperative processes during execution. For the purposes of discussion here, we assume that the problem is large or complex enough that finding the optimal solution is not feasible. This opens the door for the use of anytime algorithms, which provide solutions of increasing quality throughout the duration of their execution.

We have already discussed the initial selection of algorithms; but since our selection methods depended on approximate models, there is no assurance that the actual performance will match what we predicted. The system needs a means of monitoring the execution of the selected algorithm(s) and ensuring that the system as a whole speedily converges towards the optimal answer. This dynamic control of the inference system needs to address three primary issues:

- maximize the rate of increase in solution quality,
- resource limitations or constraints,
- provide a stable control framework.

The first two goals are typically contradictory, or at least do not always favor the same approach; the controller must balance these two goals and take a compromise position between them.

7.1 Plant Model

When developing a discrete control structure, the traditional approach is to use the feedback model shown in Figure 7.1. The plant is modeled by equations of the form(8):

$$\mathbf{x_{t+1}} = f(\mathbf{x_t}, \mathbf{u_t}) \tag{7.1}$$

$$\mathbf{y_t} = g(\mathbf{x_t}) \tag{7.2}$$

The first equation maps the current *state* of the plant ($\mathbf{x}$) and the current *control input* ($\mathbf{u}$) to the next state of the plant. The second transforms the state into the current *output* of

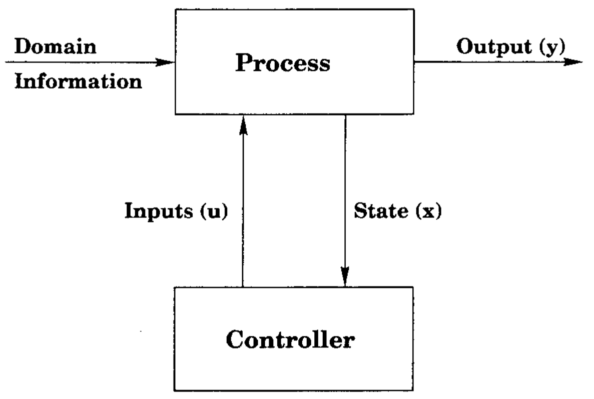


Figure 7.1 Fundamental closed-loop controller

the plant (**y**). In our situation, the distributed system itself is the *plant*; its initial state is determined by the domain information specific to the problem instance being solved and the output is the current anytime solution. To complete the model, the control inputs to the plant need to be identified as well as the composition of the state of the plant. At that point, we can develop the control strategy and assemble the system.

7.1.1 Control Inputs. Given the nature of our inference system, there are different levels at which we could exercise control over the process: control the behavior of the individual inference algorithms, control the interaction between the algorithms in a composite system or control the composition of the system itself. The first method requires detailed knowledge of the internal workings of each algorithm. While this could result in a very high-performance and stable system, our goal is to provide an algorithm- and domain-independent architecture for distributed problem solving. There is also the problem that some algorithms are not structured to allow any sort of external control; deterministic algorithms typically are this way.

Controlling the interaction between algorithms is definitely at a higher level, but would most likely prove to be counter-productive. It was clearly shown earlier that interaction between algorithms cannot result in decreased performance.

This leaves the highest level of control: managing the composition of algorithms in the system. This provides direct control over the process while still avoiding a direct dependence on a specific problem domain. There are five basic actions involved in managing the inference system:

1. **Stop the system:** we found the optimal answer (we do get lucky once in a while) or an answer that is acceptable.
2. **Stop the system:** we have run out of time or resources.
3. **Stop one algorithm:** either the algorithm has stalled, or is not performing well enough to justify leaving it running.
4. **Start one algorithm:** introduce a new algorithm to the system to increase the level of performance.
5. **Restart with new combination:** try again with a completely new set of algorithms.

The first two actions provide our termination action – either by having a way of recognizing the optimal solution when it is generated, or by using some form of utility value to determine that further efforts would not be cost-effective. Starting and stopping individual algorithms can be used to manage the system's makeup; but with the interaction between the algorithms, each algorithm's performance is no longer independent and we cannot manage them individually. We therefore push the abstraction one level further and consider each combination of algorithms as a unit and make our decisions at that level, conveying these actions to the plant through a vector of Boolean variables representing the status of each algorithm:

$$\mathcal{U} \equiv \{\mathbf{u} = \langle A_1, A_2, \ldots, A_n \rangle | A_i \in \{0,1\} \forall i \in 1 \ldots n\}.$$

7.1.2 State. The controller's purpose is to take actions that move the plant's state into a region within the state space to meet the controller's objectives. The state of the plant must contain enough information so that the controller can decide the action to take next and evaluate the progress of the system. For our composite distributed system, the state contains the following:

- the current best solution produced by the system as a whole

- the state of each algorithm in the system (running or not)

or, more formally:

$$\mathcal{X} \equiv \mathcal{S} \times \mathcal{U}$$

, where $\mathcal{X}$ is the state space of our system, $\mathcal{S}$ is the set of all solutions in the problem domain and $\mathcal{U}$ is the set of all possible control inputs, representing all the possible configurations of the system.

The controller needs to know about each algorithm in the system to determine whether the algorithm should be added to the system, removed from the system or left alone. The current best solution is the progress measure; it is used by the controller to evaluate the progress of each algorithm, as well as the entire system.

7.1.3 Final Plant Model. The general plant model equation in 7.1 can now be expanded as the following:

$$\begin{aligned}\mathbf{A} &: \mathcal{X} \rightarrow \mathcal{S} \\ \mathbf{B} &: \mathcal{U} \rightarrow \mathcal{U} \\ \mathbf{x_{t+1}} &= \langle \mathbf{A}(\mathbf{x_t}), \mathbf{B}(\mathbf{u_t}) \rangle \end{aligned} \qquad (7.3)$$

where the relation **A** models the affects of the inference algorithms on the solution component of the state and the relation **B** uses the control inputs to reconfigure the system as directed by the controller. To complete the model of the plant, we expand Equation 7.2 to provide an output projection **C** which extracts the solution from the state vector:

$$\begin{aligned}\mathbf{C} &: \mathcal{X} \rightarrow \mathcal{S} \\ \mathbf{y}_t &= \mathbf{C}(\mathbf{x_t}) \end{aligned} \qquad (7.4)$$

7.2 Control Strategy

The goal of the controller is to determine at any given point in time what system configuration will produce the best expected performance. Breaking this down, we have four separate (but related) objective functions:

1. maximize the output solution quality
2. maximize the expected performance of the system
3. minimize resource usage
4. minimize control actions

The first objective also happens to be the goal of the distributed process itself; but the controller has no direct way of affecting the solution quality. Instead, it can manipulate the composition of the system, starting and stopping algorithms as desired in an attempt to affect the output solution quality. We need a *Control Law* to produce the next control input for the plant when given the current output state of the plant:

$$\begin{aligned} \mathbf{D} &: \mathcal{X} \rightarrow \mathcal{U} \\ \mathbf{u_{t+1}} &= \mathbf{D}(\mathbf{x_{t+1}}) \end{aligned} \tag{7.5}$$

Remembering that the control inputs to the plant consist only of configuration changes, the control law reduces to selecting appropriate algorithms based on the current configuration and solution quality. We use all seven factors from the algorithm model to evaluate the currently running algorithms in addition to evaluating new algorithms. This evaluation is accomplished in a similar fashion to the initial algorithm selection by using multicriteria optimization to find the optimal compromise between the different objectives. As before, the level of improvement factor is maximized, but for run-time evaluation, the probability of improvement and interaction rate also need to be maximized. The time and space complexity factors are still minimized in addition to the resource usage and restart cost factors.

