ALGORITHM SELECTION FOR SORTING AND PROBABILISTIC INFERENCE: A MACHINE LEARNING-BASED APPROACH

by

HAIPENG GUO

M.S., Beijing University of Aeronautics and Astronautics, 1996

A DISSERTATION

Submitted in partial fulfillment of the

requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences

College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2003

COPYRIGHT

ALGORITHM SELECTION FOR SORTING AND PROBABILISTIC INFERENCE:
A MACHINE LEARNING-BASED APPROACH

HAIPENG GUO

2003

ABSTRACT

The *algorithm selection problem* aims at selecting the best algorithm for a given computational problem instance according to some characteristics of the instance. In this dissertation, we first introduce some results from theoretical investigation of the algorithm selection problem. We show, by Rice's theorem, the nonexistence of an automatic algorithm selection program based only on the description of the input instance and the competing algorithms. We also describe an abstract theoretical framework of instance hardness and algorithm performance based on Kolmogorov complexity to show that algorithm selection for search is also incomputable. Driven by the theoretical results, we propose a machine learning-based inductive approach using experimental algorithmic methods and machine learning techniques to solve the algorithm selection problem.

Experimentally, we have applied the proposed methodology to algorithm selection for sorting and the MPE problem. In sorting, instances with an existing order are easier for some algorithms. We have studied different presortedness measures, designed algorithms to generate permutations with a specified existing order uniformly at random, and applied various learning algorithms to induce sorting algorithm selection models from runtime experimental results. In the MPE problem, the instance characteristics we have studied include size and topological type of the network, network connectedness, skewness of the distributions in Conditional Probability Tables (CPTs), and the proportion and distribution of evidence variables. The MPE algorithms considered include an exact algorithm (clique-tree propagation), two stochastic sampling algorithms (MCMC Gibbs sampling and importance forward sampling), two search-based algorithms (multi-restart hill-climbing and tabu search), and one hybrid algorithm combining both sampling and search (ant colony optimization).

Another major contribution of this dissertation is the discovery of multifractal properties of the joint probability distributions of Bayesian networks. With sufficient asymmetry in individual prior and conditional probability distributions, the joint distribution is not only highly skewed, but it also has clusters of high-probability instantiations at all scales. We present a two phase hybrid random sampling and search algorithm to solve the MPE problem exploiting this clustering property. Since the MPE problem (decision version) is NP-complete, the multifractal meta-heuristic can be applied to solve other NP-hard combinatorial optimization problems as well.

Preface

Hereby I would like to record the journey of my dissertation research that started 3 and a half years ago when I joined Dr. Hsu's KDD group at K-State. I remember the first paper he handed to me was Charniak's "Bayesian networks without tears" [Cha91]. I found my interests in uncertain reasoning using Bayesian networks soon after I read the paper and implemented a Bayesian network learning algorithm, K2. When the time came for me to choose a dissertation topic, I knew that my research was going to be in the area of artificial intelligence and Bayesian networks. Since another group in our department was doing some real-time research, I thought real-time artificial intelligence might be a good topic to work on. After consulting with Dr. Hsu and other committee members, I finally set my topic to real-time AI.

At that time some of my committee members had managed to open a seminar class on "Topics in Real-time Artificial Intelligence" in the spring semester of 2001. At the class we were requested to present a related paper and the one I chosen was "On a Distributed Anytime Architecture for Probabilistic Reasoning" by Santos [SSW95]. After presenting that paper, I became more interested in the study of real-time Bayesian network inference. I started reviewing various Bayesian network inference algorithms and I realized that under real-time constraints the selection of proper algorithms became very crucial, because different algorithms' performances vary differently as properties of the input instance change. Also I noticed that in real world problem solving, one main reason that experts in any domain are called experts is because they are all very good at selecting the best problem-solving technique quickly. Therefore I focused my attention on the algorithm selection problem for Bayesian network inference.

Thus, I formed the goal of building an algorithm selection system that can act as an algorithm selection expert for selecting the best algorithm for a particular input instance so as to gain the best overall performance. Somehow I formed the idea of using another Bayesian network sitting at the meta-level as the algorithm selection reasoner to do the job since a Bayesian networks are graphical models for learning and reasoning in probabilistic expert systems. It has all components of an intelligent system: representation, learning, and inference. Many thanks to Dr. Hsu for bringing me to IJCAI-2001 and UAI-2001 in Seattle so that I got a chance to directly talk to these first-class researchers in the field. At UAI-2001 I was particularly impressed and encouraged by Eric Horvitz's paper [HRG$^+$01] on applying a Bayesian approach to tackling hard computational problems. Since the idea was basically the same as the one I came up with. It used a Bayesian network as the meta-reasoner to monitor and control the processes of hard problem solving process. I have forgot exactly what question I asked Eric after his talk, but I do remember I became more confident on my research direction afterwards.

I started more literature survey on the related fields as soon as I came back from Seattle. First I noticed that not all input instances are created equal. Some instances occur in real world applications more frequently, while others exists only in theoretical world. Correspondingly, these so-called "real-world" instances should be treated specially. At the same time I started reading papers on "No Free Lunch Theorems", which basically state that without any structural assumptions on a search or optimization problem, no algorithm can perform better on average than blind search (thus than any other algorithms). Therefore in order to solve any problem better, we need an adapted algorithm, an algorithm that is able to take into consideration the structural specificities of the problem. All of these had motivated me to study and build the algorithm selection meta-reasoner, which can assign the best algorithm to the input instance by examining and reasoning the instance's structural property.

The meta-reasoner has to gain his knowledge from somewhere. There are usual two ways to do it: analytically (deductive) and experimentally (inductive). Investigations into the theoretical direction led me to the fields of computational complexity, computability, Kolmogorov complexity, and GA-hardness. These formed the basis of the theoretical aspects of my dissertation research. Discussions with Dr. Howell have clarified many of my misconceptions. The pure analytical approach to algorithm selection was proven to be a dead end when we realized that by Rice's theorem it is impossible to have an automatic checker (a program or a Turing machine) that can read two algorithms and decide which is better, i.e., the problem is undecidable in general. This is easy to understand because Turing has showed [Tur36] that we can not even decide if an algorithm actually halts (the halting problem). Then I turned to the inductive direction without much doubt.

At UAI02 at Edmonton, I managed to co-chair a workshop on real-time decision support and diagnosis systems with Dr. Hsu, Horvitz and Santos. At the workshop I had chance to talk to Fabio Cozman on my research. I also had a good talk with Michael Horsch (at Saskatchewan) on No Free Lunch Theorems and some other stuff. These all enhanced my confidence on experimental approaches. Later Fabio and his graduate student Jaime made an open source Bayesian network generator that proved very helpful to my experiments on Bayesian network inference. I also presented my thesis proposal at AAAI02's Doctoral Consortium. Leslie P. Kaelbling (at MIT) was my assigned tutor. After I presented my talk she asked me: "Why do you use a Bayesian network? A decision tree might work just as well." And I agreed with her. Later the focus of my thesis research had been switched a little from the original *Bayesian approach* to *machine learning based approach* in which some other models (Decision tree and naive Bayes) were also studied and compared with Bayesian network. I also met James Park (at UCLA) at Edmonton. His work on MPE and MAP inspired me a lot. Later he even allowed me to share his source code, although I have never been able to make it executable.

The multifractal analysis part of this research was mainly inspired by Marek J. Druzdzel's paper on the skewness of the joint probability space of Bayesian networks. To verify his results, I generated the joint probability and plotted it in Excel. As I was looking at the shape of the joint distribution, it reminded me of the concept of fractal. This led to the discovery of the multifractal property of the joint space and its usefulness as a meta-heuristic to design algorithm for finding the most probable explanation. Druzdzel's paper [Dru94] also led me to the investigation of CPT skewness as a MPE instance feature to help select the best algorithm. It turned out that skewness is among one of the most important features in differentiating the algorithm performance space.

Coming back from Edmonton, I had a clearer picture in my mind of what I should do. The next two semesters have been a busy time for me. Implementing algorithms, running experiments, collecting and analyzing data, and finally, writing the thesis. The first draft was finished in early May, 2003.

My central goal has been to build the algorithm selection meta-reasoner for real-time Bayesian network inference. This effort has culminated in the development of a machine learning-based inductive methodology to build general, meta-level intelligent algorithm selection systems. However, many problems still remain open. For example, there could be many types of meta-level reasoners. Do we need a meta-meta-level reasoner to reason about the selection of meta-level reasoner? How much efforts should we put into the meta-level reasoning? When will thinking too much be a problem? I have grown to appreciate the true complexity and challenge of building artificial intelligent system like this task. Now I find that it is exactly that challenge which stimulates me to continue this research.

Now I have finished recording this long story of my Ph.D. research. I hope someday somebody, who has to go through this process as I have been doing, will find it worth reading.

Acknowledgements

First of all, I would like to thank my major professor, Dr. William H. Hsu, for being a continual source of inspiration and encouragement. I thank him for introducing me into this field and guiding me through my Ph.D. study. I also appreciate very much the research freedom I have enjoyed under his supervision. This work would not be possible without his intelligent and financial support.

Next, I would like to acknowledge the many insightful comments provided by my committee members: Dr. Mitchell L. Neilsen, Dr. Gurdip Singh, Dr. Shing I Chang, and Dr. Kenneth Shultis. I would also like to thank Dr. Rodney R. Howell for all the helpful discussions we had when I took an independent study with him on advanced topics in computational complexity. Many thanks to Dr. Marc Kesseboehmer (at Berlin) and Dr. Rudolf Riedi (at Rice Univ.) for proofreading the multifractal part of my dissertation. I also want to thank Dr. L. T. Fan for his comments on my multifractal study and thank Dr. Sanjoy Das for his help on ant algorithm implementations. I am also indebted to Leslie P. Kaelbling (at MIT) for her valuable comments on my dissertation proposal at AAAI-2002's Doctoral Consortium workshop.

During the past two years, it has been my good fortune to have had the opportunity to discuss these ideas with many of the authors to which I refer in the dissertation. Many inspirations of my work came from reading their papers. Some tools developed by these authors have been a great help to my experiments. In particular, I am indebted to the following researchers: Dr. Eric Horvitz (at Microsoft Research), Dr. Marek J. Druzdzel (at the University of Pittsburgh), Dr. Heikki Mannila (at Helsinki), Dr. Eugene Santos Jr. (at the University of Connecticut), Dr.

Fabio Cozman (at Brazil), Dr. Bart Selman (at Cornell), James D. Park (at UCLA), Dr. Ole J. Mengshoel, Jesse Reichler (at UIUC), Jaime S. Ide (at Brazil), and the great Weka development team at the university of Waikato, New Zealand.

I also want to thank the following friends, BNJ developing team members, and system administrators at KDD group for the wonderful experimental environment and many insightful comments they provided: Cecil P. Schmidt, Laura M. Kruse, Benjamin B. Perry, Jessica R. Perry, Julie A. Thornton, Roby Joehanes , Prashanth R. Boddhireddy, Eric Matson, James W. Plummer, and Jason J. Kieffaber.

Finally and foremost, I want to thank my parents for always believing in me and for encouraging me to take chances and to pursue my dream.

DEDICATION

TO MY PARENTS,

献给我的父母,

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Given a computational problem, there usually exist many different algorithms to solve it exactly or approximately. Different algorithms often perform better on different classes of problem instances. The *algorithm selection problem* [Ric76] asks the following question: *which algorithm should we select to solve this instance?*

The algorithm selection problem is important because of both theoretical and practical reasons. Computer scientists seek a better theoretical understanding to problem instance hardness and algorithm performance so as to deliver better algorithms for the given computational task. In practice, it helps us gain more efficient computations to solve the problem in hand. This is especially crucial for some real-time applications that are under the pressure of some hard or soft computational deadlines.

This dissertation mainly presents a machine learning-based approach to solve the above-mentioned problem and applies it to sorting and finding the Most Probable Explanation (MPE), which is a probabilistic inference problem. In this chapter, we give a brief introduction on motivations and directions of this work. We also describe our thesis and the organization of this dissertation.

1.1 Introduction and Motivations

1.1.1 Analytical versus Experimental Approaches

Algorithm comparison and algorithm selection are central topics in *computational complexity theory*, which studies the amount of resources needed in solving computational problems. The field has traditionally been divided into *algorithm analysis*, *problem complexity*, and *complexity classes*. Algorithm analysis studies the amount of resources an algorithm consumes. Problem complexity studies the amount of resources needed to solve a computational problem. The theory of complexity classes studies the classification of problems according to their intrinsic computational complexity.

The amount of resources needed by an algorithm or a problem is usually measured by functions of the length (size) of input instances, i.e., the *time complexity functions*. We know in practice that instances of the same size may require different amounts of resources because of their differences in some other characteristics. For example, an almost ordered permutation can be sorted by Insertion Sort algorithm in linear time but it takes quadratic time for Insertion Sort to sort a totally unordered permutation. Hence we need to study problem instance characteristics other than just size in order to select the best algorithm.

Classical computational complexity theory mainly relies on analytical approaches such as *worst-case analysis* and *average-case analysis*. Worst-case analysis often works well and provides a good basis for algorithm selection. However, there are still some cases where it fails. Consider Quick Sort, for example. By worst-case analysis, Quick Sort has a quadratic time complexity, yet it is fast in practice in most cases. Another example is the Simplex algorithm for solving linear programming problem. Its worst-case time complexity is exponential, but it performs extremely well in real world applications. Also, worst-case analysis treats all instances of the same size collectively as a whole although they may be very different in terms of features other than the problem size. Average-case analysis is often difficult to apply because it requires a reasonable estimate of all instances' probabilities. In most situations, this is simply infeasible.

To distinguish the varying resource requirements of instances that have the same size but are different in other features, we need to move from problem complexity to *instance complexity*. There are at least three ways to deal with this issue. The first one defines instance complexity of decision problems using Kolmogorov complexity [OKSW94]; i.e., it is defined as the size of the shortest program to solve the decision problem. The second one defines instance complexity by restricting the allowed algorithms. For example, "Instance p_1 is harder than p_2 if it takes more time for a specified algorithm A to solve p_1". The third method considers *instance classes* instead of single instances [Man85]. It defines a subproblem of the original problem by some intuitive criteria of instance easiness (or hardness) and then studies the worst-case complexity of the subproblem. Algorithms are designed and analyzed on these subproblems, with resource requirements increasing smoothly when moving to larger subproblems. The resulting algorithm is called optimal to the instance hardness measures.

Although the first method using Kolmogorov complexity has produced some interesting results, the results are mainly along the line of complexity classes and do not help much on practical algorithm selection. The second method is not very attractive because it depends on a particular algorithm. The third method has made a lot of progresses in designing adaptive sorting algorithms that are optimal to many measures of the existing orders in a sorting instance. Because of its analytical nature, this method is easy for simple problems like sorting, but hard to be applied to arbitrary NP-hard optimization problems, which are the most important ones in practice. Furthermore, although the third method provides a way to design adaptive algorithms that are optimal for some measures, it is usually impossible to design such an algorithm that are optimal for all measures. There is also an algorithm selection problem for different adaptive algorithms. This dissertation follows the third method of considering instance classes instead of single instances, but we rely more on experimental approaches rather than analytical ones.

1.1.2 Problem Instance Characteristics and Algorithm Performance

This research was partly motivated by the observation that some easy-to-compute problem features can be used as good indicators of some algorithm's performance on hard instances. This knowledge can be utilized to help select the best algorithm in order to gain more efficient overall computations. Consider the sorting example again. It is well known that if the permutation is nearly sorted, then the Insertion Sort algorithm can sort it in linear time although the algorithm's worst-case time complexity is $O(n^2)$ and the computational complexity of sorting is $O(n \log n)$. In the community of NP-hard optimization problem-solving, researchers have long noticed that the NP-complete result is just a worst-case result; i.e., not all instances are equally hard [CKT91]. Algorithms that exploit some features of the input instances can perform on the particular class of instances better than the worst-case scenario. In light of this, two of the main directions of this research are to study different instance features in terms of their goodness as a predictive measure for some algorithm's performance and to investigate the relationships between different instance features and different algorithms' performance. We aim to develop a unified machine learning-based methodology to automate the process of knowledge discovery and reasoning with regard to solving the algorithm selection problem.

1.1.3 Machine Learning Models for Algorithm Selection

Another motivation of this work came from the inspiration of automating and mimicking human expert's algorithm selection process. In many real world situations, algorithm selection is done by hand by some experts who have a good theoretical understanding to the computational complexities of various algorithms and are very familiar with their runtime behaviors. The automation of the expert's algorithm selection process thus has two aspects: the analytical aspect and the experimental aspect. Theoretically, the first aspect is hard to be automated and compiled into a program. We will examine this in detail in chapter 3. Comparatively, automating the experimental aspect is more feasible because of the progresses that have been made in experimental algorithmic [Joh02], machine learning [Mit97], and uncertain reasoning techniques [Pea88].

The difficulty of automatic algorithm selection is largely due to the uncertainty in the input problem space, the lack of understanding to the working mechanism of the algorithm space, and the uncertain factors of implementations and run-time environments. This is especially true for NP-hard problems and complex, randomized algorithms. From the viewpoint of expert systems and machine learning, the automatic algorithm selection system acts as an "intelligent meta-level reasoner" that is able to learn the uncertain knowledge of algorithm selection from its past experiences and use the learned knowledge (models) to reason on algorithm selection for the input instance in order to make the right decision.

1.2 Thesis Statement

In this dissertation, we first develop some theoretical results on automatic algorithm selection. We show, by Rice's theorem, that there does not exist a program to automatically select the best algorithm based only on the descriptions of the input instance and these algorithms. We also present a general, abstract framework of problem hardness and algorithm performance for search based on Kolmogorov complexity in order to show the difficulty of analytically deriving the algorithm's performance from the input instance. Driven by these theoretical results, we propose a machine learning-based approach to build automatic algorithm selection systems using experimental methods and machine learning techniques.

We then choose two problems, *sorting* and *finding the MPE* as our test cases. These are chosen as representatives of two important complexity classes: P and NP-complete. Sorting is an easy but fundamental problem to computer science in general. The MPE problem is interesting because it is an important inference problem in Artificial Intelligence (AI) and probabilistic reasoning using Bayesian networks. The decision version of the MPE problem is also NP-complete; many tasks of interests, for example SAT and Vertex Covering, can be converted to the MPE problem [Coo90, Shi94]. In this sense, the MPE problem is representative of the NP-hard combinatorial optimization problems.

For both problems, we apply the following procedures:

1. Identify a list of candidate algorithms for solving the problem.

2. Identify a list of feasible instance characteristics using domain knowledge.

3. Generate a representative set of test instances with different characteristic values settings uniformly at random.

4. Run the candidate algorithms on these elaborately-designed instances and collect the performance data to produce the training datasets.

5. Apply machine learning techniques on training data to induce a predictive algorithm selection model (decision tree, naive Bayes classifier, or a Bayesian network) out of it.

6. For a new instance, analyze (quickly) its characteristic values and use the learned models to infer (classification) the best algorithm to solve it.

In studying the MPE problem, we have also discovered an important *multifractal property* of the Joint Probability Distributions (JPDs) of Bayesian networks. Specifically, with sufficient asymmetry in individual prior and conditional probability distributions, the joint distribution is not only highly skewed, but it also has clusters of high-probability instantiations at all scales. Based on this clustering property, we have designed a two-phase hybrid Sampling-And-Search algorithm for solving the MPE problem in Bayesian networks. Since finding the MPE is NP-complete, we expect that this multifractal meta-heuristic can be applied to solving other NP-hard combinatorial optimization problems as well.

In summary, the thesis of this dissertation is three-fold:

1. Theoretically, automatic algorithm selection is impossible if only based on the description of the input instance and the algorithm.

2. A machine learning-based approach using experimental methods and machine learning techniques can be used to build automatic algorithm selection systems that can select the most suitable algorithm according to the input instance's characteristics.

3. The JPDs of Bayesian networks with skewed CPTs have some multifractal properties and this can be used as a meta-heuristic to design new search algorithms for solving NP-hard optimization problems.

1.3 Organization

The organization of this dissertation is as follows:

Chapter 1 gives a brief introduction of the main themes of this dissertation. The research problem is clarified, main motivations and directions are described, the three-fold thesis is stated, and the organization of the dissertation is presented.

In chapter 2, we introduce concepts and background materials from related areas and their relationship with this research. These areas include computational complexity theory, computability theory, algorithmic information theory, Kolmogorov complexity, main issues in experimental algorithmics, various machine learning techniques, probabilistic learning and reasoning models, and so on. Formal definitions are given and related works are surveyed. One goal of this dissertation is to unify some of the concepts and developments from different areas for solving the algorithm selection problem.

We then present the main theoretical results in chapter 3. Rice's theorem is applied to illustrate the infeasibility of building an automatic algorithm selection system based only on the analytical methods. Also, an abstract, general framework of problem instance hardness and algorithm performance for search based on Kolmogorov complexity is developed to discuss the related issues. These results are then applied to the study of GA-hardness. Driven by the infeasibility of analytical methods, we turn to the machine learning-based inductive approach, which is experimental in nature and exploits various machine learning techniques.

Chapter 4 describes our discovery of the multifractal property of joint probability distributions of Bayesian networks. Also, an algorithm for finding the MPE based on the multifractal meta-heuristic is developed and the experimental results are presented.

Chapter 5 reviews major issues in machine learning and describes the proposed machine learning-based approach for algorithm selection.

In Chapter 6, we apply the proposed approach machine learning-based to algorithm selection for sorting. Different presortedness measures are studied. A set of random generation algorithms are presented. The algorithmic experimental design is described and various machine learning algorithms are applied to induce the algorithm selection model from the experimental data. The overall performance of the algorithm selection system is also evaluated.

Chapter 7 applies the machine learning-based methodology to algorithm selection for the MPE problem. Several MPE instance characteristics are discussed and a random instance generation algorithm is described. Again, various machine learning algorithms are applied to induce the algorithm selection model from the experimental data and the results evaluated.

Chapter 8 summarizes the results of this dissertation and points out some open questions and future directions.

Chapter 2

Background

Automatic algorithm selection is at the crossroads of many fields including *computational complexity theory*, *computability theory*, *algorithmic information theory*, *experimental algorithmics*, *machine learning*, and *artificial intelligence.* In this chapter, we review concepts from related areas. We present the formal definitions and how they are related to this research. These concepts are being introduced partly to make the thesis more self-contained and, more importantly, to set up the formal notations that will be used throughout this thesis. Also, related works are surveyed in the end of this chapter.

2.1 Theoretical Aspects

In this section, we review concepts in computational complexity theory, computability theory, and algorithmic information theory. Complexity theory provides fundamental concepts for algorithm selection. Results in computability theory and algorithmic information theory will be used in chapter 3 to derive our theoretical results on the undecidability of automatic algorithm selection.

2.1.1 Computational Complexity Theory

Computational complexity theory [GJ79, Pap94] is a central field of computer science. The main goal of this field is to classify computational problems according to their *intrinsic computational difficulty.* It provides the basics for solving the automatic algorithm selection problem. The central question of complexity theory is: *Given a problem, how much computing resources do we need to solve it?*

Problem, Instance and Algorithm

We begin by defining terms. We distinguish *a problem* from *an instance.* Informally, a *problem* is a general question to be answered, usually having several *parameters.* A problem is described by giving a configuration of all its parameters and a statement of what properties the *answer* or *solution* is required to satisfy. An *instance* of a problem is obtained by specifying particular values for all of its parameters. Formally, a problem is a total function on strings of an alphabet, such as $\{0, 1\}$. Let $\{0,1\}^*$ or $\sum^*$ represent the set of all finite strings made up of 0s and 1s. From [Joh90], we have the following definition:

Definition 1 (Problem and Instance) *A problem is a set X of ordered pairs (I, A) of strings in $\{0,1\}^*$, where I is called an instance. A is called an answer for that instance, and every string in $\{0,1\}^*$ occurs as the first component of at least one pair.*

As an example, consider the REACHABILITY problem on a directed graph:

REACHABILITY
INSTANCE: A directed graph $G = (V, E)$ and two nodes $v_1, v_2 \in V$.
QUESTION: Is there a path from v_1 to v_2?
ANSWER: "Yes" if there is a path. Otherwise, "no".

Like all problems, the REACHABILITY problem has an infinite set of possible instances. For example, for the graph as shown in Figure 2.1, we want to ask if there exists a path between node A and E.

Notice that the REACHABILITY problem asks a question that requires a "yes" or "no" answer. Such problems are called *decision problems.* Besides decision problems, there are *counting problems*, *search problems*, *optimization problems*, etc. Counting

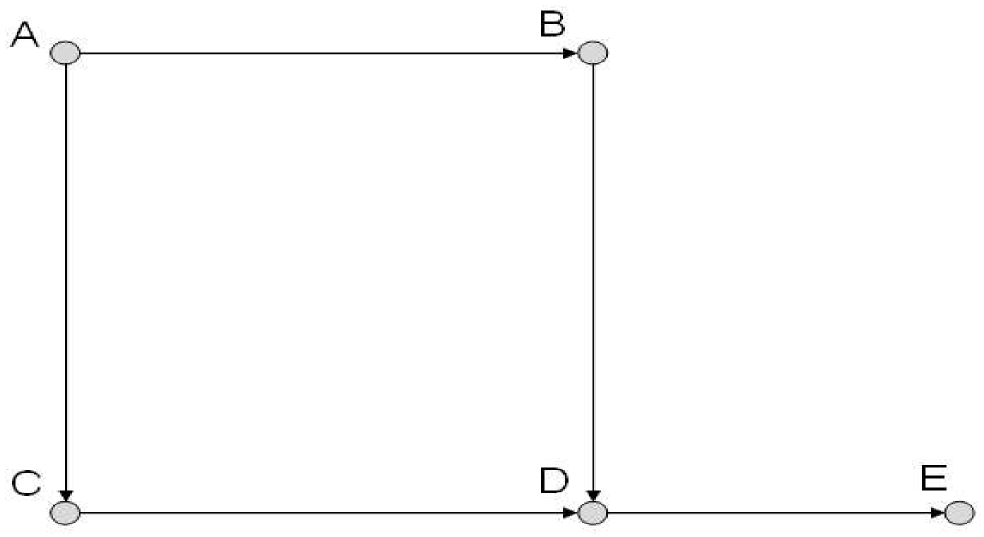


Figure 2.1: An Instance of the REACHABILITY Problem

problems are functions in which all answers are nonnegative integers. Search problems are string relations. Optimization problems are special kinds of search problems in which the objective is to find the best (maximum or minimum) of all possible solutions defined by *an objective function.*

Decision problems are a particular important class of problems. Most complexity classes are defined on decision problems because decision problems have a very natural formal counterpart called *languages*, which provide a suitable object to study in a mathematically precise theory of computation.

Definition 2 (Language) *A language is any subset of* $\{0,1\}^*$.

If L is a language, then the corresponding decision problem R_L is $\{(x, yes) : x \in L\} \cup \{(x, no) : x \notin L\}$. Given a decision problem R, the corresponding language is $L(R) = \{x \in \{0,1\}^* : (x, yes) \in R\}$. A problem is a language that is a set of strings over a finite alphabet. An instance of the problem is a string.

In computer science, to solve a problem means to find an *algorithm* that solves all instances of the problem (assuming enough time, space, and other resources are provided). The goal of any algorithm that solves problem X is as follows: given an instance x, produce an answer a such that $(x, a) \in X$.

Algorithms are general, mechanical procedures that can be followed step-by-step to solve a problem. Formally, algorithms are defined on a model of computation such as *Turing machines (TMs)* [GJ79, Pap94]. There are many variants of Turing machines. In this thesis, we will only consider *Deterministic Turing machines (DTMS)* and *Nondeterministic Turing machines (DTMS)*. First, let us look at DTMs. A DTM is composed of an infinite tape bounded on the left, a read-write tape head, and a finite control unit.

Definition 3 (Deterministic Turing Machine) *A DTM is a tuple*

$$M = (Q, \Gamma, \sum, \delta, q_0, B, F)$$

where Q is the finite set of states, Γ is the the finite set of allowable tape symbols, B is the blank symbol ($B \in \Gamma$), $\sum$ is the set of input symbols ($\sum \in \Gamma, B \notin \sum$), δ is a transition function $Q \times \Gamma \rightarrow Q \times \Gamma \times \{Left, Right, Stay\}$, q_0 is the start state ($q_0 \in Q$), F is the set of final states ($F \subseteq Q$).

The machine starts from q_0, takes a step according to δ, changes its state, prints a symbol, and advances the cursor; it then takes another step, and another, and so on. For a given input x, if a machine M halts at the final state "yes", we say that M *accepts* x; if it halts at the final state "no", then it *rejects* x; if it halts at the final state h, the output will be the string of M at the time of halting. It is possible that the machine never halts.

Turing machines provide an ideal computational model to solve string-related problems; i.e. computing string functions and accepting and deciding languages. Let L be a language. Let M be a Turing machine such that for any string x, if $x \in L$, then $M(x) =$ "yes", and if $x \notin L$, then $M(x) =$ "no". Then we say that M *decides* L. If L is decided by some Turing machine M, then L is called a *recursive language.*

We say that M accepts L whenever, for any string x, if $x \in L$, then $M(x) =$ "yes"; but if $x \notin L$, then M might halt, or it might continue forever without halting. If L is accepted by some Turing machine M, then L is called a *recursive enumerable language.*

Turing machines can be thought of as algorithms for solving string-related problems. In order to solve problems using a Turing machine, we first need to encode the problems into strings. It should be clear that any "finite" mathematical objects of interest can be represented by a finite string over an appropriate alphabet. Throughout this thesis, we shall move freely between strings and problem instances and algorithms without explicitly saying so, assuming that any problem instances and algorithms can be encoded into some strings of *0s* and *1s.* We assume the encoding scheme is "reasonable" in the sense of Garey and Johnson [GJ79]. Once we have fixed the representation, an algorithm for a decision problem can be found by simply creating a Turing machine that decides the corresponding language. That is, the Turing machine accepts if the input represents a "yes" instance of the problem, and rejects otherwise.

A Turing machine is an exceedingly simple yet amazingly powerful model of computation. As Turing claimed [Tur36], any process that can be naturally called an effective procedure is realized by a Turing machine. This is known as the famous **Church-Turing Thesis**: *everything computable is computable by a Turing machine.*

Efficient Computability and the Complexity Class P

Having defined the notion of algorithms as Turing machines, we now consider the time complexity of algorithms and problems which are central to the algorithm selection problem.

The *time* used in the computation of a DTM program M on an input x of length n is the number of steps occurring in the computation up until a halt state is entered. For a DTM program M that halts for all inputs $x \in \sum^*$, its *time complexity* is defined as follows:

Definition 4 (Time Complexity) $T_M(n) = max\{t(n) : \exists x \in \sum^*,\ |x| = n,$ *such that M takes time $t(n)$ on x*$\}$.

From the definition, we can see that the time complexity function of an algorithm expresses its time requirements by giving, for each possible input length n, the largest amount of time needed by the algorithm to solve a problem instance of that size. In this sense, it is a *worst-case analysis*.

Different algorithms have different time complexities. Particularly, computer scientists have realized that there is a significant distinction between *polynomial time algorithms* and *exponential time algorithms*. The time complexity function $f(n)$ for a polynomial time algorithm can be bounded by some polynomial function $p(n)$; i.e., there exists a constant c such that $f(n) \leq cp(n)$ for all $n \geq 0$. An exponential time algorithm cannot be so bounded and has an exponential time complexity function. Researchers associate efficient algorithms with those that terminate within time that is polynomial in the length of the input. An algorithm is called *"efficient"* if it is a polynomial time algorithm. Exponential time algorithms are considered as *"inefficient"*.

One reason for ruling out exponential rates is that the known Universe is too small to accommodate exponents as pointed out in [Lev86]. It is about 15 billion light years $\sim 10^{61}$ Planck Units wide. A system of $\gg R^{1.5}$ particles packed in R Planck Units radius collapses rapidly, be it Universe-sized or a neutron star. So the number of particles is $< 10^{91.5} \sim 2^{304} \ll 4^{4^4} \ll 5!!$. Another reason is because of computer's physical limitations [Raw92]. No computer can perform more than about 10^{15} switches per second. Beyond that, the frequency of visible lights, the energy needed for the switching, will break the chemical bonds holding solids together. Now we are already in the 10^9 operations per second range, so we can expect at best a million-fold speedup. But 2^n grows by a factor of better than a million whenever n increases by 20. Furthermore, in theory, nothing can happen faster than about 10^{23} seconds, which is the time light takes to cross the diameter of a proton. 10^{23} is only one hundred million times faster than 10^{15}, and 2^n eats that up whenever n increases by 27.

The intrinsic complexity of a computational problem is measured by the complexity of the best algorithm that has been found to solve it so far. Hence, a problem is considered as "tractable" if there exists a polynomial algorithm to solve it. In contrast, a problem is "intractable" if it is so hard that no polynomial algorithm can possibly solve it. The notion of tractability is formally captured by the complexity class P.

Definition 5 (The Complexity Class P) *The class of languages (decision problems) that can be solved by a polynomial time deterministic Turing machine.*

A wide range of problems are known to be in P, and researchers are continually attempting to identify more members. Perhaps the most significant addition to the membership list recently is the AKS algorithm [AKS02] for primality test. The complexity of AKS algorithm is $\Omega((\log n)^{12} f(\log \log n))$ time where f is a polynomial, which means the time it takes to run the algorithm is at most a constant times the number of digits to the twelfth power times a polynomial evaluated at the log of the number of digits.

Algorithm selection for problems in P is not so badly needed comparing to other much harder problems because basically all candidate algorithms are considered as efficient. However, it is still important for some crucial applications that require optimal computations. Also, although they are all polynomial time algorithms, no one is best for every situation. Thus it is of highly theoretical importance to be able to identify the best algorithm for a given task. In this thesis we shall use sorting as a case study of algorithm selection for P problems.

NP-Completeness and Intractability

We have shown that P represents the class of problems that can be *solved in polynomial time by a deterministic Turing machine.* Researchers have also identified in practice another large class of problems that can be *verified in polynomial time (by* *a deterministic Turing machine).* There is no known deterministic algorithm that can solve this class of problems in polynomial time. But if somebody claimed for a given instance of the problem that the answer is "yes", then there exists a polynomial time algorithm to check or verify the claimed answer. To capture this notion of polynomial time verifiability, we introduce the definition of Nondeterministic Turing machines (NDTMs) and the complexity class of NP as follows:

Definition 6 (Nondeterministic Turing Machine) *A Turing machine that has more than one next state for some combinations of contents of the current input symbol and current state. An input is accepted if any move sequence leads to acceptance.*

At the first glance, NDTM seems like an ill-defined concept since we have not specified what the machine should do when confronted with two or more possible transition choices. In this case, the machine takes any one of the available choices. A NDTM answers "yes" to a decision problem if any possible sequence of choices leads the machine to end up in a "yes" state, and answers "no" if every sequence of choices leads it to end up in "no" state or to not halt. Another way of understanding NDTM is seeing it as a DTM with an additional guessing head. When "solving" a problem, it first guesses an answer, then checks it. For a given input x, any NDTM program will have many possible computations, one for each possible guess. We say that a NDTM accepts x if at least one of these is an accepting computation; i.e., halts at the "yes" state. Using NDTM we can define the complexity class of NP as follows:

Definition 7 (The Complexity Class NP) *The class of languages (decision problems) that can be accepted by a polynomial time nondeterministic Turing machine.*

Clearly, a NDTM can solve any problem that a DTM can solve because DTMs are special cases of NDTMs. A natural question is whether or not there are any problems that NDTMs can solve but DTMs can not solve. This is the most important open problem in theoretical computer science; i.e., the *P versus NP* problem [Coo00].

Among all problems in NP, *NP-complete problems* are the hardest ones because if you can solve any single NP-complete problem in polynomial time, then all problems in NP can be solved efficiently; i.e. $P = NP$. To make the notion more precisely, we need to introduce the concepts of *reduction* and *completeness* [GJ79].

Definition 8 (Reduction) *We say that language L_1 is reducible to L_2 if there is a function R from strings to strings such that for all input x, the following is true: $x \in L_1$ if and only if $R(x) \in L_2$. R is called a reduction from L_1 to L_2.*

To make it more meaningful, we usually require R to be computable by a deterministic Turing machine in space $\Omega(\log n)$ and time $\Omega(p(n))$ where $p(n)$ is polynomial [Pap94].

Reduction is transitive. If R is a reduction from language L_1 to L_2 and R' is a reduction from L_2 to L_3, then the composition $R \cdot R'$ is a reduction from L_1 to L_3. This fact orders problems with respect to their difficulty. We shall be particularly interested in the problems at the end of this chain.

Definition 9 (Completeness) *Let C be a complexity class, and let L be a language in C. L is C-complete if any language $L' \in C$ can be reduced to L.*

Complete problems are an extremely central concept and tool for complexity theory. They capture the essence and difficulty of a complexity class. NP-complete problems are complete problems of class NP. The SATISFIABILITY problem is the first problem that was proven to be NP-complete by Cook in 1971 [Coo71]. It is specified as follows:

SATISFIABILITY (SAT)
INSTANCE: A set of boolean variables V and a collection of clauses C over V.
QUESTION: Is there a satisfying truth assignment for C?

The famous Cook's Theorem is stated as follows:

Theorem 1 Cook's Theorem *SAT is NP-complete.*

The proof of Cook's theorem is done by transforming a SAT problem into a nondeterministic Turing machine. Therefore, if we can solve SAT quickly, we can find solutions to nondeterministic Turing machines quickly. By the definition of NP, any NP problem can be solved by a nondeterministic Turing machine in polynomial time. So in turn if we can solve SAT quickly, we can solve any NP problem quickly. This proves that SAT is NP-complete.

Having one concrete NP-complete problem in hand, people have found many other NP-complete problems by reducing them to SAT. The proof of a problem A to be NP-complete consists of two steps. First, prove it to be in NP. Second, prove that some NP-complete problem B reduces to A. Thousands of problems from various applications, such as TSP, VERTEX COVER, CLIQUE, PARTITION, HAMILTONIAN CIRCUIT, and so on, have been shown to be NP-complete since SAT [GJ79].

Recall that if we can solve any NP-complete problem in polynomial time, then we show $P = NP$. Conversely, if we can prove $P = NP$, then all NP-complete problems can be solved efficiently. Unfortunately, this is very unlikely to be true although it is possible. Most people believe that $P \neq NP$, but nobody has been able to either prove it or disprove it so far. An aspect of the importance of NP-completeness is this: once we have shown that our problem is NP-complete, it seems reasonable to direct our efforts to the many alternative approaches available, for example, developing approximation algorithms, studying special cases, analyzing the average performance of algorithms, developing randomized algorithms, resorting to heuristic methods, and so on.

NP-hard Optimization Problems

So far we have restricted our discussions to decision problems for defining complexity classes such as P and NP. In real world applications, many practical problems are *search problems* and *optimization problems.* A search problem consists of a set of instances and, for each instance I, a set of solutions $S(I)$. An algorithm is said to solve a search problem if, given any input instance I, it returns the answer "no" whenever $S(I)$ is empty and returns some solution $s \in S(I)$ otherwise. An optimization problem is a special kind of search problem. Like search, it also consists of a set of instances and a set of solutions $S(I)$ for each instance I. Additionally, for each instance I and its candidate solution δ, an objective function that assigns a positive rational number m for each pair (I, δ) is defined. Each optimization problem is either a minimization or maximization problem. Correspondingly, the optimal solution has the minimum or maximum m value among all possible solutions. If a problem is NP-complete, then the corresponding search or optimization problem is usually NP-hard, which means at least as hard as the NP-complete problem. The notion of NP-hard is formally defined by *Turing Reduction* and the *Oracle Turing Machine*, which we refer readers to [GJ79]. But, remember the intuition it wants to capture is *"NP-complete or harder"*. In practice, we often call an optimization problem NP-complete for convenience if its decision version is NP-complete.

For NP-complete problems, automatic algorithm selection becomes significantly valuable. For any NP-complete problem, there exists some tractable special cases that are in P. We should find the dividing line and apply different algorithms for these special cases. For the general cases, there are usually many alternative algorithms available, be it approximation algorithms, randomized algorithms, or heuristic ones. These algorithms often have different properties and perform best on different classes of instances. Therefore, selecting the most appropriate algorithm for a given instance becomes imperative. Also, it is not a secret that not all instances of a NP-complete problem are equally hard. Researchers have discovered that by changing the values of some parameters of an instance, it will go through an easy-hard-easy phase-transition process [CKT91]. It is interesting and important to identify the turning points and to treat the hard instances separately using specially-designed algorithms. In this thesis, we will choose the MPE problem, whose decision version is NP-complete, as a case study of building automatic algorithm selection system for NP-hard optimization problems.

Generally speaking, there are at least two approaches to solve the algorithm selection problem: *analytically* and *experimentally*. Furthermore, there are three analytical methods of algorithm comparison and algorithm selection. The first one is to apply worst-case analysis to the algorithms, compare their complexities, and select the best one. Although this method is the most fundamental one, it does not always work well. Some algorithms may have bad worst-case time complexities but they perform very well on average in practice. Also, many complex, approximate, randomized, and/or heuristic algorithms such as genetic algorithms [Hol75, Gol89] simply resist formal analysis. The second way is to analyze the algorithms' average-case complexity. This method often requires a strong assumption on the distribution of input instances, which is hard to make in most cases. The third method considers instance classes instead of single instances [Man85]. It defines a subproblem of the original problem by some intuitive criteria of instance easiness (or hardness) and then study the worst-case complexity of the subproblem. Algorithms are designed and analyzed on these subproblems, with resource requirements increasing smoothly when moving to larger subproblems. The drawback of this method is that it is feasible only to simple problems like sorting and is hard to be applied to NP-hard optimization problems which are the most important ones in practice. This method provides a way to design adaptive algorithms that are optimal for some measures, but it is usually impossible to design such an algorithm that is optimal for all measures. There is also an algorithm selection problem for different adaptive algorithms. Furthermore, all these analytical methods are generally not suitable for making predictions about the empirical hardness of problem instances and the run time performance of the algorithms. The ultimate goal of the analytical approach is to select the best algorithm by just analyzing the description of the algorithms and the input instance (without running them). Ideally, we would like to have a program that is able to take as input both the descriptions of the instance to be solved and the candidate algorithms and return the best algorithm. We will show in next chapter the inherent hopelessness of this scheme.

In this thesis, we propose an experimental machine learning-based methodology of automatic algorithm selection in which the analytical results serve mainly as a source of domain knowledge to guide the experimental design. This methodology is more reasonable and more feasible because of the following reasons. First, even in some cases where analytical algorithm selection is possible, the theoretical results still need to be implemented and verified. Second, in practice, algorithm selection experts seldom depend solely on theoretical analysis without observing the algorithms' run time behaviors. Third, the analytical method has its inherent limitations. We will explore into this point more in chapter 3. Finally, the developments in the field of experimental algorithmics, machine learning and AI have provided a set of powerful techniques for implementing an experimental machine learning-based approach to solve the algorithm selection problem.

2.1.2 Computability Theory

One goal of computational complexity theory is to identify intractable decision problems for which no efficient (polynomial time) algorithm exists to solve them. But there even exists some problems for which *no algorithm exists at all.* Computability theory is concerned with identifying such kind of *unsolvable problems.* The core of automatic algorithm selection can be stated as the following decision problem: *Given an algorithm A, an input instance I, and an algorithm performance criteria measured as a real number C, is A's performance on I better than C?* In order to study the decidability of this problem, we briefly go over some basic concepts of computation theory [HS01] in this section.

Recursive Languages and Decidability

In the previous section, we have considered decision problems as languages, and algorithms as Turing machines that can recognize languages by methods such as determining whether a given string is in the language or not. We have also talked about recursive and recursively enumerable languages. Let us quickly recall that a language L is recursively enumerable (RE) if it is accepted by some Turing machine M. M will halt on the input w if $w \in L$. If $w \notin L$, however, M will halt or run forever. In this case, we say M does not solve the decision problem for the language L. A language is said to be recursive if there exits a Turing machine that accepts it, and also halts on all inputs. A Turing machine that always halts solves the decision problem for the language it accepts. The definition of decidability is given as follows:

Definition 10 (Decidable) *A language is decidable if and only if it is recursive.*

Intuitively, a problem is decidable if there exists an effective procedure; i.e., an algorithm (or a computer program), that solves the problem. If no such procedure exists, then the problem is undecidable. One may think that the existence of undecidable problems poses limitations to the effectiveness of analytical problem solving in mathematics.

The Existence of Undecidable Languages

The existence of undecidable languages is easy to see simply because *there are more languages than there are Turing machines* [Pap94, HS01]. We just do not have enough Turing machines for all languages. This can be shown by the difference between *countable* and *uncountable* sets. Formally a set S is countable if it is either finite or there exists some one-to-one correspondence between S and the set of natural numbers. The set of real numbers is uncountable; i.e., there are more real numbers than natural numbers. This has been shown by Cantor using the *diagonalization argument.* Each Turing machine can be encoded as a binary string. It is easy to see there is an one-to-one correspondence between the set of natural numbers and the set of Turing machines. So we say the set of Turing machines is *countable.* On the other hand, the set of languages (decision problems) is uncountable because we can build an one-to-one correspondence between the set of languages and the set of real numbers in $[0, 1]$ which is *uncountably infinite.* A language is just a set of strings over some alphabet $\sum$. We can order the strings in $\sum^*$ by an ordering and give each string an index number starting from 1. For example, $\{0, 1, 00, 01, 10, 11, \ldots\}$. Any subset of it forms a language. So a language can be encoded like this: we use a 1 in the i^{th} decimal position to denote the inclusion of the string with index number i and 0 otherwise. The set of languages is precisely the set of such encodings (real numbers in $[0, 1]$) and hence is uncountable.

Undecidability of the Halting Problem

Turing showed that the first well-defined decision problem, *the HALTING PROBLEM*, can not be settled by *any* effective computational procedures. The halting problem is as follows.

HALTING PROBLEM (HALT)
INSTANCE: A Turing machine M and an input word w.
QUESTION: Does M eventually halt on input w?

Now we give the proof for the undecidability of HALT.

Theorem 2 *HALT is undecidable.*

Proof. Assume the HALTING PROBLEM is recursive and there is a program H that takes as input a Turing machine M and an input I. If M halts, it returns "halt", otherwise it returns ''loop". Now we construct another program K using H as follows:

```
function K() {
        if(H() =="loop") return; //halt
        else while(true); //loop forever
}
```

Since K is a program, let us use K as the input to K. If H says that K halts then K itself would loop (that's how we constructed it). If H says that K loops then K will halt. In either case H gives the wrong answer for K. Thus the assumption about H is wrong. $\square$

Once we have got one undecidable problem, we can prove the undecidability of other problem by reducing it to the previously determined undecidable problem.

Rice's Theorem

Following the undecidability of $HALT$, we can show that a set of other problems are also undecidable. This is the result of *Rice's theorem* [Ric53, Hut01].

A *property* of recursively enumerable languages is a set of RE languages that possesses that particular property. A property is *trivial* if it is either empty (satisfied by no language at all), or all RE languages (satisfied by all RE language). Rice's theorem is stated as follows:

Theorem 3 Rice's Theorem *Any non-trivial property of recursively enumerable languages is undecidable.*

Proof. See [Ric53, Pap94, Hut01] □

Rice's theorem means that given an arbitrary algorithm A, there is no algorithm that decides the non-trivial property of the language defined by A. The proof of Rice's theorem consists of a reduction from the Halting Problem. It shows how one could use a property-checking algorithm to devise an algorithm for solving the Halting algorithm. We refer interested readers to [Ric53, Pap94, Hut01] for details. In chapter 3 we will use this result to show the undecidability of the algorithm selection problem.

2.1.3 Kolmogorov Complexity and Algorithmic Information Theory

In the practice of algorithm design, special-case algorithms are designed for some special class of problem instances. They work best (sometimes only) on the target class of instances because the algorithms compile some special information of the input instances. One problem of theoretical interest is how to measure the mutual information between input instances and the algorithms to solve them. Intuitively, *the more mutual information there is, the better the algorithm will perform on that instance.* The concept "information" here is different from the notion *entropy* as developed by Shannon in classical communication and information theory [Sha48, CT91]. There, entropy is defined on a *random variable* X with outcomes in a set

S. It is a measure of the information produced from the set if a specific value is assigned to X. It also measures the uncertainty change before and after we instantiate X. It is a probabilistic notion that is natural for information transmission over communication channels as it was developed. But in the context of instance hardness and algorithm performance, we are interested in measuring the information conveyed about an individual finite object (the problem instance), or a string, by another finite object (the algorithm), or another string. In order to investigate this issue, we need to look at results developed in Kolmogorov complexity [LV93] and algorithm information theory [Kol65, Cha87]. Concepts reviewed in this section will be used in chapter 3 to develop an abstract framework of problem instance hardness and algorithm performance.

Universal Turing Machine

We use Universal Turing Machine (UTM) as our computational model. A UTM U is a Turing machine that can simulate any other Turing machine T. We can encode the action table of T in a string, construct a Turing machine that expects on its tape a string describing the action table of T followed by a string describing the input tape of T, and then compute the tape that the encoded Turing machine T would have computed. A UTM can be thought of as a standard general-purpose computer that runs programs on data in the usual way. It is a fundamental fact that such machine exists and can be constructed effectively. It guarantees that some properly defined concepts using UTM have some sort of invariant property [LV93].

Kolmogorov Complexity

Kolmogorov complexity, also called *descriptive* or *algorithmic complexity* [LV90], was developed by Solomonof [Sol64], Kolmogorov [Kol65], and Chaitin [Cha66]. The basic idea is to measure the complexity of a string by *the size in bits of the smallest program that can produce it.* The idea comes from the observation that *random strings are more difficult to be compressed than strings that have internal regularities are.* The formal definition is given as follows:

Definition 11 (Kolmogorov Complexity) *Let U be the Universal Turing Machine. Let p be the shortest program that generates s on U. The Kolmogorov complexity of a string $s \in \{0,1\}^*$ is defined by*

$$K(s) = min\{|p| \mid U(p) = s\}$$

Here, $U(p) = s$ denotes that U, starting with p, terminates leaving s.

Definition 12 (Conditional Kolmogorov Complexity) *Let x and y be strings $\in \{0,1\}^*$. Let p be the shortest program that generates x on U given y. The conditional Kolmogorov complexity of x given y is defined by*

$$K(x|y) = min\{|p| \mid U(p, y) = x\}$$

By definition, it is easy to see that $K(x|\ \emptyset) = K(x)$ and $K(x|x) = 0$. These definitions are machine-independent because they are defined on the UTM. Kolmogorov complexity measures the randomness of string s by its *incompressibility*. The more random string s has a larger value of $K(s)$. $K(s)$ will never be larger than $\approx |s|$, for we can always generate s by a program whose code is 'print s', and this program is only marginally longer than s. For example, let s_1 = '$000000000000000'$, s_2 = '$479, 224, 570, 368, 102'$. Intuitively s_2 looks more random then s_1 although by Shannon's measure they have the same probability $(\frac{1}{2^{15}})$ of being selected from an ensemble of all possible strings of 15 bits. We can write a program of *'print 15 0s'*, which is shorter than its length, to generate s_1. But for s_2, perhaps we can only write a program of *'print 479,224,570,368,102'* to generate it since it is so random that we can not make use of any structural regularity to make the program shorter. So in this example $K(s_2) > K(s_1)$. The fact $K(s_1) < |s_1|$ implies there is internal structure and regularity in s_1 so that it can be compressed, while $K(s_2) \approx |s_2|$ means that s_2 is random and incompressible. For this reason, randomness means incompressibility. Because of the invariance theorem of the UTM, the notion of Kolmogorov complexity actually captures an intrinsic property of a string, or a finite object, independent of the choice of the mode of description.

Algorithmic Information Theory

Kolmogorov complexity can be used to measure how much information is encoded in a string x and how much information a string y contains about x [Kol65, LV90, LV93]. The Kolmogorov complexity of a string can be seen as the *absolute information* of the string. One interpretation of $K(x)$ is as the quantity of information needed to generate x from scratch. Similarly, $K(x|y)$ quantifies the information needed to generate x given y. If $K(x|y)$ is much less than $K(x)$, then we may say that y contains a lot of information about x. For applications, this definition of information has the advantage that it refers to individual objects, not to objects that are treated as elements of a set of objects with a probability distribution given on it [Sha48].

Definition 13 (Algorithmic Information in x about y) *Let x and y be two strings. The algorithmic information in y about x is defined by*

$$I(y : x) = K(x) - K(x|y)$$

The value $K(x)$ can be interpreted of as the amount of information needed to produce x, and $K(x|y)$ can be interpreted as the amount of information which must be added to y to produce x. By definition, $0 \leq I(y : x) \leq K(x)$ and $I(x : x) = K(x)$. When string x contains no information about x, $I(y : x) = 0$.

In our study of problem instance hardness and algorithm performance, all instances and all algorithms can be seen as strings. Hence we can use the concepts from algorithmic information theory to measure the absolute information contained in an instance and the relative information contained in an algorithm about an instance. They serve naturally as measures of instance hardness and algorithm performance.

2.2 Experimental Aspects

In this section we review major issues that arise in the experimental aspects of automatic algorithm selection. The main purpose of algorithmic experiments in this research is to generate a high quality, representative training data set that contains the knowledge we seek with regards to solving the algorithm selection problem; i.e., the dependency relationships between problem instance features and algorithm performance measures.

2.2.1 Experimental Analysis of Algorithms

In the past thirty years, the dominant method of algorithm analysis has been the asymptotic analysis of the algorithm's worst-case or average-case behavior. Although experimental analysis of algorithms has been intensively used in some other related fields such as operations research and artificial intelligence, it is almost invisible in the algorithm and data structure community. However, recently there has been a growth in interests in experimental algorithmic works [Hoo94, GGM$^+$97, Mor00, Joh02, McG02, MSF$^+$02]. This newly-developed field is called *experimental algorithmics*, which studies algorithms and data structures by joining experimental studies with the traditional theoretical analysis.

