12-1996

Inference Algorithm Performance and Selection under Constrained Resources

Brett J. Borghetti

/AFIT/GCS/ENG/96D-05

INFERENCE ALGORITHM PERFORMANCE AND SELECTION UNDER CONSTRAINED RESOURCES

THESIS

Brett Jason Borghetti
Captain, USAF

/AFIT/GCS/ENG/96D-05

Approved for public release; distribution unlimited

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U. S. Government.

/AFIT/GCS/ENG/96D-05

INFERENCE ALGORITHM PERFORMANCE AND SELECTION UNDER CONSTRAINED RESOURCES

THESIS

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

Master of Science

Brett Jason Borghetti, B.S.EE

Captain, USAF

December, 1996

Acknowledgements

I would like to thank the members of my committee: Doctor Eugene Santos, Jr., Major Sheila Banks and Doctor Henry Potoczny. You have helped make this work possible with your time and support. Also, thanks to Major Edward Williams who spent many hours teaching me the finer points of software hackeneering with C++ and the OVERMIND system.

To Scott, Bob, Louise and Dan: Thanks for putting so much time into the PESKI project. And thanks for all of your support, ideas and philosophical discussions (arguments?) about how to make a good software system... You guys have been good friends throughout this experience. Thanks to OracleJoel, the man with all of the answers. You have helped me in more ways than I know. I wish you all the best of luck in your careers.

To the Walleyball crew: James "Worf" Benslay, Todd "Spike" Kellet, Joe "Perot" Moritz et al., You guys have helped me take out some stress on the infamous "Court Number Three". For that, I thank thee.

To the GCS-96D class, and all those outliers who never made it into our email list: Thanks for making the AFIT experience a good one. Live long and produce software: Your country is counting on you...

And finally, thanks to Saralinda for providing me with constant support, encouragement and motivation I needed to get here.

This work was supported in part by AFOSR Grant #94-0006.

Brett Jason Borghetti

Table of Contents

List of Figures

Abstract

Knowing that reasoning over probabilistic networks is, in general, NP-hard, and that most reasoning environments have limited resources, we need to select algorithms that can solve a given problem as fast as possible. This thesis presents a method for predicting the relative performance of reasoning algorithms based on the domain characteristics of the target knowledge structure. Armed with this knowledge, the research shows how to choose the best algorithm to solve the problem. The effects of incompleteness of the knowledge base at the time of inference is explored, and requirements for reasoning over incompleteness are defined. Two algorithms for reasoning over incomplete knowledge are developed: a genetic algorithm and a best first search. Empirical results indicate that it is possible to predict, based on domain characteristics, which of these algorithms will have better performance on a given problem.

INFERENCE ALGORITHM PERFORMANCE AND SELECTION UNDER CONSTRAINED RESOURCES

I. Introduction

The Air Force is automating many information intensive processes. Often, the Air Force uses computers to assist or even replace personnel in extracting knowledge from large volumes of data and process that knowledge. Artificial Intelligence (AI) experts have been struggling with the problem of processing knowledge (information) for years. While AI methods are usually more efficient than non-AI (brute force) methods, they still push the limitations of computing equipment.

While many advances in AI have reduced the time spent in knowledge processing, many problems are still beyond the capabilities of today's computer hardware. One reason why the advances in AI have failed to provide extensive improvements is inherent in the nature of their techniques. Early knowledge manipulation routines utilized a single algorithm on a single computer to perform a search for the optimal solution to a problem. Experts in AI used many techniques to optimize the algorithm. Although these optimized algorithms worked well in certain cases, they were seldom able to perform well in all cases. In the last decade, however, researchers have developed a new tactic for knowledge processing [2, 7, 8, 9, 11, 12]. Several different algorithms running concurrently are used to arrive at a solution faster than any single algorithm working alone. Each different algorithm has its strengths and weaknesses in the specific case, yet when combined, the strengths of certain algorithms compensate for the weaknesses of others. When acting in concert, multiple algorithms are better suited for solving a range of problems.

The algorithms working in a group combine their strengths by generating and sharing partial solutions while they search for the final solution. When one algorithm makes a breakthrough, it shares its partial findings with the other algorithms. Thus, all algorithms advance towards a solution when any one advances.

Unfortunately, the more algorithms used in a working group, the more computational power needed. Each algorithm added to the working group also requires more communication overhead. Since most people trying to process knowledge have limited resources at their disposal, a decision must be made: Given that there are limited resources, which algorithms should be executed to achieve the best performance?

Also, since the algorithms are providing partial solutions during their execution, it may be possible to transform the part of the problem that remains to a different problem that can be solved easier with one of the other algorithms in the working group. The decision can then be rephrased: Given that there are limited resources, and the solution has reached a certain maturity, which algorithms should be executed to achieve the best performance from here? In other words, given what we know right now about the problem, what is the best strategy to solve the remainder of it.

In order to make this decision, it is necessary to predict the capability of a given strategy[1] (group of algorithms working together) to converge on a solution under the current circumstances. Prediction of the performance of a strategy relies on prediction of individual performance and prediction of interactive performance. This thesis focuses on using prediction of individual algorithm performance to develop a strategy for problem solving.

Since the performance of an algorithm varies from problem to problem, this thesis proposes that in each problem instance, the input data manipulated (the domain) contains characteristics that affect the performance of the algorithm, and if these features of the data can be isolated, they can be used to predict the performance of algorithms on that information.

The remainder of this work will expand on the concepts of feature based performance prediction and selection of algorithm strategies from this prediction. Chapter II provides the reader with the background for the topic in the domain that AFIT is working with, and an explanation of the class of algorithms that can share partial solutions. Chapter III explains why reasoning is hard, presents the challenges of working with the domain, and

[1] A strategy may contain one or more algorithms

discusses the specific algorithms used at AFIT. A method for selecting algorithms based on the characteristics of the domain is developed in Chapter IV. Chapter V presents the experimental results of this technique, and Chapter VI concludes the thesis and inspires future research in the area.

II. Background

Traditionally, many organizations generated point solutions to solve only the task they were working. In the interest of reuse, cost savings, and commonality, The Air Force Institute of Technology (AFIT) is developing an environment for acquiring, maintaining, and reasoning over a user-defined knowledge domain. This chapter presents a backround of the system, the structure it uses to store knowledge, and the class of algorithms it employs for reasoning.

2.1 PESKI

AFIT is developing the Probabilistic, Expert System Knowledge and Inference (PESKI) environment, a domain-independent architecture for building expert systems [19, 20, 21]. Figure 2.1 shows an overview of PESKI. The system is designed to store domain knowledge in the form of a probabilistic network model. It provides tools for knowledge acquisition and validation to improve the quality of the representation. PESKI also provides tools for reasoning over the stored probabilistic knowledge. The underlying structure that makes PESKI a unique expert system development environment is its knowledge representation: The Bayesian Knowledge Base (BKB).

2.2 Baysian Knowledge Base

Knowledge is continually in a state of change. Although we attempt to store knowledge so that we may inference over it, at best, stored knowledge is an out-of-focus snapshot of our universe. In most research fields, for example, researchers are continually adding knowledge to a domain so fast that no single researcher could ever be aware of all the advances in that field at any given time. By the time that new knowledge is added to a knowledge base, it may be obsolete. Thus, it is extremely important to have a knowledge base that is flexible enough to handle both unknowns (incomplete knowledge) and changes in the domain over time.

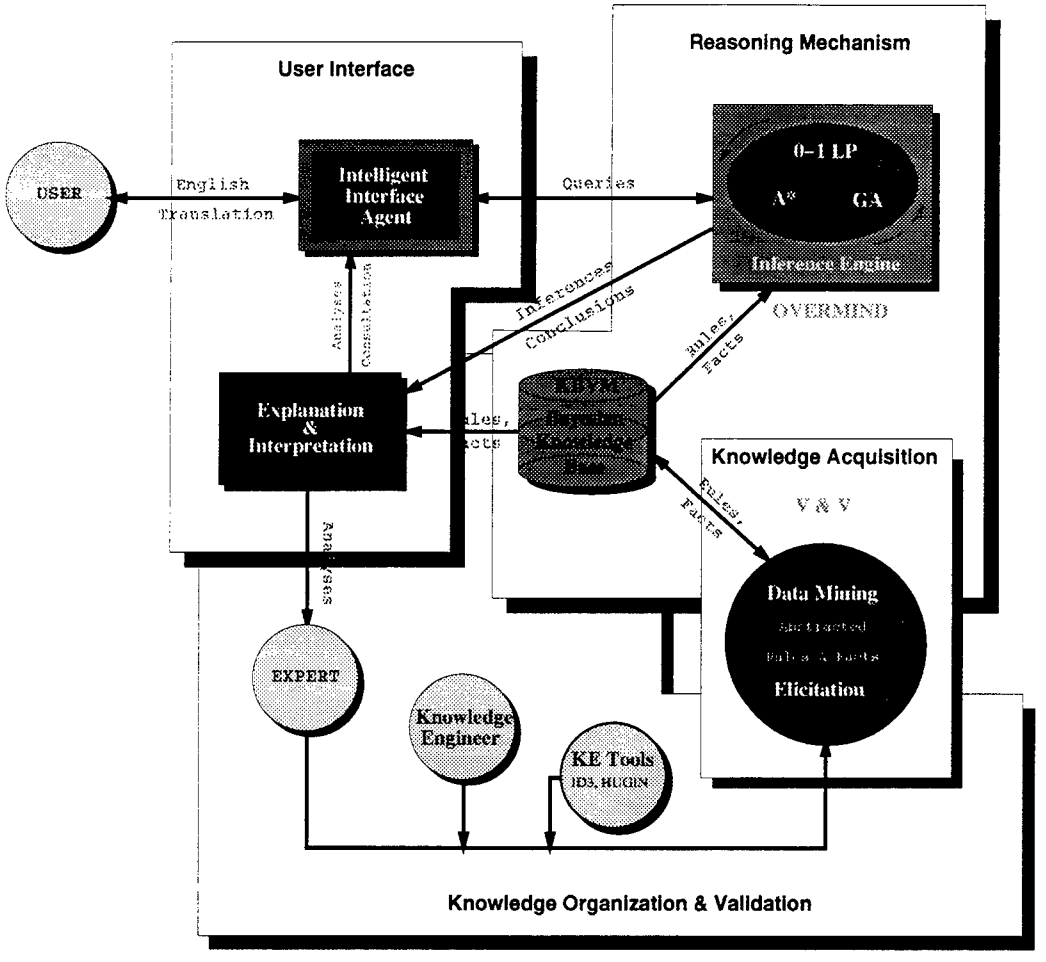

Figure 2.1 PESKI
PESKI is composed of a User Interface, a Knowledge Acquisition tool, a set of tools for Knowledge Organization and Validation, and a Reasoning Mechanism.

In the AFIT PESKI system, this domain knowledge is probabilistic information stored in a Bayesian Knowledge Base [1] format[1]. A BKB is similar to a Bayesian Network (BN) [16]; it is a directed graph capable of representing uncertainty in knowledge through a network of random variables (RV) and their probabilistic relationships. In fact, any Bayesian Network can be represented as a Bayesian Knowledge Base. An example of a Bayesian Network and the equivalent BKB is shown in figure 2.2.

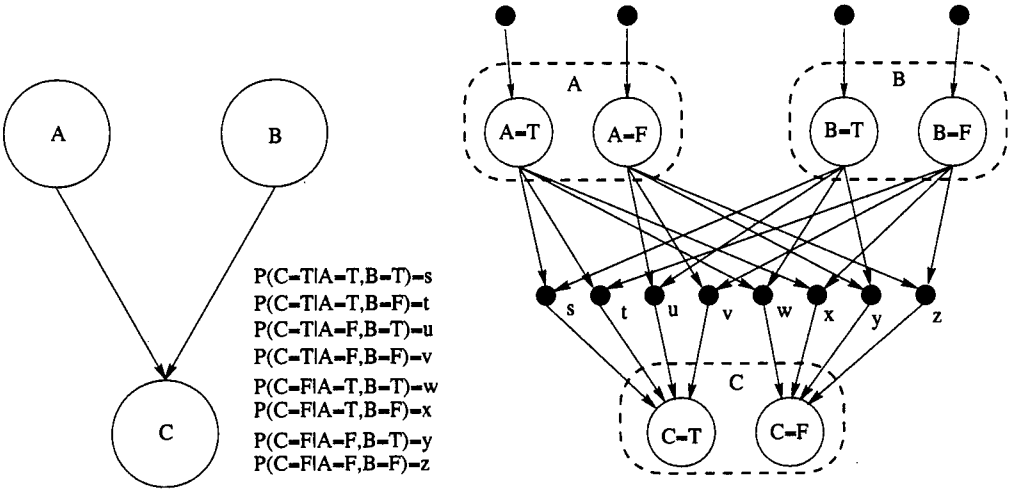

Figure 2.2 Bayesian Network and equivalent Bayesian Knowledge Base

In a probabilistic network, each random variable (or component) can take on one of a finite number of instantiations (or states). The relationships between the components (and the states of the components) are presented as conditional probabilities. Notice that each random variable in the BN is equivalent to the composite states (dotted ovals) in the BKB[2]. The dark nodes, called supports, are the "rules" that capture the relationships (conditional probabilities) between the tail condition (parents of the support) and the supported state (child of the support). Thus, each of the supports in the BKB explicitly represent one of the conditional probabilities in the conditional probability table (CPT) of the BN.

[1]The BKB format was referred to as a Bayesian Forest by Darwin Banks [1]

[2]The RVs have been shown on the BKB as a dotted oval, however in the actual BKB there is no explicit representation of components.

In a BKB, a support node with no parents is called a root support. A root node (or root state) is a state with exactly one parent, a root support[3]. We define the instantiations with no children as leaf nodes. Thus, in Figure 2.2, A=T, A=F, B=T, B=F are all root nodes and C=T, C=F are the leaf nodes.

The support nodes are key in the flexibility of the Bayesian Knowledge Base. They enable the Bayesian Knowledge Base to represent incomplete knowledge (unknowns) easier than a Bayesian Network. In a Bayesian Network, each random variable must have a fully specified CPT containing all of the possible combinations of one state from its RV and one state from each of its parent RVs, even if these relationships are unknown in the domain being modeled. Thus, a Bayesian Network is required to either completely represent RV relationships who's actual relationships are not completely known, or leave that RV out of the model. Either choice may compromise the quality of the model.

Bayesian Knowledge Bases were designed to eliminate the compromises in quality caused when a domain expert makes assumptions about unknown relationships between RVs. When the expert is missing domain information about the relationships between variables, the expert does not have to specify those relationships (arcs and supports) in the BKB, yet the expert can still express the RVs in terms of their component states without explicitly describing the relationships. When one or more of the relationships are missing, we call the BKB incomplete. Now we develop language to express subsets of BKBs so that incompleteness can be formally defined. First, we define a *collection* of states.

Definition 1. *A* collection *of states is a set of states, no more than one from each random variable in the BKB. A collection many contain fewer than one state for every RV in the BKB.*

Definition 2. *A* full *collection is a collection that has exactly one state specified for every RV in the BKB.*

Then, completeness is formally defined.

Definition 3. *A* complete *BKB is a BKB where for all arbitrary collections, every state in the collection is either a root state or a child of at least one other state in the collection.*

[3]The term "root node" may refer to either the instantiation nodes or their parent support node.

Incompleteness is the opposite of completeness.

Definition 4. *An* incomplete *BKB is a BKB where there exists at least one collection such that one or more of the states is neither a root node, nor a child of any other states in the collection.*

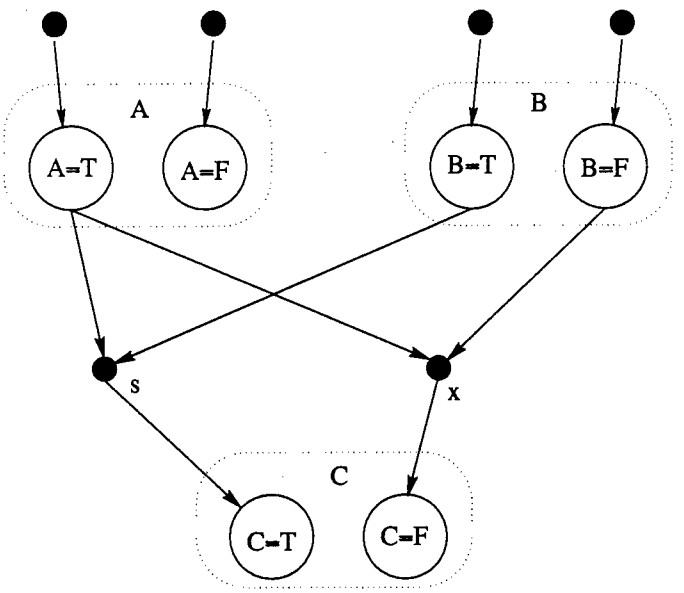

Figure 2.3 Incomplete Bayesian Knowledge Base

Figure 2.3 shows an example of an incomplete BKB. An arbitrary collection of states A=F, B=T, and C=T, cannot meet the completeness condition because C=T is neither a root node nor a child of A=F or B=T. Therefore, the BKB is incomplete. Interestingly, because a BKB does not necessarily have to be complete, the transformation of a BKB into a Bayesian Network may not always be possible.