As with the initial algorithm selection, we determine the *Pareto-optimal* set P of controls, then maximize a linear weighted sum of the above factors to determine the best control $\hat{\mathbf{u}}$. Thus, our control law from Equation 7.5 is as follows:

$$\begin{aligned} \mathbf{D}(\mathbf{x}) &\equiv \hat{\mathbf{u}} = \arg\max_{u \in \mathcal{U}} F(u) \qquad \text{where} \\ F(u) &\equiv \langle F_1(u), F_2(u), F_3(u), F_4(u), F_5(u), F_6(u), F_7(u) \rangle \end{aligned} \tag{7.6}$$

$$= \langle \mathcal{C}'_u, \mathcal{I}'_u, \mathcal{F}_u, \mathcal{T}_u^{-1}, \mathcal{S}_u^{-1}, \mathcal{R}_u^{-1} \mathcal{G}_u^{-1}, \rangle.$$

Using the ordering in Equation 6.2, the Pareto-optimal set is described by:

$$P = \{\hat{u} \in \mathcal{U} | F(u) \not\succ F(\hat{u}) \ \ \forall u \in \mathcal{U}\}. \tag{7.7}$$

The selected algorithm combination is decided by optimizing the utility equation over the Pareto-optimal set:

$$\begin{aligned} \text{Given:} \ \ & \mathbf{x} = \langle \hat{s}, u \rangle, t, \Delta t \\ & \hat{u} = \arg\max_{u \in P} U(\hat{s}, u, t, \Delta t) \end{aligned} \tag{7.8}$$

and the utility function becomes the following:

$$\begin{aligned} U(s,u,t,\Delta t) \ = \ & M_{imp} \mathcal{I}_u(s,t,\Delta t) + M_{conv} \mathcal{C}_u(s,t,\Delta t) + M_{freq} \mathcal{F}_u(u) + \\ & M_{time} \frac{1}{\mathcal{T}_u} + M_{space} \frac{1}{\mathcal{S}_u} + M_{res} \frac{1}{\mathcal{R}_u} + M_{conf} \frac{1}{\mathcal{G}_u(u)} \end{aligned} \tag{7.9}$$

Again, we insert weighting factors M to allow the user to specify preferences. Arbitrary methods are used here as well to break any ties.

7.3 Controllability, Stability and Optimality

As discussed in Chapter II, for the controller to be effective, certain issues must be addressed: controllability, stability and optimality. Each of these concepts is discussed in more depth below.

7.3.1 Controllability. The simple definition of controllability is that the controller must be able to transition the plant from any state to the set of target states within a finite amount of time. Based on the definition of the state presented earlier, the set of target states is defined as the states which contain the configuration which optimizes the the utility function, regardless of the solution. This definition flows directly from the objectives of the run-time controller.

Controllability is a necessary condition for the controller to have a desired effect on the behavior of the plant. In the plant model, the component of the model that responds to control input and affects the system configuration is the relation $B(\mathbf{u})$ from Equation 7.3. Since it simply sets the configuration to whatever the control input specifies, there are no limits on transitions between configurations and our system is *completely controllable.*

7.3.2 Stability. While plant stability is not required for a successful controller, it is desirable and can simplify the design of the controller. Simply stated, a plant is stable if a small change in the input does not cause a massive change in the output of the system(8). While it is difficult to quantify the stability of the system due to the composition of the plant and the state, we can make some qualitative observations to indicate that our system is stable:

- The only component of the state that can be changed by the system itself is the solution quality; the configuration can only be changed by the controller. Therefore, the configuration component of the state does not change except at the direction of the controller.
- The primary measure of system performance (solution quality) is monotonically non-decreasing: while a configuration change could result in a massive increase in this measure (which is desirable), the worst that could happen is that no further increases would be produced.

From these observations, we conclude that the system is stable.

7.3.3 Optimality. Optimality is desired in a controller for the primary reason that with an optimal control, the actions taken by the controller *optimize* the controller's objectives, not just *satisfy* them. For a controller to be an optimal, two conditions must be met:

1. There must be a scalar-valued function U which measures the performance of the controller;
2. There must be an *admissible control* which optimizes U.

The first condition is met by Equation 7.7, the objective function. The second condition is met trivially; all control actions are admissible since the plant is completely controllable. The evaluation process itself optimizes U using multi-criteria optimization techniques, thus our control is optimal.

7.4 Computational Issues

Unfortunately, this is not an ideal world; it takes time to calculate our evaluation factors, and the plant does not respond instantly to our control inputs. In a physical system, continuous monitoring and control is frequently necessary to be able to control the output. In our case, the output changes resulting from our control frequently lag significantly behind the control actions themselves.

These realities force the controller to constrain the amount of interaction it has with the system; it needs to intervene frequently enough to keep the system from going off the preferred course, yet not so frequently that the system does not have time to respond to the control actions.

What needs to be done is to set the monitoring time interval so that the system has time to react before we try to evaluate the results. Looking at the inputs to the performance evaluation factors, it is clear that these values only change when a new best solution is found or when the system configuration changes. This sets an obvious minimum interval for re-evaluation. There needs to be a maximum interval, however, so that the controller can interrupt a system that is no longer producing any results.

When new algorithms are started, the sample interval needs to be temporarily lengthened so that the algorithm has time to start. If we assume that the output changes within k iterations of starting a new algorithm, the monitoring should be limited so that a re-evaluation is not performed until k iterations of the *slowest* new algorithm in the current configuration. This way, the system has time to settle into the new configuration before the next evaluation.

7.5 Summary

The control model developed in this chapter provides the methodology to direct the progress of the distributed system, compensating for and correcting any configuration errors introduced by the use of approximate models. Discrete multicriteria optimization is used to balance the system configuration between conflicting objectives: system performance, resource utilization and control stability. The design of the system provides much of the stability and controllability needed by the controller. The resulting control model is independent of any problem domain, yet still uses the algorithm models to direct the distributed system to a configuration with the highest expected performance balanced with reasonable resource utilization.

VIII. Experimental Analysis

The overall goal of our experiments is to demonstrate a performance improvement when using the static selection and dynamic control versus an arbitrary single algorithm system. We divided this goal into objectives to demonstrate the performance of a particular component of the system (or the entire system):

1. static selection alone;
2. dynamic control alone;
3. static selection and dynamic control together.

8.1 Experiment Setup

Our test environment utilizes a heterogenous network of Sun workstations (both Sparc 20 and UltraSparc I) and Linux workstations (200 MHz Pentium Pro). We used OVERMIND as the framework for our distributed system (see Appendix IV). The static selection and dynamic control functionality was incorporated into a *consumer task*.

The experiments were conducted using 75 randomly generated Bayesian Networks. The networks contain from 30 to 500 random variables, each having 2 or 3 possible states. The connectivity between the random variables was kept moderate, with the number of arcs typically ranging between twice and three times the number of random variables. The conditional probability tables were generated with varying distributions, from extremely flat (small variance between entries) through extremely spiked (most entries close to zero, with one nearly 1.0).

8.2 Static Selection

We used this methodology to select from five different algorithms: a Genetic Algorithm (GA), A* search with two different heuristics: best-first and shared-cost, a hybrid stochastic search (HySS) and a Barrier Algorithm.