This is because more and more researchers have realized that theoretical results alone cannot tell the full story about algorithm's performance. Moreover, many of the recently invented algorithms, especially randomized and heuristic algorithms such as Genetic Algorithm (GA) [Gol89], are too complex for a detailed mathematical analysis to be reasonable. Usually for most such algorithms in practice, some facts are known about their performance but they have really not been fully analyzed. One of the main advantages of experimental algorithmic analysis is that it allows to investigate *how algorithmic performance statistically depends on problem characteristics.* This is the central problem of automatic algorithm selection.

Although in recent years a collection of rules-of-thumb to follow and a number of pitfalls to avoid have been accumulating regarding to the experimental analysis of algorithms [GGM$^+$97], at present we do not have a solid science of experimental algorithmics yet. There are no clear right answers for many questions on experimental analysis of algorithms. In [Joh00], Johnson has listed this among one of the challenges to theoretical computer science in the new century. In this research, we will discuss some basic issues of experimental algorithmics while centering around the algorithm selection problem. These issues include: the selection of algorithms and features, the random generation of problem instances, the measurements of algorithm performance, how to assure the reproducibility and platform-independence of the results, how to analysis the results, and so on.

2.2.2 Algorithmic Experiment Setup

Any algorithmic experiments involve *running some algorithms on some problem instances, collecting the data, and analyzing the result.* Also, any experiments have in advance some *hypotheses or questions* to test and ask. The central hypothesis of our research is that *there exists some dependency between problem instance characteristics and algorithm performance, and this knowledge can be exploited to build algorithm selection system to gain more efficient computation.* In designing experiments to investigate this hypothesis, we must consider the following issues.

Which Algorithms and Which Features

Because instance features and algorithms are the most basic elements in our experiments, the choice of the combination is all-important. Domain knowledge obtained from analytical results plays an important role in making the choice. For any particular computational problem, there usually exists a list of algorithms that have been defined and studied to a certain extent in the literature. In almost all cases, there does not exist a single algorithm that outperforms all others on the entire problem domain. Different algorithms have different properties and each algorithm typically is most efficient for only a subset of all possible instances. The knowledge of which algorithm works better on which class of instances is usually not explicitly available. Sometime we may have some vague information like this: "if the permutation has a *high presortedness*, then Insertion Sort should be used", or "if the network is *not large and sparse*, then the Clique-tree Propagation algorithm works very fast". In these cases, the cutting point or the working range of the algorithm still needs to be determined more precisely by experiments in order to facilitate automatic algorithm selection.

Selecting the candidate algorithms is often straightforward. In theory, any algorithm can be used as a candidate. But in practice, we can use domain knowledge to filter out algorithms that are obviously inferior or too difficult to implement. A good candidate algorithm should be easy to implement and by which a special class of instances can be efficiently solved.

The selection of instance features relies heavily on domain knowledge as well. A good feature should be easy to compute (usually negligible compared to the problem-solving time) and most relevant to the performance of a particular algorithm or a particular class of algorithms.

Generation of Training and Test Instances

One of the biggest practical challenges in any algorithmic experiments is assembling the required test instances. The best test instances are probably those *real world instances* that are taken from real world applications. They represent the most important set of instances. Unfortunately, it is rarely possible to collect more than a few real world problems instances for most computational problems. In this research, it has taken us nearly 3 years to accumulate only around 20 real world Bayesian networks. In some applications, real world instances can also have limitations since some potential real world instance may not have been implemented yet. Also, real world instances may not cover all problem characteristics of interest.

An alternative is to use randomly generated instances. This requires us to develop a random generator that can *generate instances with specified parametric characteristics uniformly at random.* This problem by itself is quite a challenge. It is still an open question whether there exists polynomial random generation algorithms for all NP languages [San00]. The naive approach is to exhaustively enumerate all possible instances and pick one uniformly at random. But this does not apply to many random generation problems simply because there are far too many possible instances in general. The method we use in this research is a polynomial time, almost uniform random generation algorithm based on *Markov chain approach* [Sin93, MM00, IC02]. The basic idea is to construct a dynamic stochastic process, or a finite Markov chain, whose states correspond to the set S of structures of interest. The process is able to move around the state space by means of random local perturbations of the structures. Moreover, the process is designed to be *ergodic.* If it is allowed to evolve in time, the distribution of the final state tends asymptotically to a unique *stationary distribution* π, which is independent of the initial state. The stationary distribution π can be designed to be uniform. By simulating the process for a sufficiently many steps and outputting the final state, we are able to generate elements of S uniformly at random if π is set to a uniform distribution.

Yet another choice is combining real world instances and random generation together. We can first analyze the real world instances and get a characteristic vector statistically representing the distribution of real world instances, then input the characteristic vector to the random generator to produce *random synthetic real world instances.* In our experiments, all three kinds of instances will be used.

Algorithm Performance Measurements

Measuring and collecting algorithm performance data is another practical challenge. The related issues include *what to measure, how to ensure reproducibility of the results, and how to reduce variances.*

Running time and solution quality are two most common measures of algorithm performance, although sometimes other measures like space can be important. For tractable problems like sorting, measuring actual running time (CPU time or wall clock) is important. But it may become hard to interpret because it is influenced by many uncertain factors of the environments. The granularity of the time unit may cause problems as well. In our experiments on sorting we had to find a High Resolution Time Stamp Facility [IBM02] to measure run time in *microsecond* because JDK only provides *millisecond* resolution. For algorithms solving NP-hard problems, other measures like *number of calls to a crucial subroutine* is usually a better choice. Heuristic search algorithms often need to evaluate many search points (calling the solution evaluation subroutine) during the problem solving process. Measuring the number of calls to the evaluation subroutine ensures the results independent of platforms and thus improves the reproducibility of the results. Another issue is *the balance between time and solution quality.* So far, the most effective way to fairly compare different heuristic algorithms is to allow all algorithms to consume the same amount of computation resources, with distinctions being based on the quality of solutions obtained [RU01].

Variance reduction techniques [McG92] are also necessary when dealing with randomized algorithms. If not carefully handled, the variability between runs of algorithms and between instances may obscure the results and make it hard to interpret. *Instance variance* can be reduced by using the same set of randomly generated instances for all runs so that all algorithms see the same instances. *Algorithm variance* can be reduced by performing the same algorithm on the same instances for many runs and recording the average performance.

2.3 Bayesian Networks

Reasoning under uncertainty is a common issue in our daily life and a central one in AI. Bayesian networks (BNs) [Pea88, Nea90, RN95], also known as Bayesian belief networks, belief networks, causal networks, and probabilistic networks, are currently the dominant technique in AI for uncertain knowledge representing and reasoning. The underlying principle is to apply probability theory to building intelligent systems for reasoning with incomplete information and uncertain knowledge. However, until twenty years ago it was still considered impractical to do so by mainstream AI community due to the difficulty of manipulating the exponentially-sized joint probability distributions. The breakthrough was made in early 80s' when Pearl introduced a probabilistic graphic model, namely the Bayesian networks, and published his efficient and elegant message propagation inference algorithms for particular classes of

BNs [KP83, Pea86, Pea88]. BNs combine the representational and algorithmic power of graphic theory with probability theory. They provide a compact and natural tool for representing uncertain knowledge and facilitate efficient inference and learning algorithms. BNs have proven useful in a wide range of applications including medical diagnosis, decision support systems, real-time monitoring, intelligent user interface, image analysis and so on.

Bayesian networks play an important role in this research because they serve for two purposes at two different levels. In one hand from the point of view of learning, it is a meta-level knowledge representation model (or a classifier) just like a decision tree or a Naive classifier. In the other hand, one of its inference problems, the MPE problem, is chosen as test case of our machine learning-based algorithm selection methodology. In this section, we introduce basic concepts about representation and inferences in Bayesian networks. We will introduce Bayesian networks learning in chapter 5 together with some other learning algorithms.

2.3.1 Representation

In BNs, the domain of interest is viewed as a probabilistic model that consists of a set of random variables. Each random variable can take particular values with certain probabilities [1]. In theory the JPD of the probabilistic model contains complete information of the random variables and their dependencies. But in practice it is obviously infeasible to deal with the JPD directly. BNs sidestep the JPD and work directly with *conditional probabilities* in order to reduce the representational and computational complexity.

The Syntax of Bayesian Networks

A Bayesian network [RN95] is a graph in which the following holds:

1. A set of random variables corresponds to the nodes of the network.

[1]For simplicity and without losing generality, we only consider discrete variables.

2. A set of directed links connects pairs of nodes. The intuitive meaning of an arrow from node X to node Y is that X has a direct influence on Y.

3. Each node has a Conditional Probability Table (CPT) that quantifies the effects that the parents have on the node. The parents of a node X are all those nodes that have arrows pointing to X.

4. The graph is a Directed Acyclic Graph (DAG).

The topology of the network can be thought of as an abstract knowledge base of the domain. It represents the general structure of the dependencies among these variables.

The Semantics of Bayesian Networks

The semantics of Bayesian networks can be understood in two ways. The first is to see the network as a representation of the JPDs. The second is to see it as an encoding of a collection of conditional independence statements. These two views are equivalent but they emphasize two different aspects. The former is helpful in understanding how to construct BNs. The latter is helpful in designing efficient inference algorithms.

From the first viewpoint, a Bayesian network provides a complete description of the domain. It encodes JPD in a compact manner. Every entry in the JPD can be calculated from the information in the network using the following formula, i.e., the *chain rule*:

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i|\pi(x_i)) \tag{2.1}$$

The numbers in the network − these probabilities − are interpreted as beliefs; i.e., probability is a measure of belief in a proposition given particular evidence [Che85]. That is why they are also called belief networks. This interpreting avoids the difficulties associated with the classical frequency definition of probabilities.

From the second viewpoint, topologically, the BNs use three types of local connections to decompose a JPD to a series of conditional independence statements: *linear*, *diverging*, and *converging.*

The concept of *d-separation* [RN95] is fundamental in BNs. It states that *"If every path from node X to node Y is d-separated by a set of nodes E, then X and Y are conditionally independent given E"*. This is very helpful in designing efficient inference algorithms.

A set of nodes E d-separates two nodes X and Y if and only if there is a node Z on the path from X to Y for which one of the following condition holds:

1. The path is linear and Z is in E.

2. The path is converging and Z is in E.

3. The path is diverging, and neither Z or any descendant of Z is in E.

2.3.2 Inferences

One main purpose of building Bayesian networks is to use it to perform inference; i.e., to compute answers to users' queries for prediction, diagnosis or explanation about the domain, given exact values of some observed evidence variables. There are basically two types of BN inference tasks: *belief updating* and *belief revision.* Many belief updating algorithms can be used for belief revision with just minor modifications, and vice versa.

Belief Updating

Belief updating is also called *probabilistic inference*, denoted Pr. Its objective is to calculate $P(X|E)$, the posterior probabilities of query nodes X, given some observed values of evidence nodes E.

$$P(X|E) = \frac{P(X)}{P(X,E)} \tag{2.2}$$

A simple form of it results when X is a single node; i.e., we are interested in computing the posterior marginal probabilities of a single query node. Pr mainly involves a marginalization operation over query nodes. As what the name tells, these posterior probabilities change smoothly and incrementally with each new item of evidence.

Belief Revision

The task of belief revision amounts to finding the most probable configuration of some hypothesis variables X, given the observed evidence. The resulting output is an optimal list of instantiations of the hypothesis variables, a list that may change abruptly as more evidence is obtained. Belief revision for the case when X contains all non-evidence nodes is also known as computing the *Most Probable Explanation*, or MPE. An *explanation* w for the observed evidence E is a complete assignment $\{X_1 = x_1, \ldots, X_n = x_n\}$ that is consistent with E. Computing the MPE w^* is finding an explanation such that no other explanation has higher probability.

$$P(w^*|E) = \max_w P(w|E) \tag{2.3}$$

MPE is a typical optimization (maximization) problem. Computing kMPE means finding the top k highest explanations.

In the cases when X only contains a partial subset of all non-evidence nodes, the task is called finding the Maximum a Posteriori Hypothesis, or MAP. MAP involves both marginalization and maximization.

Complexity of Inferences in Bayesian Networks

Computing Pr, MPE and MAP are all NP-hard. However, they still belong to different complexity classes. MPE is essentially a combinatorial optimization problem. MPE is NP-complete (more precisely, its decision version is NP-complete) [Shi94, Par02b]. Pr is harder. It is a counting problem and its complexity is $\#P$-complete (the functional version) [Coo90, Nea90]. Its decision version is PP-complete, which contains the languages for which there exists a nondeterministic Turing machine that the majority of the nondeterministic computations accepts a string if and only if the string is in the language [Pap94]. MAP is the hardest one. MAP combines both counting and optimization and it is NP^{PP}-complete [Par02a]. NP^{PP}-complete is the class of languages that can be recognized by a non-deterministic Turing machine in polynomial time given a PP oracle; i.e., if any PP query could be answered for free.

For a special class of BN topology such as polytrees, Pr and MPE are both polynomial [Pea88]. However, MAP remains NP-complete even on polytrees [Par02a].

Furthermore, the approximations of MPE, Pr, and MAP are NP-hard as well [DL93, AH98, Par02a]. Particular, MPE is $MAXSNP$-hard, which means means there is no approximation scheme for MPE until $P = NP$ [Pap94]. MAP on polytrees is also $MAXSNP$-hard [Par02a].

2.4 Related Works

In this section we review works that are directly related to our research of applying machine learning-based approach to solving the algorithm selection problem.

2.4.1 The Algorithm Selection Problem

The algorithm selection problem is originally formulated in [Ric76]. Later it has been mainly applied to the selection of problem-solving method in scientific computing [HCR$^+$00], specifically to the performance evaluation of numerical softwares. An abstract model of algorithm selection is also given in [Ric76] as reproduced in Figure 2.2, where x is the input instance in the problem space and w is the performance criteria. The input problem instance is represented as the feature(s) f in the feature space by a feature extraction procedure. The task is to build a selection mapping S that provides a good (measured by w) algorithm A to solve x subject to the constrains that the performance of A is optimized.

Algorithm selection can be either *static* or *dynamic*. Static algorithm selection system makes the selection and then commits to the selected algorithm, while dynamic algorithm selection system may change its selection dynamically by monitoring the running of the algorithm. One special kind of dynamic algorithm selection is *recursive algorithm selection* [LL00, LL01] in which a decision of algorithm selection needs to be made every time a recursive call is made. For example, a sorting algorithm often needs to recursively sort smaller instances. At each recursive call, you need to make the decision about which sorting algorithm to choose. The goal becomes to optimize a

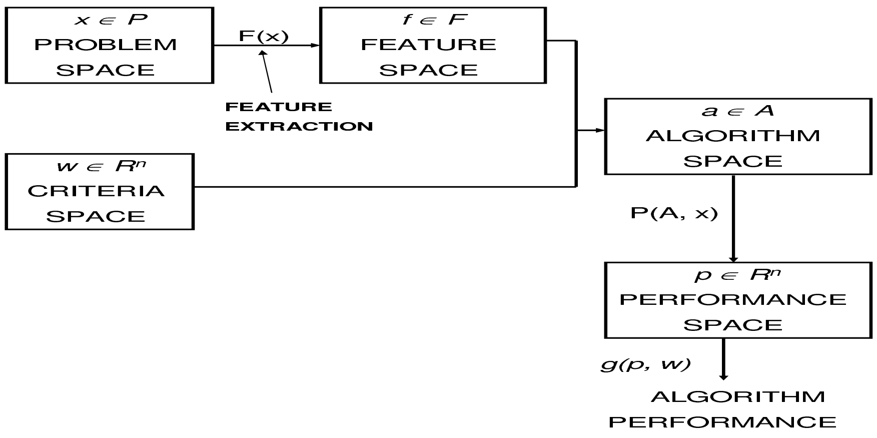


Figure 2.2: The Abstract Model of Algorithm Selection Problem

sequence of algorithm selection decisions dynamically. In this research we study static algorithm selection. The methodology developed in this research can be extended in the future to solve the dynamic algorithm selection as well.

2.4.2 Algorithm Selection in Various Applications

Algorithm or problem-solving technique selection has been studied in various fields. In data compression, Hsu and Zwarico [HZ95] studied the automatic synthesis of compression techniques for heterogenous files based on some qualitative and quantitative properties of each segment of the file. In machine learning, Brodley [Bro93, Bro94] investigated the selection of inductive learning algorithms for different learning tasks based on the learning algorithm's inductive bias or selective superiority. In planning, Fink [Fin98] described a selection technique based on statistical method to select a planning algorithm and the time bound of the problem solving. In Constraint Satisfaction Problem (CSP), Minton [Min96] developed an inductive learning system that configures constraints Satisfaction programs. In these works the knowledge representation of the meta-reasoner is often a set of “if-then” rules derived from simple statistical models. As machine learning becomes more mature, it is natural to apply more advanced models to build better algorithm selection systems.

2.4.3 Meta-reasoning Techniques

Researchers in AI have long realized that great efficient gains can be achieved by allocating a portion of costly computational resources to meta-level deliberation about the best way to solve a problem. In particular, Breese and Horvitz [BH90] studied the intelligent reformulation or restructuring of a belief network before solving the inference problem. They also described the metareasoning-partition problem; i.e., the problem of ideally apportioning resources between a meta-analysis and the actual problem solving. They presented principles for computing the ideal partition of resources under uncertainty for several types of user requirements. In the field of real-time AI, *flexible computation* [Hor90] and *anytime algorithms* [Zil93] offer an efficient mechanism to trade off computation time for quality of result. In these models computations can be interrupted at “any time” and they can still produce results of a guaranteed quality. Gomes and Selman [GS97] studied the problem of algorithm portfolio design to combine several different algorithms into a portfolio in order to gain improvement in terms of overall performance. Crawford et al. [CFGS02] proposed a framework for on-line adaptive control of problem solving, which combines ideas from control systems with adaptive solving techniques.

In the particular field of Bayesian network inference, some researchers have studied inference algorithm selection and integration. Santos [SSW95] developed a distributed architecture, OVERMIND, for unifying various probabilistic reasoning algorithms that have both anytime and anywhere properties. Anywhere algorithms can exploit intermediate results produced by other algorithms. When different algorithms have both anytime and anywhere properties, they can be harnessed together into a cooperative system that exploits the best characteristics of each algorithm. Within the same framework, Borghetti [Bor96] and Williams [Wil97] studied the algorithm selection problem for BN inference algorithms as well. Borghetti used performance profiles and Williams developed a selection process based on simple analytical principles. Jitnah and Nicholson [JN96] investigated various network characteristics and empirically studied the relationship between instance features and inference algorithm performance.

2.4.4 The Bayesian Approach

In the related field of NP-hard problem solving, researchers have applied various machine learning techniques to learn the empirical hardness of optimization problems and to tackle hard computational problems. In particular, the Bayesian learning approach was used in [HRG$^+$01] for characterizing the run time of problem instances for randomized backtrack-style search algorithms that have been developed to solve a hard class of structured constraint-satisfaction problems. The same approach was also used in [KHR$^+$02] to learn the dynamic restart policies for randomized search procedures that take real-time observations about attributes of instances and about solver behavior into consideration. In some earlier work, Horvitz and Klein [HK95] constructed Bayesian models considering the time expended so far in theorem proving.

2.5 Summary

In this chapter we have reviewed concepts from various related fields. Fields of theoretical aspects consists of computational complexity theory, computability theory, and algorithmic information theory. Fields related to experimental aspects emphasize the role of algorithmic experiments in our approach. Major experimental issues have been discussed. We have also introduced basics concepts about Bayesian networks. Finally, we have reviewed researches in various areas that are closely related to our work.

Our research aims at unifying ideas from these different areas and building algorithm selection systems using experimental, machine learning-based methods. As far as we are aware, this is the first attempt to systematically combine algorithmic methods and machine learning techniques for solving the algorithm selection problem. Most researches in algorithm selection use simple knowledge representation such as statistic models and simple "if-then" rules. Some of them rely on analytical principles that are hard to be automated and generalized. Our approach relies more on experimental methods to make the automation easier. Machine learning and data mining techniques can help us gain more insights and knowledge on algorithm's runtime behavior which analytical methods can not provide. We believe the proposed methodology can be extended to dynamic algorithm selection as well. From the perspective of solving hard problems, it also provides an effective way of integrating different algorithms together to build more powerful solvers for NP-hard optimization problems.

Chapter 3

Some Theoretical Results on Analytical Algorithm Selection

In this chapter, we present some theoretical results with the algorithm selection problem. We first show, by Rice's theorem, that *the general algorithm selection problem is undecidable.* We then propose a general, abstract theoretical framework of instance hardness and algorithm performance for search based on algorithmic information theory and Kolmogorov complexity. We claim that *algorithm selection for search is also undecidable because of the incomputability of Kolmogorov complexity.* We then apply the theoretical framework to GA-hardness to show the nonexistence of a predictive GA-hardness measure if based only on the description of the input instance and configurations of the GA. Driven by these theoretical results, we turn to a more feasible direction, *applying inductive approach*, rather than analytical approach. The proposed inductive approach relies significantly on experimental methods and machine learning techniques to build algorithm selection systems.

3.1 Undecidability of the General Automatic Algorithm Selection Problem

We have introduced in chapter 2 that any algorithm can be rendered as a Turing machine and that any problem can be rendered as a language. The automatic algorithm selection problem asks to design an algorithm, or a Turing machine, that takes as inputs both the descriptions of two candidate algorithms and a problem instance and outputs evaluations of these two algorithms' performance according to some criteria. The most common criteria includes problem-solving time and quality of the solution returned. Assuming both algorithms return the correct solution for that instance, we consider that the one taking less time performs better and thus should be selected. Hence the language (decision problem) of the algorithm selection problem can be formulated as follows:

Definition 14 (Language of the Algorithm Selection Problem)

$A_{TM} = \{< M, I, S, T > | M$ is a TM and M accepts I in T steps and the returned solution is S $\}$

Here, M corresponds to the algorithm. I is the input instance to be solved by the algorithm. S is the solution or answer for the instance and T represents a given number of steps that M operates.

Recall that Rice's theorem states that any non-trivial property of a Turing machine is undecidable. A property corresponds to a set of recursively enumerable (r.e.) languages possessing the particular property. A property is trivial is it is either empty (not satisfied by any r.e. languages) or it is satisfied by all r.e. languages. The property is non-trivial if there is both at least one Turing machine that has the property and at least one that does not have the property.

One consequence of Rice's theorem to pragmatic computer science is shown in the following grading program example [Tay98]. "*Suppose that the instructor in an introductory programming language class has asked students to write a C++ program that computes a given partial recursive function F. (We might even suppose the function is defined by only dozens of cases.) Usually, the instructor needs to review the code as well as examine a listing of output for a number of inputs. Since implementations vary tremendously from student to student, the instructor might hope for some grading program that could be run on a given student's source code so as to determine whether it in fact computes F correctly.* " Unfortunately, Rice's theorem asserts that there can be no such grading program. This is because that first $\{F\}$ is a nontrivial set of partial recursive functions, i.e., a nontrivial set of recursive enumerable languages. By Rice's theorem, the membership of the following set

$$\{ M \mid M \text{ is a Turing machine that computes } F \}$$

is undecidable. Furthermore, by Church-Turing's thesis, a C++ program is equivalent to a Turing machine. So the membership of the following set:

$$\{ P \mid P \text{ is a C++ program that computes } F \}$$

is also undecidable. We must conclude that the envisioned grading program can not exist after all (at least if Church-Turing's thesis is true). Therefore, Rice's theorem eliminates the hope of algorithmically testing the input-output behavior of arbitrary programs. This implies that *program verification techniques* are in general noneffective. In general, in order to establish program correctness, we must resign ourself to working with nonmechanistic techniques.

Going back to our algorithm selection problem, it is easy to see that the A_{TM} is a non-trivial language. Hence by applying Rice's theorem we have the following result:

Theorem 4 *A_{TM} is undecidable.*

Proof. First, A_{TM} is recursively enumerable. We can simply build a TM to simulate M on input I to see whether M halts in time T and returns solution S. Second, A_{TM} specifies a property that is non-trivial, i.e., not all recursively enumerable languages can represent a TM that accepts I in time T and returns a solution S, and there at least exists one language that satisfies this. According to Rice's theorem, any non-trivial property of recursively enumerable languages is undecidable, so A_{TM} is undecidable. $\square$

It implies that *in general, there can be no hope of finding a means of automatic algorithm selection only from the descriptions of these algorithms.* This general result should not be surprising because the HALTING PROBLEM basically says that you can not even tell whether a Turing machine (algorithm) can halt or not. A related result is the undecidability of the equivalence of two Turing machines, which says that given two Turing machines M_1 and M_2 the language $EQ = \{e(M_1)e(M_2)|L(M_1) = L(M_2)\}$ is undecidable.

3.2 An Abstract Framework of Instance Hardness and Algorithm Performance for Search

We have demonstrated the undecidability of automatic algorithm selection using Turing machine as the computational model. In this section, we look at a more restricted and practical case: automatic algorithm selection for search.

Intuitively, a search algorithm can perform better if it compiles more information about the input instance, or the search space. Hence we need to investigate the information contained in one string (the algorithm) about the other string (the problem instance). Kolmogorov complexity [LV93] is a measure of absolute information content of individual objects. The algorithmic information defined on Kolmogorov complexity measures the mutual information of two individual objects. Therefore, it is natural to use these concepts to study instance hardness and algorithm performance in search. In the following, we present a general, abstract framework of instance hardness and algorithm performance for search based on Kolmogorov complexity. The framework is still in its crude form and needs to be further refined. It is not to be used as a predictive model. Instead, we apply it to show that the general problem of automatic algorithm selection for search is impossible because of the incomputability of the mutual information between a search algorithm and an input instance. We first set the stage by introducing the Black Box Optimization (BBO) model [WM95, WM97, Cul98].

3.2.1 Black Box Optimization (BBO)

In the BBO model, we are given a function, $f : X \rightarrow Y$, for which we seek the set of optimal solutions, x^*, that maximizes or minimizes f. X is a finite set of strings of both *0s* and *1s* and Y is a finite set of real numbers. People have interchangeably called f the fitness function, objective function, cost function, search space, or simply the problem instance. A heuristic search algorithm for BBO usually starts from an initial point of the search space or a population of initial points, explores the search space while keeping a population of one or more solutions $x \in X$ and the associated y values, and tries to improve upon these solutions. The search algorithm can be seen as a function mapping from its search history to a new point or a set of new points in the search space according to some heuristic. For simplicity, we assume it never revisits points in its searching history. Algorithms such as hill-climbing, simulated annealing, tabu search, and GAs are all working in this manner.

Time and solution quality are two of the most common measures in evaluating and comparing search algorithm performance. In heuristic search, time can be measured by the number of search steps (the number of times the search space has been visited), and solution quality can be measured by the error between the global optimal and the best solution so far. In the most general case, the algorithm performance measure is defined as a function of these two factors according to users' requirements. Sometimes efficiency is more crucial whereas sometimes precision is more important. In either situation, this information can be extracted from the search histogram.

3.2.2 Random Instance and Random Algorithm

Both search problem f and search algorithm a in the BBO model can be represented as strings $\in \{0,1\}^*$. Let $t(f)$ be the string representation of the mapping table of f. It has N entries listing all pairs of (x, y) in the forms of strings of $1s$ and $0s$. Defining f amounts to specifying $t(f)$. We can observe in practice that some problem instances, or search spaces, are more regular than others. The regularity of an instance can be exploited to design efficient algorithms for them. However, algorithms that compile some useful problem domain information have more structures than blindly random search, thus they solve problems more efficient. We define *the randomness of a problem* as follows [AM88]:

Definition 15 (Randomness) *The randomness of a problem instance* $f : \{0,1\}^N \rightarrow$

R is defined in bits by

$$R(f) = log_2 K(t(f)). \tag{3.1}$$

where $K(t(f)$ is the Kolmogorov complexity of $t(f)$.

$R(f)$ ranges from ≈ 0 to $\approx N$ because of the logarithm. Using $R(f)$, we define a *random problem* versus a *structured problem.* We will fix a threshold R_0 (the particular choice of which is not crucial to most of the theory) and introduce the following definition [AM88]:

Definition 16 (Random & Structured Problem) *A problem $f : \{0,1\}^N \to R$ is random if $R(f) \geq R_0$. A problem is structured if it is not random.*

Since a search algorithm can also be seen as a function and coded into a string, we can define the concept of *random algorithm* and *structured algorithm* in the same manner. Please note that the words "random" and "structured" have precise meaning here regarding to the length of the shortest programs to generate them. By this definition, both random (probabilistic) search and general brute force enumeration search are random algorithms in terms of their Kolmogorov complexity. It is straightforward to understand that random (probabilistic) search is random (Kolmogorov). The general brute force enumeration algorithm is random because it just enumerates all points in the search space and treats all points equally. It has no bias over any search points. Thus the shortest program to generate it has to list all of its mapping pairs.

One fundamental result of Kolmogorov complexity is that nearly every string is random. Correspondingly, we can say that in the BBO model almost all problems are random problems and almost all algorithms are random algorithms. This fact suggests a method to generate a random problem (or a random algorithm) in a probabilistic manner. Suppose we have a fair coin as the ideal random (probability) source. For each entry in the truth value table $t(f)$ of a problem f, we flip a coin to set the value. This will generate a random problem with very high probability.

Random problem (instances) and random algorithm are two fundamental concepts on our subject. Random problems are the "hardest" problems because they contain no internal structures for any algorithm to exploit. An example is the *needle-in-a-haystack* problem [Gre93]. No algorithm can solve such a problem more efficiently than random search [WM95, WM97]. Random (Kolmogorov) search algorithms are the least efficient algorithms because they comply no information about the problem and just visit the search space randomly.

3.2.3 Instance Hardness and Algorithm Performance

The randomness of an instance indicates its hardness. If an instance is highly random, then it contains almost no internal structures for any algorithm to exploit. Thus it is hard to solve. To capture this intuition, we define *instance hardness* as follows:

Definition 17 (Instance Hardness) *The hardness of an problem instance $f : \{0,1\}^N \to R$ is defined as*

$$H(f) = R(f) \tag{3.2}$$

where $R(f)$ is the randomness of f.

However, the information about an instance in an algorithm indicates the algorithm's performance on solving the instance. If an algorithm contains a lot of information about the instance, it can solve the instance more efficiently. For example, the hill-climbing algorithm assumes that the search space contains a single peak, so the algorithm can solve the instance easily. We define *algorithm performance* as follows:

Definition 18 (Algorithm Performance) *The performance of algorithm a on instance f is defined as*

$$P(a, f) = I(a : f) \tag{3.3}$$

where $I(a : f) = K(f) - K(f|a)$ is the algorithmic information contained in a about f.

3.2.4 The Incomputability of Kolmogorov Complexity

We have provide natural definitions of instance hardness and algorithm performance using Kolmogorov complexity. The unfortunate fact about Kolmogorov Complexity is that it can not be explicitly computed. Given an individual object, there is no way to tell if it is incompressible as shown in the following proof [LV93].

Theorem 5 *The Kolmogorov Complexity $K(x)$ of a finite string x is not computable.*

Proof. a) There are 2^n binary strings of length n and only $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ binary programs of length $< n$. Hence there is one x (at least) that is not computed by a program of length $< n$. This x has complexity $K(x) >= n$.

b) Suppose $K(.)$ were computable. Now, for every n there is a lexicographical first x, say x_n, of complexity $K(x_n) >= n$ (by a)). But by the computability of $K(.)$ we can compute the complexities of all binary strings of length n, and hence find the lexicographical first x among them that has complexity $K(x) >= n$. By definition, this is x_n. Since the only information we required to compute x_n was the number n (can be given in $\log n$ bits) plus a fixed standard program, we obtain $K(x_n) <= logn + O(1)$. This contradicts $K(x_n) >= n$ for all n apart from a finite initial set of values of n. Hence $K(x)$ is not computable. $\square$

This limits the practical applications of Kolmogorov Complexity. As a result in our context, it means that *automatic algorithm selection for search is also not decidable.*

3.2.5 Deceptive Problems and Deceptively-solving Algorithms

Now let us consider the following question: given a problem instance, can we deliberately design an algorithm that performs *worse* than random search? If the problem happens to be a random one, it contains no internal structured information so we cannot design an algorithm to perform either better or worse than a random search. If it is a structured problem but we do not have any information about its structure, it is still hard for us to design an algorithm worse than random search because we do not know what we should deceive. The only case we can do that is when the problem is a structured one and we know the information about the problem. The resulting algorithm can be called a *deceptively-solving algorithm* for its "purpose" is seek deceptiveness and to perform worse than random search. It is a structured algorithm since it contains structural information about the problem. But the algorithm uses the structural information in a "pathological" way. On the other hand, there are algorithms that are structured and perform better than random search. These algorithms can be called *straightforwardly-solving algorithms.*

Similarly, for a given nonrandom algorithm, if we do not have information about its search strategy and dynamics, it is impossible (with probability approaching zero) to come up with a structured problem that is deceptive, i.e., worse than a random problem. But, knowing the search strategy of an algorithm, we can design a problem to fail it. We call such a problem *deceptive.* Non-deceptive problems are called *straightforward problems.*

From this analysis, we can see that for a given structured instance, structured algorithms can be divided into straightforwardly-solving and deceptively-solving algorithms. Similarly for a given structured algorithm, the space of structured problem instances can be divided into straightforward problems and deceptive problems. From the viewpoint of information, we can interpret it as this: for a given structured problem, a straightforwardly-solving algorithm contains *positive information* about its straightforward problem; a random algorithm contains *zero information* about it; a deceptively-solving algorithm contains *"negative"* information. Similarly for a given structured algorithm, it contains *positive information* about the straightforwardly-solving problems of it; it contains *zero information* about the random problems; it contains *"negative" information* about its deceptive problems.

The concept of an algorithm containing "negative information" about a problem instance is not captured by Kolmogorov complexity because $K(x|y) \geq 0$. However, it is a reasonable concept in the context of instance hardness and algorithm performance. The Turing machine was invented by Turing in order to formalize the notion of "algorithm" and "effective calculability". In [Tur36], Turing sees the operation of a Turing machine no different from the process of human's computing following some mechanic procedure. Actually, before implementation, any algorithm is only an idea about the problem solving procedure existing in the algorithm designer's brain. The designer comes up with the idea based on his information and understanding to the input instance. Since it is just an idea in the designer's brain, it can be either right or wrong, i.e., it might not agree with the true information of the input instance. In search, the searcher's information about the search space is important to his performance. An accurate map of the search space can be of great to the searcher, but a wrong one can lead to nowhere. In this case, we can say the wrong map (a poor algorithm or a wrong heuristic) contains "negative information" about the search space.

Therefore, there are three factors that cause a problem instance to be hard for a particular structured algorithm:

1. *It is a random problem, so it has no structural information for the algorithm to use.* An example is the needle-in-the-haystack problem.

2. *It is a structured problem, but the algorithm contains zero information about it.* In this case, they are mismatched.

3. *It is structured, but deceptive.* In this case we say the algorithm contains *negative information* about the problem and performs worse than a random search.

In turn we call these three factors *randomness*, *mismatch* and *deception*.

3.3 GA-Hardness Revisited

In this section, we apply the abstract theoretical framework of instance hardness and algorithm performance to study GA-hardness. We discuss several major misconceptions in previous GA-hardness research and propose some promising directions for future research.

3.3.1 Genetic Algorithms (GAs)

GAs have been defined as generic search procedures based on the mechanics of the natural selection and genetics [Hol75, Gol89]. In order to apply a GA to a search or optimization problem, we must first encode it as *artificial chromosomes*, which can be strings, parameter lists, permutations codes, or any meaningful computer codes. Second, we must have a *fitness function* that helps evaluate the value of a chromosome, or a search point. Having encoded the problem into chromosomes and fixed a fitness function to discriminating a good chromosome from a bad one, we start the evolution process by creating an initial population of the encoded chromosomes. GAs then use some *genetic operators* to process the population generation by generation. This creates a sequence of populations and, hopefully, they will contain better solutions as the evolution goes on. The most often used genetic operators include *selection*, *crossover* (recombination) and *mutation*. These operators all have some genetic meaning. Selection allows better individuals to have more offspring. This is the principle of *natural selection*: survival-of-the-fittest. It prefers better solutions to worse ones. Crossover or recombination combines pieces of parental chromosomes to form new, hopefully better offspring. This is the principle of *sexual reproduction*, a reproduction strategy used by most advanced species. It ensures mixing and recombination among the genes of the offspring. Mutation simply modifies one piece of a single parental chromosome to create new individuals. This corresponds to *gene mutation* in genetics. It represents a random walk in the neighborhood of a single individual. In the process of natural selection, good results of mutation will be kept and bad ones abandoned.

GAs can solve some hard problems quickly and reliably. They use little problem-specific information and have a clear interface. They are also noise tolerant and easily extensible. GAs have been successfully applied into a broad range of applications in search, optimization, and machine learning. However, GAs do fail at times. GAs are complex systems and are hard to design and analyze. Ever since its invention [Hol75], researchers have put a lot of effort into understanding how GAs work and what makes a function or problem hard for GAs to optimize. There are still quite a few open problems despite more than 30 years of research and application. In research into the dynamics of GAs, various models have been developed to explain how GAs works, but none of them have given a definitive answer. Most models have aimed at building a GA theory with mathematical rigor and a reliable predictive ability, but they all have only gained limited success in the simplest, idealized settings. These include schema analysis [Hol75, Gol89, Whi93], difference equations [Gol87, Gol89], Markov chains [GS87, Dav91b, NV92, Vos93], epistasis analysis [Dav91a], landscape analysis [FM93a, CL96], transform models [BG91], etc.

Another line of research interwoven with GA dynamics is GA hardness. The goal is to identify which characteristics of problem instances are hard for GAs to optimize. Many characteristics and measures have been proposed to distinguish so-called GA-hard from GA-easy problems. But, again, none of the characteristics have achieved the goal of being a reliable predictive GA-hardness measure. These include deception [Gol87, Gol89, Whi90], isolation, noise [Gol93, Gol02], multimodality [GDH92, HG95, RW99a], landscape ruggedness, fitness distance correlation [JF95], epistasis variance [Dav91a, Nau98], epistasis correlation [Nau98, KNR01], site-wise optimization measure [Nau98], etc.

3.3.2 No Free Lunch Theorems

Wolpert and Macready's No Free Lunch (NFL) [WM95, WM97] Theorems are impossibility theorems for the BBO model. There are two NFL results respectively relating to the *space of all possible problems* and the *space of all possible algorithms.*

No Free Lunch Theorem 1 *Averaging over all possible problems, no search algorithm is better than any other algorithm.*

No Free Lunch Theorem 2 *Averaging over all possible search algorithms, no search problem is harder than any other problem.*

NFLs can be proven in many ways including the original proof [WM95, WM97], an adversary approach [Cul98], or a simple counting argument [RW99a]. One important lesson we can learn from the results of NFL theorems is that without taking into account any particular information of the search problem, no algorithm can perform better on average than random search, which visits points in the search space uniformly at random. NFL theorems have been criticized as too general to be practically applicable. We will use them here to illustrate the misconceptions in some efforts of GA-hardness research.

3.3.3 Misconceptions in GA-hardness Research

Informally, research studies into GA-Hardness ask what makes a problem hard or easy for GAs. This statement contains three important terms that needs to be elaborated: *problem*, *GAs*, and *hardness*. The meaning of these terms have never been precisely clarified and agreed. Many misunderstandings and misconceptions have been made because of this. In this section, we point out some of the major misconceptions that have made previous GA-hardness researches fumbled.

Misconceptions in The Problem Space: What Problems Should Be Used?

We have shown that there are three factors that cause a problem instance hard for a GA: randomness, mismatch, and deception. *The first misconception in the problem space is that blurring the differences among these three factors and feeding GAs with problems that are too hard to be meaningful.* A random problem like the needle-in-the-haystack is hard for all algorithms. Thus it is meaningless to use it as the test case. For example, in Grefenstette's paper "Deception Considered Harmful" [Gre93], the needle-in-the-haystack problem is used as a counterexample to dismiss the utility of deception theory and Building Block (BB) hypothesis [Gol89]. Although this example does prove that deception is not the only factor for GA-hardness, it *cannot* rule out the usefulness of schema theory and BB hypothesis in explaining the effectiveness of GAs on structured problems [Gol89]. We just should not force BB hypothesis to apply also on random problems. In [Gol02], Goldberg points out a similar concept of a "design envelope" to emphasize the importance of bounding problem difficulty: "...no BB searcher should be expected to solve problems that are in some way unbounded...".

A second misconception is *using few, specific problem instances to support a* *general result.* The size of most test instances used are less than 20 or 30 bits [Gol87, Gre93]. We would all expect that the relative hardness of problem instances to vary roughly with their size. A small problem, no matter what other characteristics it has, would be easy for most algorithms to solve. The fact that a minimal, deceptive problem does not cause a simple GA to diverge does not support that GAs are "surprisingly" powerful [Gol87]. Similarly, the fact that a small problem with high deception is easy for GAs to optimize also does not disprove deception's role in making problems with larger size hard for GAs [Gre93].

Another misconception in problem space is *applying GAs on functions or problems that are too general to be realistic.* Much of the research on fundmental theory have treated GAs as a generic search algorithm to solve all functions defined in the BBO model, most of which are random problems and artificially-made functions [MFH91, FM93a, FM93b, HG95] that we would never expect to meet in real world applications. The problem space defined as $f : \{0, 1\}^* \rightarrow R$ is actually too broad. P and NP problems are only a small fraction of these problems. Most real world optimization problems are NP-complete, and they are hard enough for GAs to deal with! In some sense, working on pathological man-made functions is a waste of effort. Another related viewpoint we want to mention here is that of the NFL theorems [WM95, WM97], which have caused a great deal of debate in the genetic and evolutionary community since first being published. Some researchers have directly applied NFL theorems to GA and claimed that a GA is as poor as a random search, so a GA is futile. This conjecture is discouraging, but it also overstates the case. NFL theorems are proven over the *space of all possible problems*, while, in practice, we are only interested in solving *real-world optimization problems.* For purpose of developing a usable theory of GA-hardness, we should not care about a GA's bad performance on any single random problem or specific, artifically deceptive functions. Also, in practice, if we have any problem-specific information, we are generally willing and able to incorporate the information into GAs' operators to speed up the GA performance. Goldberg has made some other points [Gol02] against the argument from futility. We believe that the framework we have presented supports the assertion that *" ... NFL Theorem does not eat GA lunch ... "* (Goldberg 2002).

Misconceptions in The Algorithm Space: What Are GAs?

GAs have been first considered only as a simulation of natural adaptive system [Hol75], then as function optimizers [Bet81] and general purpose search procedures [Gol89] that are assumed to converge to the global optimum. Furthermore, many "advanced" operators have been invented and GAs have been changed from simple to messy [Gol89, Gol02] to something almost unrecognized. *The most common misconception in the algorithm space is considering GAs as a single algorithm and seeking a universal GA's dynamics and general separation of GA-hard and GA-easy problems.* Any efforts like this are doomed to fail because for a given problem, if we change the configuration of parameters of GAs, we can get totally different convergence results such that for one GA the problem is easy, but for the other GA it is hard. What should we label it then, GA-easy or GA-hard? It is easy to see if allowing all possible GA operators, GAs can actually model any search algorithms as defined in the BBO model. So asking the hardness of a problem for GAs is the same as averaging the hardness of the problem over all possible algorithms. NFL theorems [WM95, WM97] have told us that averaging over the space of all possible search algorithms no problem is harder than any problem.

Experience from algorithm design practice tells us that algorithms with similar parameter values often have similar behavior on same problems. We should therefore consider classifying GAs into subclasses by their parameter values. GAs in the same subclass have similar parameters and are expected to have similar performance on the same problem instances. As a consequence, *research studies into GA dynamics and GA hardness should be done separately on these subclasses of GAs.*

Misconceptions in The Performance Measure Space: What Is GA-hardness?

In complexity theory [GJ79, Lee90, Pap94], the hardness or complexity of a problem is measured by the time complexity of the provably best algorithm possible, which is a function of the size of the input. A problem is tractable if we can find a polynomial time algorithm for it, intractable if only exponential time algorithm exists. Two important complexity classes are P and NP. P is the class of polynomial time solvable problems. NP is the class of polynomial time verifiable problems. NP-complete problems are the hardest one in NP. Most real world optimization problems are NP-complete. Some of them are even NP-hard to approximate. Assuming $P \neq NP$, it is impossible to find a polynomial time algorithm, including GAs, to solve a NP-complete problem. However, for some *subclasses* of a NP-complete problem, it is possible to find a GA that solves them in polynomial time if the GA can exploit some structural characteristics of the problems. Therefore the goal of GA-hardness research is to identify the characteristics of the instances that make GAs converge in polynomial time. If a GA converges well in polynomial time on an instance class with a common feature, we can say this instances class is easy for this GA (because of the feature). Otherwise it is hard for this GA. Again, we consider different classes of GAs with similar parameter setting rather than all GAs as a whole. *One misconception in practice is measuring GA's performance only on a few instances without considering the scaling issue.* The argument made in section 3.3.3 applies here as well.

Another and more serious misconception is *taking for grant the existence of a general a priori GA-hardness measure that can be used to predict a GA's performance on the given instance.* Many efforts have been put into searching for such a measure [Dav91a, GDH92, HG95, JF95, Nau98, NK98, NK00]. Ideally, people wish to have a program that takes as inputs a specification of the problem instance and the configuration of a GA and returns an estimation to the GA's performance on that problem so that the decision can be made in advance whether we should use GA or not. Rice's theorem [Hut01] has told us that in general it is impossible to compute in advance a GA-hardness measure without actually running the algorithm on the problem. This limits the general analysis, but it does not rule out the possibility of inducing an algorithm performance predictive model from some elaborately designed experimental results. For a parameterized real world optimization problem and a list of GAs with different configurations, we can design a controlled experiment to run these algorithms on the problems, collect the data, and inducing a predictive model from the data. This approach differs from traditional analytical method to study the performance of algorithms. It relies more on empirical approaches to investigate how algorithmic performance depends on problem characteristics.

Recently Rylander [Ryl01] has proposed a new definition of GA-hardness determined by the growth rate of the search space as a function of the size of the input instance. *Minimum Chromosome Length* (MCL) is defined to measure the size of the smallest chromosome for which there is a polynomial time computable representation and evaluation function. GA complexity class NPG (the class of problems that take more than polynomial time to solve) is defined as the problems that have a linear MCL growth rate. A GA-reduction is defined as a MCL-preserving translation from one problem's representation to another. A problem is GA-hard if all other problems in a particular class reduce to it. This theory has several important limitations. First, it attempts to capture the notion that a problem is hard for a class if any problems in this class can be reduced to it. In analyzing GA-hardness, however, we are really concerned with the *running performance of a GA on the problem.* The concept "hardness" implies a poor runtime performance instead of the reducibility between problems. Second, MCL seems to be hard to compute. Third, the complexity classes defined using MCL are not much different from the complexity classes in traditional complexity theory. It emphasizes only representation's (MCL growth rate) role on making a problem hard for GAs. By application of MCL, problems that have sublinear MCL growth rates are hard for GAs. All NP-complete problems belong to this type but still some of them can be solved by GA easily(converge in polynomial time). So the theory can not handle this case. In general this approach is hard to apply in practice and further refinement is needed.

Related GA-hardness Research

In the early 1990s Goldberg [Gol93] suggested five factors that make problems hard for GAs: *Isolation, Misleadingness, Noise, Multimodality, and Crosstalk.* Among them isolation and noise can be considered as randomness, misleadingness as deception, and crosstalk as mismatch. Multimodality is not an independent factor. In [Gol02] Goldberg has built a new model of problem difficulty for GAs that consists of three core difficulties: *Noise, Scaling, and Deception.* Respectively they correspond to randomness, mismatch and deception in our model.

For a given GA, we have divided the problem space of the BBO model into random problems, straightforward problems and deceptive problems by the mutual information between the problem and the GA. Culberson [CL96, Cul98] has studied the BBO model using adversarial approach in which an algorithm passes strings to an adversary and gets back an evaluation. The indifferent adversary in his model corresponds to random problems in our model. Similarly, a friendly adversary corresponds to straightforward problems, a vicious or mischievous adversary corresponds to deceptive problems.

We have used Rice's theorem to show the impossibility of having a predictive GA-hardness measure based only on the description of the problem and the GA's configurations. Reeves and Wright [RW99b] have arrived at the same conclusion using concepts of experimental design (ED). This result has also been verified by failures of previous efforts of identifying such an indicator.

We have proposed to properly classify GAs by their parameter values and study the dynamics of each class separately. De Jong holds the similar viewpoint. In [DSG95], he says that, "...we are uncomfortable with the notion of a "GA-hard problem" independent of these other details, unless we mean by such a phase that no choice of GA properties, representations, and operators exist that make such a problem easy for a GAFO to solve...".

We have also concluded that it is necessary to apply empirical methods to study GA dynamics and GA hardness. This coincides with some other researchers' point of view as well. Reeves & Wright [RW99b] propose to view GAs as a form of sequential experimental design. Goldberg [Gol02] has committed to an *engineer's methodology* in which a Wright-like decomposition method is adopt to design competent GAs, likening the design of a GA to that of an airplane in which the engineered object is the thing rather than the theoretical model.

3.4 An Alternative Direction: Experimental Approaches

Driven by the infeasibility of analytical approaches, we turn to propose an inductive approach to study problem hardness and algorithm performance.

For future directions of GA-hardness research, we suggest that research should focus more on real world NP-complete optimization problems rather than man-made functions. We also suggest that research should study the classification of various GAs rather than considering them as a whole. Furthermore, research should give up seeking a priori GA-hardness measures based *only* on the descriptive information of the problem and the GA in favor of experimental algorithmic methods to learn the predictive model from the posterior results of running various GAs on problem instances with designed parameter settings. Such an experimental approach may contain the following steps:

1. Pick up a real world optimization problem, for example, TSP.
2. Identify a list of problem characteristics that may affect GA's performance.
3. Generate random test instances with different parameter settings.
4. Select a list of GAs with different configurations.
5. Run the GAs on these elaborately designed instances and collect the performance data.
6. Apply proper data analysis tools or machine learning methods to build a predictive model out of the experimental data.

7. For a new instance, analyze its characteristic and use the learned model to infer the best GA to optimize it.

Similarly, for solving the automatic algorithm selection problem, we propose the following experimental procedure.

1. Identify a list of feasible instance characteristics using domain knowledge.

2. Identify a list of candidate algorithms for solving the problem.

3. Generate a representative set of test instances with different characteristic value settings uniformly at random.

4. Run the candidate algorithms on these elaborately designed instances and collect the performance data.

5. Apply Bayesian network learning techniques to induce a predictive model (a Bayesian network) out of the experimental data.

6. For a new instance, analyze its characteristic and use the learned Bayesian network to infer the most appropriate algorithm to solve it.

3.5 Summary

This chapter studies the theoretical aspects of automatic algorithm selection. The main conclusions include the undeciability of both the algorithm selection problem in general and the algorithm selection for search. We have also developed a general, abstract framework of both instance hardness and algorithm performance and have applied it to discuss misconceptions in GA-hardness research. Finally, we propose a more feasible inductive approach for GA-hardness research and for automatic algorithm selection systems. The inductive approach mainly relies on experimental approaches and machine learning techniques.

Chapter 4

Some Multifractal Properties of the Joint Probability Distribution of Bayesian Networks

In this chapter, we report our discovery of some multifractal properties of the Joint Probability Distributions (JPDs) of Bayesian networks. With sufficient asymmetry in individual prior and conditional probability distributions, the JPD is not only highly skewed (as shown by Druzdzel [Dru94]), but it is also stochastically self-similar and has clusters of high-probability instantiations at all scales. Based on the discovered multifractal property, we developed and tested a two-phase hybrid random sampling and search algorithm for the MPE problem. The experimental results showed that the multifractal property provides a good meta-heuristic for solving the MPE problem. Since the MPE problem is NP-complete, the multifractal meta-heuristic could be used to solve other hard optimization problems as well. These multifractal properties also strengthen the connections between Bayesian networks and thermodynamics. These connections have recently been exploited in popular Bayesian network inference algorithms based upon models from statistical physics [YFW00, PA02], such as free energy minimization.

4.1 Motivation

Bayesian networks (BNs) [Pea88] provide a compact representation of the JPD of an uncertain domain by specifying the JPD into the product of local prior and conditional probability distributions. The JPD over its variables can be seen as being created by a multiplicative process, combining prior and conditional probabilities of individual variables. By applying the Central Limit Theorem, Druzdzel [Dru94] demonstrated that "...asymmetries in these individual distributions result in JPDs exhibiting orders of magnitude differences in probabilities of various states of the model ... In particular, there is usually a small fraction of states that cover a large portion of the total probability space ..." (Druzdzel 94). Druzdzel's result suggests that considering only a small number of the most probable states can lead to good approximations in belief updating. Some questions of interest are: where and how can we find these high-probability instantiations in the space of JPDs? Is there any internal structure in the JPD that can facilitate search? If so, how can we characterize it? This paper attempts to answer these questions by demonstrating that the JPD of a BN is a random multinomial multifractal created from a random multinomial multiplicative cascade. By applying multifractal analysis, we show the existence of multifractal structure within the JPD. More specifically, the JPD, as a multifractal measure, can be partitioned into fractal subsets such that each subset supports a monofractal measure, and the JPD consists of clusters of high-probability instantiations at all scales. Based on these multifractal properties, we have also designed and tested a new random sampling and search algorithm for finding the MPE.