For the remainder of this document, the words *complete* and *incomplete* refer to the definitions of completeness of the BKB or a collection, while the words *full* and *partial* refer to the cardinality of a collection. Thus, in Figure 2.3 A=F, B=T, C=T is a full incomplete collection; A=T, B=T, C=T is a full complete collection; A=F, B=T is a partial complete collection; and A=F, C=F is a partial incomplete collection.

2.3 Anytime Algorithms

The reason we develop a knowledge representation as a model of a domain is to facilitate reasoning about that domain. Reasoning is the method by which we gather new or non-explicit information from the stored knowledge. Flexibility in reasoning algorithms is important because reasoning is a time intensive process. This section discusses the differences in flexibility between traditional sequential algorithms and the *anytime* algorithms used by PESKI for reasoning.

In a traditional algorithm, input is provided, the algorithm computes for some time, and an output is generated. In an algorithm such as the Best First Search (BFS) algorithm, there is no meaningful solution until the final node is expanded. Once the final node is expanded, the solution is generated and takes on meaning. If time expires before the BFS has completed its search, there is no solution. With computational resources (especially cpu time) restricted in some systems, the need arose for algorithms that could generate *partial* solutions in the time they were alloted [7, 8, 9, 11, 12]. Boddy and Dean labled algorithms that produce partial solutions of non-decreasing quality over time *anytime algorithms* [2].

The trait of producing interim solutions over the course of execution makes anytime algorithms extremely flexible. For example, if the computer was turned off just before a traditional algorithm completed, the algorithm would provide no useful information. An anytime algorithm, however, could have possibly produced many partial solutions before the computer was shut down. A user could interrupt the algorithm when a solution is desired, and the algorithm would produce at least a partial solution. Some algorithms can even use the partial solutions as starting points for further calculations. These flexibilities enhance the reasoning capabilities of PESKI. In the next chapter, we shall see why it is vitally important to have flexibility when inferencing over probabilistic knowledge.

III. Reasoning Algorithms

In this chapter, we discuss several of the algorithms used for reasoning and the adaptations required to make them valid for reasoning over Bayesian Knowledge Bases. Before we begin this discussion it is necessary to explain why reasoning in general is hard and stipulate the specific challenges of reasoning with BKBs

3.1 Why is reasoning hard?

One of the most perplexing problems in Artificial Intelligence is probabilistic reasoning [16]. There are many methods for reasoning over probabilistic domains, but most methods use some form of approximation technique or exact search. Inference over probabilistic knowledge in Bayesian Networks (BN) was proven to be NP-hard by Cooper [4, 5], Dagum, Luby [6] and Shimony [22]. As we shall see, reasoning with BKBs is even more computationally expensive than reasoning with Bayesian Networks.

3.1.1 Belief revision for BKBs.

Although Bayesian Knowledge Bases provide a means of representing incomplete knowledge that Bayesian Networks cannot, the capability has its cost. As we shall see, reasoning is hindered by the existence of the incompleteness, the very quality that makes the BKB representation so flexible.

This thesis focuses on a subclass of reasoning, belief revision. In belief revision, our goal is to seek the most likely "state of the world" given evidence about the world. In a Bayesian Knowledge Base, a "state of the world" is the set of one instance (state) of each random variable in the network. Essentially, a state of the world is a full collection of states. The probability of this full collection occurring can be computed by taking the joint probability (the product of all applicable conditional probabilities)(JP) of the random variables. Thus, to find the most likely state of the world, the inference engine seeks the maximal joint probability among all possible states of the world.

The joint probability of a given state of the world is calculated by multiplying the conditional probabilities that represent the relationships between the random variables in the chosen state of the world. In a BKB, the conditional probabilities are represented by support nodes. To calculate the joint probability, take the product of the active support nodes in a given state of the world. A support node is *active* when it's tail condition is met, i.e. its only parents are a subset of the instances expressed in the given state of the world. For example, in Figure 2.2 support node s is active when A=T, B=T and C=T. Inference in a BKB involves selecting the state of the world such that the product of the values of all active support nodes (the joint probability) is maximized.

Unfortunately, in a BKB, an incomplete collection of states has no calculable joint probability because there is at least one state that is neither a root node or a child of any other state in the collection (thus no active support nodes can exist for this state). Since the JP for an incomplete solution cannot be computed, each algorithm PESKI employs for reasoning must have a method to cope with incompleteness internally.

Reasoning over the BKB is very similar to reasoning over the Bayesian Network, with one exception: handling incompleteness. We allow incompleteness in the representation to give the knowledge acquisition process flexibility. When we reason over the BKB, however, we generally want to find a complete solution because a complete solution is the only type of solution that has a calculable joint probability[1]. In a Bayesian Network, we are always guaranteed that given any states of the parent random variables we have a way of activating a specific state of the child random variable, i.e. we have completeness. In contrast, a Baysian Knowledge Base that has incompleteness may not allow a given set of instances of the parent random variable to actuate a specific instance of the child because one or more of the states are neither root nodes, nor children of any other states in the collection (recall Definition 4). This means that during inferencing we must verify that the state of the world we are examining does not contain any incompleteness. The check for of incompleteness is inexpensive if we are only looking for the *existence* of it, but finding the *level* of incompleteness in a given state of the world (finding the number of

[1]Complete solutions are also the only solutions explicit from the domain being modeled.

occurrences of incompleteness in the collection) is sometimes necessary and this process is computationally expensive.

The ability of BKBs to represent incompleteness forces checking for incompleteness in all of our BKB inference algorithms. Furthermore, techniques are required to minimize the impact of the computational complexity of the completeness checks.

3.2 Algorithms for Belief Revision

PESKI's concept for an inference engine is to utilize the strengths of several different algorithms in a joint effort to find a solution to an inference scenario. Every stock algorithm used for this purpose must be adapted to handle the unique structure of the BKB as well as the potential for incompleteness in the BKB. In the following sections, we discuss the particular adaptations of Genetic Algorithms (GA) and Best First Search (BFS) used for reasoning with incomplete knowledge in this research. Before we begin this discussion, however, it is necessary to build a foundation of criteria that each algorithm must meet.

Definition 5. *A solution is a collection created at inference time. Solutions are expressed as a vector V of width N, where N is the number of random variables in the BKB. Position i in the vector contains the state of the ith random variable. No more than one state from random variable i can occupy the ith position in the vector.*

1. The algorithm generates and reports solutions.
2. The algorithm may report both complete and incomplete solutions, however, complete solutions are always better than incomplete solutions. The algorithm may be forced to send incomplete solutions until it finds its first complete solution.
3. For each solution generated, the algorithm also generates a joint probability. Since joint probabilities are only calculable for complete solutions, the algorithm must have an internal mechanism for dealing with solutions that are incomplete. When reporting an incomplete solution, the algorithm reports a joint probability of zero.
4. The algorithm is an anytime algorithm. Solutions are produced continuously, and any time a higher quality solution is achieved, it is immediately broadcast to PESKI. Note that only full solutions are sent.

5. The algorithm must terminate when either a best possible solution is found, or a user specified time limit has expired.

Now that the foundation has been presented, we discuss the algorithms for reasoning over BKBs.

3.2.1 Genetic Algorithms. GALGO [18] used GAs to perform belief revision over Bayesian Networks; Genetic Algorithms can also be adapted for the belief revision on BKBs [3].

Genetic algorithms [10, 15, 18] are a nondeterministic approach for global search. The method, based on natural genetic evolution, allows the optimal solution to evolve from a pool of solutions called the *population.*

Each solution is a full instantiation (full collection) from the BKB; this is mapped into a *chromosome* by treating the solution vector V as an array of integers. Each random variable has one position in the array, and the contents of that position represent the state of that random variable. The GA works by using the chromosomes contained in the population to generate the members of a new population. Each iteration is called a *generation*, typically a three-stage process: chromosomes are *selected* for propagation to the next generation, then a *crossover* operator is applied to the selected chromosomes and finally a *mutation* operator is applied to the result of the crossover step. The result is a new population that is used as the basis for the next generation.

The crossover operation can take many forms, but in a simple GA it typically exchanges one or more randomly chosen segments between two chromosomes. In our representation, this is equivalent to exchanging the instantiation of small subgraphs within the BKB. The goal of this operation is to find those segments of the chromosome that produce good partial solutions and combine them to form a better full solution. The result is that the population as a whole tends to have more of these better solutions and (hopefully) eventually converges to the optimal solution. The mutation operator, on the other hand, takes a chromosome and randomly makes a change to one or more of its states. This process helps keep the population diverse so that the GA is not trapped into a local maxima in the search space.

The most critical part of the GA is the *evaluation function*[2]. This evaluator is what determines how good a solution actually is, which in turn determines which solutions contain enough good components to make them worth keeping around. A good evaluator function needs to provide enough resolution in its scoring to create smooth slopes for the algorithm to climb. Less resolution in scoring causes step-like or spiked slopes with large flat regions rather than continuous smooth inclines, resulting in poor performance for the GA.

When applied to a BKB, the GA generates *solutions*, which assign a state to each of the RVs. Once the solution is generated, it is up to us to evaluate or score it. If each of the states has exactly one support active, the solution is complete. For example "A=T", "B=T" and "C=T" is a complete solution in Figure 2.3. It is relatively easy to evaluate a complete solution. Multiply all the probabilities of active support nodes together to get a joint probability for that solution.

Joint probability can be calculated as follows:

```
Let JP = 1.0
For each state in the solution do
1.  find S, the active support node for the given state
2.  JP = JP x Value(S)
```

Recall that in order to find an active support node, all of its parent instance nodes must also be in the solution. Unfortunately for many solutions, we cannot find all of the parents for any of the supports supporting a state. When no support nodes are active for a given state in the solution, we do not have a factor for this portion of the joint probability calculation. This causes our joint probability to become undefined.

We can easily calculate a joint probability for a complete solution and therefore score it. Unfortunately, when a solution is incomplete, this scoring method no longer applies. For a real world BKB, there are likely to be many incomplete solutions (potentially there could be more incomplete solutions than complete ones). Yet there may be subsets of states in the solution that are members in the optimal solution for the network. If we threw away

[2]The evaluation function has also been referred to as a fitness function: it determines how fit the chromosome is for survival in the next generation

an incomplete solution, we could possibly loose the only chromosome that contained that portion of the solution. Therefore, we must keep certain chromosomes in our gene pool. If we kept all incomplete solutions, however, our gene pool would soon become cluttered with bad chromosomes. Thus, we develop a way of evaluating the incomplete chromosomes even though they have no defined joint probability.

Our goal is to be able to evaluate a particular solution even when that solution is incomplete. We want the score for an incomplete solution to be *lower than the lowest possible joint probability for any complete solution.* The score must be lower because we want the GA to have a lower likelihood of selecting an incomplete solution than a complete one for the next generation.

Knowing the incomplete solution must have a score lower than the lowest possible complete solution, we estimate a bound on the lowest possible complete solution as follows: For each component in the BKB, there exist a number of states, each having its own set of support nodes that could potentially activate it. We take the product of the minimum valued support nodes, one from each component to compute the minimum possible joint probability for the BKB[3]. Formally, we say

$$E = \prod_{j=1}^{n} S_{j,min} \tag{3.1}$$

where n is the number of random variables and $S_{j,min}$ is the lowest valued support for the random variable j.

Using this lower bound, incomplete solutions are scored in accordance with the requirements stated above:

$$score(V) < E \tag{3.2}$$

Exactly which method used to compute $score(V)$ is unimportant, however, because the computation is a critical step in the evaluation function, and one that is iterated over many times in each generation, it should be computationally efficient. There is one

[3]This estimate may not represent the complete solution with the lowest joint probability, but it is guaranteed to be as low or lower than the minimum possible joint probability for *any* complete solution

other factor involved in choosing an implementation: Should a solution that is "more incomplete" score lower than one that is "less incomplete?" A solution is more incomplete when it has more random variables without active support nodes (it has a higher *level* of incompleteness). Such a solution is less likely to be transformed into a complete solution in the next generation via the mutation and crossover functions of the GA.

When scoring a given solution based on the *existence* of any incompleteness, we could simply return a score that is less than E as soon as we find our first incompleteness. If we choose to score on the *level* of incompleteness rather than the existence of incompleteness in a solution, however, we must step through every state in the solution V to find all incompletenesses. This takes linear time, $O(n)$, where n is the number of supports. Typically in a real world BKB, there are many more possible incomplete solutions than complete ones due to the sparse population of supports. Therefore, the GA working on this knowledge will tend to select solutions that contain many incompletenesses. By choosing to only find the first incompleteness, i.e. the *existence* of incompleteness, we can reduce the time it would have taken to go through all of the knowledge by a factor X, where X is the average number of incompletenesses in a solution.

The problem with finding only the existence of incompleteness, is that the variance of scores in a population is severely limited. With scoring based on existence of incompleteness, the scores for all incomplete solutions must be the same[4]. The GA then has a harder time discerning which solutions to select for the next generation. This will make it difficult to reach the optimal solution. In the implementation, used for this research, we chose to search for all incompletenesses and score according to the level of incompleteness in the solution. The method is justified because a solution that has a higher level of incompleteness is less likely to be complete on the next generation than one that has a lower level of incompleteness. The equation we use is

$$score(V) = \prod_{j=1}^{n} (value(S_{j,complete}) \vee k) \tag{3.3}$$

[4]Since all incomplete solutions are by definition impossible, it would be unfair to place any variance in the scoring routine such that one incomplete solution seemed more likely than another.

where $S_{j,complete}$ is the value of an active S-node supporting state j and k is a penalty factor $k < E$. That is, for each random variable, if we find an active support for the state, use that support's probability in the product; otherwise, use the penalty in the product.

Given a BKB, replacing any one element of the joint probability calculation with the penalty factor k will cause the score to be less than E since $k < E$ by definition. If more than one incompleteness exists, the penalty factor becomes k^x where x is the number of incompletenesses; if $k > 0$, this produces the desired ordering based on the level of incompleteness. Since $k^x < k < E$, the score for multiple incompletenesses also meets the requirement of equation (3.2), which can now be restated as

$$score(V) \in (0, E) \tag{3.4}$$

Now we summarize the computation of the fitness (score) of the solution for the GA.

1. Find the lowest valued support for each random variable, $S_{j,min}$. This only needs to be done once (when the BKB is loaded).
2. Calculate $score(V)$, using formula (3.3) with $k \in (0, E)$

With this fitness function defined, we can build a Genetic Algorithm that can handle reasoning over an incomplete Bayesian Knowledge Base.

3.2.2 Best-First Algorithms. Our topological search algorithm uses best-first search as a backbone. With BFS, we are guaranteed that if we find a solution, it is the optimal one. However, an unmodified textbook best-first search could not meet the requirements of inference over a BKB. Specifically, we have added the requirements that the reported solutions are complete and that implicit exclusivities (only one state per random variable is chosen) have been maintained in the BKB[5]. Therefore, we developed an algorithm that only visited parts of the search space that contained valid solutions (solutions that are complete and meet the mutual exclusive criteria). In the next few

[5]Even though all of the states of a random variable are present in the directed graph of a BKB, we can only select *one* state per random variable to be present in our solution. Thus, all states of a given random variable are mutually exclusive, and our search space must represent this mutual exclusivity.

paragraphs, we prove that the *algorithm searches for the solution in a search tree that contains only valid solutions.* The overview of the proof is as follows:

1. We describe the search space (graph) S_1 that contains all solutions both full and partial.
2. We develop an algorithm C that visits a subset of this search space which contains only complete solutions (both full and partial). The subset of S_1 visited by this algorithm is called S_2
3. We develop an algorithm T that traverses S_3, a subset of S_1 that contains members with only one state per component in any given solution. Furthermore, this algorithm T only visits each member of S_3 once (no duplication). The algorithm effectively orders the members of S_3 into a *tree* instead of a graph.
4. Finally, we combine the actions of both algorithms such that the final search space is the intersection of S_2 and S_3. The result is a search tree S_4 that allows us to reach any and only valid solutions from the root without ever generating any duplicates.

First, we develop the notion of a search space containing the universe of possible combinations of states in the BKB:

> Let S_1 be the search space (directed graph) that contains all possible combinations of states in the graph. The search space includes the null solution and contains paths from the null solution to every possible combination of one or more states in the BKB. This search space contains both valid (A=T, B=T, C=T) and invalid (A=T, A=F, B=F, C=T, C=F) combinations of states.