Figure 8.1 shows how the selected combination performed in comparison with the other combinations for that sized network. We weighted the evaluation to favor perfor-

Network	**Initial Time Interval (sec)**						
Size (RVs)	**10**	**20**	**50**	**100**	**200**	**300**	**500**
30	.75	.83	.67	.67	.67	.67	.67
50	.83	.83	.67	.75	.92	.67	.67
100	.25	.42	.58	.67	.67	.92	.92
200	.08	.25	.42	.67	.58	.83	.58
300	0	0	.27	.55	.45	.58	.50
500	0	0	0	.57	.56	.78	.78

Figure 8.1   Success rates for algorithm selection.

mance (M_{conv} = 100) while leaving the rest of the weights at 1.0; the numbers reflect the success rate of the selection process. Our definition of success is when (1) the algorithm(s) selected were contained in the combination that produced the best solution at that time *and* (2) those algorithm(s) were the ones contributing the best solutions. Clearly, in most of the cases, the selection process chose an algorithm combination that matched the best-performing combination.

Since the goal of the static selection process is to select a configuration to get the system off to a good start, picking the best performing combination is not essential. On the other hand, the better the initial configuration performs, the easier it will be for the dynamic controller.

Looking again at the results, notice the increase in the success rate as you move from left to right. This is a result of the varied cycle times for the different algorithms. This effect can also be noticed when moving from top to bottom. As the network size increases, the cycle time for the algorithms increase; with the approximate models, it may take several cycles for the models to accurately predict the algorithm's performance. What is not immediately obvious is why the success rates fall off at the right edge of the table, especially for the smaller networks. This result is caused by the "near-sighted" models. Accuracy falls off as time increases, causing inaccurate predictions for the larger interval times relative to the size of the problem.

8.3 Dynamic Control

Our meta-level objective for the run-time control process is to keep the algorithms running to make the most progress towards the optimal solution. Thus, one measure of success could be how fast we obtain a particular solution quality, or possibly what solution quality we obtain at a given time. Lacking any absolute objectives for either measure, we use the best un-controlled run as our benchmark. We are successful if the progress of the controlled system approaches or exceeds that of the best uncontrolled run.

Since performance was used to select the benchmark, our first test is to evaluate the performance of our system with the weighting factors set to favor the performance factors (convergence rate and probability of improvement). The results of this comparison are shown in Figure 8.2. These runs were made on a network with 50 random variables and a flat probability distribution. The weights for the convergence rate and improvement factors are set to 100; all others were left at 1.0. The solid curve shows the performance of the controlled system, and the dashed curve is the performance curve for the benchmark run. The performance of the two systems is nearly identical; the only difference in the two curves is due to the stochastic nature of the GA tasks involved.

Quantitatively, the objective for the controller is to move the state of the system (the configuration of algorithms) into a region to maximize the objective function. Whether this meets our performance objective is entirely dependent on the weighting factors supplied, the actual performance of the algorithms and the problem domain. As an example, Figure 8.3 demonstrates how the weighting factors affect output solution quality. This plot compares the performance of three runs which differ only in the relative weights assigned to the resource factors. By increasing the weight of the resource factors, more priority is given to reducing the number of concurrent algorithms being used (see Figure 8.4), resulting in a reduced output solution quality. The controller's objective is met: while the solution quality is not as high as it would have been with unlimited resources, it still is as high as the controller can push it within the constraints imposed upon it.

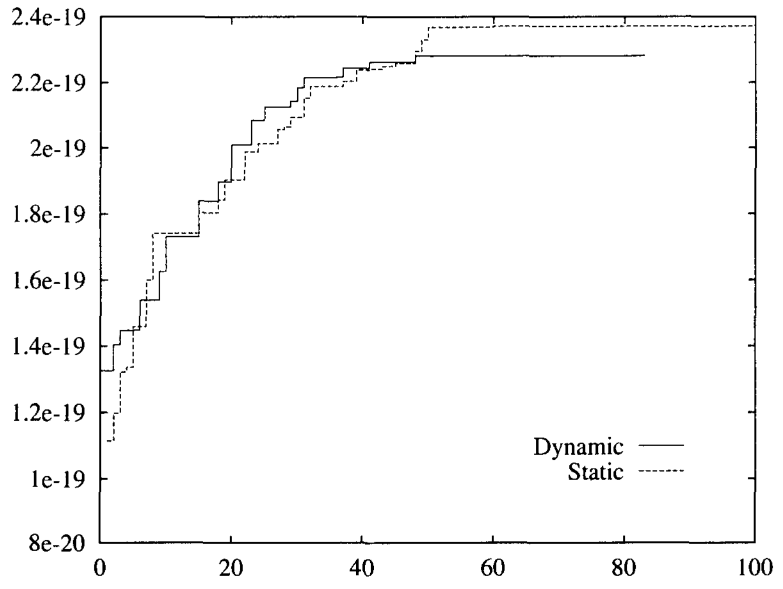


Figure 8.2 Comparison between un-controlled system and controlled system weighted for best performance.

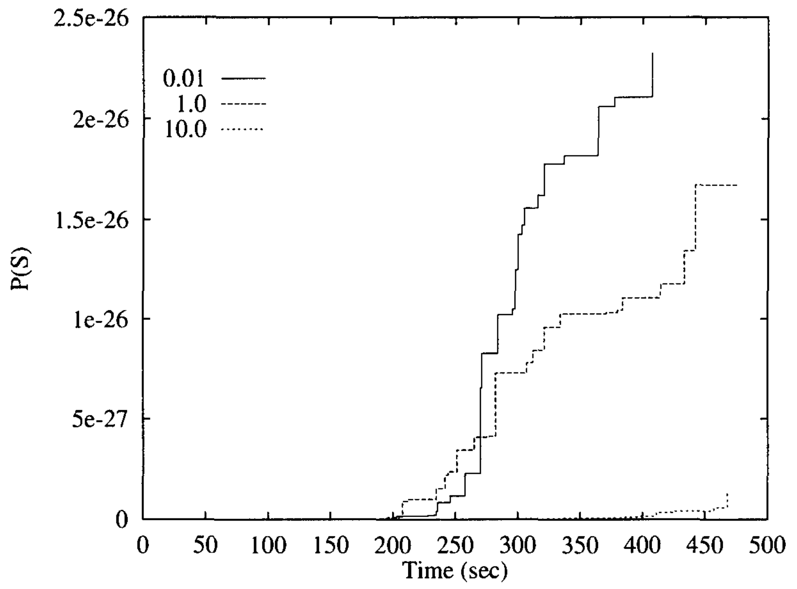


Figure 8.3 Effects of three different resource weighting factors on system solution quality.

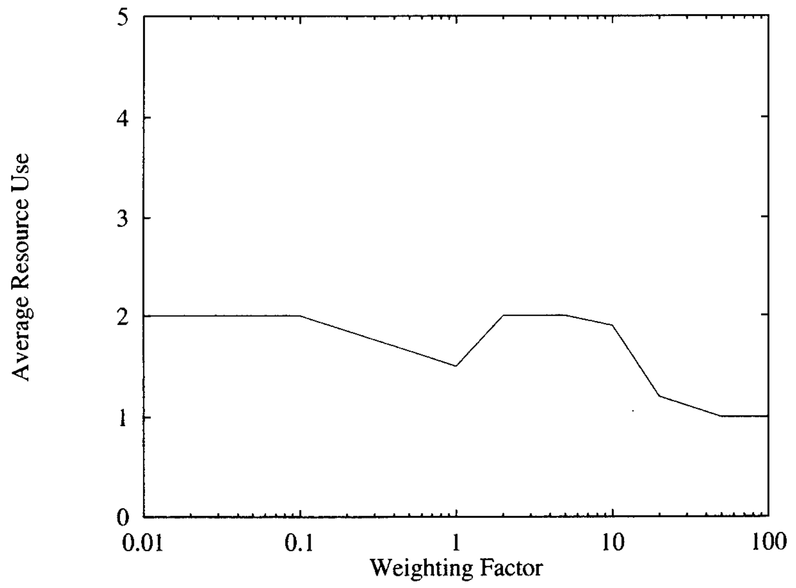


Figure 8.4 Effects of resource weighting factors on resource usage.