4.2 Multifractal Analysis

4.2.1 Fractals and Multifractals

Fractals are extremely irregular, self-similar sets [Man82, EM92]. A fractal is characterized by its *fractal dimension*. For example, the dimension of an irregular coastline may be greater than 1 but less than 2, indicating that it is not simply a "line" but

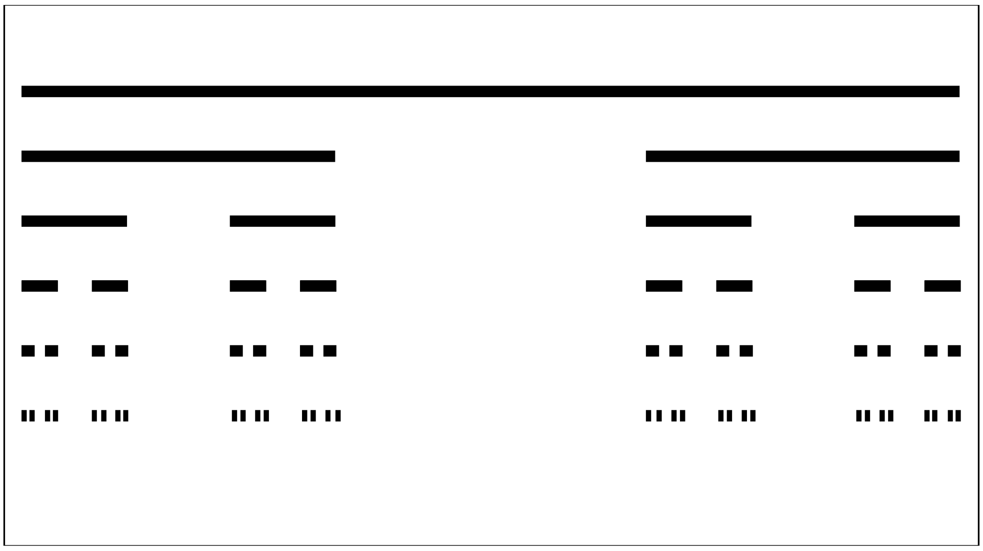


Figure 4.1: The Cantor Set

has some space-filling characteristics in the plane. The Cantor set [Man82] is the oldest and simplest man-made[1] fractal. As shown in Figure 4.1, it is constructed by removing the middle third from the unit interval and the remaining two subintervals have their middle third removed. This continues infinitely. Formally, the Cantor set is defined as follows:

$$K = \bigcap_{n=0}^{\infty} K_n \tag{4.1}$$

where $K_0 = [0, 1]$ and

$$K_n = \left(\frac{K_{n-1}}{3}\right) \bigcup \left(\frac{2}{3} + \frac{K_{n-1}}{3}\right) \qquad n = 0, 1 \ldots. \tag{4.2}$$

The dimension of a fractal set can be calculated by counting the number of covers that are required to cover the set of interest. In the Cantor set, when $n = 0$, 1 box of length 1 is needed to cover K_0; when $n = 1$, 2 boxes of 1/3 are needed for K_1, etc. Let N_δ be the number of boxes with length δ that are required to cover K, the

[1]Here we only consider "mathematical constructs". As pointed out by Dr. Rudolf Riedi, Romans had made some fractal mosaics long time before.

(Minkowski) fractal dimension is then defined as:

$$\frac{\log N_{\delta_n}(K)}{-\log \delta_n} = \frac{\log(2^n)}{-\log(3^{-n})} = \log_3 2 \tag{4.3}$$

The main difference between a *fractal* and a *multifractal* is that the former refers to a *set* while the latter refers to a *measure*. A measure μ assigns a quantity to each member of a set (the measure's support set), thus it defines a distribution of that quantity over the support set. Multifractal analysis [Man89, EM92, Har01] is related to the study of a distribution of physical or other quantities on a geometric support set. The support may be a line, plane, or a fractal. Multifractal measures are highly irregular and self-similar (exactly or stochastically). For instance, the distribution of gold over a geographical map of the USA is very irregular. It is found in high concentrations at only a few places, in lower concentrations at many places, and in very low concentrations almost everywhere. This description holds for all scales - be it on the scale of the whole country, one state, on the scale of meters, or even at a microscopic scale. Many other quantities exhibit the same behavior, i.e., the irregularity is the same at all scales, or at least statistically [Man89]. We call this kind of self-similar measure a *multifractal*. The concept of multifractal was originally introduced by Mandelbrot in the discussions of turbulence [Man72], and later applied to many other contexts such as Diffusion Limited Aggregation (DLA) pattern [BH91], earthquake distribution analysis [Har01] , and Internet data traffic modelling [RV97]. A multifractal is often generated by an elementary iterative scheme called *multiplicative cascade*.

4.2.2 Multifractal Spectrum

How can we characterize a multifractal measure? Clearly, we need more than just a fractal dimension. Simply counting the boxes as we did to the Cantor set is like counting coins without caring about the denomination. We must therefore find a description that assigns the measure in each box a weight [EM92]. In the following example, we use the Cantor measure [RV97] to illustrate the basic characterization of a multifractal. Consider the Cantor set again. Now we extend it by allocating a mass

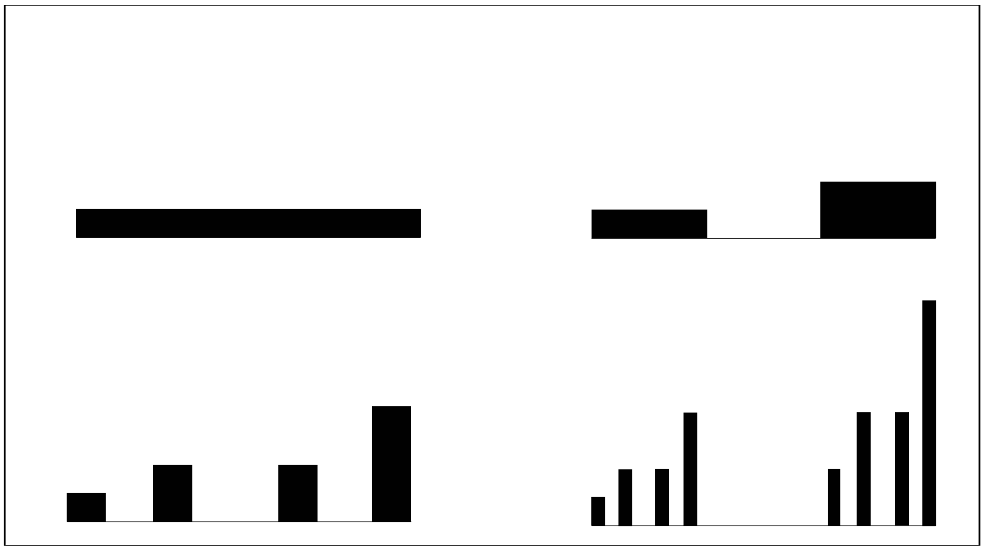


Figure 4.2: The Generating of the Cantor Measure

or a probability to each subinterval at each division. For example, we allocate 2/3 of the existing probability in an interval being divided to the right-hand subinterval, and 1/3 to the left-hand. The first four steps of generating the Cantor measure is shown in Figure 4.2.

The first step of multifractal analysis is to define α, the *coarse Hölder exponent* [Man89, EM92, Har01] , as the logarithm of μ, the measure of the box, divided by the logarithm of δ , the size of the box.

$$\alpha = \frac{\log \mu(box)}{\log \delta} \tag{4.4}$$

The multiplicative construction of μ makes it clear that the probability μ of a sequence of intervals will decay exponentially fast as the interval is being divided and shrinks down to a point. Thus α can be thought of as the local degree of differentiability of the measure, the rate of local probability change [Har01], or the strength of singularity [Rie99]. Once α is defined, we would like to draw the frequency distribution of α as follows: For each value of α, we count the number $N_\delta(\alpha)$ of boxes having a coarse

Hölder exponent equal to α; Then we define $f_\delta(\alpha)$ as the logarithm of $N_\delta(\alpha)$ divided by the logarithm of the size of the box.

$$f_\delta(\alpha) = -\frac{\log N_\delta(\alpha)}{\log \delta} \tag{4.5}$$

$f_\delta(\alpha)$ can be loosely interpreted as an approximation to the Minkowski fractal dimension of the subsets of boxes of size δ having coarse Hölder exponent α. The function $f(\alpha) = \lim_{\delta \to 0} f_\delta(\alpha)$ is called the *multifractal spectrum*. It characterizes a multifractal. The graph of $f(\alpha)$, often called $f(\alpha)$ curve [Man89, Har01], is shaped like the symbol "$\bigcap$", usually leaning to one side. Usually, there are bounds α_{min} and α_{max} such that $\alpha_{min} < \alpha < \alpha_{max}$. The α value of the peak is called α_0. Figure 4.3 plots the $f(\alpha)$ curve of the Cantor measure.

From the preceding discussion we can see that the basic idea behind multifractal analysis is to classify the singularities of the measure by strength. This strength is denoted by a singularity exponent α - the coarse Hölder exponent. Points of equal strength lie on interwoven fractal subsets. Each of these fractal subsets is a monofractal with a fractal dimension $f(\alpha)$. This is one of the several reasons for the term multifractal.

4.2.3 The Binomial Multifractal Cascade on [0, 1]

Many multifractal measures can be generated from an elementary iterative procedure called *multiplicative cascade*. The binomial measure is the very simplest multiplicatively-generated multifractal measure. Let m_0 and m_1 be two positive numbers adding up to 1. At stage 0 of the cascade, we start the construction with the uniform measure μ_0 on [0, 1]. At step $k = 1$, the measure μ_1 uniformly spreads mass (or probability) equal to m_0 on the subinterval [0, 1/2] and mass equal to m_1 on the subinterval [1/2, 1]. At step $k = 2$, [0, 1/2] is split into two subintervals [0, 1/4] and [1/4, 1/2], which respectively receive a fraction m_0 and m_1 of the total mass μ_1 on [0, 1/2]. Applying the same procedure to [1/2, 1], we obtain:

$$\mu_2[0, 1/4] = m_0 m_0, \quad \mu_2[1/4, 1/2] = m_0 m_1 \tag{4.6}$$

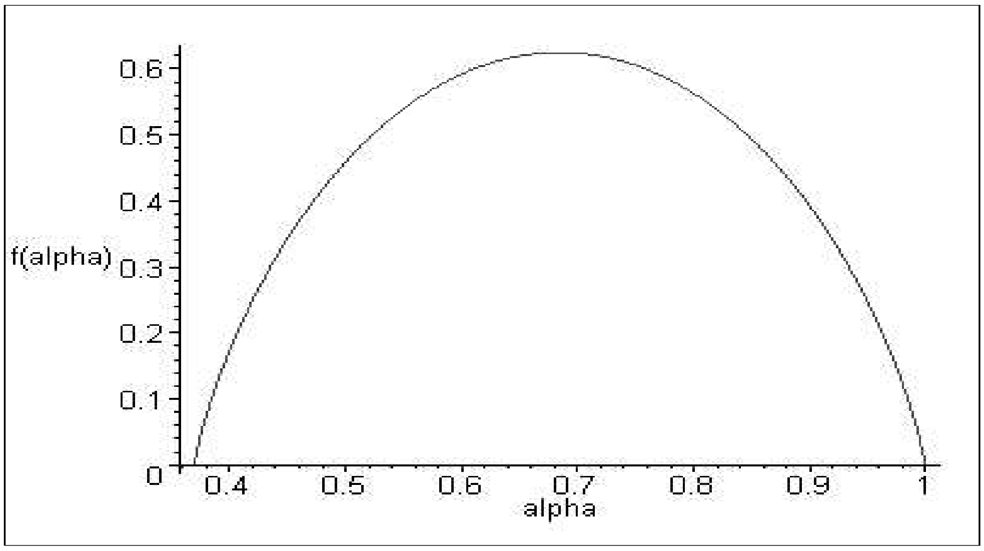


Figure 4.3: The $f(\alpha)$ Curve of The Cantor Measure

$$\mu_2[1/2,3/4] = m_1 m_0, \quad \mu_2[3/4,1] = m_1 m_1 \tag{4.7}$$

Iteration of this procedure generates an infinite sequence of measures. At step $k+1$, we assume the measure μ_k has been defined and μ_{k+1} is defined as follows: Consider an arbitrary interval $[t, t+2-k]$, where the dyadic number t is of the form:

$$t = 0.\eta_1\eta_2\ldots\eta_k = \sum_{i=1}^{k} \eta_i 2^{-k} \tag{4.8}$$

in the counting base $b = 2$. We uniformly spread a fraction of m_0 and m_1 of the mass $\mu_k[t, t+2-k]$ on the subinterval $[t, t+2-k-1]$ and $[t+2-k-1, t+2-k]$. A repetition of this scheme to all subintervals determines μ_{k+1}. The measure μ_{k+1} now is well defined. Figure 4.4 shows the measure for the binomial multiplicative process after $k = 11$ with $m_0 = 0.25$ and m_1= 0.75.

The construction of binomial multifractal can be extended in several ways. First, at each stage of the cascade, intervals can be divided not in 2 but in $b > 2$ intervals of equal size. This defines the class of *multinomial multifractals*. Second, the allocation of mass between subintervals at each step of cascade can be randomized by using a

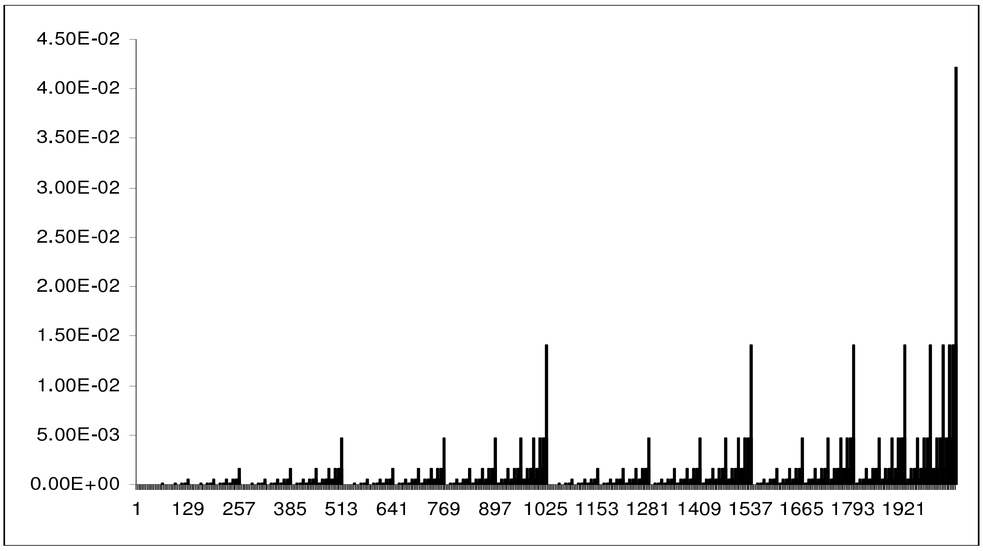


Figure 4.4: Binomial Measure with m_0 = 0.25 and m_1= 0.75

random variable as the multiplier. This defines *random multifractals.* Although the multipliers need not to be discrete, we shall use discrete ones for simplicity.

4.2.4 Probabilistic Roots of Multifractals

Because multifractal measures can be generated by, or mapped onto, multiplicative cascade, the coarse Hölder exponent can be expressed as a sum of random variables by definition [Man89]. The behavior of sums of random variables is a central topic in probability theory. There are three theorems dealing with such sums: *the Law of Large Numbers (LLN)*, *the Central Limit Theorem (CLT)*, and *the Large Deviation Theorem (LDT).* The LLN says that almost surely (with probability of 1) the sample average will converge to the expectation when k increases to infinity. The LLN guarantees the existence of α_0 and its role as the most probable Hölder exponent. But the LLN only holds in the limit $\delta \to 0$, whereas we are often dealing with a finite number of multiplicative steps k. Thus, the deviation from the expected value becomes important for finite k. The relevant information is yielded by the CLT and, far more important, by the LDT. The CLT is concerned with small fluctuations around the expected value. In this context, it shows that the appearance of a quadratic maximum in the $f(\alpha)$ of the binomial measure is not a coincidence. Consider a random variable with finite expectation EX and $Pr(X > EX) > 0$. The large deviation theory is concerned with very large fluctuations around the expected value, namely the behavior of

$$\lim_{k\to\infty} Pr\{\frac{1}{K}\sum_{h=1}^{k} X_h - EX \geq \delta\} \tag{4.9}$$

as a function δ and k. The LLN tells us that,

$$\lim_{k\to\infty} Pr\{\frac{1}{K}\sum_{h=1}^{k} X_h - EX = 0\} = 1 \tag{4.10}$$

So for $\delta = 0$, the above quantity vanishes with speed 0. For all other δ, we expect $Pr\{\frac{1}{K}\sum_{h=1}^{k} X_h - EX \geq \delta\}$ to converge to 0 as k increases to infinity. The question is, "How fast does it vanish?". The LDT states that it not only converges to 0, but also does so exponentially fast. In this section, we omit some details, but generally speaking, $f(\alpha)$ can be deduced via the large deviation theory and this provides a probabilistic basis for multifractals [EM92, Kes01]. Furthermore, large deviation theory in the continuous and/or unbounded cases exists as well, providing a full justification of the so-called *thermodynamic formalism of multifractals.* We refer the reader to for more details [EM92, Man89, Har01, Kes01].

4.2.5 Thermodynamics Formalism of Multifractals

There are more than one way to get to the multifractal spectrum $f(\alpha)$. An alternative method is the method of Moments in which we first define partition function, analogous to the partition function in thermodynamics and statistic physics [EM92, Man89],

$$Z_q(\delta) = \sum_{i=1}^{N(\delta)} \mu_i^q = \sum_{i=1}^{N(\delta)} (\delta^{\alpha_i})^q \tag{4.11}$$

Denote the number of boxes for which the coarse Hölder exponents satisfied $\alpha < \alpha_i < \alpha + d\alpha$ by $N_\delta(\alpha)d\alpha$. The contribution of the subset of boxes with α_i between α and

$\alpha + d\alpha$ to $Z_\delta(\alpha)$ is $N_\delta(\alpha)(\delta^\alpha)^q d\alpha$. Integrating over $d\alpha$ we obtain,

$$Z_q(\delta) = \int N_\delta(\alpha)(\delta^\alpha)^q d\alpha \tag{4.12}$$

If $Z_\delta(\alpha) \sim \delta^{-f(\alpha)}$, it follows that

$$Z_q(\delta) = \int \delta^{q\alpha - f(\alpha)} d\alpha \tag{4.13}$$

By keeping only the dominant contribution in the equation, and introducing

$$\tau(q) = q\alpha(q) - f(\alpha(q)) \tag{4.14}$$

, the partition function will scale like $Z_q(\delta) \sim \delta^{\tau(q)}$. It is easy to see that

$$\frac{d\tau(q)}{dq} = \alpha(q) \tag{4.15}$$

This means that $f(\alpha)$ can be computed from $\tau(q)$ and vice versa. The relation between $f(\alpha)$ and $\tau(q)$ is called a *Legendre transform* [Man89]. An interesting consequence is that *flexibly rich thermodynamic content is hidden in the concept of multifractals.* From preceding discussion, we can easily draw a correspondence between $Z(q)$ and the thermodynamic partition function $Z(\beta)$, between q and the temperature T (as the inverse of T), between α and the energy, between $f(\alpha)$ and the entropy, and between $\tau(q) = q\alpha - f(\alpha)$ and the *Gibbs free energy* $G = H - TS$. For more information on this topic, the interested reader is referred to [Man89].

4.3 Bayesian Networks as Random Multinomial Multifractals

A Bayesian network [Pea88] is a Directed Acyclic Graph (DAG) in which nodes represent random variables and arcs represent conditional dependence relationships among these variables. Each node X_i has a conditional probability table (CPT) that contains probabilities of a variable value given the values of its parent nodes, denoted as $\pi(X_i)$. A BN represents the exponentially sized joint probability distribution (JPD) in a compact manner. Every entry (an instantiation of all nodes) in the JPD can be computed from the information in the BN by the chain rule:

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i|\pi(x_i)) \tag{4.16}$$

From the multifractal viewpoint, the JPD defined by a BN with n nodes is a measure of belief distributed on an n-dimensional space of random events. Given a topological ordering of all nodes, we can map the n-dimension space to a linear interval by assigning each event an integer number as its address on that interval. For example, the linear interval for an 8-node binary BN is $[0, 255]$. The JPD of a BN can be considered as being generated from a multiplicative cascade in which number of steps n equals to the number of nodes. At each step of the cascade, intervals are divided into b subintervals, where b is the number of states of the current node, and the multiplier for allocating the probability is a random variable defined by the CPT of the current node. It is easy to see that, in the most general case, a BN corresponds to a multifractal generated by a random multinomial multiplicative cascade. The simplest multifractal - the binomial measure - corresponds to the simplest BN - a binary BN without links. Consider such an 8-node binary BN in which each node has a prior probability distribution of $(0.25, 0.75)$. The cascade contains 8 steps and generates a JPD of 256 instantiations as shown in Figure 4.5. This is actually the simplest multifractal - the binomial measure.

Now let us consider the process of an agent's incremental understanding of some uncertain domain as a multiplicative cascade process. At the beginning, the agent first identifies all random variables. Before it knows anything about the causal relationships between these variables, it has to assume a uniform distribution spreading belief evenly to all states. The agent's belief is redistributed as it learns more about the domain; i.e., the connections between nodes and the CPT values. The process of belief redistribution is a typical multiplicative cascade process similar to any other multiplicative cascade in the context of multifractals. For example, a turbulence cascade model describes the nature of energy dissipation in a turbulent fluid flow. With turbulence, the energy is introduced into the system on a large scale (storms, or stir-

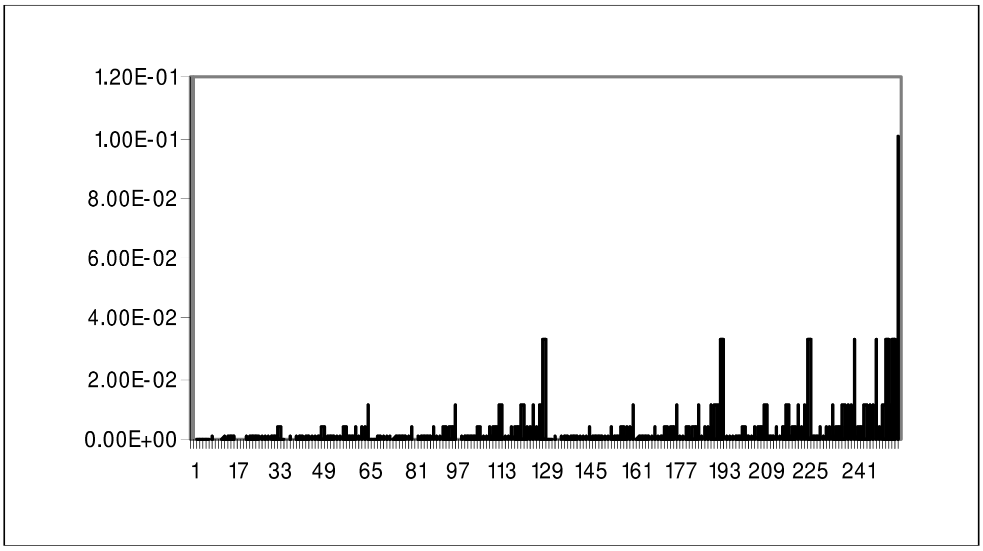


Figure 4.5: JPD of the Simplest BN with 8 Nodes

ring a bowl of water), but can only be dissipated in the form of heat on very small scales where the effect of velocity, or friction between particles, becomes important. Cascade models assume that energy is dissipated through a sequence of eddies of decreasing size until it reaches sufficiently small eddies where the energy is dissipated as heat. In the case of Bayesian networks, the belief is introduced to the domain from a high level as a uniform distribution. As we learn the CPTs, we capture the increasingly refined causal structure of the domain. These substructures keep redistributing our belief until we learn all about the domain.

4.4 Case Study: Asia and $ALARM13$

In this section, we study the multifractal properties of the joint probability distributions of two example networks: Asia and $ALARM13$.

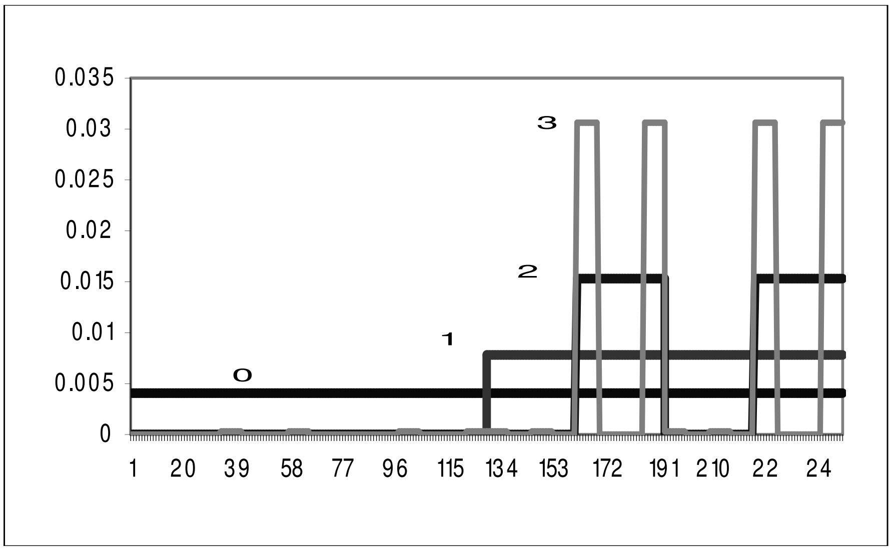


Figure 4.6: JPDs at the First Three Steps of the Multiplicative Cascade of the Asia Network

4.4.1 The Self-similarity of the JPD: Asia

We first use the Asia network [Nea90] to demonstrate the process of an agent's incremental understanding of an uncertain domain as a binomial multiplicative cascade process, and to show the JPD's self similarity property. Asia is a small binary BN containing 8 nodes, so its JPD has a total of $2^8 = 256$ instantiations.

Figure 4.6 shows the changing of the JPD during the first three steps of the cascade process of Asia network; i.e., after the CPTs of node VistAsia, Smoking and Tuberculosis are learned sequentially. From Figure 4.6, we can clearly see how the belief is redistributed as more knowledge about the domain is learned.

Figure 4.7 is the final JPD after all nodes' CPTs are learned and Figure 4.8 is the second half of the final JPD. Comparing these two figures, we can see the stochastic and self-similarity properties of the JPD.

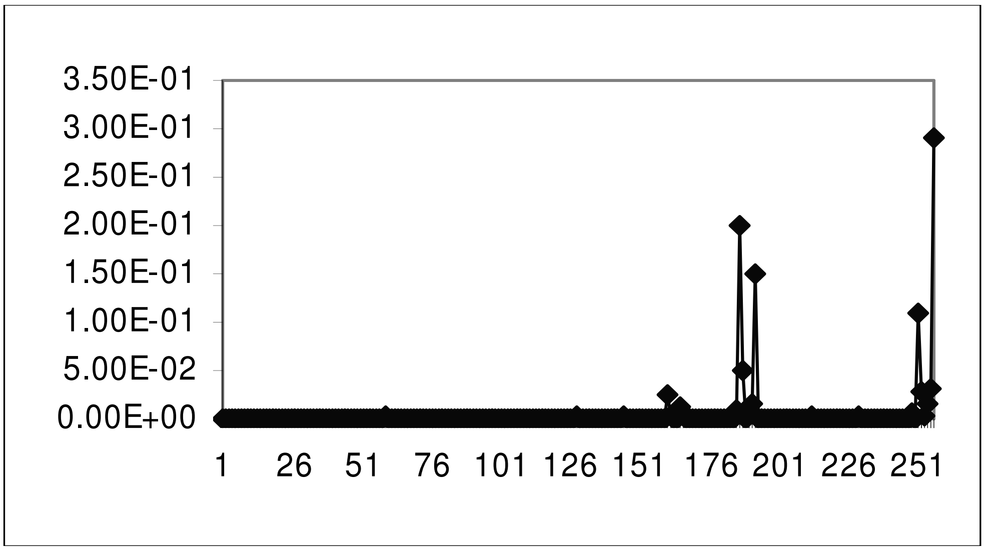


Figure 4.7: JPD of the Asia Network

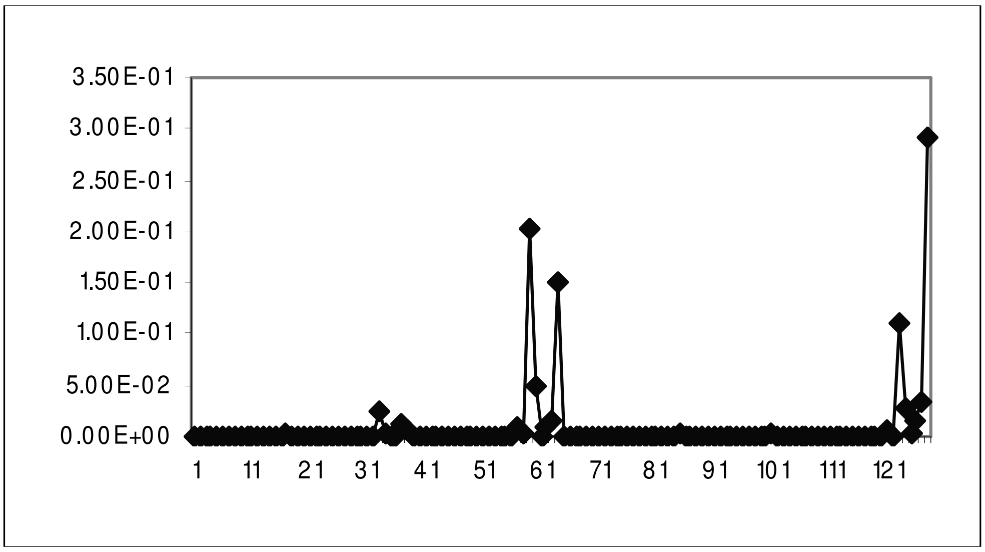


Figure 4.8: Second Half of the JPD Illustrating the Self-similarity Property

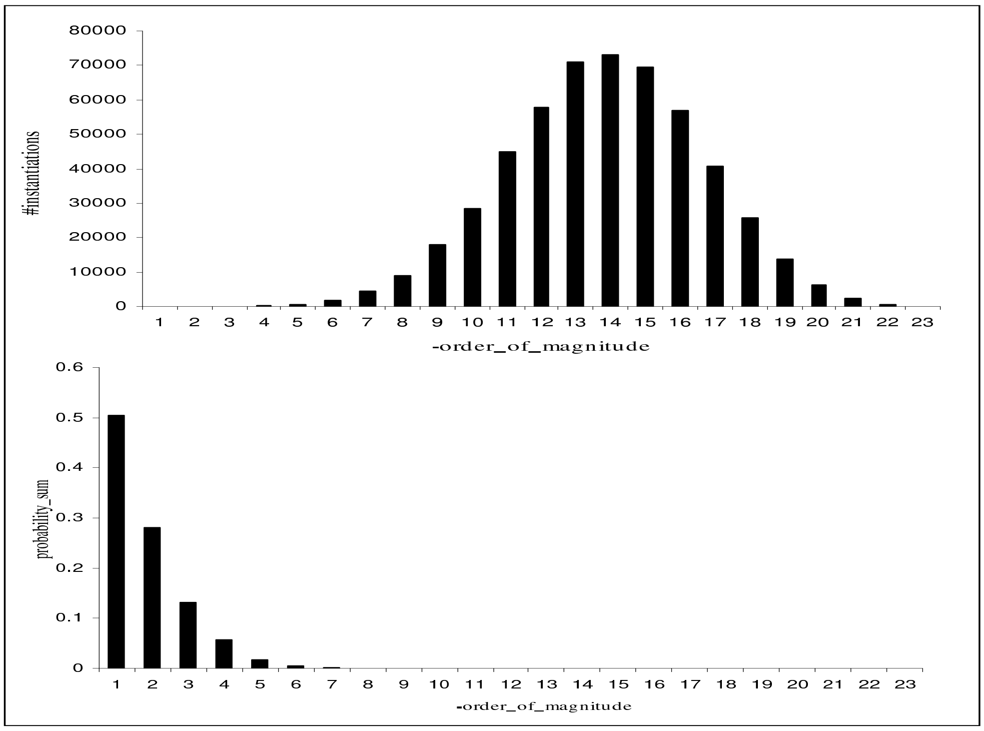


Figure 4.9: (1) Number of Instantiations At Each Order (2) Probability Sum of Instantiations At Each Order

4.4.2 The Skewed JPD: $ALARM13$

In the following we analyze the JPD of the $ALARM13$ [Dru94], a subset of the $ALARM$ network, to demonstrate its multifractal characteristic and clustering property. $ALARM13$ was the same network analyzed in [Dru94]. It contains 13 variables, resulting in 525,312 non-zero states. The probabilities of these states were spread over 23 orders of magnitude. Figure 4.9 shows the histograms of the number of instantiations distributed at each order and their contribution to the total probability space. The X-axis is the negative order of magnitude in both figures. Figure 4.9.1 shows that the histogram of number of instantiations at each order appears to be a normal distribution. Given the logarithmic scale of the X-axis, it shows that the

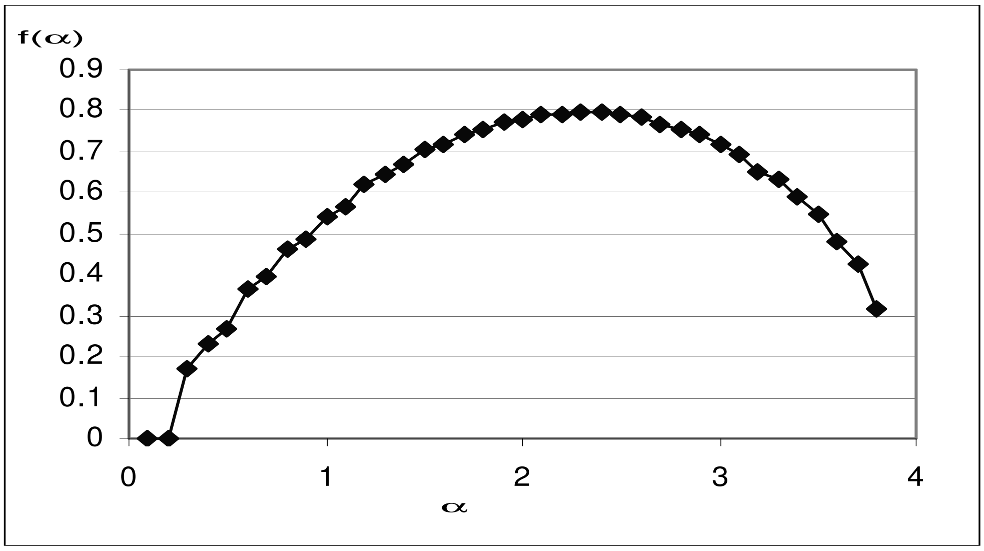


Figure 4.10: The f(α) Curve of the $ALARM13$'s JPD

actual distribution is a lognormal. The peak of figure 4.9.1 is at the order of 10^{-14}. It contains 73,256 instantiations, but its contribution to the total probability space is only $2.9E-09$. From Figure 4.9.2 we can see that high-probability instantiations, although few in number, dominate the joint probability space. Of all instantiations, there is one with probability around 0.505, 10 with probabilities between $[0.1, 0.01]$ and the total probability of 0.28, 48 with probabilities between [0.01, 0.001] and the total probability of 0.13, 208 with probabilities between [0.001, 0.0001] and the total probability of 0.058. The 267 most likely instantiations (0.05% of the total of 525,312) covers 97.45% of the total probability space. This highly skewed result has been analyzed by Druzdzel [Dru94]. In the following we show the multifractal structure of the JPD and the way instantiations at different orders of magnitudes fill the space.

4.4.3 The Multifractal Spectrum of the $ALARM13$

Applying multifractal analysis to $ALARM13$'s JPD, we get its multifractal spectrum, the $f(\alpha)$ curve, as shown in Figure 4.10. The X-axis is the coarse Hölder exponent

α, Y-axis is $f(\alpha)$ - the fractal dimension of the subset of all instantiations with the same α. This $f(\alpha)$ curve confirms that the JPD of a Bayesian network is a multifractal. It describes how these instantiations fill the probability space from the point of view of fractal dimension. We can see in Figure 4.10 that high-probability instantiations (corresponding to small α) have a low dimension, which means that they fill the probability space in a very "sparse" way; i.e., there are clusters of high-probability instantiations. The peak of Figure 4.10 has a fractal dimension of 0.79, and the corresponding coarse Hölder exponent α is around 2.5. By the definition of α, this corresponds to instantiations with probability on the order of 10^{-15}. These instantiations are around the peak of Figure 4.9.1. This means that these instantiations fill the probability space in a very "dense" way; i.e., they are almost all over the space. Finally, instantiations with very low probabilities ($\alpha = 3.8$) also have low dimensions ($f(\alpha) = 0.32$). Again, this means that low-probability instantiations (rare events) distribute sparsely as well, and clusters of them can be expected. This yields a mathematical description of the inner structure of the JPD: *there are clusters of high-probability instantiations and low-probability instantiations, but instantiations in the middle are distributed almost all over.* Interestingly, this pattern coincides with the way that people live in the real world; i.e., high-income people tend to live in the same community, as do low-income people, but the middle-class are located all over.

4.4.4 Quantifying the Clustering Property

To show the clustering property more clearly, we draw the distribution of high-probability instantiations and the distribution of low-probability instantiations in Figure 4.11. Figure 4.11.1 contains all instantiations with a probability higher than 0.0001 in which X-axis is the "address" of instantiations ranging from 0 to 525,312 and Y-axis is the actual probability value. Figure 4.11.2 contains all instantiations lower than 10^{-20} in which Y-axis is the "address" of instantiations and X-axis is just the series number of each instantiation (note we use a different X-Y here because the actual values are too small to be drawn neatly). We can see clearly there are clusters

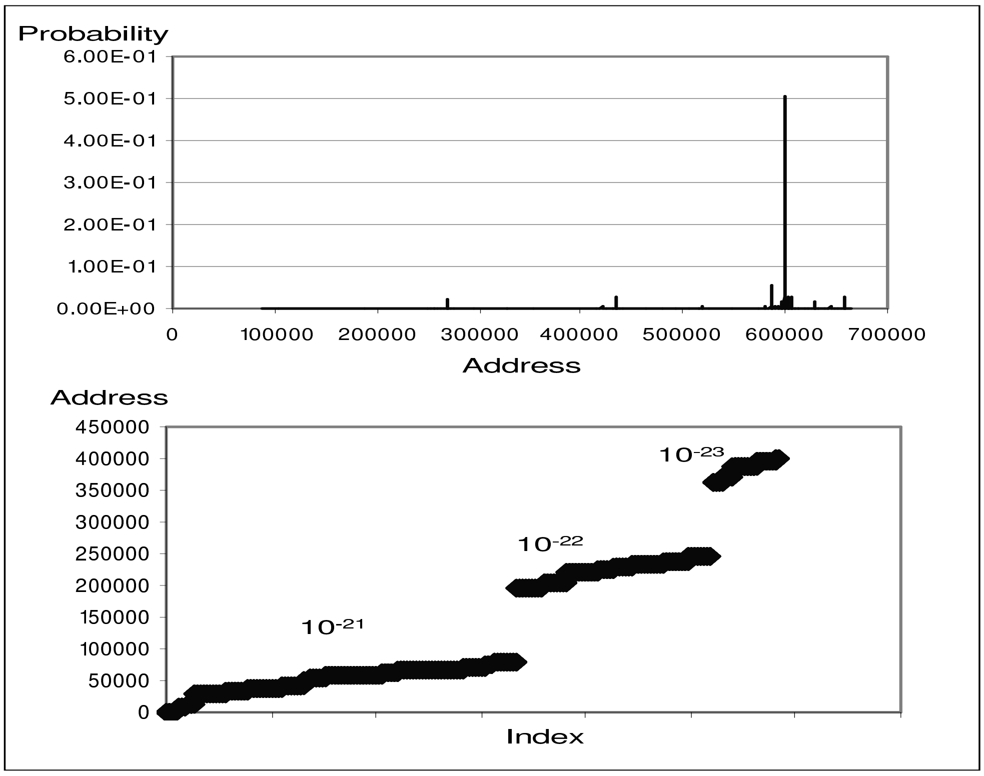


Figure 4.11: (1) The Clusters of High Probability Instantiations (2) The Clusters of Low Probability Instantiations

in both graphs.

Having shown the clustering property, the next thing we want to do is to quantify this property. We use the *Hamming Distance* between bit string representations of two instantiations to measure how far they are located from each other. An instantiation is represented as a bit string "$b_1b_2 \ldots b_n$" where n is the number of variables in the domain and b_i is the state index of each variable. For example, the Hamming distance between instantiation "00001100" and "00100001" is 4. High-probability instantiations should have small Hamming distances between each other to be a cluster, because of the clustering property. We draw the *Averaging Hamming distance (AHD)* graph for the most likely instantiation in Figure 4.12(a). The X-axis is the negative of the order of magnitudes; the Y-axis is the AHD between the most likely instantiation and all instantiations at each order of magnitude. From Figure 4.12(a) we can see that the instantiations with lower probabilities have a larger Hamming distance from the most likely instantiation; i.e., they locate far away from the most likely instantiation. We also draw the same figure for the lowest instantiations in Figure 4.12(b). In Figure 4.12(c) we put together 23 AHD graphs for instantiations of all orders to provide a global picture. Figure 4.12(c) consists of 23 segments of curves corresponding to 23 orders of magnitudes. Each curve consists of 23 points and represents the AHD graph of a randomly picked instantiation at that order. For example, the first segment in Figure 4.12(c) is Figure 4.12(a), and the last segment is Figure 4.12(b). From Figure 4.12(c) we can see that the instantiations in the middle order of magnitudes are located at almost the same distance from all other orders ($7 < AHD < 9$); i.e., they can be found at almost all places. This finding supports our previous analysis of the expected distribution pattern.

4.5 A Multifractal Search Algorithm for Finding the MPE

The JPD's multifractal property can be used as a *meta-heuristic* to develop new search algorithms for finding the MPE. Since the search space is a multifractal, both good and bad solutions will be clustered together. Hence the search should be divided into two phases: first identify the *"good communities"*, then localize the search to these regions. Also, the *quality of the community* should be evaluated along with the current search point in order to direct the search. If point A is better than point B, but B's neighbors are better than A's, we should move B to look for the global optimal. This helps the searcher escape the local optimal where simple hill climbing gets stuck. Consider the following scenario: suppose we are asked to find the best house in a city. We would first drive around to identify several areas that look good. Then we would intensively search each area. When the neighborhoods get better, our chance of hitting the best house gets higher too.

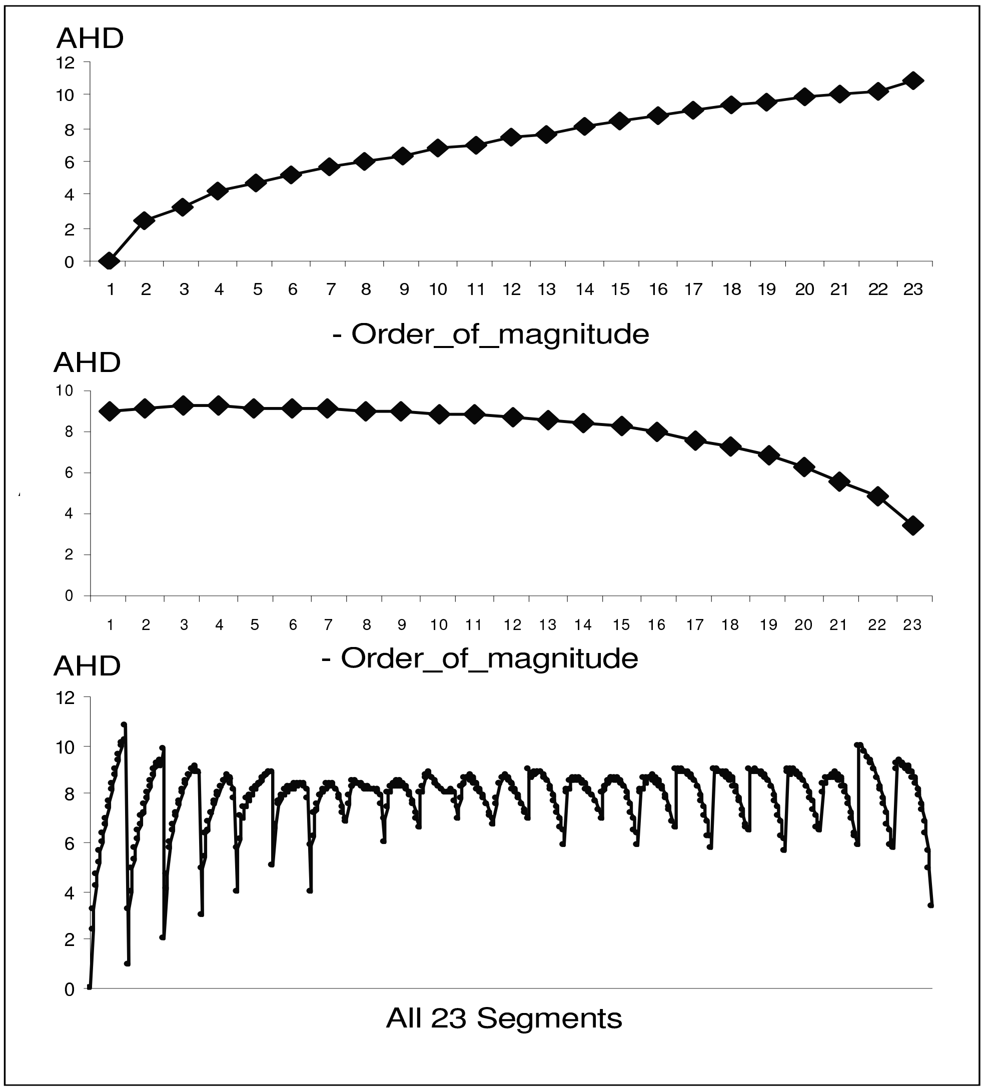


Figure 4.12: The Average Hamming Distance Graphs

Input: A BN (G, P) and an evidence set E.

Output: A complete assignment $u = (u_1, \ldots, u_n)$.

Step 1: Use the sampling algorithm to generate a set of initial good points S.

Step 2: For each point in S, start a hill climbing search using **Neighborhood Quality** as the evaluation function, then put all local optima into S^*.

Step 3: For each point in S^*, start a normal hill climbing search and return the best solution so far as the MPE.

Figure 4.13: Hybrid Random Sampling And Search Algorithm for Finding The MPE

We have developed a two phase Sampling-and-Search algorithm to solve the MPE problem, based on these meta-heuristics. This algorithm is designed to find the most probable explanation (a complete assignment) given the observed evidence. The general MPE problem is NP-complete [Shi94] and hard to approximate [AH98]. In the first phase of the algorithm, forward sampling (or any other feasible method) is used to quickly identify a set of good communities. In the second phase, a hill climbing search using *Neighborhood Quality* as the evaluation function is started for each community. An additional "repair" phase can be added by using a set of "elite solutions" and a set of "worst solutions" collected during the search process. These would refine the final solutions by flipping the variable values that do not agree with the majority "elite solutions", or those that agree with the most "worst solutions". When the stop rule is satisfied, it returns the best solution so far as the MPE.

The algorithm's performance is determined by two factors: how reliably the sampling algorithm brings the searcher to places close to the global optimal, and the strength of Neighborhood Quality, the evaluation function, to bring the searcher from a near optimal place to the global optimal. The Neighborhood Quality of a search point is defined as the sum of the likelihoods of all its nearest neighbors. It can be approximated by randomly drawing samples from its neighbors. The sampling radius can be set to a small positive value k.

We expected that the skewness of the CPTs would have an influence on the performance of the algorithm. Therefore, in our experiments we randomly generated three groups of networks with different degrees of CPT skewness to test the algorithm: skewed, normal, and unskewed. The skewness of the CPTs is computed as follows: [JN96]. For a vector (a column of the CPT table), $v = (v_1, v_2, \ldots, v_m)$, of conditional probabilities,

$$skew(v) = \frac{\sum_{i=1}^{m} |\frac{1}{m} - v_i|}{1 - \frac{1}{m} + \sum_{i=2}^{m} \frac{1}{m}} \tag{4.17}$$

The skewness for the CPT of a node is the average of the skewness of all columns. The skewness of the network is the average of the skewness of all nodes. The skewness of these three groups of networks were set to around 0.9, 0.5, and 0.1 respectively. Each group consisted of 20 networks with binary nodes. The number of nodes were 100, and the number of edges were $120 \sim 150$. These networks were set to be sparse enough so that the exact MPEs could be computed. We randomly generated 10 evidence values for each network; hence, the size of search space is 2^{90}. We used Hugin to compute the exact MPEs. For each group of networks, we counted the number of times when exact MPEs were found. We also computed the average relative error (ratio of the absolute error to the exact MPE value) and recorded the average Hamming distance between the returned MPE and the exact MPE. Table 1 summarizes the experimental results. From the results we can see that normally-skewed networks are the easiest ones for the algorithm, and unskewed networks are the hardest. In 20 normally-skewed networks we were able to find exact MPE in 19; and the one missed is very close to the global optimal (only 2 bits difference out of 100). Of 20 unskewed networks we were able to find exact MPE in only 4 of them. The average error and average Hamming distance between the returned MPE and the exact MPE are also the largest ones. These results imply that if the network is unskewed (most distributions are nearly uniform), finding MPE will be hard because the search space

Table 4.1: Results on Randomly Generated Networks

	#solved	**error**	**AHD to exact**
skewed	12/20	0.0242	1.25(25/20)
normal	19/20	0.0029	0.1(2/20)
unskewed	4/20	0.0798	3.9(78/20)

is flat. On the other hand, if it is highly skewed it will also bring trouble to the search algorithm because of the attractiveness of these steep local optimums.

4.6 Summary

We have demonstrated that the underlying JPDs of Bayesian networks are multifractals created by random multiplicative cascade processes. The JPD that has many orders of magnitude differences in probabilities of various instantiations is not only highly skewed, but also stochastically self-similar and exhibits clustering properties. The multifractal spectrum of the JPD describes how instantiations at different orders fill the joint distribution space with different fractal dimensions. In particular, both high and low probability instantiations tend to form clusters in the joint distribution space. Even though we discussed the model as a whole, the result will hold for its self-contained parts as well. we also hypothesize that it holds for dynamic models. The $f(\alpha)$ curve will show up as long as a random multiplicative cascade process is involved. The significance of this analysis is that it provides important information about characteristics of the joint probability distribution. Particularly, the clustering property can be a very useful meta-heuristic for searching the MPE. This research also bridges an analytical gap between multifractal and BNs and suggests some very interesting research directions. As we have seen, multifractals have a deep probabilistic root and a rich thermodynamic content. The fact of BN being a multifractal draws our attention to connection between thermodynamics and BNs [YFW00, PA02]. Also, because the MPE problem is NP-Complete, we should also expect to observe the same multifractal structure in the solution space of other hard combinational problems such as MAXSAT and TSP. Applying the multifractal metaheuristic to solve these problems would be a very interesting topic to investigate in the future.

The JPD's multifractal property also provides a potential instance hardness measure to be used in algorithm selection for the MPE problem. One possible future direction is to study how the multifractal property of a Bayesian network's JPD influences various MPE algorithms' relative performance. If the multifractal property can differentiate the MPE algorithm space very well, then we can try developing an efficiently computable measure to characterize it and use it as a predictive measure for the corresponding MPE algorithm's performance.

Chapter 5

Machine Learning Techniques

In this chapter, we review some machine learning techniques that will be used to construct algorithm selection systems for sorting and the MPE problem. This includes data preprocessing methods such as *feature selection* and *discretization*; machine learning algorithms such as *decision tree learning*, *the naive Bayes classifier*, and *Bayesian network learning*; some basic methods to evaluate the learned models such as *cross-validation* and *confusion matrix*; and three meta-learning schemes: *bagging*, *boosting*, and *stacking*. Finally, we give an overview of the process of applying machine learning techniques to solve the algorithm selection problem.

5.1 Machine Learning and Data Mining

5.1.1 Introduction

Our goal is to experimentally collect training data on problem characteristics and algorithm performance, and to induce a predictive algorithm selection model from the training data. This is a typical task in *machine learning* [Mit97] and *data mining* [WF99].

The goal of machine learning is to build computer programs that improve automatically with experience. Dating mining is about automatically analyzing data and discovering valuable implicit patterns and regularities. For example, one data mining application is the process of discovering the customer loyalty pattern from a database of customers' choices along with customers' profiles. Data mining involves machine learning in a practical way. Many machine learning algorithms have proven to be of great practical value in data mining. More precisely, we define machine learning as follows [Mit97]:

Definition 19 (Machine Learning) *A computer program is said to "learn" from experience E with respect to some class of tasks T and performance P, if its performance at tasks in T, as measured by P, improves with experience E.*

Machine learning can be either *supervised* or *unsupervised*. In supervised learning, there is a specified set of classes and each example of the experience E is labelled with the appropriate class. The goal is to generalize from the examples so as to identify to which class a new example should belong. This task is also called *classification*. In contrast to supervised learning is unsupervised learning. The goal here is often to decide which examples should be grouped together, i.e., the learner has to figure out the classes on its own. This is usually called *clustering*. In this thesis, we will be concerned with supervised learning.

5.1.2 Example: the Weather Problem

In a typical supervised machine learning problem, the experience E, or the data, is represented as a set of *training examples* or *instances*. Each example is described by a fixed number of *features*, or *attributes*. Features typically take two types of values: nominal or numeric. Among all of the features, there is a special one that serves as a label denoting the class of this example. The task T is to classify examples. The performance P is simply the classification accuracy. There is usually another set of examples called *testing examples*. The training examples are used to produce the learned model and the testing examples are used to evaluate the classification accuracy. When testing, the class labels are not presented. The algorithm takes as input a test example, and returns as output the class label for that example.