Now we present a rule that we use to generate a subset of S_1 that contains only complete solutions.

Rule 1. *The selection criteria for which node to choose when expanding a solution in algorithm C requires that if we add a node to an expansion, that node must be either a root node, or have all of its parents already in the current solution*[6]*. Furthermore, we restrict C such that it only can generate solutions with at most one state per random variable.*

Lemma 1. *Only complete solutions are generated by the selection criteria.*

[6]The parental constraint is an efficient means for guaranteeing completeness, but prevents the algorithm's use on cyclic graphs. Generally, the knowledge acquisition process will not allow cyclic graphs [1], however, if the problem domain includes cyclic graphs, other measures must be taken to ensure the solutions are complete.

Proof. When expanding a partial solution, the selection criteria requires that the nodes it considers to add to the solution are either root nodes or nodes that have all of their parents already in the partial solution. Since the definition of an incomplete solution is that it is a solution that contains at least one state that is not a root node and does not have all of its parents specified in the solution, the selection criteria cannot generate incomplete solutions. □

Now we describe a restricted search space that only contains complete solutions.

> Let S_2 be the search space (directed graph) of all complete solutions, both partial and full. This space is a subset of S_1 that can be generated in algorithm C by Lemma 1

The problem with a textbook best-first search is that its search space S_2 is a graph. Since there may be more than one path from the root node to a given partial solution, we may traverse an entire path only to discover that it terminates in a partial solution that we have already found through some other path. To eliminate this duplication, we develop a search *tree* that guarantees no two paths will ever arrive at the same partial (or full) solution.

> Let S_3 be the search tree of all possible full and partial assignments of the BKB that have no more than one state selected from each random variable. This tree is a subset of S_1, but is not a subset of S_2. Now suppose that there is a hypothetical algorithm T that can generate S_3.

Now we develop the hypothetical tree generating algorithm T. A BKB is a directed graph (DG). A map M_0 is an arbitrary ordering of instantiation nodes in the DG. The underlying concept of the algorithm T is to grow a tree by pulling a partial solution out of a container [7] Q, expand that solution by filling in one member of the solution vector, and put the new expansion back in the container. It continues this process until no solution vectors remain in the container. We formally define T as an algorithm that generates a search tree. The definitions and steps for the algorithm follow, along with a set of rules that govern its execution.

[7] This container is not the equivalent to the open queue on in the best first routine. It is a conceptual device that contains no ordering of its members.

Definition 6. *The* null *solution is a vector where for all i, V_i has not been instantiated.*

Definition 7. *A* partial *solution is a vector where there exists an i such that V_i has not been instantiated.*

Definition 8. *A* full *solution is a vector where for all i, V_i has been instantiated.*

1. Attach M_0 to the null solution. Put the null solution in the container Q.
2. Pull a partial solution V and its attached map M from the container Q.
3. Remove M_{top} from the top of M. The remainder of the map is M'
4. Create a new solution vector V' by inserting of M_{top} into a copy of V.
5. Attach a one copy[8] of M' to V', and one copy to V. Place both V' and V back in the container Q.
6. Repeat Steps 2 through 5 until the container is empty.

Rule 2. *[No Conflicting Instantiations] We can only expand a branch from a given solution by adding nodes that do not already have their random variable instantiated (i.e. if A=T already exists in solution V, we cannot use node A=F also). If we cannot use the top node on the map, we remove it and try the next node.*

Rule 3. *[No Further Expansions] If we remove a partial solution vector V from the container Q and we cannot expand V any further, because there are no more members of the map, we throw V away.*

Rule 4. *[Leaves] A branch is terminated by a leaf. Each leaf is a full solution vector (one state for each random variable). When we arrive at the leaf, we don't have to put the leaf in the container because we will not be expanding it any further.*

Now we prove that the algorithm T produces a search tree that

- Contains no more than one state per random variable in any solution.

[8] In the actual algorithm, we use an indexing scheme on a global map rather than pass copies of the map every time, but here we use the concept of passing copies for conceptual ease.

- Is capable of achieving all possible full solutions (both complete and incomplete)

Lemma 2. *Algorithm T only generates solutions that contain no more than one state per random variable in any solution.*

Proof. Rule 2 forces any solution to contain no more than one state for every position in the vector. Thus, our the property holds. □

The proof that the search tree is capable of achieving all possible full solutions follows. It is comprised of several Lemmas and a culminating Theorem.

Lemma 3. *Given a partial solution V and a map M, all expansions that extend V with a single non-conflicting member of M will be generated as immediate descendants of V.*

Proof. By Steps 2 through 6 and Rules 2 and 3, we are forced to make one expansion for every member of the map that does not conflict. Thus, given a solution V with map M, we will generate $n =\mid M \mid -c$ new child solutions where $\mid M \mid$ is the number of members in the map and c is the number of members of M that conflict with members already present in V. □

Lemma 4. *The root will be expanded once for every node in the map.*

Proof. Lemma 3 showed that given a solution V and map M, we will generate $n =\mid M \mid -c$ expansions of the solution, where c is the number of nodes in the map that conflict with V. When expanding the root, we are starting with V_0, the vector that has no members, and thus no conflicts. $c = 0$ and $n =\mid M \mid$, the size of the map. Therefore, every node in the map becomes a member of a immediate child solution of the root. □

Lemma 5. *If a solution V is not instantiated in position i, and any node that could fill i is present in the map M , all nodes that can fill the position i will be in the immediate descendants of V.*

Proof. By Lemma 3, every member of the map that does not conflict will be generated as an immediate descendant of V. Since position i is currently empty, there are no conflicts to prevent us from using each node that could fill position i in a separate immediate descendant from V. □

Lemma 6. *If a node is removed from map M' because it caused a conflict with solution V', that node is already a member of a partial solution expanded by an ancestor of V'.*

Proof. If a node on map M' conflicts with position i in V', we know that the node in position i must have been added during an expansion in an ancestor of V'. Let us call the solution that was a parent of this expansion V. V was uninstantiated in position i. By Lemma 5 all other nodes that could fill this position (i.e. conflicting nodes) became members of the direct descendants of V. □

Lemma 7. *If a node is a member of a solution V, that node does not need to be in the map for future expansions of V or any expansions of the descendants of V.*

Proof. By Rule 2 **No Conflicting Instantiations**, once a solution vector V' is generated by expanding solution V with a node from M, all of the descendants of V' will also contain that node. Therefore the node can safely be removed from the map M for future expansions of V and future expansions of V' without eliminating any potential leaf solutions that include the node. □

Lemma 8. *If a node conflicts with V, that node does not need to be in the map for future expansions of V or any expansions of the descendants of V.*

Proof. By Rule 2 **No Conflicting Instantiations**, if a node has already been removed from a map M because it conflicted with an ancestors' solution vector, it will still conflict with this solution vector and every descendant of this solution vector. Thus, the node is not needed in this map M or the map of any descendant of this solution to reach all potential leaf nodes. □

Lemma 9. *If a node was removed from map M by a previous step, we are guaranteed that all leafs containing this node have been or will be visited.*

Proof. The proof for this Lemma is composed of two parts: one for the case that the node was removed because it conflicted with a member of the solution and the other for the case if it was removed during execution of Step 3.

Part 1: If a node was deleted because it was a conflict with a previous solution, then Lemma 6 shows that this node will be a member of another branch.

Part 2: If a node is not present in a given solution's map, and that node is not in conflict with any member of the solution, then it must have been removed using Steps 2 through 6 in an expansion that has already occurred in another branch of the search tree. As such, we are guaranteed that either there is at least one partial solution in the container that has this node as a member or that all of the leaf vectors that contained that node have been expanded by Rules 3 **No Further Expansions** and 4 **Leaf Nodes**. We need not concern ourselves with this node because Lemmas 3 and 7 ensure that partial solutions containing the node will be expanded to leaf nodes if they have not been already. □

Lemma 10. *There can never be more than one path from a vector V to a descendant V' of that vector that contains a node used to expand vector V.*

Proof. Once a solution vector V is expanded, the expanding node M_{top} is removed from the map and no future expansions of the vector V or any expansions of its descendants can possibly generate descendants that use M_{top}. Thus, there can never be more than one path from a vector V to a unique leaf node that contains M_{top}. □

Theorem 1. *The algorithm T, guarantees to develop a tree that contains every full solution (leaf) composed of the members in the map. (Algorithm T generates a search tree S_3.)*

Proof. The algorithm does not cease until the container is empty (Step 6). The container can only be empty when the single step expansion function pulls the last solution from the container and does not produce any descendants to place in the container. There are only two occasions when Steps 2 through 6 do not put new solutions back in the container: when the expansion uses the last node on the map and creates a leaf or when the expansion removes the last node on the map because it is a conflict. In either case, the entire map M_0 is exhausted, and we know from Lemma 9 that all leaves containing each node of the original map must have been visited. We know that this search space is a tree because for any solution, Lemma 10 states that there is only one path from the root node V_0 to it. □

Let S_4 be $S_2 \cap S_3$. This is a search tree of all possible full and partial complete assignments of the BKB that meet the mutual exclusivity requirement. As a tree, it guarantees that no two paths will arrive at the same partial solution. Because of the intersection with S_2 it guarantees that there are only complete solutions in the tree.

Theorem 2. *The intersection of S_2 and S_3 is a search tree S_4 that contains only complete solutions.*

Proof. Since S_3 is a tree containing *all* solutions, and S_2 is a graph that contains only complete solutions, their intersection produces a tree (because a tree is a subset of a graph) that contains only complete solutions (because complete solutions are a subset of all solutions). □

IV. Algorithm Selection

This chapter stresses the importance of generating a strategy for inference and presents the key research topics that inspired this thesis' method of inference strategy development.

In the last chapter, we developed algorithms for reasoning over Bayesian Knowledge Bases. If one of these algorithms could be shown to always outperform the others, we would keep it and discard the rest. Unfortunately, Wolpert and Macready showed "that all algorithms that search for an extremum of a cost function perform exactly the same" and that "In particular, if algorithm A outperforms algorithm B on some cost functions, then loosely speaking there must exist exactly as many other functions where B outperforms A" [23]. Thus, *given that we don't know anything about the data we are searching over*, finding a search algorithm that would out-perform all other algorithms under all circumstances is impossible. Without the possibility of finding the perfect algorithm, these authors suggest that researchers adopt *special case* algorithms to handle inference over specific classes of problems. The special case algorithms should be chosen depending on the nature of the problem to be solved.

The decision of which algorithm to choose in a single algorithm system for a special case is simple: choose the one that has the best performance in that special case. Determining a priori which algorithm will have the best performance in the special case is the inspiration for this thesis. Given the NP-hardness of reasoning over probabilistic networks, and the limited resources available, our only hope is to choose the algorithm that has the best chance of achieving a high solution quality quickly.

Similarly, if the inference system is using multiple algorithms to solve a problem, we must be sure to choose the algorithm mix that has the best expected performance. Wolpert and Macready's work on performance of two algorithms [23] can intuitively be extended: running a mix of algorithms A,B, and C over a many instances of data will have an average performance no better than algorithm mix D, E, and F. Therefore, if we wish to achieve

a better than average performance, we must choose a custom algorithm mix that is suited for the particular case we are inferencing over.

The choice of which algorithms to run on a given problem is constrained. We have limited computing resources and limited time, yet we desire an answer as soon as possible. Common sense tells us that the longer we are willing to wait, or the more computational power we use to solve a problem, the better the answer will be. In most cases, over the course of one inference problem, our set of computing resources will be fixed; it is unlikely that a new computer will be purchased and incorporated into the reasoning process before the system has provided a solution. The two variables that remain, *time* and *solution quality*, are traits that are inversely proportional when regarding the inference process [24] If we examine the solution quality of a given algorithm over the time it took to produce that solution, we would be looking at a performance profile of that algorithm (solution quality vs. time curve). If we had a way of knowing a priori what the performance profiles of algorithms A through F were for a given instance of a problem, we could determine which algorithms we would use in our mix. One task required of choosing the appropriate set of algorithms is to predict their performance profiles.

An exact algorithm with the anytime property, an algorithm that can produce results of monotonic increasing quality at successive interruptions to their processing, will have a consistent performance profile every time it is run on the same problem. Yet the performance profile of an exact anytime algorithm may vary from one problem instance to another. This phenomenon implies that there are characteristics inherent within the problem that affect the performance profile of the algorithm. If these characteristics can be isolated, the performance of an exact anytime algorithm can be predicted. Zilberstein and Russell [24] have classified performance profiles of anytime algorithms into several categories: Performance Distribution Profiles(PDP), Expected Performance Profiles(EPP) and Conditional Performance Profiles(CPP). PDPs and EPPs depend on the overall average response of algorithms. CPPs, however, describe performance of an algorithm in response to a specific characteristics of the input. Discovering domain characteristics in the input that effect variance in performance improves the accuracy of our CPP. Preliminary work in this area performed by Jitnah and Nicholson shows that domain characterization may

be useful for predicting inference algorithm performance on a belief network for a new application domain using Likelihood Weighting, Logic Sampling and Jensen's algorithms over belief networks [13].

An anytime approximation algorithm such as a genetic algorithm will rarely exhibit the same performance each time it is run on a specific problem. In fact, Zilberstein states that performance profiles in the strict sense can only be defined for deterministic algorithms [24]. Because the approximation algorithm doesn't have to "build" a solution by taking incremental steps through some search space (as does an exact algorithm), it may discover the optimal solution at any time in its execution. In fact, the optimal solution may actually be a member of the first population. On the other hand, the approximation algorithm may never find the optimal solution because of its semi-random solution generation process. At first, this may seem to inhibit our ability to predict performance. But as we shall see, the *average* performance of an approximation algorithm does tend to vary from one problem to the next.

The performance of an algorithm is based on two factors: the characteristics of its input domain and the reaction of the algorithm to the input. In this work, each algorithm is treated as a *black box*: If we assume that an algorithm will perform exactly the same with a given input (or with some average performance, in the case of an approximation algorithm), we can abstract the behavior of the algorithm from its performance. By doing this, we shift the responsibility of the performance of an algorithm to its input. If we can create a model of the input that captures features that affect the performance of the algorithm, we can predict the performance profile of the algorithm without knowing anything about the algorithm itself.

In the process of characterizing the input, we must keep in mind the following criteria when choosing the domain characteristics we wish to examine:

- Each feature must be measurable and representable as a single numerical value.
- Preferably, each feature should be mutually exclusive from all other features (such that each feature can be varied independently from the others).

- The time needed to calculate the features (or characteristics) of the input must be small relative to the time required for inference.

In the PESKI environment, the input to the inference algorithm is the knowledge stored in a Bayesian Knowledge Base. Since a BKB is essentially a graph that stores costs, it can be modeled in terms of various topological and probabilistic characteristics.

Topological characteristics range from basic countable features of the graph such as number of nodes and number of arcs, to computable features such as cyclomatic complexity [14] (the number of independent paths through a network). Cost characteristics range from the simply calculated cost-range metric to the more complex measure of extremeness [17]. This research uses a subset of these characteristics to predict the performance profile of an anytime algorithm.

Since each class of algorithm is different, and each requires its own set of characteristics for performance prediction, our complete set of characteristics must be the union of the individual sets that affect each algorithm we are examining. Once the set of characteristics has been chosen, we can begin to correlate the effects the characteristics have on the performance of the algorithms.

4.1 Predicting Performance

This section extends the usage of Zilberstein and Russell's concept of conditional performance profiles [24]. Originally, the CPPs were used to show how a characteristic of the input affected the performance of an algorithm. Here we present a method for using performance data gathered off line to predict future performance of algorithms on new problems. Ultimately, the ***predicted*** performance of algorithms allow us to select the algorithm that is most likely[1] to have the best performance on that problem.

Figure 4.1 shows a method for estimating future performance. A knowledge base is analyzed into a set of characteristics. Each algorithm is executed individually on a single knowledge base and provides a set of solutions. These solutions are analyzed and the performance profiles for each algorithm are generated. A mapping relation is created between

[1]Based on statistical analysis of the collected data.