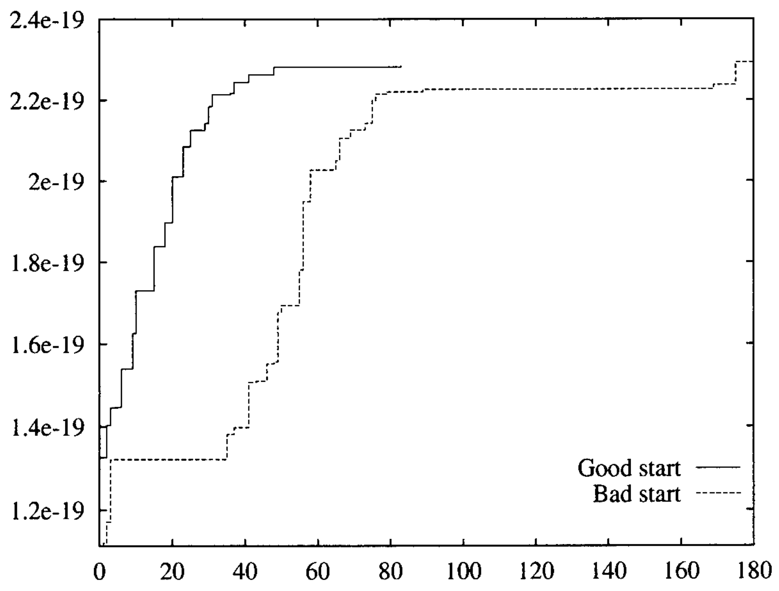


Figure 8.5 Effects of start-up configuration on solution quality.

8.4 Total System

As has been mentioned previously, the dynamic controller can take the system and move it into an appropriate state no matter where the system starts out. However, if the initial algorithm selection does its job properly and selects a high-performance set of algorithms, the controller has a much easier job and the performance of the system is as good or better than just starting out with an arbitrary algorithm set. Figure 8.5 shows this type of behavior. If we purposely choose the worst performing algorithm to start with, the system takes longer to reach a particular solution quality level. But when the selection process chooses a *good* set to begin with, the system reaches the same solution quality level faster.

8.5 Summary

Using our experimental setup, we performed four tests to demonstrate the system's capabilities: initial selection success, non-degradation, user priority and a poor start test. Our prototype system clearly demonstrated its abilities by successfully completing the four experiments; but these tests also showed the potential of the system for future progress.

IX. Conclusions

The focus of this research has been to model the process of managing a distributed system of cooperating processes. We are particularly interested in those domains where finding the optimal solution is either intractable or impossible. In these situations, algorithms typically exist that can produce some form of sub-optimal solution, but, the problem of choosing which one to use for a given situation is often difficult as well. In fact, the No Free Lunch Theorems show that in the general case, this selection cannot be made without some knowledge of the way problem characteristics affect an algorithm's performance.

In Chapter V we developed a method for capturing the essential information about a problem domain and the algorithms to manipulate that domain. The resulting models encapsulate this information in the form of general performance metrics useful in the selection and evaluation of algorithms and algorithm combinations. The encapsulation makes it possible for the controller to be developed independent of any problem domain, yet still utilize the algorithm models to determine algorithm performance for the problem domain being studied. Chapter VI develops a method for selecting the initial algorithm configuration for the system based on the models' performance prediction. During run-time, the algorithm models are used to evaluate the progress of the algorithms and make configuration changes as indicated in an attempt to improve performance as presented in Chapter VII.

The power of this approach is the use of approximate "near-sighted" models of the algorithms. They are typically easy to compute, and may be effective in modeling the algorithm's behavior in the near future. There does not seem to be a need to have extremely accurate models, as the controller monitors and adjusts the system configuration as the algorithm(s) proceed.

To demonstrate the capabilities of this approach, we applied it to uncertain inference over Bayesian Networks. We characterized the problem domain, extracting the significant features from the networks that affect the performance of the algorithms. Appendix B shows the results of analyzing five different inference algorithms and incorporated them into the distributed environment. The resulting system performed as expected: when priority was given to solution quality (instead of resource usage, etc.), the system produced results comparable to the best individual algorithm. With the capability of the system to incorporate other factors into the selection process, we also demonstrated the ability of the controller to reach a reasonable compromise configuration between solution quality and resource consumption.

9.1 Future Work

The impetus for this research came from the frequently asked question: "What tool should I use for this job?" This research has proposed a simple method for making such decisions in the face of several conflicting objectives, with some very promising results. We have just scratched the surface of modeling this decision process; our results, while promising, definitely leave some room for analysis and expansion. As with the human decision making process, we are working with limited information; the more accurate the knowledge we use, the better the results might be. Our use of simplistic algorithm models gives us the capability to make quick decisions; adding more detail (accuracy) to the models would improve the accuracy of controller's decisions. This would improve overall system performance as well as reduce the unnecessary use of system resources.

Currently, our approach uses fixed constants in some of the factors to account for variations in hardware configuration, system speed and so forth. The accuracy of the models could be enhanced by adding the capability to use reinforcement learning to determine more appropriate values for these constants. Additionally, a hardware model could be added to allow the system to better utilize a varied hardware base.

In Chapter VII, we discuss the different levels of control possible for this system, and decided on the highest level for this work. In order for the controller to exert more precise control over the system, it would need to use a lower level of control: at least down to the process interaction level. This would require explicit management of the interaction between processes, as well as some management of the processes themselves. This fine-grained approach to system management has the potential for improving the performance of the system considerably.

As an example of the extensibility of this methodology, it could be applied to other problem domains: likely candidates are planning(8) and other intractable problems. With the models as presented, the only issue involved is selection and modeling of the anytime algorithms appropriate to the domain. The control method itself is unchanged, as it relies only on the abstract algorithm model.

9.2 Contributions

The contributions of this research include the following:

- Development of a structural model for intelligent control of a cooperative distributed process
- Development of a general structural model for cooperative distributed process interaction using anytime and anywhere algorithms
- Instantiation of the general model for uncertain inference using Bayesian Networks
- Analysis and modeling of inference algorithms and a characterization of Bayesian Networks

Appendix A. Barrier Method

This inference algorithm operates by processing the random variables in the Bayesian Network one at a time; effectively eliminating that variable by *subsuming* itself and its children into a *meta-variable.* This consolidates each disconnected component of the network into a single meta-variable; the solution can then be determined by choosing the highest probability state for each meta-variable.

A.1 Definitions

neighbor Random variable A is a neighbor of random variable B if one of the following conditions hold:

- random variable A is a child of random variable B (i.e. it is conditionally dependent on random variable B)
- random variable A is a parent of random variable B (i.e. random variable B is conditionally dependent on random variable A)
- random variable A is a parent of one of random variable B's children

A.2 Algorithm

```
while(unprocessed RVs left)
{
  choose RV to process
  determine neighborhood of RV
  determine minimal set of
    variables and meta-variables that reference RV
  subsume this set into RV
}
```

The algorithm itself is straightforward. The order in which the random variables are processed has no effect on the outcome; it can, however, have a significant effect on performance. It is also possible to predict the performance of this algorithm without actually generating the solution: simply remove the subsumption step in the algorithm and track the size and dimensionality of the neighborhoods for each iteration. The complexity of this step is $O(n \log(n))$.

The neighborhood of the target random variable is determined by starting with the set of neighbors as defined above, then removing any variables that have already been processed.