Now let us look at a simple example: the *Weather* problem. The dataset is shown in Table 5.1. It contains 14 examples describing the weather of days. Each example

Table 5.1: The Weather Problem

Instance#	**Features**				**Class**
	Outlook	Temperature	Humidity	Windy	Play
1	sunny	85	85	FALSE	no
2	sunny	80	90	TRUE	no
3	overcast	83	86	FALSE	yes
4	rainy	70	96	FALSE	yes
5	rainy	68	80	FALSE	yes
6	rainy	65	70	TRUE	no
7	overcast	64	65	TRUE	yes
8	sunny	72	95	FALSE	no
9	sunny	69	70	FALSE	yes
10	rainy	75	80	FALSE	yes
11	sunny	75	70	TRUE	yes
12	overcast	72	90	TRUE	yes
13	overcast	81	75	FALSE	yes
14	rainy	71	91	TRUE	no

has five attributes: Outlook, Temperature, Humidity, Windy and Play. "Play" is the class attribute denoting whether the day is a good one for sports or not. The task T is to learn a model from the data and to use the model to predict whether a given day being a good day for sports is "yes" or "no".

5.1.3 The Learning Problem for Automatic Algorithm Selection

In our algorithm selection problem, the task T is to learn how to predict the best algorithm for an arbitrary instance according to its features. The target function we want to learn is a discrete-valued function $V : F \rightarrow A$, which maps any problem instance characteristic vector to its best algorithm. The experience E is the data collected from algorithmic experiments. The performance P is the accuracy of algorithm selection. We state the algorithm selection learning problem as follows:

The algorithm selection learning problem:
Task T: selecting the best algorithm performance.
Measure P: percent of correct algorithm selection predictions.
Training experience E: results collected from algorithmic experiments.

Since the target function, $Best\_Algorithm(f)$, has discrete values, this kind of task is often referred to as a *classification problem.* The task of such problems is to classify the examples into one category from a discrete set of possible categories. In our research, the set of categories is the set of all candidate algorithms.

One perspective on machine learning is viewing learning as the process of searching a very large space of all possible *hypotheses* in order to determine the one that best fits the available training data and any prior knowledge held by the learner [Mit97]. According to this perspective, our hypotheses space consists of all possible dependency relationships between problem characteristics and algorithm performance. A hypothesis space can be defined by more than one *representation*, such as linear functions, logical descriptions, decision trees, artificial neural networks, and so on. For each of these hypothesis representations, the corresponding learning algorithms takes advantage of a different underlying structure of the search space. Throughout this thesis, we will mainly consider three representations and the corresponding learning algorithms: *decision tree* [Qui86, Mit97], *the naive Bayes classifier* [Mit97, JL95], and Bayesian networks [Pea88, Nea90, RN95].

5.2 Machine Learning Algorithms

5.2.1 Decision Tree Learning

Decision tree learning is one of the most popular inductive learning methods. It has been applied to a broad range of tasks from medical diagnosis to credit risk assessment. In decision tree learning, the learned function is represented by a decision tree. Decision trees are essentially sets of "if-then" rules. They classify training examples by sorting them down the tree from the root to some leaf node, which

```
ID3(Examples, Target, Attributes)
    Create a root node;
    **If** all Examples have the same Target value, give the root this label;
    **Else if** Attributes is empty label the root according to the most common
    value;
    **Else begin**
      Calculate the information gain for each attribute, according to the
       average entropy formula;
      Select the attribute, A, with the lowest average entropy (highest
       information gain) and make this the attribute tested at the root;
      **For** each possible value, v, of this attribute
         Add a new branch below the root, corresponding to A = v;
         Let Examples(v) be those examples with A = v;
         **If** Examples(v) is empty, make the new branch a leaf node labelled
          with the most common value among Examples;
         **Else** let the new branch be the tree created by
          ID3(Examples(v), Target, Attributes - A);
    **end begin**
**end**
```

Figure 5.1: ID3 Algorithm for Decision Tree Learning

provides the classification of the example. Each node in the tree represents a test of some attribute of the training example, and each branch corresponds to one of the possible values for its source node (attribute).

In the language of logic, decision trees represent a *disjunction of conjunctions of constraints on the attribute values of training examples.* Each path from the root node to a leaf corresponds to a conjunction of attribute tests. The whole tree is a disjunction of these conjunctions.

Decision Tree Learning Algorithms: ID3

ID3 and its successor, C45, are two of the most important algorithms for learning a decision tree from data. ID3 was developed by Quinlan in 1986 [Qui86]. It was extended to C45 later in 1993 [Qui93].

The basic algorithm, ID3, learns a decision tree by constructing it top-down. At each node, the following question is asked: "which attribute should be tested here?" The question is answered using a statistical test to determine how well an attribute alone classifies the training examples. The best attribute is then selected and used as the test at the current node of the tree. A descendant of the node is created for each possible value of this attribute. The training examples are also sorted to the appropriate descendant node, i.e., down the branch corresponding to the example's value for this attribute. The entire process is repeated using the training examples associated with each descendant node to select the best attribute to test at that point of the tree. Attributes that have been incorporated higher in the tree are excluded. Therefore any attribute can appear at most once along any path through the tree. This process continues for each new leaf node until: (1) every attribute has already been included along this path through the tree, or (2) all training examples associated with this leaf node all have the same target attribute values, i.e., they all belong to the same classification.

We can see that ID3 is essentially a top-down, greedy algorithm searching through the space of all possible decision trees because it never looks back to reconsider previous choices.

The most important issue in the ID3 algorithm is how to select which attribute to test at each node in the tree. Intuitively, we want to select the attribute that is most useful for classifying examples at this point. In order to measure this statistical property, we will need to define *information gain* that measures how well a particular attribute classifies the training examples according to their target attribute values. We begin by defining *entropy* in order to define information gain precisely.

Definition 20 (entropy) *Suppose S is a collection of training examples. Each example has several attributes and one of them is called the target attribute that can take on k different values. The entropy of S relative to this k-wise classification is define as*

$$Entropy(S) = \sum_{i=1}^{c} -p_i \log_2 p_i \tag{5.1}$$

Information gain, a measure of the effectiveness of an attribute in classifying the training examples, is simply the reduction in entropy caused by partitioning the examples according to this attribute.

Definition 21 (Information Gain) *Suppose S is a collection of examples and A is an attribute of the examples. The information gain $Gain()$ of an attribute A relative to S is defined as*

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{S_v}{S} Entropy(S_v) \tag{5.2}$$

Inductive Bias in ID3

Given a collection of training examples S, there are typically many decision trees that are consistent with S. Because of ID3's simple-to-complex greedy search strategy, ID3 chooses the first acceptable tree it sees in its search process. We say that ID3's *inductive biases* are: (1) in favor of shorter trees over longer ones, (2) in favor of trees that place the attributes with high information gain closest to the root. ID3's inductive bias is partly justified by the famous **Occam's Razor** which states that *"Entities should not be multiplied unnecessarily"*. In the language of machine learning, it means that *"Prefer the simplest hypothesis that fits the data"*. Another argument is as follows: because there are fewer short hypotheses than long ones, it is less likely that a short hypothesis coincidentally fits the data. Debates over these arguments remain unsolved even today.

C4.5 Extensions

A variety of extensions to the basic ID3 algorithm has been developed and the resulting algorithm is called C4.5 [Qui93]. These extensions [WF99] include *incorporating numeric attributes*, *handling missing values*, and *avoiding overfitting by rule post-pruning.*

Numeric attributes Many real world applications provide numeric data. C4.5 discretizes the numeric attribute A by picking a threshold that produces the greatest information gain. The procedure for binary splits is as follows: First, sort all examples according to the numeric attribute A. Then, identify adjacent examples that differ in their target classification and generate a set of candidate threshold in between the corresponding values of A. It has been shown in [Fay91] that the value of the threshold that maximizes information gain must always lie at such a boundary. Finally, these candidate thresholds are evaluated by comparing the information gain associated with each and the best one is used to discretize A into a binary value. Extensions that spilt numeric attributes into multiple intervals rather than binary are discussed in [FI93].

Missing values Some dataset may contain missing values for certain attributes. One strategy to deal with this issue is to assign it the most common value at the current node. A more complex procedure [Mit97, WF99] is to assign a probability to each of the possible values of A. These probabilities can be estimated based on the observed frequencies of the values of A at the current node. The example with missing values is then converted into fractional examples that can be used to compute information gain.

Pruning In some cases when there is noise in the data or when the number of examples is too small to be a representative sample of the true target function, ID3 can produce trees that *overfit* the training examples. Overfitting is defined as follows:

Definition 22 (Overfit) *A hypothesis, h, is said to overfit the training data if there exists another hypothesis, h', such that h has smaller error than h' on the training data but $h\prime$ has smaller error on the the entire test data than h.*

Rule post-pruning [Mit97] is one technique to avoid overfitting the data. It involves the following steps:

1. Infer the decision tree from the training data (allowing overfitting).
2. Convert the learned tree to a set of if-then rules
3. Prune each rule by removing any preconditions that result in improving its estimated accuracy.

4. Sort the pruned rules by their estimated accuracy, and consider them in sequence when classifying subsequent instances.

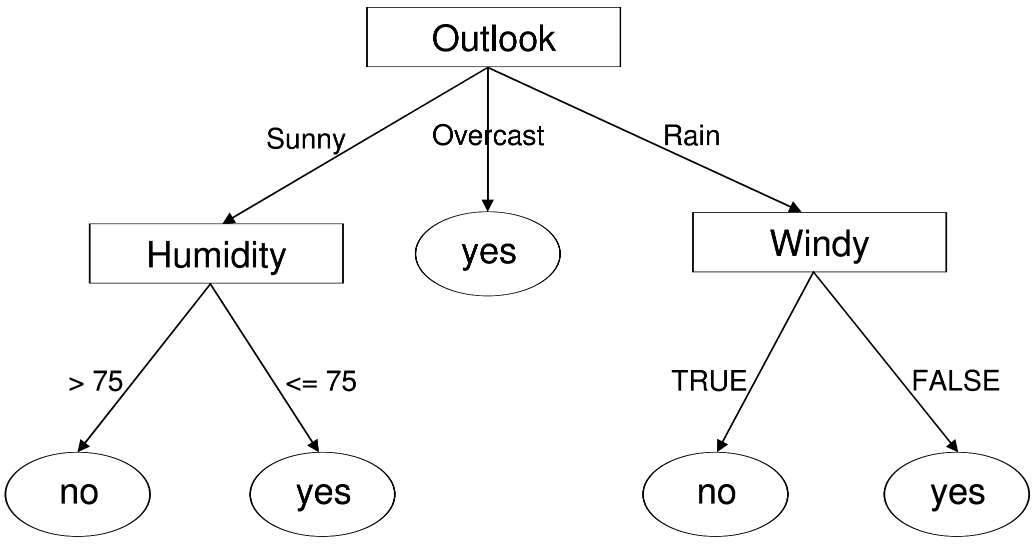


Figure 5.2: The Decision Tree of the Weather Problem

Applying C4.5 on the Weather Dataset

By applying C4.5 on the Weather dataset in Table 5.1, we can get a decision tree as shown in Figure 5.2. The size of the tree is 8 and the number of leaves is 5. It can be represented as a disjunction of conjunctions as follows:

$$\begin{gathered}(Outlook = Sunny \bigwedge Humidity \leq 75)\\ \bigvee(Outlook = Overcast)\\ \bigvee(Outlook = Rain \bigwedge Windy = FALSE)\end{gathered}$$

Using the learned decision tree to classify the example $< Sunny, 85, 85, FALSE >$ initially involves examining the feature at the root of the tree: Outlook. Its value is *Sunny*, so the left branch is followed. Next we examine the value of Humidity and again the left branch is followed because its value $85 \geq 75$. This brings us to a leaf node and the instance is classified as "no".

5.2.2 Naive Bayes Classifier

Naive Bayes classifier is one simple yet highly practical Bayesian learning method [Mit97, RN95]. It has been shown many times that the naive Bayes classifier rivals or even outperforms more sophisticated classification algorithms on many datasets. It is based on the following Bayes theorem, which provides a way to calculate the posterior probability $P(h|D)$ from the prior probability $P(h)$ together with $P(D)$ and $P(D|h)$.

Theorem 6 (Bayes Theorem)

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)} \tag{5.3}$$

Suppose we have a set of training examples S. Each instance x in S is described by a conjunction of attribute values $< A_1, A_2, \ldots, A_n >$. The target function $f(x)$ can take on any value from some finite set V. The learning task is to determine the target value of a new instance $< a_1, a_2, \ldots, a_n >$.

Usually a Bayes method classifies a new instance by assigning the most probable target values, v_{MAP}, given the attribute values $< a_1, a_2, \ldots, a_n >$.

$$v_{MAP} = argmax\ P(v_j \mid a_1, a_2, \ldots, a_n), \qquad v_j \in V \tag{5.4}$$

Applying Bayes theorem, it can be rewritten as

$$\begin{aligned} v_{MAP} &= argmax\ \frac{P(a_1, a_2, \ldots, a_n\ |v_j)P(v_j)}{P(a_1, a_2, \ldots, a_n)} \\ &= argmax\ \ P(a_1, a_2, \ldots, a_n\ |v_j)P(v_j), \quad v_j \in V \end{aligned} \tag{5.5}$$

Now we want to estimate the two terms in the above equation based on the training data S. It is easy to calculate $P(v_j)$ simply by counting the frequency with which each target value $v_(j)$ occurs in S. But estimating $P(a_1, a_2, \ldots, a_n\ |v_j)$ is infeasible unless we have a *huge* set of training examples. This is because its probability is so small that we need a large training dataset in order to obtain reliable estimates.

The naive classifier computes this term based on a simplifying assumption that *the attribute values a_i are independent of each other given the target value v_j*. Therefore the probability of observing the conjunction $< a_1, a_2, \ldots, a_n >$ is just the product of the probabilities for the individual attributes:

$$P(a_1, a_2, \ldots, a_n \mid v_j) = \prod_i P(a_i \mid v_j) \tag{5.6}$$

Substituting this into Equation (2.5) we have the equation used by the naive Bayes classifier as follows:

$$v_{NB} = argmax\ P(v_j) \prod_i P(a_i \mid v_j), \qquad v_j \in V \tag{5.7}$$

where v_{NB} denotes the target value output by the naive Bayes classifier.

We can see that the naive Bayes classifier greatly simplifies the learning by assuming that features are independent given the class. It also assumes that no hidden attributes influence the learning. Some of the advantages of the naive Bayes classifier include its simplicity, clear semantics, and impressive performance in practice. The main weakness comes from its strong assumption of feature independence. Another point that needs to mentioned is that the learning algorithm does not explicitly search through the space of all hypotheses. The hypothesis is computed simply by counting the frequency of various data combinations in the training examples.

Missing Values and Numeric Attributes

In the naive Bayes classifier learning, missing values cause no problem at all. If a value is missing, it is simply not included into the frequency counting. Numeric values are often handled [JL95, WF99] by assuming that they are generated from a Gaussian distribution $P(a_i|v_j) = N(\mu_i, \delta_i)$. For each numeric attribute a_i and each class v_j, the mean μ and standard deviation δ are calculated as follows:

$$\mu = \frac{1}{N} \sum_{i=1}^{N} a_i \tag{5.8}$$

$$\delta = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (a_i - \mu)^2} \tag{5.9}$$

where N is the number of examples in class v_j.

Then $P(a_i|v_j)$ is simply estimated by the Gaussian distribution:

$$P(a_i|v_j) = \frac{1}{\sqrt{2\pi}\delta} e^{-\frac{(a_i-\mu)^2}{2\delta^2}} \tag{5.10}$$

$P(v_j)$ can be calculated in the same way as in nominal attributes by frequency counting.

The normal distribution assumption we just described is another restriction on Naive Bayes method. However, in practice if we know what the distribution for the numeric value is, we can use the standard estimation procedures for that distribution rather than a Gaussian distribution. Another strategy to deal with numeric values is simply to discretize the data before learning.

Applying Naive Bayes on the Weather Dataset

By applying the naive Bayes learning on the Weather data, we can learn a simple Bayes classifier as shown in Figure 5.3. The root node 'Play' has a prior distribution of $P(yes, no) = [0.62, 0.38]$.

For $P = yes$, Outlook has a conditional distribution of $P(O|P = yes)$ = [3/12, 5/12, 4/12]; Temperature has a Gaussian distribution of $P(T|P = yes)$ = N(72.9697, 5.2304); Humidity has a Gaussian distribution of $P(H|P = yes)$ = N(78.8395, 9.8023); and the distribution of Windy is $P(W|P = yes)$ = [4/11, 7/11].

For $P = no$, Outlook has a conditional distribution of $P(O|P = no)$ = [4/8, 1/8, 3/8]; Temperature has a Gaussian distribution of $P(T|P = no)$ = N(74.8364, 7.384); Humidity has a Gaussian distribution of $P(H|P = no)$ = N(86.1111, 9.2424); and the distribution of Windy is $P(W|P = no)$ = [4/11, 7/11].

When using the naive Bayes classifier to classify a new instance < Sunny, 85, 85, FALSE >, the Bayes rule is applied to compute the MAP on Play: first computing $P(Play| < Sunny, 85, 85, FALSE >)$ and then pick up the value with larger probability as the class of the instance. For this example, the returned posterior marginal probabilities of Play is [0.1566, 0.8434], so we label the instance as "no".

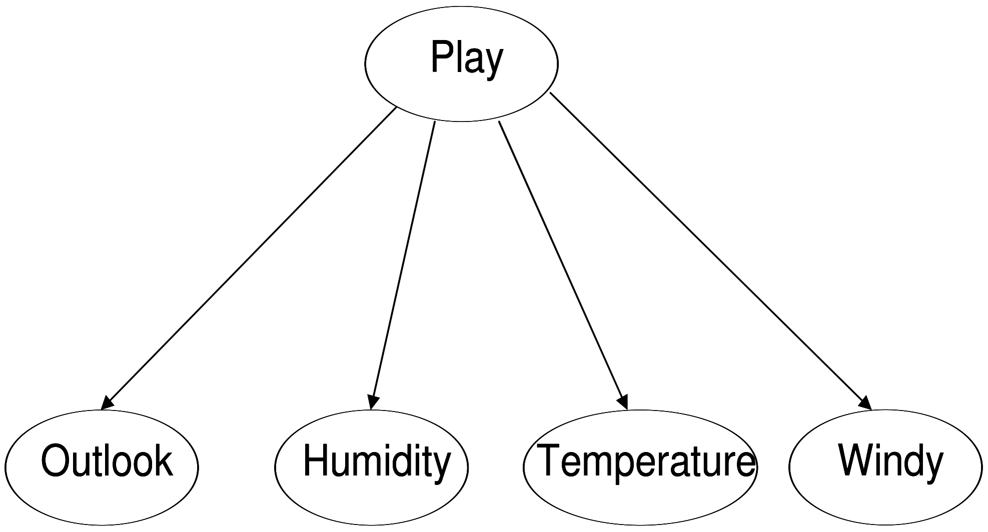


Figure 5.3: The Naive Bayes Classifier of the Weather Problem

5.2.3 Bayesian Network Learning

Learning a Bayesian network from data [CH92, Hec96, RN95, Mit97] is to search for a network that has a high posterior probability given the training data. The learning algorithm takes as input the training data and some domain knowledge and then outputs the network structure and the CPTs. In practice, the Bayesian network learning problem has several varieties. The structure of the network can be *known* or *unknown*, and the variables in the network can be *observed* or *hidden* [Hec96, RN95].

Our research aims to learn a network that contains dependency relationships between instance features and algorithm performance. We have partial knowledge about the network structure, i.e., the nodes representing instances features should be the parents of nodes that represent algorithms. We will also assume that there are no hidden variables. This assumption is reasonable in practice because we have used domain knowledge to select the algorithms and the corresponding problem features. Another difficulty with learning with hidden variables and unknown structure is that, at present, no good general algorithms are known for this kind of problem.

The Bayesian network learning algorithm we will use is a *search-and-scoring based algorithm*, namely $K2$, using a *Bayesian score* developed by [CH92]. The algorithm searches through the spaces of all possible structures looking for a structure that best fits the data according to some scoring criteria. Since the search space is usually huge, the application of heuristic search is justifiable.

The Bayesian score used in $K2$ is stated in the following theorem.

Theorem 7 (The Bayesian Score) *Suppose Z is a set of n random variables. Each variable $x_i \in Z$ has r_i possible value assignments: $(v_{i1}, v_{i2}, \ldots, v_{ir_i})$. D is a training dataset of m cases (examples). Each case contains a value assignment of each variable in Z. Let B_S denote a Bayesian network structure containing just all variables in Z. Each variable x_i in B_S has a set of parents π_i. Let w_{ij} denotes the jth unique instantiation of π_i relative to D and there are q_i such unique instantiation of π_i. N_{ijk} is the number of cases in D in which variable x_i has the value of v_{ik} and π_i is instantiated as w_{ij}. Let $N_{ij} = \sum_{k=1}^{r_i} N_{ijk}$. Then,*

$$P(B_S, D) = P(B_S) \prod_{i=1}^{n} g(i, \pi_i) \tag{5.11}$$

where

$$g(i, \pi_i) = \prod_{j=1}^{q_i} \frac{(r_i - 1)!}{(N_{ij} + r_i - 1)!} \prod_{k=1}^{r_i} N_{ijk}! \tag{5.12}$$

This result can be used to find the most probable network structure given a training dataset. However, the space of all possible structures is exponentially huge, so some heuristics are often used to make the search more efficient. $K2$ uses a greedy heuristic to search the best parents set for each variable. $K2$ assumes a uniform prior distribution of all structures. It also assumes that there is an ordering α available on the variables such that x_i can be the parent of x_j if and only if x_i precedes x_j in α. The $K2$ algorithm is described in Figure 5.4.

The parameter values (CPT probabilities) are calculated by the following formula.

$$E(\theta_{ijk} | D, B_S, \xi) = \frac{N_{ijk} + 1}{N_{ij} + r_i} \tag{5.13}$$

```
Procedure K2
For i:=1 to n do
   $\pi_i = \phi$;
   $P_{old} = g(i, \pi_i)$;
   OKToProceed := true
   while OKToProceed and $|\pi_i| < u$ do
      let $z$ be the node in $Pred(x_i) - \pi_i$ that maximizes $g(i, \pi_i)$;
      $P_{new} = g(i, \pi_i \bigcup z)$;
      if $P_{new} > P_{old}$ then
         $P_{old} := P_{new}$ ;
         $\pi_i := \pi_i \bigcup z$ ;
      else OKToProceed := false;
   end while
   write("Node:", "parents of this nodes :", $\pi_i$ );
end for
end K2
```

Figure 5.4: The K2 Algorithm

The K2 algorithm takes as input a set of n nodes, an ordering α on the nodes, an upper bound u on the number of parents a node may have, and a database D of m training examples. For each variable, it first assumes the node has no parents, and then adds to its parent set new node incrementally from among the predecessors in the ordering such that the added parent node increases the probability of the resulting structure by the largest amount (in this sense it uses a greedy search strategy). It stops adding parents when no improvement can be made or when it has got u parents. It then outputs the parents of the node.

We first discretize the weather data and then run K2 on it. The learned Bayesian network is shown in Figure 5.5. The order used is <Outlook, Temperature, Humidity, Windy, Play>. The classification process of Bayesian networks is the same as the naive Bayes classifier's, i.e., computing the posterior marginal probabilities of the class node and then taking the value with maximum probability.

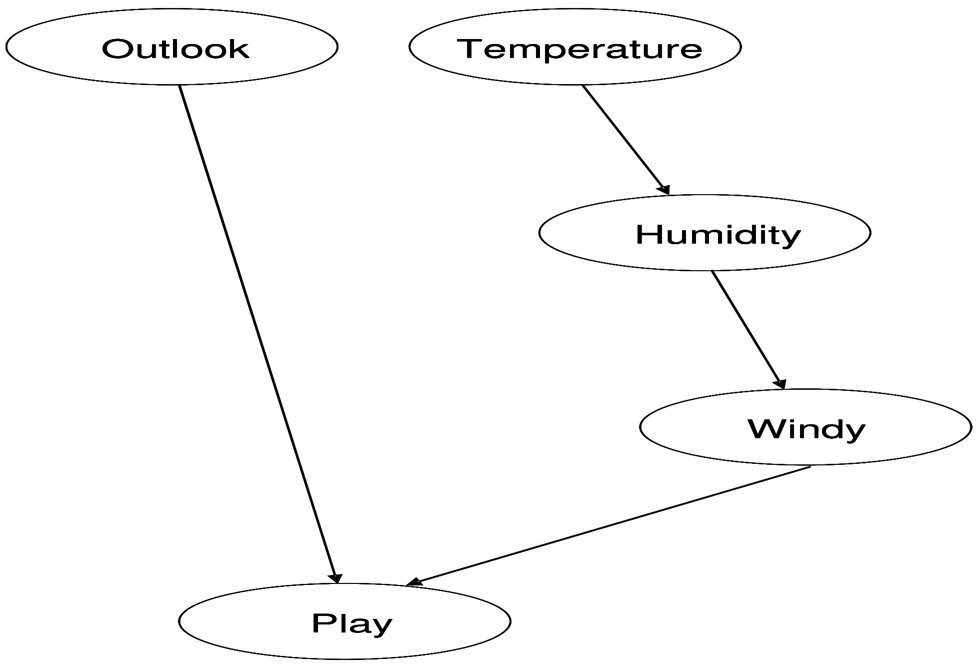


Figure 5.5: The Bayesian Network of the Weather Problem Learned by K2

Bayesian Networks and the Naive Bayes Classifier

Recall that the naive Bayesian classifier makes use of the independency assumption that *all the attributes* $a_1, a_2, \ldots, a_n$ *are conditionally independent given the target value* v. This assumption dramatically reduces the complexity of learning the target function. However, in many cases the assumption is overly restrictive. In contrast to the naive Bayesian classifier, Bayesian networks is less constraining by allowing conditional independence assumptions that apply to subsets of the variables rather than all variables. Thus it provides an intermediate approach that is less restrictive comparing to the naive Bayesian classifier, yet more tractable than dealing with the exponentially-sized joint probability space directly.

5.3 Evaluation of the Learning Algorithms

Evaluating the learning algorithm is an important part of machine learning [WF99]. For the evaluation of a single classifier, the most common criterion is *classification accuracy*. Stratified k-fold cross-validation is the standard method in practice to compute the classification accuracy. Another common way is using a *confusion matrix* and the *Kappa statistic* to describe the disagreement between the actual classes and the predicted classes by the learned classifier. If the classifier can predict classifications with probability weights, the *Root Mean Square Error* (RMSE) can be applied. For comparing two classifiers on the same test data, a *paired t-test* is often used to compare the average error rate over several cross-validations.

5.3.1 Classification Accuracy and Error Rate

Classification accuracy is simply defined as the percentage of test examples correctly classified by the learning algorithm. *Error rate* is one minus the classification accuracy.

Strictly speaking, all we can measure is just the *sample error*; it is impossible to get the exact *true error*. The true error is the error rate over the entire unknown distribution D of examples. The sample error is the error rate over the sample of data that is available. Fortunately, the sample error has been shown to be a good estimate to the true error [Mit97]. More specifically, suppose we want to estimate the true error of a hypothesis h based on the sample error, $error_s(h)$, measured over a sample S of n examples. Suppose again we are given the following conditions:

- the n samples drawn independently from each other, and independent of h, according to the probability distribution D
- $n \geq 30$
- h commits r errors over these n examples, i.e., $error_s(h) = r/n$.

Under these conditions, statistic theory tells us that the following assertions are true:

- Given no other information, the most probable value of the true error, $error_D(h)$, is the sample error $error_s(h)$.
- With approximately 95% probability, the true error lies in the interval

$$error_s(h) \pm 1.96\sqrt{\frac{error_s(h)(1-error_s(h))}{n}}$$

When enough data is available, we can just use the training set to learn the model and use a different test set to estimate the classification accuracy, supposing they both are representative and are drawn from the same distribution. But, data is often scarce and expensive in practice. When the amount of data is limited, the most common performance evaluation method is *repeated cross-validation*. The idea is to partition the data into training and test sets in different ways. The learning algorithm is trained and tested for each partition and the classification accuracy averaged. This often provides a more reliable estimate of the true classification accuracy.

5.3.2 Stratified k-fold Cross-validation

In practice, a standard way of evaluating the classification accuracy is called *stratified k-fold cross-validation* [WF99] in which the training data is randomly divided into k mutually exclusive subsets of approximately equal size. In each subset, the class is represented in approximately the same proportions as in the whole data set. The learning algorithm is executed and the learned model tested k times. For each iteration, one subset is held out as test set and the remaining $k-1$ subsets are used for training. Finally, the k estimates are averaged to yield the overall classification accuracy. Often times, a single stratified k-fold cross-validation might not produce a reliable estimate, so we typically run cross-validation many times and average the results. In this research, we will use ten ten-fold cross-validations, i.e., executing the learning algorithm one hundred times on data sets that are all nine-tenth the size of the original data.

5.3.3 Confusion Matrix and Kappa Statistic

One possible method of evaluating classification experiments is to count the number of correctly and wrongly classified data. This gives a rough impression of how good the classification is. In order to get a better interpretation of the result, it is also useful to know which classes of data were most often misplaced.

The *confusion matrix*, a matrix containing information about the actual and predicted classes [KP98], is a convenient tool for counting. In the confusion matrix, all of the columns represent the predicted classes, and thus a piece of data belongs to the column if it is classified as belonging to this class. The rows represent the actual classes, and a piece of data is thus represented in a particular row if it belongs to the corresponding class. A perfect classification results in a matrix with 0's everywhere but on the diagonal. A cell which is not on the diagonal but has a high count signifies that the class of the row is somewhat confused with the class of the column by the classification system.

After the confusion matrix is fixed, we can calculate the *Kappa statistic* (K), or Kappa coefficient, to quantify the agreement between the actual and the predicted classifications [Coh60, Kra82]. The Kappa statistic measures the proportion of agreement after chance agreements (samples that are accidentally classified correctly) have been removed from considerations. The Kappa statistic is defined as follows:

$$K = \frac{P(A) - P(E)}{1 - P(E)} \tag{5.14}$$

where $P(A)$ is the accuracy of observed agreement, $P(E)$ is the estimate of chance agreement. They are computed by the following equations:

$$P(A) = \frac{\sum_{i=1}^{n} m_{ii}}{T} \tag{5.15}$$

and

$$P(E) = \frac{\sum_{i=1}^{n} m_{i+} m_{+i}}{T^2} \tag{5.16}$$

where T is the total sum of all numbers in the matrix, m_{ii} is the numbers on the diagonal, m_{i+} is the marginal total of row i, and m_{+i} is the marginal total of column i.

K increases to one as agreement by chance decreases and become negative as less than chance agreement occurs. $k = 0$ when the agreement equals chance agreement.

5.3.4 Evaluating Predicted Probabilities: RMSE

Some classifiers not only predict the 0 or 1 classification, but also provide a probability weight to each class label. Therefore, this type of classifier calculates a probability vector $p = [p_1, \ldots, p_k]$ for the classes, $\sum_{i=1}^{k} p_i = 1$. The true classification can be represented as a vector of 0 and 1: $a = [a_1, \ldots, a_k]$, $a_k = 0$ or 1. In this case the $RMSE$ is frequently used to measure the difference between the true classification and the estimated class probabilities over several cross-validations.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{k}(p_i - a_i)^2}{k}} \tag{5.17}$$

5.3.5 Comparing Two Classifiers: Paired T-test

Often times, we want to compare two learning algorithms, say C4.5 and naive Bayes, on the same problem to see which is better. We can apply both algorithm on the same training data, then use the same cross-validation splits for each algorithm. Finally, we can get a pair of results on the same test data. A statistic method called the *paired t-test* [WF99] can be used to compare the average error rate of these two results.

Given two paired sets X_i and Y_i of n result values, the paired t-test determines if they differ from each other in a significant way. Let $\bar{X} = \sum_{i=1}^{n} X_i/n$ and $\bar{Y} = \sum_{i=1}^{n} Y_i/n$, let $\hat{X}_i = (X_i - \bar{X})$ and $\hat{Y}_i = (Y_i - \bar{Y})$. t is defined by:

$$t = (\bar{X} - \bar{Y})\sqrt{\frac{n(n-1)}{\sum_{i=1}^{n}(\hat{X}_i - \hat{Y}_i)^2}} \tag{5.18}$$

This statistic has $n - 1$ degrees of freedom. A table of Student's t-distribution confidence intervals can be used to determine the significance level at which two distributions differ.

Table 5.2 is the confidence limits table for Student's distribution with 9 degrees of freedom [1]. In practice, 5% or 1% is often used as the confidence level. If the value

[1] In our evaluation scheme the number of experiments is 10.

Table 5.2: Confidence Limits for the Student's Distribution with 9 Degrees of Freedom

$Pr(X \geq z)$	z
0.1%	4.30
0.5%	3.25
1%	2.82
5%	1.83
10%	1.38
20%	0.88

of t at the confidence level is greater than z or less than $-z$ in Table 5.2, then we conclude that there is a significant difference between the two learning algorithms on the dataset. It is easy to see which is better by comparing their error rates.

5.3.6 Evaluation of Learning on the Weather Problem

We have learned a decision tree from the Weather data using C4.5 as shown in Figure 5.2. Now we list the evaluation result in Table 5.3.

Table 5.3: Evaluation Result of C4.5 on the Weather Data

10 stratified 10-fold cross-validation evaluation summary	
Total #Instances	14
Correctly Classified Instances	9
Classification Accuracy	64.2857
Error Rate	35.7143
Kappa statistic	0.186
Root mean squared error	0.4818
Confusion Matrix	$\begin{array}{cc c} a & b & \leftarrow classified\ as \\ \left(\begin{array}{cc} 7 & 2 \\ 3 & 2 \end{array}\right) & & \begin{array}{c} a \\ b \end{array} \end{array}$

5.4 Data Preprocessing

We have introduced three models and their learning algorithms: decision tree learning, the naive Bayes classifier, and Bayesian network learning. When applying machine learning algorithms to practical data mining problems, there are some other important processes that can improve the performance of learning. In the following, we introduce two important data preprocessing methods: *feature (attribute) selection* and *discretization*.

5.4.1 Feature Selection

In many practical situations, there are often too many features or attributes (infinitely many in theory) for learning algorithms to handle. Most of them are irrelevant or redundant. In practice, the data must be preprocessed to select the most relevant attributes to use in learning. This is called feature selection [Hal99]. The best way to select relevant attributes is manually, based on a good understanding of the problem and the meaning of the attributes. This is a way to combine a domain expert's knowledge into learning. However, automatic methods can also be very useful sometimes, especially when human expert fails to make the precise choice between two closely related attributes. Feature selection also makes the model more compact and speeds up the learning process, although this maybe outweighed by the computation used for feature selection.

There are two important feature selection approaches: the *filter* method and the *wrapper* method [KJ97]. The former evaluates the worth of feature subsets based on general characteristics of the data. The latter wraps the machine learning algorithm that will ultimately be used into the feature selection process and uses it to evaluate the feature subsets.

Both methods involve searching the space of all combinations of features for the subset that is most likely to predict the class best. Within each categories, algorithms can be further differentiated by what evaluation functions are used and how the space of feature subsets is explored. The search space is exponential in the number of features. Therefore, heuristic searches are often used. For example, greedy search and genetic algorithm are two common search strategies for feature selection.

Both filters and wrappers have their advantages and disadvantages. Wrappers often give better results in terms of the final predictive accuracy of the learning algorithm because the feature selection is optimized for the particular learning algorithm. But wrappers are very expensive to run and can be intractable for datasets with many features. Also, wrappers are less general because they are tightly related with a learning algorithm. Wrappers must be rerun when switching to a new learning algorithm. Filters are much faster than wrappers and are independent of the learning algorithms.

In this thesis, we will use a GA wrapper [WF99, KJ97, Hsu02] to perform feature selection before learning. We will also compare the model learned with and without feature selection to see how feature selection might improve the overall learning. Again, we have tested it on the Weather dataset using a GA wrapper with C4.5 evaluator. The result shows that feature selection is able to identify Temperature as an irrelevant attribute and reduces the number of features by one.

5.4.2 Discretization of Numeric Attributes

Discretization is a procedure that takes a data set and converts all continuous attributes to nominal. It is a necessary preprocessing step for learning algorithms that require nominal attributes. Recent research also shows that some common machine learning algorithms benefit from treating all features in a uniform fashion. In this case discretization is also useful.

Equal-width and Equal-frequency Intervals

The simplest and most straightforward discretization method is using predetermined *equal-width intervals*. It involves sorting the values of a continuous feature and dividing the range of these values into k equally sized bins. This is often done with the help of domain knowledge at the time when data is collected. But it runs in the risk

Table 5.4: The Discretized Weather Data

Instance#	**Features**				**Class**
	Outlook	Temperature	Humidity	Windy	Play
1	sunny	(82.9-inf)	(83.6-86.7]	FALSE	no
2	sunny	(78.7-80.8]	(89.8-92.9]	TRUE	no
3	overcast	(82.9-inf)	(83.6-86.7]	FALSE	yes
4	rainy	(68.2-70.3]	(92.9-inf)	FALSE	yes
5	rainy	(66.1-68.2]	(77.4-80.5]	FALSE	yes
6	rainy	(-inf-66.1]	(68.1-71.2]	TRUE	no
7	overcast	(-inf-66.1]	(-inf-68.1]	TRUE	yes
8	sunny	(70.3-72.4]	(92.9-inf)	FALSE	no
9	sunny	(68.2-70.3]	(68.1-71.2]	FALSE	yes
10	rainy	(74.5-76.6]	(77.4-80.5]	FALSE	yes
11	sunny	(74.5-76.6]	(68.1-71.2]	TRUE	yes
12	overcast	(70.3-72.4]	(89.8-92.9]	TRUE	yes
13	overcast	(80.8-82.9]	(74.3-77.4]	FALSE	yes
14	rainy	(70.3-72.4]	(89.8-92.9]	TRUE	no

of producing bad choices of boundaries if the values are not very evenly distributed.

An improvement to equal-width is using *equal-frequency intervals*. It sorts the values of a feature, divides the ranges into a predetermined number of k bins, and assigns $\frac{1}{k}$ of the values to each bin.

Both equal-width and equal-frequency binning are *unsupervised* discretization methods because they do not make use of the class in determining interval boundaries. Supervised methods have advantages of taking classes into account during discretization process and thus can produce better intervals.

Entropy-based Discretization Using MDL Stopping Rule

In practice, one of the best general techniques for supervised discretization is *the entropy-based method with the MDL stopping rules* developed by Fayyad and Irani [FI93]. This method uses the class information entropy of candidate partitions to select bin boundaries. The idea is similar to the process of splitting a numeric attribute during the learning of a decision tree.

Suppose we are given a set of instances S, a continuous feature A to be discretized, and a partition boundary T. T partitions S into two subsets S_1 and S_2. Let there be k classes $C_1, \ldots, C_k$ and let $P(C_i, S)$ be the proportion of examples in S that have class C_i. The *class entropy* of S is defined as:

$$E(S) = -\sum_{i=1}^{k} P(C_i, S) \log(P(C_i, S)). \tag{5.19}$$

The class information entropy of the partition introduced by T is given by:

$$E(A, T; S) = \frac{S_1}{S} E(S_1) + \frac{S_2}{S} E(S_2). \tag{5.20}$$

For a given feature A, the boundary T_{min} that minimizes the entropy over all possible partitions is selected as a binary discretization boundary. This procedure is applied recursively to both of the subsets introduced by T_{min} until it is time to stop, thus creating multiple intervals on feature A.

The *Minimal Description Length Principle* is used to determine the stopping criterion. The recursive discretization process stops if and only if the information gain is smaller than a threshold:

$$Gain(A, T; S) < \frac{\log_2(N-1)}{N} + \frac{\log_2(3^k - 2) - kE(S) + k_1 E(S_1 + k_2 E(S_2)}{N} \tag{5.21}$$

where N is the number of examples in S, k_i is the number of classes in S_i, and

$$Gain(A, T; S) = E(S) - E(A, T; S) \tag{5.22}$$

Applying this algorithm to the Weather data, we can get a discretized data set as shown in Table 5.4. Note that both Temperature and Humidity have been discretized to 10 intervals.

5.5 Meta-learning: Combining Multiple Models

Data preprocessing methods engineer the input data to improve the overall performance of learning. In contrast there are methods that engineer the output from machine learning algorithms. In this section we will introduce three such meta-learning schemes – bagging, boosting, and stacking – that combine multiple learned models to form a stronger model.

5.5.1 Bagging

Bagging [Bre96, WF99] stands for "*bootstrap aggregating*". The basic idea of bagging is to generate multiple training datasets from the original dataset using bootstrap sampling and induce a model from each dataset. When classifying a new instance, all learned models vote on the final classification. The class receives the most votes is returned as the predicted class. In bagging, multiple models are learned in a parallel manner. The algorithm is described in Figure 5.6.

```
Bagging($n$ training examples $D$, **int** $t$, a test instance)
    **Learning**
    **for** i=1 to $t$
        Sample $n$ instances $D_i$ with replacement from $D$;
        Apply the learning algorithm to $D_i$;
        Save the learned model $M_i$;

    **Classification**
    **for** i=1 to $t$
        Predict class of instance using $M_i$;
**return** class that has been predicted by most models.
```

Figure 5.6: Algorithm for Bagging

5.5.2 Boosting

Boosting [FS96, WF99] also adopts the idea of voting to combine multiple models, but it learns multiple models in an iteratively sequential manner. The learned models in boosting often complement each other; i.e., every one of them is good at predicting class for a particular subset of examples. Another difference is that boosting uses a weighted vote that assigns more weight to the more successful models. In boosting, each instance also has a weight. Initially, all training instances are assigned equal weights. The learning algorithm is then applied and a classifier is generated. Each instance is reweighed according to the output of the learned classifier. The weight of correctly classified instances is decreased, and that of misclassified ones is increased. In the next iteration, another classifier is built from the reweighed data. Consequently, it focuses more on classifying the misclassified instances correctly. The iteration keeps on going until the stopping criterion is satisfied. Finally, this process will generate a series of classifiers complementing each other. The weights used to update instances after each iteration are determined by the error rate e of the learned classifier:

$$Weight_{new} = \frac{e}{1-e} Weight_{old} \tag{5.23}$$

The iteration stops whenever $e \geq 0.5$ or $e = 0$.

In forming the predictions, outputs from all classifiers are combined using a weighted vote. The weight of a classifier is determined by how well it performs on the training dataset from which it was built. It is calculated by the following formula.

$$Weight_{vote} = -\log \frac{e}{1-e} \tag{5.24}$$

The weights of all classifiers that vote for a particular class are summed and the class with the greatest total weight is returned as the predicted class. The algorithm is described in Figure 5.7.

5.5.3 Stacking

In both bagging and boosting, the multiple models being combined are all the same type. Stacking [Wol92, WF99] is able to combine models built by different learning algorithms. For example, suppose we have learned a decision tree, a naive Bayes classifier, and a Bayesian network from the same data. We can use stacking to combine them all together to form a new classifier. Stacking does so by introducing a *meta-learner* to learn which classifier is the most reliable one and to decide how best to combine the predictions of all classifiers. Therefore there are two levels of

```
Boosting($n$ training examples $D$, a test instance)
    **Learning**
    Assign equal weight to each training example;
    STOP = **false**;
    **while** (!STOP)
        Apply the learning algorithm to $D_i$, save the learned
        model $M_i$, compute and save the error rate $e$;
        **if** $e \geq 0.5$ **or** $e = 0$
            STOP = **true**;
        **for** each training example in $D_i$
            **if** the instance is correctly classified
                Multiply the weight by $\frac{e}{1-e}$;
        Normalize all weights;
    **end while**

    **Classification**
    Assign weight of zero to all classes;
    **for** each learned model $M_i$
        add $-\log \frac{e}{1-e}$ to the predicted class;
**return** class with the greatest weight.
```

Figure 5.7: Algorithm for Boosting

learning in stacking. The level-0 learners, or base learners, are trained as they would be if they were stand-alone using the original training data. The level-1 learner, or the meta-learner, needs a different level-1 training data that reflects the performance of level-0 learners. This is done by letting each level-0 learned model classify an instance, and attaching to all predictions the actual class value. Usually, a portion of the original training data is reserved for generating level-1 training data in order to reduce the biases. Since most of the work is already done at level-0, the level-1 learner is usually a simple classifier, such as a linear model. The algorithm outline for stacking is described in Figure 5.8.

```
Stacking($n$ training examples $D$, a test instance)
    **Learning**
    Hold out a subset of training example $D_1$ for level-1 learning;
    Apply $k$ learning algorithms to the remaining data;
    Save the learned $k$ models $M_i$;
    Use these $k$ models to classify $D_1$ and generate level-1 training data;
    Apply a simple linear regression to produce the level-1 classifier

    **Classification**
    Apply $M_i$ to classify the test instance;
    Use the result to form a level-1 instance;
    Apply the level-1 model to decide the best class;
**return** the best class.
```

Figure 5.8: Algorithm for Stacking

5.6 Overview of the Learning Process for Algorithm Selection

Working flow of our machine learning-based approach for algorithm selection consists of all procedures we have discussed so far. More specifically, we have instance generation, data collection, data preprocessing (discretization and feature selection), learning, multiple models combination (bagging, boosting and stacking), and model evaluations. Figure 5.9 gives an overview of the whole process. In the following two chapters, we will apply this process to algorithm selection for sorting and the MPE problem.

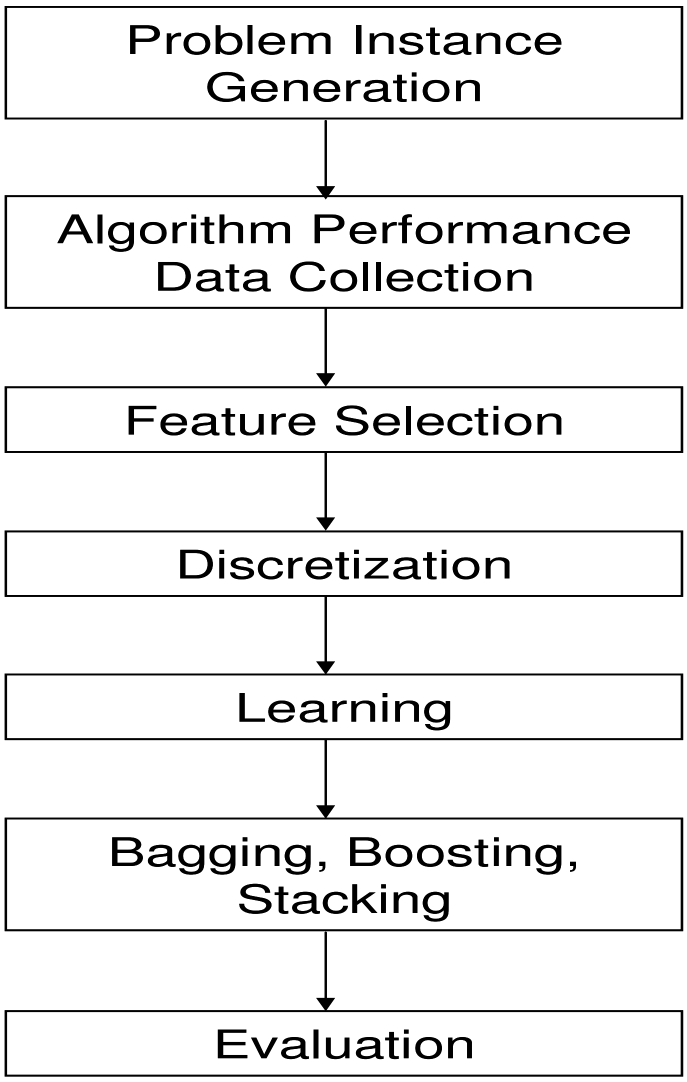

Problem Instance Generation
Algorithm Performance Data Collection
Feature Selection
Discretization
Learning
Bagging, Boosting, Stacking
Evaluation

Figure 5.9: Overview of the Machine Learning-based Approach for Algorithm Selection

Chapter 6

Algorithm Selection for Sorting

In this chapter we apply the machine learning-based approach to the selection of a sorting algorithm. More specifically, we examine the relationships between three important presortedness measures − INV, RUN, REM − and the performance of five well-known sorting algorithms: Insertion Sort, Shellsort, Heapsort, Mergesort, and Quicksort.

6.1 Sorting

Sorting is the process of rearranging a given sequence of comparable items into ascending or descending order [Knu81]. It is one of the most fundamental problems in computing and has been intensively studied for many years. Moreover, it is also a very practical problem. It is estimated that over 25 percent of the world's CPU running time is being spent on sorting [Knu81].

To simplify matters, we will assume without loss of generality that the orderable items are unique integers and the entire sort can be done in the main memory. This type of sorting is called *internal sorting.* We will not consider *external sorting* in which sorts can not be performed in main memory and must be done on disk or tape [Knu81, Wei99]. We will also assume that the input is a permutation of numbers from 1 to n. We assume this abstraction to simplify analysis and implementations, but the algorithms cannot assume this and exploit that information to speed up the sort. We will use the ascending order as the correct one.

We will use the *comparison-swap model* [Raw92] described as follows. The problem is that

- we have an array of integers in arbitrary order.
- we want an ordered array of the integers.

Our environment is that

- the integers are all different (for simplification).
- we can derive the order information only by comparing integers.
- we can preserve the order information only by swapping integers.

It is well-known that sorting in this comparison-swap model has a lower bound complexity of $O(n \log n)$ [Raw92], where n is the number of items to be ordered.

6.2 Sorting Algorithms

It would be impossible to investigate and compare all of the sorting algorithms and their variations. In our experiments we have chosen the following 5 sorting algorithms;

- Insertion Sort
- Shellsort
- Heapsort
- Mergesort
- Quicksort

These have been chosen because they are representatives of the various categories of sorting algorithms. Next we briefly introduce each algorithm. Interested readers should consult [Knu81, Raw92] for details.

6.2.1 Insertion Sort

Insertion sort is one of the simplest sorting algorithms. It sorts the list by incrementally inserting unsorted items into a sorted sublist. Insertion sort consists of $n-1$ passes. For pass $p = 1$ through $n-1$, it ensures that the elements in positions 0 through $p-1$ are already in sorted order. Insertion sort makes use of this fact to insert the unsorted element into its correct place relative to the sorted sublist.

Insertion sort is $O(n^2)$. The worst case occurs when the input is in reverse order. The average-case complexity is also $O(n^2)$ for insertion sort. However, one important fact about insertion sort is that it is linear on nearly-sorted input. The presortedness of an input list can be quantified by various measures. We will examine three of them in later sections.

6.2.2 Shellsort

Insertion sort is slow because it is inefficient at moving data. To move an element takes time proportional to that distance because insertion sort exchanges only adjacent elements. *Shellsort* [Boo63], named after its inventor, D. L. Shell, is a simple extension of insertion sort. It gains speed by comparing and exchanging elements that are distant. Shellsort uses an *increment sequence* $h_1, h_2, \ldots, h_t$. After a phase, using some increment h_k, for every i, we have $a[i] \leq a[i + h_k]$ and all elements spaced h_k apart are sorted. When the subsequences are all sorted, the resulting array is nearly sorted and a final pass of insertion sort is called to completely sort it. Because h_k decreases as the algorithm runs, Shellsort is sometimes referred to as *diminishing increment sort.*

Any increment sequences will work as long as $h_1 = 1$, but some are better than others. Shell suggested a natural choice for increment sequences: $h_t = \lfloor n/2 \rfloor$ and $h_k = \lfloor h_{k+1} \rfloor$. Although the worst-case running time of Shellsort using this increment sequences is $O(n^2)$, its performance is quite acceptable in practice even for large permutations. Another advantage of Shellsort is that it requires only a small amount of coding to get it working, while other sorting algorithms are significantly more complicated, if a little more efficient.

6.2.3 Heapsort

Heapsort [Wil64] first uses $O(n)$ time to build a binary heap of n elements. It then performs n **deleteMin** operations so that the elements leave the heap in sorted order. By recording and copying these elements back to the array, we finally have a sorted array. Since each **deleteMin** operation takes $O(\log n)$ time, the total running time is $O(n \log n)$. Because Heapsort finds the m smallest elements in $O(n + m \log n)$ time, it is preferred in cases where incremental sorting is needed.

6.2.4 Mergesort

Mergesort recursively sorts the array using a divide-and-conquer strategy. Mergesort recursively breaks large sequences into smaller subsequences and then sorts and merges these subsequences into one sorted list. The time to merge two sorted subsequences is linear. The worst-case running time of mergesort is $O(n \log n)$, and the number of comparisons used is nearly optimal. The drawback is that it uses extra memory.

6.2.5 Quicksort

Quicksort [Hoa61] is the fastest known sorting algorithm in practice. Like mergesort, quicksort is also a divide-and-conquer recursive algorithm. It uses the idea of "partitioning" in its recursive steps. The divide phase partitions the array into two disjoint parts: the "small" elements on the left and the "large" ones on the right. The conquer phase sorts each part separately. Because of the work of the divide phase, it does not need a merge phase to combine partial solutions. Quicksort has an average-case running time of $O(n \log n)$, although its worst-case is $O(n^2)$. Moreover, the worst-case behavior is never observed in practice and can be made exponentially unlikely with a little effort [Wei99]. In quicksort the way of picking the "pivot" to partition the list is crucial. We use *Median-of-Three* partitioning. It uses as its pivot the median of the left, right, and center elements. It has been proven efficient in practice and actually reduces the running time by 5 percent [Wei99].

6.3 Instance Characteristics: Measures of Presortedness

Nearly sorted sequences are common in practice [Knu81]. Such instances are easy in the sense that a small amount of work is needed to sort them. When the sorting algorithms take advantage of the existing order in the input, the time taken to sort is a function of the size of the input sequence and the disorder in the sequence. These types of algorithms are called *adaptive* [Man85]. If the disorder of the sequence is small, the algorithm can do better than $O(n \log n)$. For example, insertion sort is linear for nearly sorted sequences. The existing order in a sequence, or *presortedness*, can be quantified by more than one measure. [ECW92] lists 11 different measures of presortedness. We will study 3 of them: *the number of inversions*, *the number of runs*, and *the longest ascending subsequence*. These were chosen because they are the most natural and representative measures of presortedness.

6.3.1 Inversions (INV)

Let $A = < a_1, a_2, \ldots, a_n >$ be a permutation of set $\{1, 2, \ldots, n\}$. An inversion is a pair of elements in the wrong order.

Definition 23 (Inversions)

$$INV(A) = |\ \{\ (i,j)\ |\ 1 \leq i < j \leq n \ and \ a_i > a_j\ \}\ | \tag{6.1}$$

INV(A) is an important presortedness measure and has been intensively studied in [Knu81]. For an already sorted sequence, we have $INV(A) = 0$, and for a sequence in reverse order, $INV(A) = \frac{n(n-1)}{2}$. INV(A) indicates how many exchanges of adjacent elements are needed to sort A. In this sense it is an accurate performance indicator for algorithms using only adjacent element exchanges such as *bubble sort*. However, it generally has two drawbacks. First, it takes $O(n^2)$ time to compute exact INV. Second, inputs of the following type

$$< k+1, k+2, k+3, \ldots, 2k, 1, 2, 3, \ldots, k >$$

have a quadratic number of inversions, but intuitively they are almost in order. They can be sorted fast using merging.

Inversion Table

The inversion table $< b_1, b_2, \ldots, b_n >$ of permutation $< a_1, a_2, \ldots, a_n >$ is obtained by letting b_j be the number of elements to the left of j that are greater than j. For example, the permutation

$$A =< 4\ 3\ 6\ 2\ 1\ 5 > \tag{6.2}$$

has the inversion table

$$B(A) =< 4\ 3\ 1\ 0\ 1\ 0 >, \tag{6.3}$$

because 4, 3, 6, 2 are to the left of 1; 4, 3, 6 are to the left of 2; and so on. There are a total of 9 inversions in A. By definition, we always have the following relations

$$0 \leq b_1 \leq n-1,\ 0 \leq b_2 \leq n-2,\ \ldots,\ 0 \leq b_{n-1} \leq 1,\ b_n = 0. \tag{6.4}$$

Bijection about INV

One important fact about inversions is that *the inversion table uniquely determines the corresponding permutation.* It is easy to compute A given $B(A)$ [Knu81]. This correspondence is important because we can translate a problem stated in terms of permutations into an equivalent problem stated in terms of inversion tables, and the latter may be easier to solve. We will use this bijection to design a random generation algorithm that can generate permutations with given size and a specified INV uniformly at random.

6.3.2 Runs (RUN)

RUN is the number of ascending substrings, or the "runs up", of a permutation.

Definition 24 (Runs)

$$RUN(A) = |\ \{\ i \mid 1 \leq i < n \ and\ a_i > a_{i+1}\ \}\ | \tag{6.5}$$

For example, the permutation

$$A = <|4|\ 3\ 6|\ 2|\ 1\ 5|> \tag{6.6}$$

has 4 runs. RUN is an important measure because it represents the number of sorted subsequences of the input. For an already sorted sequence, we have $RUN(A) = 1$, and for a sequence in reverse order, $RUN(A) = n$. RUN reflects the intuition of presortedness that *a small number of ascending runs indicates a high degree of presortedness.* It is also easy to compute, i.e., in $O(n)$ time. The drawback with this measure is that it does not capture local disorders well. For example,

$$< 2, 1, 4, 3, \ldots, n, n-1 >$$

produces a lot of runs (n/2), even though it is almost sorted and can be quickly sorted using exchanges of adjacent elements.

Again we want to randomly generate permutations with a specified number of runs for our algorithmic experiments. We need to translate the random generation problem into an simpler, equivalent problem, as we did for INV. In the following we introduce some new concepts about permutations that are useful to design the random generation algorithm.

Representations of Permutations

Any permutation can be represented in at least three notations: *one-line notation*, *cycle-notation*, and *two-line notation* [SW86]. One-line notation is the most common representation. For example, $A = < 3, 5, 1, 4, 2 >$. Cycle-notation is based on the fact that *any permutation can be written as a product of disjoint cycles.* For $A = < 3, 5, 1, 4, 2 >$, we have $A = (13)(25)(4)$. Any cycle of length one in A corresponds to a fixed point of A. In this example $A(4) = 4$ is a fixed point. Two-line notation lists $A(i)$ under i like this:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 5 & 1 & 4 & 2 \end{pmatrix}$$

Bijection about RUN

The cycle-notation, i.e., the decomposition of A into disjoint cycles, gives us another bijection that is useful for help in designing random generation algorithms that can generate permutations with given size and RUN uniformly at random. We put the smallest number of each cycle at the end of that cycle, and put the cycle in order of the last entries. This defines the *canonical cycle decomposition of* A. For example, the canonical cycle decomposition of $A = < 463281795 >$ is (4261)(3)(895)(7). We have another permutation by removing the parentheses: $\phi(A) =< 426138957 >$, in on-line notation. The mapping $\phi(A)$ is a bijection because we can recover A from $\phi(A)$ as follows: the first cycle of A is the initial segment of $\phi(A)$ ending at 1. The next cycle of A ends in the smallest number not appearing in the first. All remaining cycles are calculated in the same manner. For $n = 4$, the bijection is given in Table 6.1.

The first entries of A and $\phi(A)$ are always the same. This is because the last entry of the first cycle of A is 1, so ϕ maps 1 to $\phi(A)_1$. Thus $A_1 = \phi(A)_1$. We also notice that the *falls*[1], or descents, of elements in $\phi(A)$ must *lie inside the cycles of* A. In the example $\phi(A) = < 4|26|1389|57 >$, $\phi(A)$ has 3 falls: $4 \to 2$, $6 \to 1$, and $9 \to 5$. So, for any fall $\phi(A)_i\phi(A)_{i+1}$ in $\phi(A)$, the following is true in A: $A_j = \phi(A)_i, A_m = \phi(A)_{i+1}$, *and* $j > m$. Clearly, the reverse is also true: any such j and m in A also gives a fall in $\phi(A)$. *If a permutation has* k *falls, it also has* $k+1$ *runs.* Therefore we have the following important theorem.

Theorem 8 *The number of permutations* $\phi(A)$ *of* n *with* $k+1$ *runs is equal to the number of permutations* A *whose two-line notation has* m *below* j *with* $j > m$ *in exactly* k *positions.*

Let us take one entry in Table 6.1 as an example: $\phi(A)_4 = < 4321 >$. The number

[1]In general, a_ia_{i+1} is a fall if $a_i > a_{i+1}$

Table 6.1: Bijection $\phi(A)$ for All Permutations of $n = 4$

A	$\phi(A)$
1234	1234
1243	1243
1423	1432
4123	4321
4132	4213
1432	1423
1342	1342
1324	1324
3124	3214
3142	3421
3412	3142
4312	4231
4321	4132
3421	3241
3241	3412
3214	3124
2314	2314
2341	2341
2431	2413
4231	4123
4213	4312
2413	2431
2143	2143
2134	2134

of runs in $\phi(A)_4$ is 4. We expect that there are 3 positions in two-line notation of $A_4 = < 4123 >$ where the element in the upper line is larger than the corresponding element in the below line:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ \wedge & \vee & \vee & \vee \\ 4 & 1 & 2 & 3 \end{pmatrix}$$

Theorem 8 and the mapping $\phi(A)$ in

Table 6.1 provides an important bijection according to which we can translate the random generation problem with given $RUN(A) = k$ into the random generation of permutations $\phi(A)$ whose two-line notation has $k - 1$ positions satisfying the above-mentioned $j > m$ property. Again, the latter problem is easier to solve than the original problem.

6.3.3 Longest Ascending Subsequence (LAS) and REM

LAS of a permutation A is the length of the longest ascending subsequence of A.

Definition 25 (LAS)

$$LAS(A) = max\{t|\ \exists i_1, \dots, i_t\ s.t.\ 1 \leq i_1 \leq \dots \leq i_t \leq n\ and A_{i_1} \leq \dots \leq A_{i_t}\}. \quad (6.7)$$

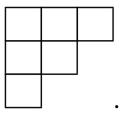
For example, one longest ascending subsequence of $A = < 463281795 >$ is $< 4689 >$, so $LAS(A) = 4$. Clearly, $1 \leq LAS(A) \leq n$. If A is sorted, $LAS(A) = n$. $LAS(A)$ attains its minimum value 1 for a list in reverse order. A related measure is $REM(A) = n - LAS(A)$, which indicates how many numbers have to be removed from A to make a sorted list. A large LAS (small REM) guarantees little local disorders that INV and RUN cannot do. $LAS(A)$ and $REM(A)$ can be computed in $O(n \log n)$ [Fre75].

To solve the random generation problem for LAS or REM, we first translate it into an easier problem using the following combinatorial properties of permutations.

Integer Partitions and Young Diagram

An *integer partition* of n is a k-tuple of positive numbers $\lambda = (\lambda_1, \ldots, \lambda_k)$ if $(\lambda_1 + \ldots + \lambda_k) = n$ and $\lambda_1 \geq \ldots \geq \lambda_k \geq 1$. k is the number of parts of λ. For example, a partition of 6 into 3 parts can be either $(3, 2, 1)$ or $(2, 2, 2)$. We can also describe them by giving the number of times that a part i occurs, called the multiplicity of i, like this: $3^1 2^1 1^1$ or 2^3.

It is useful to picture a partition as an array of squares, or cells, left justified, in decreasing order. For example, $\lambda = (3, 2, 1)$ is given by


.

Such diagrams are called *Young diagrams.*

Young Tableaux

Let λ be a partition of n. A Young tableaux of *shape* λ is the Young diagram of λ with each cell filled with a positive number such that the entries of each row are in increasing order from left to right, and the entries of each column are increasing from top to bottom. For example,

1	3	5
2	4	
5		

is a Young tableaux of shape (3, 2, 1). For simplicity, we will just call it a tableaux.

Bijection about LAS: the Schensted Correspondence

Schensted correspondence is a bijection between multiset permutations and pairs of tableaux of the same shape. We will consider a special case of it on permutations instead of multiset permutations. Then we have the following theorem: [Knu81, SW86].

Theorem 9 *There is a one-to-one correspondence between the set of all permutations of* $\{1, 2, \ldots, n\}$ *and the set of ordered pairs* (P, Q) *of tableaux formed from*

$\{1, 2, \ldots, n\}$, *where* P *and* Q *have the same shape.*

For example, for $n = 3$ the bijection is as follows.

$$1\ 2\ 3 \Longleftrightarrow \begin{array}{|c|}\hline 1\\ \hline 2\\ \hline 3\\ \hline\end{array} \quad \begin{array}{|c|}\hline 1\\ \hline 2\\ \hline 3\\ \hline\end{array} \qquad 1\ 3\ 2 \Longleftrightarrow \begin{array}{|c|c|}\hline 1&3\\ \hline 2\\ \cline{1-1}\end{array} \quad \begin{array}{|c|c|}\hline 1&3\\ \hline 2\\ \cline{1-1}\end{array}$$

$$2\ 1\ 3 \Longleftrightarrow \begin{array}{|c|c|}\hline 1&2\\ \hline 3\\ \cline{1-1}\end{array} \quad \begin{array}{|c|c|}\hline 1&2\\ \hline 3\\ \cline{1-1}\end{array} \qquad 2\ 3\ 1 \Longleftrightarrow \begin{array}{|c|c|}\hline 1&2\\ \hline 3\\ \cline{1-1}\end{array} \quad \begin{array}{|c|c|}\hline 1&3\\ \hline 2\\ \cline{1-1}\end{array}$$

$$3\ 1\ 2 \Longleftrightarrow \begin{array}{|c|c|}\hline 1&3\\ \hline 2\\ \cline{1-1}\end{array} \quad \begin{array}{|c|c|}\hline 1&2\\ \hline 3\\ \cline{1-1}\end{array} \qquad 2\ 3\ 1 \Longleftrightarrow \begin{array}{|c|c|c|}\hline 1&2&3\\ \hline\end{array} \quad \begin{array}{|c|c|c|}\hline 1&2&3\\ \hline\end{array}$$

Because the Schensted correspondence is a bijection, we can build (P, Q) from a permutation A and recover A from (P, Q). These algorithms are described in detail in [Knu81, SW86].

One remarkable property of the Schensted correspondence is that the length of the longest ascending subsequence of A is hidden in the shape of the corresponding tableaux P or Q. We have the following theorem.

Theorem 10 *Let* A *be a permutation of* $\{1, 2, \ldots, n\}$ *and* (P, Q) *be the pair of tableaux determined by the Schensted correspondence. The number of rows of* P *(or* Q*) is the length of the longest ascending subsequence of* A.

Therefore, by using the Schensted correspondence we can translate the problem of generating random permutations with given size and a specified $LAS(A) = k$ into the problem of generating pairs of tableaux (P, Q) of k rows, and then recover the permutation from (P, Q).

6.4 Random Generation of Permutations with Given Degree of Presortedness

The random generation of test instances with some specified characteristics is important for empirical analysis of algorithms. In its abstract form, the problem is to generate elements of a finite set S of combinatorial structures at random from a uniform distribution. That is, all elements should be selected with the same chance. The exact random generation problem appears to be intractable for many important structures, so interest has focused on finding efficient randomized algorithms that approximate it. The brute-force algorithm of exhaustively listing all possible elements in S and choosing one at random is often not viable because S can be extremely large. In this section we apply a *Markov approach* [Sin93] to generate permutations with given degree of presortedness uniformly at random.

At the heart of this Markov approach is a simple algorithmic paradigm that simulates a Markov chain whose states are the set S of combinatorial structures and which converges to a uniform distribution over S.

6.4.1 The Markov Chain Approach to Random Generation Problems

Let I be a countable set [2]. Each $i \in I$ is called a *state* and I is called the *state-space*. We call that $\lambda = (\lambda_i : i \in I)$ is a *distribution* on I if $0 \leq \lambda_i \leq \infty$ and $\sum_{i \in I} \lambda_i = 1$. Let X be a random variable and suppose we set $\lambda_i = P(X = i)$. λ defines the distribution of X; i.e., X is a random variable that takes value i with probability λ_i. We say that a matrix $P = (p_{ij} : i, j \in I)$ is *stochastic* if every row $p_{ij} : j \in I$ is a distribution. P is called *doubly stochastic* if every column $p_{ij} : i \in I$ is also a distribution.

Definition 26 (Markov Chain) *We say that* $(X_n)_{n \geq 0}$ *is a* Markov chain *with* initial distribution λ *and* transition matrix P *if* X_0 *has distribution* λ *and the distribution of* X_t*, defining by the transition matrix* $(p_{ij} : j \in I)$*, given all previous values*

[2] For simplicity we will only consider finite sets .

$X_0, \ldots, X_{t-1}$ *only depends on* X_{t-1}*, i.e.,*

$$P(X_t|X_0, \ldots, X_{t-1}) = P(X_t|X_{t-1}) = p_{i_{t-1}i_t} \tag{6.8}$$

A natural question for a Markov chain is to determine the probability of X in a given state after some finite steps. For any positive integer s, the *s-step transition matrix* is simply the power $P^s = p_{ij}^{(s)} = P(X_{t+s} = j|X_t = i)$, independent of t. A Markov chain is called *irreducible* if for each pair of states $i, j \in I$ there is a s such that $p_{ij}^{(s)} > 0$, i.e., any state can be reached from any other state in a finite number of steps. A Markov chain is called *aperiodic* if the greatest denominator of all s such that $p_{ii}^{(s)} > 0$ is 1, i.e., any state has a non-zero probability of staying unchanged. A Markov chain is called *ergodic* if there exists a distribution π over I such that

$$\lim_{s\to\infty} p_{ij}^{(s)} = \pi_j \quad \forall i, j \in I$$

i.e., the chain converges to the *stationary distribution* π.

Necessary and sufficient conditions for a chain being ergodic are that it should be irreducible and aperiodic, i.e., any finite Markov chain that is irreducible and aperiodic is ergodic. *An ergodic Markov chain with a doubly stochastic transition matrix has a stationary distribution that is uniform.*

Based on the above theoretical results, we can solve the random generation problem by constructing an ergodic finite Markov chain whose states correspond to elements in I and whose transition matrix is doubly stochastic. If the chain is allowed to evolve in time, the distribution of the final state tends asymptotically to a uniform stationary distribution over I, i.e., all elements in I are generated randomly with the same probability $\frac{1}{|I|}$. By simulating the chain for a sufficient number of steps and outputting the final state, we are able to generate elements of I from a distribution which is arbitrarily close to uniform.

For example, the following card-shuffling process based on random transpositions generates a permutation of all cards uniformly at random [Sin93]. For a natural number n, let S_n denote the set of permutations of the set $[n] = \{0, \ldots, n-1\}$.

Consider a deck of n cards labelled with the elements of the set $[n]$, and identify $\rho = (\rho_0, \ldots, \rho_{n-1}) \in S_n$ with ordering of the deck in which the ith card from the top if ρ_i. We define a Markov chain with state-space S_n, and the transitions are made by picking two cards from the deck at random and exchanging them. We incorporate a self-loop probability for each state. This chain is irreducible because a path exists between any two permutations. It is also aperiodic because of the non-zero self-loop probability. Moreover, the transition matrix of the chain is doubly stochastic. Therefore, the stationary distribution is uniform. It has also been shown that the chain is *rapidly mixing*, i.e., it is close to stationary after visiting only a small fractional of its state space. More specifically, the number of simulation steps required to achieve tolerance ϵ is $O(n^4(n \log n + \log \epsilon^{-1}))$.

6.4.2 The Random Generation Problem for INV

Let n and k be two integer numbers, $n > 0$ and $0 \leq k \leq \frac{n(n-1)}{2}$. The problem is to generate a permutation A of the set $[n] = \{1, \ldots, n\}$ uniformly at random such that $|A| = n$ and $INV(A) = k$. Using the bijection between a permutation and its inversion table we can translate this problem into generating the corresponding inversion tables uniformly at random. Recall that the inversion table $B = < b_1, b_2, \ldots, b_n >$ of permutation $A = < a_1, a_2, \ldots, a_n >$ is obtained by letting b_j be the number of elements to the left of j that are greater than j. Hence the problem is to generate the inversion table B uniformly at random such that $|B| = n$, $\sum_{i=1}^{n-1} b_i = k$, and b_i satisfies the following properties

$$0 \leq b_1 \leq n-1,\ 0 \leq b_2 \leq n-2,\ \ldots,\ 0 \leq b_{n-1} \leq 1,\ b_n = 0.$$

For example, for $n = 4$ and $k = 3$, the permutations of length 4 and INV 3 and their corresponding inversion tables are listed in Table 6.2.

To state it more clearly, we can think of each b_i as a bin with a capacity of $n-i$ and the task as filling these n bins with k balls without violating the capacity constraints. We construct a Markov chain whose states are all possible inversion tables of $< n, k >$, or all possible bin-filling solutions. The transitions are made by picking up two bins i

Table 6.2: Permutations and the Inversion Tables for $n = 4$, $k = 3$

Permutations A	**Inversion Tables B**
4 1 2 3	1 1 1 0
2 4 1 3	2 0 1 0
1 4 3 2	0 2 1 0
3 2 1 4	2 1 0 0
3 1 4 2	1 2 0 0
2 3 4 1	3 0 0 0

and j at random, and transferring one ball from b_i to b_j whenever possible. If $i = j$, or $|b_i| = 0$, or b_j is already full, then do nothing. Therefore we automatically have a non-zero self-loop probability. We start a random walk like this in the state-space, and after enough steps stop and return the current state (an inversion table) and convert it into a permutation. The algorithm is listed in Figure 6.1. We now prove the algorithm can generate a permutation with number of inversions of k uniformly at random.

Theorem 11 *The Markov chain generated by algorithm 6.1 is irreducible.*

Proof: A Markov chain is irreducible if any two states of the chain intercommunicate, that is, there is a path of non-zero probability from any state reaching another state. We show this by constructing a canonical path from any legal inversion table $B = < b_1, b_2, \ldots, b_n >$ with $\sum_{i=1}^{n} b_i = k$ to the inversion table $U = < k, 0, \ldots, 0 >$. From any $B \neq U$, there is a non-zero probability of picking up a pair $< i, j >$ such that $i = 1$, $j > 1$ and $B_j \geq 1$. So we will have to decrease b_j and increase b_1 by 1 at each pick until we reach U. The reverse process is also true. Therefore there always exists a path of non-zero probability between any two states of the chain. This shows the chain is irreducible. $\square$

Theorem 12 *The Markov chain generated by algorithm 6.1 is aperiodic.*

Proof: A Markov chain is aperiodic if it is always possible for the chain to stay at the same state. This is easy to see from step 2 of the algorithm. The chain will

```
Input:   length of the permutation (n), number of inversions (k),
         number of iterations (N), $0 \leq k \leq \frac{n(n-1)}{2}$.
Output:  A permutation $A$ of $\{1, \ldots, n\}$ and $INV(A) = k$.

Step 1:  Initialize a legal inversion table $B =< b_1, b_2, \ldots, b_n >$
         such that $\sum_{i=1}^{n} b_i = k$.

Step 2:  Repeat the next loop $N$ times:
             Pick up two numbers $1 \leq i, j \leq n$ uniformly at random;
             If $i = j$, or $b_i = 0$, or $b_j = n - j$, then do nothing;
             else $b_i = b_i - 1$, $b_j = b_j + 1$ .

Step 3:  Convert the current inversion table $B$ into the corresponding
         permutation $A$ and return $A$.
```

Figure 6.1: Algorithm: Randomly Generating Permutation with INV = k

stay unchanged when the "if" condition in step 2 is not satisfied (there is a self-loop probability greater than zero). □

Theorem 13 *The transition matrix defined by algorithm 6.1 is doubly stochastic.*

Proof: We have constructed the chain to have a symmetric transition matrix. Paths between two states have the same probability in both directions. There is a self-loop probability in step 2 that equals to one minus the probability of other moves. For example, the transition matrix for $n = 4$, $k = 3$ is shown in Table 6.3. Therefore, rows and columns of the transition matrix both add to one. □

Theorem 14 *The Markov chain generated by algorithm 6.1 is ergodic and it converges to a uniform stationary distribution.*

Proof: Following from the previous theorems, the chain is irreducible, aperiodic, and doubly stochastic; therefore it is ergodic and its stationary distribution is a uniform one. □

Table 6.3: Transition Matrix Defined by Algorithm 6.1

	0210	**1110**	**1200**	**2010**	**2100**	**3000**
0210	7/9	1/9	1/9	0	0	0
1110	1/9	5/9	1/9	1/9	1/9	0
1200	1/9	1/9	7/9	0	0	0
2010	0	1/9	0	6/9	1/9	1/9
2100	0	1/9	0	1/9	6/9	1/9
3000	0	0	0	1/9	1/9	7/9

6.4.3 The Random Generation Problem for RUN

Here we have $n > 0$ and $1 \leq k \leq n$. The problem is to generate a permutation A of the set $[n] = \{1, \ldots, n\}$ uniformly at random such that $|A| = n$ and $RUN(A) = k$. This time we use the bijection for RUN as shown in Table 6.1 to translate this problem into generating the corresponding two-line notation permutations uniformly at random. Recall that there is a one-to-one correspondence between a permutation with k runs and the permutation whose two-line notation has exactly $k-1$ positions where the upper-line element j is larger than the lower-line element m. Consider the following example again:

$$u = \begin{pmatrix} 1 & 2 & 3 & 4 \\ \wedge & \vee & \vee & \vee \\ 4 & 1 & 2 & 3 \end{pmatrix} \Leftrightarrow v = \phi(u) = < \ 4\ 3\ 2\ 1 \ > .$$

To generate v with $RUN(v) = 4$, we can first generate $u =< \ 4\ 1\ 2\ 3 \ >$ whose two-line notation has 3 positions where $j > m$, and then recover $v = \phi(u)$ from u. Note that the upper-line in the two line notation is always $< \ 1\ 2\ 3\ 4 \ >$.

Again we construct a Markov chain with states that correspond to all possible permutations and whose two-line notations have exactly $k-1$ positions, where the upper-line number is larger than the lower-line number. The transitions are made by randomly picking up two columns of the two-line notation permutation and exchanging the lower-line numbers if doing so does not change the property of having $k-1$ positions where the upper-line number is larger. We start a random walk like this

```
Input:   length of the permutation (n), number of runs (k),
         number of iterations (N), $1 \leq k \leq n$.
Output:  A permutation $A$ of $\{1, \ldots, n\}$ and $RUN(A) = k$.

Step 1: Initialize a legal permutation $B =< b_1, b_2, \ldots, b_n >$ as the lower-line
        of the two-line notation permutation such that there are only $k - 1$
        positions in $B$ where $b_i < i$;

Step 2: Repeat the next loop $N$ times:
        Pick up two numbers $1 \leq i, j \leq n$ uniformly at random;
        If exchanging $b_i$ and $b_j$ destroys the property, then do nothing;
        else exchange $b_i$ and $b_j$.

Step 3: Convert the current two-line notation permutation $B$ into the
        corresponding permutation $A = \phi(B)$ and return $A$.
```

Figure 6.2: Algorithm: Randomly Generating Permutation with RUN = k

in the state-space of the chain; after enough steps, stop and return to the current state (a permutation in two-line notation) and convert it back to the permutation according to the bijection ϕ. The algorithm is listed in Figure 6.2. The proof of the ergodic property is similar to that of algorithm 6.1.

6.4.4 The Random Generation Problem for LAS and REM

The problem is to generate a permutation A of the set $[n] = \{1, \ldots, n\}$ uniformly at random such that $|A| = n$ and $LAS(A) = k$ or $REM(A) = n - k$. Here we have $n > 0$ and $1 \leq k \leq n$.

The bijection used for the random generation problem for LAS is the correspondence between the set of all permutations of $\{1, 2, \ldots, n\}$ whose LAS are k and the set of ordered pairs (P, Q) of tableaux formed from $\{1, 2, \ldots, n\}$, where P and Q have the same shape and the number of rows of P (or Q) is k. This problem is a little more complicated because we need to take care of both the shape and the number of

```
Input:   length of the permutation (n), number of runs (k),
         number of iterations (N), $1 \le k \le n$.
Output:  A permutation $A$ of $\{1, \ldots, n\}$ and $LAS(A) = k$.

Step 1:  Randomly generate a shape $\lambda$ of $k$ rows;
         Initialize a pair of Young tableaux $(P_0, Q_0)$ of shape $\lambda$;
         boolean CHANGESHAPE = false.

Step 2:  Repeat the next loop $N$ times:
         if(CHANGESHAPE) then randomly generate shape $\lambda$ of $k$ rows,
                 generate $(P_0, Q_0)$ of shape $\lambda$ uniformly at random;
         else randomly generate a new pair of $(P, Q)$ of shape $\lambda$,
                 set CHANGESHAPE to true if $S(P_0, Q_0)$ is re-generated;

Step 3:  Convert the current pair of tableaux $(P, Q)$ to the
         corresponding permutation $A = S(P, Q)$ and return $A$.
```

Figure 6.3: Algorithm: Randomly Generating Permutation with LAS = k

rows of the tableaux.

The state-space of the chain contains all tableaux of k rows. Note that they may have different shapes. Each shape represents a subspace of the whole state-space. In the initialization step, we first generate a shape λ_0 of k rows by randomly partitioning n into k parts. Then we select two Young tableaux P_0 and Q_0 of shape λ_0 uniformly at random, i.e., filling the cells of the shape. This is done by inserting the number n into a corner position of shape λ with the right probability, then inserting $n-1$ into a corner of the remaining shape, etc. The random selection algorithm is given in [NW78]. We then start the random walk in the state-space of the chain from P_0 and Q_0. At each iteration we use a random walk to explore the subspace defined by the current shape. The transitions are made by randomly selecting (filling) a new tableaux of the current shape. We change the shape when the initial point of this subspace is regenerated. Simulating this process for a sufficient number of steps, we

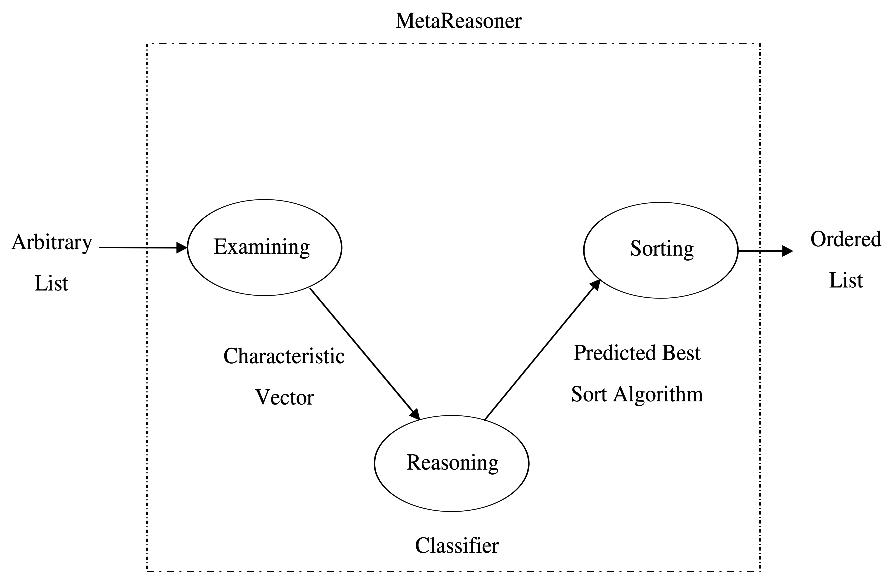


Figure 6.4: Algorithm Selection Meta-Reasoner for Sorting

stop and convert the current state, a pair of tableaux (P, Q) with the same shape of k rows, to the corresponding permutation $A = S(P, Q)$ defined by the Schensted correspondence S, and then return A. The algorithm is described in Figure 6.3. The proof of the ergodic property is also similar to that of algorithm 6.1.

6.5 Experiment Setups and Environment

The overall goal of our experiments is to use the machine learning-based approach to investigate how instance features affect the performance of various sorting algorithms. The algorithm space consists of five algorithms: insertion sort, shellsort, heapsort, mergesort, and quicksort. The feature space consists of instance size and three measures of presortedness: INV, RUN and REM. The working procedure of the sorting algorithm selection meta-reasoner is illustrated in Figure 6.4.

The experiments can be divided into three phases: *data preparation*, *model induction*, and *model evaluation*. In data preparation, we first generate a set of training permutations with different characteristic values. We then run all sorting algorithms on these permutations and collect the algorithm performance data to get a simple matrix like Table 6.4. We call this the *training data.* We use each algorithm's running time to measure its performance. The algorithm that consumes the least time in sorting the instances is labelled as the *best.* For learning algorithms that require discrete data, we need to first discretize the training data. In model induction, we run various machine learning algorithms on the training data to induce the predictive algorithm selection models. We will consider three basic models: decision tree learning (C4.5), the naive Bayes classifier, and Bayesian network learning (K2). They are all used as classifiers to catalog instances by the best algorithm to solve them. Finally, we evaluate the learned classifiers and the overall performance of the algorithm selection system using various test datasets.

Most of our learning experiments are conducted in Weka 3, an open-source machine learning software in Java. Weka [WF99] is a collection of machine learning algorithms for solving real-world data mining problems. It contains tools for data preprocessing, classification, regression, clustering, association rules, evaluation, and visualization. For decision tree learning (C4.5) and the naive Bayes learning, we use Weka's implementations. For Bayesian network learning we have implemented our own K2 and managed to plug it into Weka so that we can use Weka's evaluation modules. We also use Hugin's implementation of the clique tree propagation algorithm to build the methods for classifying new instances. In our experiments we use the IBM High Resolution Time Stamp Facility to measure algorithm running time in microseconds. Our hardware platform includes two Quad 450 Xeon Linux machines and an 1GHz AMD Athlon WinXP machine. Most training instances are generated on linux machines and all learning and evaluations are conducted on the Windows machine.

Table 6.4: Basic Experiment Setup: Instances vs. Algorithms

	Feature 1	...	Feature m	Algo. 1	...	Algo. n	Best
Instance 1							
Instance 2							
...							
Instance k							

6.6 Experimental Results and Evaluation: The Induction of Predictive Algorithm Selection Models for Sorting

In this section we report the results of a series of learning experiments on sorting algorithm selection. The first experiment is designed to verify some well-known observations about sorting algorithm performance on some specific datasets. The second one is to determine which measure of disorder is the best feature for sorting algorithm selection. The third applies various learning algorithms on the same training dataset to determine which is the best model for sorting algorithm selection. The fourth experiment evaluates overall performance of the learned model as a meta-level reasoner.

6.6.1 The Training Datasets

We have prepared 4 training datasets for our learning experiments. Let us call them D_{sort1}, D_{sort2}, D_{sort3}, and D_{sort4}. The first three datasets contain instances of specific characteristics. D_{sort1} contains all possible permutations of $\{1, 2, 3, 4, 5, 6, 7, 8\}$, totaling 40,320 instances. This represents small size sorting instances. D_{sort2} contains 500 nearly-sorted permutations of size 1,000. D_{sort3} contains 500 randomly disordered permutations[3] of size 1,000. The fourth dataset, D_{sort4}, is designed to be the most representative training dataset. It contains a total of 1,875 permutations of size varying from 10 to 1,000 and presortedness measures varying from 0 to 1. More specifically,

[3]The random generation algorithm used is Algorithm 235(Random Permutation) in [Dur64].

it is composed of the following seven smaller sub-datasets. The first sub-dataset contains 525 permutations that are generated by setting the three presortedness measures to $\{0.1, 0.3, 0.5, 0.7, 0.9\}$ and the sizes to $\{10, 20, 50, 100, 200, 500, 1000\}$. The second one includes 100 permutations of reverse order with size increasing by 10 from 10 to 1000. The third one contains 100 totally ordered permutations. The fourth one contains 100 permutations in the form of $\{2, 1, 4, 3, 6, 5, 8, 7, 10, 9\}$, which all have small numbers of inversions but large numbers of runs. The fifth sub-dataset contains 100 permutations in the form of $\{6, 7, 8, 9, 10, 1, 2, 3, 4, 5\}$. They all have small numbers of runs but large numbers of inversions. The sixth one contains 500 randomly generated permutations of size 1000, and the seventh contains 425 nearly-sorted permutations of size 1000. For each permutation, we first compute three presortedness measures INV, RUN, and REM and record the computational time of each measure. We then run all 5 sorting algorithms on that permutation, record the running time of each algorithm and the winner, i.e., the one that takes the least time to sort the instance.

6.6.2 Experiment 1: Verifying Sorting Algorithm Performance on Some Specific Datasets

In this experiment we apply decision tree learning algorithm C4.5 on D_{sort1}, D_{sort2} and D_{sort3} to see what it can learn about these datasets with specific characteristics.

C4.5 on D_{sort1}: Instances of Small Sizes

D_{sort1} contains all possible permutations of $\{1, 2, 3, 4, 5, 6, 7, 8\}$; totaling 40,320 instances. For permutations of size 8, $INV \in [0, 28]$, $RUN \in [1, 8]$, $REM \in [0, 7]$. Without loss of generality, these presortedness measures have been normalized to [0, 1].. Distributions of each measure taking different values are illustrated in Figure 6.5.

Each of the three measures correspond to an intuitive idea of presortedness. INV measures global presortedness, RUN measures local presortedness, and REM seems to combine elements of both. However, they are not quite independent of each other. Theoretically, comparing these measures is by no means an easy job. Figure 6.6 shows the relationships between INV, RUN and REM values of all permutations of

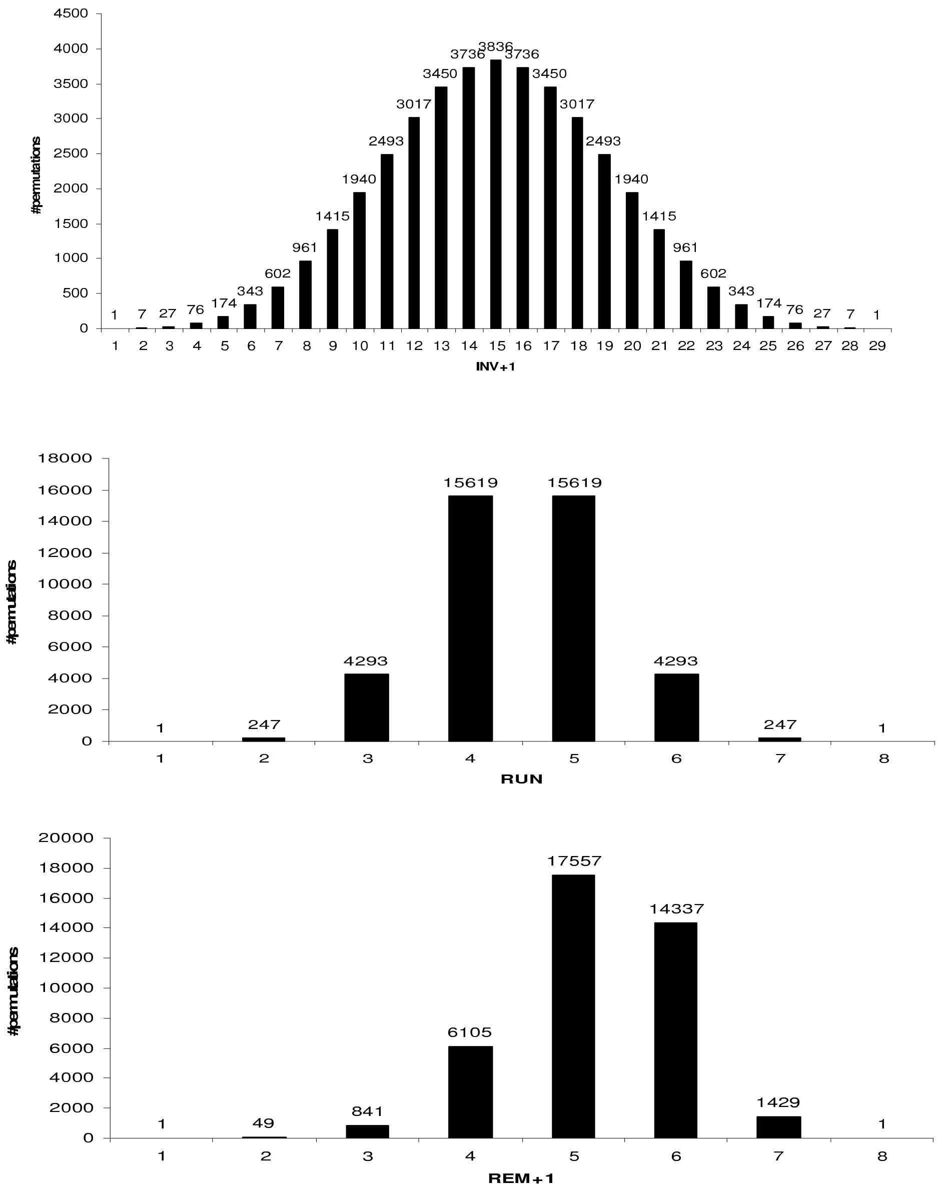


Figure 6.5: Number of Permutations vs. Presortedness Measures (size = 8)

Table 6.5: Statistics of INV, RUN and REM Values of D_{sort1}

	INV	RUN	REM
Mean	0.50	0.50	0.60
StdDev	0.14	0.12	0.12

size 8. The statistics of each measure are listed in Table 6.5. When preparing the training data, we compute these three measures and run all five sorting algorithms for all permutations. We also record the computational time of each measure and the running time of each algorithm. The average times are shown in Figure 6.7. From the figure we can see that on average insertion sort is the fastest algorithm, and INV takes the longest time to compute.

Table 6.6: Training Dataset D_{sort1}

Permutation	Size	Inv	Run	Rem	Winner
1	8	0.0	0.0	0.0	shell
2	8	0.36	0.14	0.14	shell
3	8	0.07	0.14	0.14	insertion
...	...	...	...	...	...
40,320	8	1.0	1.0	1.0	shell

The final training dataset fed into the learning algorithm consists of five attributes: *size, inv, run, rem*, and *winner*, where the winner is the *target attribute* to be predicted based on other attributes of the permutation in question. This is a typical supervised learning, or classification, problem.

By applying C4.5 on D_{sort1} we get the following result, as shown in Figure 6.8. The learned decision tree has only one leaf: *insertion*. This means that the learner "thinks" the best algorithm selection strategy for D_{sort1} is just committing to the insertion sort algorithm. The classification accuracy is 68.4449%. The confusion matrix shows the numbers of misclassified instances of each class. The experimental

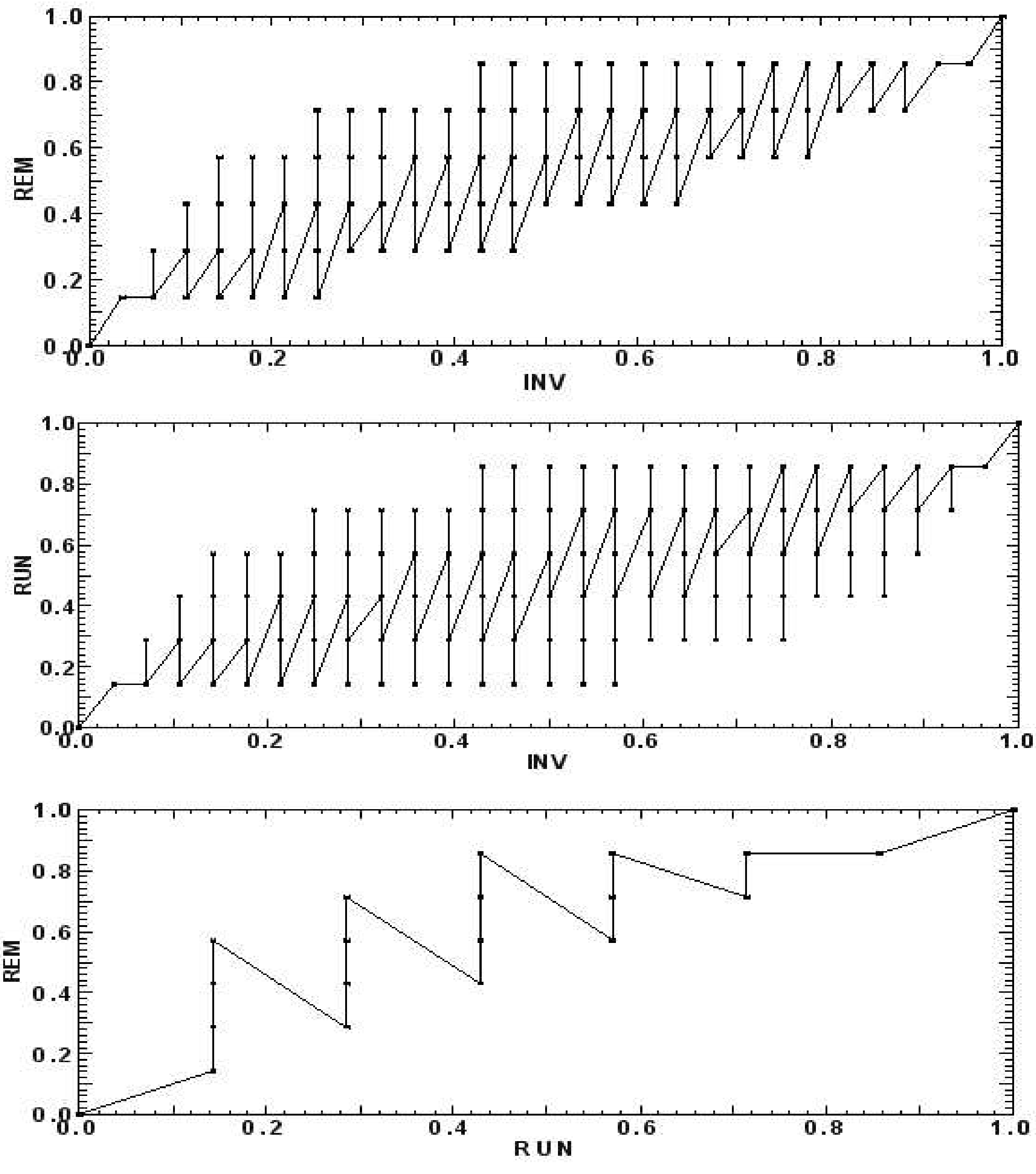


Figure 6.6: The Relationships among INV, RUN and REM for Permutations of Size 8

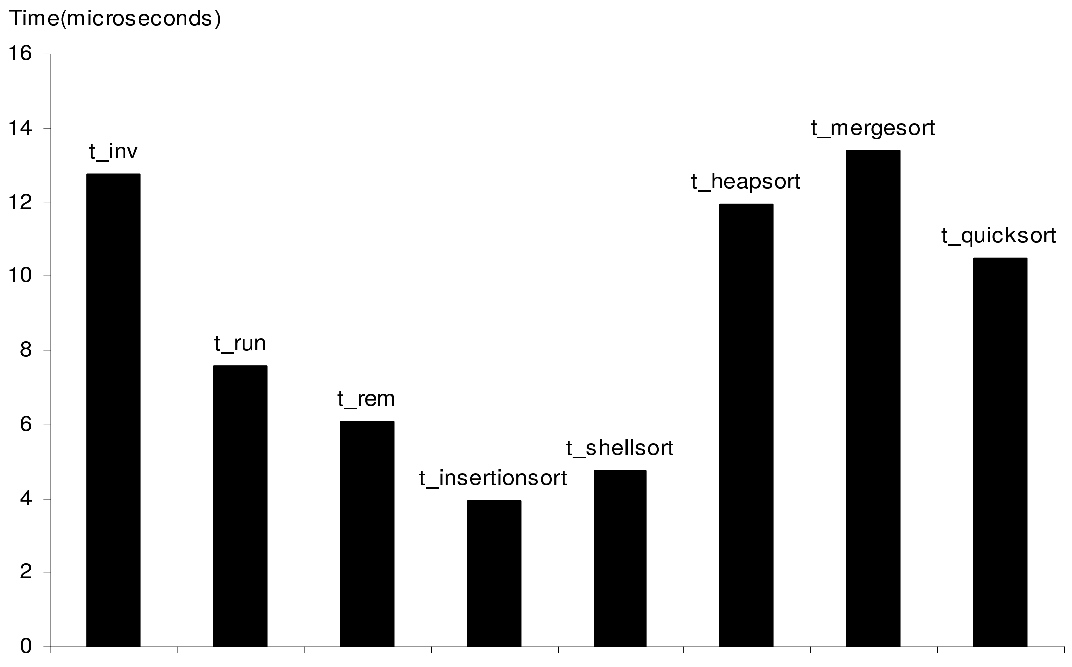


Figure 6.7: The Computational Time of All Measures and the Running Time of All Sort Algorithms on D_{sort1}

result verifies the following observation regarding to sorting algorithm selection for small permutations.

Observation 1 *Insertion sort is best for sorting small permutations.*

C4.5 on D_{sort2}: Nearly Ordered Instances

D_{sort2} contains 500 nearly-sorted permutations of size 1,000. These instances are generated by swapping 10 pairs of randomly selected elements from the totally ordered permutation. The statistics of D_{sort2} are shown in Table 6.7. By applying C4.5 on D_{sort2} we also get a decision tree of one leaf, as shown in Figure 6.9: *insertion.* The classification accuracy is 100% on D_{sort1}. This result verifies the following observation of sorting algorithm selection on nearly-sorted permutations:

=== Run information ===

Scheme: weka.classifiers.j48.J48 -C 0.25 -M 2
Relation: sorting
Instances: 40320
Attributes: 5 size inv runs rem winner
Test mode: 10-fold cross-validation

=== Classifier model (full training set) ===

J48 pruned tree

: insertion (40320.0/12723.0)

Number of Leaves : 1
Size of the tree : 1
Time taken to build model: 11.41 seconds

=== Stratified cross-validation ===

Correctly Classified Instances 27597 68.4449%
Incorrectly Classified Instances 12723 31.5551%
Kappa statistic 0
Mean absolute error 0.1923
Root mean squared error 0.3101
Total Number of Instances 40320

=== Confusion Matrix ===

$$\begin{pmatrix} a & b & c & d & e & <-- & classified\ as \\ 27597 & 0 & 0 & 0 & 0 & | & a = insertion \\ 7850 & 0 & 0 & 0 & 0 & | & b = shell \\ 289 & 0 & 0 & 0 & 0 & | & c = heap \\ 4 & 0 & 0 & 0 & 0 & | & d = merge \\ 4580 & 0 & 0 & 0 & 0 & | & e = quick \end{pmatrix}$$

Figure 6.8: C4.5 Result on D_{sort1}

Table 6.7: Statistics of INV, RUN and REM Values of D_{sort2}

	INV	RUN	REM
Minimum	2.0E-6	0.0010	0.0010
Maximum	0.0040	0.0020	0.0020
Mean	0.0014	0.0020	0.0020
StdDev	9.5E-4	4.5E-5	4.48E-5

Table 6.8: Statistics of INV, RUN and REM Values of D_{sort3}

	INV	RUN	REM
Minimum	0.47	0.47	0.93
Maximum	0.53	0.52	0.95
Mean	0.50	0.50	0.94
StdDev	0.01	0.009	0.003

Observation 2 *Insertion sort is best for sorting nearly sorted permutations.*

C4.5 on D_{sort3}: Random Instances

D_{sort3} contains 500 random permutations of size 1,000. They are generated by Durstenfeld's random permutation generation algorithm [Dur64]. The statistics of D_{sort3} are shown in Table 6.8. Applying C4.5 on D_{sort3} we again get a decision tree of one leaf as shown in Figure 6.10. However, this time the selected algorithm is *quick sort*. The classification accuracy is 95.2% on D_{sort3}. This result verifies the following observation of sorting algorithm selection on random disordered permutations:

Observation 3 *Quick sort is best for sorting random disordered permutations.*

```
=== Run information ===

Scheme: weka.classifiers.j48.J48 -C 0.25 -M 2
Relation: sorting
Instances: 500
Attributes: 5 size inv runs rem winner
Test mode: 10-fold cross-validation

=== Classifier model (full training set) ===

J48 pruned tree
————————————

: insertion (500.0)
Number of Leaves : 1
Size of the tree : 1
Time taken to build model: 0 seconds

=== Stratified cross-validation ===

Correctly Classified Instances 500 100%
Incorrectly Classified Instances 0 0%
Kappa statistic 1
Mean absolute error 0
Root mean squared error 0
Total Number of Instances 500

=== Confusion Matrix ===
```

$$\begin{pmatrix} a & b & c & d & e & <-- & classified\ \ as \\ 500 & 0 & 0 & 0 & 0 & | & a = insertion \\ 0 & 0 & 0 & 0 & 0 & | & b = shell \\ 0 & 0 & 0 & 0 & 0 & | & c = heap \\ 0 & 0 & 0 & 0 & 0 & | & d = merge \\ 0 & 0 & 0 & 0 & 0 & | & e = quick \end{pmatrix}$$

Figure 6.9: C4.5 Result on D_{sort2}

```
=== Run information ===

Scheme: weka.classifiers.j48.J48 -C 0.25 -M 2
Relation: sorting
Instances: 500
Attributes: 5 size inv runs rem winner
Test mode: 10-fold cross-validation

=== Classifier model (full training set) ===

J48 pruned tree
————————————

: quick (500.0/24.0)
Number of Leaves : 1
Size of the tree : 1
Time taken to build model: 0.24 seconds

=== Stratified cross-validation ===

Correctly Classified Instances 476 95.2%
Incorrectly Classified Instances 24 4.8%
Kappa statistic 0
Mean absolute error 0.037
Root mean squared error 01361
Total Number of Instances 500

=== Confusion Matrix ===
```

$$\begin{pmatrix} a & b & c & d & e & <-- & classified\ as \\ 0 & 0 & 0 & 0 & 0 & | & a = insertion \\ 0 & 0 & 0 & 0 & 1 & | & b = shell \\ 0 & 0 & 0 & 0 & 15 & | & c = heap \\ 0 & 0 & 0 & 0 & 8 & | & d = merge \\ 0 & 0 & 0 & 0 & 476 & | & e = quick \end{pmatrix}$$

Figure 6.10: C4.5 Result on D_{sort3}

Table 6.9: Statistics of INV, RUN and REM Values of D_{sort4}

	size	INV	RUN	REM
Minimum	10	0.0	0.0	0.0
Maximum	1000	1.0	1.0	1.0
Mean	556.80	0.35	0.35	0.54
StdDev	380.01	0.31	0.30	0.40

6.6.3 Experiment 2: Determining the Best Feature for Sorting Algorithm Selection

In this experiment we use a representative training dataset, D_{sort4}, to determine the best feature for sorting algorithm selection. D_{sort4} contains 1,875 permutations with size varying from 10 to 1,000 and presortedness measures varying from 0 to 1. The distributions of size and presortedness measures in D_{sort4} are visualized in Figure 6.11 and Figure 6.12. Their statistics are listed in Table 6.9.

Wrapper-based Feature Selection

We first apply a GA-wrapped C4.5 feature selection classifier on D_{sort4} to see which feature subset is the best. The wrapper uses C4.5 as the evaluation classifier to evaluate the fitness of feature subsets. A simple genetic algorithm is used to search the attribute space. Both the population size and the number of generations are 20. The crossover probability is 0.6 and the mutation probability is 0.033. Figure 6.13 and Figure 6.14 shows the configurations and the running information of the wrapper-based classifier. The GA converges at generation 13 and outputs the result subset: $\{1, 2, 3, 4\}$. Thus, all features are selected, i.e., no other feature subset is better than the complete feature set.

Individual Feature Comparison

A feature is good for algorithm selection if it takes a short time to compute and has a higher classification accuracy. In order to investigate which feature is the best

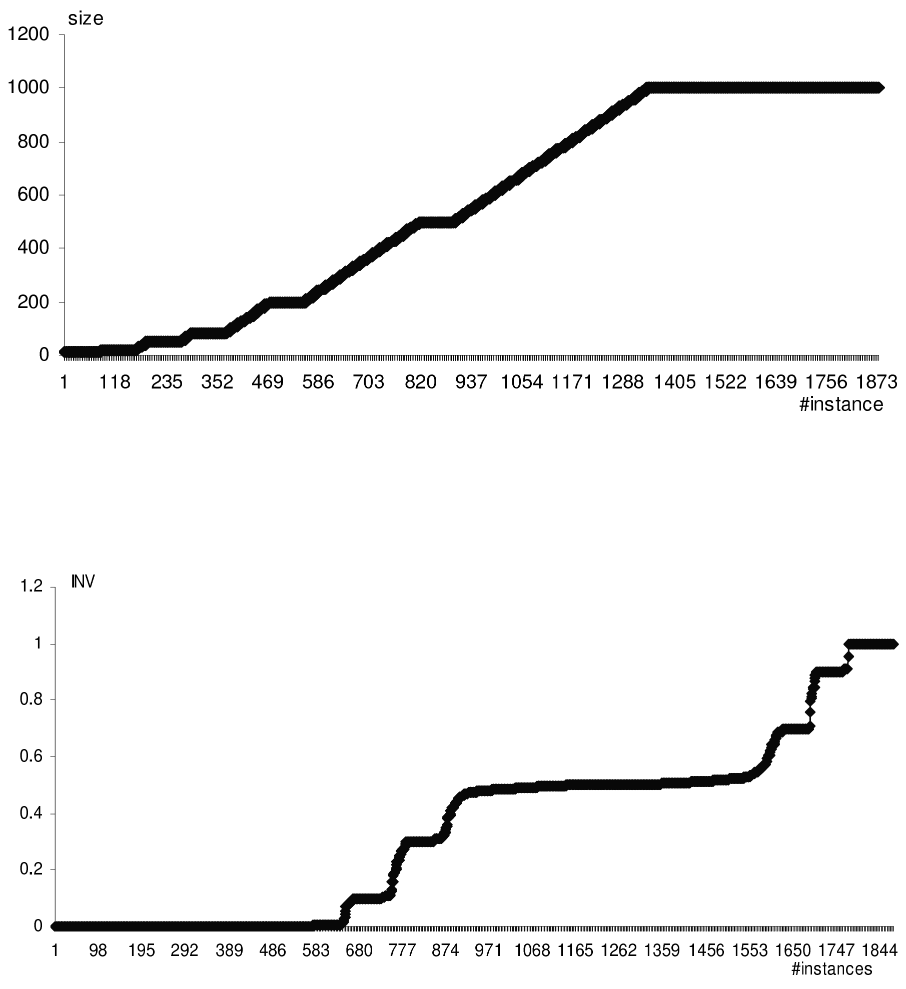
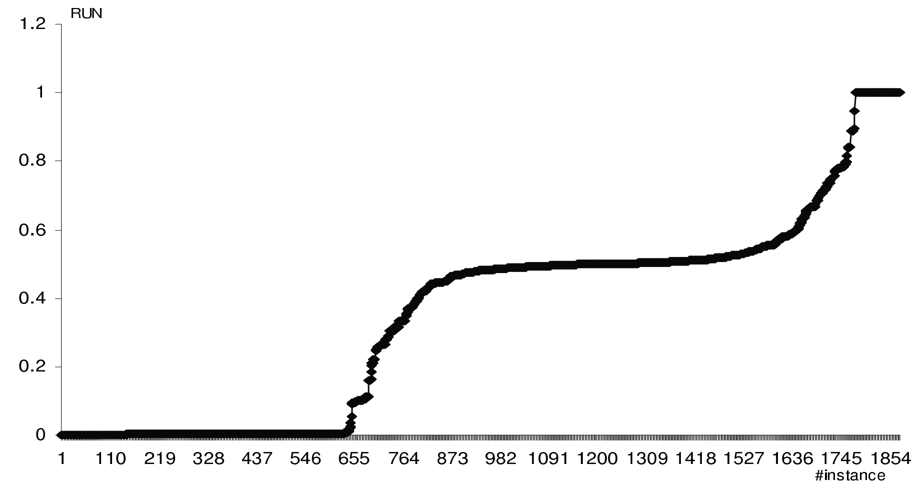


Figure 6.11: The Distribution of Size and INV of D_{sort4}



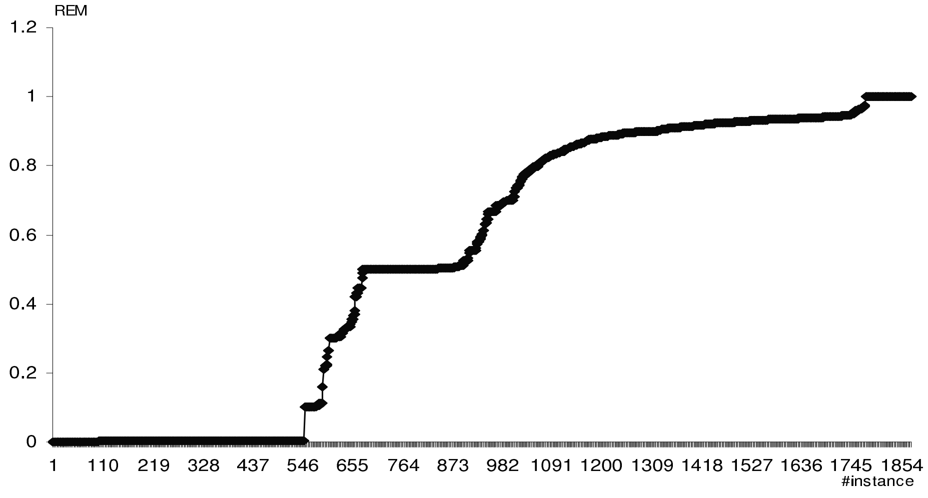


Figure 6.12: The Distribution of RUN and REM of D_{sort4}

```
=== Run information ===
Scheme: weka.classifiers.AttributeSelectedClassifier -B "weka.classifiers.j48.J48 -C
0.25 -M 2" -E "weka.attributeSelection.WrapperSubsetEval -B weka.classifiers.j48
.J48 -F 5 -T 0.01 -S 1 – -C 0.25 -M 2" -S "weka.attributeSelection.GeneticSearch
-Z 20 -G 20 -C 0.6 -M 0.033 -R 20 -S 1"
Relation: sorting
Instances: 1875
Attributes: 5
  size
  inv
  runs
  rem
  best
Test mode: 10-fold cross-validation
=== Classifier model (full training set) ===
AttributeSelectedClassifier:
=== Attribute Selection on all input data ===
Search Method: Genetic search.
  Population size: 20
  Number of generations: 20
  Probability of crossover: 0.6
Attribute Subset Evaluator (supervised, Class (nominal): 5 best):
  Wrapper Subset Evaluator
  Learning scheme: weka.classifiers.j48.J48
  Scheme options: -C 0.25 -M 2
  Accuracy estimation: classification error
  Number of folds for accuracy estimation: 5
```

Figure 6.13: Parameters of Attribute Selection GA-Wrapper on D_{sort4}

```
=== Run information ===
AttributeSelectedClassifier: GA-wrapped C4.5
   Initial population                        Generation: 13
   merit scaled subset                       merit scaled subset
   0.15093 0.11721 (2)                      0.05333 NaN (1 2 3 4)
   0.15093 0.11721 (2)                      0.05333 NaN (1 2 3 4)
   0.14987 0.11810 (4)                      0.05333 NaN (1 2 3 4)
   0.05333 0.19907 (1 2 3 4)                0.05333 NaN (1 2 3 4)
   0.06752 0.18717 (2 3 4)                  0.05333 NaN (1 2 3 4)
   0.06752 0.18717 (2 3 4)                  0.05333 NaN (1 2 3 4)
   0.13003 0.13474 (1 4)                    0.05333 NaN (1 2 3 4)
   0.05717 0.19585 (1 2 3)                  0.05333 NaN (1 2 3 4)
   0.12747 0.13689 (3)                      0.05333 NaN (1 2 3 4)
   0.15093 0.11721 (2)                      0.05333 NaN (1 2 3 4)
   0.05717 0.19585 (1 2 3)                  0.05333 NaN (1 2 3 4)
   0.07093 0.18431 (1 2)                    0.05333 NaN (1 2 3 4)
   0.29067 0.00000 (1)                      0.05333 NaN (1 2 3 4)
   0.12747 0.13689 (3)                      0.05333 NaN (1 2 3 4)
   0.14987 0.11810 (4)                      0.05333 NaN (1 2 3 4)
   0.29067 0.00000 (1)                      0.05333 NaN (1 2 3 4)
   0.13003 0.13474 (1 4)                    0.05333 NaN (1 2 3 4)
   0.12747 0.13689 (3)                      0.05333 NaN (1 2 3 4)
   0.15093 0.11721 (2)                      0.05333 NaN (1 2 3 4)
   0.15093 0.11721 (2)                      0.05333 NaN (1 2 3 4)
Selected attributes: (1,2,3,4) : 4
```

Figure 6.14: Result of Attribute Selection GA-Wrapper on D_{sort4}

for sorting algorithm selection, we first divide the 5-column training data D_{sort2} into three small training datasets D_{sort4A}, D_{sort4B} and D_{sort4C}. Each has 3 columns of data. D_{sort4A} contains $\{size, INV, winner\}$. D_{sort4B} contains $\{size, RUN, winner\}$. D_{sort4C} contains $\{size, REM, winner\}$. We run C4.5 on these three datasets to see which gives the highest classification accuracy. The result shows that classification accuracy of the model induced from D_{sort4A} is the highest: 94.51%. The confusion matrices are shown as follows.

$$INV, D_{sort4A}: \left(\begin{array}{llllllr} a & b & c & d & e & <-- & classified\ \ as \\ 780 & 9 & 0 & 0 & 2 & | & a = insertion \\ 12 & 161 & 0 & 0 & 37 & | & b = shell \\ 0 & 0 & 0 & 0 & 20 & | & c = heap \\ 2 & 1 & 0 & 0 & 5 & | & d = merge \\ 7 & 8 & 0 & 0 & 831 & | & e = quick \end{array}\right)$$

$$RUN, D_{sort4B}: \left(\begin{array}{lllllll} a & b & c & d & e & <-- & classified\ \ as \\ 749 & 17 & 0 & 0 & 25 & | & a = insertion \\ 12 & 159 & 0 & 0 & 39 & | & b = shell \\ 1 & 0 & 0 & 0 & 19 & | & c = heap \\ 3 & 1 & 0 & 0 & 4 & | & d = merge \\ 52 & 11 & 0 & 0 & 783 & | & e = quick \end{array}\right)$$

$$REM, D_{sort4C}: \left(\begin{array}{lllllll} a & b & c & d & e & <-- & classified\ \ as \\ 746 & 42 & 0 & 0 & 3 & | & a = insertion \\ 100 & 78 & 0 & 0 & 32 & | & b = shell \\ 0 & 0 & 0 & 0 & 20 & | & c = heap \\ 2 & 1 & 0 & 0 & 5 & | & d = merge \\ 15 & 11 & 0 & 0 & 820 & | & e = quick \end{array}\right)$$

We also compare the average computational time of each measure to see which one takes the least time. The results show that on average RUN takes the least time to compute: 76 microseconds, which is significantly better than INV (12,859) and REM (15,021). We list the complete results in Table 6.10. *Because the computational time of RUN is significantly smaller than the other two measures and its classification accuracy is only slightly worse than INV, we conclude that RUN is the best presortedness measure for sorting algorithm selection.*

Since RUN is the best feature, in the following two experiments we will use D_{sort4B}

Table 6.10: Computation Time and Classification Accuracy of INV, RUN and REM

	INV	RUN	REM
average time (microseconds)	12,859	76	15,021
accuracy (%)	94.51	90.19	87.68

as the training dataset. This saves us the time of computing INV and REM, while we still have an acceptable classification accuracy of around 90%.

6.6.4 Experiment 3: Determining the Best Model for Sorting Algorithm Selection

In this experiment, we induce the predictive model of sorting algorithm selection from D_{sort4B}. As introduced in previous chapters, there are many representations and learning schemes that can be applied. In the following we will investigate three basic representations: decision tree, the naive Bayes, and Bayesian networks. We will also look at three meta-learning schemes: bagging, boosting, and stacking.

In Bayesian network learning we need to first discretize the input data. In bagging [Bre96], we use C4.5 as the base classifier to bag. The bag size was the same as D_{sort4B}. The number of bagging iterations was set to 10. The boosting method used was Freund & Schapire's *Adaboost M1* [FS96] method. The basis classifier used was C4.5. The maximum number of boost iterations was set to 10. In stacking [Wol92], the three base classifiers were C4.5, the naive Bayes classifier and Bayesian network learning. The meta-classifier used was C4.5.

We run these 6 learning schemes on D_{sort4B} and compare the results to see which one learns the best model; i.e., the one that has highest classification accuracy and needs the least reasoning time when classifying the new instance.

The experimental results of these 6 learning schemes are listed in Table 6.11. We can see that Bayesian network learning has the highest classification accuracy at 90.45%. The second and third best models are bagging and C4.5. We also no-

Table 6.11: Classification Accuracy of 6 Different Learning Schemes on D_{sort4B}

	C4.5	NaiveBayes	BayesNet	Bagging	Boosting	Stacking
accuracy (%)	90.03	74.23	90.61	90.50	88.94	89.69
StdDev (%)	0.26	0.06	0.12	0.2	0.52	0.49

tice that NaiveBayes has the worst performance at only 74.23%; all other inducers' classification accuracies are near 90%.

Confusion matrices of the first three best inducers – BayesNet, Bagging and C4.5 – are shown as follows:

$$BayesNet: \begin{pmatrix} a & b & c & d & e & <-- & classified\ as \\ 748 & 19 & 0 & 0 & 24 & | & a = insertion \\ 11 & 162 & 0 & 0 & 37 & | & b = shell \\ 1 & 0 & 0 & 0 & 19 & | & c = heap \\ 3 & 1 & 0 & 0 & 4 & | & d = merge \\ 43 & 14 & 0 & 0 & 789 & | & e = quick \end{pmatrix}$$

$$Bagging: \begin{pmatrix} a & b & c & d & e & <-- & classified\ as \\ 748 & 17 & 0 & 0 & 26 & | & a = insertion \\ 13 & 157 & 0 & 0 & 40 & | & b = shell \\ 1 & 0 & 0 & 0 & 19 & | & c = heap \\ 3 & 1 & 0 & 0 & 4 & | & d = merge \\ 44 & 10 & 0 & 0 & 792 & | & e = quick \end{pmatrix}$$

$$C4.5: \begin{pmatrix} a & b & c & d & e & <-- & classified\ as \\ 750 & 18 & 0 & 0 & 23 & | & a = insertion \\ 14 & 158 & 0 & 0 & 38 & | & b = shell \\ 1 & 0 & 0 & 0 & 19 & | & c = heap \\ 3 & 1 & 0 & 0 & 4 & | & d = merge \\ 56 & 10 & 0 & 0 & 780 & | & e = quick \end{pmatrix}$$

Besides classification accuracy, the time needed to classify an instance (reasoning time), is also an important criterion to evaluate the sorting algorithm selection model. Among the above three best models, the decision tree is the most efficient on classifying a new instance because it simply applies a set of "if-then" rules. Bagging needs to combine multiple models and vote to produce the final classification, so it takes a

Table 6.12: Reasoning Time (microseconds) and Classification Accuracy (%) of the Best Three Models

	C4.5	BayesNet	Bagging
accuracy	90.03	90.61	90.50
time	43.35	14,845.35	8,690

longer time. Bayesian networks are much slower compared to decision trees because the reasoning process involves first discretizing the data and then performing inference (MAP) by propagation. We list the average reasoning times and classification accuracies of the three models on D_{sort4B} in Table 6.12 to compare their performance.

We can see that while their classification accuracies are almost the same, decision tree has a much better reasoning time. Therefore we can draw the conclusion that *among the models we have experimented with decision tree is the best for sorting algorithm selection.*

6.6.5 Experiment 4: Evaluating the Sorting Algorithm Selection System

From our previous experimental results we know that RUN is the best feature and C4.5 decision tree learning is the best learning scheme for sorting algorithm selection. The learned decision tree is shown in Figure 6.15. It is the core of the sorting algorithm selection system. When a new instance comes, the "meta-reasoner" first examines the instance and calculates its features, including size and the number of runs. It then uses the decision tree model to select the best algorithm according to the features. Finally, it calls the selected algorithm and sorts the instance. The total time of this process consists of three parts:

$$T_{total} = T_{examining} + T_{reasoning} + T_{sorting} \tag{6.9}$$

Obviously the algorithm selection system is worth having only if the time spent on examining and reasoning can be compensated from the gain of selecting the best

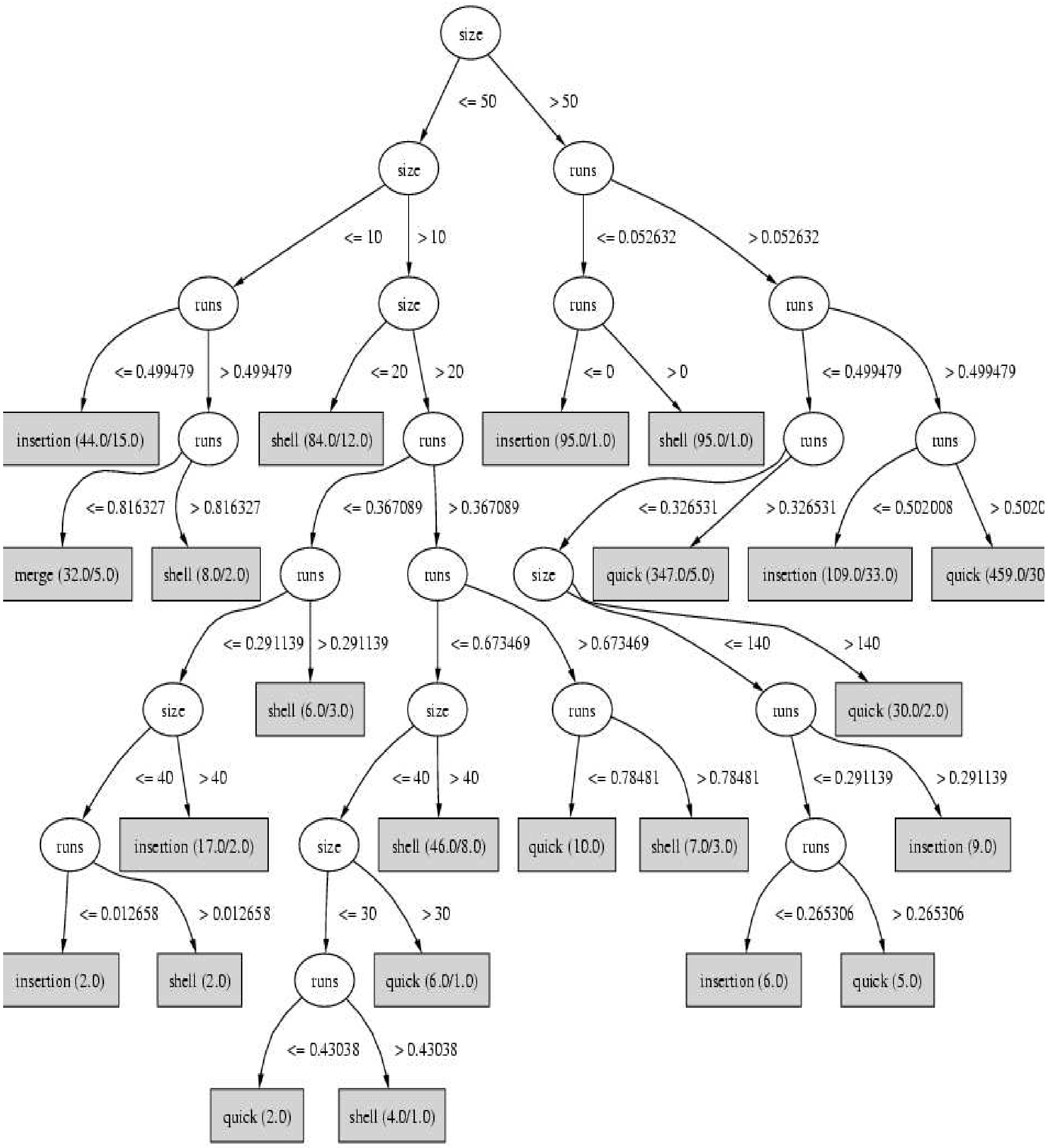


Figure 6.15: The Learned Decision Tree from D_{sort4B}

Table 6.13: Statistics of Test Dataset D_{sort5}

	Size	INV	RUN	REM
Minimum	1,000	0	0	0
Maximum	1,000	0.52	0.52	0.95
Mean	1,000	0.26	0.26	0.05
StdDev	0	0.11	0.10	0.21

algorithm. In this section, we conduct experiment 4 to show that for some datasets the algorithm selection system can actually provide the best overall sorting performance. Our test data set D_{sort5} contains 950 nearly-sorted instances and 50 randomly disordered permutations of size 1,000. The statistics of D_{sort5} are shown in Table 6.13.

The classification accuracy of our learned model on D_{sort5} is 98.4%. There are 16 incorrectly classified instances out of a total of 1,000. The confusion matrix is as follows.

$$\left(\begin{array}{llllllll}
a & b & c & d & e & <-- & & classified\ \ as \\
942 & 0 & 0 & 0 & 0 & | & & a = insertion \\
2 & 0 & 0 & 0 & 0 & | & & b = shell \\
1 & 0 & 0 & 0 & 1 & | & & c = heap \\
0 & 0 & 0 & 0 & 0 & | & & d = merge \\
12 & 0 & 0 & 0 & 42 & | & & e = quick
\end{array}\right)$$

The total time of computing the RUN values on D_{sort5} is $T_{examining} = 91,475$ microseconds. The time of reasoning (classification) is $T_{reasoning} = 21,723$ microseconds. The actual time spent on sorting is $T_{sorting} = 211,068$ microseconds. Thus, the total time consumed by the system during sorting D_{sort5} is $T_{total} = 324,266$ microseconds. Approximately, the algorithm selection system spends 28% of the total time on examining, 7% of the total time on reasoning, and 65% on sorting. Figure 6.16 compares T_{total} of the algorithm selection system (the 6th bar) with the time of each sort algorithm (first five bars) applying solely on D_{sort5}. We can see that the algorithm selection system outperforms all sorting algorithms on this dataset. The last bar in Figure 6.16 is the optimal sorting time which is collected by actually running all five sorting algorithms at the same time and using the first one that returns a sorted list. The algorithm selection system would perform like that if it could save the time for examining and reasoning and its classification error be 100%.

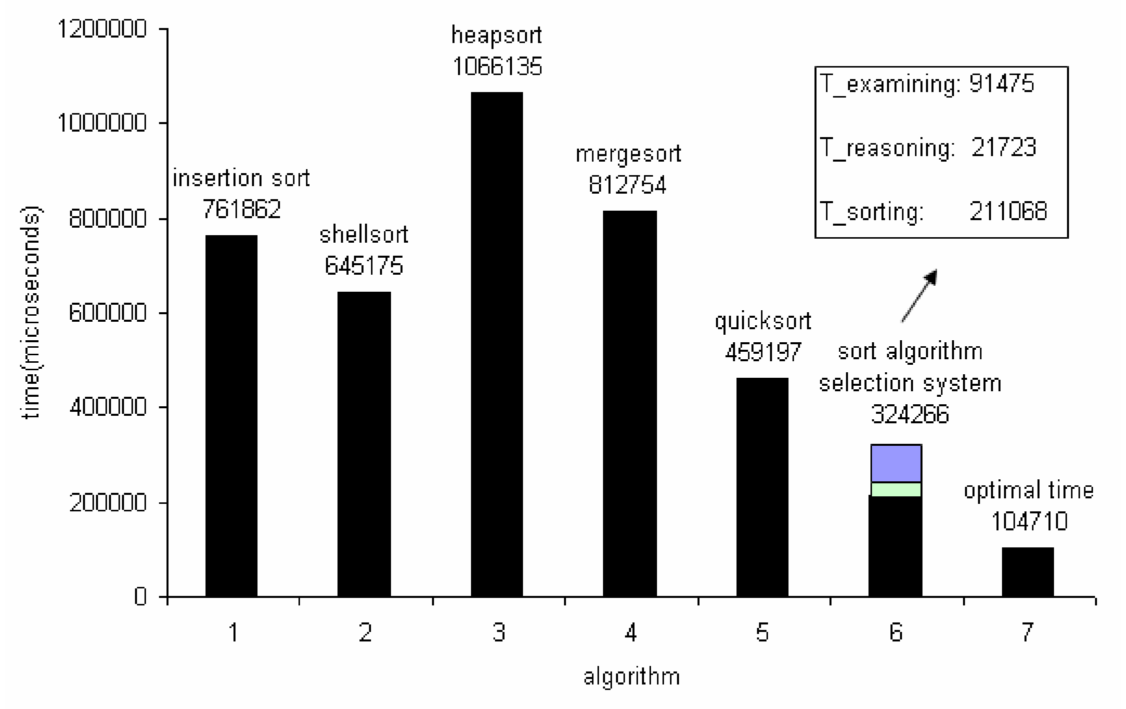


Figure 6.16: Time Spent by Each Algorithm on D_{sort5}

To see the working range of the algorithm selection system on such dataset, examine Figure 6.17 [4]. Figure 6.17 describes the change of computational time consumed by insertion sort, quick sort, and the algorithm selection system as we incrementally add 100 more randomly ordered instances to D_{sort5}. From Figure 6.17, we can see that, for the first 950 nearly sorted instances, insertion sort uses the least amount of time and quick sort uses the most amount of time. The algorithm selection system lies in between because, comparing to insertion sort, the algorithm selection system takes extra examining and reasoning time. As we start adding randomly ordered instances,

[4]In figure 6.17, T_{total} is the total time spent by the algorithm selection system, $T_{total} = T_{examining} + T_{reasoning} + T_{sorting}$.

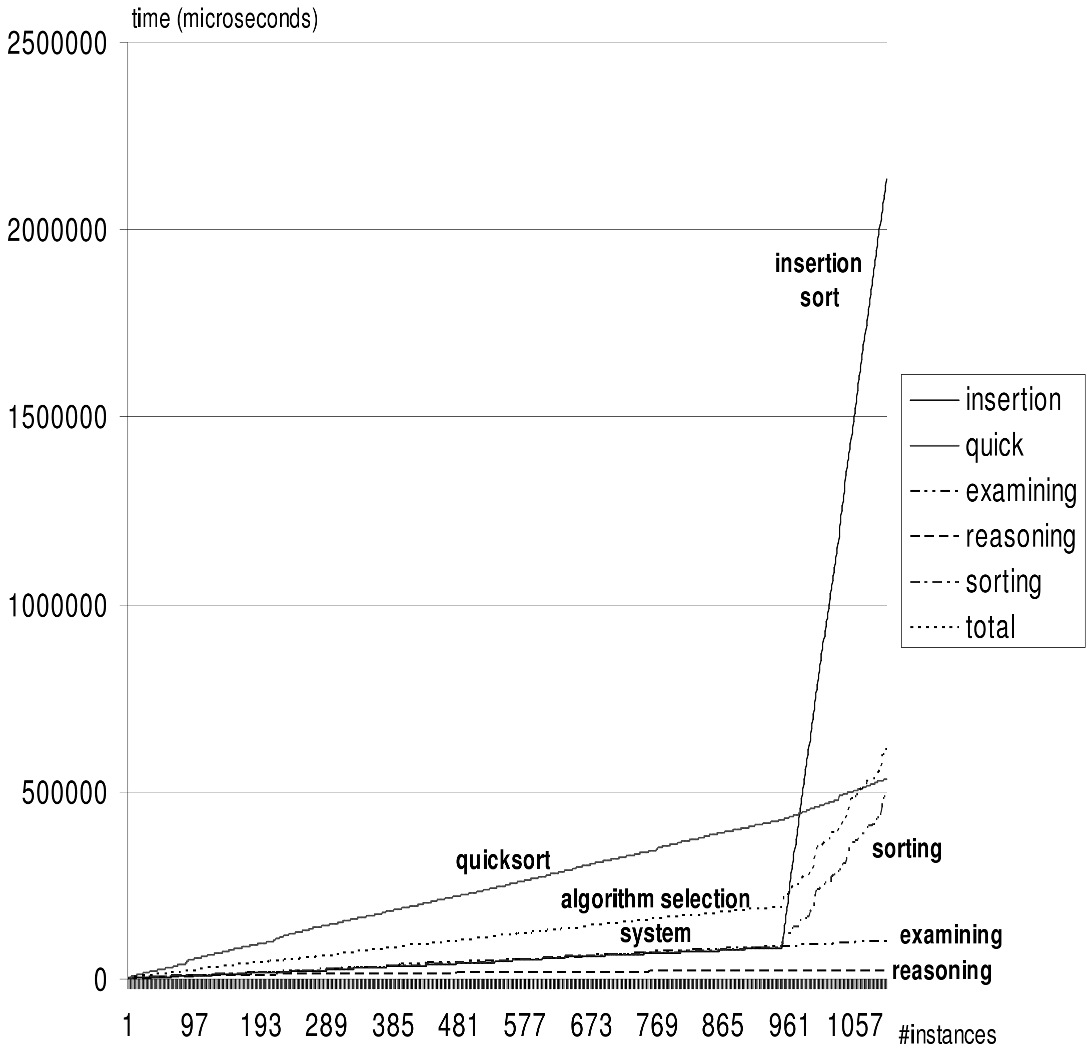


Figure 6.17: Working Range of the Algorithm Selection System (D_{sort5})

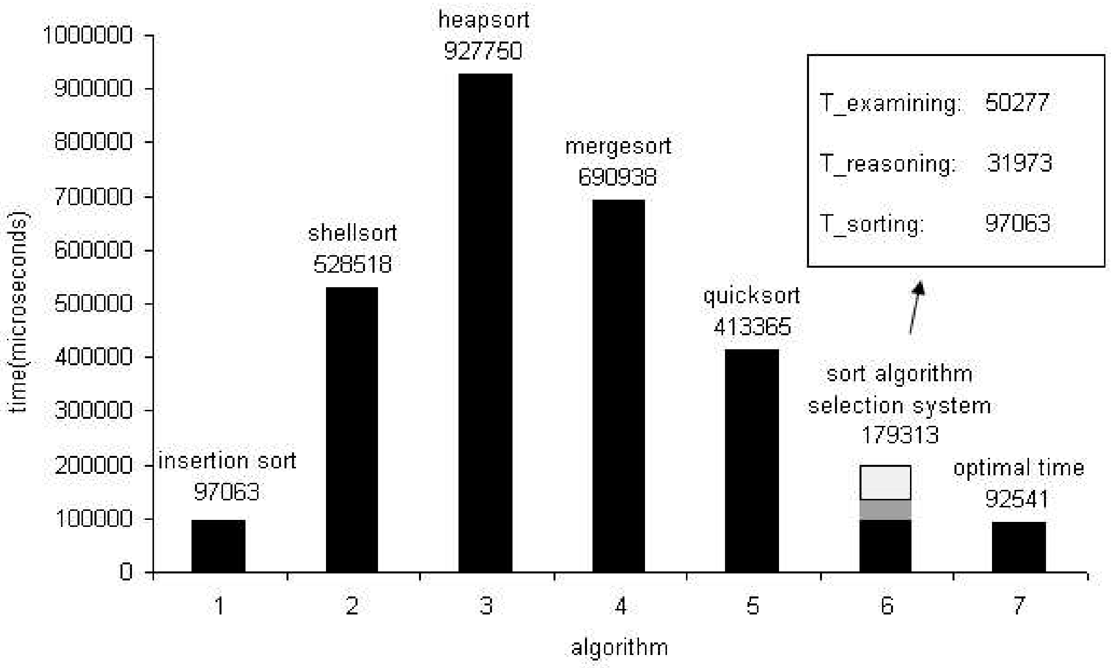


Figure 6.18: Time Spent by Each Sort Algorithm on Nearly Sorted Permutations

time used by insertion sort jumps up sharply and becomes the worst algorithm soon after around the 965th instance is added. In the mean time, quick sort's performance is not affected as much. If we keep adding more randomly ordered instances, the algorithm selection system will lose its advantage over quick sort little by little because, compared to quick sort, it always takes some extra examining and reasoning time. Finally, after around the 1070th instance ia added, quick sort outperforms the algorithm selection system and becomes the overall best algorithm. The "working region" of the algorithm selection system ranges approximately between 965 and 1070; i.e., the algorithm selection system performs the best if in the data set the ratio of nearly ordered instances to randomly ordered instances is larger than around 1.58%(15/950) and less than around 12.63%(120/950).

We have also evaluated the sorting algorithm selection system's overall perfor-

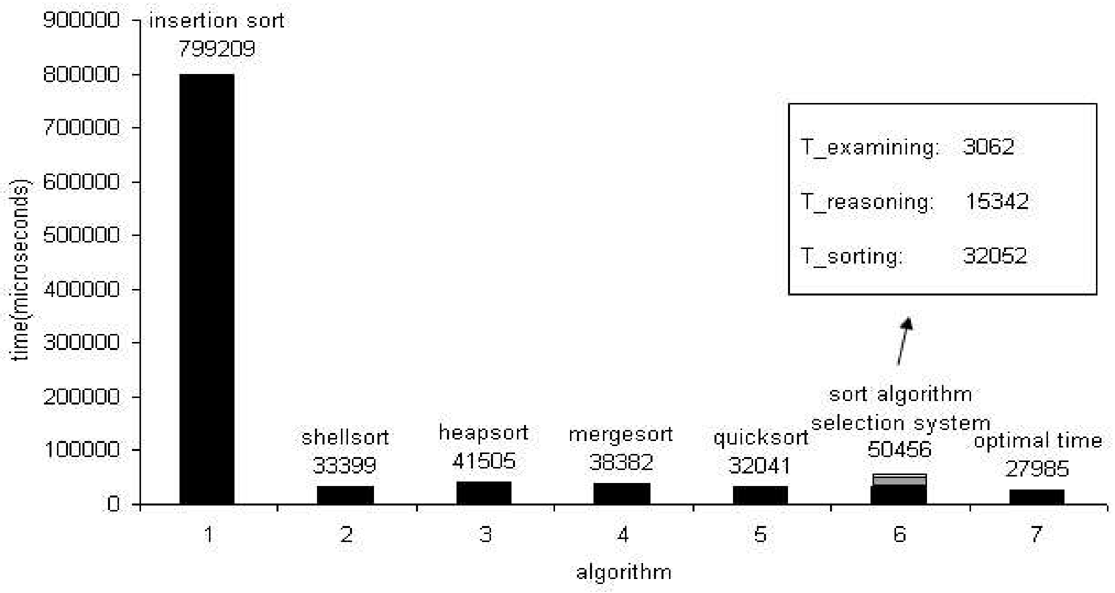


Figure 6.19: Time Spent by Each Sort Algorithm on Reversely Ordered Permutations

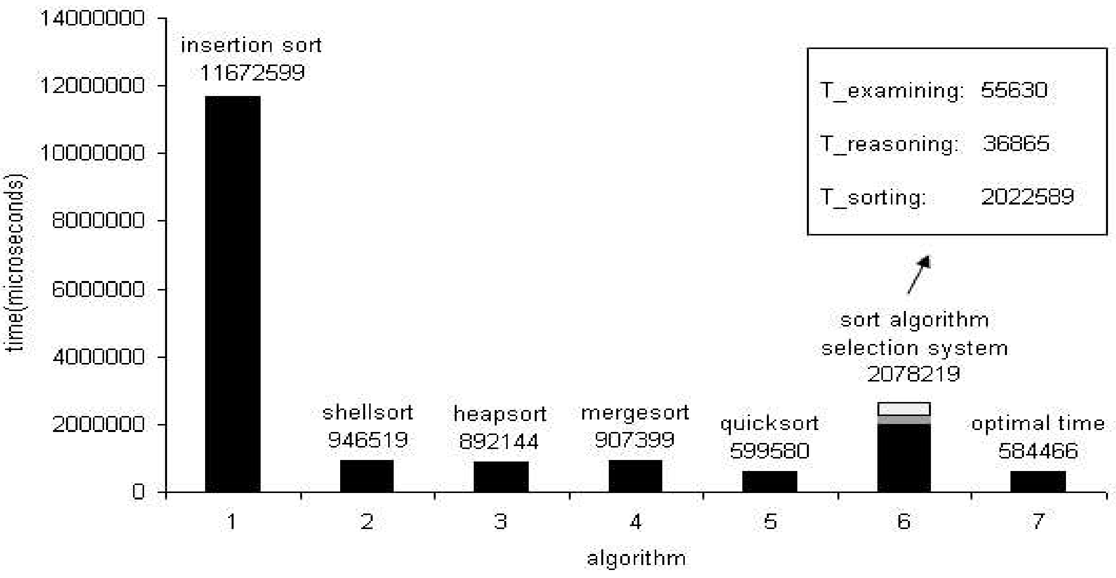


Figure 6.20: Time Spent by Each Sort Algorithm on Randomly Ordered Permutations

mance on 3 other test datasets, all containing permutations of the same type. $D_{SortTest1}$ contains 1,000 nearly-sorted instances. $D_{SortTest2}$ contains 100 permutations in reverse order. $D_{SortTest3}$ contains totally 1,000 random permutations. The results are shown in Figure 6.18, Figure 6.19, and Figure 6.20. The correspondent classification accuracies are 99.4%, 77% and 85.3%. For nearly-sorted permutations, insertion sort is the best and the selection system ranks the second. The algorithm selection system is outperformed by insertion sort only because it has to spend some extra time on examining and reasoning. Because the classification accuracy is very high (99.3%), the actual sorting time (the black portion of bar 6) is almost as good as the optimal sorting time (bar 7). For permutations in reverse order, shellsort is the best and slightly outperforms quicksort. The algorithm selection system is much better than the worst algorithm insertion sort and slightly worse than all others. This is mainly due to its low classification accuracy. For totally random permutations, quicksort is the best. The algorithm selection system performs worse than all other algorithms except insertion sort. This is because although its classification accuracy is 85.3%, it misclassifies those 14.7% instances to insertion sort which is quadratic on these random permutations. The underlying reason is that the tradeoff we made on selecting RUN as the best predictive presortedness measure: although it takes least time to compute, it can not distinct local disordered permutations from totally random ones very well.

In all three tests, there is no case where the algorithm selection system is the worst. The result suggests that knowing the underlying distribution of the input instances could help the meta-level reasoner to have more flexible decision making and thus gain the best performance. For example, if the meta-reasoner can quickly sense that the distribution of the input instances have changed from nearly-sorted instances to totally-random instances, it can then correspondingly modify its algorithm selection model and select the best algorithm for these most recent inputs without even examining and reasoning on all input instances. However, this will require another reasoner that acts at the meta-meta-level.

6.7 Summary

We have applied the learning-based approach to sorting algorithm selection. Our experimental results show that machine learning techniques can be used to build an algorithm selection system to gain more efficient computation for polynomial problems such as sorting. In algorithm selection for polynomial computational problems, time is the most important factor. Therefore, besides classification accuracy, the computational time of the instance feature and the reasoning time of the model are crucial criteria in determining which feature and which model are the best for algorithm selection. In sorting, we have experimentally found that the number of runs of the input permutation is the best feature and decision tree is the best model. The algorithm selection system built using RUN and decision tree can provide the best overall performance with a highly competitive classification accuracy (around 90%). The system spends some time on meta-level examining and reasoning to select the best sort algorithm before it starts sorting. By doing this it is able to achieve a better overall performance on some datasets containing both nearly-sorted and highly-random permutations.

Chapter 7

Algorithm Selection for the Most Probable Explanation Problem

In this chapter, we apply the proposed machine learning-based approach to algorithm selection for the Most Probable Explanation (MPE) problem. A MPE problem instance consists of three components: network topology, CPTs, and the observed evidence. Correspondingly, the instance features also include three categories: topological type and connectedness of the network; size and skewness of CPTs; and proportion and distribution of evidence nodes. The MPE algorithms under consideration include: exact clique-tree propagation algorithm, Gibbs sampling, forward sampling; random restart hill-climbing, Tabu search, and Ant Colony Optimization.

7.1 The MPE Problem

Finding the most probable explanation is an important probabilistic inference problem in Bayesian networks. Recall that a Bayesian network B is a pair (G, P) where G is a directed acyclic graph of n nodes and P is a set of prior and conditional probability tables one for each node in G. An evidence E is a set of instantiated nodes. An explanation for the evidence E is a complete assignment of all node values $\{X_1 = x_1, \ldots, X_n = x_n\}$ that is consistent with evidence E. The most probable explanation is an explanation such that no other explanation has higher probability. Given these notations and definitions, the MPE problem is defined as follows.

THE MPE PROBLEM

INSTANCE: A triple (G, P, E) where (G, P) defines a Bayesian network of n nodes and E is an evidence set $E = \{E_1 = e_1, \ldots, E_k = e_k\}$.

QUESTION: Computing a MPE of E in (G, P).

It has been shown that exact MPE computation is NP-hard [Shi94]. Furthermore, approximating MPE to within a constant ratio-bound is also NP-hard [AH98].

7.2 Algorithms for Finding the MPE

The MPE problem belongs to an important type of probabilistic inference problem using Bayesian networks called belief revision. The other type of Bayesian network inference problems is belief updating, which computes the posterior belief over the query nodes given the evidence E. Belief updating is also known as probabilistic inference. In practice, algorithms for belief updating can often be used for belief revision, possibly with only slight modifications, and vice versa.

Bayesian network inference algorithms [Guo02] can be roughly classified as exact or approximate. Exact algorithms include three main categories: conditioning, clustering, and elimination. Approximation algorithms include model-simplification methods, stochastic sampling algorithms, search-based algorithms, and loopy propagation. In this chapter, we will study the problem of selecting the best out of six different MPE algorithms: one exact algorithm, two sampling algorithms, two search-based algorithms, and one hybrid algorithm combining both sampling and search. The classification of these algorithms is shown in Figure 7.1.

7.2.1 Exact Algorithm: Clique-tree Propagation

Clique-tree propagation algorithm, also called junction-tree or join-tree propagation, is the most popular exact algorithm for Bayesian network inference in practice. The basic idea is converting the network topology to a probabilistic equivalent polytree by clustering nodes into meganodes. It first transforms the network into a tree of cliques,

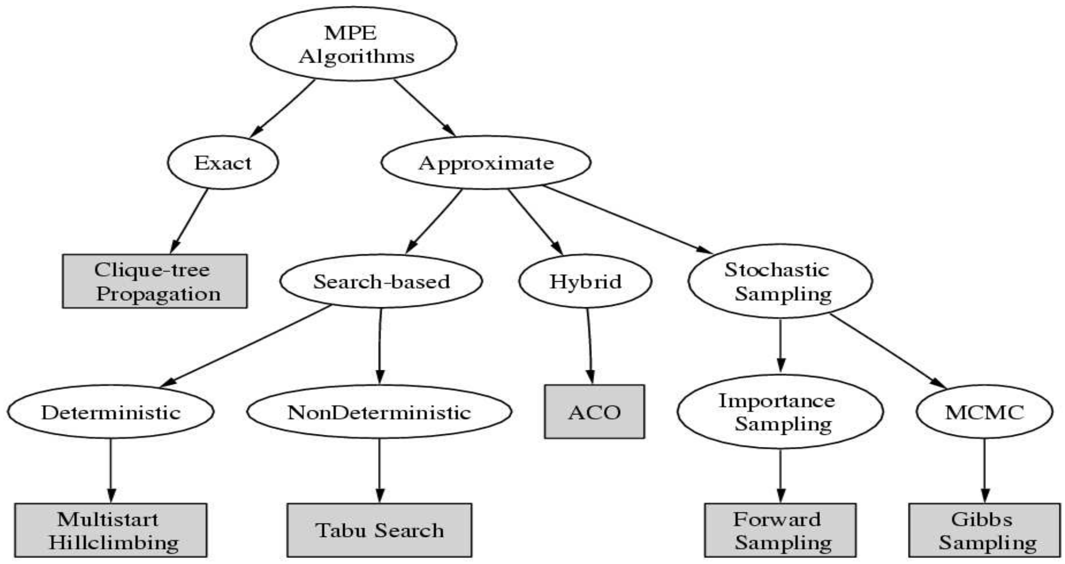


Figure 7.1: Candidate MPE Algorithms

then performs belief propagation over the tree using Pearl's linear time polytree propagation algorithm [Pea88]. The first phase consists of moralization, triangulation, and clique-tree identification. The second phase is a two-way message propagation. When applied to belief updating, the second phase performs a summation propagation. For finding the MPE, it performs a maximization propagation. For details of the algorithm, we refer readers to [HD96, Nea90, LS88]. The clique-tree propagation's time and space complexities are both exponential in the size of the largest clique of the transformed undirected graph. In practice, it works well for sparse networks even if the size of network is very large (up to one thousand nodes, for example). However, as the network becomes dense, the algorithm often runs out of memory before you can feel the exponential growth of time. In our study, we will use Hugin's implementation of clique-tree propagation, which is currently the standard and the best implementation available.

7.2.2 Stochastic Sampling Algorithms

Stochastic sampling algorithms [CD00] can be divided into *importance sampling algorithms* and *Markov Chain Monte Carlo (MCMC) methods*. They differ from each other in the way samples are drawn. In importance sampling algorithms, samples are drawn independently from an importance function. The importance function can be different from the CPTs, and they may be updated during the sampling process as well. In MCMC methods, the samples are drawn depending on the previous samples. The sampling distribution of a variable is computed from its previous sample given the states of its Markov blanket nodes. Importance sampling algorithms include *Logic Sampling* [Hen88], *Forward Sampling (Likelihood Weighting)* [FC89, SP89], *Backward Sampling* [FF94], *Self-Importance Sampling and Heuristic Importance Sampling* [SP89], *Adaptive Importance Sampling* [CD00], etc. MCMC methods are divided into *Gibbs sampling* [Pea87, Pea88], *Metropolis sampling*, and *Hybrid Monte Carlo sampling* [GG84, Mac98, GRS96]. Both sampling algorithms are easy to implement and can be applied on a large range of network sizes. But, when the network is large and the evidence is unlikely, the most probable explanation will also be very unlikely. The probability of the algorithm being hit by any sampling schemes is low. This is the main drawback of sampling algorithms.

Gibbs Sampling

Gibbs Sampling [Pea87, Pea88], also called Stochastic Simulation, is a Markov Chain Meto Colo (MCMC) method. It starts from a random initial assignment, then generates a sample from the previous sample by randomly "flipping" the state of an non-evidence node X_i. X_i is randomly selected and the sampling distribution $P(X_i)$ is computed from its previous sample given the states of its Markov blanket nodes.

$$P(X_i|X/X_i) = \alpha P(X_i|\pi(X_i) \prod_j P(Y_j|\pi(Y_j))) \tag{7.1}$$

where α is a normalization constant and Y_j is jth child of X_i. After generating a given number of samples, the best sample so far is returned as the approximate

```
Input:  A BN $(G, P)$ and an evidence set $E$.
Output: A complete assignment $u = (u_1, \ldots, u_n)$.

Step 1: Randomly generate an initial sample
    that agrees with the evidence;
    Set the $best\_so\_far$ to the initial sample.

Step 2: Randomly select an non-evidence node $X_i$,
    compute $P(X_i)$ from the values of its Markov
    blanket nodes in the previous sample, sample
    from $P(X_i)$ and set $X_i$ to the sampled value.

Step 3: Evaluate the new sample and update
    $best\_so\_far$, decrease $n\_samplesNeeded$.

Step 4: If $n\_samplesNeeded > 0$, goto Step 2;
    else return $best\_so\_far$.
```

Figure 7.2: Gibbs Sampling for Finding the MPE

MPE. Research has shown although the convergence of Gibbs sampling is theoretically guaranteed, the convergence rate is extremely slow.

Forward Sampling

Forward sampling [CD00] is an importance sampling algorithm in which the importance functions are the prior CPTs and never change. It samples each non-evidence variable in turn according to topological order. The forward sampling algorithm for the MPE problem is listed in Figure 7.3.

7.2.3 Search-based Algorithms

Search algorithms have been studied extensively in solving hard combinatorial optimization problems. Since the MPE is a NP-Hard optimization problem, researchers have applied various optimization algorithms to solve it- the best first search [SC99],

```
Input:   A BN $(G, P)$ and an evidence set $E$.
Output:  A complete assignment $u = (u_1, \ldots, u_n)$.

Step 1: For each non-evidence node $X_i$,
    draw random sample from $P(X_i|\pi(X_i))$.

Step 2: Evaluate the new sample and update
    $best\_so\_far$, decrease $n\_samplesNeeded$.

Step 3: If $n\_samplesNeeded > 0$, goto Step 1;
    else return $best\_so\_far$.
```

Figure 7.3: Forward Sampling for Finding the MPE

linear programming [San91], stochastic local search [KD99], genetic algorithms [Men99], etc. [Par02b] has also tried to convert the MPE problem to other NP-complete problem such as MAX-SAT, and then use SAT-solver to solve the MPE problem indirectly. Search algorithms often use some meta-heuristic to guide the search in order to avoid getting stuck into a local optima. The most popular meta-heuristics include various Hillclimbing algorithms [Hro01], Simulated Annealing [KGV83], Tabu Search [GLW93], Genetic Algorithms [Gol89], etc.

Random-restart Hill Climbing

Hill climbing [Hro01] is a greedy local search method. It goes to its best neighbor whenever possible. It is east to get stuck to a local optimal. Random-restart Hill Climbing randomly restarts another Hill climbing when a local optimal is reached. It stops after the stop criteria is satisfied and returns the best solution so far. The success of this algorithm depends very much on the landscape of the search space. If there are few local optimums, it will find a good solution very quickly.

```
Input:   A BN $(G, P)$ and an evidence set $E$.
Output:  A complete assignment $u = (u_1, \ldots, u_n)$.

Step 1: Randomly generate a start point
        that agrees with the evidence;

Step 2: While the best neighbor $B^*$ is better
        than the current point, update $best\_so\_far$,
        decrease $n\_samplesNeeded$, move to $B^*$.

Step 3: If $n\_samplesNeeded > 0$, goto Step 1;
        else return $best\_so\_far$.
```

Figure 7.4: Random-restart Hill Climbing for Finding the MPE

Tabu Search

Tabu search [GLW93, Hro01] is a heuristic based on local search. Local search algorithms like Hill climbing are memory-less algorithms, which means the next step depends only on the current feasible solution and not on the search history. The idea of Tabu search is to store some information about a set of the last feasible solutions generated, called a *Tabu list*, and use this information when generating

the next solution. Notice that the searcher moves to B^* even it is worse than the current point. This is the main difference from the local search algorithm. Usually the update of the Tabu list is done by inserting the recently-visited point into the Tabu list. Tabu search uses the Tabu list to forbid revisiting any points that have been visited in the last k steps. This can avoid repetitions and short cycles. There are some other advanced advanced features that can be added to improve Tabu search. A simple Tabu search algorithm for the MPE problem is described in Figure 7.5.

```
Input:  A BN $(G, P)$ and an evidence set $E$.
Output: A complete assignment $u = (u_1, \ldots, u_n)$.

Step 1: Randomly generate a start point
    that agrees with the evidence and
    Add it into the Tabu list.

Step 2: Find the best neighbor $B^*$ that is not
    in the Tabu list; move to $B^*$
    update $best\_so\_far$ if necessary;
    decrease $n\_samplesNeeded$; update Tabu list .

Step 3: If $n\_samplesNeeded > 0$, goto Step 2;
    else return $best\_so\_far$.
```

Figure 7.5: Tabu Search for Finding the MPE

7.2.4 Hybrid Algorithm: Ant Colony Optimization for the MPE Problem

Ant Colony Optimization (ACO) [DG97] studies artificial systems that take inspiration from the behavior of real ant colonies and are used to solve hard combinatorial optimization problems. The ACO meta-heuristic was first introduced by Dorigo in his Ph.D. thesis [Dor92], and was recently defined by Dorigo, Di Caro and Gambardella [DCG99].

Ant algorithm optimization was inspired by the observation of real ant colonies' foraging behavior; in particular, how ants can find the shortest paths between food sources and their nest. Ants deposit on the ground a chemical substance called *pheromone* while walking from the nest to the food sources and vice versa. This forms *pheromone trails*. Ants can smell pheromone and choose their path in favor of the trails with strong pheromone concentrations. Other ants can also use the pheromone trails to find the way to the food (or to the nest). Therefore pheromone provides an indirect way of communications among the ant colony. It has been shown experimentally that this foraging behavior can give rise to the emergence of shortest paths if employed by a colony of ants. A colony of ants is able to choose the shortest path from the nest to the food source and back by exploiting the pheromone trails left by the individual ants.

Based on this emergent behavior of ant colony, researchers have developed artificial ant systems to solve hard discrete optimization problems. In an ant system, artificial ants are created to explore the search space simulating real ants searching their environment. The objective values correspond to the quality of the food and the length of the path to the food. An adaptive memory corresponds to the pheromone trails. Also, the artificial ants can make use of a local heuristic function to help make the decision among a set of feasible solutions. In addition, usually a pheromone evaporation mechanism is included to allow the ant colony to slowly forget its past history so that it can direct its search towards new directions that have not been explored in the past. ACO can be seen as a hybrid optimization algorithm that combines the advantages of both sampling and search. Each ant is a sample and its search decision making is affected by the pheromone dropped by previous ants. The search process can be seen as a cooperative learning process.

The ant system was first used on the Travelling Salesman Problem (TSP) [DG97]. From then on, it has been applied to the Job Shop Scheduling Problem [CDMT93], to the Graph Coloring Problem [DH97], etc.

The first task of applying ACO to any optimization problem is to restate the problem as a shortest or longest path problem of ant systems. In the following, we convert the MPE to *a longest path ant search problem with conditional branches and an order constraint.*

In the ant system, artificial ants build solutions (explanations) of the MPE problem by moving on the problem network from one node to another. Ants must visit all nodes in the *topological order* defined by the Bayesian network, i.e., all parent nodes must be visited before visiting the child node. For evidence nodes, ants are only allowed to take the branches that agree with the evidence. Each number of the CPT at one node represents a *conditional branch.* The memory of each ant contains the already visited nodes and the selected branches, as well as three tables for each node: *a Pheromone Table (PT)*, *a Heuristic Function Table (HFT)*, and *an Ant Decision Table (ADT).* All three tables have the same structure as the CPTs. We use the CPTs as the HFTs and they keep unchanged.

The ADT $A_i = [a_{ijk}]$ of node i is obtained by the composition of the local pheromone trail values τ_{ijk} with the local heuristic values η_{ijk} as follows:

$$a_{ijk} = \frac{[\tau_{ijk}]^\alpha [\eta_{ijk}]^\beta}{\sum_j [\tau_{ijk}]^\alpha [\eta_{ijk}]^\beta} \tag{7.2}$$

where j is the jth row and k the kth column of the corresponding table at the ith node. α and β are two parameters that control the relative weight of pheromone trails and heuristic values.

The probability with which an ant chooses to take which conditional branch while building its tour is:

$$p_{ij} = \frac{a_{ij\pi_i}}{\sum_j a_{ij\pi_i}} \tag{7.3}$$

where π_i is the column index of the ADT and its value is conditioned on the values of parent nodes of ith node.

After ants have built their tour (an explanation), each ant deposits pheromone $\Delta\tau_{ijk}$ on the corresponding pheromone trails (the conditioned branches of each node). The pheromone value being dropped represents the quality of this solution. Since we want to find the Most Probable Explanation, we use the likelihood of this tour as the pheromone value. Suppose the generated tour is $\{x_1, \ldots, x_n\}$, the pheromone value is as follows:

$$\Delta\tau_{ijk} = \begin{cases} P(x_1, \ldots, x_n) & if j = x_i, k = \pi(x_i) \\ 0 & \text{otherwise} \end{cases} \tag{7.4}$$

in which $P(x_1, \ldots, x_n)$ is computed by the chain rule:

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i|\pi(x_i)) \tag{7.5}$$

Updating the pheromone tables is done by adding a pheromone value to the corresponding cells of the old pheromone tables. Each ant drops pheromone to one cell of each pheromone table of each node, i.e., the *jth* row, *kth* volume of the PT at *ith* node. After dropping the pheromone, an ant dies. The pheromone evaporation procedure happens just before ants start to deposit pheromone. The main role of pheromone evaporation is to avoid stagnation when all ants end up selecting the same tour. The pheromone tables are updated by the addition of new pheromone by ants and pheromone evaporation as follows:

$$\tau_{ijk} = (1 - \rho)\tau_{ijk} + \Delta\tau_{ijk} \tag{7.6}$$

where $\tau_{ijk} = \sum_{l=1}^{m} \Delta\tau_{ijk}$, m is the number of ants at each iteration, and $\rho \in (0, 1]$ is the pheromone trail decay coefficient.

In practice, an optional *daemon activity* can be added to collect useful global information to drop additional pheromone. In doing this, it will bias the ant search process from a non-local perspective. For example, a *daemon* can be can allowed to observe all ants' behavior, give extra "rewards" to the best ants and "punish" the worst ants by adding and removing pheromone.

Figure 7.6 lists the basic ANT-MPE algorithm. After initialization, the algorithm generates a batch of ants for several iterations. At each iteration, the ants sample the ant decision tables to produce a trail, evaluate the trails by CPTs, and save best sample of this iteration. Then pheromone is dropped and the pheromone tables are updated. The pheromone evaporation and an optional daemon action are triggered right after. At the end of each iteration, the ADTs are updated and normalized. This procedure stops when the number of iterations runs out. The best solution so far is returned as the approximate MPE. All parameter values are set at the initialization step. The initial amount of pheromone is set to a same small positive constant on all pheromone tables. The number of ants per iteration is set to 100.

7.3 Characteristics of the MPE Problem Instances

A MPE instance consists of three parts: the network, the CPTs, and the evidence. The instance characteristics we will study are also centered around these aspects.

```
Input:   A BN $(G, P)$ and an evidence set $E$.
Output:  A complete assignment $u = (u_1, \ldots, u_n)$.

Initialization:
    Initialize $\alpha$, $\beta$, $\rho$, $n\_interations$, $n\_ants$;
    Set PTs to 0, ADTs to uniform, HFTs to
    CPTs; Set $best\_trail\_so\_far$ to null.

Step 1: Generate $n\_ants$ ant trails by sampling
     the ADTs, compute the pheromone
     values, decrease $n\_interations$, update
     $best\_trail\_so\_far$

Step 2: Update PTs by dropping pheromone,
     pheromone evaporation, and daemon actions.

Step 3: Compute new ADTs from CTs ,
     and HFTs, normalize ADTs.

Step 4: If $n\_interations > 0$, goto Step 1;
     else return $best\_trail\_so\_far$.
```

Figure 7.6: The ANT-MPE Algorithm

7.3.1 Network Characteristics

Network characteristics include network topological type and network connectedness.

Network Topological Type

By definition, all Bayesian networks are DAGs. Within DAGs, we distinguish between *single connected graphs* and *multiply connected graphs*. In singly connected directed graphs, there is at most one directed path between any two nodes. Multiply connected graphs can have more than one path between two nodes, thus there are loops in the graphs. Within singly connected graphs, we distinguish *trees*, *polytrees*, and *two-level networks*. In trees (directed), each node can have only one parent. Polytrees allow one node to have more than one parents, but there are at most one path between any two nodes in the *underlying undirected graphs*, i.e., the underlying undirected graph is a tree. Two-level networks are also called noisy-OR models. They describe so-called noisy-OR relation, which is a generalization of the logical OR. In a two-level noisy-OR model, nodes at the root level represent all the possible causes and nodes at the leaf level represent effects. In general, noisy-OR models provide a simpler model and make the learning and inference easier. One famous example of noisy-OR network is the QMR-DT network [SMH$^+$91]. All trees and polytrees are singly connected, but not vice versa. Two-level networks are singly connected because all paths have length 1. But they are not polytrees because their underlying undirected graphs can have loops just as multiply connected graphs do.

A complex topology is more expressive than a simple one. But it also increases the computational complexity. The inferences for both trees and polytrees are polynomial because the underlying undirected graphs have tree structure. For graphs whose underlying undirected graphs contain cycles, inference becomes intractable, i.e., the general MPE problems for two-level networks and multiply connected networks are both NP-hard [Shi94, SD02].

Network Connectedness

Let the number of nodes of a network be *n_nodes* and the number of arcs be *n_arcs*. Network connectedness *conn* can be calculated as simply $conn = \frac{n\_arcs}{n\_nodes}$. The simplest topology, a polytree, has $n - 1$ arcs [1]. The most complex topology, a fully connected DAG, has $\frac{n(n-1)}{2}$. Thus, *conn* lies between $[\frac{n-1}{n}, \frac{n-1}{2}]$. Usually, a graph is said to be dense if $conn > 2$. In practice, the exact clique-tree propagation algorithm runs fast on sparse networks. But if the network becomes dense, the induced-width (maximum clique size) of its underlying undirected graph can be large. Because both the time and space complexity of the clique-tree propagation algorithm are exponential in the induced-width of the underlying undirected graph, it becomes intractable quickly as

[1] We assume the underlying undirected graph is a connected graph.

the network becomes dense. Often times, we get an out-of-memory error before we can feel the exponential growth of the running time.

7.3.2 CPT Characteristics

CPT characteristics include CPT size and CPT skewness.

CPT Size

A node's CPT size is the number of cells in its CPT, which is the product of the state space of its parents times its own state space. Since we fix the node state space to binary, we can just use the number of parents of a node to measure the CPT size. In our experiment, we consider the maximum number of parents of a node: *max_parents*. The performance of the exact algorithm is influenced by the *conn* and the *max_parents* because these factors affect the size of the largest clique of the underlying undirected graph.

CPT Skewness

The skewness of the CPTs is computed as follows [JN96]: For a vector (a column of the CPT table), $v = (v_1, v_2, \ldots, v_m)$, of conditional probabilities,

$$skew(v) = \frac{\sum_{i=1}^{m} |\frac{1}{m} - v_i|}{1 - \frac{1}{m} + \sum_{i=2}^{m} \frac{1}{m}} \tag{7.7}$$

The skewness for the CPT of a node is the average of the skewness of all columns whereas the skewness of the network is the average of the skewness of all nodes. The skewness has an influence on the performance of the sampling and search-based algorithms.

7.3.3 Evidence Characteristics

Evidence characteristics includes the proportion and the distribution of evidence nodes. Let the number of evidence nodes be *n_evid*. The evidence proportion is simply $\frac{n\_evid}{n\_nodes}$. Usually, more evidence nodes implies more unlikely evidence. Hence, the MPE will also be quite unlikely and the probability that it is hit with any sampling scheme is not very high.

The distribution of evidence nodes will also affect the hardness of the MPE problem. If most evidence nodes are "cause" nodes, the problem is called *predictive reasoning.* If most evidence nodes are "effect" nodes, the problem is called *diagnostic reasoning.* It has been proven that in singly connected networks, predictive reasoning is easier than diagnostic reasoning [SD02]. More specifically, strictly predictive belief updating and belief revision in singly connected networks can be performed in time linear in the size of the network, while diagnostic belief updating and belief revision are NP-hard. "Strictly predictive" is defined as follows [SD02]:

Definition 27 *Given a Bayesian network (G, P), with evidence over a set of nodes E, an inference problem is called strictly predictive if the evidence nodes have no non-evidence parents in G.*

Similarly, we define "Strictly diagnostic" as follows:

Definition 28 *Given a Bayesian network (G, P), with evidence over a set of nodes E, an inference problem is called strictly diagnostic if the evidence nodes have no non-evidence children in G.*

In our experiments, we will consider three types of evidence distributions: strictly predictive, strictly diagnostic, and randomly distributed evidence.

7.4 The Random Generation of MPE Instances

The random generation of MPE instances with controlled parameter values is also based on the Markov-chain method we introduced before [IC02]. We construct and simulate a Markov chain to "walk" randomly in the space of all possible networks that satisfy our constraints. Such a Markov chain will be irreducible if any graph can be reached from any other graphs. Also, the chain will be there if a non-zero self-loop probability to guarantee the chain can stay unchanged. If the "random walk" is governed by a doubly stochastic transition matrix, the stationary distribution for the Markov chain is uniform. The control parameters can include topological type, number of nodes, maximum and minimum number of arcs, maximum number of parents of each node, number of states of each node, skewness of the CPTs, and so on. In our experiment, we set the nodes to binary. We consider three topology types: polytree, two-level singly connected networks and multiply connected networks. After generating the network and CPTs, we can randomly generate a set of evidence nodes of the given type (predictive, diagnostic or random) to form a complete MPE instance. Another aspect of instance features is the user's requirement on computational resources, i.e., the user can provide a computational "deadline" for the MPE problem. Solutions must return before the deadline.

7.5 Experiment Setups and Environment

In the following experiments, we apply the proposed machine learning-based approach to investigate algorithm selection for finding the MPE. The MPE algorithms include an exact algorithm: the clique-tree propagation algorithm; two stochastic sampling algorithms: Gibbs sampling and forward sampling; two heuristic search algorithms: random restart hill-climbing and Tabu search; and one hybrid algorithm, the Ant Colony Optimization (ACO) algorithm. The MPE instance features we will investigate include number of nodes, network topological type, network connectedness, the maximum number of parents, the network CPT skewness, and the proportion and distribution of the evidence nodes.

The general MPE problem is NP-hard, which means we can not expect a polynomial time algorithm assuming $P \neq NP$. Exact clique-tree propagation algorithm can only solve polytrees and sparse networks efficiently. So our first goal is to identify the class of MPE instances for which the clique-tree propagation algorithm is applicable.

When the exact algorithm is not applicable (most probably due to an out-of-memory error), we need to look at various approximate algorithms. Our second goal is to learn the predictive model that can determine which approximate algorithm is

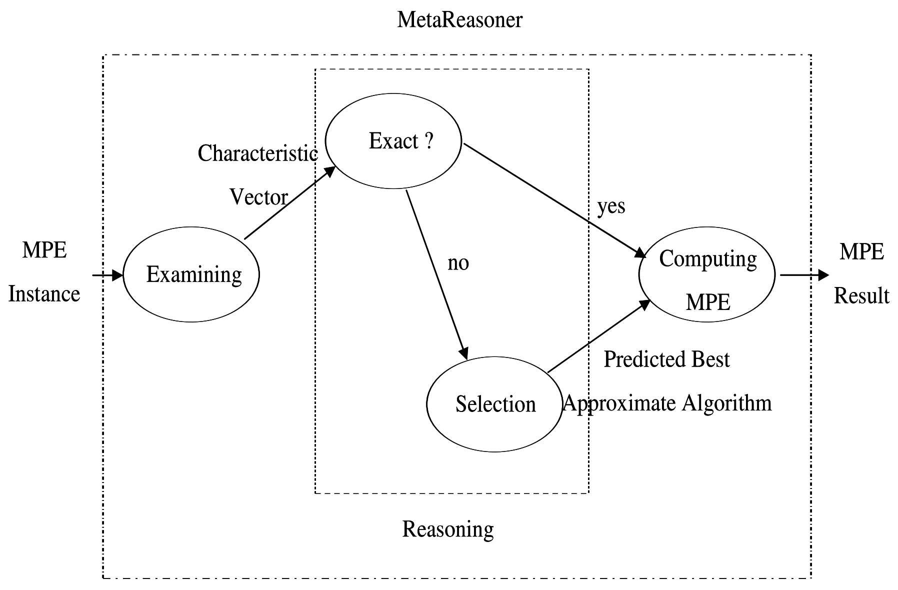


Figure 7.7: Algorithm Selection Meta-Reasoner for the MPE Problem

the best for the input MPE instance according to its characteristics. So far, the most effective way to fairly compare different heuristic algorithms is to allow all algorithms to consume the same amount of computation resources, with distinctions being based on the quality of solutions obtained [RU01]. In our experiments, we will give each algorithm a given numbers of samples or search-points and then compare the quality of solutions that each algorithm returns. The algorithm returns the highest MPE will be labelled as "winner". We also record when the highest MPE is generated by the algorithm. If two algorithms return the same MPE, the one that spends less time (numbers of samples or search points) will be labelled as "winner". The working procedure of the MPE algorithm selection meta-reasoner is illustrated in Figure 7.7.

Again, our experiments consists of three phases: data preparation, model induction, and model evaluation. In data preparation phase, we first generate MPE instances with different characteristic values. We then run all algorithms on all training instances and collect the performance data to make the training dataset. We then run various machine learning algorithms on the training data to induce the predictive algorithm selection models. Like what we did for sorting algorithm selection, we will consider three different kinds of models: decision tree learning (C4.5), naive Bayes classifier, and Bayesian network learning (K2). Finally, we evaluate the learned classifiers and the MPE algorithm selection system.

7.6 Experimental Results and Evaluation: the Induction of Algorithm Selection Models for Finding the MPE

In this section, we conduct a set of experiments to induce predictive algorithm selection models for the MPE problem. We will first extract characteristics of some real world Bayesian networks. We then generate our training datasets based on these real world characteristics. Once we have the training data, we apply machine learning algorithms to induce the algorithm selection models. Finally, we evaluate the learned models. Our first experiment is designed to learn the model to decide when the exact clique-tree propagation algorithm is applicable to the input MPE instance. The first experiment determines when the exact algorithm is applicable, i.e., it learns the concept of "exactly computable". The second experiment is designed to select the best feature subset for approximate algorithm selection. In the third experiment, we induce the approximate algorithm selection model that selects the best among a set of sampling and search-based approximate MPE algorithms. The fourth experiment compares the MPE algorithms' performance on different specific datasets. The fifth experiment is used to evaluate the learned model as a meta-level reasoner for MPE algorithm selection.

7.6.1 Characteristics of Real World Bayesian Networks

Since the space of all possible MPE instances is infinitely large and at the same time, many extreme characteristics are rarely encountered in real world applications, it's reasonable and necessary to consider only a subset of it, the set of "real world

Table 7.1: Characteristics of Real World Bayesian Networks

name	n_nodes	n_arcs	conn	n_roots	maxParents	skewness	maxClique
alarm	37	46	1.24	12	4	0.84	5
barley	413	602	1.46	76	2	0.87	6
cpcs179	179	239	1.36	12	8	0.76	9
cpcs54	54	108	2.0	13	9	0.25	15
diabetes	413	602	1.46	76	2	0.87	6
hailfinder	56	66	1.18	17	4	0.50	5
insurance	27	52	1.93	2	3	0.70	8
link	724	1125	1.55	184	3	0.68	> 22
munin1	189	282	1.49	34	3	0.88	> 22
munin2	1003	1244	1.24	249	3	0.89	9
munin3	1041	1397	1.34	259	2	0.55	11
pigs	441	592	1.35	145	2	0.55	11
water	32	66	2.06	8	5	0.75	11

problems" (RWP). In order to simulate the set of real world Bayesian networks, we first extract the real world distributions of all characteristic parameters from a collection of real world samples, then generate Bayesian networks and MPE problem instances based on the extracted distributions. Once we can generate synthetic real world Bayesian networks and synthetic instances of real world inference problems, we can then run our candidate algorithms on these instances to generate the training data.

We have collected 13 real world Bayesian networks. Let us call this dataset D_{RWBN}. Their characteristics are listed in Table 7.1. From the analysis results, we can see that the number of nodes varies from around 30 up to 1,000, the connectedness from 1.0 to 2.0, the maximum number of parents is below 10, and the skewness varies from 0.25 to 0.87. The maximum clique sizes, if available, are all smaller than 20.

7.6.2 The Training Datasets

We use the analysis results of real world Bayesian network characteristics to guide the generation of training datasets. More specifically, we set the ranges of the instance features as follows: *n_nodes* $\in$ {50, 100}; *conn* $\in$ { 1.0-1.1, 1.1-1.2, 1.2-1.5, 1.5-1.8, $>$ 1.8 }; *topology* $\in$ { polytree, twolevel, multiply }; *maxParents* $\in$ { 3, 5, 8, 10}; *skewness* $\in$ { 0.1, 0.5, 0.9 }; *evidPercent* $\in$ { 0.1, 0.2, 0.3 }; *evidDistribution* $\in$ { predictive, diagnostic, random }.

The first training dataset, D_{MPE1}, is used to induce the model that decides when the exact clique-tree propagation algorithm should be selected. Both time and space complexity of the exact algorithm are exponential in the maximum clique size of the underlying undirected graph. In practice, an exact clique-tree propagation algorithm is applicable only for sparse networks, which usually have smaller cliques. As the network becomes dense, exact clique-tree propagation algorithms become infeasible and often generate out-of-memory exceptions. In order to determine when to use an exact inference algorithm, we generate D_{MPE1} as follows: We first randomly generate networks with connectedness varying from 1.0 to 2.0 and maximum number of parents varying from 3 to 10. The number of nodes used are $\{30, 50, 80, 100, 120, 150, 200\}$. We then run Hugin on these randomly generated networks and record the performance. To perform inference, Hugin first compiles the network into a clique tree. We record the maximum clique size and label the network as "yes" instance if the compilation is successful. Otherwise if it throws out an out-of-memory error, we label the instance as "no". D_{MPE1} has four numeric attributes: *n_node*, *topology*, *connectedness*, and *maxParents*. The target class, *ifUseExactAlgorithm*, takes boolean values representing whether exact algorithm is applicable or not. We also put these 13 real world networks into D_{MPE1}, which contains a total of 1,893 instances.

The second training dataset, D_{MPE2}, is generated to induce the model to select the best among a set of approximate MPE algorithms. Since polytrees are easy for exact algorithm, D_{MPE2} only contains two-level and multiply networks. We generate a set of networks with different characteristic values and then run all 5 approximate algorithms on all networks with different evidence settings. We run the approximate inference algorithms for a given number of samples or search points and label the instance using the best algorithm that returns the best MPE value using less samples or search points. The total number of samples given to each algorithm was 300, 1000, or 3,000. D_{MPE2} has 8 attributes: *n_node*, *topology*, *connectedness*, *maxParents*, *skewness*, *evidPercent*, *evidDistri*, and *n_samples.* The target class is the best algorithm for this instance. D_{MPE2} contains 5,184 instances generated from 192 networks.

7.6.3 Experiment 1: Determining When the Exact MPE Algorithm Is Applicable

In experiment 1, we first apply machine learning algorithms on D_{MPE1} to induce the model to predict when the exact MPE algorithm should be used. We run the same 6 learning schemes used in the previous chapter – C4.5, naive Bayes, Bayes networks, bagging, boosting and stacking – and compare the results to see which one learns the best model; i.e., the one that has highest classification accuracy. The statistic of D_{MPE1} is shown in Table 7.2.

Table 7.2: Statistics of Attribute Values in D_{MPE1}

	n_nodes	conn	maxParents
Minimum	27	0.97	2
Maximum	1,041	4.27	10
Mean	122.30	1.53	4.50
StdDev	81.74	0.49	1.30

	topology			ifUseExactAlgo	
label	multiply	twolevel	polytree	yes	no
count	1293	278	322	1221	672

The experimental results are listed in Table 7.3. We can see that boosting C4.5 has the highest classification accuracy of 94.81%. The second and third best models are C4.5 and bagging C4.5. We also notice that NaiveBayes has the worst performance of

Table 7.3: Classification Accuracy of 6 Different Learning Schemes on D_{MPE1}

	C45	NaiveBayes	BayesNet	Bagging	Boosting	Stacking
accuracy (%)	94.80	82.79	90.06	94.75	94.81	94.56
StdDev (%)	0.27	0.36	0.24	0.25	0.23	0.45

only 82.79%, while all other inducers' classification accuracies are higher than 90%.

Since C4.5 and boosting have almost the same classification accuracy but C4.5 has a simpler model, we will use C4.5 as the best model for exact MPE algorithm selection. The confusion matrix of C4.5 is as follows:

$$C4.5 : \begin{pmatrix} a & b & <-- & classified\ \ as \\ 1,153 & 68 & | & a = yes \\ 31 & 641 & | & b = no \end{pmatrix}$$

The learned decision tree is shown in Figure 7.8. From the structure of the tree we can see that the basic rule for exact algorithm selection is that *exact clique-tree propagation algorithm is applicable if the network is small or sparse.*

We also test the learned C4.5 and boosting model on 13 real world networks. The classification accuracy of C4.5 is 84.62%; i.e., it correctly classify 11 out of 13 real world instances. The confusion matrix is as follows:

$$C4.5 : \begin{pmatrix} a & b & <-- & classified\ \ as \\ 9 & 2 & | & a = yes \\ 0 & 2 & | & b = no \end{pmatrix}$$

If we use boosting C4.5, all 13 networks can be correctly classified. The confusion matrix is as follows:

$$C4.5\ Boosted : \begin{pmatrix} a & b & <-- & classified\ \ as \\ 11 & 0 & | & a = yes \\ 0 & 2 & | & b = no \end{pmatrix}$$

We have also tried cost-sensitive classification. It gives the same result for both datasets. The cost matrix used is:

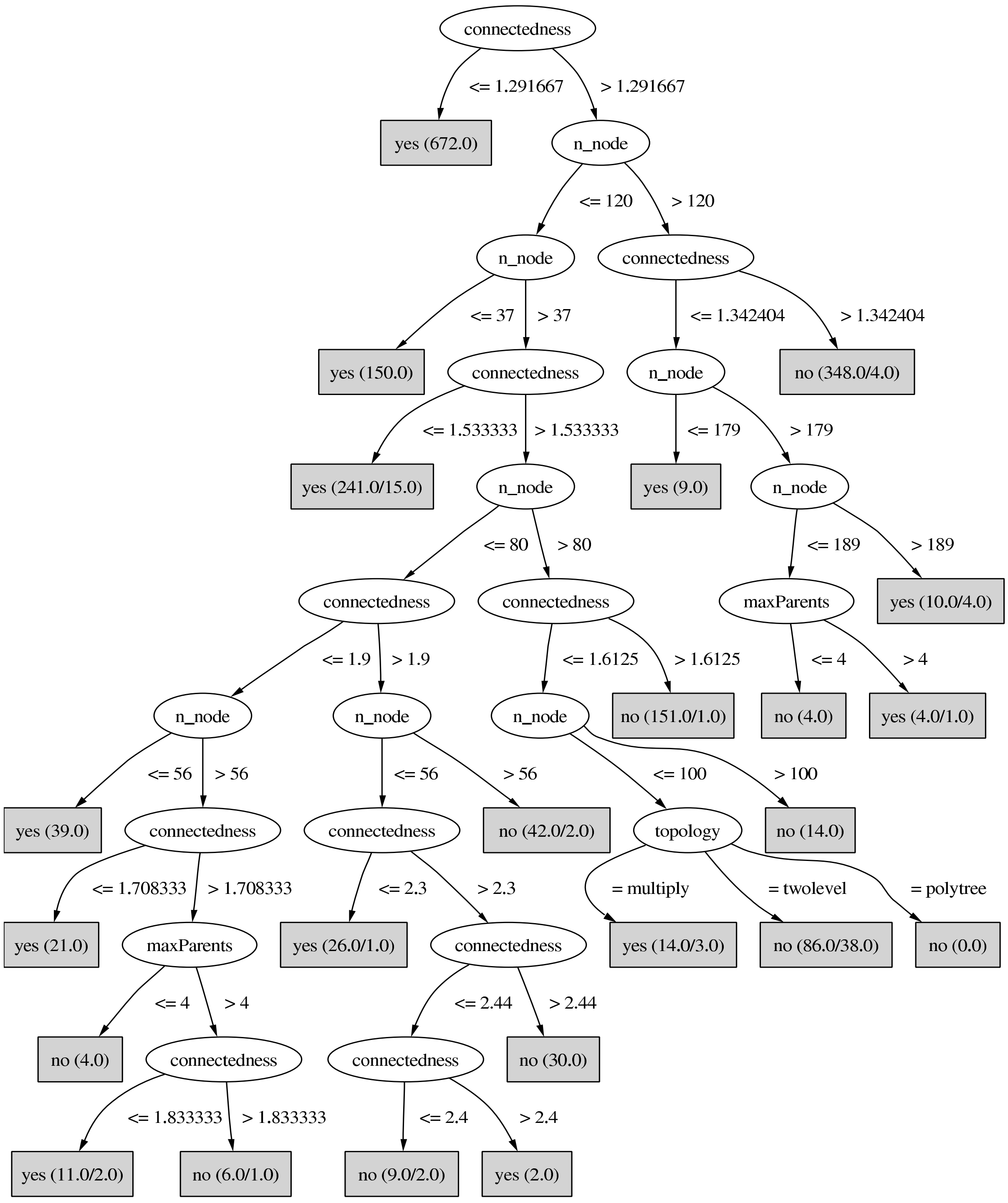


Figure 7.8: The Learned Decision Tree for Exact MPE Algorithm Selection

```
0   1  % if true class yes and prediction no, penalty is 1
10  0  % if true class no and prediction yes, penalty is 10
```

7.6.4 Experiment 2: Wrapper-based Feature Selection

In the following experiments, we look at the approximate MPE algorithm selection problem. The training dataset used is D_{MPE2}. The format of D_{MPE2} is shown in Table 7.4. It contains 5,184 instances. Each instance has 9 attributes. The first 8 are predictive attributes and the last one is the target class attribute which labels the best approximate algorithm for this instance. The statistics of D_{MPE2} are listed in Table 7.5. We can see that Gibbs sampling has never been the winner while ant colony optimization algorithm is the best for nearly half of the instances.

Table 7.4: Format of Training Dataset D_{MPE2}

#nodes	topology	conn	maxParents	skewness	evid%	evidDist	#samples	bestAlgo
50	multiply	4.56	9	0.1	10	predictive	300	multi_hc
50	multiply	4.56	9	0.1	10	random	300	aco
...	...	...	...	...	...	...	...	...
100	multiply	3.80	8	0.5	30	diagnostic	3000	aco
100	multiply	3.80	8	0.5	30	random	3000	aco

Like what we did for sorting, we first apply a GA-wrapped C4.5 feature selection classifier to see which feature subset is the best. The wrapper uses C4.5 as the evaluation classifier to evaluate the fitness of feature subsets. A simple genetic algorithm is used to search the attribute space. Both the population size and number of generations are 20. The crossover probability is 0.6 and the mutation probability is 0.033. The configuration and the output of the GA wrapper is shown in Figure 7.9. The feature subset selected by the GA is { *n_node*, *skewness*, *evidPercent*, *evidDistri*, *n_samples* }. The classification accuracy of the induced model is 76.97%. The confusion matrix is as follows:

Table 7.5: Statistics of Attribute Values in D_{MPE2}

	#nodes	**conn**	**maxParents**	**skewness**	**evid%**	**#samples**
Minimum	50	1.19	3	0.09	10	300
Maximum	100	4.88	10	0.90	30	3,000
Mean	75	2.49	5.64	0.50	20	1,433
StdDev	25	1.26	2.43	0.33	10	1,144

	topology		**evidDist**		
label	multiply	twolevel	predictive	diagnostic	random
count	3,240	1,944	1,728	1,728	1,728

	bestAlgorithm				
	gibbsSampling	forwardSampling	multiHC	tabu	aco
count	0	862	1,077	578	2,667
percentage	0%	16.62%	20.78%	11.15%	51.45%

$$\left(\begin{array}{llllllll} a & b & c & d & e & <-- & classified\ \ as \\ 0 & 0 & 0 & 0 & 0 & | & a = gibbs\_sampling \\ 0 & 645 & 0 & 0 & 217 & | & b = forward\_sampling \\ 0 & 8 & 848 & 207 & 14 & | & c = multi\_hc \\ 0 & 28 & 87 & 422 & 41 & | & d = tabu \\ 0 & 435 & 98 & 59 & 2075 & | & e = aco \end{array}\right)$$

In the following experiments, we used the selected feature subset to learn the predictive algorithm selection model.

7.6.5 Experiment 3: Determining the Best Model for Approximate MPE Algorithm Selection

In this experiment, we apply machine learning algorithms on the selected feature subset of D_{MPE2} to induce the model for the selection of approximate MPE algorithms.

We ran the same six learning schemes on the selected subset of D_{MPE2} to see which learns the best predictive model.The experimental results are shown in Table 7.6. From the result, we can see that the model induced by C4.5 has the highest classification accuracy of 77.75%. Naive Bayes classifier has the worst performance.

```
 === Run information ===
Attribute Subset Evaluator (supervised, Class (nominal): 9 best_algorithm):
 Wrapper Subset Evaluator
 Learning scheme: weka.classifiers.j48.J48
 Scheme options: -C 0.25 -M 2
 Accuracy estimation: classification error
 Number of folds for accuracy estimation: 5
Selected attributes: 1,5,6,7,8 : 5
  n_node, skewness, evdiPercent, evidDistri, n_samples
```

Figure 7.9: Parameters and Output of Attribute Selection GA-Wrapper on D_{MPE2}

The classification accuracy of Bayesian networks learning K2 is 76.08%. The learned Bayesian network is shown in Figure 7.10.

In the learned network two evidence nodes are disconnected from the graph. We can draw two conclusions from this result: First, it shows that number of nodes, skewness of the CPTs and the number of samples are three most important features for MPE algorithm selection. Second, since we know from domain knowledge and statistics of the training data that there exists a dependency relationships between evidence characteristics and sampling algorithm performance, the result implies that these dependencies are too weak for K2 to capture due to K2's greedy search strategy. The classification accuracies reported are test accuracies computed by ten-fold cross validation. We draw the classification accuracies of all 10 folds for C4.5, naive Bayes, and K2 in Figure 7.11.

Table 7.6: Classification Accuracy of 6 Different Learning Schemes on D_{MPE2}

	C4.5	NaiveBayes	BayesNet	Bagging	Boosting	Stacking
accuracy (%)	77.75	72.77	76.08	75.44	77.16	77.36
StdDev (%)	0.23	0.03	0.01	0.27	0.26	0.32

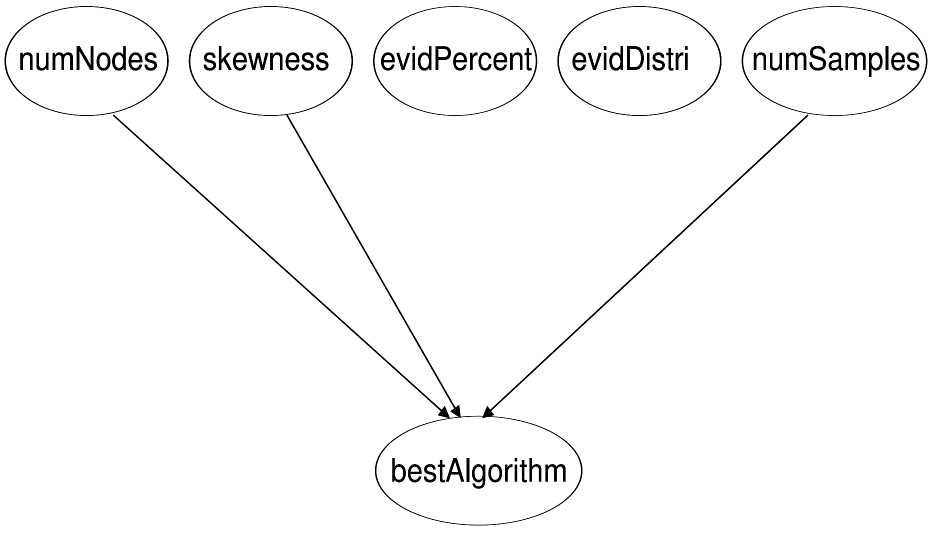


Figure 7.10: The Learned BN for Approximate MPE Algorithm Selection

The confusion matrices of C4.5, naive Bayes classifier and K2 are shown as follows:

$$C4.5 : \begin{pmatrix} a & b & c & d & e & <-- & classified\ \ as \\ 0 & 0 & 0 & 0 & 0 & | & a = gibbs\_sampling \\ 0 & 684 & 0 & 0 & 178 & | & b = forward\_sampling \\ 0 & 9 & 856 & 203 & 9 & | & c = multi\_hc \\ 0 & 25 & 88 & 424 & 41 & | & d = tabu \\ 0 & 433 & 96 & 61 & 2077 & | & e = aco \end{pmatrix}$$

$$naiveBayes : \begin{pmatrix} a & b & c & d & e & <-- & classified\ \ as \\ 0 & 0 & 0 & 0 & 0 & | & a = gibbs\_sampling \\ 0 & 858 & 0 & 0 & 4 & | & b = forward\_sampling \\ 0 & 9 & 1016 & 43 & 9 & | & c = multi\_hc \\ 0 & 28 & 290 & 222 & 38 & | & d = tabu \\ 0 & 833 & 134 & 23 & 1677 & | & e = aco \end{pmatrix}$$

$$K2 : \begin{pmatrix} a & b & c & d & e & <-- & classified\ \ as \\ 0 & 0 & 0 & 0 & 0 & | & a = gibbs\_sampling \\ 0 & 492 & 0 & 0 & 370 & | & b = forward\_sampling \\ 0 & 7 & 912 & 147 & 11 & | & c = multi\_hc \\ 0 & 25 & 142 & 370 & 41 & | & d = tabu \\ 0 & 340 & 98 & 59 & 2170 & | & e = aco \end{pmatrix}$$

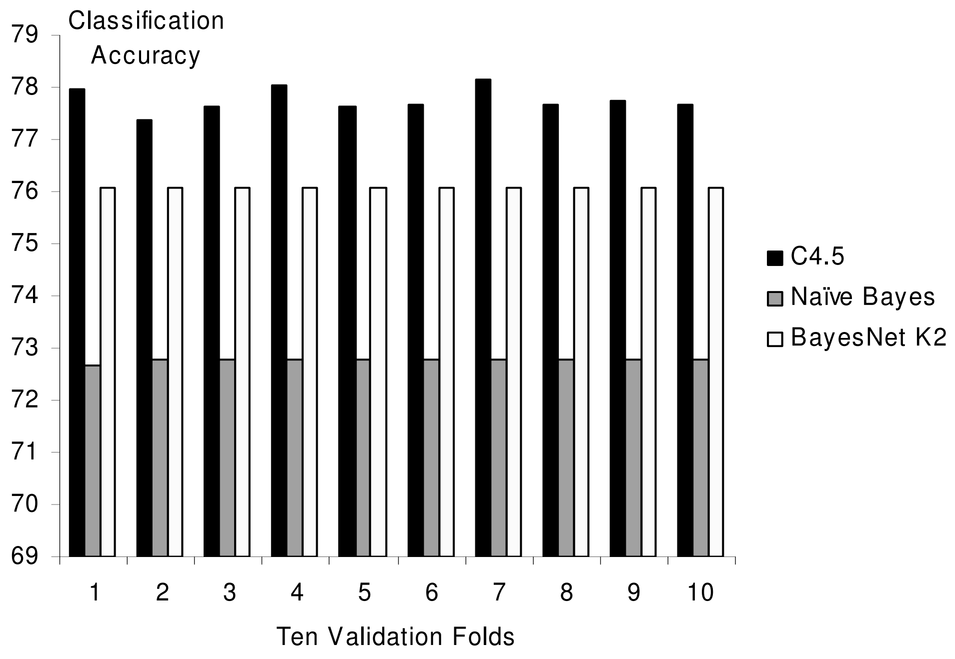


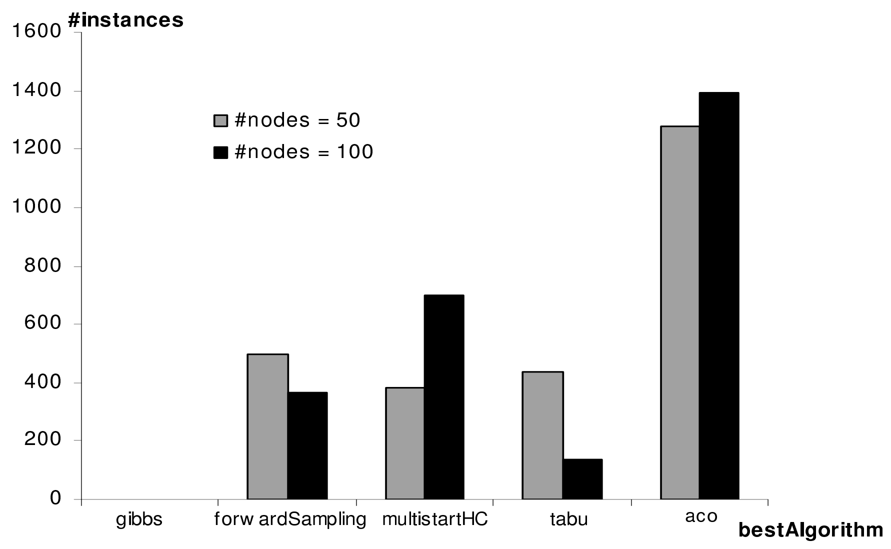
Figure 7.11: Classification Accuracies of Ten-fold Cross Validation

7.6.6 Experiment 4: MPE Algorithm Performance on Specific Datasets

We know from the result of feature selection that *n_node*, *skewness*, *n_samples*, *evidPercent* and *evidDistribution* are the most relevant features for approximate MPE algorithm selection. In this experiment, we use these features to partition D_{MPE2} into smaller subsets and compare the algorithm performance on these resulting specific sub-datasets.

Partitioning D_{MPE2} by Number of Nodes

Figure 7.12 shows the partition of D_{MPE2} by *n_nodes*. We can see that number of nodes affects the relative performance of two search algorithms while forward sampling and ACO are almost not affected. As the network size increases from 50 to 100, multi-



n_Nodes	Number of Times of Being bestAlgorithm				
	gibbsSampling	forwardSampling	multiHC	tabu	aco
50	0	496	380	439	1,277
100	0	366	697	139	1,390

Figure 7.12: Partitioning D_{MPE2} by Number of Nodes

start hill climbing becomes the best algorithm more frequently and the chances for tabu search being the best drops significantly. This phenomenon can be explained by the constant size of the tabu list used. When network becomes larger while the tabu list remain the same size, the tabu list becomes relatively smaller. This may affect tabu search's performance and make it lose its best algorithm position to multi-start hillclimbing.

Partitioning D_{MPE2} by Number of Samples

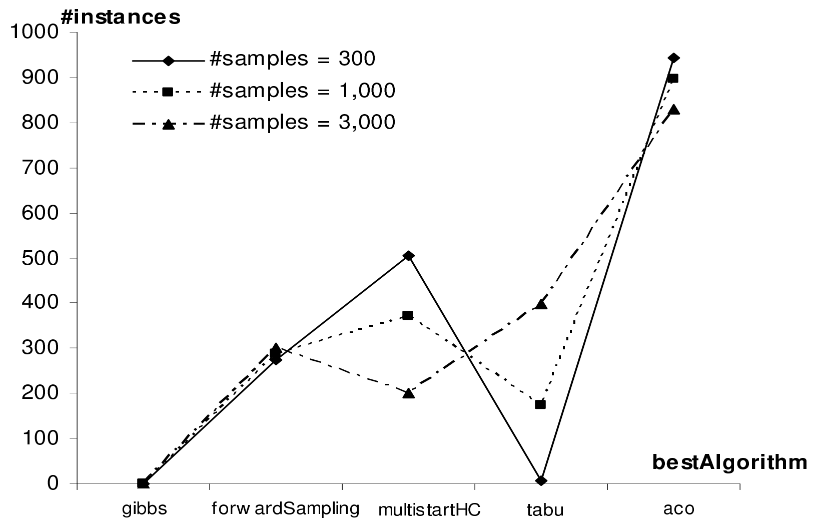
The time used by the algorithm is directly proportional to the number of samples. Figure 7.13 shows the partition of D_{MPE2} by *n_samples*. Again, the relative performances of two search algorithms are affected, but forward sampling and ACO's are not. When the given number of samples increases from 300 to 1,000 to 3,000, tabu search becomes the best algorithm more often and multi-start hillclimbing loses its top rank. Tabu search seems to be able to utilize available number of search points better than multi-start hillclimbing.

Partitioning D_{MPE2} by Skewness

Figure 7.13 shows the partition of D_{MPE2} by CPT skewness. we can see that skewness has an significant influence on the relative performance of these algorithms. When the skewness is low, the search space is flat and search algorithms perform much better than sampling algorithms. Multi-start hillclimbing wins the best algorithm two times more than tabu search. When the skewness is around 0.5, ACO outperforms all other algorithms almost of the time. When the skewness increases to 0.9, forward sampling and ACO are the winners. We should also notice that forward sampling works better only for highly skewed networks and ACO works for both highly-skewed networks and medium-skewed networks.

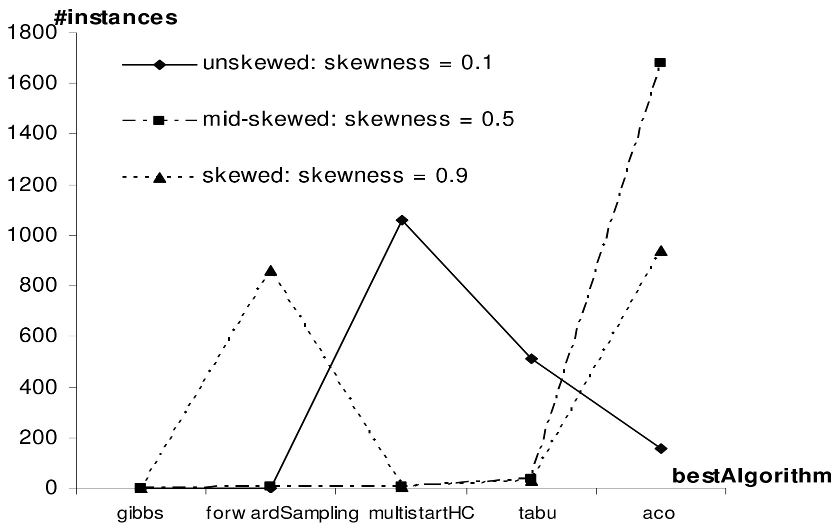
Partitioning D_{MPE2} by *evidPercent*

The more evidence nodes we have, the less likely the evidence is. Likelihood of the evidence directly affects the sampling algorithm's performance. Figure 7.15 shows the



n_Samples	Number of Times of Being bestAlgorithm				
	gibbsSampling	forwardSampling	multiHC	tabu	aco
300	0	274	505	7	942
1,000	0	286	372	174	896
3,000	0	302	200	397	829

Figure 7.13: Partitioning D_{MPE2} by Number of Samples



skewness	Number of Times of Being bestAlgorithm				
	gibbsSampling	forwardSampling	multiHC	tabu	aco
0.1	0	0	1059	512	157
0.5	0	4	9	174	1677
0.9	0	858	9	28	942

Figure 7.14: Partitioning D_{MPE2} by CPT Skewness

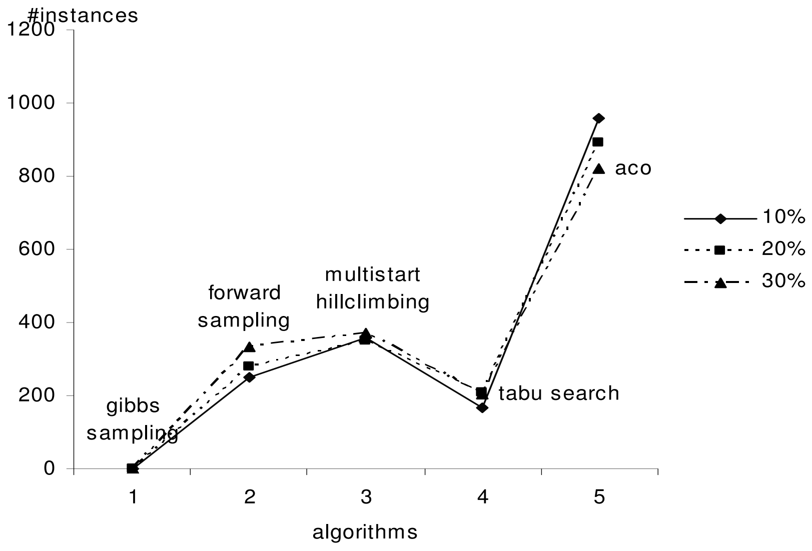
partition of D_{MPE2} by *evidPercent*. In general, changing evidence percentage does not affect two search algorithms' relative performance. But it does affect forward sampling and ACO. From the curves, we can see that ACO is out-performed by forward sampling as the percentage of evidence nodes increases from 10% to 30%. We should also note that evidence percentage's influence is much weaker than skewness'.

Partitioning D_{MPE2} by *evidDistribution*

Figure 7.16 shows the partition of D_{MPE2} by *evidPercent*. We can see that the relative performance of multi-start hillclimbing is not affected by evidence distribution. Tabu search is only slightly affected. It seems as though diagnostic inference is relatively hard for forward sampling but easy for ACO. Also, random distributed evidence is relatively hard for ACO but easy for forward sampling.

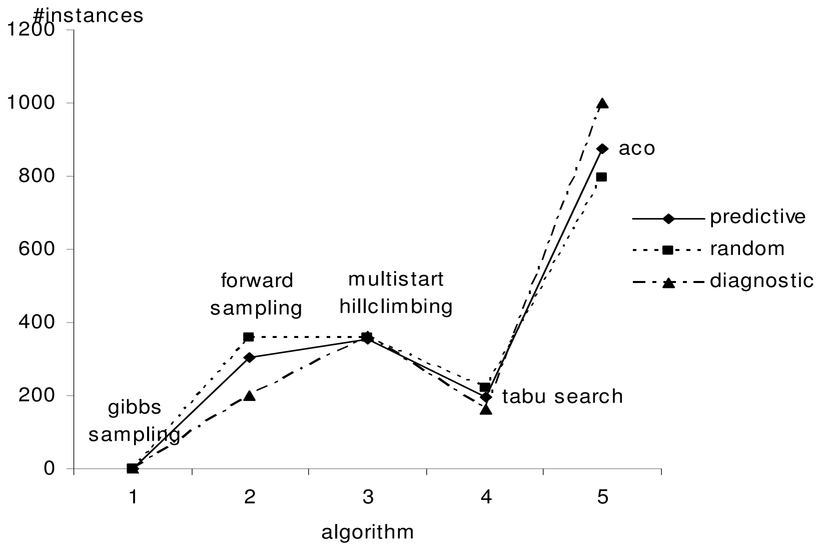
Running All Algorithms on Three Real World Networks

In this section, we show the results of running all algorithms on three real world networks without evidence: *ALARM*, *CPCS54*, and *CPCS179*. From Figure 7.17, we can see that on *ALARM*, both forward sampling and ACO find the same MPE after around 100 samples, but forward sampling hits the MPE earlier than ACO. Figure 7.18 shows the search history of each algorithm running on CPCS54. The top figure contains first 1,000 samples and the bottom figure contains total 5,000 samples. We can see that in the first 300 samples, ACO is leading. After the first 300 samples, however, forward sampling takes the lead until tabu search finds a better solution after around 900 samples. However, the final winner is multi-start hillclimbing after around 1,990 samples. This example shows that number of samples is an important factor in determining the best algorithm for unskewed networks. Figure 7.19 shows the search history of all algorithms on CPCS179. The trend in Figure 7.19 is very similar to Figure 7.17. This is because both *ALARM* and *CPCS179* have skewed CPTs.



evidPercent(%)	Number of Times of Being bestAlgorithm				
	gibbsSampling	forwardSampling	multiHC	tabu	aco
10	0	248	358	165	957
20	0	281	348	208	891
30	0	333	371	205	819

Figure 7.15: Partitioning D_{MPE2} by Evidence Percentage



evidDistri	Number of Times of Being bestAlgorithm				
	gibbsSampling	forwardSampling	multiHC	tabu	aco
predictive	0	303	356	195	874
random	0	357	357	219	795
diagnostic	0	202	364	164	998

Figure 7.16: Partitioning D_{MPE2} by Evidence Distribution

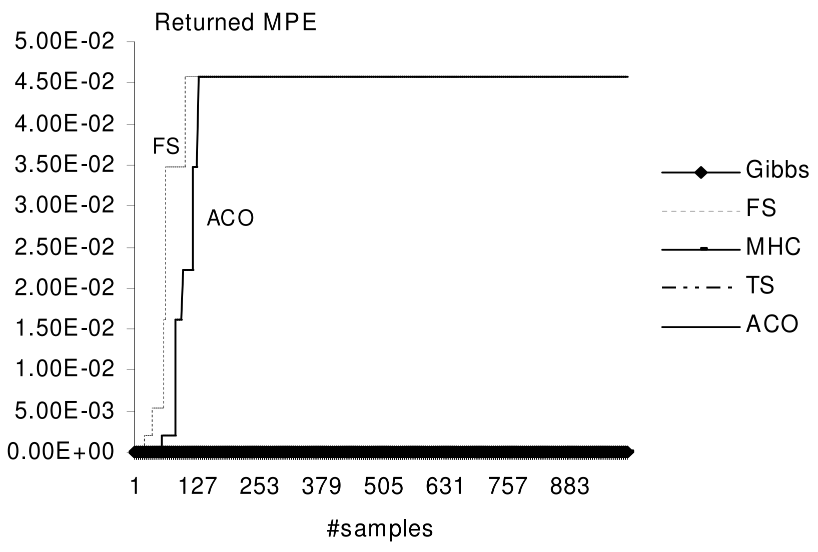
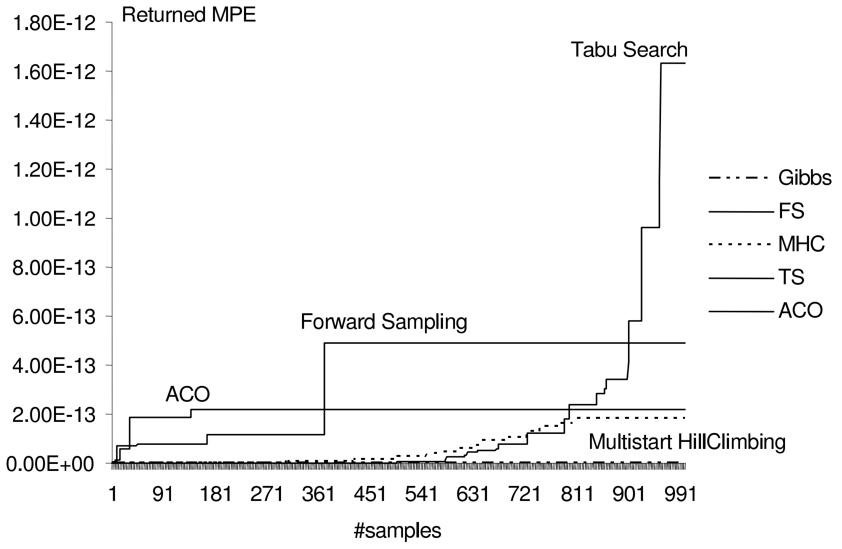


Figure 7.17: Search History of All Algorithms on ALARM Network

Verifying the results by C4.5 Decision Tree

From the previous analysis, we have learned some basic facts about the working regions of these approximate MPE algorithms. In general, we have observed the following results: First, *skewness* is the most important feature in determining the best algorithm. Both search algorithms and sampling algorithms work better on unskewed networks. Second, *n_nodes* and *n_samples* have an obvious influence on the relative performance of multi-start hillclimbing and tabu search. But they have very little influence on forward sampling and ACO. Third, *evidence percentage* and *distribution* affect forward sampling and ACO very much, but generally they do not affect search algorithms.

This knowledge can be verified from the decision tree learned by C4.5. The tree is shown in Figure 7.20. We can see that the root node is *skewness.* The left branch of the tree, representing low skewness instances, contains only multi-start hillclimbing and tabu search. The right branch of the tree, representing skewed instances, con-



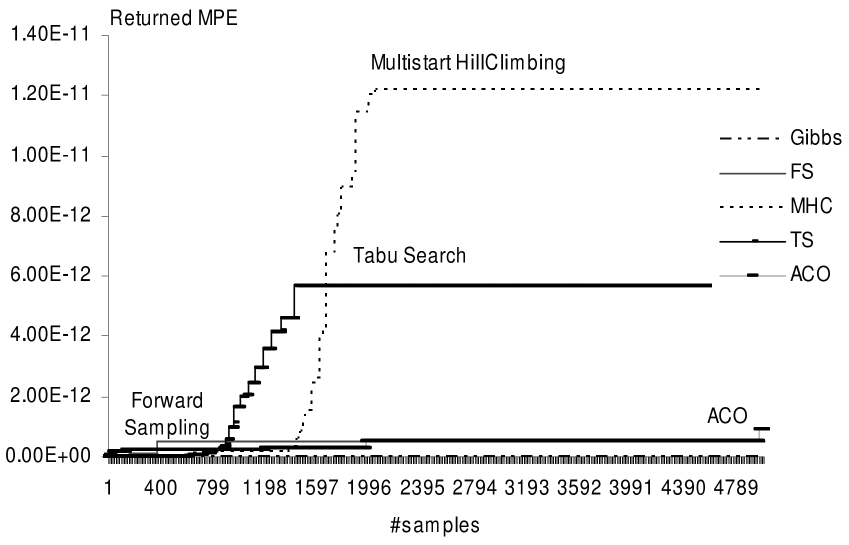
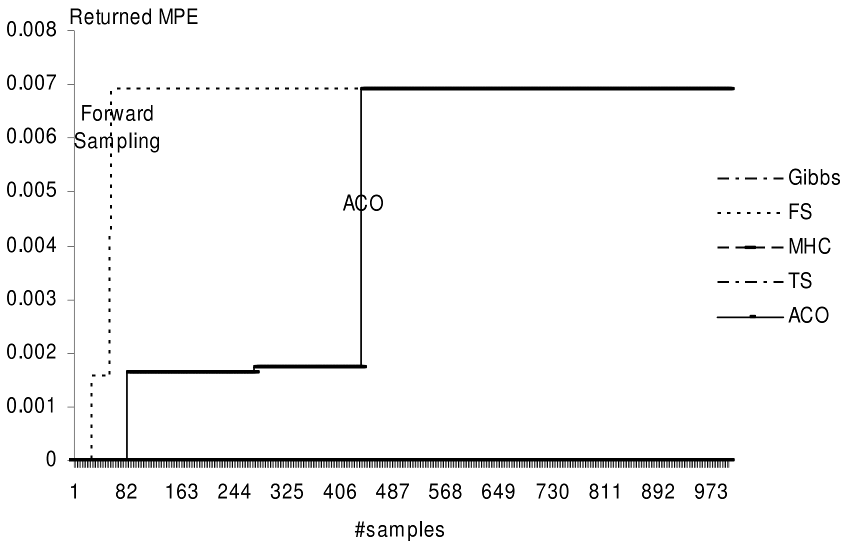


Figure 7.18: Search History of All Algorithms on CPCS54 Network



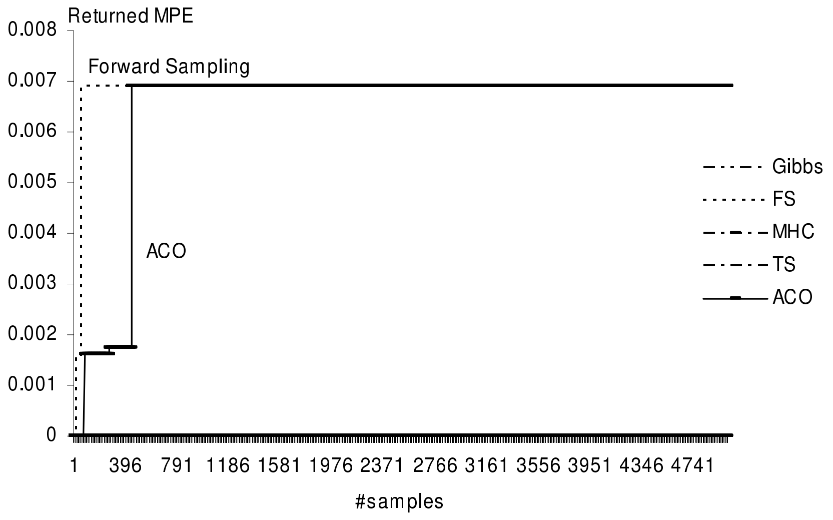


Figure 7.19: Search History of All Algorithms on CPCS179 Network

tains only forward sampling and ACO. Also, in the left branch, mainly *n_nodes* and *n_samples* are used to divide the instance space, which implies that their influence on these two search algorithms is significant. In the right branch, *evidPercent* and *evidDistribution* are used right after skewness to further divide the subbranches. Also, this is the region where forward sampling and ACO are highly competing with each other.

7.6.7 Experiment 5: Evaluating the MPE Algorithm Selection System

In this experiment, we evaluate the learned MPE algorithm selection system on two test datasets. The algorithm selection's first test dataset $D_{MpeTest}$ contains 405 instances. Statistics of $D_{MPETest}$ are listed in Table 7.7. The second test dataset is the real world dataset D_{RWBN}. The system contains two classifiers, one for exact algorithm selection, the other for approximate algorithm selection. Let us call them exactMPESelector and approximateMPESelector. For a given MPE instance, exactMPESelector determines if exact clique-tree propagation algorithm should be used. If the classification result is "yes", the system then execute the exact MPE algorithm. If the classification result is "no", the approximateMPESelector will be used to select the best approximate algorithm. The selected algorithm is then executed and the final MPE value returned.

System Evaluation on Synthetic Networks

We first apply exactMPESelector on $D_{MpeTest}$. It identifies 243 "yes" instances correctly. Then we apply approximateMPESelector on the rest 162 "no" instances. The result shows that there are 123 correctly classified instances and 39 incorrectly classified instances. The classification accuracy is 75.93%. The confusion matrix is as follows:

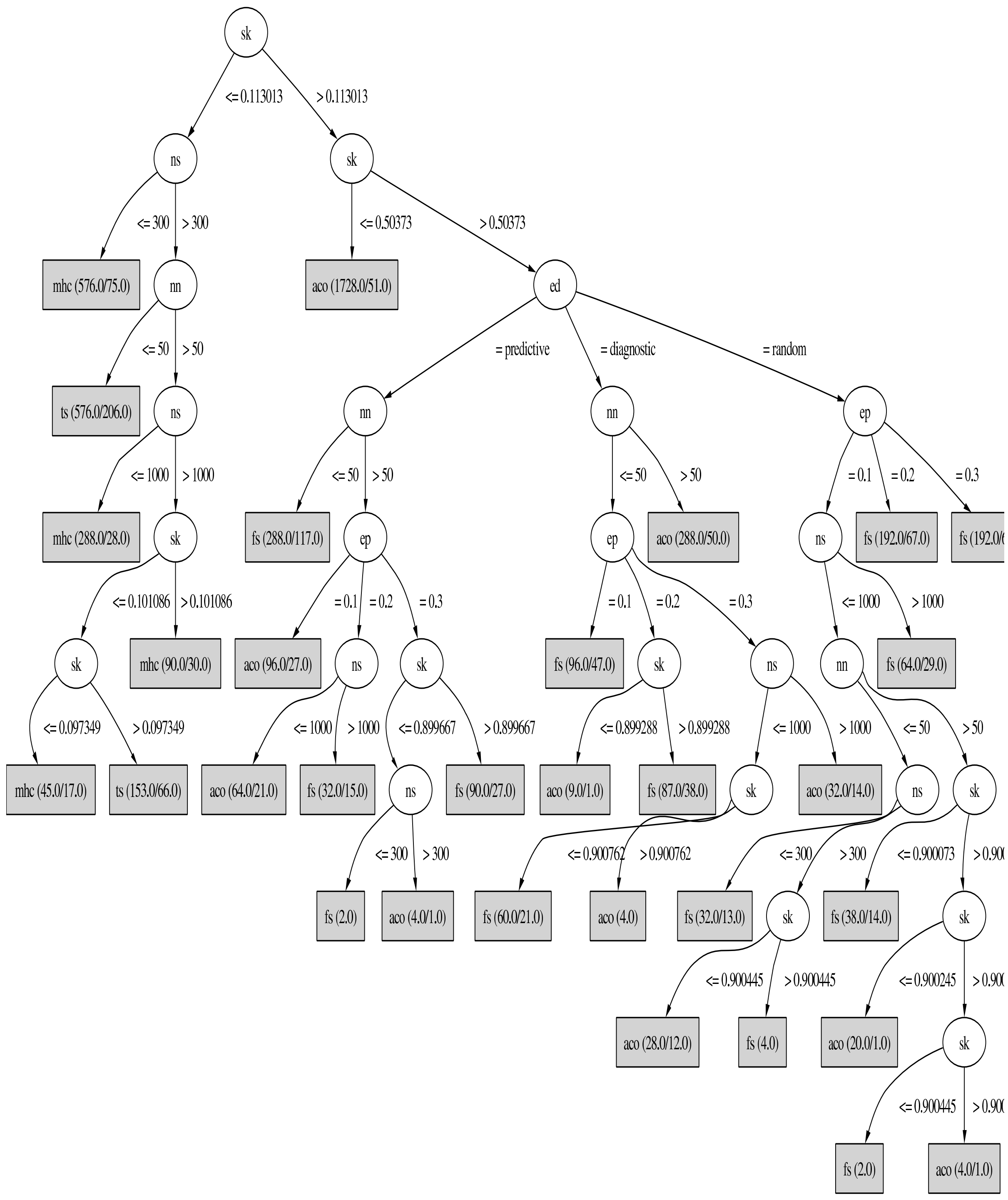


Figure 7.20: The Learned Decision Tree for Approximate MPE Algorithm Selection

Table 7.7: Statistics of Attribute Values in $D_{MPETest}$

	n_nodes	conn	maxParents
Minimum	50	0.98	3
Maximum	100	4.83	10
Mean	800	1.91	4.30
StdDev	24.52	1.20	2.12

	topology			ifUseExactAlgo	
label	multiply	twolevel	polytree	yes	no
count	135	135	135	243	162

$$\left(\begin{array}{llllllll}
a & b & c & d & e & <-- & classified\ \ as \\
0 & 0 & 0 & 0 & 0 & | & a = gibbs\_sampling \\
0 & 22 & 0 & 0 & 17 & | & b = forward\_sampling \\
0 & 0 & 17 & 4 & 1 & | & c = multi\_hc \\
0 & 0 & 0 & 4 & 2 & | & d = tabu \\
0 & 13 & 1 & 1 & 80 & | & e = aco
\end{array}\right)$$

To show that the algorithm selection system outperforms any single algorithm, we partition these 162 “no” instances into three groups according to their skewness. There are 27 unskewed instances, 54 medium-skewed instances, and 81 highly-skewed instances. For each group of instances, we plot the total MPE returned by each algorithm and compare it with the total MPE returned by the algorithm selection system. The results are shown in Figure 7.21, Figure 7.22 and Figure 7.23. On medium-skewed and highly-skewed instances, the algorithm selection system returns the largest total MPE values. On unskewed instances, the system returns the second largest MPE value of 2.1×10^{-26}. But the largest total, computed by multi-start hill climbing, is only 2.2×10^{-26}. The algorithm selection system's result is almost as good as it gets. Therefore, on all 162 instances, the algorithm selection system returns the largest total MPE value and outperforms all algorithms. The returned MPE values are listed in Table 7.8.

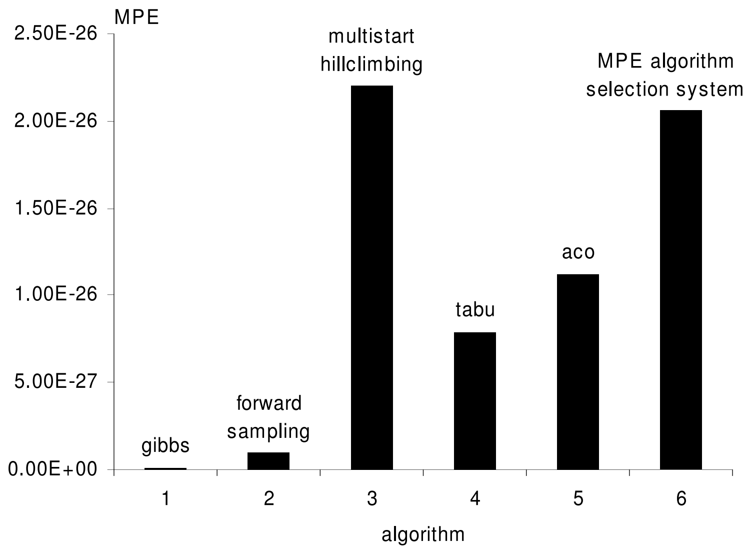


Figure 7.21: Total MPE of All Algorithms on Unskewed Instances

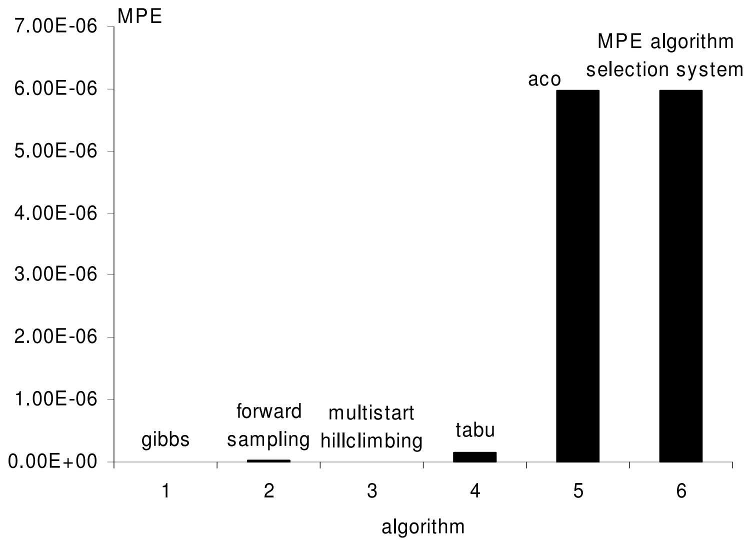


Figure 7.22: Total MPE of All Algorithms on Medium-skewed Instances

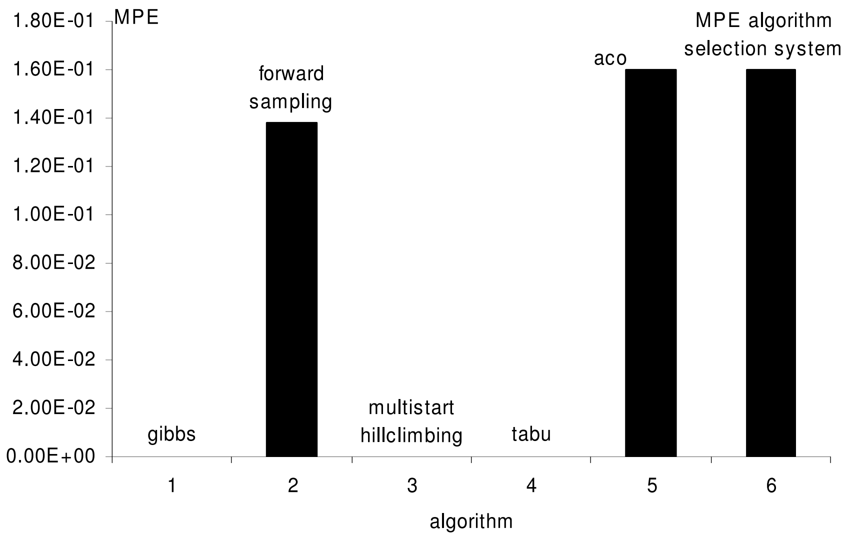


Figure 7.23: Total MPE of All Algorithms on Highly-skewed Instances

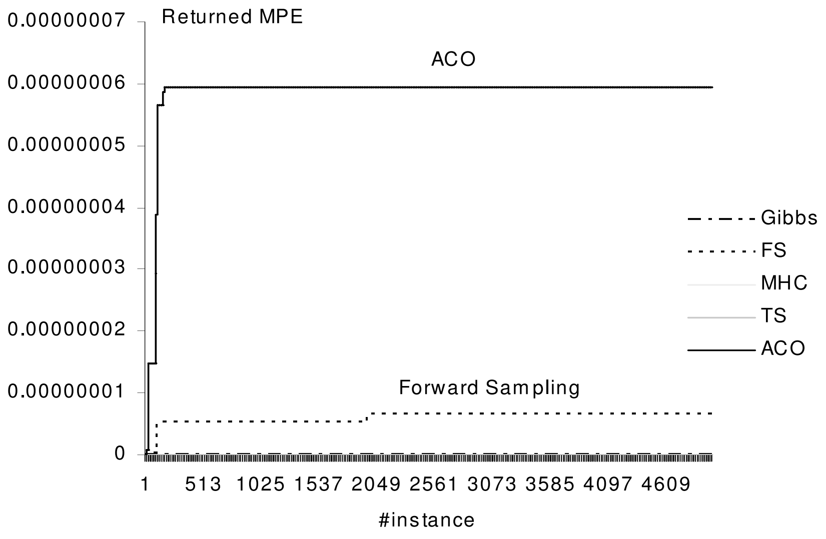


Figure 7.24: Search History of All Algorithms on Munin1 Network

Table 7.8: Total MPE Returned by All Algorithms and the Algorithm Selection System on Test Instances

skew	Gibbs	FS	MHC	TS	ACO	Total
0.1	4.7×10^{-29}	1.0×10^{-27}	2.2×10^{-26}	7.9×10^{-27}	1.1×10^{-26}	2.1×10^{-26}
0.5	1.4×10^{-14}	2.8×10^{-8}	6.2×10^{-9}	1.6×10^{-7}	6.0×10^{-6}	6.0×10^{-6}
0.9	1.9×10^{-46}	0.14	1.4×10^{-10}	2.5×10^{-14}	0.16	0.16

System Evaluation on Real World Networks

Now we test the system on real world networks. We have reported that using the boosting C4.5 classifier all 13 real world networks can be correctly classified by exactMPESelector. There are 11 "yes" networks. Exact MPE values of these 11 exactly computable networks are listed in Table 7.9 along with the predicted best algorithm and the best MPE returned by approximate algorithms. On these networks, all predicted best approximate algorithms also agree with actual best approximate algorithms.

The two "no" networks are *link* and *munin1*. The approximateMPESelector selects ACO as the best approximate algorithm for both networks. *link* has 724 nodes and a huge joint probability space of 5.77×10^{277} states. Its CPT skewness is 0.68. There are 20,502 numbers in its CPTs and 13,715 of them are zeros. Given 5,000 number of samples, all algorithms returned MPE of 0 for *link*. This is due to its huge state space and low skewness. By applying greedy sampling, we can get a MPE of 1.04×10^{-79}. *munin1* has 189 nodes. Its state space has 3.23×10^{123} states. Its CPTs are more skewed than *link* and has a skewness of 0.89. There are total 19,466 numbers in its CPTs and 10,910 of them are zeros. Given 5,000 number of samples, ACO returns the best MPE of 5.93×10^{-8}. Forward sampling finds the second best MPE of returned 6.61×10^{-9}. All other algorithms only return 0. These results are also summarized in Table 7.10. The search history of all approximate algorithms are shown in Figure 7.24.

The test results on both synthetic and real world networks illustrate that the pro-

Table 7.9: MPE of 11 Exactly Computable Bayesian Networks

Network	**Predicted Best Algo.**	**Predicted Best Appro. Algo.**	**Exact MPE**	**Approximate Best MPE**
alarm	exact	FS	0.04565	0.04565
barley	exact	ACO	3.67×10^{-37}	-
cpcs179	exact	ACO	0.0069	0.0069
cpcs54	exact	ACO	1.87×10^{-11}	5.78×10^{-12}
diabetes	exact	ACO	3.67×10^{-37}	-
hailfinder	exact	ACO	1.44×10^{-12}	3.44×10^{-14}
insurance	exact	FS	0.002185	0.002185
munin2	exact	ACO	8.74×10^{-37}	1.23×10^{-37}
munin3	exact	ACO	2.49×10^{-37}	7.07×10^{-40}
pigs	exact	ACO	5.03×10^{-88}	1.31×10^{-141}
water	exact	FS	3.08×10^{-4}	3.08×10^{-4}

Table 7.10: MPE of *Link* and *Munin1*

Network	**Number of Samples**	**Predicted Best Algo.**	**Actual Best Algo.**	**Best MPE**
link	5,000	aco	-	0
munin1	5,000	aco	aco	5.93×10^{-8}

posed machine learning-based approach can be used to solve the algorithm selection problem for the MPE problem. As a meta-level reasoner, the learned models can be used to make reasonable decision on selecting exact and best approximate MPE algorithms for the input MPE instance. The learned MPE algorithm selection system provides the best overall performance for solving the MPE problem.

7.7 Summary

In this chapter, we have studied the use of machine learning-based approaches to build an algorithm selection system for the MPE problem. The system consists of two predictive models (classifiers). The first one decides if exact MPE algorithm is applicable. Its test classification accuracy is 94.80%. If the first classifier classifies the instance as “not exactly computable”, the second classifier is then used to determine which approximate algorithm is the best for the input MPE instance. The second classifier has a classification accuracy of 76.97%. Different MPE instance characteristics have different properties and affect different algorithms' performance. The experimental results show that CPT skewness is the most important feature for approximate MPE algorithm selection. It also shows that search-based MPE algorithms work better on unskewed networks and sampling algorithms work better on skewed networks. Other features, such as *n_nodes*, *n_samples*, *evidPercent* and *evidDistri*, all affect these algorithms' relative performance to some degree, although not as strong as *skewness* does. Our learned algorithm selection system uses some polynomial time computable instance characteristics to select the best algorithm for the NP-hard MPE problem and gains the best overall performance in terms of the returned MPE values. This scheme could be used for algorithm selection of other NP-hard problems as well. In general, it is applicable to solve algorithm selection for any computationally hard problems in various fields.

Chapter 8

Conclusions

In this chapter we summarize the contributions of this work and identify the main issues to be refined and studied in the future.

8.1 Contributions

In this thesis we have studied the algorithm selection problem both theoretically and experimentally. We have also studied some multifractal properties of the joint probability space of Bayesian networks and how to apply them to solve the MPE problem.

Theoretically, we have shown the undecidability of the general automatic algorithm selection problem by applying Rice's theorem. We have also developed an abstract framework of problem hardness and algorithm performance based on Kolmogorov complexity and applied it to the study of GA-hardness.

Experimentally, we have proposed and implemented a machine learning-based algorithm selection system. The experimental results on sorting and the MPE problem have proven that this approach is useful for algorithm selection in both P and NP-hard computational problems. For P problems, time is the most important criteria. We accordingly look for instance features that can provide good classification accuracy and are easy to compute compared to the actual computation time. For NP-hard problems, the algorithm selection system consists of two classifiers. The first one encodes the concept of "exactly computable". The second one is responsible for selecting the best approximate algorithm.

In summary, the major contributions of this work consist of:

- A novel learning-based approach to automatic algorithm selection for sorting and finding the MPE.
- An abstract theoretical framework of problem hardness and algorithm performance based on Kolmogorov complexity, and proof of the infeasibility of a purely analytical approach to building automatic algorithm selection systems.
- A multifractal analysis of the joint probability distributions of Bayesian networks and the use of the multifractal meta-heuristic for solving the MPE problem.

Other contributions include:

- A study of GA-hardness using the problem hardness and algorithm performance framework.
- The development of a two phase Sampling-And-Search algorithm for finding the MPE using the multifractal property of the JPD.
- Applying Ant Algorithms to solve the MPE problem.
- The development of a set of random instance generation algorithms for using Markov chain technique.

The significance of this research lies in the following aspects:

1. This research is the first to systematically apply experimental algorithmic and machine learning methods to solve the algorithm selection problem. It provides a practical machine learning-based approach to build algorithm selection systems for both tractable and intractable problems. To the artificial intelligence and machine learning community, this research identifies an important

application field; − the empirical analysis of algorithms. The techniques developed in this research could also be very helpful in building real-time intelligent systems that require highly efficient solvers or reasoning engines. To the experimental algorithmics community, the methodology applied in this research introduces a set of powerful tools to analyze instance hardness and algorithm performance; − machine learning and uncertain reasoning approaches. The genetic algorithm and evolutionary computation community may benefit from the theoretical results and the experimental methodology as well; − our approach could be applied to help solve the notorious GA-hardness problem experimentally.

2. The discovery of multifractal properties of Bayesian networks' JPDs is very original. It points out that the structure of a JPD can be analyzed by means of the theoretical machinery developed in the field of fractals. Furthermore, we have shown how the theoretical insight leads to an approximate algorithm for finding MPE in Bayesian networks, an algorithm whose behavior can be predicted on theoretical grounds. To the community of uncertain AI and Bayesian networks, it may lead to a series of results that will push the boundaries of what we can do in both sampling and search-based algorithms. It also provides a promising direction to general NP-hard problem solving because the MPE problem (decision version) is NP-complete. It also provides a novel view of multifractals for the multifractal community. The computational capability of Bayesian networks may provide a powerful toolkit to manipulate multifractal models.

3. This research is the first to apply an ant colony optimization algorithm to solve the MPE problem. It is also the first one that thoroughly investigates the role of skewness in Bayesian network inferences. The study of CPT skewness' influence on sampling and search-based inference algorithms' performance provides new knowledge to our understanding of the working regions of these algorithms.

4. The random permutation generation algorithms developed provide for the first time a uniform approach of generate permutations with a given degree of different presortedness measures uniformly at random. This is very helpful to anyone who wants to experimentally study various sorting algorithms' performances.

8.2 Open Questions and Future Work

Many contributions of this work, including the methodology and the results of experimental and theoretical investigations, are interesting on their own. However, they also give rise to many questions and suggest a number of directions for future research. In the following, we briefly mention some of these issues.

8.2.1 Theoretical Aspects

Information in Computation

The concept of "information" seems to play an important role in instance complexity and algorithm performance. Knowing problem or instance specific information can help us design more efficient algorithms. However, not all information is easily accessible. Given an instance, what information can we extract out of it? How expensive it is? How can we measure the information that each algorithm contains about the instances that it aims to solve? How does this information affect (absolutely or relatively) instance hardness and algorithm performance? Also, the concept of information here seems to be different from Shannon's entropy and Kolmogorov's algorithmic information defined by program-size complexity. Since an algorithm before implementation is just an intuitive idea in a human's mind, the information it assumes could be wrong. This implies that the information in computation could be "negative". As pointed out by Gregory Chaitin [Cha02]: "Perhaps a new kind of algorithmic information theory could be built in which lying plays a role and then there would be negative information". But how can we exactly formalize this concept of "negative information"?

GA-hardness Study

We have applied our abstract framework of instance hardness and algorithm performance to the study of GA-hardness and propose some future directions. What could be done next is to verify our hypothesis by applying the machine learning approach to study GA-hardness. This will involve the problem of how to classify the space of all possible GAs by their performance on some practical controllable optimization problems.

8.2.2 Experimental Aspects

Random Generation

We have developed a set of random generation algorithms using the Markov chain approach. However, the rate of convergence of these chains have not been theoretically analyzed. Techniques used in [Sin93] can be applied to derive the rapidly mixing bounds of these Markov chains.

Efficient Approximate Algorithms for Computing INV

In sorting, a number of inversions is the best indicator of many sort algorithms' performance but it is almost as expensive as the actual sorting process to compute. It would be very useful to have an efficient algorithm that approximates the number of inversions in a given permutation.

Applying the Machine Learning-based Methodology to Algorithm Selection for Other Problems

We have applied the proposed learning-based approach for sorting and the MPE problem. As an algorithm selection technique, it can be used for other problems as well. For example, it can be used to solve the algorithm selection problem for belief updating and MAP. Belief updating is a $\#P$-problem and MAP is NP^{pp}-complete [Par02a]. In belief updating, the goal is to compute the marginal probabilities. MAP is even harder because it includes both marginalization and maximization. Although

CPT skewness plays plays an important role in the algorithm selection for the MPE algorithm, this may not be true for belief updating and MAP. It is interesting to investigate what feature is the most important one in algorithm selection within these problems.

Real-time Learning: Adaptive to Input Instances Changes

In many real-world applications, distributions of the input instances may change over time. Correspondingly, if the meta-reasoner can sense the distribution change of the input instances and update its reasoning model from time to time, a better accuracy can be achieved. This requires the meta-reasoner to put more weight on these most recent input instances and to be adaptive by conducting sort of real-time learning. It is interesting to investigate the related issues so as to build an autonomic algorithm selection system.

8.2.3 Multifractal Analysis Study

We have applied the multifractal property of the joint probability space to design an algorithm for the MPE problem. A natural followup is to investigate how to use it to design algorithms for belief updating as well. Another promising direction is that we can apply multifractal analysis to the solution space of other NP-hard problems such as $MAX - SAT$ and TSP. The multifractal meta-heuristic should be able to apply to any NP-hard combinatorial optimization problem as well as being able to solve any NP-hard combinatorial optimization problem.

Bibliography

[AH98] A. M. Abdelbar and S. M. Hedetniemi. Approximating MAPs for belief networks in NP-hard and other theorems. *Artificial Intelligence*, 102:21–38, 1998.

[AKS02] M. Agrawal, N. Kayal, and N. Saxena. PRIMES in P. 2002.

[AM88] S. Y. Abu-Mostafa. Random problems. *Journal of Complexity*, 4(4), 1988.

[Bet81] A. D. Bethke. *GAs as Function Optimizers*. PhD thesis, 1981.

[BG91] C. L. Bridges and D. E. Goldberg. The nonuniform Walsh-schema transform. In G. J. Rawlins, editor, *Foundations of genetic algorithms*, pages 13–22. Morgan Kaufmann, San Mateo, 1991.

[BH90] J. S. Breese and E. Horvitz. Ideal reformulation of belief networks. In *UAI90*, pages 129–144, 1990.

[BH91] A. Bunde and S. Havlin. *Fractals and Disordered Systems*. Springer, 1991.

[Boo63] J. Boothroyd. Algorithm 201: Shellsort. *Comm. ACM*, 8(6):445, Aug. 1963.

[Bor96] B. J. Borghetti. Inference algorithm performance and selection under contrained resources. Master's thesis, 1996.

[Bre96] L. Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996.

[Bro93] C. Brodley. Addressing the selective superiority problem: Automatic algorithm/model class selection. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 17–24, 1993.

[Bro94] C. Brodley. *Recursive Automatic Algorithm Selection for Inductive Learning.* PhD thesis, Amherst, 1994.

[CD00] J. Cheng and M. J. Druzdzel. AIS-BN: An adaptive importance sampling algorithm for evidential reasoning in large Bayesian networks. *Journal of Artificial Intelligence Research*, 13:155–188, 2000.

[CDMT93] A. Colorni, M. Dorigo, V. Maniezzo, and M. Trubian. Ant system for job-shop scheduling. *Journal of Operations Research, Statistics and Computer Science*, 34:39–54, 1993.

[CFGS02] L. S. Crawford, M. P. J. Fromherz, C. Guettier, and Y. Shang. A framework for on-line adaptive control of problem solving. In *AAAI Spring Symposium on Intelligent Distributed and Embedded Systems*, Stanford, CA, 2002.

[CH92] G. F. Cooper and E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9(4):309–347, 1992.

[Cha66] G. J. Chaitin. On the length of programs for computing finite binary sequences. *Journal of the Association of Computing Machinery*, 13:547–569, 1966.

[Cha87] G. J. Chaitin. *Algorithmic Information Theory.* Cambridge University Press, Cambridge, 1987.

[Cha91] E. Charniak. Bayesian networks without tears. *AI Magazine*, 12(4):50–63, 1991.

[Cha02] G. Chaitin. Personal communication, September 2002.

[Che85] P. Cheeseman. In defense of probability. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 1002–1009. Morgan Kaufmann, 1985.

[CKT91] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, IJCAI-91, Sydney, Australia*, pages 331–337, 1991.

[CL96] J. Culberson and J. Lichtner. On searching α-ary hypercubes and related graphs. In R. K. Belew and M. D. Vose, editors, *Foundations of Genetic Algorithms 4*, pages 263–290, 1996.

[Coh60] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20:37–46, 1960.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[Coo90] G. F. Cooper. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence*, 42:393–405, 1990.

[Coo00] S. A. Cook. The P versus NP problem. 2000.

[CT91] T. M. Cover and J. A. Thomas. *Elements of Information Theory.* John Wiley and Sons, 1991.

[Cul98] J. Culberson. On the futility of blind search: An algorithmic view of No Free Lunch. *Evolutionary Computation Journal*, 6(2):109–128, 1998.

[Dav91a] Y. Davidor. Epistasis variance: A viewpoint on GA-hardness. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms.* Morgan Kaufmann, 1991.

[Dav91b] T. Davis. *Toward an Extrapolation of the Simulated Annealing Convergence Theory onto the Simple Genetic Algorithm.* PhD thesis, 1991.

[DCG99] M. Dorigo, G. Di Caro, and L. M. Gambardella. Ant algorithms for discrete optimization. *Artificial Life*, 5:137–172, 1999.

[DG97] M. Dorigo and L. M. Gambardella. Ant colonies for the traveling salesman problem. *BioSystems*, 43:73–81, 1997.

[DH97] D. Costa D and A. Hertz. Ants can colour graphs. *Journal of the Operational Research Society*, 48:295–305, 1997.

[DL93] P. Dagum and M. Luby. Approximating probabilistic inference in Bayesian belief networks is NP-hard. *Artificial Intelligence*, 60:141–153, 1993.

[Dor92] M. Dorigo. *Optimization, Learning and Natural Algorithms.* PhD thesis, 1992.

[Dru94] M. J. Druzdzel. Some properties of joint probability distributions. In *UAI94*, pages 187–194, 1994.

[DSG95] K. A. DeJong, W. Spears, and D. Gordon. Using markov chains to analyze gafos. In L. D. Whitley and M. D. Vose, editors, *Foundations of Genetic Algorithms 3*, San Francisco, CA, 1995. Morgan Kaufmann.

[Dur64] R. Durstenfeld. Algorithm 235: Random permutation. *Communications of the Association for Computing Machinery*, 7:420, 1964.

[ECW92] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, 1992.

[EM92] C. J. G. Evertsz and B. B. Mandelbrot. *Multifractal Measures*, pages 921–953. Springer-Verlag, 1992.

[Fay91] U. M. Fayyad. *On the induction of decision trees for multiple concept learning.* PhD thesis, Ann Arbor, MI, 1991.

[FC89] R. Fung and K. C. Chang. Weighting and integrating evidence for stochastic simulation in Bayesian networks. In *Uncertainty in Artificial Intelligence 5*, pages 209–219, 1989.

[FF94] R. Fung and B. D. Favero. Backward simulation in Bayesian networks. In *Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence*, pages 227–234, San Francisco, CA, 1994. Morgan Kaufmann Publishers.

[FI93] U. M. Fayyad and K. B. Irani. Multiinterval discretization of continuous-valued attributes for classification learning. In *Proc. of the 13th International Joint Conference on Artificial Intelligence IJCAi-93*, 1993.

[Fin98] E. Fink. How to solve it automatically: Selection among problem solving methods. In R. G. Simmons, M. M. Veloso, and S. Smith, editors, *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 128–136, 1998.

[FM93a] S. Forrest and M. Mitchell. Relative building-block fitness and the building-block hypothesis. In L. D. Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 109–126, San Mateo, CA, 1993. Morgan Kaufmann.

[FM93b] S. Forrest and M. Mitchell. What makes a problem hard for a GA : Some anomalous results and their explanation. *Machine Learning*, 13(2-3):285–319, 1993.

[Fre75] M. L. Fredman. On computing the length of longest increasing subsequences. *Discrete Math.*, 11:29–35, 1975.

[FS96] Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In *International Conference on Machine Learning*, pages 148–156, 1996.

[GDH92] D. E. Goldberg, K. Deb, and J. Horn. Massive multimodality, deception and genetic algorithms. In R. Manner and B. Manderick, editors, *Parallel Problem Solving from Nature 2*, pages 37–46. North Holland, 1992.

[GG84] S. Geman and D. Geman. Stochastic relaxation, Gibbs distribution and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741, 1984.

[GGM+97] I. P. Gent, S. A. Grant, E. MacIntyre, P. Prosser, P. Shaw, B. M. Smith, and T. Walsh. How not to do it. Technical Report Report 97.27, 1997.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NPCompleteness.* Freeman, 1979.

[GLW93] F. Glover, E. Taillardand M. Laguna, and D. Werra. *Tabu Search*, volume 41. 1993.

[Gol87] D. E. Goldberg. *Simple GAs and the minimal deceptive problem*, pages 74–88. Pitman, London, 1987.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

[Gol93] D. E. Goldberg. Making genetic algorithms fly: a lesson from the wright brothers. 2:1–8, February 1993.

[Gol02] D. E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms.* Kluwer Academic Publishers, 2002.

[Gre93] J. J. Grefenstette. Deception considered harmful. In L. D. Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 75–91, San Mateo, CA, 1993. Morgan Kaufmann.

[GRS96] W. Gilks, S. Richardson, and D. Spiegelhalter. *Markov chain Monte Carlo in practice.* Chapman and Hall, 1996.

[GS87] D. E. Goldberg and P. Segrest. Finite markov chain analysis of genetic algorithms. In *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*, pages 28–31, Cambridge, MA, 1987.

[GS97] C. P. Gomes and B. Selman. Algorithm portfolio design: Theory vs. practice. In *Uncertainty in Artificial Intelligence: Proceedings of the Thirteenth Conference (UAI-1997)*, pages 190–197, San Francisco, CA, 1997. Morgan Kaufmann Publishers.

[Guo02] H. Guo. A survey of algorithms for real-time Bayesian network inference. In H. Guo, E. Horvitz, W. H. Hsu, and E. Santos, editors, *AAAI/KDD/UAI-2002 Joint Workshop on Real-Time Decision Support and Diagnosis Systems*, Edmonton, Alberta, Canada, 2002.

[Hal99] M. A. Hall. *Correlation-based Feature Subset Selection for Machine Learning.* PhD thesis, 1999.

[Har01] D. Harte. *Multifractals: Theory and Applications.* Chapman and Hall CRC, 2001.

[HCR$^+$00] E. N. Houstis, A. C. Catlin, J. R. Rice, V. S. Verykios, N. Ramakrishnan, and C.E. Houstis. PYTHIA-II: a knowledge/database system for managing performance data and recommending scientific software. *TOMS*, 26(2):227–253, 2000.

[HD96] C. Huang and A. Darwiche. Inference in belief networks: A procedural guide. *Intl. J. Approximate Reasoning*, 15:225–263, 1996.

[Hec96] D. A. Heckerman. A tutorial on learning with Bayesian networks. Technical Report 95–06, Microsoft Research, 1996.

[Hen88] M. Henrion. Propagating uncertainty in Bayesian networks by probabilistic logic sampling. In J. Lemmer and L. Kanal, editors, *Uncertainty in Artificial Intelligence 2*, pages 149–163, 1988.

[HG95] J. Horn and D. E. Goldberg. Genetic algorithm difficulty and the modality of fitness landscapes. In L. D. Whitley and M. D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 243–269, San Francisco, CA, 1995. Morgan Kaufmann.

[HK95] E. Horvitz and A. Klein. Reasoning, metareasoning, and mathematical truth: Studies of theorem proving under limited resources. In *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence (UAI–95)*, pages 306–314, San Francisco, CA, 1995. Morgan Kaufmann Publishers.

[Hoa61] C. A. R. Hoare. Algorithm 64: Quicksort. *Comm. ACM*, 4(7):321, June 1961.

[Hol75] J. H. Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, Ann Arbor, MI, USA, 1975.

[Hoo94] J. N. Hooker. Needed: an empirical science of algorithms. *Operations Research*, 42(2):201–212, 1994.

[Hor90] E. Horvitz. *Computation and Action Under Bounded Resources.* PhD thesis, 1990.

[HRG$^+$01] E. Horvitz, Y. Ruan, C. Gomes, H. Kautz, B. Selman, and D. M. Chickering. A Bayesian approach to tackling hard computational problems.

In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, August 2001.

[Hro01] J. Hromkovic. *Algorithmics for Hard Problems.* springer, 2001.

[HS01] S. Homer and A. L. Selman. *Computability and Complexity Theory.* Springer Verlag New York, 2001.

[Hsu02] W. H. Hsu. *Control of Inductive Bias in Supervised Learning using Evolutionary Computation: A Wrapper-Based Approach.* 2002.

[Hut01] H. Huttel. On Rice's theorem, 2001.

[HZ95] W. H. Hsu and A. E. Zwarico. Automatic synthesis of compression techniques for heterogeneous files. *Software: Practice and Experience*, 25(10):1097–1116, 1995.

[IBM02] IBM. High resolution time stamp facility, 2002.

[IC02] J. S. Ide and F. G. Cozman. Random generation of bayesian networks. In *Brazilian Symposium on Artificial Intelligence*, Recife, Pernambuco, Brazil, 2002.

[JF95] T. Jones and S. Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In L. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 184–192, San Francisco, CA, 1995. Morgan Kaufmann.

[JL95] G. H. John and P. Langley. Estimating continuous distributions in Bayesian classifiers. In *UAI95*, pages 338–345, 1995.

[JN96] N. Jitnah and A. E. Nicholson. Belief network algorithms: A study of performance based on domain characterization. In *PRICAI Workshops*, pages 168–187, 1996.

[Joh90] D. S. Johnson. A catalog of complexity classes. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 69–161. 1990.

[Joh00] D. S. Johnson. Challenges for theoretical computer science. Technical report, 2000.

[Joh02] D. Johnson. A theoretician's guide to the experimental analysis of algorithms. In M. H. Goldwasser, D. S. Johnson, and C. C. McGeoch, editors, *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, pages 215–250. 2002.

[KD99] K. Kask and R. Dechter. Stochastic local search for Bayesian networks. In *International Workshop on Artificial Intelligence and Statistics*, 1999.

[Kes01] M. Kessebhmer. Large deviation for weak Gibbs measures and multifractal spectra. *Nonlinearity*, 2001.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 1983.

[KHR$^+$02] H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and Bart Selman. Dynamic restart policies. In *Proceedings AAAI-2002*, 2002.

[KJ97] R. Kohavi and G. John. Wrappers for feature subset selection. *Artificial Intelligence journal, special issue on relevance*, 97(1-2):273–324, 1997.

[KNR01] L. Kallel, B. Naudts, and C. R. Reeves. Properties of fitness functions and search landscapes. In L. Kallel, B. Naudts, and A. Rogers, editors, *Theoretical Aspects of Evolutionary Computing*, pages 175–206. Springer, Berlin, 2001.

[Knu81] D. E. Knuth. *The art of computer programming: Sorting and Searching*, volume 3. Addison-Wesley, 1981.

[Kol65] A. K. Kolmogorov. Three approaches to the quantitative definition of information. 1:1–7, 1965.

[KP83] J. H. Kim and J. Pearl. A computational model for combined causal and diagnostic reasoning in inference systems. In *Proceedings of IJCAI-83*, pages 190–193, Karlsruhe, Germany, 1983.

[KP98] R. Kohavi and F. Provost. Glossary of terms. *Machine Learning*, 30:271–274, 1998.

[Kra82] H. C. Kraemer. *Kappa coefficient.* John Wiley & Sons, New York, 1982.

[Lee90] J. V. Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity.* Elsevier and MIT Press, 1990.

[Lev86] L. Levin. Lecture botes on fundamentals of computing, 1986.

[LL00] M. G. Lagoudakis and M. L. Littman. Algorithm selection using reinforcement learning. In *Proc. 17th International Conf. on Machine Learning*, pages 511–518. Morgan Kaufmann, San Francisco, CA, 2000.

[LL01] M. G. Lagoudakis and M. L. Littman. Selecting the right algorithm. In *Proceedings of the 2001 AAAI Fall Symposium Series: Using Uncertainty within Computation*, Boston, MA, 2001.

[LS88] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert system. *Royal Statistic Society*, 50:154–227, 1988.

[LV90] M. Li and P. Vitanyi. Kolmogorov complexity and its applications. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 189–254. 1990.

[LV93] M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer-Verlag, New York, 1993.

[Mac98] D. MacKay. Introduction to monte carlo methods. In M. Jordan, editor, *Learning in graphical models*. The MIT Press, Cambridge, Massachusetts, 1998.

[Man72] B. B. Mandelbrot. *Possible refinement of the lognormal hypothesis concerning the distribution of energy dissipation in intermittent turbulence*, pages 331–351. Springer, NY, 1972.

[Man82] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W.H. Freeman and Co., NY, 1982.

[Man85] H. Mannila. *Instance Complexity for Sorting and NP-complete problems*. PhD thesis, Department of Computer Science, University of Helsiki, 1985.

[Man89] B. B. Mandelbrot. Multifractal measures, especially for geophysicists. *Pageopg*, 131(133), 1989.

[McG92] C. C. McGeoch. Analyzing algorithms by simulation: Variance reduction techniques and simulation speedups. *ACM Computing Surveys*, 24:195–212, 1992.

[McG02] C. C. McGeoch. Experimental analysis of algorithms. In P. Pardalos and E. Romeijn, editors, *Handbook of Global Optimization, Volume 2: Heuristic Approaches*. Kluwer Academic Publishers, 2002.

[Men99] O. J. Mengshoel. *Efficient Bayesian Network Inference: Genetic Algorithms, Stochastic Local Search, and Abstraction*. PhD thesis, 1999.

[MFH91] M. Mitchell, S. Forrest, and J. H. Holland. The royal road for GAs: Fitness landscapes and GA performance. In *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*. MIT Press, 1991.

[Min96] S. Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints*, 1(1/2):7–43, 1996.

[Mit97] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[MM00] G. Melancon and M. B. Melou. Random generation of DAGs for graph drawing. Technical Report INS-R0005, Dutch research center for Mathematical and computer science, 2000.

[Mor00] B. Moret. Towards a discipline of experimental algorithmics. In *5th DIMACS Challenge*, DIMACS Monograph Series, 2000.

[MSF$^+$02] C. C. MaGeoch, P. Sanders, R. Fleischer, P. Cohen, and D. Precup. Searching for big-oh in the data: Inferring asymptotic complexity from experiments. In *Lecture Notes in Computer Science: Proceedings of the Dagstuhl Seminar on Experimental Algorithmics*. Springer-Verlag, 2002.

[Nau98] B. Naudts. *Measuring GA-hardness*. PhD thesis, Antwerpen, Netherlands, 1998.

[Nea90] R. E. Neapolitan. *Probabilistic Reasoning in Expert Systems: Theory and Algorithms*. John Wiley and Sons, New York, 1990.

[NK98] B. Naudts and L. Kallel. Some facts about so called GA-hardness measures. Technical Report 379, Centre de Mathématiques Appliquées, Palaiseau, 1998.

[NK00] B. Naudts and L. Kallel. A comparison of predictive measures of problem difficulty in evolutionary algorithms. *IEEE-EC*, 4(1):1, April 2000.

[NV92] A. Nix and M. D. Vose. Modelling genetic algorithms with markov chains. In *Annals of Mathematics and Artificial Intelligence 5*, pages 79–88, 1992.

[NW78] A. Nijenhuis and H. S. Wilf. *Combinatorial Algorithms for Computers and Calculators*. ACADEMIC PRESS, 1978.

[OKSW94] P. Orponen, K. Ko, U. Schoning, and O. Watanabe. Instance complexity. *Journal of the ACM*, 41(1):96–121, 1994.

[PA02] P. Pakzad and V. Anantharam. Belief propagation and statistical physics. In *Conference on Information Science and Systems*, 2002.

[Pap94] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[Par02a] J. Park. MAP complexity results and approximation methods. In *Proceedings of the 18th Annual Conference on Uncertainty in AI (UAI)*, pages 388–396, 2002.

[Par02b] J. Park. Using weighted MAX-SAT engines to solve MPE. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI)*, pages 682–687, 2002.

[Pea86] J. Pearl. Fusion, propagation, and structuring in belief networks. *Artificial Intelligence*, 29:241–248, 1986.

[Pea87] J. Pearl. Evidential reasoning using stochastic simulation of causal models. *Artificial Intelligence*, 32(2):245–257, 1987.

[Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan-Kaufmann, San Mateo, CA, 1988.

[Qui86] J. R. Quinlan. Induction on decision trees. *Machine Learning*, 1:81–106, 1986.

[Qui93] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[Raw92] G. J. E. Rawlins. *Compared to What?* W H Freeman, 1992.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 89:25–59, 1953.

[Ric76] J. R. Rice. The algorithm selection problem. In M. V. Zelkowitz, editor, *Advances in computers*, volume 15, pages 65–118. 1976.

[Rie99] R. Riedi. An introduction to multifractals. Technical report, Rice University, 1999.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[RU01] R. L. Rardin and R. Uzsoy. Experimental evaluation of heuristic optimization algorithms: A tutorial. *Journal of Heuristics*, 7:261–304, 2001.

[RV97] R. Riedi and J. L. Vehel. Multifractal properties of tcp traffic: A numerical study. Technical report, Rice University, 1997.

[RW99a] S. Rana and L. D. Whitley. Search, binary representations and counting optima. In L. D. Davis, K. De Jong, M. D. Vose, and L. D. Whitley, editors, *Evolutionary Algorithms*, pages 177–189. Springer, New York, 1999.

[RW99b] C. R. Reeves and C. C. Wright. *Genetic Algorithms and the Design of Experiments*, pages 207–226. Springer, New York, 1999.

[Ryl01] B. Rylander. *Computational Complexity and the Genetic Algorithm.* PhD thesis, 2001.

[San91] E. Santos. On the generation of alternative explanations with implications for belief revision. In *UAI91*, 1991.

[San00] M. Santini. *Random generation and approximate counting of combinatorial structures.* PhD thesis, 2000.

[SC99] S. E. Shimony and E. Charniak. A new algorithm for finding MAP assignments to belief network. In *UAI99*, 1999.

[SD02] S. E. Shimony and C. Domshlak. Complexity of probabilistic reasoning in singly connected (not polytree!) Bayes networks. *submitted for publication*, 2002.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27, 1948.

[Shi94] S. E. Shimony. Finding MAPs for belief networks is NP-hard. *Artificial Intelligence*, 68:399–410, 1994.

[Sin93] A. Sinclair. *Algorithms for Random Generation and Counting: A Markov Chain Approach.* Birkhauser, 1993.

[SMH$^+$91] M. A. Shwe, B. Middleton, D. E. Heckerman, M. Henrion, E. J. Horvitz, and H.P. Lehmann. Probabilistic diagnosis using a reformulation of the INTERNIST-1/QMR knowledge base: I. the probabilistic model and inference algorithms. *Methods of Information in Medicine*, 30(4):241–255, 1991.

[Sol64] R. Solomonoff. A formal theory of inductive inference. *Information and Control*, 7:1–22, 1964.

[SP89] R. D. Shachter and M. A. Peot. Simulation approaches to general probabilistic inference on belief networks. In *Uncertainty in Artificial Intelligence 5*, volume 5, pages 221–231, New York, 1989. Elsevier Science Publishing Company.

[SSW95] E. Santos, S. E. Shimony, and E. Williams. On a distributed anytime architecture for probabilistic reasoning. Technical Report AFIT/EN/TR94-06, Department of Electrical and Computer Engineering, Air Force Institute of Technology, 1995.

[SW86] D. Stanton and S. White. *Constructive Combinatorics.* Springer-Verlag, 1986.

[Tay98] R. G. Taylor. *Models of Computation and Formal Language.* Oxford University Press, 1998.

[Tur36] A. Turing. On computable numbers, with an application to the Entscheidungs problem. In *Proc. Lond. Math. Soc.*, volume 2, pages 230–365, 1936.

[Vos93] M. D. Vose. Modelling simple genetic algorithms. In *Foundations of Genetic Algorithms 2*. Morgan Kaufmann, 1993.

[Wei99] M. A. Weiss. *Data Structures and Algorithms Analysis in JAVA*. Addison-Wesley, 1999.

[WF99] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, 1999.

[Whi90] L. D. Whitley. Fundamental principles of deception in genetic algorithms. In G. Rawlins, editor, *Foundations of Genetic Algorithms 1*, pages 221–241, 1990.

[Whi93] L. D. Whitley. An executable model of a simple genetic algorithm. In *Foundations of Genetic Algorithms 2*, pages 45–62. Morgan Kaufmann, 1993.

[Wil64] J. W. J Williams. Algorithm 232: Heapsort. *Comm. ACM*, 7(6):347–348, June 1964.

[Wil97] E. M. Williams. *Modelling Intelligent Control of Distributed Cooperative Inferencing.* PhD thesis, 1997.

[WM95] D. H. Wolpert and W. G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe, NM, 1995.

[WM97] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.

[Wol92] D. H. Wolpert. Stacked generalization. *Neural Networks*, 5:241–259, 1992.

[YFW00] J. Yedidia, W. Freeman, and Y. Weiss. Bethe free energy, Kikuchi approximations, and belief propagation algorithms. Technical Report 2001-16, MERL, 2000.

[Zil93] S. Zilberstein. *Operational Rationality through Compilation of Anytime Algorithms*. PhD thesis, 1993.