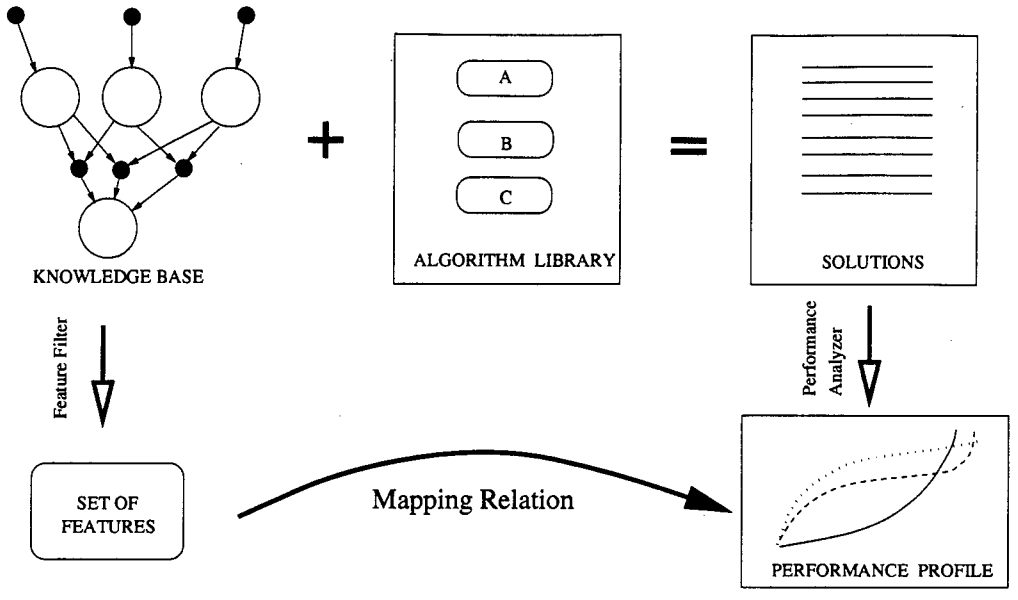

Figure 4.1 The Transformation: Features to Performance

the characteristics and the performance profile for each algorithm. The mapping relation is used to predict the performance of algorithms based on the domain characteristics of the knowledge structure.

Now we examine the mapping relation formally.

A conditional performance profile is a set Π of tuples $< \tau, \sigma >$ where τ is the time from the start of execution and σ is a solution quality at that time. Typically $0 \leq \sigma \leq 1$. Formally, the mapping relation λ can be defined in terms of the set of characteristics Γ, the algorithms A and the performance tuples Π:

$$\lambda : \Gamma \times A \rightarrow \Pi \tag{4.1}$$

To predict the performance of an algorithm, we must develop the λ relation. Development of this mapping relation is composed of five steps:

1. Find a set of characteristics of the network that are *differentiating characteristics* (DC).

2. Collect networks that exhibit a variety of values over all of the selected DCs. This collection is called the *test suite.*

3. Gather data on the performance of algorithms when the DCs are varied.

4. Generate the relation that can be used to predict performance of the algorithms.

5. Test the relation.

Now we examine the steps to achieve this goal in detail.

4.2 Differentiating Characteristics

To be a differentiating characteristic, a feature must possess the quality that varying it causes different performance responses on each algorithms. For example, let us suppose their is some hypothetical characteristic γ that we wish to examine.

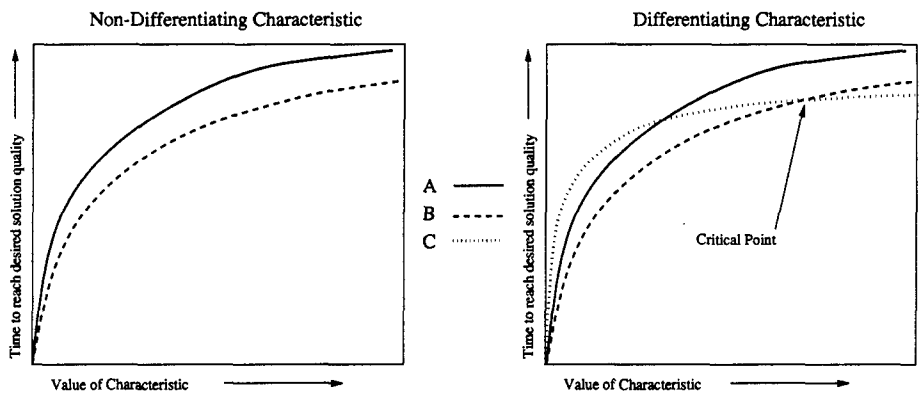

Figure 4.2 Non-Differentiating vs. Differentiating Characteristics

Figure 4.2 shows how the value of γ might affect two hypothetical algorithms A and B[2]. On the left side of the figure we see that as γ increased, it took longer to reach a desired solution quality for both algorithms. Even though they both have different curve constants, the two curves never intersect once they leave the origin. Thus, we would always select algorithm B for inference because regardless of the value of γ, algorithm B always

[2]The hypothetical performance graphs shown assume that all other performance-affecting characteristics except for γ were held constant. In reality, it is nearly impossible to hold all other characteristics constant while varying only one. These graphs present an idealistic situation.

takes less time than algorithm A to reach a given solution quality. Therefore, γ is not a differentiating characteristic between algorithms A and B. If we look at the right side of the figure, we notice that the performance curve for hypothetical algorithm C intersects the performances of both the other algorithms. Thus, when γ is greater than the critical point (the projection of the intersection of curves B and C on to the x axis), we would choose to execute algorithm C, but when it is below the critical point, we would run algorithm B.

A secondary requirement of a DC is that it must be easily computable. If the time to compute a characteristic takes as long as actually executing the inference algorithm, it would be unwise to invest the time to compute it.

Now each of the characteristics examined in this research is discussed.

4.2.1 RV Count. RV count is one of the easiest metrics to compute for a BKB. RV count is the number of random variables in the BKB. RV count is a potential DC for differentiating performance of Genetic Algorithms and Best First Search because it is a topological characteristic of the network. A GA should be affected less by the variance of RV Count than BFS.

4.2.2 Completeness. Completeness is a measure of how fully stipulated our knowledge is, and is related to the number of support nodes[3]. Hence, completeness is a topological characteristic of our graph. Completeness is a key element that differentiates BKBs from Bayesian Networks.

Every Bayesian Network can be converted into a BKB. Bayesian Networks, by definition contain fully specified (complete) knowledge, therefore an equivalent BKB is also complete[4]. We define the upper bound of our completeness metric as the number of rules (support nodes) in this fully specified network (S_{max}). If we removed several support nodes and their adjacent arcs from the BKB it would become less than fully specified. Our lower bound for the completeness metric is defined by the minimum possible number of rules

[3]The number of support nodes is equivalent to the number of conditional probabilities: essentially this is how many "rules" we have in our knowledge. Therefore, the more support nodes we have, the more rules we have and the more complete our knowledge.

[4]To be more precise, a complete network is one that has a computable joint probability for every combination of instances of each component.

in a BKB. Banks' **Constraint 1** [1] for BKBs forces every instantiation node to have as a minimum one support in its tail. Thus, there must be at least as many supports in a BKB as their are instances of all its random variables. The lower bound of completeness (S_{min}) is equal to the number of instances (states) in the BKB. With these upper and lower bounds established, our metric for completeness is defined as the relationship of the number of support nodes (*supports*) in a BKB relative to the upper and lower bounds for that BKB.

$$completeness = \frac{\mid supports \mid - S_{min}}{S_{max} - S_{min}} \tag{4.2}$$

Note that we defined the maximum number of supports (S_{max}) with reference to a Bayesian Network. This value can also be calculated by figuring how many supports would be necessary to make a given BKB complete. This number is equivalent to the product of the number of states for every parent random variable having an arc to a child random variable multiplied by the number of states of the child's RV. For example if our BKB from Figure 2.3 had only supports s and x present (the minimum number of supports required by **Constraint 1**), we could calculate the number of supports required to make it complete by taking the product of the number of states of each parent random variable (2×2) and the number of states of the child random variable (2). Thus, $S_{max} = 2 \times 2 \times 2 = 8$ the number of support nodes to make this network complete.

Completeness is predicted to be a DC between the GA and the BFS because each algorithm has a different method for dealing with incompleteness. Our BFS, for example, does not expand a path to a solution if it contains any incompleteness. If there are many incompletenesses, the BFS may have to search for a long time before finding a path that doesn't contain incompletenesses. A GA, on the other hand, is not prevented from generating incomplete solutions, but it must evaluate each one it generates. Therefore, the level of completeness of the BKB should have a different effect on the performance of the GA than on the BFS.

4.2.3 Cyclomatic Complexity. McCabe's Cyclomatic Complexity [14] (CC) is traditionally used as a software engineering metric to determine if a section of source code is too complex to maintainable. Cyclomatic Complexity is effectively the number of independent paths through the code[5].

Formally, CC is

$$e - n + 2 \tag{4.3}$$

where e is the number of edges in the BKB and n is the number of nodes (both state nodes and support nodes). Note when CC is computed in the software arena, a virtual "Start Node" and "Finish Node" are added to the network. All root nodes in the network are then connected via arcs to the "Start Node", and all leaf nodes are connected via arcs to the "Finish Node". Thus, we effectively have 2 extra nodes in our computation and $|leafnodes| + |rootnodes|$ extra edges.

We adopt this metric because it has potential to be a DC between GAs and BFSs. A Best First Search can potentially travel down every independent path in the BKB before it finds a solution. In contrast, Genetic Algorithm is not concerned with the topological arrangement of nodes in the BKB, except in verification of completeness. Therefore, we assume that all else equal, varying the cyclomatic complexity would cause a Best First Search to demonstrate variances in performance, while the GA was unaffected.

4.2.4 Skewness. Skewness is a metric proposed by Jitnah and Nicholson [13] in their work on domain characterization. The equation they used is

$$skv(v) = \frac{\sum_{i=1}^{m} |\frac{1}{m} - v_i|}{1 - \frac{1}{m} + \sum_{i=2}^{m} \frac{1}{m}} \tag{4.4}$$

Skewness is computed for each random variable, where $v = (V_1, .., V_m)$ is the conditional probability table[6] for that RV. The top of the $skv(v)$ fraction essentially computes

[5]McCabe's metric describes the maximum number of independent paths, and thus, the number of test cases required to show that a piece of code works properly

[6]Although a conditional probability table is only formally defined for BNs, an equivalent structure can be defined for a BKB: it is a table composed of the supports that are parents of each of the states in a random variable. Thus, in Figure 2.2, the table is composed of supports s through z

the variance of the conditional probability table, and the bottom is a scaling factor that scales $skv(v)$ between 0 and 1.

In this work, we used the equation

$$skv(v) = \frac{\sum_{i=1}^{m} |\frac{1}{m} - v_i|}{\frac{m-2}{n_0} + 1} \tag{4.5}$$

where n_0 represents the number of states of the random variable being examined. The difference between the equation 4.4 and 4.5 is in the denominator. The denominator of equation 4.4 allowed the skewness to exceed 1.0, while equation 4.5 guarantees the skewness to be no greater than 1.0.

Given a conditional probability table, the maximum skewness occurs when all members of the table are either 0.0 or 1.0. In a probability table for a R, a n_0 state random variable, there exists t possible entries containing a probability of 1.0.

$$t = \prod_{i=1}^{k} n_i \tag{4.6}$$

where k is the number of parent random variables of R and n_i is the number of states of the ith parent. Thus, using the numerator of Equation 4.4, our maximum possible skewness is

$$t \times \frac{m-1}{m} + t \times (n_o - 1) \times \frac{1}{m} \tag{4.7}$$

which is reduceable to

$$\frac{m-2}{n_0} + 1 \tag{4.8}$$

Hence, we use Equation 4.8 as the denominator for our calculation of skewness.

Skewness is selected as a DC because it is a probabilistic characteristic, and therefore is more likely to affect the GA than the BFS algorithm. We expect that a BKB that has a large average skewness across all of the random variables is more likely to have a larger set of possible unique joint probabilities. With a larger set of joint probabilities, the gaps between possible joint probabilities become smaller. Hence there are smoother probabilistic slopes for the GA to climb. The BFS should not be affected by the smoothness of the probabilistic

slopes because it makes low level decisions about which path to take by selecting the path with the higher probability, and is not concerned how much higher the probability of that path is. Thus, skewness should differentiate the performance the GA from the BFS.

4.2.5 Extremeness. In a "normal" diagnostic network [17] each random variable has a most likely state (instantiation node). That is, the highest joint probability in the network is obtained by inferencing over the path containing the most likely states for each random variable. When evidence is obtained that does correspond to the most likely states of the random variables, we are said to have a fault, or problem condition. For example, the "normal" state of the human body is good health. If we could create a ideal network that represented the state of the human body, the most likely state of the network, given no symptoms of ailment would describe a condition of good health. When we introduce evidence of a symptom (a fault state) to the network, them most likely state of the network would change to reflect the condition of the body with the related health problem(s).

From a probabilistic viewpoint, in order to make an inference algorithm find the "normal" state of the network, the normal instantiation nodes must be much more likely (be supported by much higher probability support nodes) than the other instances of their respective random variables. Poole has shown that in such an "extreme" network [17], certain algorithms perform better (converge on a solution faster) than in non-extreme networks.

Poole [17] has indicated that to be "extreme", a network must exhibit the property that given n as the total number of non-normal instantiation nodes (fault states) in the network, there exists an upper bound $f = \frac{1}{n}$ such that none of the fault instantiations can have a conditional probability greater than f. We extend this notion to define the *level* of extremeness of a network by replacing the previous equation with $f = \frac{1}{e \times n}$. Hence, given that we know the highest non-normal probability in a network is f and the number of fault nodes is n we can calculate the extremeness of the network by

$$e = \frac{1}{f \times n} \tag{4.9}$$

With this equation, we get a network that Poole would consider extreme whenever $e \geq 1$.

In an extreme network, the solution space is a flat landscape of extremely low probability solutions with a handfull of very high probability spikes protruding from it. Since a GA seems to perform better in a smoothly sloped solution space, and a BFS seems to be able to quickly find the spiked joint probabilities, extremeness is a potential DC between GAs and BFSs.

4.3 Collecting Networks

After identifying which features are likely to affect the performance of our algorithms, the next step is to gather networks that exhibit a range of values for each network. Considering that there are very few BKBs in existence currently, it became necessary to generate networks. AFIT had a large number of randomly generated Bayesian Networks in existence. By converting these networks into BKBs then modifying the BKBs through several instances of each characteristic, we were able to generate a large sample space for our algorithm tests.

4.4 Gathering Performance Data

The next phase collects performance data for each algorithm for each candidate differentiating characteristic. Several networks were generated for each value of a characteristic $\gamma_1 \ldots \gamma_n$ in the previous phase. Each anytime algorithm in the set Λ is executed on each network[7]. Each run of a specific algorithm creates a performance curve of time vs. solution quality. When all of these performance curves are combined on a graph, they form a CPP, essentially a "contour" map (a way of representing a three-dimensional space in two dimensions) of the algorithm's performance over various values of γ. By taking a horizontal slice across the CPP and projecting the intersection of the performance curves onto the x axis, we achieve a dataset of time to reach a given solution quality (the utility

[7]Approximation algorithms necessitate examining an average performance curve over several runs because of their non-deterministic nature. Exact algorithms, in contrast require only a single execution because they are deterministic.

of the algorithm) vs. characteristic value. The subsequent plot of this data is a utility curve that reveals the characteristic's effect on the algorithm.

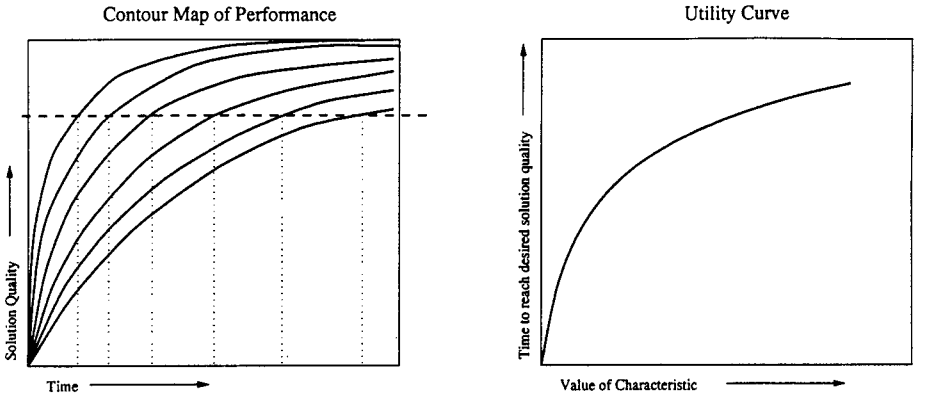

Figure 4.3 Conditional Performance Profile and Utility Curve

Figure 4.3 shows a hypothetical algorithm's CPP, and its corresponding x axis projection, the utility curve. When we plot utility curves from several algorithms, we achieve a plot similar to the one in Figure 4.2. From these multi-curve plots, it is easy to locate the critical points where two or more algorithms' utility curves intersect.