The set of variables that reference the target random variable consists of any meta-variable that has subsumed the target variable or is dependent on the target variable, as well as any random variable with the target variable in its conditional tail. This set initially contains its children in the original network. As the random variables are processed or subsumed into meta-variables, the outermost enclosing meta-variable containing the child is used instead.

Creating the meta-variables through the process of *subsuming* is what gives this algorithm its unique behavior. *Subsuming* is similar to clustering in that it combines several random variables into a single entity for the purpose of simplifying the Bayesian Network, but in the process of combining the variables together, it also reduces the dimensionality of the problem by eliminating the target random variable from the resulting conditional tables.

```
for(all combination of unprocessed neighbor variable values)
{
  for(all states of RV)
    determine joint conditional probability of RV and all children
  save best value and state into meta-variable
}
```

This seemingly simple process reduces the target random variable's dimensionality to zero – a single state of the random variable is saved in the meta-variable for each combination of the neighbor variables. The probability value stored is the probability of the target random variable combined with the probabilities of its children for the given combination of neighbor variable states.

In terms of the entire network, we have replaced the portion of the joint probability equation involving the target random variable and its children with a new term that does not depend on the target variable. Consider a network with the target random variable A, its parents $\pi(A)$, its children C_1 through C_n and their parents $\pi(C_1)$ through $\pi(C_n)$ (not including A). Processing A would replace this part of the joint probability equation:

$$P(A|\pi(A))P(C_1|A, \pi(C_1)) \dots P(C_n|A, \pi(C_n))$$

with the equivalent part:

$$\begin{aligned} \gamma(A', C_1, \dots, C_n|\pi(A), C_1, \dots, C_n, \pi(C_1), \dots, \pi(C_n)) = \\ P(A'|\pi(A))P(C_1|A', \pi(C_1)) \dots P(C_n|A', \pi(C_n)) \end{aligned}$$

The notation A' is used to indicate that the variable has been processed and is zero-dimensional. This now requires that all the random variables contained in the meta-variable be used as a single unit. The resulting $\gamma()$ term is not a true conditional probability because of the extended set of dependencies, but it reflects the maximum value of the target random variable combined with its children. The underlying conditional probability expressions depend on the neighbors of the target random variable, thus the use of the conditional notation. This term represents the value of the meta-variable containing the random variable A and its children and is used whenever one of these random variables is needed. With the random variable A eliminated, the dimensionality of the meta-variable is the combined dimensionality of the children of A; this is less than the dimensionality of the original set of conditional probability terms.

Extending the above scenario, let D be a random variable which is a parent of C_1 (D has no parents or other children). Processing D would involve subsuming its child C_1; but it has already been subsumed, so we use the γ term in its place;

$$P(D)\gamma(A', C_1, \dots, C_n|\pi(A), C_1, \dots, C_n, \pi(C_1), \dots, \pi(C_n))$$

is replaced by

$$\gamma(A', D', C_1, \ldots, C_n | \pi(A), C_1, \ldots, C_n, \pi(C_1), \ldots, \pi(C_n)) =$$
$$P(D')\gamma(A', C_1, \ldots, C_n | \pi(A), C_1, \ldots, C_n, \pi(C_1), \ldots, \pi(C_n))$$

Notice that the new meta-variable contains not only D and its child C_1 but also A and the rest of its children, but the dimensionality of the new meta-variable is unchanged because the new variable D was eliminated in the subsumption; this results in an overall reduction in dimensionality since the original term for random variable D was subsumed into the new meta-variable.

As the algorithm proceeds through the network, each random variable is processed; its children (or their containing meta-variables) are subsumed and the dimensionality is reduced by one until each disconnected subnet within the Bayesian Network has been subsumed into one meta-variable. With all the variables processed, each meta-variable has a dimensionality of zero; it contains the optimal solution to its part of the network. The overall optimal solution is then determined by assembling the component solutions.

Appendix B. Algorithm Models

This appendix describes the development of the models for the algorithms used in our experiments. The development follows the standard three-phase process: analysis, design and then implementation. For these models, the analysis is performed as described in Chapter V, determining the *probability of improvement* (P_{imp}) and *level of improvement* (E_{imp}). The design phase applies the results of the analysis to determine the six domain-independent factors required for evaluating the progress potential of the algorithms:

1. **Time Complexity $\mathcal{T}$ (static and dynamic)** The amount of time in seconds that this algorithm will take to find the optimal solution. If the algorithm is an approximate algorithm, it is the number of seconds to complete one iteration of the process.
2. **Space Complexity $\mathcal{S}$ (static and dynamic)** The amount of space in machine words required by the algorithm during the entire course of the solution process.
3. **Resource Cost $\mathcal{R}$ (static and dynamic)** An indicator of the cost in system resources to run this algorithm (CPUs, special peripheral equipment, etc.).
4. **Restart Cost $\mathcal{G}$ (dynamic)** The number of seconds spent in starting this algorithm.
5. **Expected Magnitude $\mathcal{C}(x, t, \Delta t)$ (static and dynamic)** The solution quality level the algorithm is expected to produce given a starting level and a time interval. This factor is derived from the E_{imp} analysis.
6. **Probability of Improvement $\mathcal{I}(x, t, \Delta t)$ (dynamic)** The likelihood that the algorithm will produce a solution better than the provided solution quality level during the specified time interval. This factor is derived from the P_{imp} analysis.

The above list also indicates where each factor is used: *static* refers to the initial selection of algorithms and *dynamic* is the run-time control phase. It is in this phase of model development where problem-domain specific information is used; this is where the characteristics of the problem instance can be used to affect the algorithm's performance potential.

The final step is implementing the models. We implemented the basic algorithm model as the C++ virtual base class shown in Figure B.1. Each specific algorithm inherits this base class and overloads the methods to specify that algorithm's behavior.

B.1 Design Considerations

While the model presented is domain-independent by design, it should also be platform independent in implementation. The OVERMIND system encourages the use of networks of possibly dissimilar equipment; the time and space complexity factors should be specified such that a simple constant can be provided to adapt the factor to different hardware platforms. The base class shown provides this capability through the scaling factors available for the first four factors.

Also note that the emphasis in developing the models is approximation; we need models which can be evaluated quickly. The inevitable error from reality is compensated for by the continual monitoring and control activities.

B.2 Genetic Algorithms (GAs)

The GAs used in our experiments were *simple* GAs based on the Genesis(12) framework utilizing both mutation and crossover. We represented a solution (a complete assignment to the underlying Bayesian Network) as an array of integers; each element of the array was converted to a 3-bit binary value for the GA to work with.

B.2.1 Time Complexity. Since the GA is an approximate algorithm, the time complexity is for one iteration; in GA terminology, a *generation*. GAs work on a population of solutions (or *genes*); each member of the population is treated as a unit for the mutation and crossover operations, but the performance evaluation for each member is $O(n)$. This gives us an overall complexity of $O(pn)$. Incorporating our scaling factor, we get our final complexity for one generation as:

$$\mathcal{T} = C_{time}PN \tag{B.1}$$

```
class Algorithm
{
public:
  double timeScale;
  double spaceScale;
  double resourceScale;
  double restartScale;

  Algorithm(void);
  Algorithm(double time, double space, double resource, double restart);

  // info member functions
  virtual char *name(BNStats &stats);
  virtual char *command(BNStats &stats);
  virtual char *args(BNStats &stats);

  // the evaluation methods
  virtual double timeComplexity(BNStats &stats);
  virtual double spaceComplexity(BNStats &stats);
  virtual double magnitudeImprovement(BNStats &stats,
                                      double total_time, double interval,
                                      double start);
  virtual double probabilityImprovement(BNStats &stats, double curr_value,
                                        double total_time, double interval);
  virtual double resourceCost(BNStats &stats);
  virtual double restartCost(BNStats &stats);
  virtual int numberSteps(BNStats &stats, double interval);
};
```

Figure B.1 Base class for the algorithm models

where P is the population size and N is the number of random variables in the network.

B.2.2 Space Complexity. The space complexity of a GA is limited to that required to store the members of the population, and as such is also $O(pn)$, making our space complexity model

$$\mathcal{S} = C_{space}PN \tag{B.2}$$

B.2.3 Resource Cost. We are using the number of CPUs required as our resource cost. For this simple GA, only one CPU is needed:

$$\mathcal{R} = C_{resource} \tag{B.3}$$

B.2.4 Reconfiguration Cost. The start-up activities for the GA consist of loading the Bayesian Network and generating the initial population; since both of these are $O(n)$ processes, the restart cost is

$$\mathcal{G} = C_{restart}N \tag{B.4}$$

B.2.5 Expected Magnitude. The GA improves its solutions by crossover and mutation; both processes are random in nature and currently difficult to model exactly. For the sake of simplicity, we assume that the best member of the population is the one that drives the increase for the population as a whole. While not always true, it does provide a consistent starting point for determining the *expected* improvement for the entire population.

During a crossover operation, a portion of the gene is exchanged with the same portion of another gene; this affects the quality of the solution through the changed assignments of the affected random variables. This change could be an improvement or a degradation for each gene involved; but between the two genes at least one of them should show an improvement. If we assume that at least one of the random variables in the swapped portion improves, the expected magnitude of that improvement would be on the order of P_{max}/P_{avg} where P_{max} is the largest entry in any of the conditional probability tables, and P_{avg} is the average conditional probability value in the previous solution. This

```
function $\mathcal{C}(x, t, t_i)$
    let $s =$ number of generations in interval $t_i$
    let $\mathcal{C} = x.s$

    for($s$ generations)
        $\mathcal{C} = \mathcal{C}\ \frac{P_{max}}{P_{avg}}$

    return $\mathcal{C}$
```

Figure B.2 Algorithm for calculating expected solution value at time $t + t_i$

process repeats every generation, starting with the value of the current solutions, producing the expected result. The expected number of generations s is derived from the *time complexity* factor. The algorithm for this process is in Figure B.2.

B.2.6 Probability of Improvement. We define the probability of improvement as the likelihood that the algorithm will produce a solution with a quality value larger than some reference value (usually the current best quality value) during the time interval specified. The GA will process a number of generations during the interval, and it only takes one of them producing an improvement to cause an increase over the interval; the likelihood of this happening is the converse of the likelihood that none of the generations produce an improvement. If we define the likelihood of one generation not producing any improvement as the likelihood that the the value of the new assignment to a random variable will be less than the old assignment, we end up with

$$\mathcal{I} = 1 - [P(P(r_i|\pi(r_i)) < P(r|\pi(r)))]^s \tag{B.5}$$

where r_i is the new assignment, r is the old assignment and s is the number of generations in the interval.

B.3 Hybrid Stochastic Search (HySS)

The functionality of this algorithm was described earlier. As with the GA, the solution is represented by an integer vector corresponding to a complete assignment to all random variables in the network.

B.3.1 Time Complexity. HySS also operates on a population, and each member of the population is evaluated by an $O(n)$ process; but HySS also uses a table to accumulate its results and prevent duplicate solutions. Adding to this table is an $O(n^2)$ operation. Accounting for our scaling factor gives us

$$\mathcal{T} = C_{time}PN^2 \tag{B.6}$$

where P is the population size and N is the number of random variables in the network.

B.3.2 Space Complexity. HySS requires $O(pn)$ space to store the population, and $O(pn)$ space to store the table:

$$\mathcal{S} = C_{space}P^2N^2 \tag{B.7}$$

B.3.3 Resource Cost. We are using the number of CPUs required as our resource cost. For this algorithm, only one CPU is needed:

$$\mathcal{R} = C_{resource} \tag{B.8}$$

B.3.4 Reconfiguration Cost. The start-up activities for HySS consist of loading the Bayesian Network and generating the initial population; since both of these are $O(n)$ processes, the restart cost is

$$\mathcal{G} = C_{restart}N \tag{B.9}$$

B.3.5 Expected Magnitude. The HySS algorithm is similar to a GA with one major difference: each member of the population is manipulated independent of all others. The member is scanned to determine which random variable provides the largest amount of increase when changed; even though only one random variable is changed, we are guaranteed that it will increase. This difference allows us to use the same method to determine the expected improvement shown in Figure B.2.

B.3.6 Probability of Improvement. With all the members of the population being manipulated independently, the likelihood that at least one will improve is the converse of the likelihood that none of them will improve. Accounting for the number of generations in the specified interval gives us

$$\mathcal{I} = 1 - [[P(P(r_i|\pi(r_i)) < P(r|\pi(r)))]^N]^s \tag{B.10}$$

where r_i is the new assignment, r is the old assignment and s is the number of generations in the interval.

B.4 Heuristic Search Methods

We used two types of heuristic search for our experiments, one using a Shared Cost heuristic(3, 4) and the other using a simple cost-so-far (best first) heuristic. Both types of search use the same framework, so they have the same complexity and cost factors; the only difference is in the performance factors.

B.4.1 Time Complexity. The time requirements for any time of heuristic search is combinatoric: in the worst case, all possible solutions need to be evaluated before finding the optimal answer. Since this is an exact algorithm, this factor reflects the entire solution process:

$$\mathcal{T} = C_{time} d^N \tag{B.11}$$

where d is the average number of states for the random variables in the network.

B.4.2 Space Complexity. The primary usage of space in the search process is in maintaining the open queue – those partial solutions which have been generated but not processed. Our framework expands partial solutions one random variable at a time, so (worst case) it is possible for the open queue to contain most of the possible solutions. The expansion process puts d new partial solutions on the queue for each one it removes; this sets the space factor to

$$\mathcal{S} = C_{space}(d-1)^N \tag{B.12}$$

B.4.3 Resource Cost. We are using the number of CPUs required as our resource cost. For this search framework, only one CPU is needed:

$$\mathcal{R} = C_{resource} \tag{B.13}$$

B.4.4 Reconfigure Cost. The start-up cost for the search is loading the Bayesian Network and performing any preprocessing for the heuristic. For these heuristics, the worst case is $O(n)$:

$$\mathcal{G} = C_{restart}N \tag{B.14}$$

B.4.5 A Shared Cost.*

B.4.5.1 Expected Magnitude. The shared-cost heuristic condenses the values of the unassigned random variables into a small number of table entries; this has the effect of smoothing out many of the features in the conditional probability tables. This creates a breadth-first type effect in the expansion pattern. Since our 'improvement' comes from replacing an arbitrary assignment with our own assignment, we can assume an improvement of P_{max}/P_{avg} each time a new 'level' is reached (a new variable is replaced) in the expansion, all we have to do is estimate the rate at which new levels are reached.

With a breadth-first expansion, new levels are only reached when most (if not all) partial solutions at the previous level have been processed; the number of levels reached can be determined by taking the logarithm of the number of nodes expanded. The base of the logarithm (d) is the average number of states for the random variables. The expected value at the end of the interval is then

$$\mathcal{C} = (P_{max}/P_{min})^{\log_d(s)} \tag{B.15}$$

where s is the number of iterations in the time interval.

B.4.5.2 Probability of Improvement. Within the specified interval, we could have the opportunity to traverse multiple levels. In order to have an overall increase in solution quality at the end of the interval, all traversals need to increase[1]:

$$\mathcal{I} = P(P_{max} > P_{avg})^{\log_d(s)} \tag{B.16}$$

B.4.6 Best First.