4.5 Creating a Relation

The next phase is to coalesce the critical points into a mathematical relation that can be used in the decision process for selecting which algorithm to run under constrained resources. The method this research uses is to examine the critical points. Whenever a decision needs to be made, we simply determine whether or not the DCs for a network are above or below the critical points. Then we choose an algorithm accordingly. The Conclusions and Future Research chapter discusses the limitations of this method in greater detail.

Once the relation has been created, we can use it to predict which algorithm will perform better on the given BKB. The next chapter shows the experimental results that allow a relation to be built.

V. Experiments

This chapter focuses on the effects of the distinguishing characteristics on the performance profiles of the GA and the BFS. Empirical results show that there are critical points in the utility graphs comparing these two algorithms.

Eighty-five networks were created, over a variety of values for characteristics. Metrics for the characteristics were collected for each graph. For each metric, graphs were categorized into subgroups based on the value of the metric. Then the average performance of each subgroup was taken. The resulting average performance curves were compared between the GA and the BFS. In the conditional performance profiles (CPP) presented in the next sections, the X axis represents time in seconds and the Y axis is the normalized logarithm of the solution quality[1]. The snapshot graphs (SS) provide a view of the characteristic (X axis) vs. the log solution quality at a given instant in time (Y axis). In the utility curves, (UC) the X axis represents the value of the characteristic, while the Y axis is the time to achieve a constant solution quality. Where two lines cross on a UC is the critical point where both algorithms perform the same.

5.1 Extremeness

This section examines the effect of extremeness on the performance of the GA and BFS algorithms. Figures 5.1 and 5.2 show the results of extremeness for the algorithms in the three subgroups: extremeness = 0.00-0.05, 0.05-0.50, and 0.50+. Each contain the average performance of both algorithms over several complete BKBs that fit into the extremeness subgroup. The actual averages for the low, medium and high extremeness metrics are 0.03832, 0.2724 and 20.53.

Figures 5.3 and 5.4 show zoom views of the effects of Extremeness on the BFS and the GA. Notice how the BFS is affected by extremeness: when reasoning over a knowledge base

[1]The curve is normalized by taking dividing the actual solution quality at any point by the best solution quality found by any algorithm in the system for that BKB. The graphs are presented on a log scale, therefore a 0.0 represents the maximum solution quality achieved by any algorithm for that BKB.

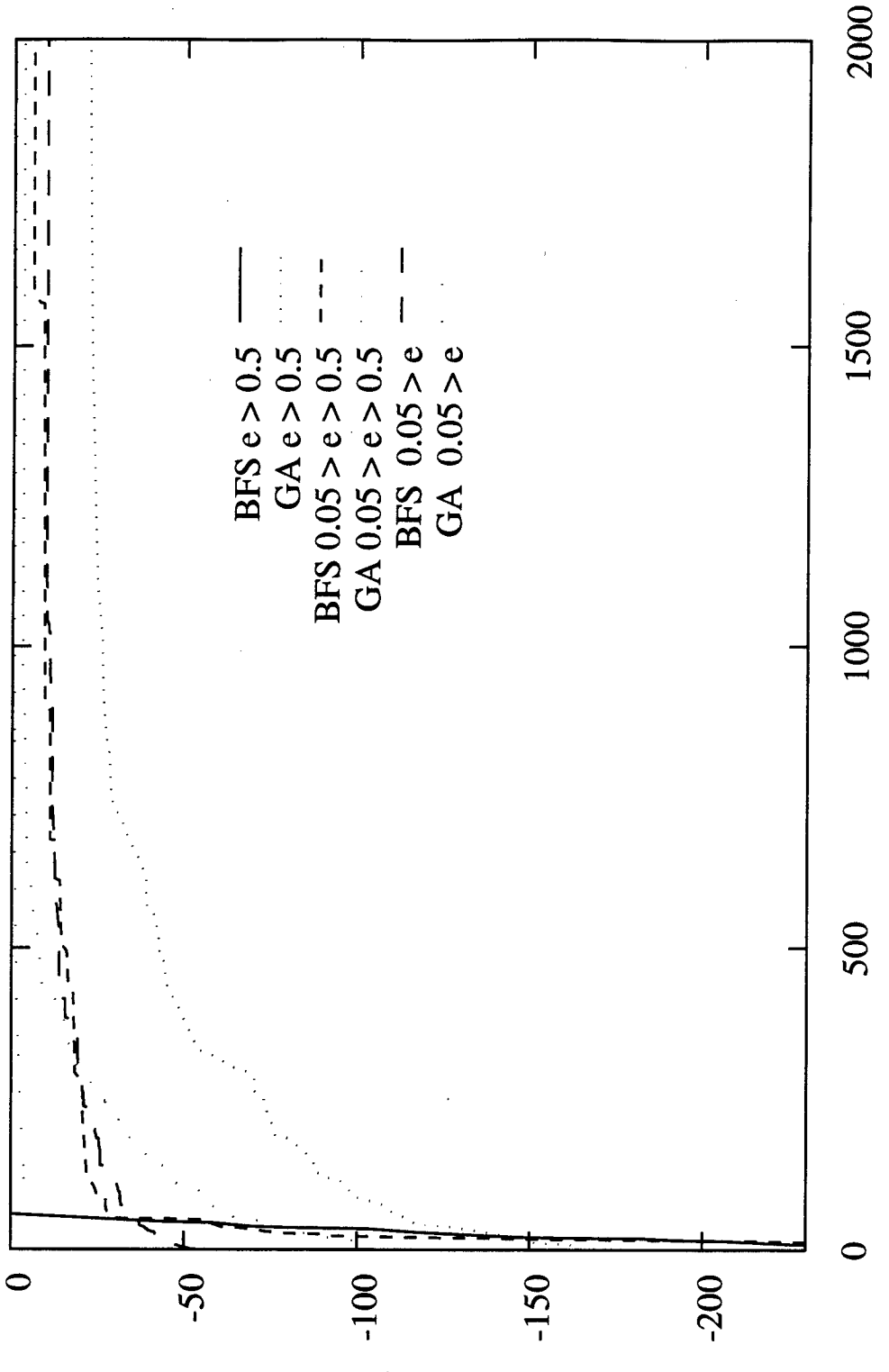

Figure 5.1 CPP: Effects of Extremeness on Performance of Algorithms

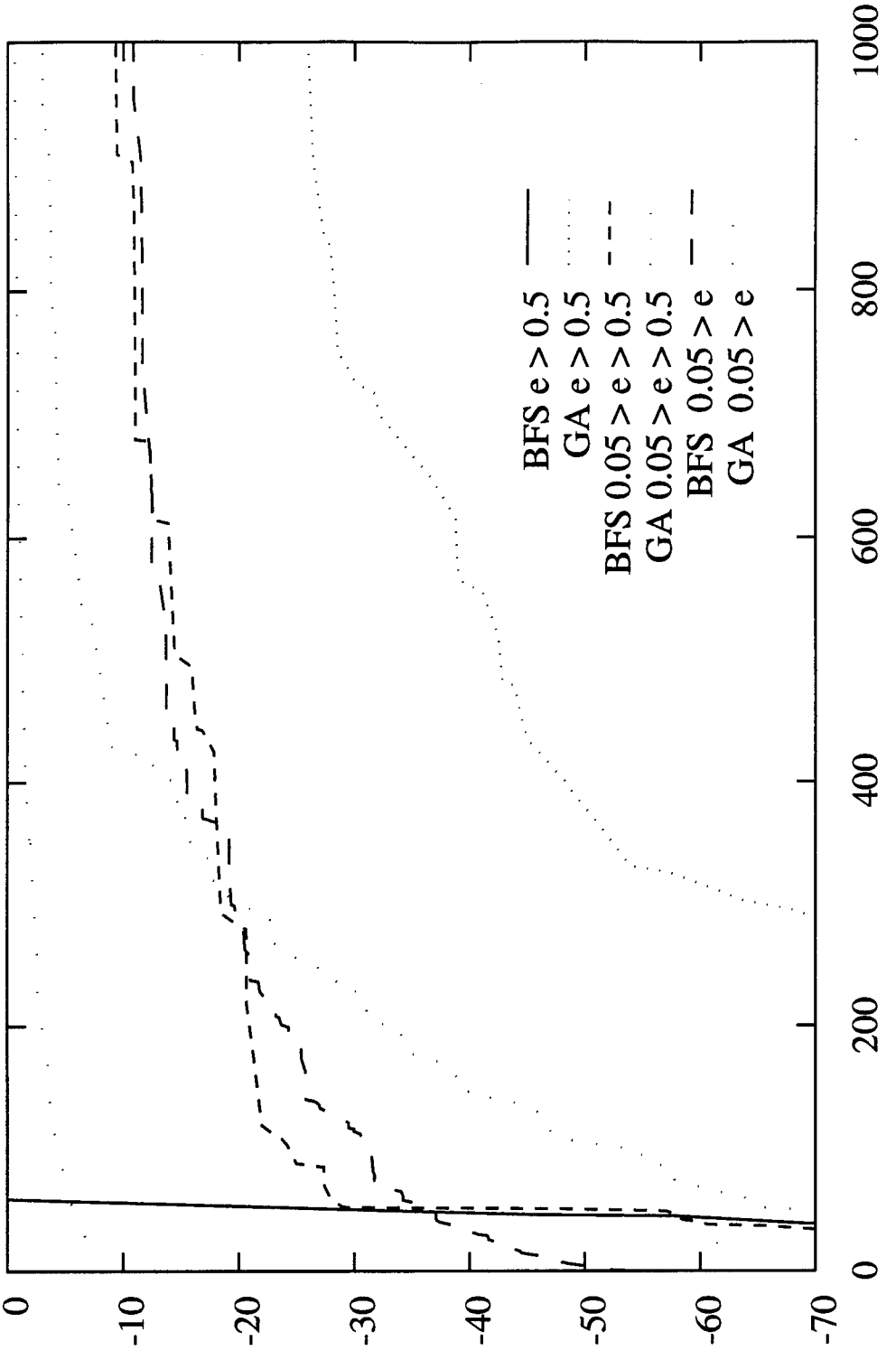

Figure 5.2 CPP: Effects of Extremeness on Performance of Algorithms (Closeup View)

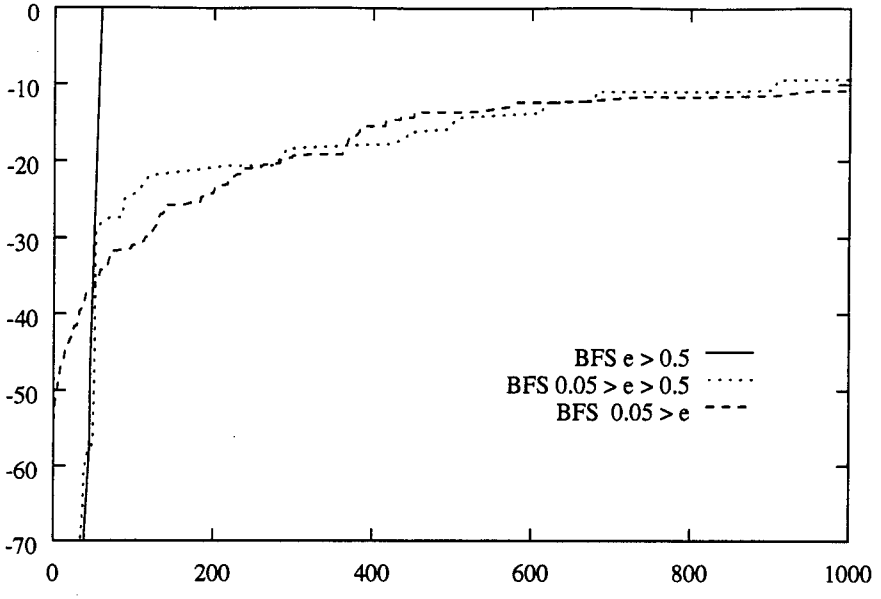

Figure 5.3 CPP: Effects of Extremeness on Performance of BFS (Closeup View)

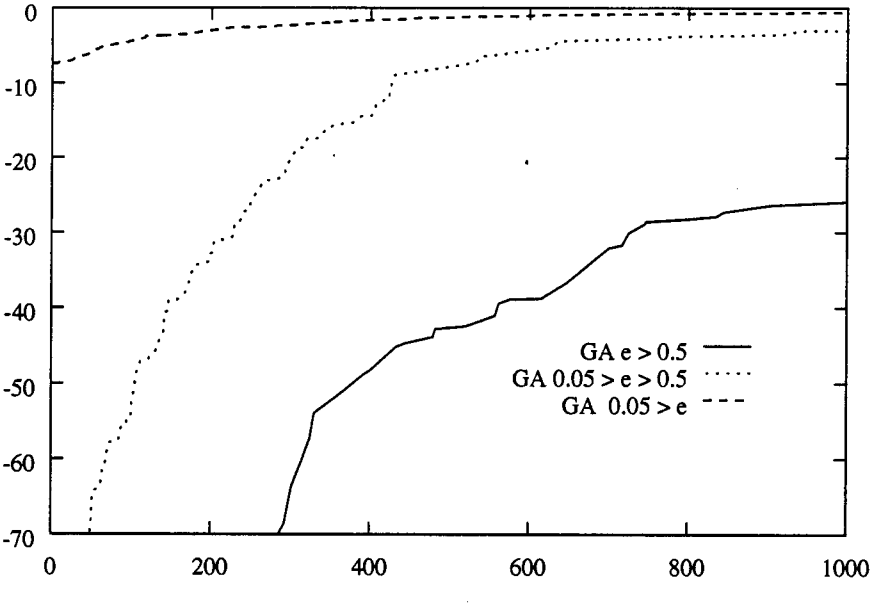

Figure 5.4 CPP: Effects of Extremeness on Performance of GA (Closeup View)

with high extremeness, it performs very well, yet if the extremeness drops below 0.5, its performance is sluggish. In contrast, the GA seems to do very well with low extremenesses, and very poorly with high ones.

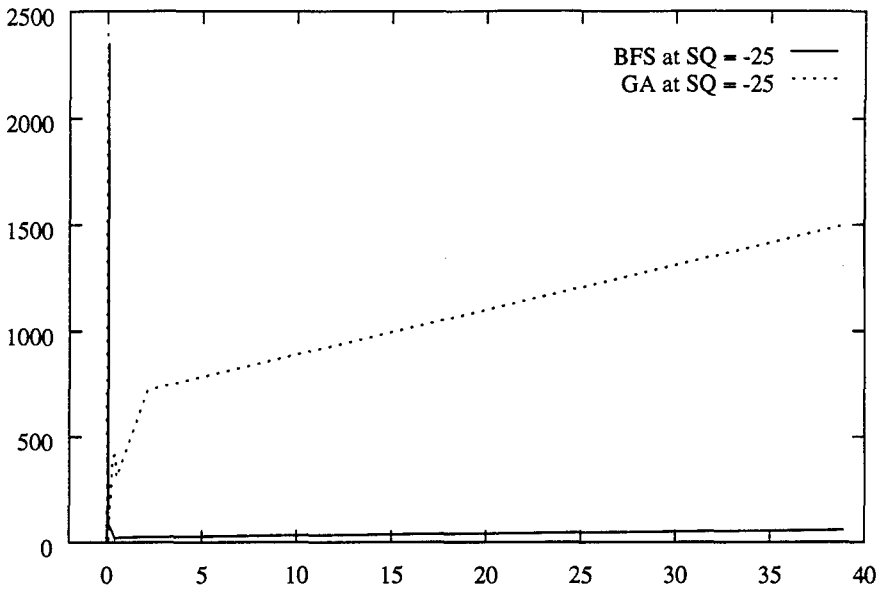

Figure 5.5 UC: Effects of Extremeness on BFS and GA at SQ=-25

The next step in the method is to generate the utility curves for each algorithm. The utility curves were plotted by taking a horizontal slice across the CPPs at a log solution quality of -25. The intersections of the various performance curves with this slice were projected onto the X axis to determine the value of the Y axis (time to reach log solution quality -25) in the utility curves. Then the utility curves were plotted. Note that for Figures 5.5 and 5.6, instead of using the average values for the subgroups of extremeness, we plotted the actual values observed from each network's performance. The result is a much more detailed utility curve.

Notice the critical point (crossing of the algorithms' utility curves) in Figure 5.6. At extremeness of approximately 0.1075, this utility curve shows that both algorithms took the same time to reach a normalized log solution quality of -25. Thus, if the extremeness of a network was greater than 0.1075, our strategy choosing scheme should choose to execute a BFS algorithm, yet if it was less than 0.1075, a GA gives a better performance.