B.4.6.1 Expected Magnitude. For the best first heuristic, we also assume an increase of P_{max}/P_{avg} each time a new level is reached; but this heuristic is much more sensitive to the features in the conditional probability tables. The determination of how often to advance to a new level is related to the extremeness(27) of the distribution in the conditional probability tables:

Case 1 if $(P_{max})^N < (P_{min})^{N-1}$, the search progresses in a breadth-first fashion since the best entries at each level are not good enough to beat anything else at its level to progress more rapidly: $l = \log_d(s)$.

Case 2 if the network is extreme, then the search is depth first, but it is guaranteed that the optimal solution is at the bottom of the first descent. In this case, each expansion descends another level: $l = s$.

Case 3 in all other cases, we scale the base of the logarithm in order to achieve a middle ground between the breadth- and depth-first descent rates:

$$scale = \frac{P_{max}^2 - P_{avg}}{\sigma}$$

$$l = \log_{d \times scale}(s)$$

Our final expected magnitude is therefore

$$\mathcal{C} = (P_{max}/P_{min})^l \tag{B.17}$$

[1]this is not necessarily true, but without knowledge of the relative magnitudes of the increases and decreases, there is no way of determining the likelihood of an overall increase

B.4.6.2 Probability of Improvement. Using the same arguments as for the shared-cost heuristic, we only achieve an overall increase if we get an increase at all levels:

$$\mathcal{I} = P(P_{max} > P_{min})^l \tag{B.18}$$

B.5 Barrier Computation

We included the Barrier Computation algorithm in our experiments even though it is not an anytime algorithm for two main reasons. First, it is an interesting algorithm with performance characteristics that are clearly defined. We also wanted to demonstrate the flexibility of the control model in being able to accommodate both anytime and non-anytime algorithms in the same collection.

B.5.1 Time Complexity. The time complexity of this algorithm depends mainly on the maximum size of the *neighborhood* of the random variables in the Bayesian Network. The neighborhood of a random variable is determined by finding all the other variables connected to it after the network has been triangulated(26). Each iteration of the algorithm processes the neighborhood of one random variable; this action modifies the neighborhood size of the neighboring random variables, typically resulting in a neighborhood size of $d^{h/2}$ where h is the largest neighborhood size prior to the start of the algorithm. This results in a time complexity of

$$\mathcal{T} = C_{time} N d^{h/2} \tag{B.19}$$

B.5.2 Space Complexity. The space complexity is driven by the neighborhood tables that must be stored, and hence is the same as the time complexity:

$$\mathcal{S} = C_{space} N d^{h/2} \tag{B.20}$$

B.5.3 Resource Cost. We are using the number of CPUs required as our resource cost. For this algorithm, only one CPU is needed:

$$\mathcal{R} = C_{resource} \tag{B.21}$$

B.5.4 Reconfiguration Cost. The start-up activities for this algorithm consist of loading the Bayesian Network and chording the graph to determine the initial neighborhoods. Loading the network is $O(n)$, but chording the network is $O(np^2)$ where p is the maximum number of parents for all random variables in the network. This makes our restart factor:

$$\mathcal{G} = C_{reconfig} N p^2 \tag{B.22}$$

B.5.5 Expected Magnitude. Calculating the expected magnitude for this algorithm poses a challenge because it only produces one answer – the optimal solution! Looking at the usage of this factor, however, we can still provide a usable result even if we cannot estimate the value of the optimal solution.

First, if the estimated solution time (t_s) is within the specified interval, we simply return 1.0: this guarantees that the algorithm will be selected for use.

On the other hand, if we know that if the interval provided does not contain the time we estimated for completing the solution, we could simply return 0.0. This, however will most likely prevent this algorithm from being selected for anything but extremely small problems. What we need to do is provide a value that makes the algorithm competitive for medium size problems without making it appealing for problems that are too large for it to solve in reasonable time. One way to accomplish this would be to simply return the value at the start of the interval – this would not indicate exceptional performance, but would not immediately disqualify the algorithm:

$$\mathcal{C}(\mathbf{x}, t, \Delta t) = \begin{cases} 1.0 & \text{if } (t + \Delta t) > t_s \\ \text{s} & \text{otherwise} \end{cases} \tag{B.23}$$

B.5.6 Probability of Improvement. For this algorithm (and any non-anytime algorithm), the only time we can expect to see an improvement is when the algorithm finishes. This is not too useful for selection purposes, as it almost guarantees that the algorithm will not be selected unless the solution time falls within the initial interval. In order to alleviate this problem, we can distribute the probability of finding a solution over the entire estimated solution time t_s instead of leaving it as a spike at the end of the interval. This allows us to use the probability of improvement as an indication of how close to the final solution we are:

$$\mathcal{I}(\mathbf{x}, t, \Delta t) = \begin{cases} (t + \Delta t)/t_s & \text{if } (t + \Delta t) < t_s \\ 1.0 & \text{otherwise} \end{cases} \tag{B.24}$$

Bibliography

1. Antsaklis, Panos J. and Kevin M. Passino. "Introduction to Intelligent Control Systems with Hight Degrees of Autonomy." *An Introduction to Intelligent and Autonomous Control* chapter 1, 1–26, Kluwer, 1993.

2. Boddy, Mark. "Anytime Problem Solving Using Dynamic Programming." *Proceedings of the National Conference on Artificial Intelligence*. 738–743. Menlo Park, CA: AAAI Press, 1991.

3. Charniak, Eugene and Saadia Husain. "A New Admissible Heuristic for Minimal-Cost Proofs." *Proceedings of the National Conference on Artificial Intelligence*. 446–451. Menlo Park, CA: AAAI Press, 1991.

4. Charniak, Eugene and Solomon E. Shimony. "Cost-Based Abduction and MAP Explanation," *Artificial Intelligence*, *66*:345–374 (1994).

5. Cooper, Gregory F. *Probabilistic Inference Using Belief Networks is NP-hard.* Technical Report KSL-87-27, Medical Computer Science Group, Stanford University, 1987.

6. Dagum, Paul and Michael Luby. "Approximating Probabilistic Inference in Bayesian Belief Networks is NP-hard," *Artificial Intelligence*, *60 (1)*:141–153 (1993).

7. Dean, Thomas and Mark Boddy. "An Analysis of Time-Dependent Planning." *Proceedings of the National Conference on Artificial Intelligence*. 49–54. Menlo Park, CA: AAAI Press, 1988.

8. Dean, Thomas and Michael Wellman. *Planning and Control.* Morgan Kaufmann, 1991.

9. Garvey, Alan and Victor Lesser. "Design-to-time Scheduling and Anytime Algorithms," *SIGART Bulletin*, *7*(2):16–19 (1996).

10. Geist, Al, et al. *PVM 3 User's Guide and Reference Manual.* Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, 1994.

11. Grass, Joshua and Shlomo Zilberstein. "Anytime Algorithm Development Tools," *SIGART Bulletin*, *7*(2):20–27 (1996).

12. Grefenstette, John J., "A User's Guide to GENESIS." Navy Center for Applied Research in Artificial Intelligence, Naval Research Laboratory, Washington, DC, 1987.

13. Hansen, Eric A. and Shlomo Zilberstein. "Monitoring Anytime Algorithms," *SIGART Bulletin*, *7*(2):28–33 (1996).

14. Hayes-Roth, Barbara. "A Blackboard Architecture for Control," *Artificial Intelligence*, *26*:251–321 (1985).

15. Hayes-Roth, Barbara. "Intelligent Control," *Artificial Intelligence*, *59*:213–220 (1993).

16. Hayes-Roth, Barbara. "Opportunistic Control of Action in Intelligent Agents," *IEEE Transations on Systems, Man and Cybernetics*, *23*(6):1575–1587 (November 1993).