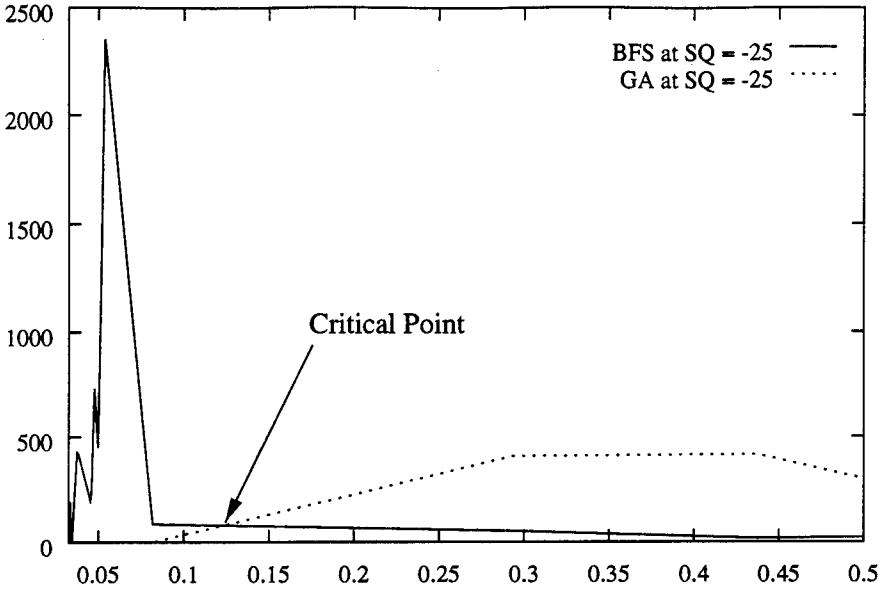

Figure 5.6 UC: Effects of Extremeness on BFS and GA at SQ=-25 (Closeup View)

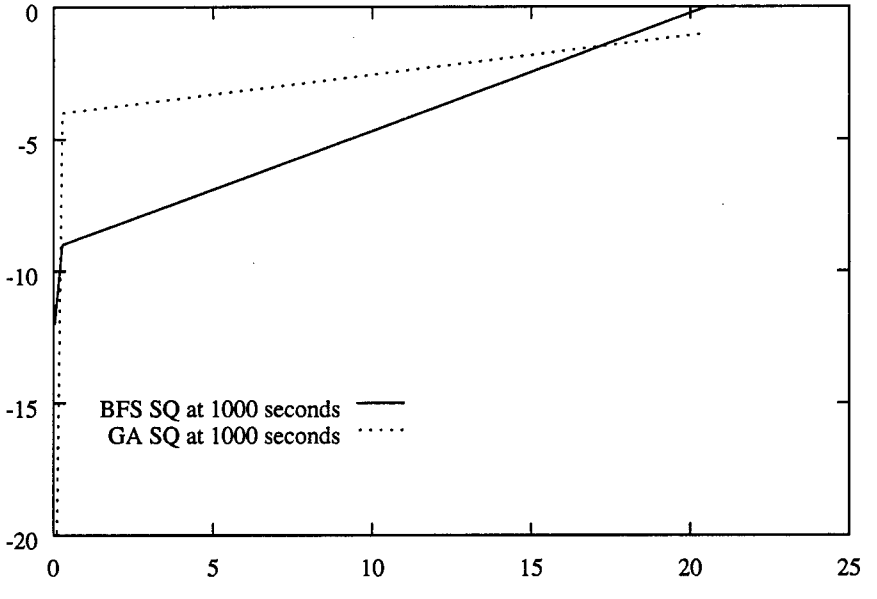

Figure 5.7 SS: Effects of Extremeness on BFS and GA at Time = 1000 seconds (Closeup View)

Unfortunately, a log solution quality of -25 is far from optimal. If we wish to compare the utility of algorithms as they approach the optimal solution, we need another indicator for showing average performance after the algorithms performances have leveled off. From Figure 5.1, we see the performance of the algorithms seem to level off at about 1000 seconds. Figure 5.7 is a snapshot of extremeness versus log solution quality.

Notice that at very low extremenesses, the BFS outperforms the GA. Through the midrange (1.0 < extremeness < 17.0) the GA performs the best, but above 17.0, the BFS once again becomes the algorithm of choice. Thus, once a CPP has leveled off, a snapshot graph can point out the relative performance of algorithms based on the value of a characteristic from that time forward.

5.2 RV Count

The effect of RV count (the number of random variables in the BKB) on the performance of the BFS and GA is examined in this section. Two groups of 20 BKBs each were formed, one of 20 RV BKBs and one of 30 RV BKBs. Figure 5.8 shows the resultant CPP.

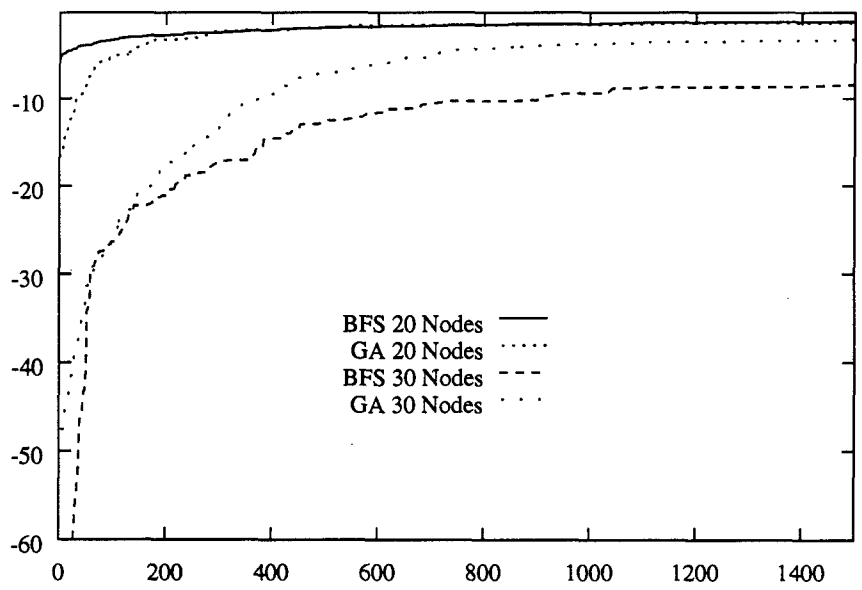

Figure 5.8 CPP: Effects of Random Variable Count on Performance of Algorithms

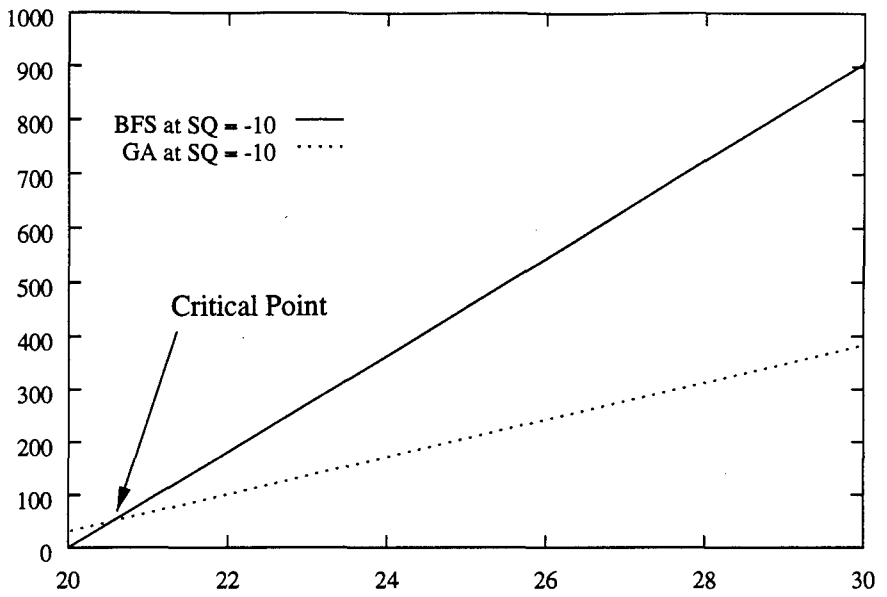

Figure 5.9 UC: Effects of Random Variable Count on Performance of Algorithms at SQ = -10

In Figure 5.9 we see the utility curve for the GA and the BFS when log solution quality is -10. Notice the critical point at approximately 21. Using this UC as a decision making tool, and assuming all other features equal, we would choose the BFS algorithm when the RV count was below 21 and the GA when the node count was above 21.

Another intersting feature of this graph is the slope of the two curves for BFS and GA. Notice that the slope for the GA is lower than the slope of the BFS (approximately 35 compared to 90 respectively). This phenomenon shows that the RV count has a much greater affect on the BFS than the GA, adding evidence to our claim that a topological-based search algorithm is more likely to be affected by topological features than a probabilistic algorithm.

5.3 Completeness

The effect of completeness on the performance of the algorithms is presented in this section. Four subgroups were composed of BKBS with 33, 50, 75 and 100 percent completeness. Figure 5.10 presents a picture of the performance profiles for the GA and

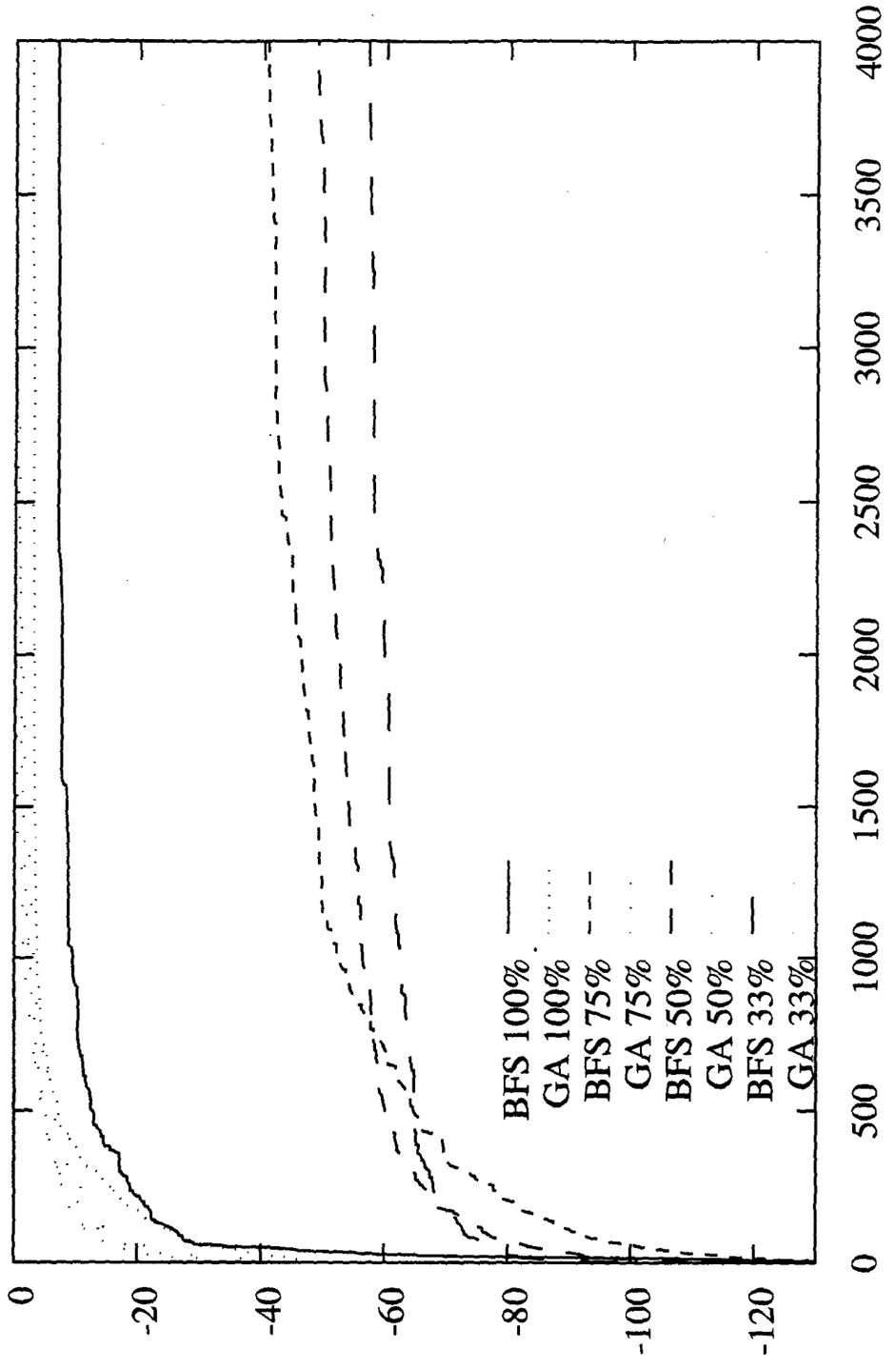

Figure 5.10 CPP: Effects of Completeness on Performance of Algorithms

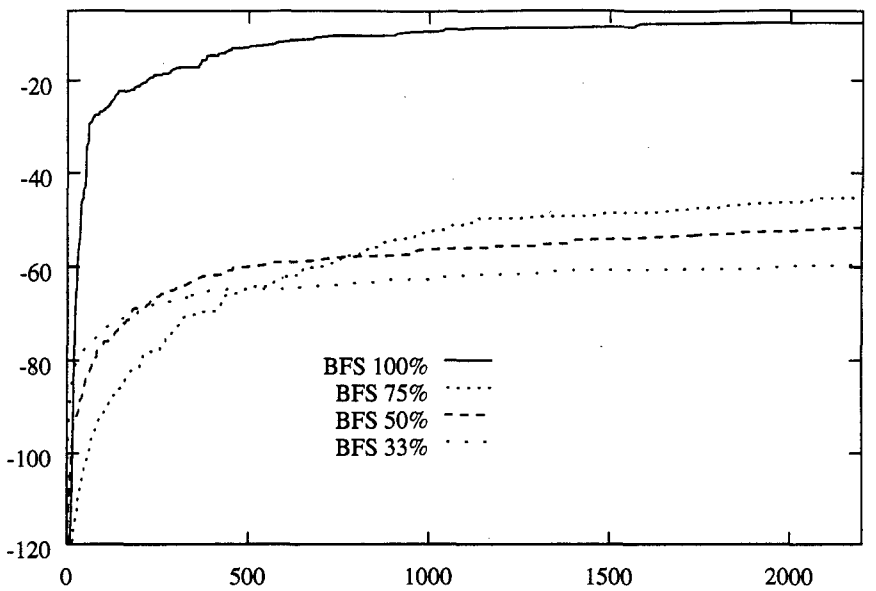

Figure 5.11 CPP: Effects of Completeness on Performance of BFS (Closeup View)

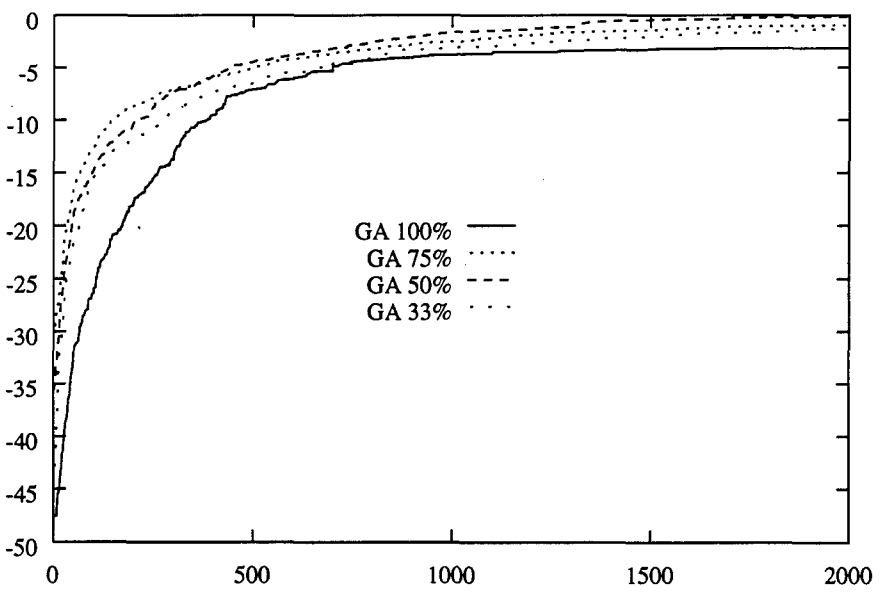

Figure 5.12 CPP: Effects of Completeness on Performance of GA (Closeup View)

BFS on each of the four groups. Figures 5.11 and 5.12 show close up views of the performances over each of the four groups.

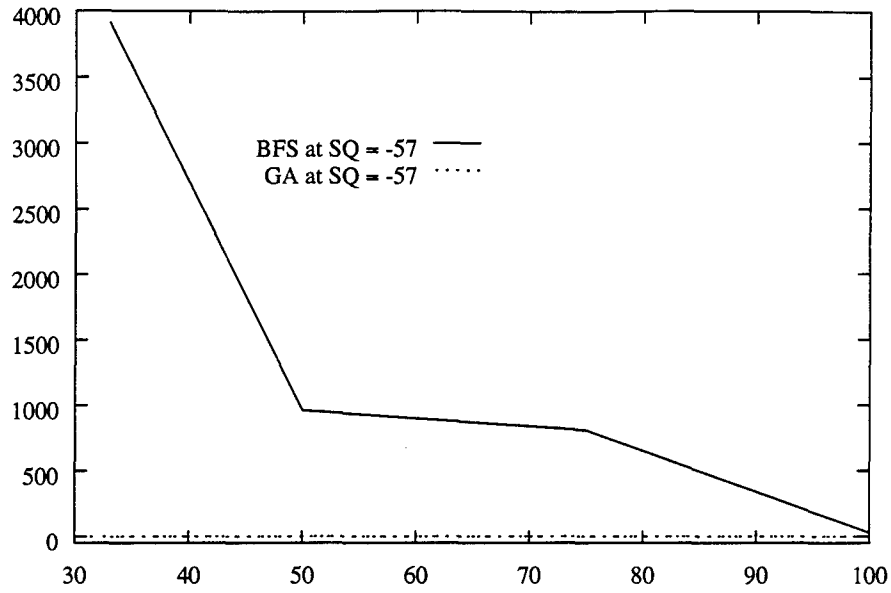

Figure 5.13 UC: Completeness vs. Time

A utility curve for the log solution quality of -57 is provided in Figure 5.13. A solution quality of -57 was the maximum average solution quality achieved by the BFS algorithm for a 33 percent complete graph. Therefore this was the maximum SQ point useable for a utility curve. Notice that the BFS is highly affected by the changes in completeness while the GA appears unaffected. Since completeness is a toplogical characteristic for a BKB, we would assume that it has a greater effect on the BFS than the GA.

5.4 Cyclomatic Complexity

This section examines the effect of cyclomatic complexity on the performance of the GA and BFS algorithms. Figure 5.14 shows the results of CC for the algorithms. The three subgroups, CC = 3000-4999, 5000-7999, and 8000-10999 each contain the average performance of both algorithms over 12 BKBs that fit into the CC subgroup. The actual averages for the low, medium and high CC metrics are 3918.3, 6135.6 and 9557.66.

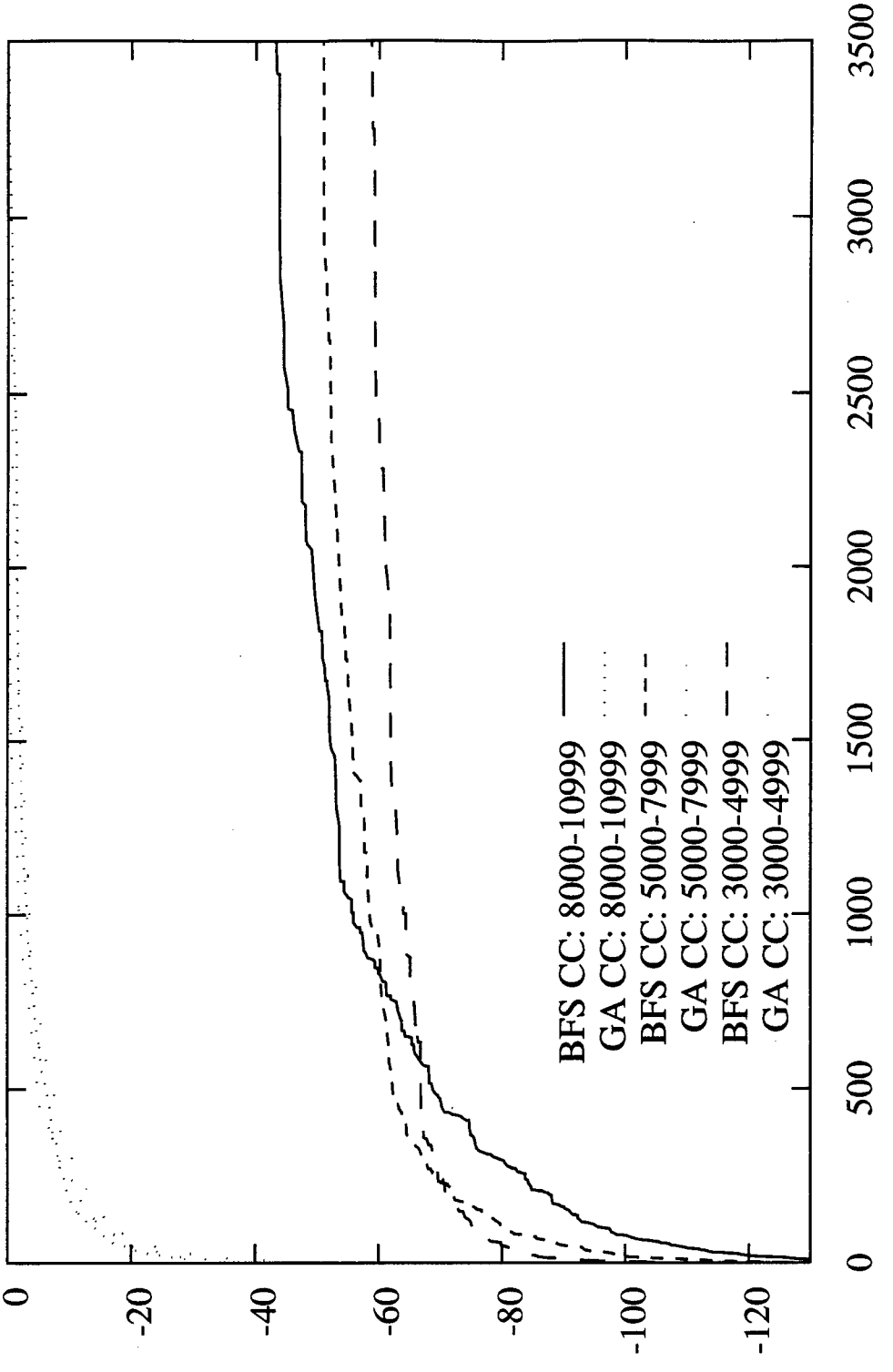

Figure 5.14 CPP: Effects of Cyclomatic Complexity on Performance of Algorithms

Close views of the performance of the BFS and the GA can be seen in figures 5.15 and 5.16. When examining these figures, observe that the x and y axis scales are different for each figure.

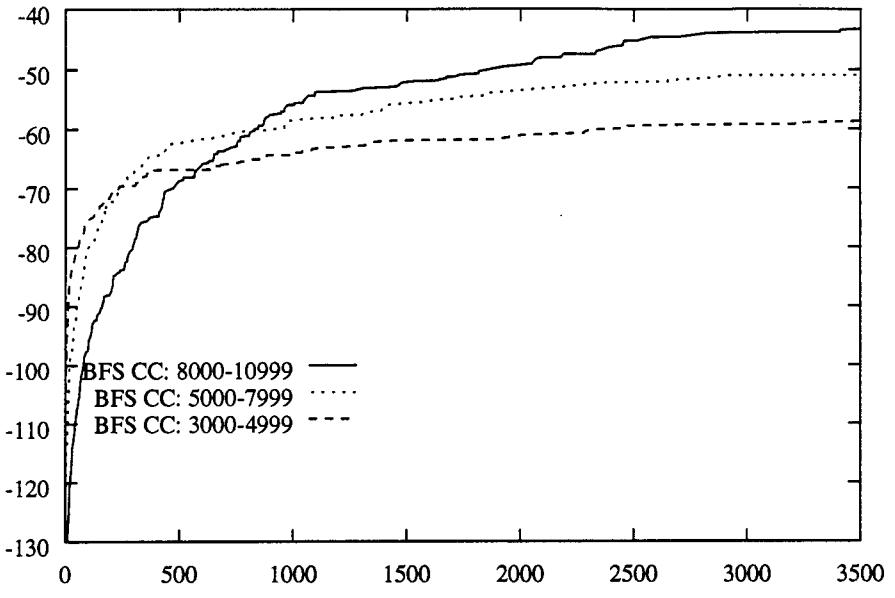

Figure 5.15 CPP: Effects of Cyclomatic Complexity on Performance of BFS (closeup view)

In Figure 5.15 we see that there effects of cyclomatic complexity on the BFS are substantial, yet in Figure 5.16, we see that CC has little effect on the performance of the GA. As expected, CC, a topological characteristic, has a greater effect on performance of a topological reasoning algorithm than on a probabilistic algorithm.

The utility graph presented in figure 5.17 depicts the time needed to achieve a constant log solution quality of -60.0. This was the maximum achievable average solution quality for the low CC group. The utility curves demonstrate that the GA could always achieve a solution quality of -60.0 before the BFS. The BFS took less time as the cyclomatic complexity increased, while the GA held a constant time to reach that quality. Notice that there are no critical points in this graph. When there are no critical points on a particular UC, one algorithm always outperforms the other algorithm for the range of the characteristic expressed and the solution quality desired.

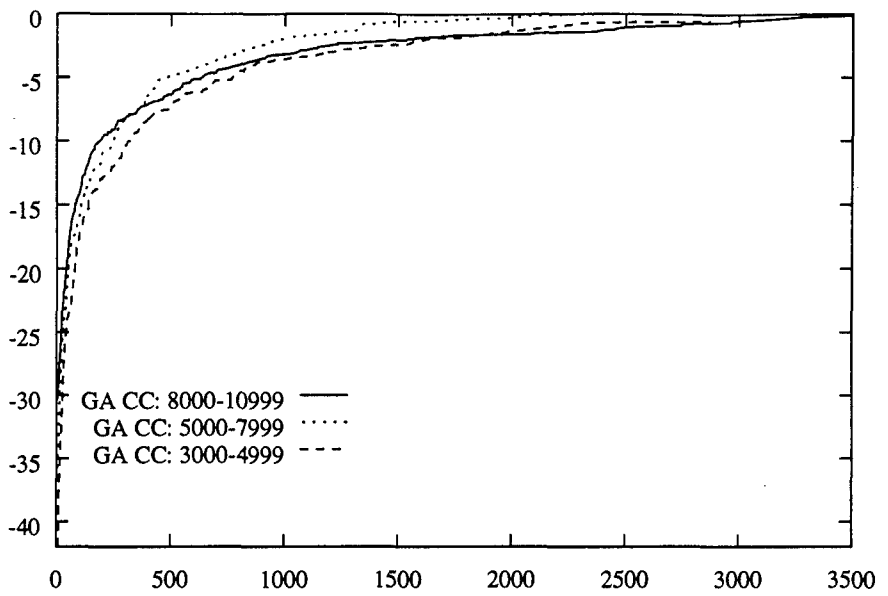

Figure 5.16 CPP: Effects of Cyclomatic Complexity on Performance of GA (closeup view)

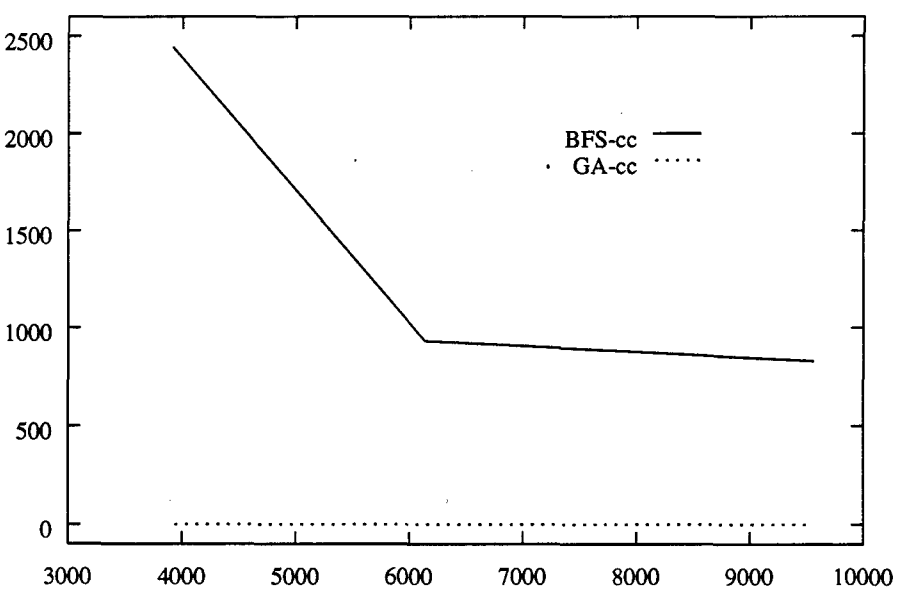

Figure 5.17 UC: Cyclomatic Complexity vs. Time

5.5 Skewness

The effect of average skewness on the performance of algorithms was examined for several BKBs. The results are presented in this section. Figure 5.18 presents an overview of the effects of skewness, while Figures 5.19 and 5.20 present closeup views of the effects of skewness on the BFS and the GA.

Figure 5.21 is a utility curve generated for a desired log solution quality of -13.5. Notice the critical point at a skewness of 0.94. Decisions made on the basis of this UC would indicate that if skewness was below 0.94, we would choose to use a GA, while if the skewness was above 0.94, the BFS would reach the desired solution quality the fastest.

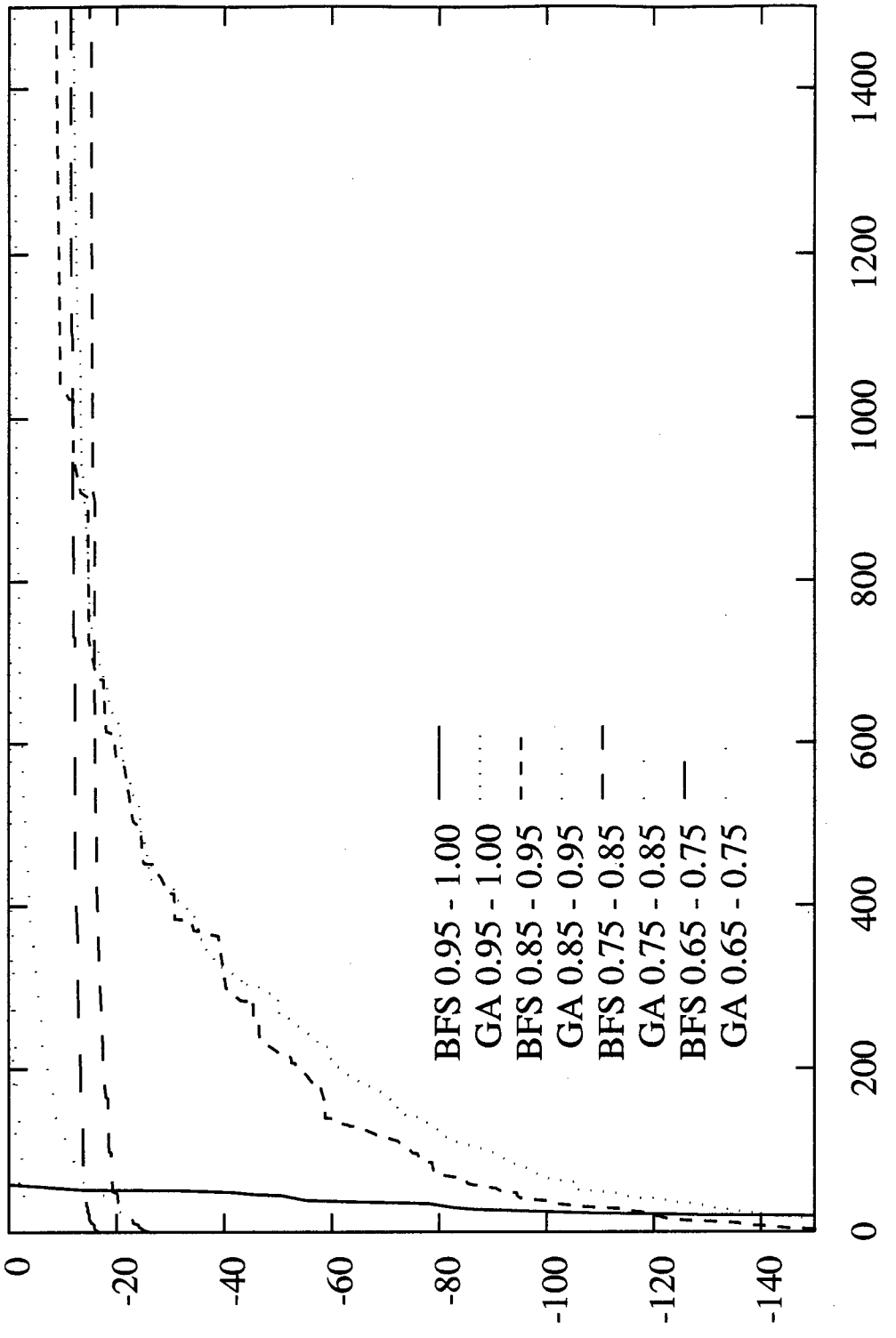

Figure 5.18 CPP: Effects of Skewness on Performance of Algorithms

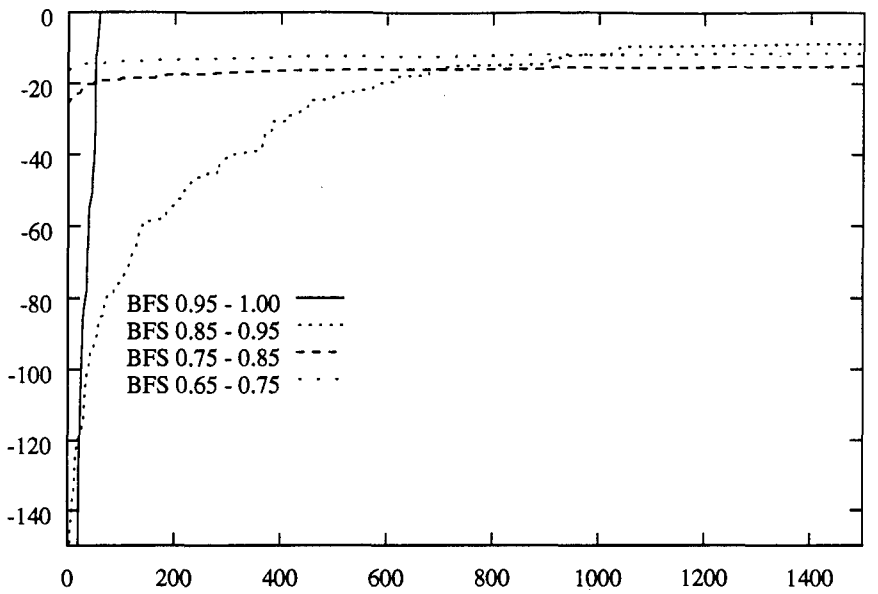

Figure 5.19 CPP: Effects of Skewness on Performance of BFS

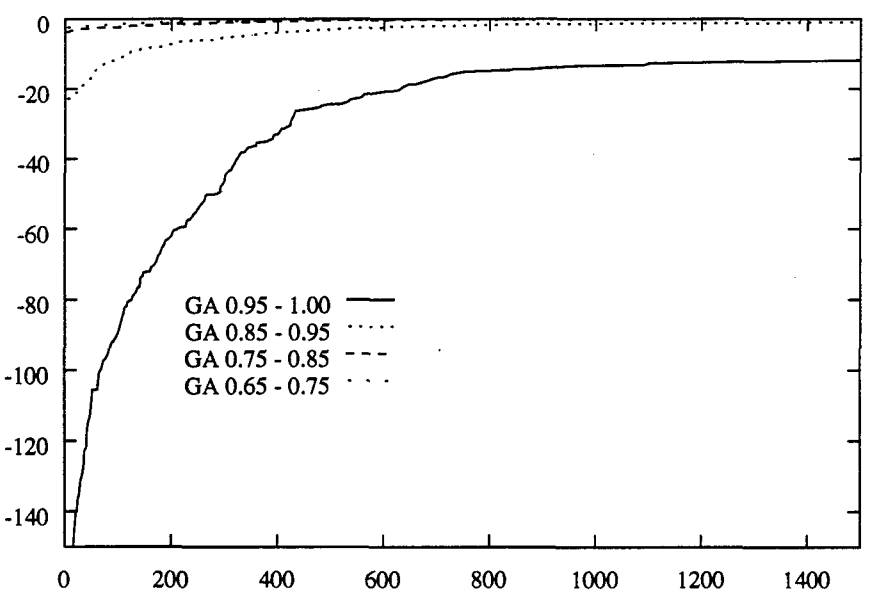

Figure 5.20 CPP: Effects of Skewness on Performance of GA

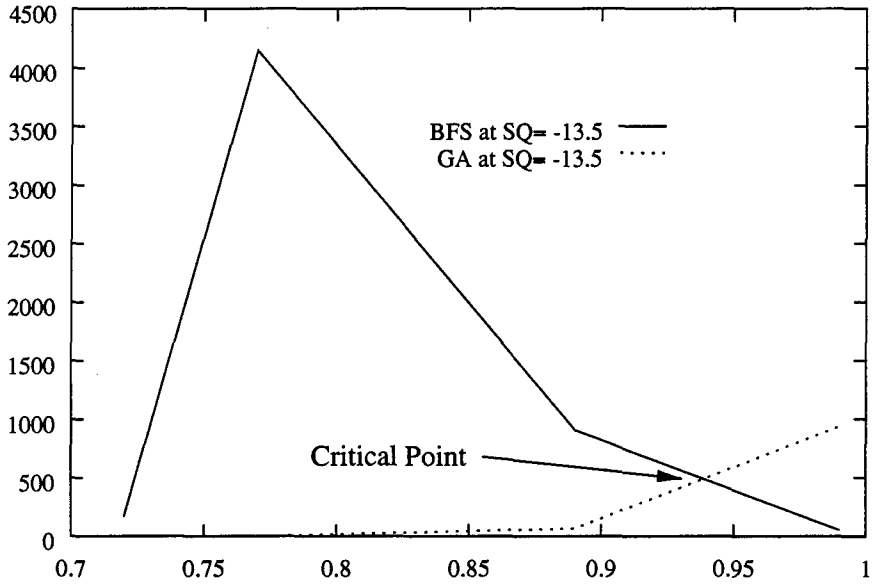

Figure 5.21 UC: Effects of Skewness on Performance of GA

VI. Conclusions and Future Research

The last chapter presented the experimental results of the effects of domain characteristics on the performance of algorithms. This chapter presents a discussion of those results and their applicability towards performance prediction and selection under constrained resources. This chapter also elaborates the benefits and limitations of the approach and concludes with a description of future research in this area.

6.1 Discussion of Results

The experimental results presented in the last chapter comprised the first three steps in the five step strategy building process outlined in Section 4.1:

1. Find a set of characteristics of the network that are *differentiating characteristics* (DC).
2. Collect networks that exhibit a variety of values over all of the selected DCs. This collection is called the *test suite.*
3. Gather data on the performance of algorithms when the DCs are varied.
4. Generate the relation that can be used to predict performance of the algorithms.
5. Test the relation.

The first three steps in this process showed that performance of algorithms is related to the characteristics of their input domain. The fourth step promoted the concept of the *critical point,* the value of a characteristic where the utility of using either algorithm was the same.

Chapter V simulated the fourth step in the strategy building process for each characteristic on an individual basis by explaining which algorithm should be chosen for various values of that characteristic.

For the BFS and GA, we found that RV count, skewness and extremeness each had discernible critical points in their utility graphs. These characteristics are applicable in the decision of which algorithm to use for best performance. The other characteristics,

cyclomatic complexity, and completeness are useful in conceptualizing the behavior of an algorithm under various values of the characteristic.

6.2 Limitations of this approach

The benefits of this approach are that it is easy to understand, and also provides very clear indications of which algorithm should be used based on the critical points in the utility curves. However, this approach has several limitations which may curtail its widespread use.

The approach is time intensive. It requires a suite of domain instances to be run on each algorithm so that performance data can be gathered and the critical points can be calibrated. The research performed for this thesis took over 680 hours of CPU time on Sun Ultra-Sparcs to generate performance data for two algorithms. Few organizations can afford an entire month of CPU time for calibration purposes. Furthermore, if any of the algorithms are modified, the entire test suite must be re-run to generate performance curves. Then utility curves must be generated, and new set of critical points must be calibrated.

The approach is experimental in nature. It relies on a statistical analysis of a large sample space of data to predict future performance of algorithms. While statistical analysis can uncover trends, no number of experiments is sufficient to prove a theory.

No number of experiments is adequate to prove a theory, however, statistical analysis of large samples can show trends. This approach relies on a statistical analysis of a large sample space of performance curves, further increasing the preprocessing needed for accurate calibration of critical points.

The approach treats the algorithms as black boxes. By predicting performance solely on the domain characteristics, we cannot hope to comprehend the idiosyncrasies of the algorithm. There may be a niche of "abnormal" performance that is caused by the internal workings of the algorithm, and if our test suite is not comprehensive enough, we may miss that anomaly. Even if we locate the anomaly with this approach, we may not have a method to cope with the abnormal performance when calibrating the critical points.

6.3 Future Research

There were several focuses of this research effort. They are listed below in order of importance.

- Demonstrate the validity of predicting conditional performance profiles using domain characteristics as discriminators.
- Develop algorithms for inference over BKBs
- Extend the awareness of the benefits and limitations of using the Bayesian Knowledge Base structures for inference.

Chapter III discussed the benefits and limitations of the Bayesian Knowledge Base as a structure for inference. We showed that while the BKB's structure eases the knowledge acquisition process, it puts the burden of handling incompleteness on the inference engine. As shown in Figures 5.10, 5.11, and 5.12, the more incompleteness in a BKB, the worse the performance of a topological algorithm, even though there are fewer arcs and supports remaining in the BKB. This problem presents a large area of available future research: coping with incompleteness. The research can be decomposed into two subareas: Coping with incompleteness before the BKB is inferenced over (in the knowledge acquisition or verification and validation phases,) and coping with incompleteness during inference.

This thesis made several assumptions about coping with incompleteness during the development of inference techniques (Chapter IV):

- A complete solution (even a zero probability complete solution) is always better than an incomplete solution
- All incomplete solutions can be ordered by their *level* of incompleteness. Thus, the higher the level of incompleteness in a given solution, the worse that solution is.

These assumptions may need to be challenged in future research. By modifying or eliminating them, new policies for dealing with incompleteness during inference could allow more efficient inference algorithms.

While this research shows that predicting performance based on domain characteristics is plausible, there are many limitations in the approach. One of the assumptions

made for this research is that the randomly generated BKBs represented a large range of potential real world BKBs. Currently, no real world validated BKBs exist, and the time required to develop and validate a real world knowledge domain precluded generating the many BKBs that would be needed to perform the experiments exhibited in the last chapter. Future research, however, could validate the results of the experiments seen in the last chapter using real world BKBs.

Another topic that needs to be explored is the effect of the *algorithm* on the performance curves. In this research, the algorithm was treated as a black box, and a model of the BKB was used as input to the black box. With this approach, the only method for predicting performance was through analysis of experimental results, a time consuming process with limited applicability. If, however, the algorithm as well as the data could be modelled, the "experiments" could be replaced by mathematical simulations. The output of the simulation would be the performance predictions of the various algorithms and a choice of which algorithm to run.

This research focused on predicting and choosing which single algorithm would demonstrate optimal performance for a given problem instance. Future research should explore which set of algorithms to run based on how they act individually and how they act in concert. With all of the possible combinations of algorithms, the number of experiments required to determine the set of algorithms with optimal performance would grow combinatorically. Therefore, this area of research cannot be undertaken until a method for developing a model of an algorithm is complete. Once the model of each algorithm is valid, future researchers can begin development of a model for how anytime algorithms interact. Then the optimal set of algorithms can be chosen for inference under constrained resources.

At a higher level, this research is applicable for predicting the performance of any algorithm that has a performance profile that varies with changes the characteristics of its input (the method is not applicable if the algorithms performance is independent of the input characteristics). With the capability of predicting the performance of multiple algorithms, the methods presented in this thesis can be used to determine which algorithm will perform the best for a specific problem.

Bibliography

1. Banks, Darwin O. *Acquiring consistent Knowledge for Bayesian Forests*. MS thesis, Air Force Institute of Technology, March 1995.

2. Boddy, Mark and Thomas Dean. *Solving Time-Dependent Planning Problems*. Technical Report CS-89-03, Providence RI: Brown University, February 1989.

3. Borghetti, Brett J., et al. "Inferencing Over Incomplete Solution Spaces with Genetic Algorithms for Probabilistic Reasoning." *Online Proceedings of the 1996 Midwest artificial Intelligence and Cognitive Science Conference.*, edited by M. Gasser. URL. http://www.cs.indiana.edu/event/maics96/Proceedings/Port.html, April 1996.

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

5. Cooper, Gregory F. "The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks," *Artificial Intelligence*, *42*:393–405 (1990).

6. Dagum, Paul and Michael Luby. "Approximating probabilistic inference in Bayesian belief networks is NP-hard," *Artificial Intelligence*, *60*:141–153 (1993).

7. Dean, Thomas, et al. "Deliberation Scheduling for Time-Critical Sequential Decision Making." *Proceedings of the Ninth Conference on Uncertanty in Artificial Intelligence*. 1993.

8. Dean, Thomas, et al. "Planning Under Time Constraints in Stochastic Domains," *Artificial Intelligence*, *76*:35–74 (1995).

9. Eric J. Horvitz, Gregory F. Cooper, David E. Heckerman. "Reflection and action Under Scarce Resources: Theoretical Principles and Emperical Study." *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*. 1121–1127. San Mateo, CA: Morgan Kaufmann, August 1989.

10. Goldberg, David E. *Genetic Algorithms in Search, Optimization & Machine Learning*. Reading, MA: Addison-Wesley, 1989.

11. Horvitz, Eric J. "Reasoning about Beliefs and Actions under Computational Resource Constaraints." *Proceedings of the Third Workshop on Uncertainty in Artificial Intelligence*. 429–444. Mountain View, CA: AAAI and Association for Uncertainty in Artificial Intelligence,, July 1987.

12. Horvitz, Eric J. "Reasoning Uder Varying and Uncertain Resource Constraints." *Proceedings of the Sevent National Conference on Artifiacial Intelligence*. 111–116. San Mateo, CA: Morgan Kaufmann, August 1988.

13. 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.

14. McCabe, Thomas J. "A Complexity Measure," *IEEE Transactions on Software Engineering*, *SE-2*(4):243–245 (October 1976).

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

16. Pearl, Judea. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco, CA: Morgan Kaufmann Publishers, Inc, 1988.

17. Poole, David. "Average-case analysis of a search algorithm for estimating prior and posterior probabilities in Bayesian networks with extreme probabilities." *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*. 606–612. August 1993.

18. 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.

19. Santos, Jr., Eugene. *A Fully Integrated Probabilistic Framework for Expert Systems Development*. Technical Report, Department of Electrical and Computer Engineering, Air Force Institute of Technology, 1993.

20. Santos, Jr., Eugene. *A Distributed Anytime System With Co-operative Task Management For Reasoning Under Uncertainty : The PESKI Engine*. Technical Report, Department of Electrical and Computer Engineering, Air Force Institute of Technology, 1995.

21. 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.

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

23. Wolpert, David H. and William G. Macready. *No Free Lunch Theorems for Search*. Technical Report 95-02-010, 1399 Hyde Park Rd. Santa Fe, NM, 87501, USA: The Santa Fe Institute, February 1995.

24. Zilberstein, Shlomo and Stuart Russell. "Optimal Composition of Real-Time Systems," *Artificial Intelligence*, *82*:181–213 (1996).

Vita

Brett Borghetti [redacted] He graduated from Worcester Polytechnic Institute in 1992 with a Bachelor of Science degree in Electrical Engineering. Brett also received his commission in May 1992 through the Reserve Officer Training Corps. He entered active duty service in the Air Force in January 1993 and worked for two and a half years at the Training System Product Group , Wright Patterson AFB, Ohio. He began pursuing a Master of Computer Systems degree at the Air Force Institute of Technology in May 1995.

[redacted]

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1996	Master's Thesis

4. TITLE AND SUBTITLE
Inference Algorithm Performance
and Selection Under Constrained Resources

5. FUNDING NUMBERS

6. AUTHOR(S)
Brett J. Borghetti

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
2950 P Street
WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
/AFIT/GCS/ENG/96D-05

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Dr Abraham Waksman
AFOSR/NM
110 Duncan Ave
Bolling AFB, DC 20332
202-404-7496

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)*

Knowing that reasoning over probabilistic networks is, in general, NP-hard, and that most reasoning environments have limited resources, we need to select algorithms that can solve a given problem as fast as possible. This thesis presents a method for predicting the relative performance of reasoning algorithms based on the domain characteristics of the target knowledge structure. Armed with this knowledge, the research shows how to choose the best algorithm to solve the problem. The effects of incompleteness of the knowledge base at the time of inference is explored, and requirements for reasoning over incompleteness are defined. Two algorithms for reasoning over incomplete knowledge are developed: a genetic algorithm and a best first search. Empirical results indicate that it is possible to predict, based on domain characteristics, which of these algorithms will have better performance on a given problem.

14. SUBJECT TERMS
artificial intelligence, reasoning, inference, distributed processing,
uncertainty, probabilistic networks, expert systems, Bayesian Knowledge Base

15. NUMBER OF PAGES
71

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