17. Hayes-Roth, Barbara. "An architecture for adaptive intelligent systems," *Artificial Intelligence*, *72*:329–365 (1995).

18. Heckerman, D.E. "An empirical comparison of three inference methods." *Proceedings of the Fourth Workshop on Uncertainty in Artificial Intelligence,* Minneapolis, MN. 158–169. August 1988.

19. Hogg, Tad and Bernardo A. Huberman. "Better Than the Best: The Power of Cooperation." *1992 Lectures in Complex Systems V*. SFI Studies in the Sciences of Complexity, edited by Lynn Nadel and Daniel Stein, 165–184, Reading, MA: Addison-Wesley, 1993.

20. Hogg, Tad and Colin P. Williams. "Solving the Really hard Problems with Cooperative Search." *Proceedings of the Eleventh National Conference on Artificial Intelligence.* 231–236. Menlo Park, CA: AAAI Press, 1993.

21. Huberman, Bernardo A., et al. "An Economics Approach to Hard Computational Problems," *Science*, *275*:51–54 (January 3, 1997).

22. Jitnah, N. and A.E.Nicholson. *Belief Network Inference Algorithms: a Study of Performance Based on Domain Characterisation.* Technical Report TR-96-249, Clayton, VIC, 3168 Australia: Monash University, 1996.

23. Kornfeld, Ari. "Causal diagrams: Clarifying Uncertainty," *AI Expert* (November 1991).

24. Michalewicz, Zbigniew. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag, 1992.

25. Morjaia, Mahesh A., et al. "Commercialization of EPRI's Generator Expert Monitoring System (GEMS)." *Expert System Application for the Electric Power Industry.* Phoenix: EPRI, 1993. Also: GE techreport GER-3790.

26. Pearl, Judea. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, CA: Morgan Kaufmann, 1988.

27. Poole, D. "Average-case analysis of a search algorithm for estimating prior and posterior probabilities in Bayesian networks with extreme probabilities." *Proc. 13th International Joint Conf. on Artificial Intelligence*. 606–612. August 1993.

28. Rojas-Guzman, Carlos and Mark A. Kramer. "GALGO: A Genetic ALGOrithm Decision Support Tool for Complex Uncertain Systems Modeled with Bayesian Belief Networks." *Proceedings of the Conference on Uncertainty in Artificial Intelligence.* 368–375. San Francisco, CA: Morgan Kaufmann Publishers, 1993.

29. Santos, Jr., Eugene. "A Fast Hill-Climbing Approach Without An Energy Function for Finding MPE." *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*. 170–179. 1993.

30. Santos, Jr., Eugene, et al. *On a Distributed Anytime Architecture for Probabilistic Reasoning*. Technical Report AFIT/EN/TR95-02, Department of Electrical and Computer Engineering, Air Force Institute of Technology, 1995.

31. Shimony, Solomon E. "Finding MAPs for Belief Networks is NP-hard," *Artificial Intelligence*, *68*:399–410 (1994).

32. Shimony, Solomon Eyal and Eugene Santos, Jr. "Exploiting Case-Based Independence for Approximating Marginal Probabilities," *International Journal of Approximate Reasoning*, *14*(1):25–54 (1996).

33. Vincent, Thomas L. and Grantham Walter J. *Optimality in parametric systems*. Wiley, 1981.

34. Vose, Michael D. "Modeling Simple Genetic Algorithms." *Foundations of Genetic Algorithms 2* edited by L. Darrell Whitley, Morgan Kaufmann, 1993.

35. Welch, Robert L. "Real Time Estimation of Bayesian Networks." *Proceedings of the Conference on Uncertainty in Artificial Intelligence*. 533–544. San Francisco, CA: Morgan Kaufmann Publishers, 1996.

36. Whitley, Darrell. "An Executable Model of a Simple Genetic Algorithm." *Foundations of Genetic Algorithms 2* edited by L. Darrell Whitley, Morgan Kaufmann, 1993.

37. Williams, Edward, et al. "Experiments with Distributed Anytime Inferencing: Working with Cooperative Algorithms." *Proceedings of the AAAI-97 Workshop on Building Resource-Bounded Reasoning Systems*. Menlo Park, CA: AAAI Press, August 1997.

38. Wolpert, David H. and William G. Macready. "No Free Lunch Theorems for Optimization," *IEEE Transactions on Evolutionary Computation*, *1*(1):67–82 (Aoruk 1997).

39. Zhang, Weixiong and Richard E. Korf. "Performance of linear-space search algorithms," *Artificial Intelligence*, *79*(2):241–292 (1996).

40. Zhang, Weixiong and Richard E. Korf. "A study of complexity transitions on the asymmetric traveling salesman problem," *Artificial Intelligence*, *81*(1-2):223–239 (1996).

41. Zilberstein, Schlomo. "Composition and Monitoring of Anytime Algorithms." *Proceedings of the IJCAI-95 Workshop on Anytime Algorithms and Deliberation Scheduling*. 1995.

42. Zilberstein, Shlomo and Stuart Russell. "Optimal composition of real-time systems," *Artificial Intelligence*, *82*(1-2):181–213 (1996).

Vita

Edward Williams [redacted] graduated from the University of Washington with a Bachelor of Science in Electrical Engineering in 1984. He was commissioned into the Air Force in December, 1984; his first tour of duty was in Denver, Colorado, before earning his Master of Science in Computer Engineering from the Air Force Institute of Technology in December, 1990; his primary area of research was in high performance computer architectures. His next duty was as a deputy program manager for the Satellite Control Network System Program Office in Los Angeles, California. He began his doctoral research at the Air Force Institute of Technology in 1994; his next assignment is to the Information Warfare Battle Lab in San Antonio, Texas.

[redacted]

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1997	3. REPORT TYPE AND DATES COVERED interim

4. TITLE AND SUBTITLE MODELING INTELLIGENT CONTROL OF DISTRIBUTED COOPERATIVE INFERENCING	5. FUNDING NUMBERS
6. AUTHOR(S) Edward M. Williams	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Dr. Abe Waksman AFOSR/NM 110 Duncan Ave Bolling AFB DC 20332-8080	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The ability to harness different problem-solving methods together into a cooperative system has the potential for significantly improving the performance of systems for solving NP-hard problems. The need exists for an intelligent controller that is able to effectively combine radically different problem-solving techniques with *anytime* and *anywhere* properties into a distributed, cooperative environment. This controller requires models of the component algorithms in conjunction with feedback from those algorithms during run-time to manage a dynamic combination of tasks effectively. This research develops a domain-independent method for creating these models as well as a model for the controller itself. These models provide the means for the controller to select the most appropriate algorithms, both initially and during run-time. We utilize the algorithm performance knowledge contained in the algorithm models to aid in the selection process. This methodology is applicable to many NP-hard problems; applicability is only limited by the availability of anytime and anywhere algorithms for that domain. We demonstrate the capabilities of this methodology by applying it to a known NP-hard problem: uncertain inference over Bayesian Networks. Experiments using a collection of randomly generated networks and some common inference algorithms showed very promising results. Future directions for this research could involve the analysis of the impact of the accuracy of the algorithm models on the performance of the controller; the issue is whether the increased model accuracy would cause excessive system overhead, counteracting the potential increase in performance due to better algorithm selection.

14. SUBJECT TERMS probabilistic networks, uncertainty, control, artificial intelligence	15. NUMBER OF PAGES 105
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL