INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa • University of Ottawa

A New Search Technique for Mining Relationship Rules

William Elazmeh

Thesis submitted to the School of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Masters of Computer Science

The Ottawa-Carleton Institute for Computer Science
University of Ottawa

Abstract

Data Mining is extracting knowledge from data. Knowledge may be presented using rules which express relationships between sets of items involved in the rule. The nature and strength of the relationship is determined by an evaluation measure used to evaluate the rules. A relationship may correspond to an association, an implication, a correlation, a causality, a dependency, etc. This thesis considers the problem of finding the strongest N rules from transaction data. Existing algorithms (e.g. Apriori) employ search techniques based on Best-First Search methods which impose high demands on memory. Searching Depth-First solves this memory problem but can demand long execution time. This drawback can be avoided by using iterative deepening search where the Depth-First Search is run with a progressively relaxed search bound on each successive iteration. Relaxing the search bound can be based on various characteristics of the search space, such as a change in the tree depth (MSDD algorithm), or a change in the quality of states being explored. The key issue is the strategy by which the search bound is relaxed for subsequent iterations. We propose an iterative deepening search algorithm $ID_{G_{max}}$ using a search bound strategy based on the quality of states explored. We explore different methods for relaxing the search bound to improve the performance. We present experimental results to evaluate the performance of these search bound relaxation strategies and compare them to the MSDD algorithm. We also show that the choice of the search bound relaxation method can significantly influence the performance of the rule mining algorithm in both memory and time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction & Organization

In recent years, the flood of information has consistently been increasing. Many activities are being recorded in numerous databases. The volume of such data is rising sharply. A satellite orbiting the earth sends Gigabytes of information for every minute event it observes. Such an overflow of transaction data presents researchers with more difficult challenges. The first of such challenges is storage. Where can we store such a large volume of data? The answer to this question has been demonstrated by a significant improvement in the development of advanced storage devices.

Once data is stored, then what do we do with it? More specifically, how do we organize and retrieve information from the data? The answer probably lies in the evolution of databases. Databases are usually queried to retrieve information they contain, or to compute information based on transactions stored in the database. However, the growth in the size of these transaction databases continues. At some point, available storage devices will reach their limits and will no longer be able to accommodate the large volume of data. A solution to this problem may be the following; Instead of storing the data, we may attempt to extract useful knowledge

from it. Organizations have been interested in such knowledge of real life patterns in search of a deeper understanding of trends by which events occur. This idea brings us to yet a new question; how can we understand knowledge buried among transactions. This thesis presents a newly designed rule mining algorithm, $ID_{Ci_{max}}$, to extract relationships among sets of items from transaction datasets. Moreover, the experiments in this thesis evaluate such search algorithms and present new challenges.

1.1 Problem Statement & Motivations

Databases record information about events in the form of transactions. A collection of such transactions may contain patterns in which events occur. These patterns may be expressed as strong confident rules that frequently hold true. A relationship rule indicates the existence of a relationship between two sets of items. These relationships may be of different types. For example, consider the two sets of events, $S_1 = \{$ *Oil Price Increases, Value of $CDN Decreases* $\}$ and $S_2 = \{$ *Fuel Price Increases* $\}$. A relationship rule involving S_1 and S_2 suggests the existence of a relationship between S_1 and S_2, thus indicates a relationship between rising oil prices, decreasing value of the Canadian Dollar, and increasing fuel prices. Extracting such rules from databases can help organizations to:

- discover previously unknown trends.
- verify that desired policies have, indeed, been implemented in practice.
- attempt to predict what events will take place based on previously observed relationship rules.

Recently, extensive research has been conducted in the area of extracting knowledge from databases [33, 15]. In this thesis we attempt to address the problem of extracting knowledge from a volume of transactions. This knowledge can be expressed in many forms. One such form is a relationship rule from which we may conclude a relationship between events involved in data transactions. Mining relationship rules involves searching the space of all possible rules among all combinations of event-sets present in the transaction data. A user-supplied evaluation measure of interest dictates the type and strength of the relationship expressed by the rules. The strength of the relationship is used to rank these rules.

This problem requires the exploration of a large search space of an exponential size. Depending on the nature of the desired rules, the problem may become complex. For instance, mining optimized rules with constraints from data is NP-hard [10, 28]. The search space may easily suffer from a state space explosion which causes the search algorithm to suffer with respect to two main constraints. The first is CPU execution time, and the second is memory requirements. Most existing algorithms may attempt to optimize one or the other constraint. In fact, the most commonly used algorithm, Apriori [5], is incapable of offering any guarantees against state space explosions.

The MSDD algorithm [31] has been improved to minimize the memory requirements by using the iterative deepening search technique. MSDD uses the depth of a state in the search tree as the main search bound strategy for iterative deepening which may provide an unreliable assistance when searching for a set of desired states. This unreliability is based on the fact that desired states may appear at various depth levels of the search tree regardless of their desired quality. Thus, if the search is guided to

generate and explore portions of the search tree which contain very few (or none) desired states, then, the search may spend an extensive period of time exploring undesired states before it reaches desired states, which may be found very deep in the search tree.

Our main motivation, in this thesis, is to minimize the memory requirements of a relationship rule mining algorithm without compromising the execution time. Therefore, we propose a new rule mining algorithm $ID_{G_{max}}$ which employs the iterative deepening search technique. However, $ID_{G_{max}}$ uses search bound strategies based on the desired quality of states in the search tree. We present and compare three different search bound strategies (Conservative, Dynamic, and Exhaustive strategies) against the MSDD search algorithm. We discuss our experimental results which show a significant speed up in CPU execution time over the MSDD search algorithm.

1.2 Contributions

Our research focuses on developing a new search algorithm which heuristically explores portions of a state-tree search space. This thesis makes several contributions listed below:

1. This thesis puts forward a novel search algorithm designed to explore tree-based search spaces, $ID_{G_{max}}$. The algorithm is independent of the state evaluation function (the state evaluation function is an input to the algorithm). In our experiments, the $ID_{G_{max}}$ search algorithm uses the G rule evaluation measure to express dependencies among sets of items. The G measure was introduced and used for a similar application in [30].

2. $ID_{G_{max}}$ is shown to be significantly faster than most existing state exploration algorithms, in particular its closest rival, MSDD.

3. The idea of depth-first iterative deepening search is a classic search technique introduced in [21]. The $ID_{G_{max}}$ search algorithm uses iterative deepening and depth-first search to reduce its memory requirements. The search is guided by using a heuristic estimation of a state quality measure as the main search bound strategy employed by iterative deepening. However, the execution time is reduced by a new systematic method of relaxing the search bound. Our experiments compare three such methods of relaxing the search bound for iterative deepening in search of a conclusion as to which method is most appropriate for better performance.

4. Finally, this thesis reviews existing work relating to rule mining algorithms from transaction data and discusses possible interpretations of discovered rules based on their evaluation measures. This thesis also reviews the work related to state space exploration techniques. Furthermore, this review presents a possible implementation of an existing search technique, Depth-First Branch and Bound, customized to our search problem. However, we have not evaluated this possible implementation.

1.3 The Organization of this Thesis

In chapter 2, we provide a brief background in the area of data mining, in particular, the area of mining association rules which are a form of rules that express an association relationship among item-sets involved. The

chapter also reviews related work for generating and evaluating relationship rules. Finally, the chapter presents several measures which can be used to evaluate relationship rules. In chapter 3, we present some related work in the area of search space exploration techniques. The chapter defines several criteria of evaluation to use when comparing search algorithms. The chapter also reviews generic and domain specific methods which attempt to reach for a set of desired states in a search space. In chapter 4, we present our newly developed search algorithm which searches for the strongest N states in a tree-based search space, followed by a discussion of our design methods and experimental approach when evaluating its performance. In chapter 5, we compare $ID_{G_{max}}$ search algorithm (using the three different search bound strategies) to our implementation of the existing MSDD search algorithm. In chapter 6, we present a summary of our research, conclusions based on the experimental results, and a list of possible future work ideas.

Chapter 2

Background and Basics

This chapter provides the basic knowledge and the background assumed in this thesis work. The chapter also presents an overview of related research conducted in the field of mining association rules from data. The chapter begins by introducing relationship rules, then, presents a discussion of existing methods for generating, evaluating, and interpreting extracted rules. Finally, a short chapter summary concludes with a brief review.

2.1 What are "Relationship Rules"

The increase of information flow called for the creation of databases. By definition, databases hold a generous volume of transactions recording many related events. In [33], *Knowledge Discovery* is defined as "the non-trivial extraction of implicit, previously unknown, and potentially useful information from data". Useful knowledge can be extracted from databases in many forms. One such form is a rule which expresses a relationship to reflect a trend or a pattern. For example, a store manager wants to know customers' buying habits, or what we call "patterns". The store manager

can use such patterns to improve the process of stocking products on the store shelves. Hypothetical examples of such patterns are: customers who buy chocolate bars tend to also buy cigarettes; or the customers who buy cigarettes and milk also buy motorcycle magazines. These patterns, when extracted from store transaction data, indicate a relationship among the events of buying cigarettes, buying milk, and buying motorcycle magazines.

2.1.1 Relationship Rules - A Formal Definition

Given a set of transaction data, where each transaction contains a subset of items from the overall set of items I, a relationship rule is an expression of the form $X \Rightarrow Y$, where X and $Y \subseteq I$. The rule suggests the existence of a relationship between the sets of items X and Y. The interpretation of this relationship is subject to the evaluation measure used to rank these extracted rules. For instance, an association rule expresses an association relationship among itemsets involved which may be interpreted as the sets of objects or events that "occur together". Thus, a relationship rule is a rule of the following form:

$$X_1, X_2, \cdots, X_n \Rightarrow Y_1, Y_2, \cdots, Y_m$$

where $X_i \in I$, $Y_j \in I$ and $X \cap Y = \phi$.

For example, an aircraft manufacturer runs a series of test flights on a recently built aircraft. The aircraft is flown R number of flights. An on-board computer monitors several functionalities of different sub-systems of the aircraft. For each of the flights R_i and for a particular list of sub-systems I, the on-board computer generates a record indicating whether or not a sub-system reports a failure message during that flight, thus, generating a database of R records. This database is then searched to discover

the strongest rules that relate the failing sub-systems. In other words, the discovery process yields the strongest patterns (expressed as relationship rules) of failure messages being reported among the sub-systems of the aircraft. Such rules are useful for many purposes, eg. diagnostics and verification. To illustrate the idea of relationship rules, consider the list of sub-systems of an aircraft shown in Table 2.1.

Bit Number	Unit Description
1	AUTO FLIGHT
2	COMMUNICATIONS
3	ELECTRICAL POWER
4	FLIGHT CONTROLS
5	HYDRAULIC CONTROLS
6	ICE AND RAIN
7	LANDING GEAR
8	NAVIGATION
9	ENGINE FUEL CONTROL
10	ENGINE CONTROLS

Table 2.1: A List of SubSystem Units of an Aircraft

A possible relationship rule may be:

$$\text{ICE AND RAIN} \Rightarrow \text{LANDING GEAR}$$

The above rule indicates the existence of a relationship between the events of a failure being reported by the *ICE AND RAIN* unit and by the *LANDING GEAR* unit. Such a rule can be used for either, verifying the functionality of units, or for diagnostics to determine whether the failure of some units is related to the failure of any other units. In [2], The itemset $X = \{$*ICE AND RAIN*$\}$ is called the *antecendent* of the rule, and the itemset $Y = \{$*LANDING GEAR*$\}$ is called the *consequent* of the rule. Having introduced the problem

of "mining" rules from a database in [2] (in this case association rules), it is now possible for a database to process the following queries:

- Given a set of items I, and a measure of interest G, find the N highest G-scoring relationship rules. These rules can help the discovery of sets of items involved in a relationship specified and measured by G.
- Find all the rules that have a specific item (or more) $i \in I$ as a consequent of a rule. Such rules can help decision makers to discover what set of items they can require to accomplish a relationship with the particular itemset $\subset I$.
- Find all the rules that have a specific item (or more) $i \in I$ as a antecendent of a rule. Such rules can help decision makers to discover what set of items they can acquire form a relationship with the specified itemset $\subset I$.
- Given two different itemsets S_1 and $S_2 \subset I$, find all rules relating the two sets. These rules can help planning events involving the two itemsets S_1 and S_2.

Applied to the test flights example, the process of mining the set of all records collected from the flights results in discovering many relationship rules. Therefore, it is now possible, for instance, to query the set of all possible rules to find:

- the strongest N relationship rules involving failures reported by different sets of sub-system units.
- all the rules that have a failure reported by the unit *ENGINE FUEL CONTROLS* as a consequent of a rule. Such rules can help the discovery

of what set of units are required to fail in relation to a failure reported by the *ENGINE FUEL CONTROL* unit.

- all the rules that have a failure reported by the unit *ELECTRICAL POWER* as a antecendent of a rule. Such rules can help the discovery of what set of units are acquired to report a failure related to the failure of the *ELECTRICAL POWER* unit.

- all the rules that indicate a relationship between failures reported by the sets of units $S_1 = \{$*COMMUNICATIONS, NAVIGATION*$\}$ and $S_2 = \{$*FLIGHT CONTROLS, ENGINE CONTROLS, LANDING GEAR*$\}$. These rules can suggest possible relationships between failures reported by subsystem units in the two sets of units.

Obviously, the set of all possible rules between all possible subsets of items can be very large. The work in this thesis shows the development of an efficient algorithm to explore the space of all possible rules to discover the strongest N relationship rules, i.e. the set of N rules which have the highest desired quality.

2.1.2 Binary Representation of Relationship Rules

Let $I = I_1, I_2, \cdots, I_m$ represent a set of binary attributes, called the *Items*. Let T represent a set of transactions, called the *Database*. Each transaction t is represented by a binary vector with $t[k] = 1$ if the item I_k is present in the transaction t, and $t[k] = 0$ otherwise. For each transaction recorded in the database, there exists a tuple indicating the presence or absence of items in the transaction. For example, consider the test flights example, shown in the previous section. A record of a test flight where the units *COMMUNICATIONS, NAVIGATIONS,* and *HYDRAULIC CONTROLS* report one or

more failure messages has the corresponding transaction represented by the tuple: (the binary bits from left to right are shown in table 2.1)

$$< 0, 1, 0, 0, 1, 0, 0, 1, 0, 0 >$$

For a relationship rule of the form $X \Rightarrow Y$, where X and Y are sets of items in the itemset I, and $X \cap Y = \phi$, the sets X and Y are each represented as a binary tuple (vector) very similar to that of a transaction. In this representation, the presence of an item I_k in the set X is indicated by $X[k] = 1$ and $X[k] = *$ indicates the indifference of the presence or absence of item I_k in the rule. In other words, a symbol $*$ indicates a "do not care" value. For example, the relationship rule:

HYDRAULIC CONTROLS, ENGINE CONTROLS

$\Rightarrow$

LANDING GEAR, AUTO FLIGHT

has the following binary representation:

$$< *, *, *, *, 1, *, *, *, *, 1 > \Rightarrow < 1, *, *, *, *, *, 1, *, *, * >$$

Thus, we use a single bit to represent information about a single item or attribute. Using the "do not care" entry in the binary representation of rules restricts the rule to positive interpretation only which eliminates rules involving items not present in the itemsets.

2.1.3 Positive Relationship Rules

Relationship rules, of the form $X \Rightarrow Y$, suggest the existence of a relationship among items present in X and Y. Thus, the presence of items in the set X is related to the presence of items in the set Y. A negative relationship rule indicates that the relationship involves the presence

and/or absence of items from either sets X or Y. Using an appropriate rule evaluation measure [12], such rules can express a negative nature. This means that the presence of some items in either of the sets X or Y may be related to the absence of items in the other set. This thesis work considers only positive relationship rules because the absence of an item from the data transaction is simply inconclusive as to whether or not it may appear in future transactions. Therefore, we consider the absence of an item as insufficient information. Consequently, our binary representation is incapable of representing the absence of items. However, for our algorithm, the evaluation test used to evaluate the rules can easily be modified to include positive and negative relationship rules without affecting the execution of the search algorithm.

2.1.4 Representation of Multi-Valued Attributes

Some data transactions may consist of attributes which have multiple values. We can still use the binary representation to represent these attributes, however, we need to accommodate multiple values. Our solution for this problem is to simply use more binary bits to represent each of the values. In other words, to account for more values per attribute, we can use a single binary bit to represent the presence or absence of each of the values for every attribute. The obvious consequence of this approach is a significant increase in the length of the representation of transaction records which leads to an increase in the number of bits used to represent rules and transactions. However, this increase does not affect the size of the search space since all different values for each of the attributes must be used when generating all possible rules between itemsets.

2.2 A Brief History of Association Rules Research

In this section, we present some of the advances in rule mining research. Data analysis is an important task required for decision making. Organizations use the analysis of transaction data as an important tool to improve the quality of their decision making processes. However, until recently, data analyses have mostly depended on statistics based on cumulative data, eg. cumulative sales for a period of time. The introduction of computerized systems to perform business tasks, such as tracking inventory, allowed organizations to collect and store inventory, sales, and business transactions information. In [2], such data is described as a *collection of basket data type transactions*. Collecting such transaction data resulted in a massive volume of data which lead to the use of database systems to organize and analyze such data.

Finding an association, a dependence, or a correlation relationship between two or more itemsets is a significant task many researchers and decision makers are interested in. In [2], Agrawal made progress towards improving database functionalities by introducing the problem of mining association rules from transaction data. He defined the *problem of mining a large collection of basket data type transactions for association rules between itemsets* . Several algorithms have been developed to extract, or "mine", association rules from data. The following is a categorization of such algorithms, presented in [10].

The first category is the algorithms that mine association rules which satisfy a set of hard constraints, such as *confidence & Support* [2, 3, 9]. The second category of algorithms uses a heuristic approach in mining association rules. The concept is to find predictive association rules without

making any promises on the predictiveness or the completeness of the resulting rules. Such algorithms are found in [14, 26]. The third category is the algorithms that discover the most interesting association rules. Such algorithms require a measure of "interestingness" to evaluate the quality of association rules as defined by some interestingness metric, [16, 19, 34]. However, measuring the "interestingness" or the "goodness" of a rule is not a simple task. Many measures of such quality exist and have been successfully used, some are, *Confidence & Support* [2], gain [16], variance and Chi-squared test [28, 29], entropy gain [27, 28], gini [27], laplace [14, 42], lift [1], conviction [13], and dependency strength [30].

Agrawal *et al.*, in [2], derived a method to evaluate confident association rules from a frequent itemset. His method is based on the *Confidence* and *Support* of an association rule. In [5], Agrawal extended his association mining algorithm (Apriori) to handle a large number of transactions, known as (AprioriHybrid). The work in [20] addressed the problem of discovering a very large number of association rules by proposing a technique known as *Rule Template* which makes it possible to describe the structure of interesting rules. The work in [37] presents an attempt to improve the *Apriori* algorithm by reducing the number of passes on the dataset when computing the support for items found in the frequent itemset. The *Apriori* algorithm is discussed in more detail in the subsequent chapter.

Toivonen *et al.* [40], proposed methods for reducing the number of discovered association rules by forming a *Rule Cover*. A rule cover is defined as a subset of all possible rules such that for every rule there exists a rule applicable to it in the rule cover. In addition, this paper derived an algorithm to find a rule cover which is used to prune the set of all association rules found to eliminate redundancy. This paper , [40], also discusses the

grouping of association rules by clustering. In the same year, Srikant & Agrawal [36] brought forward hierarchy-based association rules where an association relationship between itemsets can be expressed in terms of a taxonomy (*an is-a hierarchy*). Such association rules he calls. *Generalized Association Rules.*

In [4], Agrawal *et al.* developed the *Quest Data Mining System* as a result of an on-going research on the functionality required to develop data-intensive decision support applications. In [17], the data mining research group at Simon Fraser university developed the *DBMiner* system which integrates several data mining techniques, including mining simple association rules and generalized association rules, into one system. The *DBMiner* system performs an interactive mining of multi-level knowledge in large relational databases. This system uses a variety of data mining techniques such as characterization. comparison, association, classification, prediction, and clustering. All the techniques are to be used interactively using the *On-Line Analytical Processing* technique developed by the same group. Ali *et al.* [8] used association rules to perform a *Partial Classification* task. The rule based classification approach helps process datasets when the number of attributes is large with many attributes have missing values, or when the class distribution is very skewed. They also show that association rules prove useful in understanding the low-frequency classes. Recently, in [23], Liu *et al.* report on similar work to deal with the integration of classification and association rule mining.

The interest in the quality of association rules discovered from data reached a mature level in 1997. Brin *et al.* [12] generalized association rules beyond the market baskets where **basket data** is defined in general terms as each $b_i \subseteq I$ is a **basket** of items, where $I = i_1, \cdots, i_k$ is a set of

k items and $B = b_i, \cdots . b_n$ is a set of n subsets of I. The work developed the notion of mining correlations, what they call (*generalized associations*), where both presence and absence of an item are considered for generating rules. Their research proposed using the *Chi-Squared* test for correlations to measure the statistical significance of rules. Similar work was published in [35] describing the *dependence rules* which identify statistical dependence in the presence and absence of items in the itemsets. However, in this case, the task is to discover causal structures as opposed to association rules. Megiddo *et al.* [24], studied the quality of association rules based on whether rules "*overfit*" the data or not. They addressed the problem of the discovered associations being "*False Discoveries*" which are not statistically significant. They also proposed a predictive use of association rules by computing the confidence intervals of the *Support* and *Confidence* proposed in [2].

Originating from research for structures in multiple streams of data, Oates & Cohen [30] derived an algorithm to evaluate temporal dependency structures found in streams of data. The work lead to the development of the *Multi-Stream Dependency Detection MSDD* algorithm which finds rules that express statistical dependencies between structures in multiple streams of data. In this case, the rule strength is measured by the G statistic [43]. In [31], Oates *et al.* introduced the use of iterative deepening search technique to mine such structures, an idea very similar to the work of this thesis.

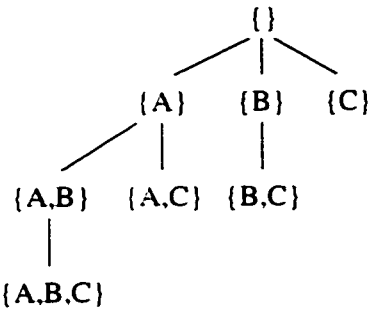

Figure 2.1: All Possible Subsets of Itemset $I = \{A, B, C\}$.

2.3 Generating the Space of All Possible Rules

Relationship rules represent the existence of a relationship between two sets of items X and Y, and has the form $X \Rightarrow Y$. Both X and Y are subsets of the itemset I. In order to explore the space of all possible relationship rules in search of a desired set of rules, we must first generate the space of all possible subsets of items, then use these generated itemsets to form the space of all possible relationship rules. In this section we show how this process of generating the space of all possible rules is simplified into a simple state generating task in a tree-based search space. For example, consider a set of three items $I = \{A, B, C\}$. Figure 2.1 shows all the possible itemsets generated from I. A data mining algorithm searches the space of all possible relationship rules between any two itemsets to find the strongest N rules. When generating the search space we avoid repeating itemsets, i.e. we consider $\{A, B\}$ the same as $\{B, A\}$. We use a binary vector representation of the itemset, where the presence of each item is represented by a binary bit = 1 or a "do not care" otherwise. The search space starts with the most general relationship rule $\{\} \Rightarrow \{\}$. We build the search tree by specializing one bit at a time, i.e. setting a

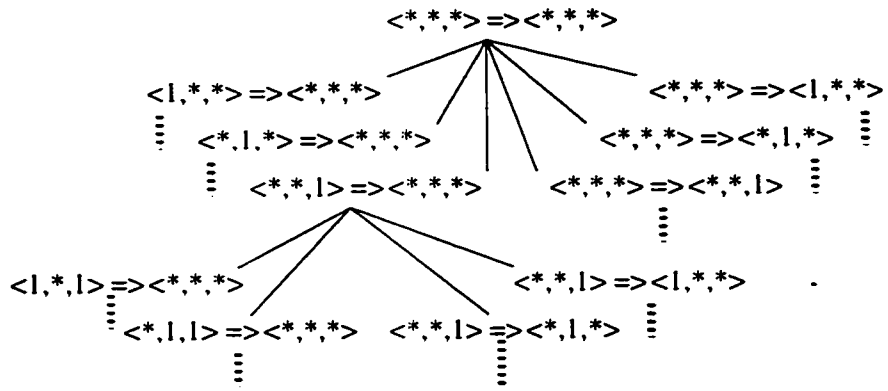

Figure 2.2: All Rules between Itemsets of $I = \{A, B, C\}$.

bit to 1. Figure 2.2 shows that a successor of a state is generated by specializing one additional bit. Therefore, at depth d each state will have d bits specialized. There are three options for a single bit to be specialized. A bit can be specialized in the LHS or in the RHS of the relationship rule, or in neither. Therefore, using the example of three bit itemsets, we start with the most general relationship rule $< *, *, * > \Rightarrow < *, *, * >$ and specialize every bit at a time, once in the LHS, then once in the RHS of the relationship rule. Figure 2.2 illustrates this generation method of all possible relationship rules between three itemsets of the set I. The size of the search space is exponential by the number of bits which represent items in the itemsets. Recall, in a relationship rule, each bit may be specialized, either in the LHS, in the RHS, or not specialized at all. Therefore, the size of the search space is 3^n, where n is the number of bits used to represent values of items from I. A general relationship rule has fewer specialized bits, on the other hand, a specialized relationship rule has more bits set to 1. Ideally, the most useful relationship rules are those that have the highest

relationship strength, and preferably, the most general LHS and have the most specific RHS. Thus, such rules require the least information and provide the most possible information with strong relationship measure.

2.4 Evaluation of Relationship Rules

Researchers have long been interested in the quality of relationship rules discovered from databases. In fact, in [10], Agrawal *et al.* describe a class of association rule mining algorithms which search for the most interesting association rules according to a quantified measure of interest. However, quantifying "interestingness" or "goodness" is not a simple task. The interpretation of "interestingness" is dictated by the test used to evaluate the relationship rule. Classical statistics in [7, 12, 25] interpret correlations and [35] interprets causality, etc. In this section, we will review some of the measures of "interestingness" used to evaluate the quality of simple relationship rules.

2.4.1 Definitions & Notations

Recall, a relationship rule is a rule of the following form:

$$X_1, X_2, \cdots, X_n \Rightarrow Y_1, Y_2, \cdots, Y_m$$

where X and Y are itemsets of I, e.g. $X \subset I$, $Y \subset I$ and $X \cap Y = \phi$. To evaluate such a relationship rule, we compute the counts in the following contingency table for the rules $X \Rightarrow Y$ from the transaction dataset:

	Y	$\bar{Y}$	$\sum row$
X	n_1	n_2	$r_1 = n_1 + n_2$
$\bar{X}$	n_3	n_4	$r_2 = n_3 + n_4$
$\sum col$	$c_1 = n_1 + n_3$	$c_2 = n_2 + n_4$	T

The entries n_1, n_2, n_3, n_4 are respectively described as; the number of data transactions containing the itemsets $X \wedge Y$, $X \wedge \bar{Y}$, $\bar{X} \wedge Y$, $\bar{X} \wedge \bar{Y}$. The totals r_1, r_2, c_1, c_2 are respectively described as the number of data transactions containing the itemsets $X, \bar{X}, Y, \bar{Y}$. T is the total number of transactions found in the dataset. From this contingency table, there are four possible forms of relationship rules, they are: $X \Rightarrow Y$, $X \Rightarrow \bar{Y}$, $\bar{X} \Rightarrow Y$, $\bar{X} \Rightarrow \bar{Y}$. These four forms are supported by n_1, n_2, n_3, n_4 records respectively. A negated itemset indicates that the absence of the itemset is involved in the relationship. for example, the relationship rule $\bar{X} \Rightarrow Y$ represents the relationship between the absence of itemset X and the presence of itemset Y. Our work is focused on the first of these forms of relationship rules which have the form $presence(X) \Rightarrow presence(Y)$.

2.4.2 Support & Confidence Evaluate Association Rules

The first evaluation method measures the frequency with which itemsets X and Y occur together. This measure indicates an association relationship between itemsets involved in the relationship rule. More precisely, we measure how often an association occurs between the itemsets X and Y. Moreover, when an association rule of the form $X \Rightarrow Y$ occurs, we measure how confident such an association is. In [2], Agrawal *et al.* introduced the problem of mining from a set of transactions, known as *basket data*, association rules which have minimum *Support* and high *Confidence*.

Support

Defined in [2], the support of an association rule is the ratio of the number of transactions that contain items X and items Y to the total number of transactions available. ($\frac{n_1}{T}$).

Confidence

Also presented in [2], the confidence of an association rule is the ratio of the number of transactions that contain items Y of all the transactions that contain items X, ($\frac{n_1}{r_1}$). The confidence is defined to be the conditional probability that itemset Y occurs, given itemset X., i.e $P(Y|X)$ which is computed by $P(X \wedge Y)/P(X)$. Using the entries of the contingency table, the confidence is computed by n_1/r_1. With this measure, the notation "$\Rightarrow$" does not represent a real "implication" relationship, because the confidence measure is an estimate of the conditional probability of Y given X.

We apply the definitions of the support & confidence to the test flights example presented in Table 2.1 on page 9. For example, "whenever the *ICE AND RAIN* unit reports a failure, then, the *HYDRAULIC CONTROLS* and the *LANDING GEAR* units also report a failure". Using our formal definition, this association rule is written as:

ICE AND RAIN $\Rightarrow$ HYDRAULIC CONTROLS, LANDING GEAR

Then, X = { *ICE AND RAIN* }, and Y = { *HYDRAULIC CONTROLS, LANDING GEAR* }. A possible contingency table for this rule may be:

	Y	$\bar{Y}$	$\sum row$
X	127	34	161
$\bar{X}$	42	3238	3280
$\sum col$	169	3272	3441

Then, from the above contingency table, $Support = 127/3280 \approx 39\%$, and $Confidence = 127/161 \approx 79\%$. On the one hand, the support shows the percentage of test flights in which all three units reported a failure. This justifies making such an association because the three units fail together frequently enough. On the other hand, the confidence shows the probability of the right-hand-side of the association occurring while the left-hand-side has already occurred.

A data mining algorithm searching for the strongest N relationship rules can use this evaluation method to establish a rank between possible rules in the search space. Thus, the search algorithm can prune weak and undesired associations to reduce the size of the search space in an attempt to enhance its performance. In this case, pruning is based on the support of an association rule. The search algorithm prunes rules whose support is below a threshold. This approach is shown to work well in evaluating association rules defined on a market basket problem [2]. However, as Brin *et al.* show in [12], the *support & confidence* approach has several problems, such as; the measure of *support* does not perform well when evaluating negative association rules. In addition, pruning the space of all possible rules based on the frequency limits the evaluation to those rules which involve itemsets that occur frequently in data transactions. Therefore, strong associations are found among frequent items only which prevents the evaluation of significant but rare associations. Furthermore, the space of all possible associations may contain already known or confusing associations. An example of such a rule is: "marriage and divorce always occur together", a true but already known association. The *Support & confidence* evaluation is incapable of distinguishing these associations.

2.4.3 Evaluating Correlation Rules by The *Chi-Square* Test

The *Chi-Square* test is a method to test for independence between itemsets involved in a relationship rule. The *Chi-Squared* test extends the relationship between itemsets X and Y to a correlation relationship defined in [12]. Given a 2-dimensional contingency table, we apply the *Chi-Square* test to measure the degree to which the itemsets X and Y are independent. We first compute the value of the *Chi-Square* statistic (χ^2), where $\chi^2 = 0$ indicates the itemsets are independent. However, if the χ^2 statistic is greater than a cutoff value at the desired confidence level, then the assumption of independence is rejected. Cutoff values can be obtained from widely available tables for *Chi-Squared* distribution with appropriate confidence levels. Therefore, for our contingency table, we first compute the expected values for each of the entries $\hat{n_1}, \cdots, \hat{n_4}$ under the assumption of independence:

$$\hat{n_1} = P(X \wedge Y)T = P(X)P(Y)T = (\frac{r_1}{T})(\frac{c_1}{T})T = \frac{r_1 c_1}{T}$$

$$\hat{n_2} = P(X \wedge \bar{Y})T = P(X)P(\bar{Y})T = (\frac{r_1}{T})(\frac{c_2}{T})T = \frac{r_1 c_2}{T}$$

$$\hat{n_3} = P(\bar{X} \wedge Y)T = P(\bar{X})P(Y)T = (\frac{r_2}{T})(\frac{c_1}{T})T = \frac{r_2 c_1}{T}$$

$$\hat{n_4} = P(\bar{X} \wedge \bar{Y})T = P(\bar{X})P(\bar{Y})T = (\frac{r_2}{T})(\frac{c_2}{T})T = \frac{r_2 c_2}{T}$$

The χ^2 statistic value is then computed by the formula:

$$\chi^2 = \sum_{i=1}^{4} \frac{(n_i - \hat{n_i})^2}{\hat{n_i}}$$

Then, the computed χ^2 statistic value is compared to a threshold cutoff value for χ^2 taken at a confidence level α, denoted by χ^2_α. If $\chi^2 > \chi^2_\alpha$ we reject the assumption of X and Y being independent. For example, if we have the following contingency table for the rule S_1 of the form $X \Rightarrow Y$:

	Y	$\bar{Y}$	$\sum row$
X	17	8	25
$\bar{X}$	5	28	33
$\sum col$	22	36	58

Then,

$$\chi^2 = \frac{(17 - \frac{25\times22}{58})^2}{\frac{25\times22}{58}} + \frac{(8 - \frac{25\times36}{58})^2}{\frac{25\times36}{58}} + \frac{(5 - \frac{33\times22}{58})^2}{\frac{33\times22}{58}} + \frac{(28 - \frac{33\times36}{58})^2}{\frac{33\times36}{58}} = 16.874148$$

This χ^2 test value is greater than 3.84 (the value of $\chi^2_{0.95}$ taken from the χ^2 tables with confidence of 95%). Therefore, we reject the assumption of independence and we say that X and Y are dependent with confidence of 95%. Combined with the concept of minimum support, in [35], the χ^2 test can be used to extend association rules into *correlation* rules. Therefore, the χ^2 test is used to introduce an evaluation method to measure correlation relationships. Using the χ^2 test has advantages [12], such as, the χ^2 test is based on a statistical theory which takes into account all possible combinations of presence and absence of various itemsets. The χ^2 test captures *Correlations* in contrast with the *Confidence* which captures directional implications. However, the χ^2 test has some limitations and is recommended for use only when [12]:

1. all cells in the contingency table have expected values greater than 1;
2. and, at least 80% of the cells in the contingency table have an expected value greater than 5. In the case of a 2-dimensional contingency table, all four entries must have expected values greater than 5.

These conditions are often broken when evaluating association rules [12]. The work in [12] proposes a solution to this problem by using an exact calculation for the probability. However, calculating the exact probability

rather than the χ^2 approximation is not a simple task when the expected value of a cell is very small. Brin *et al.* in [12] suggests to ignore cells which have very small expected values. The justification for this is similar to that of the *Minimal Support* argument, if two sets of items X and Y are correlated by considering cells with very small expected values, then the correlation represent a rare event. This statement is based on the assumption that in many applications, events which have expectations less than 1 are considered as uninteresting [12]. Therefore, the χ^2 test may not be capable of capturing rare correlations.

2.4.4 Evaluating Dependencies by The G Measure

In [30], Oates & Cohen developed an algorithm, $MSDD$, to search for the highest K dependency score values found in a stream of data. The strength of the dependencies is measured by a user-supplied measure. They proposed using the G measure, a statistical measure of non-independence. We apply this method to relationship rules in an attempt to mine dependency rules that can provide a statistically reliable measure of risk assessment. A relationship rule can express dependencies of the form $X \Rightarrow Y$ which means that the itemsets X and Y are dependent. The strength of such dependency can be measured by computing the G score value. Therefore, given the 2×2 contingency table below for the relationship rule S of the form $X \Rightarrow Y$:

	Y	$\bar{Y}$	$\sum row$
X	n_1	n_2	r_1
$\bar{X}$	n_3	n_4	r_2
$\sum col$	c_1	c_2	T

The G value for the above table is computed by:

$$G = 2\sum_{i=1}^{4} n_i log(\frac{n_i}{\hat{n}_i})$$

Similar to the expected values computed for the χ^2 test, $\hat{n}_i$ is the expected value of n_i under the assumption of independence and is computed in the same way as for χ^2, see Section 2.4.3 on page 24. Lower values of G, near zero, indicate that X and Y are independent. As G increases, the strength of independence weakens, therefore, higher values for G indicate non-independence between X and Y [30]. A dependency is an unexpectedly frequent or infrequent co-occurrence of events over time. The G measure is very similar to a cross-entropy computed for two discrete probability distributes (namely $\frac{n_i}{T}$ and $\frac{\hat{n}_i}{T}$ where $i = 1, \cdots, 4$) [30]. It is easy to show that for the above case, the cross-entropy is given by $\frac{G}{2T}$. The G measure can also relate to a χ^2 distribution with one degree of freedom to compute the error probability of rejecting the null hypothesis of independence. Therefore, the G measure can be used as a test for statistical significance. An interesting feature of the G measure lies in the fact that it accounts for all possible dependencies between the itemsets $X, \bar{X}, Y$, and $\bar{Y}$ because of the contribution of all four entries in the contingency table into the G score value. For example, consider the following 2×2 contingency table for the rule S_1 of the form $X \Rightarrow Y$ (same example as in 2.4.3):

	Y	$\bar{Y}$	$\sum row$
X	17	8	25
$\bar{X}$	5	28	33
$\sum col$	22	36	58

Then,

$$G = 2(17log(\frac{17}{\frac{25\times22}{58}})+8log(\frac{8}{\frac{25\times36}{58}})+5log(\frac{5}{\frac{33\times22}{58}})+28log(\frac{28}{\frac{33\times36}{58}})) = 17.577017$$

In contrast, its χ^2 test value is:

$$\chi^2 = 16.874148$$

If we consider the entries in the above contingency table, $n_1 > n_2$ which indicates a frequent positive dependency from X to Y. Both of the G measure and the χ^2 test were capable of measuring the relationship for frequent positive relationships (a dependency or a correlation). The following contingency table for a rule S_2 shows the entries for a relationship which is not as strong as the previous one. Instead, this example shows a strong support for a negative relationship rule of the form $\bar{X} \Rightarrow Y$. The reason for the weak relationship in the positive form is that $n_1 < n_3$.

	Y	$\bar{Y}$	$\sum row$
X	7	1	8
$\bar{X}$	15	35	50
$\sum col$	22	36	58

Then,

$$G = 2(7log(\frac{7}{\frac{8\times22}{58}}) + 1log(\frac{1}{\frac{8\times36}{58}}) + 15log(\frac{15}{\frac{50\times22}{58}}) + 35log(\frac{35}{\frac{50\times36}{58}})) = 9.877405$$

In contrast, its χ^2 test value is:

$$\chi^2 = 9.684975$$

Indeed, both of the measures, G and χ^2, were capable of evaluating the strength of the relationships, dependency and correlation respectively. For both measures, this relationship is weaker than that of S_1.

Infrequent relationship rules are observed when the three entries n_1, n_2, and n_3 are significantly less than n_4. However, the relationship may still be of some strength. Consider the following contingency table for the rule S_3 of the form $X \Rightarrow Y$:

	Y	$\bar{Y}$	$\sum row$
X	3	1	4
$\bar{X}$	1	53	54
$\sum col$	4	54	58

Then,

$$G = 2(3log(\frac{3}{\frac{4\times4}{58}}) + 1log(\frac{1}{\frac{4\times54}{58}}) + 1log(\frac{1}{\frac{54\times4}{58}}) + 53log(\frac{53}{\frac{54\times54}{58}})) = 14.652742$$

In contrast, its χ^2 test value is:

$$\chi^2 = 31.033779$$

If we compare the score values computed by both of the G measure and the χ^2 test for all previous three examples,

Rule	G Score	χ^2
S_1	17.577017	16.874148
S_2	9.877405	9.684975
S_3	14.652742	31.033779

The ranking of these rules by their increasing G scores is S_2, S_3, S_1. In contrast, the χ^2 test seems to indicate that the strength of relationship S_3 is the highest of them all. In fact, the χ^2 test suffers from the expected values of n_1, n_2, and n_3 being less than 5, a deficiency of the χ^2 test reported in [12]. Therefore, since n_4 is significantly greater than the other three entries, the χ^2 test fails to reflect the true strength of the relationship.

Thus, the G measure is more capable of evaluating infrequent relationship rules than the χ^2 test.

In this thesis, we concentrate on relationships that indicate dependency strength in a positive direction, i.e. we evaluate rules whose contingency table shows support of the relationship between the presence of the itemset X and the presence of itemset Y, n_1, higher than the support of the relationship between the presence of the itemset X and the absence of itemset Y, n_2. Therefore, we force a G value of 0 for those relationship rules whose $n_1 \leq n_2$ and proceed to evaluate all possible combinations of n_1 and n_3 which may include strong but infrequent dependencies. However, we observed that when the number of transactions unrelated to the relationship rule $X \Rightarrow Y$ is significantly increased, i.e. n_4 is significantly increased, the G measure scales up the range of the G values preserving the relative strength of dependencies in most cases except for those infrequent dependencies. For example, if we added 20000 to entry n_4 of all the previous examples, we observe that the G values increase, however, the relative ranking of these G scores, shown on page 29, remains mostly the same except S_3 becomes the lowest G value. This behavior of G is a consequent of allowing all entries (n_1, n_2, n_3, and n_4) to contribute to the G score.

An important feature of the measure G is its lack of a logical interpretation of relationship rules. Thus, rules evaluated by the G measure cannot exhibit any properties of subsumption between different relationship rules. A G score value for the rule $X \Rightarrow Y$ represents a statistical significance of the LHS being followed by the RHS exactly as the itemsets X and Y appear in the rule. Items in X must occur together and items in Y must also occur together, then evaluating the relationship rule $X \Rightarrow Y$ by G determines the strength of the dependency between the two itemsets. This

interpretation eliminates the idea of subsumption between different relationship rules. For example, consider two relationship rules, the first is: S_4 of the form $X \Rightarrow Y$ where $X = \{a\}$ and $Y = \{b\}$, and whose contingency table is:

	$Y = \{b\}$	$\bar{Y} = I - \{b\}$	$\sum row$
$X = \{a\}$	6	1	7
$\bar{X} = I - \{a\}$	2	1	3
$\sum col$	8	2	10

The G score of S_4 is 0.447335. The second rule S_5 is also of the form $X \Rightarrow Y'$ (more special than Y) where $X = \{a\}$ and $Y' = \{b, c\}$ and its contingency table is:

	$Y = \{b, c\}$	$\bar{Y} = I - \{b, c\}$	$\sum row$
$X = \{a\}$	5	2	7
$\bar{X} = I - \{a\}$	1	2	3
$\sum col$	6	4	10

Since Y' is more specialized than Y, fewer transactions match Y' than those that matched Y. However, the G score for S_5 is 1.265374. Since S_5 has a G score greater than that of S_4, we cannot say that $S_4 = \{a\} \Rightarrow \{b\}$ is subsumed in $S_5 = \{a\} \Rightarrow \{b, c\}$. In fact, each of the rules expresses a dependency between the itemsets as opposed to between the individual items. Following this reasoning, we reject any form of subsumptions of rules evaluated by the G measure which, for the RHS (the itemset Y) of the rule, may be:

- Given the two rules $\{a\} \Rightarrow \{b\}$ and $\{a\} \Rightarrow \{c\}$, we cannot conclude the rule $\{a\} \Rightarrow \{b, c\}$, even if the two initial rules have the same G score values, because, transactions containing $\{b\}$ may be different than

those containing $\{c\}$. Thus, transactions containing the intersection of $\{b\}$ and $\{c\}$ may be fewer than those used to compute the G value. Hence, $\{a\} \Rightarrow \{b, c\}$ matches a different set of transactions used to compute a different G score.

- Given the two rules $\{a\} \Rightarrow \{b\}$ and $\{a\} \Rightarrow \{b, c\}$, we cannot conclude any subsumption properties between the two rules, eg. $\{a\} \Rightarrow \{c\}$, even if they both produce the same G score. This statement follows the above reasoning.

Similar to the RHS, it is possible to construct examples to reject any forms of subsumption properties for the LHS of relationship rules. The difference is specializing the LHS (itemset X) instead of the RHS (itemset Y). Therefore, when evaluated by the G measure, a relationship rule of the form $X \Rightarrow Y$ expresses a dependency between the itemsets X and Y. The G score measures how non-independent these itemsets are by assessing the statistical significance of the dependency. Thus, if the relationship rule $X \Rightarrow Y$ has a high G value, the interpretation that the itemset Y is more likely to occur given that the itemset X has already occurred is an incorrect interpretation. The correct interpretation of the G evaluation measure follows that the occurrence of the itemset X significantly affects the occurrence of the itemset Y. In other words, if $X \Rightarrow Y$ has a high G score value, this rule does not indicate that $P(Y|X)$ is high, instead, it indicates that $P(Y|X)$ significantly increases, or $P(Y|X) >> P(Y)$.

In section 2.3, we showed that the space of all possible relationship rules can be generated as a search tree. The generation is accomplished by specializing a single bit for every level of the tree. Oates and Cohen, in [30], derived an upper bound estimate on the evaluation measure G for all

descendants of a given dependency rule, called G_{max}. This upper bound on the G scores of all the descendants can be extremely useful for pruning relationship rules whose highest possible G is less than the strongest N rules found. Moreover, the search algorithm can safely terminate when all the subtrees of rules have a G_{max} scores less than the lowest G score from the strongest N rules found so far. This G_{max} is defined as an upper bound on all the descendants of a state S. For example, consider a state S in the tree of all possible rules. The state S represents a relationship rule of the form $X \Rightarrow Y$ whose contingency table is below:

	Y	$\bar{Y}$	$\sum row$
X	n_1	n_2	r_1
$\bar{X}$	n_3	n_4	r_2
$\sum col$	c_1	c_2	T

S will be specialized to produce its successor in the search tree. If the specialization occurs in the LHS of S, i.e the itemset X is specialized to X', then, the number of records in the dataset which match X, r_1, decreases by $\delta_1 + \delta_2$ producing the rule $X' \Rightarrow Y$ which has the following contingency table:

	Y	$\bar{Y}$	$\sum row$
X'	$n_1 - \delta_1$	$n_2 - \delta_2$	$r_1 - \delta_1 - \delta_2$
$\bar{X}'$	$n_3 + \delta_1$	$n_4 + \delta_2$	$r_2 + \delta_1 + \delta_2$
$\sum col$	c_1	c_2	T

But if the specialization in S occurs in the RHS of the rule, i.e the itemset Y is specialized to Y', then the number of records in the dataset which match Y, c_1, decreases by $\delta_1 + \delta_2$ producing the rule $X \Rightarrow Y'$ which has the following contingency table:

	Y'	$\bar{Y}'$	$\sum row$
X	$n_1 - \delta_1$	$n_2 + \delta_1$	r_1
$\bar{X}$	$n_3 - \delta_2$	$n_4 + \delta_2$	r_2
$\sum col$	$c_1 - \delta_1 - \delta_2$	$c_2 + \delta_1 + \delta_2$	T

In [30], Oates and Cohen showed that descendants of a state S, whose contingency table is given by (n_1, n_2, n_3, n_4), are constructed by specializing the LHS first, and then the RHS. When the LHS is specialized, any descendants of S, S' will have the same RHS itemset and a more specific LHS itemset. And when the RHS is specialized, deeper in the search tree, descendants of S, S' will have the same LHS itemset and a more specific RHS itemset. It can be shown that G is maximized in two cases: when $\delta_1 = n_1$ and $\delta_2 = 0$, or when $\delta_1 = 0$ and $\delta_2 = n_3$. Therefore, for states S with non-empty RHS, G is maximum when:

$$G_{max}(n_1, n_2, n_3, n_4) = max(G(n_1, n_2, 0, n_3 + n_4), G(0, n_1 + n_2, n_3, n_4))$$

For states whose RHS is empty, both the LHS and the RHS can be specialized further in the search tree. Oates and Cohen showed that G can also be maximized when:

$$G_{max}(n_1, n_2, n_3, n_4) =$$

$$max \left(\begin{array}{l} (1) \\ \text{if } n_1 \leq n_2 + n_3 + n_4 \text{ then } G(\frac{n_1+n_2+n_3+n_4}{2}, 0, 0, \frac{n_1+n_2+n_3+n_4}{2}) \\ (2) \\ \text{if } n_1 \geq abs(n_2 - n_3) \text{ then } G(0, \frac{n_1+n_2+n_3}{2}, \frac{n_1+n_2+n_3}{2}, n_4) \\ \text{else if } n_2 > n_3 \text{ then } G(0, n_2, n_1 + n_3, n_4) \\ \text{else } G(0, n_1 + n_2, n_3, n_4) \end{array} \right)$$

Thus, we use G_{max} to compute an upper bound on all the G scores of all the descendants of a state S in the search tree for all descendants which

have empty or non-empty RHS. In this thesis work, we use the G measure as the main state evaluation measure, and use the G_{max} as an upper bound on the G scores for pruning undesired states to reduce the size of the search space.

2.5 Chapter Summary

This chapter provided a basic knowledge of relationship rules. It starts with an introduction of relationship rules, what they are, what they mean, and how they can be defined formally. The chapter also supplied examples of relationship rules and how we could represent them using binary bit representation. This chapter, then proceeded to briefly review the history of association rules and the research conducted in this area. Thereafter, the chapter illustrated how all possible itemsets could be generated to produce all possible rules between all itemsets. Once relationship rules are generated, they must be subjected to an evaluation method to determine the nature of the relationship they represent and the level of relationship quality. This chapter reviewed some of such existing methods of interpreting relationships.

Chapter 3

Search Space Exploration

In chapter 2, we showed that the problem of mining relationship rules can be reduced into a search problem where a search algorithm explores a state tree search space looking for a desired set of states. The desired set of states contains the highest scoring N states measured by a state evaluation function. The search space, in this case, is the space of all possible rules among the binary representations of items involved in the dataset. The search space is generated by specializing the binary bits on both sides of the rule to generate new rules. This chapter reviews existing search algorithms related to solving such class of search problems. We identify and discuss issues involved in studying and evaluating search algorithms which explore tree-based search spaces.

First, we define the search problem, describe the structure of the search space, and state criterion by which search algorithms can be evaluated. The purpose is to set an evaluation technique to be used when comparing our proposed search algorithm to existing search algorithms. In addition, we establish a common understanding of the nature of the problem by analyzing features and shortcomings of other search algorithms. In subsequent

sections, we present reviews of existing search algorithms traditionally used to solve the problem at hand. Such search algorithms are divided into two families, one is the family of **Generic** search strategies, and the second is the family of domain **Specific** search strategies. The generic search strategies include: Best-First search, Depth-First search, Depth-First Iterative Deepening search, and Depth-First Branch and Bound search. The family of domain specific search strategies consists of: the Apriori search algorithm [5], and the Multi-Stream Dependency Detection algorithm [30, 31]. For each of these search strategies, generic or specific, we examine their strong points, and single out their weaknesses to arrive at several useful properties to be used for the development of a new search algorithm we present in the next chapter.

3.1 Search Problem Definition

The search problem is to find a desired set of the N states from a tree-based search space using an arbitrary state evaluation function. The search space is a finite state space tree denoting all possible states from which we must extract the "best" N states where:

1. each state represents a relationship rule in the space.
2. the root is the initial state.
3. from any state X, we can generate all successors states.

States are assumed to be more general than their successor states, i.e. successor states can be generated by specializing the parent state. We use a state evaluation function to indicate a score of how desired a state is.

3.2 Techniques to Evaluate a Search Algorithm

This section lists and defines properties used by researchers to evaluate the performance of search algorithms.

3.2.1 Storage Requirement

When evaluating a search algorithm, researchers traditionally measure the amount of run-time memory required by the search algorithm. Measuring storage requirements may determine the existence of any limitations on the size of the search space and/or how the search algorithm scales up. For instance, a search algorithm which maintains a list of open states through which the search algorithm advances can be memory-inefficient due to the high cost of storage. For the open list, the storage requirement depends on how many open states the list contains at any given point in time. Therefore, large search spaces can result in higher number of open states to maintain in the open list.

3.2.2 Execution Time Requirement

Considered as one of the most important evaluation criterion, the CPU execution time indicates how long a search algorithm takes to find the desired set of states in the search space. The execution time requirement of a search algorithm is usually measured on different classes of problems for which the algorithm claims the ability to solve. The obvious purpose of developing time-efficient search algorithms is to reduce the execution time required to solve any problem in its class.

3.2.3 Number of States Explored

When searching for a desired set of states in a large search space, a search algorithm may explore states that may (or may not) be desired. The number of such states can reflect the cost of reaching the set of desired states. This process of exploration is also known as the expansion of a state which points directly at the number of states expanded. Counting the number of expanded states provides a criteria by which we may rank search algorithms. An objective of a time-efficient search algorithm is to reduce the number of states it expands to reach the desired states.

3.2.4 Number of States Generated

Most search algorithms start at an initial state, explore a subset of states in the search space, then if successful, the search algorithm arrives at the set of desired states. When expanding a state, a search algorithm may generate several of its successor states, perhaps, order them, then select a candidate state for further exploration. In other words, for every state expanded, the search algorithm may generate some (or all) of its successor states and may process a subset of them. Different search algorithms may expand different-size portions of the generated states. Moreover, different search algorithms may generate different portions of the search space. Nonetheless, generating a state may require additional cost on execution time and/or storage space. Thus, the number of states generated by a search algorithm may suggest added cost when solving a particular problem. Therefore, comparing the number of states generated by different search algorithms can establish, yet another, criteria of ranking these algorithms. Similarly, the objective here is to scale down the number of

generated states.

3.2.5 Number of States Evaluated

Solving the problem of finding a set of desired states in a search space involves the selection of a set of candidate states from all states found in the search space. This process of selecting candidate states is based on establishing an order of available states by evaluating each one of them. State evaluation is usually accomplished by applying an evaluation function to a state. After generating successor states of an expanded state, a search algorithm may evaluate these successor states to determine the set of candidate states to either be included in the set of desired states, or to be considered for further state expansion. The process of evaluating a state may be costly when consulting the dataset. Therefore, if the algorithm evaluates more states, it consumes more CPU time. Different search algorithms may generate different portions of the search space and may evaluate some or all of the generated states. In our case, we evaluate all generated states, thus, the number of generated states is the same as the number of evaluated states.

3.2.6 Number of Passes on Datasets

Evaluating a state in the state space is a decision-making activity which may require collecting information about the state, about the search space, or about the desired states, etc. Most importantly, is the information about the problem domain as presented by the data. A search algorithm may need to consult the available dataset during the state evaluation process. When consulting the dataset, the algorithm may perform a number of passes

on the dataset. If the dataset is significantly large, the search algorithm may require additional work due to consulting the dataset more frequently. Therefore, monitoring the number of passes on the data set may provide us with a method to rank different search algorithms and determine their effectiveness more realistically.

3.3 Strategies for Efficiency

In this section, we describe existing methods of how search algorithm attempt to enhance their efficiency. The main strategies used by researchers are: ordering the set of candidate states for expansion, pruning undesired states, and stopping the search algorithm.

3.3.1 Ordering The Set of Candidate States

After selecting a set of candidate states from the list of successor states, a search algorithm may order this list of candidate states to continue exploring the search space. State ordering can cause the search algorithm to reach the desired states sooner. For instance, ordering candidate states for expansion based on their potentials of reaching desired states can sometimes cause the search algorithm to find the desired states earlier. The ordering criteria depends on properties of the search space and the evaluation function used to determine the quality of a state. These properties are usually specific to the problem domain. In fact, properties used to order the list of candidate states rely heavily on the choice of the state evaluation function simply because the aim is to explore the most beneficial candidate state available to the algorithm. However, ordering a list of candidate states implies some form of a sorting operation which is well known to consume

computational cycles. First, this added cost will, most certainly, affect the execution time of the search algorithm, especially for larger search spaces. Second, state ordering may affect the quality of states found at different stages of the search, higher scoring states are reached earlier on in the search. An optimal search algorithm should yield the same set of states regardless of the ordering criteria of candidate states.

3.3.2 Pruning The Search Space

Pruning is a technique many search algorithms use to reduce the number of candidate states to consider for exploration. The ideal situation would have the search algorithm explore the optimal number of states required to reach the desired set of states. However, reality is much different. In most cases, search algorithms use heuristics to guide the search towards the desired goal. Heuristic functions, by definition, are incapable of guaranteeing the expansion of the optimal number of states to reach the desired set of states. However, heuristics attempt to assist the search to reach the solution more efficiently. Therefore, using the evaluation function as a heuristic measure for pruning may allow the algorithm to perform an effective pruning of the search space, thus, reducing the amount of search the algorithm has to perform. However, pruning must be done in a correct method without overlooking any useful states in the search space. We study the effectiveness of the pruning techniques employed by a search algorithm, without compromising the correctness of solution states found.

3.3.3 Search Stopping Conditions

A search algorithm explores a search space by generating and evaluating a set of states, selecting a set of candidates from these states, then expanding some (or all) of the candidate states until it reaches the desired ones. Terminating the search algorithm must guarantee two conditions, the first is that the set of reached states is correct, and the second is that the search algorithm terminates immediately after reaching the set of desired states without performing any additional work. For instance, a restrictive stopping condition may cause the search algorithm to terminate too early, thus, the search algorithm terminates before it reaches all desired states. Furthermore, a loose stopping condition may cause the search algorithm to perform additional unnecessary work which translates into added execution cost. A correct search algorithm always reaches the set of desired states regardless of the problem domain avoiding unnecessary work.

3.3.4 State Evaluation Functions Related to Problem Domains

Searching for a set of desired states in a search space involves a decision making step to examine the quality of states explored. An evaluation function is usually used to evaluate the quality of a state to determine whether or not a state is desired. Traditionally, the state evaluation is based on the cost required to reach that state. Alternatively, state evaluation can be based on its "goodness" score. This "goodness" score expresses how well a state fits the desired properties. Furthermore, state evaluation functions may provide heuristics to help guide the search through the search space directly (or as directly as possible) to the desired states. State evaluation heuristics may also provide effective pruning strategies which may

yield a significant reduction in the size of the search space. Given a state evaluation function to reflect a measure of an interest related to a particular problem domain, a heuristic use of this function can determine how well a search algorithm performs in that domain. Therefore, if the state evaluation function is supplied as an input to the search algorithm, then the search becomes independent of the problem domain in the sense that the algorithm becomes capable of exploring search spaces using a state evaluation function that measures particular interests related to a specific domain. Our proposed search algorithm takes the state evaluation function as an input, therefore, the quality of desired states found by our search algorithm reflects how well the supplied state evaluation function is suited for the problem domain.

3.4 Generic Search Algorithms Strategies

In this section we present an overview of generic search strategies several of which are well known search algorithms that have been used for state search space exploration. Heuristic search algorithms exist in the literature to solve a wide variety of space exploration problems. We focus on the major ones. We applied these search algorithms to our search problem to demonstrate the feasibility of their use and illustrate their advantages and disadvantages. Some of these generic search strategies include: Breadth-First Search, Best-First Search (BFS), Depth-First Search (DFS), Depth-First Iterative Deepening Search (DFIDS), and Depth-First Branch and Bound Search (DFBBS).

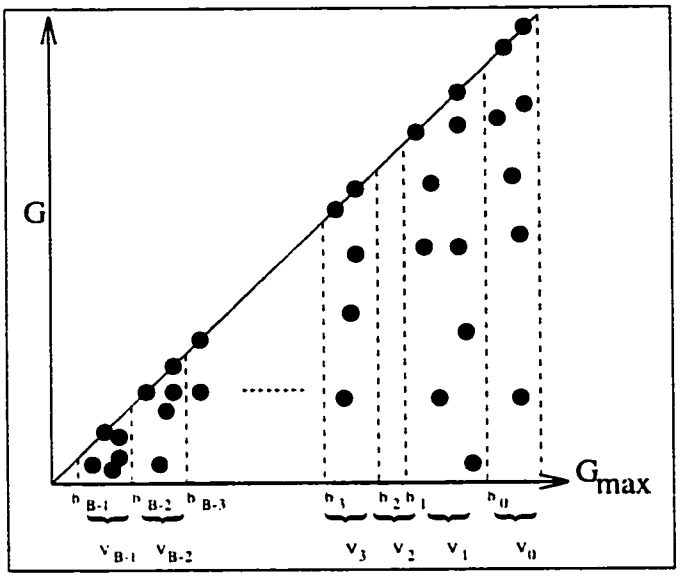

Figure 3.1: The Search Space Illustrated on G & G_{max}

3.4.1 Search Space Definitions Using An Evaluation Function G

In this section, we adapt notations and analysis done by [41] to our problem of searching for the N-highest G-scoring states in a tree-based search space. The following analysis is restricted to state space trees where for each state (s), $G_{max}(s)$ denotes an upper bound on the state evaluation score G of all successors of s (including s itself). We also assume that G_{max} is monotonic with non-increasing values where $G_{max}(s) \leq G_{max}(s')$ if s' is a successor of s. In our analysis, we use the following notations (shown in Figure 3.1):

- Let B denote the total number of different G_{max} values taken over all states in the state space tree.
- Let b_i denote the ith-highest G_{max} value where b_0 is the highest G_{max} value, b_1 is the second highest G_{max}, etc. Thus, $b_0, b_1, b_2, \cdots, b_{B-1}$ is a sequence of all G_{max} values in a decreasing order.

- We define the *heuristic branching factor* (br) as the ratio of the number of states of a given G_{max} value to the number of states with the next smaller G_{max} value. For instance, when the function G is simply the depth of the tree, then the *heuristic branching factor* (br) is the brute-force branching factor used in [22]. We assume that $br > 1$ [1].

- Let V_i denote the set of states whose G_{max} value is the ith-highest of the G_{max} values. i.e. the set V_0 contains all the states whose G_{max} values are equal to b_0. the set V_1 contains all the states whose G_{max} values are equal to $b_1, \cdots$, and the set V_{B-1} is the set of states that have the lowest G_{max} value which are equal to b_{B-1}. Thus $V_0, V_1, \cdots, V_{B-1}$ is a sequence of sets of states in decreasing order of their G_{max} values. The descendants of a state in V_i can only occur in those V_j for which $j \geq i$ because G_{max} is a monotonic function with non-increasing values. V_0 contains the starting state (the root of the state space tree) and is assumed to be a singleton set. When the G_{max} is the depth of the state, we can assume that the sequence of sizes $|V_i|$ of state sets is a geometric progression with ratio br = the *brute-force heuristic branching factor*. This progression follows a combinatorial increase followed by a decrease in the number of successor states. Alternatively, when G_{max} is a measure of quality, this assumption is not true.

- Let b_d denote the minimum G_{max} value of the "best" N states. Therefore, the search algorithm should not terminate without exploring states in V_i for $i \geq d$.

In this thesis, we use the above notations to analyze the complexity of a search algorithm which attempts to reach the N-highest G-scoring states

[1] This assumption is related to the performance of DFIDS.

in a tree-based search space.

3.4.2 Breadth-First Search

Breadth-First Search is a well known search strategy which explores all states one step away from the initial state, then expands all states which are two steps from the initial state, then three steps away, etc. Breadth-First search explores the state space tree level by level based on the depth of the state space tree until goal states are reached. Breadth-First search requires a queuing mechanism to store candidate states found in every level in order. As the search advances into deeper levels of the tree, the number of states found at every level increases based on the branching factor of every state. The work in [41] shows that for Breadth-first search, the storage requirement increases linearly by the number of states explored. This implies costly memory requirements when the levels in the state space tree have a very high number of states and/or the tree is very deep. Our problem assumes an exponential number of states as a function of the tree depth. Therefore, the Breadth-First Search proves ineffective for our search algorithm due to its high storage cost.

3.4.3 Best-First Search (BFS)

Similar to Breadth-First Search, Best-First Search is a generic search algorithm which explores the state space tree using an open list concept. The open list stores the frontier states of a partially explored search tree. The literature defines BFS to expand candidate states in a non-decreasing order of cost $f(s)$, the cost of reaching that state. However, to customize BFS to our problem, we use the "goodness" measure of a state instead of

search cost. Hence, BFS expands the states in a non-increasing order of "goodness" scores $G(s)$, because we require BFS to expands the successor state which has the highest "goodness" score first to yield the "best" G-scoring state instead of the lowest cost state, as usually defined. When $G(s) = depth(s)$ then BFS becomes a Breadth-First Search [41]. BFS uses a priority queue technique to keep track of generated successors states for further exploration. These successor states are explored according to their desired quality measure, i.e. the best successor state is expanded first. For complexity analysis, BFS expands all states in V_i for $i < d$. Let N_{BFS} denote the number of states expanded by BFS. Then using notations from section 3.4.1 on page 45, we have:

$$N_{BFS} = \sum_{i=0}^{i=d} |V_i| = \sum_{i=0}^{i=d} br^i = \frac{br^{d+1} - 1}{br - 1} \text{ for } br > 1$$

However, like Breadth-First Search, storage requirements for BFS is linear in the number of states expanded [41] resulting in an enormously high storage requirement for moderate size problems. Since our problem deals with an exponentially large state space with a high branching factor, BFS proves impractical to search such a space.

3.4.4 Depth-First Search (DFS)

The main benefit of using DFS is solving the memory requirement problem seen in search algorithms which use the Breadth-First search technique. DFS offers a solution based on generating only one successor of the most recently expanded state recursively and up to a depth cutoff. When DFS reaches the depth cutoff, it backtracks and generates the next available successor and proceeds in a recursive fashion. This DFS explores and maintains a single path from the root of the tree to the current state being

explored. Consequently, the storage requirement for DFS is linear in the depth of the search tree. Furthermore, DFS limits the storage requirements when constructing successor states by simply making a small change in the parent state when advancing into the search tree. When backtracking, parent states can be constructed by undoing these changes to the successor states. Using recursive expansion of states along with backtracking, DFS may search very deep into the search tree before it backtracks to a next available successor, particularly when the depth of the desired states is unknown. Therefore, DFS is categorized as a time-bounded algorithm rather than a space bounded algorithm due to the recursive searching and backtracking. One method to avoid this effect is to require a depth cutoff at which DFS terminates. A low estimate of the depth cutoff may cause the algorithm to terminate before finding all the desired states. On the other hand, a high estimate of the depth cutoff may cause the algorithm to consume an expensive run-time cost to find the desired states [21, 22]. Alternatively, if states are generated based on their potentially desired quality measure, then fewer undesired states may be generated which, possibly, reduces the size of the search tree. In addition, an aggressive but correct pruning of the search space, especially early in the search, can help reduce backtracking by eliminating potentially undesired states, so that DFS becomes more effective. Therefore, given that our problem's search space grows exponentially in size, a simple DFS can become very costly when used to find a set of desired states. However, DFS provides us with a hint as to how we can reduce the storage requirements. In the meantime, an important issue is the possibility of using DFS to explore search spaces without suffering form its exponential run-time requirement by either computing a suitable depth cutoff value and/or generating states with a focus

on the desired quality. If so, how can we determine a suitable cutoff value such that the search terminates correctly in a reasonable time?

3.4.5 Beam Search

Beam search represents a classic compromise between memory required and CPU time. Like Breadth-First Search, Beam Search explores the search tree level by level. However, unlike Breadth-First Search, Beam Search advances through the levels of the tree on the best w states. For each level, the search explores the best w states from the open list, generates their successors, then inserts into the open list the best w successors and the remaining states are ignored. Therefore, the size of the open list is limited to best w states. This idea may work to balance the memory and the CPU requirements, however. Beam Search does not backtrack to search for the next best states after exploring w states on every level. Therefore, it may not be useful for search for the best N states in the search tree.

3.4.6 Depth-First Iterative Deepening Search (DFIDS)

DFIDS is found to reduce storage requirements due to using DFS as its main search strategy [21] where, similar to DFS, the storage requirement is linear in the depth of the state space tree. DFIDS [22] performs multiple Depth-First searches in the search space with a progressive depth cutoff, which is called the search bound. First, DFIDS performs a DFS with a search bound of one step away from the root, then starts a new DFS with a search bound of two steps away, and so on. For each subsequent DFS, DFIDS increments the search bound to proceed deeper into the search space tree until it reaches the desired states. The process of performing

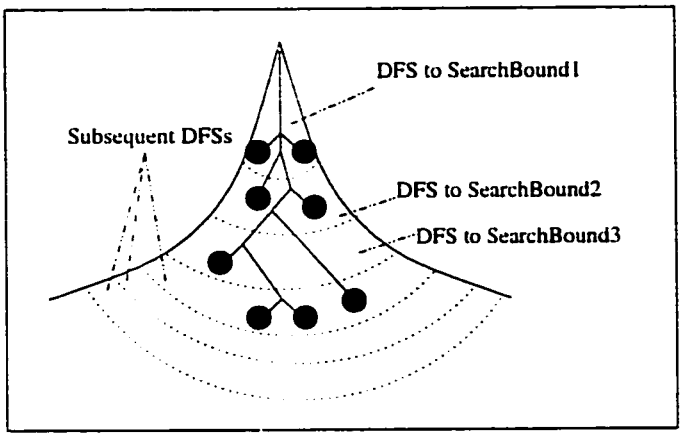

Figure 3.2: Iterative Deepening Illustrated

a single DFS with a fixed search bound is called a single DFS pass (*see Figure 3.2*). When the search bound is based on the G_{max} values in the search space, DFIDS explores the search tree based on the decreasing sequence of G_{max} values $b_0, b_1, b_2, \cdots, b_{B-1}$. In [21], Korf shows this approach is guaranteed to find the desired solution by reiterating the DFS on the search space tree until the desired set of states is found. Unfortunately, DFIDS has some drawbacks. In every DFS pass, DFIDS revisits states that have been explored in previous DFS passes in addition to those new states found in the new search bound. Therefore, carrying out a large number of DFS passes on a large search space can degrade the performance of the algorithm in terms of performing wasted computations due to revisiting previously expanded states frequently. Consequently, DFIDS may consume an increasing amount of processing time when executed improperly. However, [21] also shows that the wasted computations do not affect the asymptotic growth of the running time for exponential tree searches. Using notations and analysis developed in [41] and customized to our search

problem in section 3.4.1 on page 45, we present the following complexity analysis of DFIDS. During a given DFS pass, DFIDS expands previously expanded states plus new states found in the newly relaxed search bound. Let X_{DFIDS} denote the number of expanded states by DFIDS, and let P_i denote the number of states expanded by DFIDS in a DFS pass i, then:

$P_0 = |V_0| = br^0$

$P_1 = |V_0 \cup V_1| = br^0 + br^1$

$P_2 = |V_0 \cup V_1 \cup V_2| = br^0 + br^1 + br^2$

$\vdots$

$P_i = |V_0 \cup V_1 \cup \cdots \cup V_i| = br^0 + br^1 + \cdots + br^i$

Then, the total number of states expanded by DFIDS is:

$$X_{DFIDS} = \sum_{i=0}^{i=d} P_i = \sum_{i=0}^{i=d} \sum_{j=0}^{j=i} |V_j| = \sum_{i=0}^{i=d} \sum_{j=0}^{j=i} br^j$$

Using $\sum_{j=0}^{j=i} br^i = \frac{br^{i+1}-1}{br-1}$ for $br > 1$, then:

$$X_{DFIDS} = \sum_{i=0}^{i=d} \frac{br^{i+1} - 1}{br - 1}$$

We break the fraction into two terms, we have:

$$X_{DFIDS} = \sum_{i=0}^{i=d} \frac{br^{i+1}}{br - 1} - \sum_{i=0}^{i=d} \frac{1}{br - 1}$$

We factor $\frac{br}{br-1}$ out:

$$X_{DFIDS} = \frac{br}{br - 1} \sum_{i=0}^{i=d} br^i - \sum_{i=0}^{i=d} \frac{1}{br - 1}$$

Then, we substitute the sum of the series (as shown above):

$$= \frac{br}{br - 1} \frac{br^{d+1} - 1}{br - 1} - \frac{d + 1}{br - 1}$$

But the number of states expanded by BFS is $X_{BFS} = \frac{br^{d+1}-1}{br-1}$. Thus for $br > 1$:

$$X_{DFIDS} = \frac{br}{br - 1} X_{BFS} - \frac{d + 1}{br - 1}$$

Therefore:

$$X_{DFIDS} \approx \frac{br}{br-1} X_{BFS}$$

Obviously, DFIDS expands more states than BFS [21, 38, 41]. When the *heuristic branching factor* (br) > 1, i.e. when the search tree grows exponentially with the state evaluation function, then the running time of iterative deepening is $O(br^d)$. However, DFIDS asymptotically expands more states than BFS when br is close to 1. When $br \leq 1$, the search tree does not grow exponentially with the state evaluation function, and iterative deepening encounters its worst case performance. The worst case, reported in [32, 41], occurs when each of the states in the search tree have a unique state evaluation score. Recall, in a DFS pass, DFIDS expands previously expanded states plus new states found in the new bound. If the number of new states for each of the DFS passes is 1, then the search is faced with a degenerate search tree with 1 successor for each of the states. Then, the performance of DFIDS deteriorates to $O(X_{BFS}^2)$ [32].

3.4.7 Depth-First Branch and Bound Search (DFBBS)

Customized to using the state evaluation function instead of exploration cost for the search bound, the Depth-First Branch and Bound search starts with a low bound on the desired state evaluation scores. Whenever DFBBS finds a state with a better evaluation score, the search bound on states to be explored is raised. DFBBS searches the space in a depth-first fashion until it reaches the set of desired states. In this manner, eventually, DFBBS finds the states that have the highest desired quality scores. Like DFIDS, this behavior requires the state evaluation function to be monotonic, in a non-increasing order. Unlike DFIDS and like BFS, DFBBS does not

1. $BestN$ = {N randomly generated states and their G scores }
2. $SearchBound = min_G(BestN)$.
3. s = the root of the search tree.
4. $DFBBS(s, SearchBound, BestN)$:
   - S = { all successor states of s with their G scores }.
   - for $s \in non\_empty(S)$ do:
     - if $G(s) > SearchBound$ then
       - replace the minimal G-scoring state in $bestN$ with s.
       - SearchBound = $min_G(BestN)$.
     - if $G_{max}(s) > SearchBound$ then
       - $DFBBS(s, SearchBound, BestN)$.
     - Prune $r \in S$ where $(G_{max}(r) \leq SearchBound)$.

Figure 3.3: A Possible Implementation of The DFBBS Algorithm

perform multiple passes on the search tree. Figure 3.3 shows a possible implementation of the DFBBS to search for the highest G-scoring N states. The algorithm starts with a randomly generated set of Best N states. The search starts at the root of the search tree and recursively descends into the successor sub-trees. Whenever the algorithm reaches a state whose G score is greater than the current search bound, then the DFBBS updates the set of best N states and raises the search bound accordingly. The algorithm recursively descends on successor states whose G_{max} scores are greater than the current search bound. This algorithm terminates naturally by backtracking past the root state. Note that when the search algorithm, described in Figure 3.3, replaces the minimal G-scoring state $s_1 \in BestN$ by a newly reached state s_2 where $G(s_2) > G(s_1)$, the search bound is updated to the minimal G score found in the set of best N states rather

than to the $G(s_2)$ because there may exist multiple states in $BestN$ whose G scores are equal to $G(s_1)$. This algorithm may be effective, however. due to time limits, we have not evaluated this algorithm against the DFIDS.

3.5 Specific State Search Strategies

The following sections present overviews of existing rule mining algorithms developed by researchers in the field of data mining, specifically in the area of mining rules from transaction datasets. These overviews identify the two algorithms against which our proposed algorithm may be compared. Throughout our study. we use these two algorithms as benchmarks analyzing their weaknesses and strengths. These two algorithms are *Apriori* [5] and Multi-Stream Dependency Detection *MSDD* [30, 31].

3.5.1 The Apriori Algorithm

The *Apriori* algorithm for discovering association rules between items in a dataset was introduced by Agrawal in [5]. The problem of mining for association rules is decomposed into two parts:

1. Find all sets of items that have support of at least s, call them *the large itemsets.* In [5], Agrawal proposed the *Apriori* and *AprioriTid* algorithms for solving this problem.

2. Use the large itemsets to generate all possible association rules between possible itemsets. For every large itemset I, find all non-empty subsets of I. For every such subset a, produce a rule of the form $a \Rightarrow (I - a)$ if the ratio of $support(I)$ to $support(a)$, the confidence of the rules $a \Rightarrow (I - a)$, is at least c. Since the itemset a represents

1. I_1 = {set of large 1-itemsets }.
2. for ($k = 2$; $non\_empty(I_{k-1})$; $k++$) do
   - $C_k = Generate\_New\_Candidates(I_{k-1})$
   - for all records $r \in$ dataset do:
     - C_r = all subsets of the set of items that appear in the record r, this is determined by calling the $subset(C_k, r)$ function.
     - for all candidates $rc \in C_r$ count the records in which items from rc appear. (count the records that have items rc in them).
   - I_k ={candidates in C_k whose support $\geq$ minimum support S}.
3. Results = the union of all I_k.

Figure 3.4: The Apriori Algorithm

the LHS of the rule, we must consider all subsets of I to generate all the different sets of the RHS. Agrawal introduced a fast algorithm to solve this problem in [6].

Algorithm Apriori

The method of generating and evaluating the discovered rules used by the Apriori algorithm is based on the *Confidence & Support* method described in section 2.4.2. Illustrated in Figure 3.4, the Apriori algorithm starts by generating the large 1-itemsets (I_1 contains sets of items with minimum support), then, the Apriori algorithm uses this I_1 to generate C_2, the set of candidate itemsets for the second pass. Thereafter, the algorithm scans the dataset and records the support counts for the candidate itemsets in C_2. The counting of support is performed by determining the candidates in C_2 which are contained in a given dataset record r. Similarly, for a subsequent pass k, the Apriori algorithm continues to use the large (k-1)-

1. insert into C_k the union of any two itemsets in I_{k-1} whose first $k-2$ items are the same and the last items are different.
2. for each itemset $c \in C_k$ do
   - for each (k-1)-subset s of c do
     if ($s \notin I_{k-1}$) then delete c from C_k.

Figure 3.5: The Algorithm of *Generate_New_Candidates*() Function.

itemset I_{k-1} to generate the candidate itemsets C_k, then, scans the dataset records and counts the support for candidate itemsets in C_k. The counting of support is done by determining the candidates in C_k which are contained in a given dataset record r. This task is performed by the *subset*() function which stores itemsets in a hash-tree data structure which returns a set of references to desired itemsets, (see [5] for description of the functionality and buffer management of the *subset*() function).

The function *Generate_New_Candidates*(I_{k-1}) consists of two steps, join and prune, both are shown in Figure 3.5. The first step joins any two itemsets in I_{k-1} whose first $k-2$ items are the same and the last items are different. This join produces $C'_k = \{X \cup X'\}$ such that $X, X' \in I_{k-1}$ and $X \cap X' = \{x_1, x_2, \cdots x_{k-2}\}$. The second step prunes C'_k deleting all itemsets $c \in C'_k$ for those c that have at least one $(k-1)$-subset $\notin I_k$. This prune step produces $C'_k = \{X \in C'_k | X$ contains k members of $I_{k-1}\}$. For example, let $I_4 = \{(1,2,3,5),(1,2,3,6),(1,3,4,5),(1,3,4,6),(1,2,5,6),(1,3,5,6),(2,3,5,6),(3,4,5,6)\}$. The join step produces $C_k = \{(1,2,3,5,6),(1,3,4,5,6)\}$. The prune step deletes $(1,2,4,5,6)$ from C_5 because one of its (4) subsets, $(2,4,5,6) \notin I_4$.

The main deficiency of the Apriori algorithm is the list of candidate

1. Perform a depth-limited run of iterative deepening on an estimated set of items I'.
2. Remove all items in I' that are not involved in the best k rules found.
3. Add some small number of randomly selected items from $I - I'$ into I'.

Figure 3.6: The MSDD Algorithm.

itemsets. Like the open list in Best-first search technique, the candidate list concept requires an enormous amount of memory space for a moderate size but dense search space. A dense search space has a high state population in each of the search bounds (or depth bounds), for instance if the search space has a high heuristic branching factor. It thus grows exponentially with the state evaluation function, thus producing high state populations in each of the search bounds. Enumerating all possible association rules between itemsets can suffer from a combinatorial explosion [11]. Many papers presented improvements to how the Apriori algorithm enumerates all frequent itemsets from data [5, 13]. However, these papers do not deal with the combinatorial explosion of the search space. This limitation of the Apriori algorithm motivates our thesis work to develop an efficient rule mining algorithm which consumes less memory.

3.5.2 Multi-Stream Dependency Detection Algorithm (MSDD)

In [30], Oates & Cohen devised an algorithm to detect dependencies in a multi-stream data, known as *MSDD*. The algorithm is based on applying a variation of the Best-First Search to the space of all possible rules generated from categorical data streams. After rules are generated using a method very similar to that of the Apriori algorithm, MSDD uses the G measure

to evaluate and establish a rank between the discovered dependencies. The main deficiency of this approach is once again the open list of candidate states. The memory requirements of such a strategy increase linearly with the number of expanded states. In [31], Oates introduced an improvement to the MSDD search approach by employing the iterative deepening technique applied to the space of all possible dependencies. In this section, we review the latest improvement of the MSDD algorithm which is very similar to the search algorithm we propose in this thesis work, $ID_{G_{max}}$. Figure 3.6 presents the MSDD algorithm which starts at the root of the search tree of all possible rules. The algorithm iteratively performs a series of Depth-First searches in search of the highest k values of G scores. The generation of the search tree is done by simply specializing rules starting from the most general rule (at the root of the tree) to more specific rules. The series of the Depth-First searches is performed with a progressively increasing search bound based on the depth of states in the search tree. The algorithm relies on pruning undesired rules according to the G_{max} heuristic to reduce the number of rules it explores. In addition, Oates *et al.* in [31] attempt to reduce the size of the search space by using several methods. First, when the number of the fringe states decreases below a user-defined fraction of the size of the search space, MSDD stops the iterative deepening and proceeds with a single Depth-First search without limitations on the search bound. This feature reduces the re-exploration of already expanded states. Second, when expanding a state (a rule), the children are generated by adding a single new item from the available item set I to the rule, thus producing a new child of that rule. The children are ordered such that the search expands those successors that have higher values of G. Moreover, the exploration of successor states is based on exploring states that have

1. Initialize the $SearchBound_{TreeDepth} = 1$.
2. While (there are more fringe states whose G_{max} score > lowest G found) do:
   - perform a Depth-First Search to a maximum $SearchBound_{TreeDepth}$.
   - increment $SearchBound_{TreeDepth}$ by 1.

Figure 3.7: $ID_{TreeDepth}$ Search Algorithm

the set of most frequent items (items which appear in data transactions more frequently) under the assumption that frequently appearing items continue to be useful in the successor rule.

Third, and finally, the performance enhancing feature of MSDD lies in using an Apriori method to determine which items can be removed from I such that the search algorithm generates rules involving useful item sets only. Initially, the algorithm starts with an approximation of the set of useful items I'. This approximation is determined by performing an exhaustive search on all the subsets of all items I to a shallow depth. Then the MSDD search is performed a fixed number of times where items observed to be involved in the rules, which have the k highest G scores, are used to adjust I'. A single DFS pass of the iterative deepening of the MSDD algorithm introduced by [31] was shown in Figure 3.6.

Our Experimental Implementation of MSDD ($ID_{TreeDepth}$)

To correctly compare our new algorithm to MSDD, we produced an implementation of the MSDD algorithm which uses iterative deepening with a search bound based on the depth of states in the search tree, we call it $ID_{TreeDepth}$ and is illustrated in Figure 3.7. $ID_{TreeDepth}$ enhances the efficiency of the search using similar features to those used by MSDD. For

instance, $ID_{TreeDepth}$ relies on pruning based on the G_{max} to reduce the size of the search space. $ID_{TreeDepth}$ also uses state ordering to explore those states which are expected to produce a higher G score value early on in the search. However, $ID_{TreeDepth}$ has a few alteration to ensure a justified comparison. The differences are listed below.

First, $ID_{TreeDepth}$ does not use any approximation of the set of items involved in the set of desired rules. The justification is simply because we require our algorithm, $ID_{G_{max}}$, to guarantee the discovery of the strongest N rules in the space. In [31], Oates states that this approximation does not guarantee the convergence of the estimated set of frequent items I' to the actual set of frequent items used in the rules which have the k highest G scores.

The second difference between MSDD and $ID_{TreeDepth}$ is that MSDD returns the highest k values of the G scores found in the space. Many rules may have the same G-score values. $ID_{TreeDepth}$ returns the N rules which have the highest G scores, they may all have the same G scores. This difference is required to ensure that $ID_{TreeDepth}$ and $ID_{G_{max}}$ search for the same result.

The third difference between $ID_{TreeDepth}$ and MSDD is that, unlike MSDD, $ID_{TreeDepth}$ continuously performs DFS passes limited to a maximum search bound without switching to a single unlimited Depth-First search at any point of execution. This difference is used to ensure consistent comparisons between the different search bound strategies used by both algorithms. When using the depth of a state as a search bound strategy, the effectiveness of stopping the iterative deepening and switching to a single unlimited DFS is based on the consistent behavior of the combinatorial increase followed by a decrease in the size of the search space shown

in [30]. However, $ID_{G_{max}}$ uses a different search bound strategy which behaves differently with respect to the shape of the search space it generates. Therefore, we omitted this feature from our MSDD implementation.

3.6 Chapter Summary

This chapter presented the related work review of existing strategies to explore state spaces. In this chapter we defined the search problem and presented properties traditionally used to evaluate the quality of search algorithms designed to explore search spaces. In subsequent sections of this chapter, we reviewed both generic and specific state space exploration methods. The generic search methods are BFS, DFS, DFIDS, and DFBBS. The domain specific search strategies are the Apriori, and the MSDD algorithms. In both cases, generic and specific, we explained how the algorithms work and presented limitations we observed in an attempt to justify the use of such techniques in the development of our $ID_{G_{max}}$ search algorithm.

Chapter 4

Proposed State Space Search Strategies

In this chapter we propose a search algorithm $ID_{G_{max}}$ which attempts to search for the "best" N states in a tree-based search space. Our proposed algorithm is a generic search algorithm which is independent of the problem domain. The $ID_{G_{max}}$ search algorithm uses characteristics of existing search techniques shown in Chapter 3, to be effective in exploring state space trees. In subsequent discussions, we study and analyze variations of the $ID_{G_{max}}$ algorithm with different search bound strategies to determine the extent to which we succeed in addressing issues that improve the performance and quality of results.

4.1 Definitions

In this section, we introduce the functions needed by the $ID_{G_{max}}$ algorithm when it explores the search space. Using the search space notations introduced in 3.4.1, our analysis deal primarily with state space trees as

described in [41] using essential functions, such as:

1. For a state X in the search space, we let $G(X)$ denote a state evaluation function used to determine the quality score of X, The function G is not assumed to exhibit any monotonic or systematic properties.

2. For a state X, we let $G_{max}(X)$ denote a heuristic upper bound function on the G scores of all descendants of X, i.e. $G_{max}(X)$ denotes the highest possible G score among all descendants of X.

3. Since G_{max} is an upper bound on G, then $G(Y) \leq G_{max}(X)$ for any Y which is a successor of X. As the depth of states increases in the search tree, the state evaluation function G may be non-monotonic. However, G_{max} must exhibit a monotonic property with non-increasing values. Therefore, $G_{max}(Y) \leq G_{max}(X)$ for all Y which is a successor of X. We attempt to improve the performance of the $ID_{G_{max}}$ search algorithm by using the $G_{max}(X)$, for two purposes. We use it first, to prune states which are guaranteed to produce low G scores, and second, to order states when selecting a candidate state from the list of candidate states for recursive exploration.

4. For a state X in the search space, let $generate(X, L)$ denote a function which generates all successor states of the state X into a list of successor states L.

5. For a list of states L, let $Evaluate(L, G, G_{max})$ denote a function which computes $G(Y)$ and $G_{max}(Y)$ for each state $Y \in L$.

4.2 The $ID_{G_{max}}$ Search Algorithm

This section illustrates how the $ID_{G_{max}}$ search algorithm explores state space trees. The main search strategy of $ID_{G_{max}}$ is iterative deepening with a search bound based on the non-increasing values of G_{max}. We also introduces several variations on the strategy of relaxing the search bound used by $ID_{G_{max}}$ to further speed up the search.

4.2.1 Variables & Terms

The following introduces several terms, variable names, and notations used in presenting the $ID_{G_{max}}$ algorithm in a pseudocode notation. The high level view of the algorithm requires that we present the overall structure of the search algorithm independent of the implementation details.

- $SearchBound_{G_{max}}$ is the search bound used by the iterative deepening to limit each of the DFS passes to explore states whose $G_{max} \geq SearchBound_{G_{max}}$. $SearchBound_{G_{max}}$ is initially set to $+\infty$ and decreases in each subsequent DFS pass.

- $BestNBuffer$ is a buffer list to hold the "best" N states found. The states are sorted in decreasing order of their actual G scores in this buffer.

- *Root* is the initial state of the state space tree.

- $ObservedG_{max}$ is a data structure in which a DFS pass records observed G_{max} score values less than the current $SearchBound_{G_{max}}$, along with a count of the number of observed states that have each such G_{max} score.

- *CurrentState* is the state of the space currently being expanded. This state is initially the *Root* of the state-space.
- *SuccessorList* is a list of successor states of the *CurrentState*.
- *ListOfStatesToDo* is a list of information indicating which items have been specialized so far. This list represents an agenda by which the algorithm generates desired states eliminating duplicates.
- *LastExpanded* is the number of states expanded in the most recently completed DFS pass.
- *Relaxation_Factor* is a number between 0 and 1 used to control the growth of the search tree based on the *LastExpanded* value. This factor is used in the computation of the amount by which the $SearchBound_{G_{max}}$ is relaxed for the next DFS pass.
- *Threshold* is a number used to impose a threshold on the number of states, from the $ObservedG_{max}$, the DFS is allowed to expand in the next DFS pass. This number is used in the computation of the $SearchBound_{G_{max}}$ for the next DFS pass.

4.2.2 G_{max}-based Iterative Deepening Search $ID_{G_{max}}$

This section presents a pseudocode algorithm description of $ID_{G_{max}}$ from a high level, and then addresses each of the subtasks performed by the iterative deepening process. The most general level of the $ID_{G_{max}}$ search algorithm proceeds as follows, illustrated in Figure 4.1. $ID_{G_{max}}$ search algorithm performs multiple DFS passes with a $SearchBound_{G_{max}}$ starting at $+\infty$ and decreasing in subsequent DFS passes. In every DFS pass. $ID_{G_{max}}$ executes two main operations; first, the function DFS starts a

1. Initialize the $SearchBound_{G_{max}} = +\infty$.
2. While (not($Stop$)) do:
   - $DFS(Root, SearchBound_{G_{max}}, BestNBuffer, ObservedG_{max}, ListOfStatesToDo)$.
   - $Stop = CheckStopping(\ SearchBound_{G_{max}}, BestNBuffer\ )$.
   - $SearchBound_{G_{max}} = Next\_SearchBound_{G_{max}}(\ ObservedG_{max}, LastExpanded)$.

Figure 4.1: The $ID_{G_{max}}$ Search Algorithm

Depth-First search at a $CurrentState = Root$ (initially) in which it generates and evaluates successor states, prunes the list of successor states, updates the list of best N states reached ($BestNBuffer$), and recursively explores states whose G_{max} scores are $\geq SearchBound_{G_{max}}$. In addition, the function $DFS()$ records the G_{max} values it observes when evaluating the generated states. Only G_{max} scores less than the current $SearchBound_{G_{max}}$ are recorded. The $DFS()$ function also records counts of states that have each of such G_{max} scores into $ObservedG_{max}$. Having completed a DFS pass, $ID_{G_{max}}$ sets up for the next DFS pass. $ID_{G_{max}}$ determines whether or not it is safe to terminate execution. This is done by the $CheckStopping()$ function which returns a boolean true value indicating whether or not the set of desired states are reached in the $BestNBuffer$. If not, the algorithm will perform the next DFS pass. Then, $ID_{G_{max}}$ proceeds to its second main operation which is to compute the $SearchBound_{G_{max}}$ to use for the next DFS pass based on information in $ObservedG_{max}$. This computation is performed by the function $Next\_SearchBound_{G_{max}}()$.

1. $Generate(CurrentState, SuccessorList, ListOfStatesToDo)$.
2. $Evaluate(SuccessorList, G, G_{max}, ObservedG_{max})$.
3. while $(not(is\_empty(SuccessorList))$ do:
   - $Prune(SuccessorList, SearchBound_{G_{max}})$.
   - $Prune(SuccessorList, Min_G(BestNBuffer))$.
   - Select $S = Max_{G_{max}}(SuccessorList)$.
   - $Update\_BestN(BestNBuffer, S)$.
   - $Update\_Agenda(ListOfStatesToDo, S)$.
   - $DFS(S, SearchBound_{G_{max}}, BestNBuffer, ObservedG_{max}, ListOfStatesToDo)$.
   - $Remove(SuccessorList, S)$.

Figure 4.2: $DFS()$ Algorithm

A Single DFS Pass

On a more specific level of the IDG_{max} algorithm, we focus on the execution of a single DFS pass which mainly consists of generating, evaluating, and exploring states whose G_{max} scores are less than or equal to the current $SearchBound_{G_{max}}$. The process is a simple Depth-First search technique limited to states that satisfy the $SearchBound_{G_{max}}$ constraint. A single DFS pass is divided into three stages, as shown in Figure 4.2. The first uses a state generation method, $Generate()$ which is appropriate for the search space, to generate successor states of the $CurrentState$ into the $SuccessorList$ according to the state generation agenda, $ListOfStatesToDo$. This state generation agenda is used to direct DFS to generate only desired successor states and eliminate duplications. The idea is to keep track of which items have already been specialized

to generate states. An example of the state generation method is given in the section 2.3. The second stage, *Evaluate*(), evaluates all successor states found in the *SuccessorList* by computing both their G and the G_{max} scores. *Evaluate*() may encounter G_{max} values less than the current $SearchBound_{G_{max}}$. These G_{max} values are inserted into the $ObservedG_{max}$ data structure. In the third stage, where most of the work is performed, $DFS()$ continuously prunes states whose G_{max} scores are less than the $SearchBound_{G_{max}}$ limiting the DFS search to explore states found in the current search bound only. $DFS()$ also prunes states whose G_{max} scores are less than the worse G score in the $BestNBuffer$. This pruning prevents the search from expanding states which are potentially worse or equal to the lowest G-scoring state reached. After pruning, $DFS()$ selects a candidate successor state, S, whose G_{max} score is the highest among all successors for recursive exploration. $DFS()$ updates the $BestNBuffer$ to possibly include S then sets up for the recursive expansion of S by updating the state generation agenda, $ListOfStatesToDo$. Then, $DFS()$ recursively descends into the search tree rooted at S. Once $DFS()$ returns from the recursive call, it removes the state S from the $SuccessorList$. This stage continues until the $SuccessorList$ becomes empty when $DFS()$ backtracks to the parent state. The completion of the current DFS pass occurs when $DFS()$ backtracks past the root of the search tree. At this point, $ID_{G_{max}}$ becomes ready to proceed to the next task of checking the stopping condition.

Checking The Stopping Condition for $ID_{G_{max}}$ Algorithm

A major drawback of using the iterative deepening technique lies in the number of DFS passes performed [22]. If iterative deepening performs at

1. if ($PrevDB_{G_{max}} < Min_G(BestNBuffer)$)) OR (there are no more G_{max} scores left to explore in $Observed_{G_{max}}$) then
   $Stop = TRUE$

Figure 4.3: $CheckStopping()$ - $ID_{G_{max}}$ Stopping Condition

least one unnecessary pass, the performance of the search algorithm can deteriorate significantly because in every DFS pass, iterative deepening expands all states explored in the previous DFS pass, plus new states found in the newly computed search bound. Therefore, we need to ensure that $ID_{G_{max}}$ stops after performing the required number of DFS passes. Illustrated in Figure 4.3 is the stopping condition for $ID_{G_{max}}$ search algorithm. $ID_{G_{max}}$ terminates in one of two situations, first. if the most recently completed DFS pass is known to have been the last DFS pass required. A DFS pass is known to be the last required when, at the end of the DFS pass, the $SearchBound_{G_{max}}$ is less than the lowest G scoring state in the $BestNBuffer$. The analogy follows:

At the end of a DFS pass. if the search tree has been explored to a search bound (based on G_{max}) worse than the lowest quality state found so far, since G_{max} is an upper bound on the G scores of all successors of a states, then there cannot exist any potentially high G scoring states in the remaining portion of the search tree because all the potentially high scoring states have been considered in the most recently completed DFS.

The second case of safe termination occurs when the $Observed_{G_{max}}$ list becomes empty. This condition being true guarantees that there does not exist any potentially interesting fringe states whose G_{max} is less than $Min_G(BestNBuffer)$. In other words, if none of the fringe states survive

1. $Threshold = Relaxation\_Factor * LastExpanded$
2. $SearchBound_{G_{max}} = get\_G_{max}(ObservedG_{max}, Threshold)$

Figure 4.4: $Next\_SearchBound_{G_{max}}()$ Algorithm

the pruning process, then there does not remain any potentially interesting states to explore.

Computing A New $SearchBound_{G_{max}}$ For DFS

Having completed a DFS pass and having determined the necessity of performing a subsequent DFS pass, the $ID_{G_{max}}$ search algorithm prepares for the subsequent DFS pass by computing a new search bound. Different score values of G_{max} are recorded from fringe states. A fringe state is a state which has been generated and evaluated but remains unexplored in the most recently completed DFS pass. The $ObservedG_{max}$ data structure stores the G_{max} scores of such states along with a count of the number of states which had each of the values. Computing a new search bound is simply selecting a new G_{max} score to which the $SearchBound_{G_{max}}$ is set for limiting the subsequent DFS pass. $ID_{G_{max}}$ selects a new G_{max} value for the $SearchBound_{G_{max}}$ based on the $Relaxation\_Factor$ which is a factor of the total number of states expanded in the most recently completed DFS pass.

As shown in Figure 4.4, we first compute the $Threshold$ number as a factor of the number of states explored in the most recently completed DFS pass. The $Relaxation\_Factor$ is a user-supplied number. The $Threshold$ number is used to limit the number of fringe states considered for relaxation of the $SearchBound_{G_{max}}$. Given the $Threshold$ number, we retrieve a

G_{max} value from the $Observed_{G_{max}}$ which guarantees a maximum number of fringe states less than or equal to the *Threshold* number. Then we set the $SearchBound_{G_{max}}$ to the retrieved G_{max} score. When selecting a new G_{max} score to use as $SearchBound_{G_{max}}$, we propose three strategies; the first, the **Conservative** strategy, is more classical and uses the highest observed G_{max} score less than the current $SearchBound_{G_{max}}$. Thus, IDG_{max} search algorithm explores the state space in a decreasing order of G_{max} values, one G_{max} value for each DFS pass. Complementary to the conservative strategy, the second strategy, is the **Exhaustive** strategy which uses the lowest observed G_{max} score less than the current $SearchBound_{G_{max}}$. This strategy causes the next DFS pass to include all the current fringe states in the search tree. The third strategy is the medium of the other two strategies. The **Dynamic** Strategy uses the *Relaxation_Factor*, described above, to limit the number of additional fringe states G_{max} values. The threshold on the number of fringe states to consider is based on a fraction of the number of states expanded in the previous DFS pass. Thus, the IDG_{max} search algorithm explores the state space in a decreasing order of G_{max} score values, but each DFS pass explores many G_{max} values.

4.2.3 Discussion

The idea behind the IDG_{max} algorithm is to use iterative deepening to limit the DFS search to explore states in a search bound based on the sequence of different G_{max} values, $b_0, b_1, \cdots, b_{B-1}$ found in the search space. The IDG_{max} algorithm explores the sets of states $V_0, V_1, \cdots, V_{B-1}$ in a decreasing order of their G_{max} values. For example, for some DFS pass, iterative deepening explores the search space to $SearchBound_{G_{max}} = 10$, and let the observed G_{max} scores of the fringe states, b_i's, be $[8, 6, 5, 3]$. Then, a

conservative search bound strategy decreases the $SearchBound_{G_{max}}$ to the next available G_{max} score value. Therefore, in the subsequent DFS pass, the algorithm sets the $SearchBound_{G_{max}}$ to $b_i = 8$ and proceeds to explore states whose G_{max} scores are greater or equal to 8. An exhaustive search bound strategy sets the new $SearchBound_{G_{max}} = 3$, therefore, it considers all the G_{max} values of the fringe states. Or alternatively, a dynamic search bound strategy sets the new search bound based on the number of states expanded so far. Let's say it sets $SearchBound_{G_{max}} = 5$. Then, in the subsequent DFS pass the search will explore states whose G_{max} values are greater or equal to 5. This process of iterating the DFS and computing new search bounds continues until the set of desired states is reached. Ideally, we desire a balance between the growth in the search space and the gain of exploring states which adds to the work-load of the search algorithm. On the one hand, performing a single DFS pass with a small amount of gain, i.e. exploring few new states due to an insufficient search bound relaxation which causes iterative deepening to perform more DFS passes. When iterative deepening performs more DFS passes, the algorithm expands and generates more states because subsequent DFS passes re-explore states which have been explored in previous DFS passes plus states in the new search bound. This additional cost of re-exploration can add up to a significant increase in the overall work-load.

On the other hand, relaxing the search bound by a large amount could cause a significant increase in the time requirements for a DFS pass to complete, especially when the search space suffers from an explosion in the number of states, a drawback of Depth-First search. Therefore, we need to set the $SreachBound_{G_{max}}$ such that we achieve a balance between the cost of exploring the state space and the gain acquired in each DFS pass. In

other words, we desire to explore the least number of states to find the most desired states consuming the least time and space. The $ID_{G_{max}}$ algorithm attempts to balance this aspect of the search by limiting the relaxation of the search bound to a fraction of the number of states explored in the previous DFS pass. However, the following illustrates a limitation of our three search bound strategies using the previous example. Suppose that the $SearchBound_{G_{max}} = 10$ for a given DFS pass. After the completion of this DFS pass, the search would have explored states whose $G_{max} \geq 10$. If the observed G_{max} values of the fringe states are $[8, 6, 5, 3]$, then, in the next DFS pass, a conservative search bound strategy chooses $G_{max} = 8$, an exhaustive search bound strategy chooses $G_{max} = 3$, and a dynamic strategy chooses any value from $[8, 6, 5, 3]$ based on a fraction of the number of states expanded so far. All three strategies, in this case, cause the next DFS pass to include all states whose G_{max} values are between 10 and 8. However, there may exist an explosive number of states whose G_{max} values are equal to 9. This can cause the performance of $ID_{G_{max}}$ to suffer dramatically. This limitation is due to the fact that we cannot discover all of the G_{max} values in earlier DFS passes because G_{max} is an estimated upper bound on G.

In addition, we showed, in Section 3.4.6 on page 50, that the worst case performance for iterative deepening occurs when each of the states has a unique G_{max} value. Therefore, the list of $ObservedG_{max}$ values can grow exponentially when the *heuristic branching factor* is close to 1. However, this is also the worst case performance for the iterative deepening where it asymptotically expands more states than BFS. This case points out the worst case performance of $ID_{G_{max}}$ which has been shown to be $O(X_{BFS}^2)$, where X_{BFS} is the number of states expanded by Best-First Search.

4.3 Chapter Summary

This chapter presented the $ID_{G_{max}}$ search algorithm which uses iterative deepening with a search bound strategy based on a heuristic upper bound (G_{max}) on the state quality (G). The $ID_{G_{max}}$ explores the search space in a decreasing order of these G_{max} values for each of the DFS passes. The amount by which the search bound is decreased for each of the DFS passes can be computed by three different strategies, the conservative, the dynamic, and the exhaustive strategies. The following chapter presents experimental results of comparing the performance of these three search bound strategies against each other and against our implementation of the MSDD search algorithm.

Chapter 5

Experimental Results

This chapter presents the experimental results obtained in this research work. The presentation starts by presenting a study statement followed by a section to motivate our experimental objectives. Before presenting the experimental results, we list and describe datasets we use throughout the experiments, then, the experimental results are presented and discussed for each of the experiments.

5.1 Study Statement, A Comparative Study

This section gives a brief overview of our studies and experimental results. These experiments are described in detail in subsequent sections of this chapter. Recall, the $ID_{G_{max}}$ search algorithm is an iterative deepening search algorithm searching for the best N states in a tree-based search space. The $ID_{G_{max}}$ search algorithm uses the G_{max} values as basis for the search bound strategy. The $ID_{TreeDepth}$, an implementation of the MSDD algorithm [31], uses the depth of states in the search tree as basis for the search bound strategy employed by iterative deepening. The main

objective of our study is to compare the effectiveness of using the three search bound relaxation strategies (the **Conservative**, the **Dynamic,** or the **Exhaustive** strategy) against each other and against the $ID_{TreeDepth}$ algorithm. The performance is measured based on the evaluation criterion presented in section 3.2.

5.2 Objectives & Motivation

This section explains the objectives and motivation of each of the experiments conducted in this thesis. In addition, this section states some expectations of the experimental results. There are three main motivations for our experiments. First, we conduct an experiment to confirm the proper implementation of the $ID_{G_{max}}$ search algorithm. The results of this experiment will show that the $ID_{G_{max}}$ search algorithm finds the best N solution states in a tree-based search space. In addition, we perform a comparison experiment between the existing iterative deepening search technique, $ID_{TreeDepth}$ and the $ID_{G_{max}}$ search algorithm on small size search spaces. The purpose of this experiment is to show an improved performance of the search algorithm by using our modified versions of the search bound strategies. In addition, the experiment will show how these various strategies of relaxing the $SearchBound_{G_{max}}$ measure against the $ID_{TreeDepth}$ search algorithm. The second experiment compares the different search bound strategies (the conservative, the dynamic, and the exhaustive) and the $ID_{TreeDepth}$ algorithm on larger search spaces.

Our experimental results will show a significant improvement in most of the search algorithm evaluation criteria. Theoretically, we expect the conservative search bound relaxation strategy to cause the $ID_{G_{max}}$ to per-

form more DFS passes resulting in unnecessary computations. This is even more apparent for datasets where there are many G_{max} values that are close to each other and each has fewer states, this property is known to worsen the performance of iterative deepening. In contrast, the exhaustive strategy can exhibit better performance, however, the performance of this exhaustive strategy is expected to deteriorate when faced with an explosive number of states in the new search bound. This deficiency is a consequence of the exhaustive strategy considering all the observed G_{max} values of the fringe states which can contain an exponential number of states in their subtrees. Our proposed solution is to use the dynamic search bound relaxation strategy which takes into account, to some extent, the number of states found in each of the G_{max} bounds. This approach controls the relaxation of the search bound such that the algorithm may find a suitable balance between the number of states explored in each DFS pass and the gain of exploring new states. The motivation is to ensure that iterative deepening performs an appropriate workload to find the set of N desired states while, possibly, avoiding explosive portions of the search space.

5.3 Datasets Used

In this section we present an overview of datasets used in our experiments. For each of these datasets, we discuss their characteristics expected to influence the outcome of the experiments.

5.3.1 Dataset I: The *Software Development* Dataset

Obtained from *Revenue Canada*, the dataset contains records of software applications development. The idea is that software development occurs

by assembling new software components from an existing set of components. The dataset records instances of software development by indicating which software components (an attribute) are being used in each of the application development records. table 5.1 shows samples of such software development records. Each column represents each of the existing software components (C_i) available to programmers. Each row represents an instance of an application (App_i) development. The data is kept in a binary representation. The presence of a software component C_j in an application App_i is indicated by a 1 in the corresponding column. A column entry of 0 indicates the absence of the component C_j from the App_j instance.

Component	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	...	C_{81}
App_1	0	0	1	1	1	1	1	1	1	0	...	...
App_2	0	0	0	1	1	0	1	0	0	0	...	...
App_3	0	0	1	1	1	1	1	1	1	0	...	...
App_4	0	0	1	1	1	1	1	1	1	0	...	...
App_5	1	0	0	0	1	1	0	1	1	1	...	...
App_6	1	0	0	0	1	1	0	1	1	1	...	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮...	...
App_{38}	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮...	...

Table 5.1: The Binary *Software Development* Dataset

The dataset contains 81 different software components, i.e. 81 columns. The total number of records is 38. Unfortunately, the number of records is small. However, it seems to be sufficient to present interesting characteristics for our experiments. Mining this dataset produces relationship rules which, when evaluated by the G measure, express dependencies between different sets of software components used in software development. For

example, consider the following relationship rule:

$$C_3, C_5 \Rightarrow C_{20}, C_{34}$$

Having been evaluated by the G measure, this relationship rule states that the presence of software components found in the RHS in an application is dependent on the presence of software components found in the LHS. The strength of this dependency is expressed by the G score computed for this relationship rule. Thus, interpreting this dependency in the context of developing new software applications is: the inclusion of C_3 and C_5 indicates that the probability of also including components C_{20} and C_{34} increases significantly.

5.3.2 Dataset II: The *Flight Errors* Dataset

This dataset is a real life dataset obtained from *Air Canada* airlines, thanks to *Rob Wylie* of the *CNRC*. The data contains records collected from flights performed using *AirBus* aircrafts. The data is a collection of failure messages recorded during many flights of several aircrafts. These messages are logged by an on-board computer which monitors different components of the aircraft. A flight record in the dataset presents a list of failure messages reported by several system units during a flight. In addition, system units are designed in a particular hierarchy, known as the *ATA Hierarchy*, an industry standard decomposition of an aircraft into sub (and sub-sub and sub-sub-sub) systems. Table 5.2 shows a sample list of system units found at the highest level of the *ATA* hierarchy. Given the structured design of the system units, we may be interested in relationships between failure messages being reported from different system units.

Bit Number	Attribute Description
1	AIR CONDITIONING
2	AUTO FLIGHT
3	COMMUNICATIONS
4	ELECTRICAL POWER
5	EQUIPMENT FURNISHINGS
6	FIRE PROTECTION
7	FLIGHT CONTROLS
8	FUEL
9	HYDRAULIC POWER
10	ICE AND RAIN
11	INDICATING RECORDING SYS
12	LANDING GEAR
13	LIGHTS
14	NAVIGATION
15	OXYGEN
16	PNEUMATIC
17	WATER WASTE
18	AIRBORNE AUXILIARY POWER
19	DOORS
20	STD PRACTICES - ENGINE
21	POWER PLANT
22	ENGINE TURBINE TURBOPROP
23	ENGINE FUEL CONTROL
24	IGNITION
25	AIR
26	ENGINE CONTROLS
27	ENGINE INDICATING
28	EXHAUST
29	OIL
30	STARTING

Table 5.2: The *Flight Errors* Attributes - *ATA* Top Level

These units should be on the same level of the *ATA* hierarchy. This representation of the data allows the data mining search algorithm to discover relationship rules between subsets of these system units reported failures. These relationship rules may be used to perform diagnostics on the aircraft. Or, these relationship rules may be used to confirm that the sequence of events when failures occur comply with the design of the aircraft. We can build a binary dataset as shown in Table 5.3, every row represents an instance of a flight. A column represents a system unit, as described in the *ATA* hierarchy design. All the units are on the same level with respect to the *ATA* hierarchy design. An entry of 1 in a row indicates that the corresponding column unit has reported at least one failure message in this particular flight. In our experiment, we built several such datasets on different levels of the *ATA* hierarchy because one might consider searching for relationship rules within a unit (between the sub-units) to determine some characteristics of a smaller scope of failures. The highest level dataset included 30 units (bits) and 43941 test flights (records).

Bit #	1	2	3	4	5	6	7	8	9	10	11	12	···	30
$Flight_1$	1	1	0	0	0	0	0	0	0	1	1	1	···	1
$Flight_2$	0	1	1	0	0	0	0	0	0	0	1	0	···	0
$Flight_3$	1	1	0	0	0	0	0	0	0	0	1	0	···	1
$Flight_4$	0	0	0	0	0	0	0	0	0	0	1	1	···	0
$Flight_5$	0	1	1	1	0	1	0	1	0	0	1	0	···	0
$Flight_6$	0	0	0	0	0	0	1	0	0	0	1	0	···	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 5.3: The Binary *Flight Errors* Dataset

A discovered relationship rule from this dataset may be:

INDICATING RECORDING SYSTEM ⇒ NAVIGATION, WATER WASTE

When the G measure is used for rule evaluation, the above relationship rule is interpreted as: the probability of the combined failure of the *WATER WASTE* and the *NAVIGATION* units increases significantly after observing the failure of the *INDICATING RECORDING SYSTEM* unit. The strength of this dependency is reflected by the G value of this relationship rule.

5.3.3 Dataset III: The *U. S. Congressional Voting* Dataset

Obtained from the *UCI* Machine Learning Repository, this dataset contains 435 instances of congressmen voting on 16 different bills. In addition, the congressman's party (*Republican* or *Democrat*) is also recorded. We use a binary representation of this dataset whose attributes have multiple values (*yes* or *no*). The method to represent multi-valued attributes is described in section 2.1.4. Using such a binary representation, table 5.4 shows a list of attributes and their binary representation for this dataset. Each row in the table represents a column attribute used in the dataset.

Bit Number	Attribute Description = Value
1	handicapped infants = no
2	handicapped infants = yes
3	water project cost sharing = no
4	water project cost sharing = yes
5	adoption of the budget resolution = no
6	adoption of the budget resolution = yes
7	physician fee freeze = no
8	physician fee freeze = yes
9	el salvador aid = no
	(continued on the next page)

Table 5.4: The *Congressional Voting* Attributes

Bit Number	Attribute Description = Value
	(continued from the previous page)
10	el salvador aid = yes
11	religious groups in schools = no
12	religious groups in schools = yes
13	anti satellite test ban = no
14	anti satellite test ban = yes
15	aid to nicaraguan contras = no
16	aid to nicaraguan contras = yes
17	mx missile = no
18	mx missile = yes
19	immigration = no
20	immigration = yes
21	synfuels corporation cutback = no
22	synfuels corporation cutback = yes
23	education spending = no
24	education spending = yes
25	superfund right to sue = no
26	superfund right to sue = yes
27	crime = no
28	crime = yes
29	duty free exports = no
30	duty free exports = yes
31	export administration act south africa = no
32	export administration act south africa = yes
33	political party = democrat
34	political party = republican

Table 5.4: The *Congressional Voting* Attributes

Discovering interesting relationships (evaluated by G) may provide very useful information about dependencies between voting responses. For ex-

ample, consider the following relationship rule:

physician fee freeze = yes, el salvador aid = yes $\Rightarrow$ political party = republican

This G-evaluated relationship rule expresses the dependency that if a congressman votes *yes* on the bills *physician fee freeze* and *el salvador aid*, then the probability of him being a *republican* increases significantly.

5.3.4 Dataset VI: The *Solar Flare* Dataset

This dataset is also obtained from the UCI Machine Learning Repository and contains 323 instances of features captured for one active region on the sun. Table 5.5 shows a list of features used as attributes for each of the solar flares recorded. In the binary representation of data, each of the attributes has several values. Therefore, we use a single bit to represent the presence of each of the values for each of the attributes. For example, the attribute *class code* has 6 possible values, we use bits 1 through 6 to represent each value as shown in Table 5.5. The data contains instances for solar flares where the features described are recorded. A relationship rule miner can search for the strongest relationships between different feature sets. An example of such a relationship rule is:

largest spot size code = X,
did region become historically complex on this pass across the sun's disk = no,
C-class flares production by this region in following 24hrs (severe flares) = 0
$\Rightarrow$
class code = B

This relationship rule lists some of the conditions the *class code* = *B* depends on. When G is used as the rule evaluation measure, the above relationship rule states that the probability of the *class code* = *B* increases

significantly when observing the combinations of feature values which appear in the LHS of the dependency rule. The nature of the relationship is a dependency measured by the G score of the rule.

Bit Number	Attribute Description = Value
1	class code = G
2	class code = F
3	class code = E
4	class code = D
5	class code = C
6	class code = B
7	largest spot size code = X
8	largest spot size code = S
9	largest spot size code = R
10	largest spot size code = K
11	largest spot size code = H
12	largest spot size code = A
13	spot distribution code = C
14	spot distribution code = X
15	spot distribution code = O
16	spot distribution code = I
17	activity = unchanged
18	activity = reduced
19	evolution = growth
20	evolution = no growth
21	evolution = decay
22	prev. 24hrs flare activity = 1
23	prev. 24hrs flare activity = 3
24	historically-complex = no
25	historically-complex = yes
26	did region become historically complex
	on this pass across the sun's disk = no

(continued on the next page)

Table 5.5: The *Solar Flare* Attributes

Bit Number	Attribute Description = Value
	(continued from the previous page)
27	did region become historically complex on this pass across the sun's disk = yes
28	area = large
29	area = small
30	area of the largest spot > 5
31	area of the largest spot <= 5
32	C-class flares production by this region in following 24hrs (common flares) = 2
33	C-class flares production by this region in following 24hrs (common flares) = 1
34	C-class flares production by this region in following 24hrs (common flares) = 0
35	M-class flares production by this region in following 24hrs (moderate flares) = 4
36	M-class flares production by this region in following 24hrs (moderate flares) = 2
37	M-class flares production by this region in following 24hrs (moderate flares) = 1
38	S-class flares production by this region in following 24hrs (moderate flares) = 0
39	S-class flares production by this region in following 24hrs (severe flares) = 1
40	S-class flares production by this region in following 24hrs (severe flares) = 0

Table 5.5: The *Solar Flare* Attributes

5.3.5 Dataset V: The *Contact Lenses* Dataset

This dataset was also obtained from the UCI Machine Learning Repository. The dataset contains instances to describe the factors affecting the decision

to fit a patient with a type of contact lenses, if any. The dataset contains 24 instances of 4 multi-valued attributes along with 3 class labels being *fit with hard contact lens. fit with soft contact lens*, and *should not be fitted with any contact lenses.* The attributes and their values are described in Table 5.6 . Similar to the binary representation of the previous datasets, we use a single binary bit to represent each of the values for each of the attributes.

Bit Number	Attribute Description = Value
1	age of the patient = presbyopic
2	age of the patient = pre-presbyopic
3	age of the patient = young
4	spectacle prescription = hypermetrope
5	spectacle prescription = myope
6	astigmatic = yes
7	astigmatic = no
8	tear production rate = normal
9	tear production rate = reduced
10	the patient should not be fitted with contact lenses
11	the patient should be fitted with soft contact lenses
12	the patient should be fitted with hard contact lenses

Table 5.6: The *Contact Lenses* Attributes

A sample relationship rule which may be found from this dataset is:

astigmatic = no. tear production rate = normal

$$\Rightarrow$$

the patient should be fitted with soft contact lenses

Evaluating this relationship rule by the G measure expresses the dependency of the probability of *fitting a patient with soft contact lenses* on the feature values appearing in the LHS. The interpretation of this rule is

that the probability of *fitting a patient with soft contact lenses* increases significantly when the conditions appearing in the LHS are observed.

This particular example is useful to explain the difference between the G measure of dependency and some other measures. For instance, for some rule evaluation measure, this relationship rule may mean that if the conditions of the LHS are observed, then the patient should be fitted with soft contact lenses. However, for the G measure, this is not a correct interpretation. When using the G measure, the interpretation of the relationship rule is that if the conditions of the LHS are observed, then the probability of having to fit the patient with soft contact lenses increases significantly. For example, if 10% of the overall population of patients are fitted with soft contact lenses, however, in the sub-population which satisfy the LHS conditions, only 35% are fitted with soft contact lenses. The above dependency rule does not mean that a patient who satisfies the observed conditions of the LHS should be fitted with soft contact lenses, instead, the probability of the patient having to be fitted with soft contact lenses increases significantly. It may be the case that for some patients, the best treatment is fitting them with hard contact lenses. This dataset is particularly interesting because it contains all possible combinations of attribute values. This is a characteristic which might affect the performance of the rule mining algorithm.

5.4 Experiment I: $ID_{G_{max}}$ Performance - Small Scale

In this section we present the first of our experiments. In this experiment we compare relationship rules produced by all three implementations of the $ID_{G_{max}}$ algorithm and by the implementation of the $ID_{TreeDepth}$ algorithm

to those produced by a program which exhaustively generates all possible relationship rules in the search space. This provides confirmation that, on the datasets used, the implementations correctly find the best N rules. It also invites a discussion of the comparative performance of all four different strategies of iterative deepening search algorithm. We begin by presenting an analysis of the characteristics of datasets used for this experiment. The purpose is to understand the influence of search space properties on the performance of each of the iterative deepening search algorithms. Then, we present a discussion of our experimental results.

5.4.1 Characteristics of The Datasets

This section presents an overview of the several dataset characteristics we consider when interpreting our experimental results. Such characteristics may help us understand properties of different domains where we apply the search algorithms. We first consider the number of states that have $G > 0$ score in the search space. This number represents the number of possibly desired states. Second, we consider the number of different G_{max} values in the search space. This number can reflect some properties of the branching factor of the search space. Third, we count the number of states that have each of the G_{max} values in the search space. This property may indicate a possible explosion in the search space. In addition, we record the number of records and the total number of possible items present in the datasets. Table 5.7 shows these properties of datasets we use in this experiment. The table records in its columns the number of records of the dataset, the number of bits used to represent the dataset, the number of unique G_{max} values, the number of states whose $G_{max} > 0$, the number of states whose $G > 0$, and the number of states whose $G_{max} = G$ respectively.

Dataset	Records	Bits	Unique G_{max}	Rules $G_{max} > 0$	Rules $G > 0$	Perfect G_{max}
Software Development (Bits 1 - 10)	38	10	40	15838	15071	1685
Software Development (Bits 21 - 30)	38	10	24	8215	7422	321
Congressional Voting (Bits 1 - 10)	435	10	528	779	516	0
Congressional Voting (Bits 11 - 20)	435	10	510	814	563	0
Congressional Voting (Bits 21 - 30)	435	10	557	810	515	0
Contact Lenses	24	12	28	1174	518	9

Table 5.7: Analysis of Selected Portions of Datasets

Since most of these characteristics consider states in the set of all possible states of the search space, this type of analysis is possible only on small size datasets which have fewer number of attributes and values. Using a binary representation of a given dataset which contains K binary bits, the presence of a given binary bit in a relationship rule has three possibilities, the bit may appear on the LHS, on the RHS, or may not appear in the rule at all. Therefore, the total number of possible states (rules) in the search space is 3^K. For example, the number of all possible relationship rules for the *Contact Lenses* data set which contains 12 bits is $3^{12} = 531441$ rules. We compute the different properties of datasets by using a program to generate all relationship rules exhaustively. However, due to the exponential nature of the search space, our analysis is limited to approximately 10 bits to keep the execution time reasonable. For this experiment, we use the *Contact Lenses* dataset (12 bits and 531441 rules)

along with selected 10-bit groups from each of the *Software Development*, and *US Congressional Voting* datasets. However, considering only a subset of the bits from a dataset can potentially eliminate many relationship rules from the search space which may be stronger and more specialized. The selection of such groups of bits is based on observed characteristics of the search space. The aim is to conduct performance evaluation experiments on a controlled or previously known search spaces to understand the behavior of the search algorithms.

The *Software Development* Dataset

The first group of bits $(1-10)$ of the *Software Development* dataset have a high number of potentially desired states whose $G_{max} > 0$. The heuristic use of the G_{max} reduces the number of potentially desired states to 15838 which is $\frac{15838}{3^{10}} \approx 27\%$ of the entire search space. Table 5.7 shows that 95% of these potentially desired states have a $G > 0$ and that G_{max} was a perfect upper bound $\frac{1685}{15838} \approx 11\%$ of the time. Therefore, we expect a search algorithm to perform a lengthy search due to the high number of states a search algorithm needs to explore. Figure 5.1(A) on page 93 shows the number of states which have each of the G_{max} score values. We observe several highly populated G_{max} score values scattered over the G_{max} range. The heavily populated G_{max} scores are $5-10$, $15-20$, $30-35$, and $45-50$. This portion of the *Software Development* dataset is characterized by having a relatively low number of different G_{max} values, however, few of these G_{max} values have a high number of states. Also, the sequence of the heavily populated G_{max} values is discontinuous, i.e. scanning the search space based on a decreasing G_{max} value faces high concentrations of states followed by lower concentration of states. Figure 5.1(B) shows

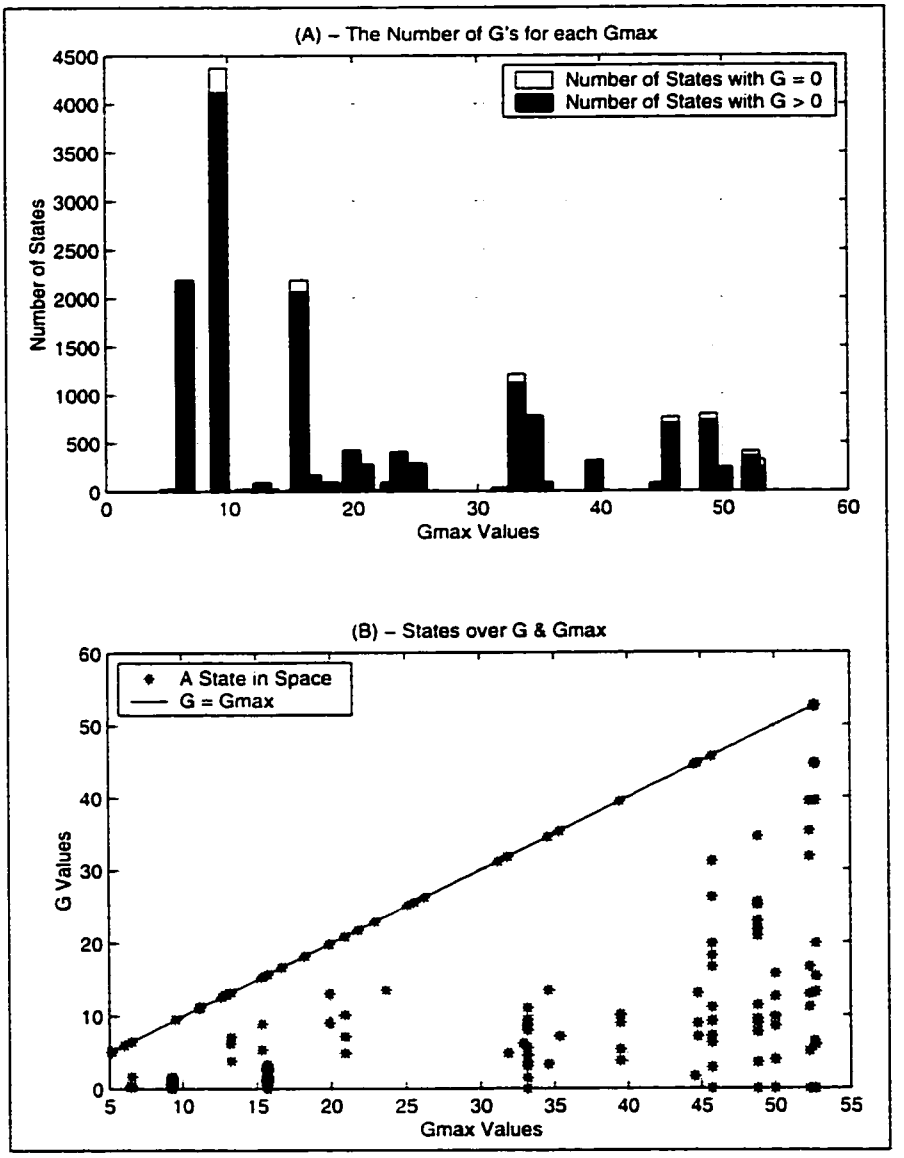

Figure 5.1: The *Software Development* Characteristics (bits 1 – 10)

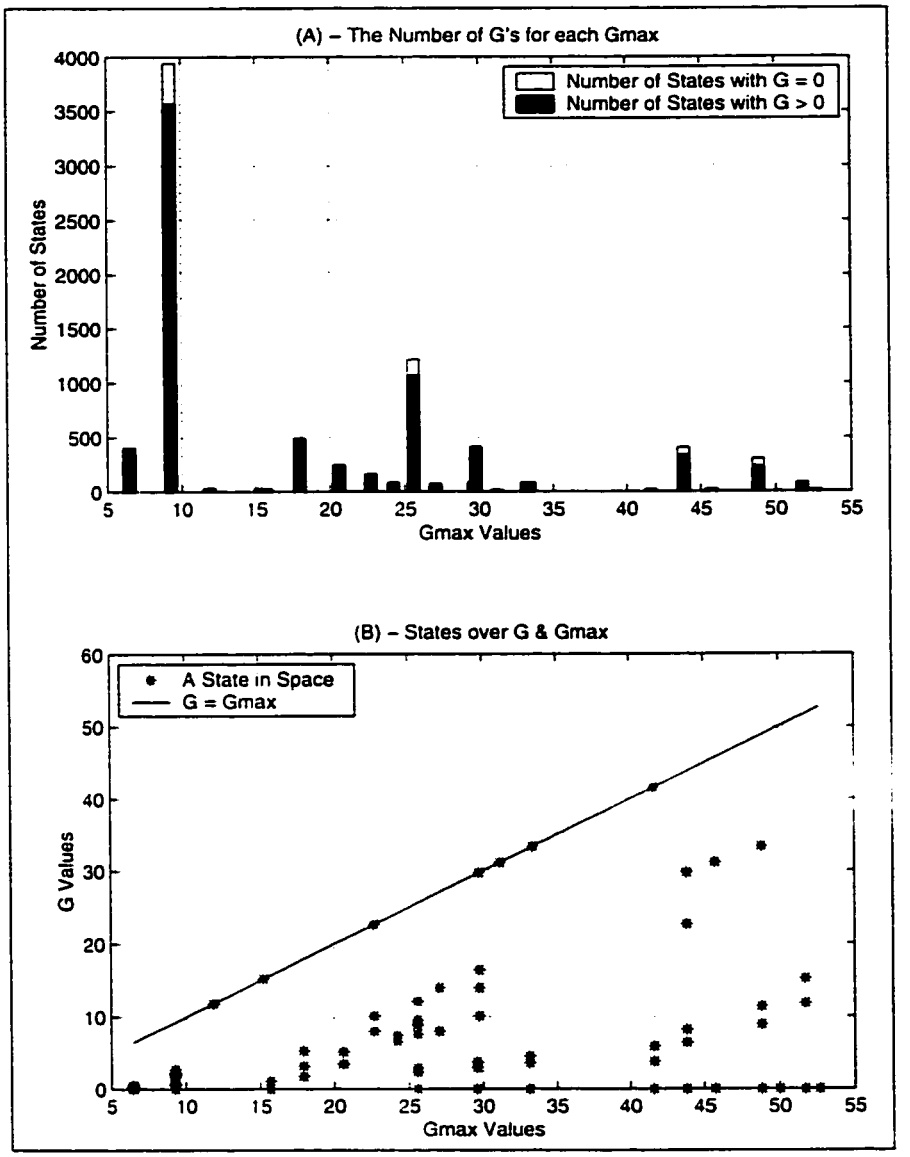

Figure 5.2: The *Software Development* Characteristics (bits 21 – 30)

how the G_{max} scores of all states related to their G score. Many states lie on the diagonal line where $G = G_{max}$ values. This indicates a perfect G score estimate by the G_{max}. However, more states seem to be scattered throughout the entire space with many being incorrectly estimated by G_{max} to have a high G values. Such states are found in the bottom right corner of the figure.

The second group of the *Software Development* dataset, bits $(21 - 30)$, also have a high number of potentially desired states, i.e. whose $G_{max} > 0$. This number of potentially desired states is $\frac{8215}{3_{10}} \approx 14\%$ of the entire search space, 90% of these states have $G > 0$ and G_{max} was perfectly accurate $\frac{321}{8215} \approx 4\%$ of the time. Characteristics of this second group are shown in Figure 5.2 on Page 94. The dataset is similar to the previous dataset where the number of different G_{max} score values is low and the overall number of states with $G_{max} > 0$ is high. In addition, we find many ranges of G_{max} score values where we find high populations of states, mainly the ranges $40 - 50$, $25 - 30$, $15 - 20$, and the most populated is $5 - 10$. Table 5.7 on page 91 shows that this state space contains fewer states than that of the first group of bits and the G_{max} values have similar properties to those of the first group of bits with fewer states where $G = G_{max}$. This group has a single concentration of states of equal G_{max} values $(5 - 10)$ which contains approximately 40% of the potentially desired states.

The *U.S. Congressional Voting* Dataset

Table 5.7 shows the statistics numbers for all of the three portions of the *U.S. Congressional Voting* dataset. The number of potentially desired states is approximately 1.4% of the entire search space. A healthy number of states have $G > 0$, approximately 66% of the states whose $G_{max} > 0$

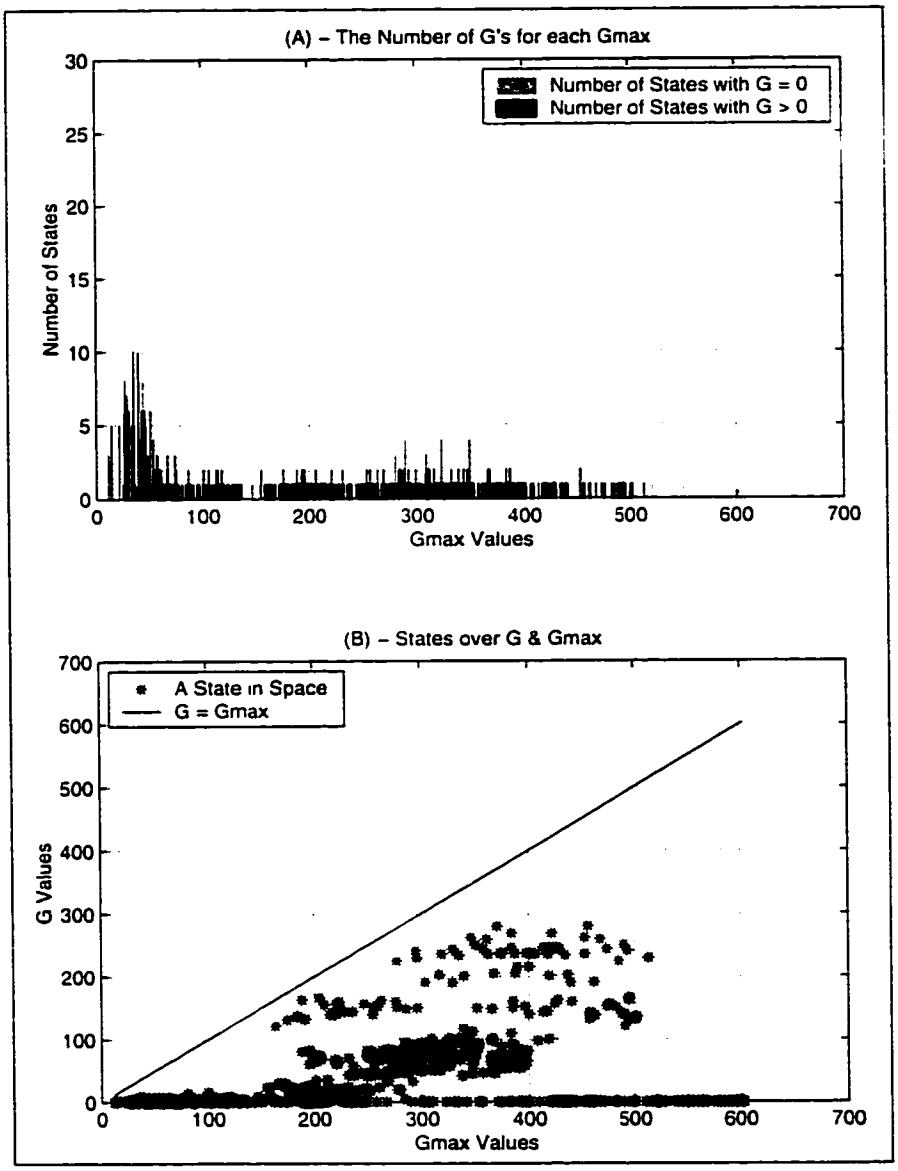

Figure 5.3: The *Congressional Voting* Characteristics (bits 11 – 20)

have $G > 0$, but G_{max} is never accurate in estimating an upper bound on G, i.e. none of the states have $G = G_{max}$. The three selected groups from this dataset exhibit different properties than those of the *Software Development* dataset. We analyzed all three portions of this dataset (bits $1-10$, $11-20$, and $21-30$). However, we only show the analysis of group $(21-30)$ because all three groups exhibited the same space structure. Figure 5.3(A) shows that states are scattered over all the G_{max} values. The portions of this dataset have high numbers of different G_{max} values that are very close in value. This suggests that a search algorithm, using the G_{max} values as search bounds, must explore many G_{max} bounds to discover desired states in the search space. The number of states in any G_{max} value seems to be relatively low when compared to those of the *Software Development* datasets. Unlike the *Software Development* dataset, the number of states for each of the G_{max} scores seems to increase fairly smoothly as the G_{max} value decreases. Figure 5.3(B) shows a high numbers of states in the search space that have an actual $G = 0$ or close to 0 while the G_{max} upper bound is much higher. This suggests that the G_{max} upper bound estimates are much higher than the actual G values in this dataset, i.e. no states are found on the diagonal line where $G = G_{max}$ scores. This observation hints that the G_{max} upper bound of the G state evaluation function does not perform well in this problem domain. Therefore, using this dataset may provide additional understanding of how search algorithms react to a failing G_{max}.

The *Contact Lenses* Dataset

As illustrated in Table 5.7, the G_{max} heuristic reduces the number of potentially desired states whose $G_{max} > 0$ to 0.22% of the entire search space.

This dataset has the lowest ratio, 44% of states whose $G > 0$ to those whose $G_{max} > 0$ which means that this dataset is moderately rich in states that are expected to have high G scores. Moreover, G_{max} seems to be imperfect in estimating an upper bound on G, only 9 states have $G = G_{max}$. Due to the small number of attributes and values, we are able to use the entire dataset represented using 12 binary bits to perform this analysis. Unlike the other datasets, the *Contact Lenses* dataset possesses the interesting property that it contains all possible combinations of all attribute values. Figure 5.4(A) on page 99 shows that the number of different G_{max} values is low compared to those of the *Congressional Voting* dataset. Also, the G_{max} scores are spread further than the previous datasets. Figure 5.4(B) on page 99 presents the distribution of states over the ranges of G and G_{max}. This dataset contains few strong rules where G_{max} was capable of estimating their G scores. The overall distribution of states indicates that G_{max} succeeds in isolating few stronger states from other ones.

5.4.2 Experimental Results I: Discussion

In this section, we present the results of this experiment for each of the datasets listed in Table 5.7. The set up for this experiment is straight forward, we performed two different executions of all four implementations of the rule mining algorithms on the datasets. In the first run, we fixed the value of the number of desired rules (the value of N) and varied the *Relaxation_Factor*. In this case N is limited to a maximum value of the total number of states whose $G > 0$ shown in Table 5.7. For the second run, we fixed the value of the *Relaxation_Factor* to a selected value and varied the value of N. We selected the best performing *Relaxation_Factor* value from the first run. We used an *Ultra Sparc 10* UNIX machine by *Sun*

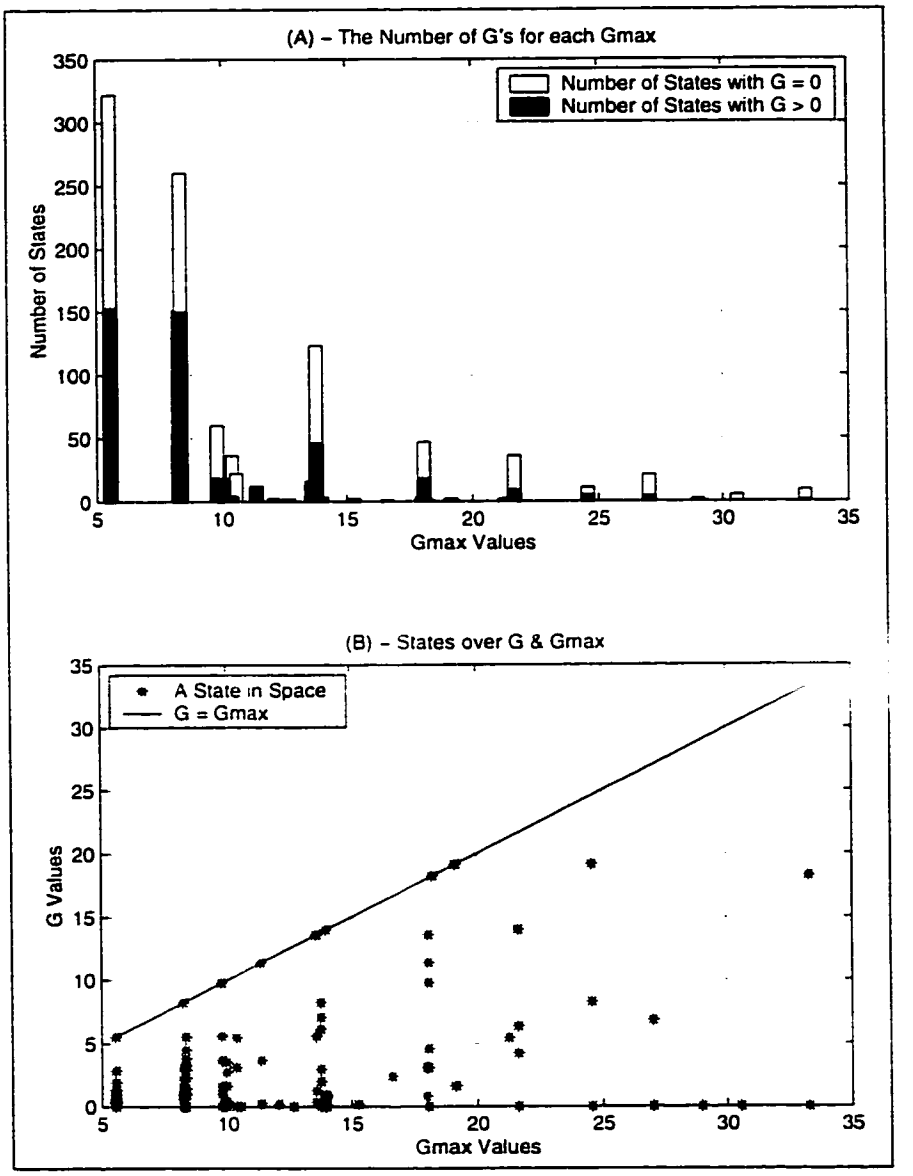

Figure 5.4: The *Contact Lenses* Characteristics (all 12 bits)

Microsystems running Solaris OS. For each of the datasets, we show two figures, the first figure plots the CPU times (part (A)), number of states expanded (part (B)), and number of states generated (part (C)) for each of the search algorithms when N is fixed and the *Relaxation_Factor* is varied. The second figure plots the same three measurements when the *Relaxation_Factor* is fixed and N is varied.

This experiment verifies the implementations of all four algorithms. Relationship rules produced by all four implementations were found to be identical to those produced by exhaustively generating all possible relationship rules in the search space. This confirms that all four implementations of the rule mining algorithms are capable of extracting any rules from any of the datasets used. In addition, with respect to the performance of each of the four algorithms, the results are very encouraging.

The *Contact Lenses* Results

Figure 5.5 on page 101 shows the experimental results obtained for a fixed value of $N = 519$ (all states whose $G > 0$) and various values of the *Relaxation_Factor* (used only by the $Dynamic\_ID_{G_{max}}$). We observe that the $Conservative\_ID_{G_{max}}$ algorithm requires the most CPU time, expands and generates the most states. This poor performance is expected since the iterative deepening is forced to perform a new DFS pass for each of the G_{max} score values exploring new states as well as previously explored states. Figure 5.4(A) showed that these G_{max} values contain very few states each of which result in very little gain in each of the DFS passes performed by $Conservative\_ID_{G_{max}}$. The $ID_{TreeDepth}$ algorithm has the second worst performance. Its performance is significantly better than that of the $Conservative\_ID_{G_{max}}$, however, it is consistently outperformed

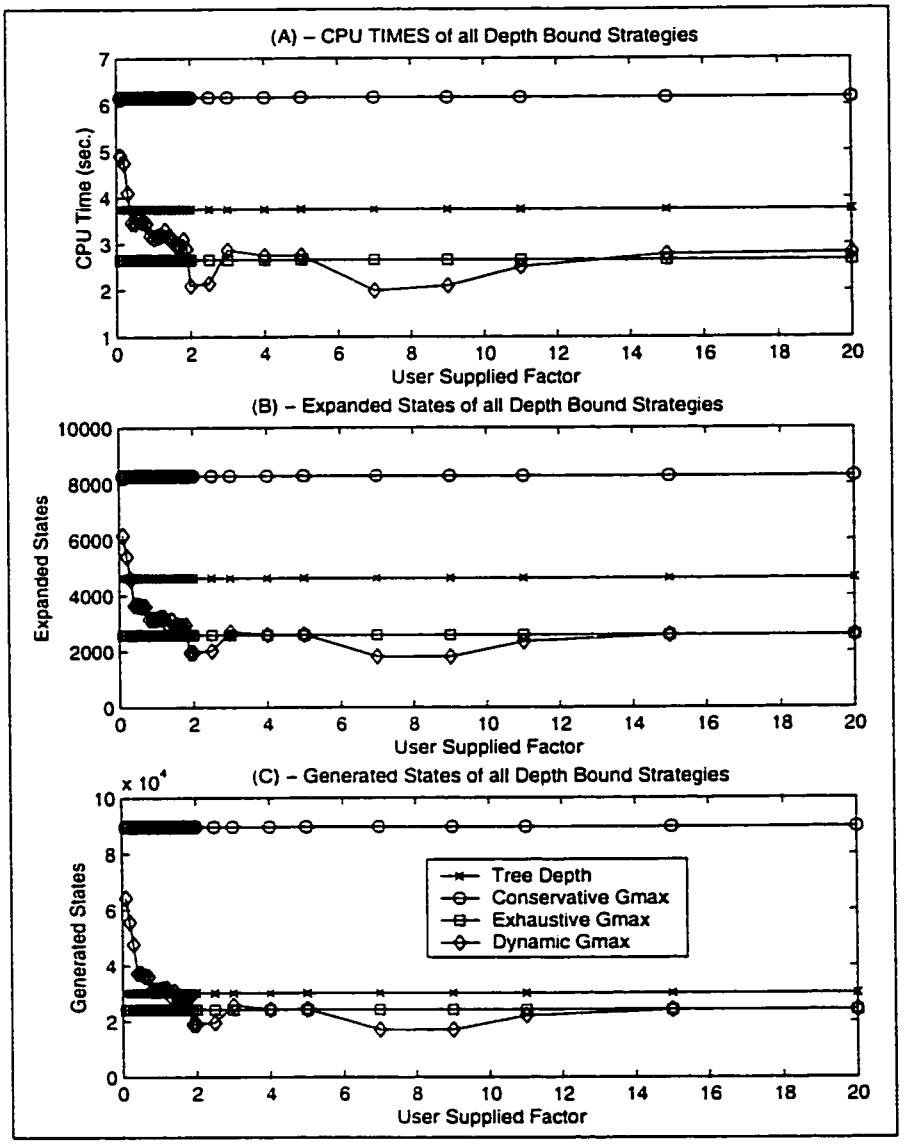

Figure 5.5: The *Contact Lenses* Results ($N = 519$)

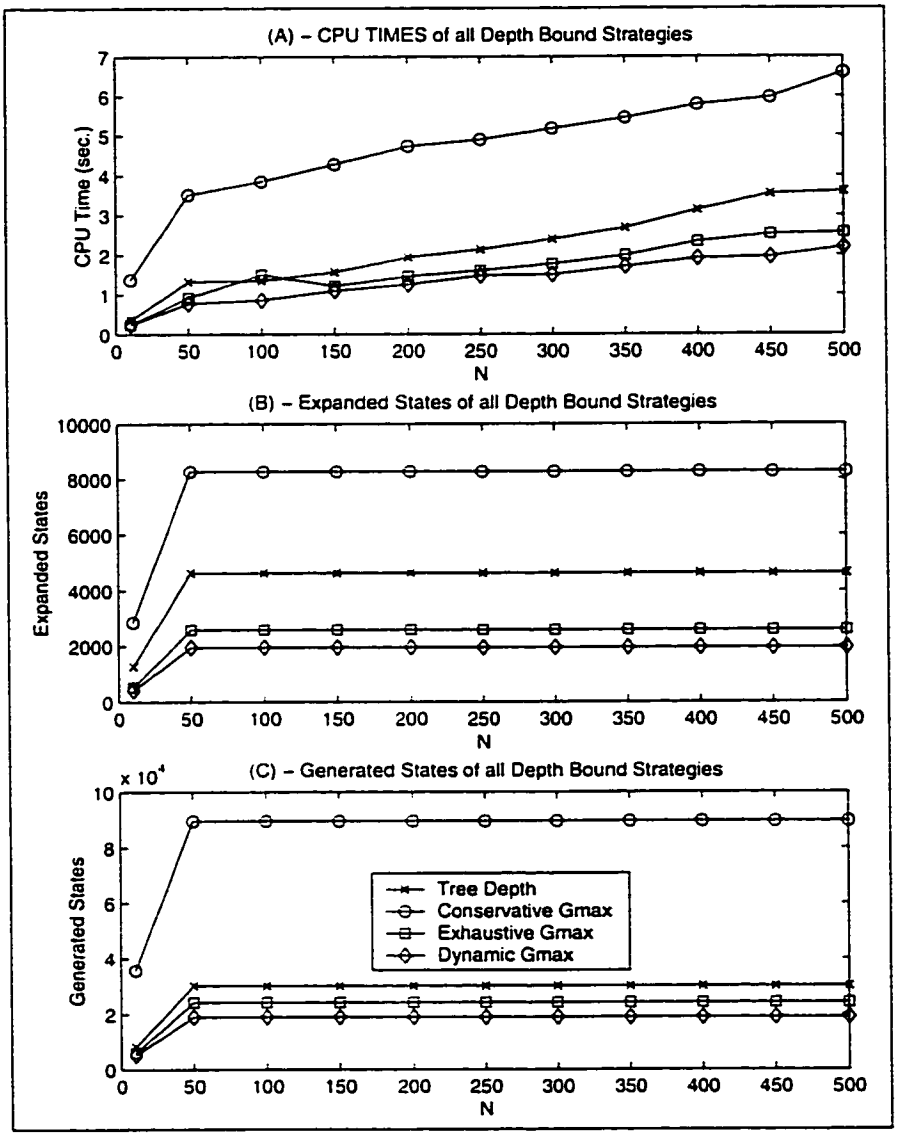

Figure 5.6: The *Contact Lenses* Results ($Relaxation\_Factor = 1.9$)

by the other two algorithms $Exhaustive\_ID_{G-max}$ and $Dynamic\_ID_{G_{max}}$. The $Exhaustive\_ID_{G-max}$ consumes slightly more CPU time, generates and expands slightly more states than the $Dynamic\_ID_{G_{max}}$ which starts with a performance worse than that of the $ID_{TreeDepth}$ algorithm but as the $Relaxation\_Factor$ is increased, the $Dynamic\_ID_{G_{max}}$ algorithm improves to outperform all of the other algorithms. The reason for this improved performance is the increased sensitivity of to the number of fringe states in the search tree by the $Dynamic\_ID_{G_{max}}$ search algorithm. At some point, with a $Relaxation\_Factor$ in either ranges $1.9 - 2.5$ or $7 - 9$, the $Dynamic\_ID_{G_{max}}$ algorithm finds a suitable balance between the cost of performing a number of DFS passes and the gain of discovering the desired states. This balance is based on a suitable *heuristic branching factor* of the search tree such that the iterative deepening technique becomes more effective. For a large value of the $Relaxation\_Factor$, the $Dynamic\_ID_{G_{max}}$ performs very similar to the $Exhaustive\_ID_{G-max}$ expanding and generating the same number of states but consuming slightly more CPU time due to the maintenance of the $Observed_{G_{max}}$ data structure, an unnecessary task for the $Exhaustive\_ID_{G-max}$ algorithm.

Having fixed the $Relaxation\_Factor = 1.9$, a value observed to be effective, Figure 5.6 on Page 102 shows the experimental results obtained for the fixed $Relaxation\_Factor = 1.9$ and a varying $N \in [10 \cdots 519]$. The observations are consistent with the above ranking of the four algorithms. In addition, we observe that the $Dynamic\_ID_{G_{max}}$ consistently performs slightly better than the $Exhaustive\_ID_{G_{max}}$ but both of these algorithms perform significantly better than the other two. The $ID_{TreeDepth}$ algorithm seems to have a larger gap in the expanded states than in CPU time or in generated states. This observation indicates that using the G_{max} instead

of the actual depth of states as the search bound strategy produces a more suitable *heuristic branching factor* for the iterative deepening technique.

The *Congressional Voting* Results

We used the groups of bits (1-10, 11-20, and 21-30) of the available 34 bits. However, the experimental results were all similar for the three portions of this dataset. We only show the results for the second group of bits (bits 11-20) because it illustrates the worst case performance for the G_{max}-based algorithms. Recall the characteristics of this dataset, the number of examples in the dataset is significantly higher than the other datasets, 435 records. This means the number of passes on the dataset becomes more important because when evaluating a state, a search algorithm performs a complete pass on all records in the dataset. In the implementations of all four algorithms, we evaluate every generated state. Thus, if an algorithm generates more states, then the CPU time may suffer. This dataset also has a higher number of G_{max} values each of which has a small number of states whose score $G > 0$. In this case, the G_{max} is extremely inaccurate in estimating an upper bound on the G scores in this domain, see Figure 5.3 on page 96. This immediately causes the $Conservative\_ID_{G_{max}}$ to become extremely inefficient because iterative deepening must start a new DFS for each of the G_{max} values, and each will explore a very small number of new states causing the algorithm to perform an enormous amount of work for very little gain. Therefore, we chose to exclude the $Conservative\_ID_{G_{max}}$.

Using similar presentation to the previous dataset results, the experimental results for the second group of bits (21-30) of this dataset are presented in Figure 5.7 on page 105 which shows the performance of $ID_{TreeDepth}$, $Exhaustive\_ID_{G_{max}}$, and $Dynamic\_ID_{G_{max}}$ algorithms for a

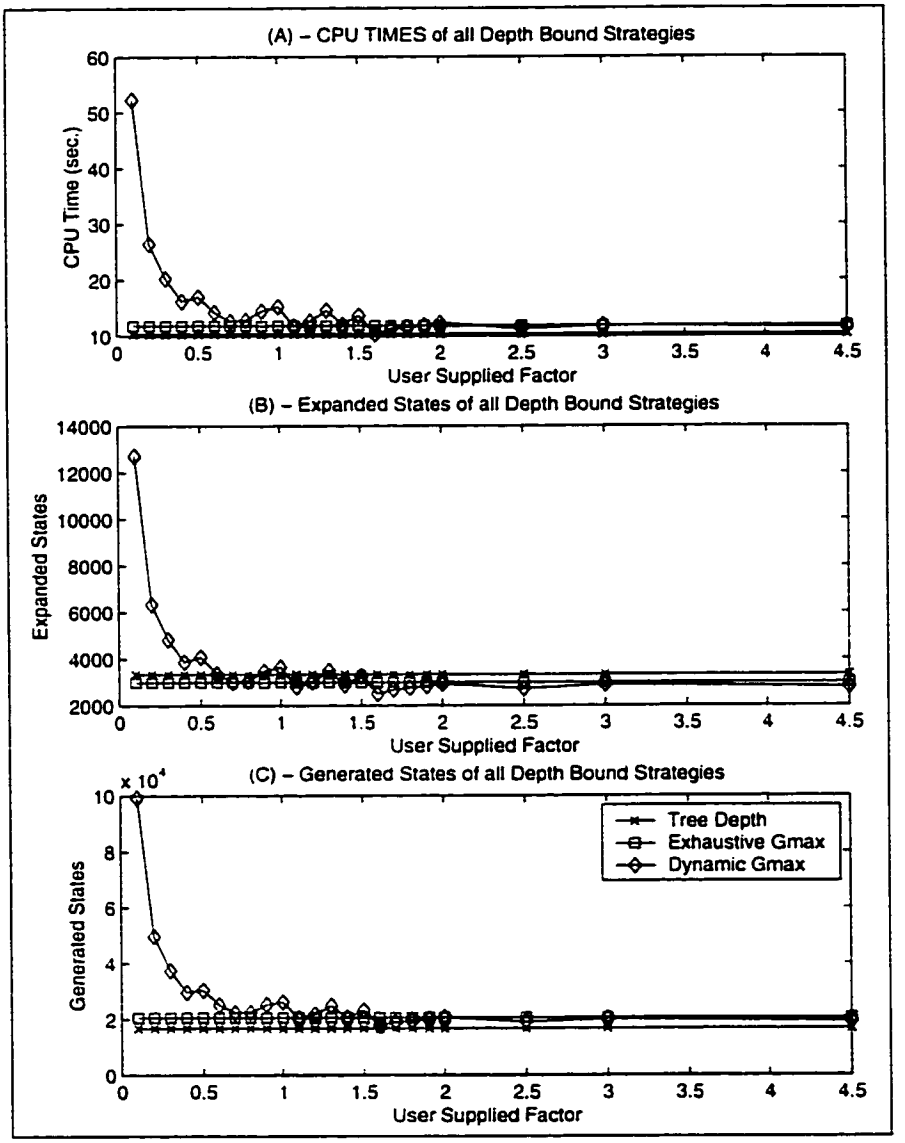

Figure 5.7: The *Congressional Voting* Results (bits $11-20, N = 564$)

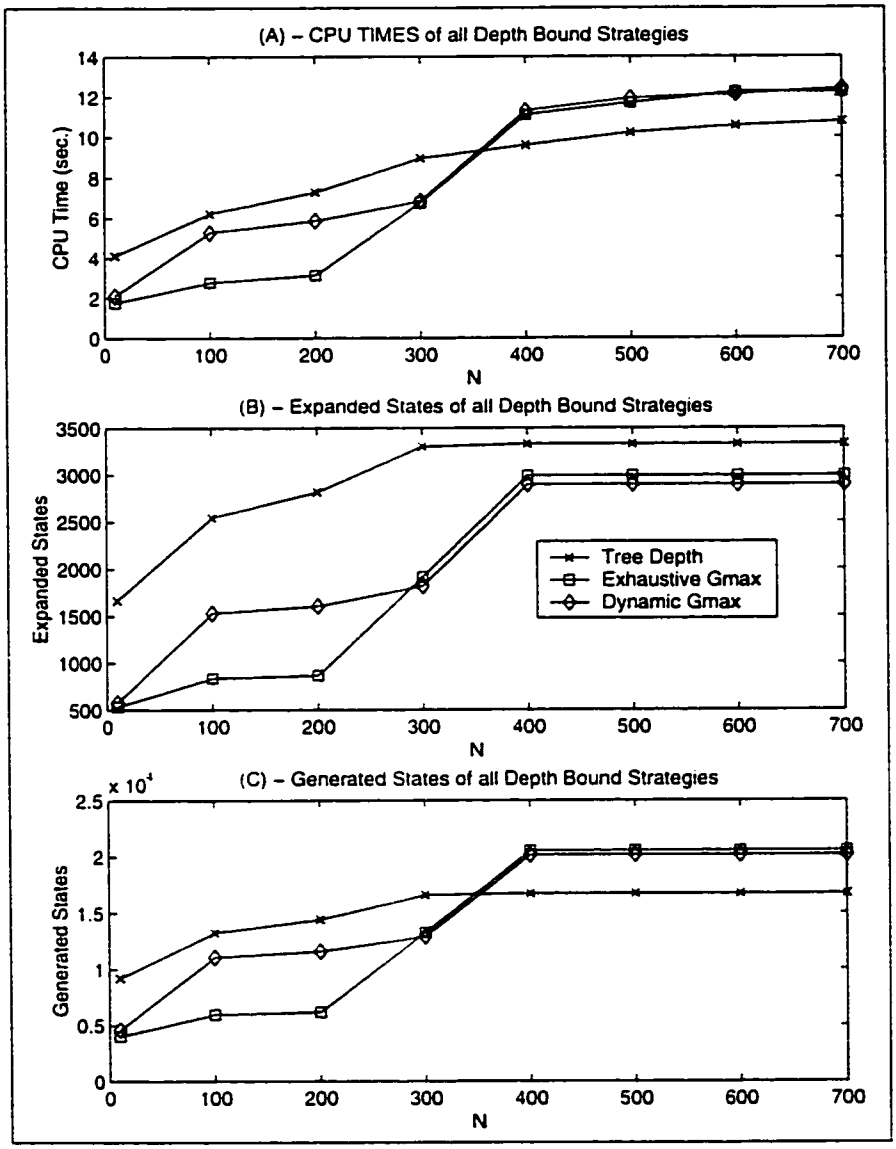

Figure 5.8: The *Congressional Voting* Results (bits $11-20$,$Relaxation\_Factor = 3.0$)

fixed N (greater than the number of states whose $G > 0$) and a varying *Relaxation_Factor*. The $Dynamic\_ID_{G_{max}}$ algorithm starts with a poor performance according to all measures then improves to match that of the $Exhaustive\_ID_{G_{max}}$. However, both algorithms are outperformed by the $ID_{TreeDepth}$ algorithm in both CPU time and in number of states generated but not in number of states expanded. As mentioned earlier, due to the high number of records, in this dataset, generating a state is more costly than expanding it. Therefore, the $ID_{TreeDepth}$ is the winner simply because it generates fewer states. If we consider Figure 5.8 on page 106 which shows the performance of the three search algorithms for a fixed *Relaxation_Factor* = 3.0 and an increasing N, we notice that the $Exhaustive\_ID_{G_{max}}$ and the $Dynamic\_ID_{G_{max}}$ algorithms outperform the $ID_{TreeDepth}$ for $N \leq 300$ with the $Exhaustive\_ID_{G_{max}}$ being the best of them all. This strong performance is due to the fact that this search space has many G_{max} values with few states in each of the G_{max} values, thus, the $Exhaustive\_ID_{G_{max}}$ considers the lowest G_{max} value it observes as a search bound for the subsequent DFS which causes fewer DFS passes to be performed. However, for values of $N > 300$, the $ID_{TreeDepth}$ performs better than the other two because it generates fewer but expands more states. The G_{max}-based algorithms use the G_{max} to generate, expand, and prune states in the search space. Thus, the *heuristic branching factor* for this search space is related to the behavior of the G_{max} values in the search space. The $ID_{TreeDepth}$ uses the G_{max} to expand and prune states only. The search bound is based on the depth of states in the search states. In this case, the *heuristic branching factor* is based on a combinatorial increase followed by a decrease in the search space, presented in [30]. Since the G_{max} is a failing upper bound estimate of the G values, shown in Figure

5.3 on page 96, where many of the $G_{max} > 0$ values have a $G = 0$, then, G_{max} may lead to generating a search space with a *heuristic branching factor* unsuitable for the iterative deepening search. The *heuristic branching factor* of $ID_{TreeDepth}$ is different than that of the G_{max}-based algorithms, therefore, $ID_{TreeDepth}$ does not suffer from a failing G_{max} as much. On the other hand, since $ID_{TreeDepth}$ explores the search space using the depth of states as the search bound, it suffer from the deficiencies of Depth-First Search by expanding an exponential number of states if the G_{max} fails to guide the search effectively. This deficiency explains the higher number of states $ID_{TreeDepth}$ expands. Despite this observation, the performance of all three search algorithms seem comparable considering all performance measures. The difference in their performance is still considered small due to the small size of the dataset. When we consider all 34 bits of this dataset, this observation reveals that the deficiencies of $ID_{TreeDepth}$ become very costly and that the $Dynamic\_ID_{G_{max}}$ emerges as the best.

The *Software Development* Results

We selected two groups of bits $(1 - 10)$, and $(21 - 30)$ from the available 81 bits. The number of examples for this dataset is relatively low, only 38. Similar to the previous presentation of results, Figure 5.9 on page 109 shows the performance of all four search algorithms for a fixed N (greater than the number of states whose $G > 0$) with a varying $Relaxation\_Factor$ for group $(1 - 10)$ of bits. According to all measures, the $Conservative\_ID_{G_{max}}$ exhibits the worst performance of all four algorithms, which is consistent with all the other experimental results. The $Dynamic\_ID_{G_{max}}$ performs significantly better than the other three algorithms. Recall that in this dataset several small G_{max} values are highly

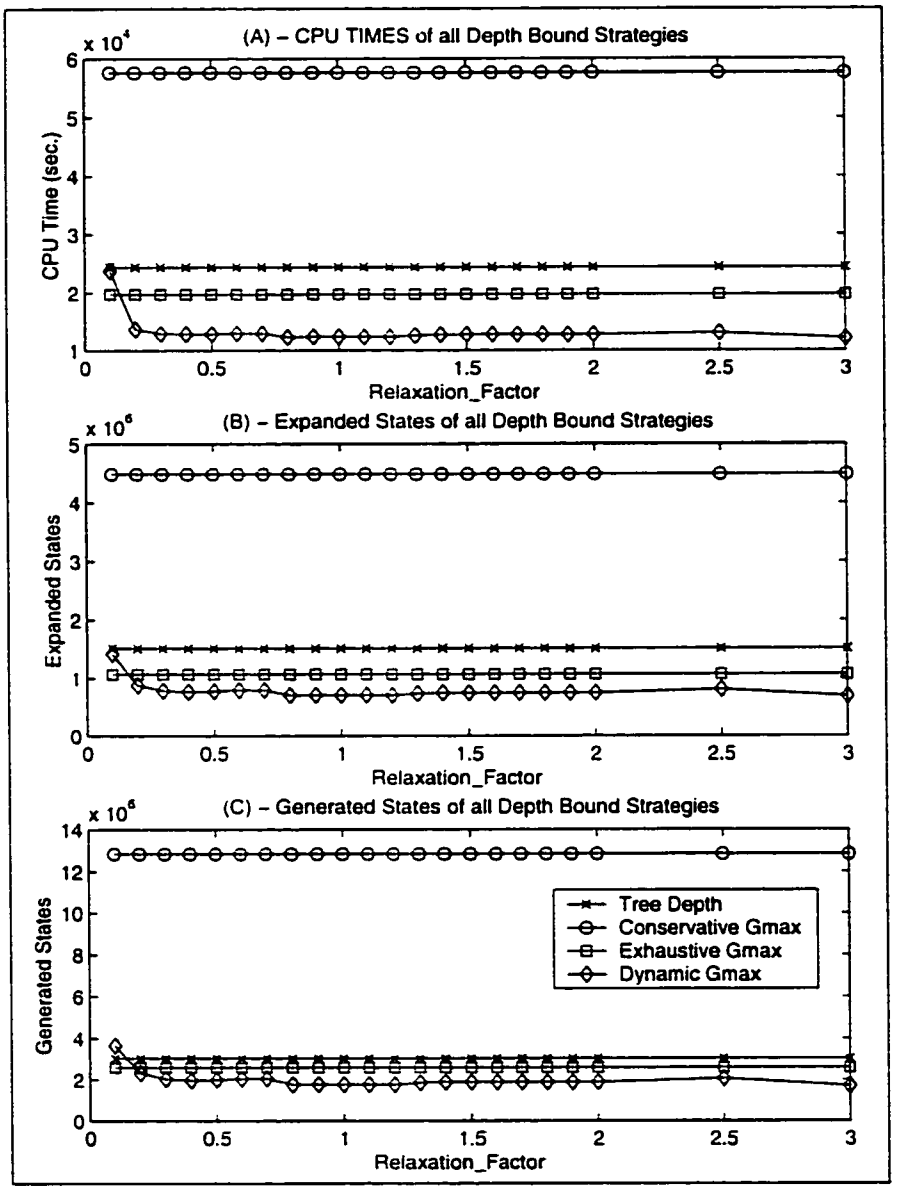

Figure 5.9: The *Software Development* Results (bits $1-10$,$N = 15072$)

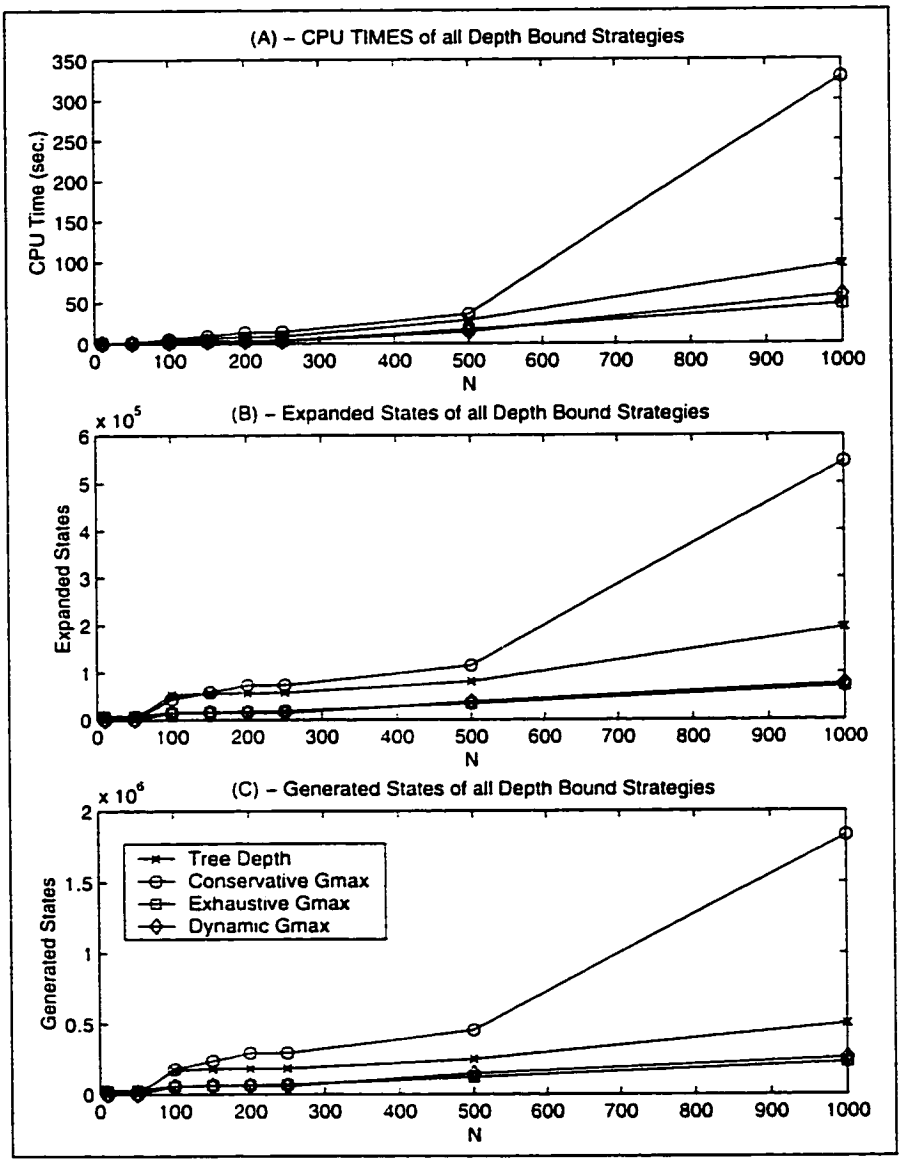

Figure 5.10: The *Software Development* Results (bit $1-10$,$Relaxation\_Factor = 0.8$)

populated with states. Since the $Dynamic\_ID_{G_{max}}$ controls the search bound based on the number of fringe states found in each of the DFS passes (using the $Relaxation\_Factor$), then, the $Dynamic\_ID_{G_{max}}$ is capable of avoiding having to search a high number of states especially when theses small values of G_{max} are encountered earlier on in the search. In this case, the $Exhaustive\_ID_{G_{max}}$ drops the search bound to the lowest value of G_{max} it encounters and searches more states early on causing the DFS to spend much more CPU time expanding and generating more states. The $Exhaustive\_ID_{G_{max}}$ and the $Dynamic\_ID_{G_{max}}$ perform significantly better than the $ID_{TreeDepth}$ algorithm by all measures. Since this search space is considered fairly rich with potentially desired states (states whose $G_{max} > 0$ are approximately 27% of the entire search space), then, using the G_{max}-based *heuristic branching factor* generates a search space focused on potentially desired states only, Table 5.7 shows that 95% of states whose $G_{max} > 0$ have $G > 0$. These potentially desired states can be scattered throughout the space of all possible states. Therefore, using a state-depth-based *heuristic branching factor* may cause the $ID_{TreeDepth}$ to have to generate and expand more states whose G_{max} and G are equal to 0 in order to reach potentially desired state, thus explaining the poor performance of the $ID_{TreeDepth}$ search algorithm.

Figure 5.10 shows the performance measures of the four algorithms for a fixed $Relaxation\_Factor = 0.8$ and a varying N. This figure shows a rapidly developing performance gap illustrating the extreme inefficiency of the $Conservative\_ID_{G_{max}}$. The $ID_{TreeDepth}$ is out performed by the remaining two algorithms which consume less CPU time, generate and expand fewer states indicating that generating the search space using the G_{max} significantly improves the iterative deepening performance especially

when G_{max} is reflective of the G scores. In the previous dataset, the G_{max} failed in estimating the upper bounds on the G scores, thus $ID_{TreeDepth}$ performed better. However, in this dataset, G_{max} is more successful in estimating these upper bounds. therefore, the G_{max}-based algorithms performed much better by searching multiple values of G_{max} in each of the DFS passes. The gap in the number of expanded states between the $ID_{TreeDepth}$ and the other two algorithms is larger than in the other measures because $ID_{TreeDepth}$ generates the search space based on the depth of states which, similar to the previous dataset, may cause the iterative deepening to expand undesired states to reach the desired ones. In addition, for $N = 1000$, the $Exhaustive\_ID_{G_{max}}$ seems to perform slightly better simply because the pruning becomes more effective in helping the $Exhaustive\_ID_{G_{max}}$ to prevent the G_{max}-based search bound from progressing into lower values while performing fewer DFS passes. Figure 5.10 shows a small performance gap between the $Dynamic\_ID_{G_{max}}$ and the $Exhaustive\_ID_{G_{max}}$.

The performance measures of all four algorithms on the second group of bits $(21 - 30)$ of the *Software Development* dataset are shown in Figure 5.11 (for a fixed $N = 7423$) on page 113. These results are very consistent with the previous group of bits. The $Conservative\_ID_{G_{max}}$ algorithm is the most inefficient, the $ID_{TreeDepth}$ is consistently outperformed by the other two algorithms with a bigger gap in the number of expanded states measure. An additional observation is that in all three measures and for some values of the *Relaxation_Factor*, the performance of $Dynamic\_ID_{G_{max}}$ improves significantly than that of the $Exhaustive\_ID_{G_{max}}$. For large values of the *Relaxation_Factor* the $Dynamic\_ID_{G_{max}}$ performs the same as the $Exhaustive\_ID_{G_{max}}$ because the former explores the same G_{max}

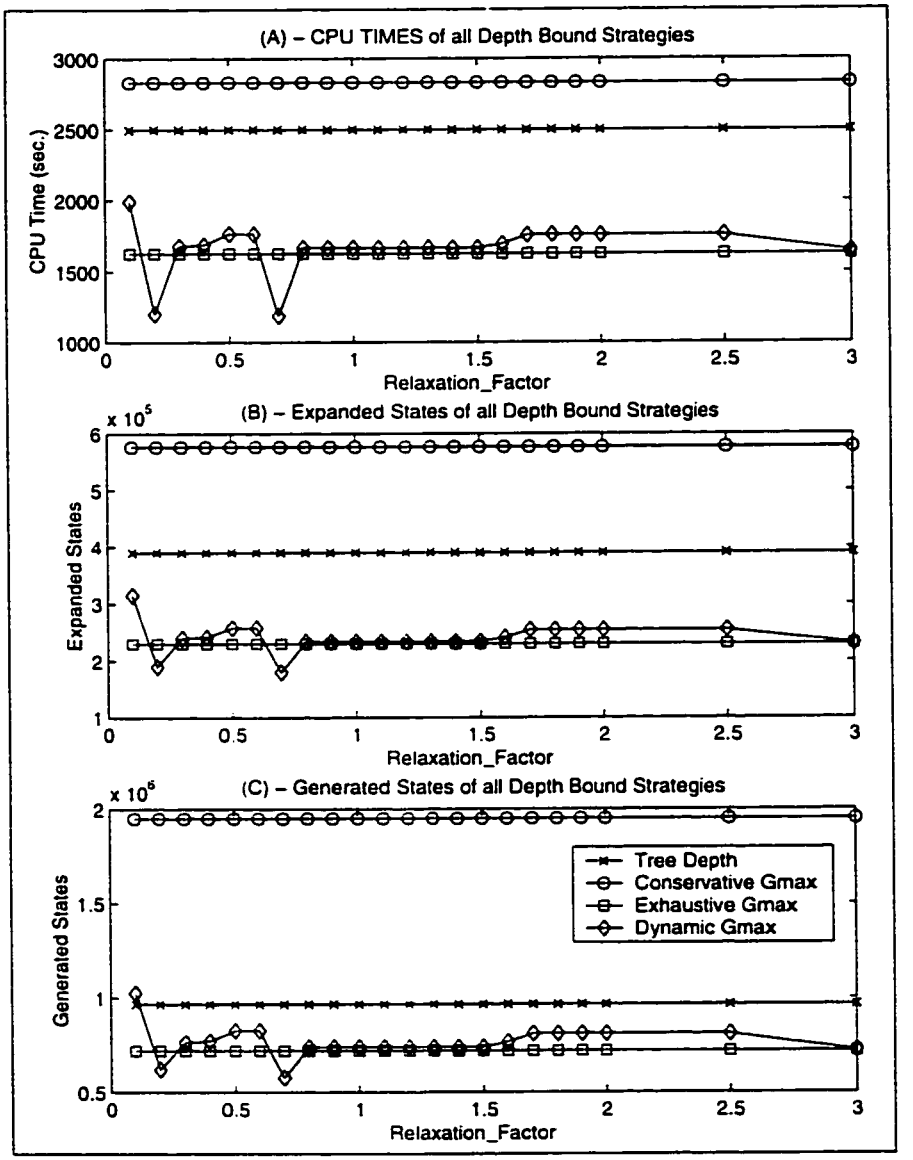

Figure 5.11: The *Software Development* Results (bits $21-30, N = 7423$)

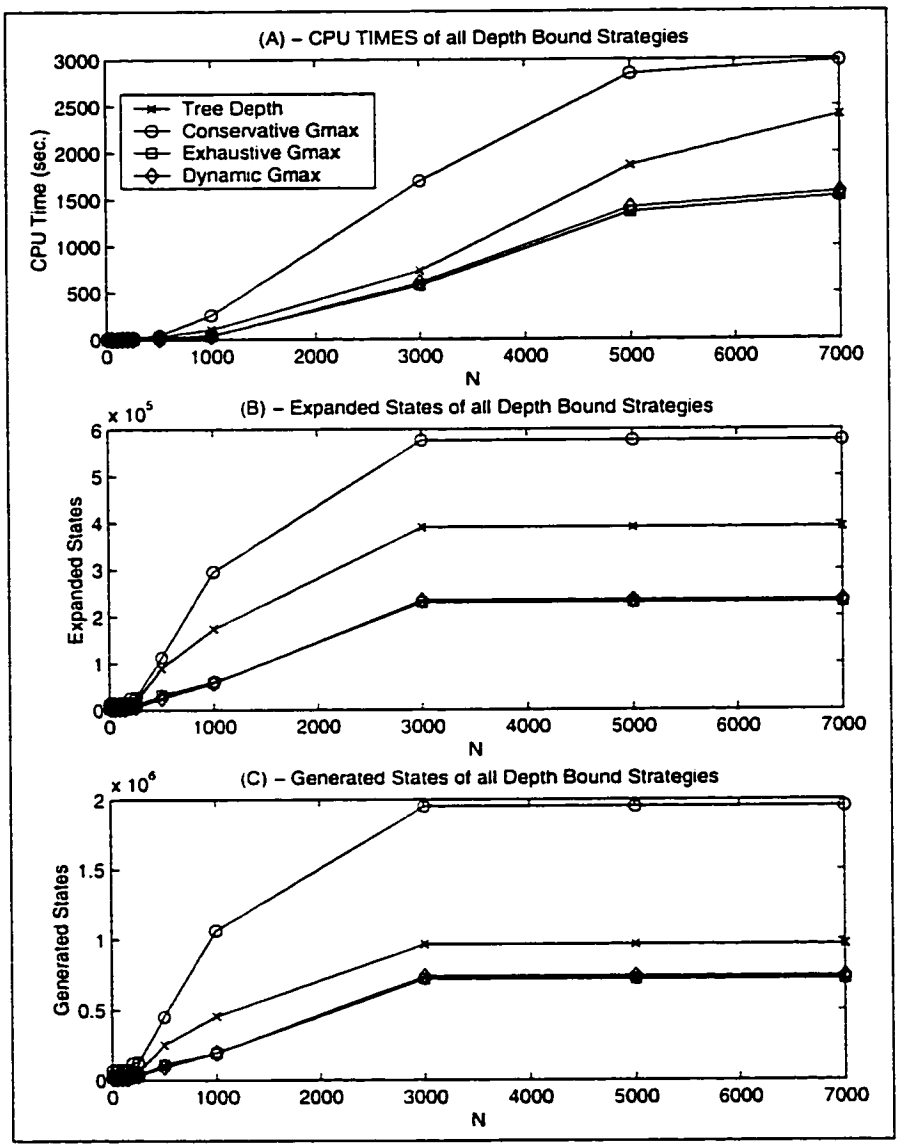

Figure 5.12: The *Software Development* Results (bit $21-30$,$Relaxation\_Factor = 0.8$)

bounds as the latter. These bursts of improved performance suggest that some choices of the *Relaxation_Factor* values suit the shape of the search space better than others, i.e. for such values, the search algorithm explores smaller portions of the search space. This observation suggests that the *Relaxation_Factor* value must be chosen carefully so that the search algorithm can explore the search space in a more efficient fashion. It may even be more useful to dynamically change the *Relaxation_Factor* as the $Dynamic\_ID_{G_{max}}$ starts each DFS. This thesis work only illustrates the effectiveness of the *Relaxation_Factor* in speeding up the $Dynamic\_ID_{G_{max}}$ search algorithm. This thesis does not address such dynamic change of the *Relaxation_Factor*. In Figure 5.12, the experimental results this portion of the *Software Development* dataset shows the performance of the four algorithms with a fixed $Relaxation\_Factor = 0.8$ and a varying N. The main observation is a major degradation in the performance of all four search algorithms as N is increased beyond $N = 1000$. This observation illustrates an explosive property of the search tree. The $Conservative\_ID_{G_{max}}$ algorithm is known to be inefficient because it starts a new DFS for every G_{max} value exploring previously expanded states plus states in the new G_{max} value. The $ID_{TreeDepth}$ algorithm, however, suffers from the explosive nature of the search space when searching states at deeper levels of the search tree because it uses a simple depth-first strategy. Thus, $Conservative\_ID_{G_{max}}$ and $ID_{TreeDepth}$ algorithms prove ineffective compared to the G_{max}-based algorithms exploring multiple G_{max} values in each DFS.

5.4.3 Conclusions

In this experiment, we ran the $ID_{TreeDepth}$, the $Conservative\_ID_{G_{max}}$, the $Exhaustive\_ID_{G_{max}}$, and the $Dynamic\_ID_{G_{max}}$ search algorithms on small search spaces. We chose search spaces limited to approximately 10 bits to allow us to exhaustively analyze properties of the search space to determine their influence on the performance of the four algorithms. In this section, we summarize our conclusions based on the experimental results we presented. First, we verified the correctness of the implementations of all four algorithms. Relationship rules (or states) discovered by all four implementations were identical to those generated exhaustively. Therefore, we conclude that all four algorithms are capable of extracting any relationship rules found in any search space provided that the experiment meets the preconditions.

Second, the $Conservative\_ID_{G_{max}}$ search algorithm, whose search bound is based on exploring one additional G_{max} value for each DFS, proves very inefficient. This deficiency appears in many situations, especially when the search space has many G_{max} values each containing few states, a situation shown in 3.4.6 to be the worst case performance of the iterative deepening technique. Or in the situation when the G_{max} values contain an explosive number of states. Since $Conservative\_ID_{G_{max}}$ adds a single G_{max} value to the search tree for each DFS, if the desired N states are found in the first few G_{max} values, then the $Conservative\_ID_{G_{max}}$ may perform well, however, for larger values of N which span many G_{max} values, the $Conservative\_ID_{G_{max}}$ has to perform many DFS (one for each value) expanding previously expanded states plus states found in the new G_{max} value. Therefore, it performs the worst of all four algorithms.

Third, generating and exploring the search space based on the G_{max} values provides the iterative deepening with a more suitable *heuristic branching factor* which allows the search algorithm to generate states with a focus on the potentially desired states. The $ID_{TreeDepth}$ generates the search space, in this case, based on the method of specializing the rules controlled to a maximum depth where the *heuristic branching factor* follows a consistent behavior of a combinatorial increase followed by a decrease in the size of the search space [30]. This approach is shown in the case of the *Congressional Voting* dataset (especially when N is large) to be more effective than using a failing G_{max} when estimating the upper bounds on G for small sizes of datasets. However, the subsequent experiment shows that these results do not scale up too well. The $ID_{TreeDepth}$ algorithm suffers from the deficiencies of the DFS technique and expands more states in an attempt to reach the desired states especially when G_{max} can be more useful. When the G_{max} is more successful in estimating the upper bounds on the G scores, the G_{max}-based algorithms exploring the search tree using iterative deepening and adding multiple G_{max} values for each of the DFS passes prove more effective and show a significant speed up.

Finally, the $Dynamic\_ID_{G_{max}}$ algorithm is capable of outperforming the $Exhaustive\_ID_{G_{max}}$ algorithm. The $Exhaustive\_ID_{G_{max}}$ can suffer from a very high concentration of states found in lower values of the G_{max} bounds. This algorithm drops the search bound to the lowest observed G_{max} value. Thus, if these low G_{max} values are observed early on in the search tree, the $Exhaustive\_ID_{G_{max}}$ may have to generate and explore a high number of states despite the pruning of undesired states. On the other hand, the $Dynamic\_ID_{G_{max}}$ controls the G_{max}-based search bound by the $Relaxation\_Factor$ which is related to the number of states the algorithm

expands. Our experiments show that for some *Relaxation_Factor* values, the $Dynamic\_ID_{G_{max}}$ can avoid exploring G_{max} bounds which may contain an explosive number of states and may be misleading, i.e. their G scores are much lower than their G_{max} upper bounds.

5.5 Experiment II: $ID_{G_{max}}$ Performance - Large Scale

In this section, we present the second of our experiments to show the scaling up of the performance of the $ID_{G_{max}}$ search algorithm. In this experiment, we run the $ID_{TreeDepth}$, the $Exhaustive\_ID_{G_{max}}$, and the $Dynamic\_ID_{G_{max}}$ search algorithms on large datasets. We exclude the $Conservative\_ID_{G_{max}}$ search algorithm from this experiment due to the poor performance in the previous experiment.

Dataset	Records	Bits
Solar Flare	323	40
Flight Errors (Air Canada)	22433	30
U. S. Congressional Voting	435	34

Table 5.8: Datasets Used in Experiment II

The datasets we use are listed in Table 5.8. Scaling up is accomplished in two ways; one, by using a higher number of attributes that have more values, and second, by using more data records. The effects of such increase in size is mainly characterized by two main properties; first, the search space is exponentially large, for example if the number of bits is 40 then the size of the search space is 3^{40}. Second, the number of passes on records in the datasets significantly influences CPU time, for example, the *Flight Errors* dataset contains 22433 records which are consulted by the search

algorithms whenever a state is evaluated. All four implementation of search algorithms evaluate every generated states, an algorithm performs better if it generates fewer states. Therefore, this scale up can determine how all three search algorithm perform when required to address a large search space.

5.5.1 Experimental Results II: Discussion

Similar to the presentation of previous experimental results, in this section, we present the results of this experiment for each of the datasets listed in Table 5.8. For each dataset, we show two figures, the first figure plots the CPU Time, the number of states expanded, and the number of states generated by the search algorithms for a fixed number of desired states N and varying $Relaxation\_Factor$ values. The second figure plots the same three measures for a fixed $Relaxation\_Factor$ value and varying N values.

The *Solar Flare* Results

Figure 5.13 on page 120 shows the performance of the three algorithms for a fixed $N = 100$ and various values of the $Relaxation\_Factor$. Again the results of this experiment are consistent with the previous experiment. The $ID_{TreeDepth}$ performs the worst of the three algorithms because the search explores a different search space which may be bigger than the space generated by the G_{max} values while suffering from deficiencies related to the Depth-First search. This is illustrated by the larger gap of performance in the number of states expanded measure. The interesting observation is that the performance of the $Dynamic\_ID_{G_{max}}$ is significantly better than that of the $Exhaustive\_ID_{G_{max}}$ algorithm. This poor performance is also attributed to the well known deficiency of the Depth-First Search (DFS).

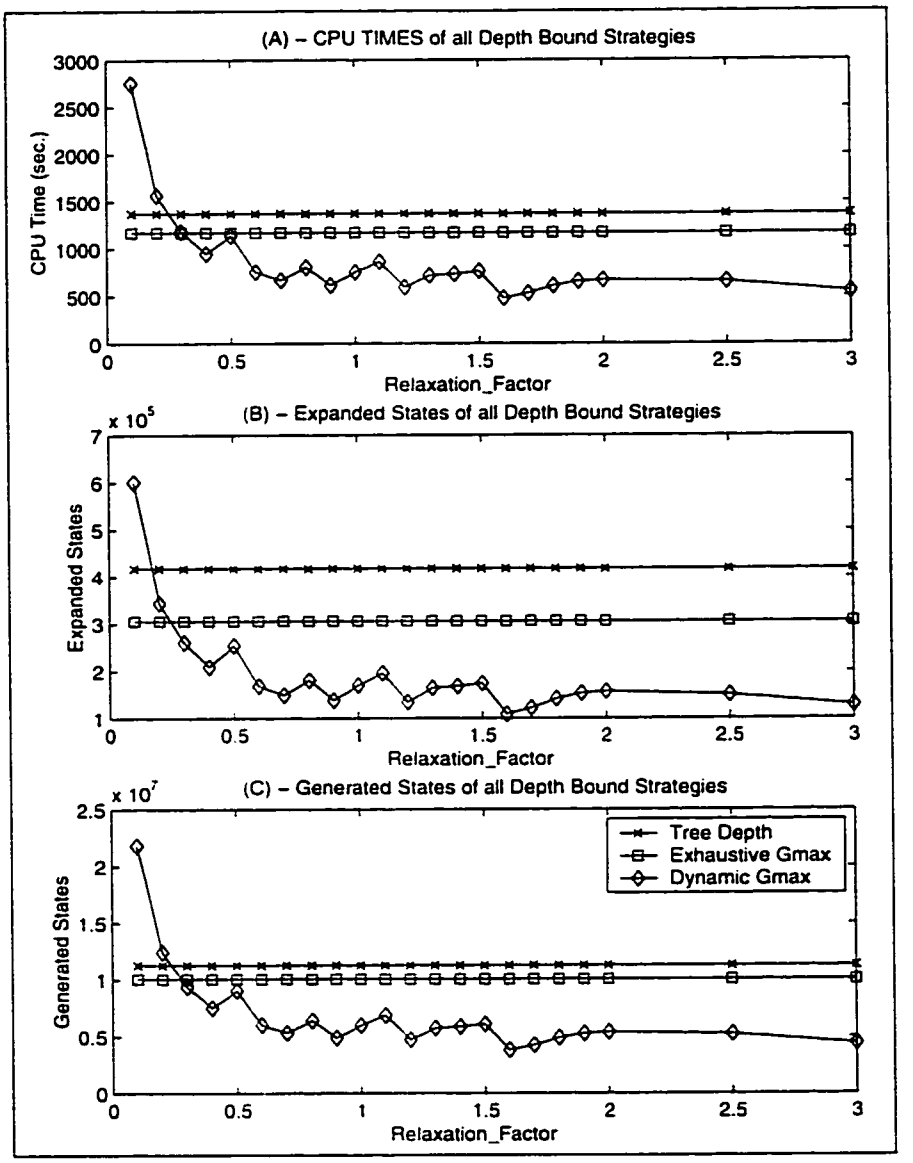

Figure 5.13: The *Solar Flare* Results (40 bits,$N = 100$)

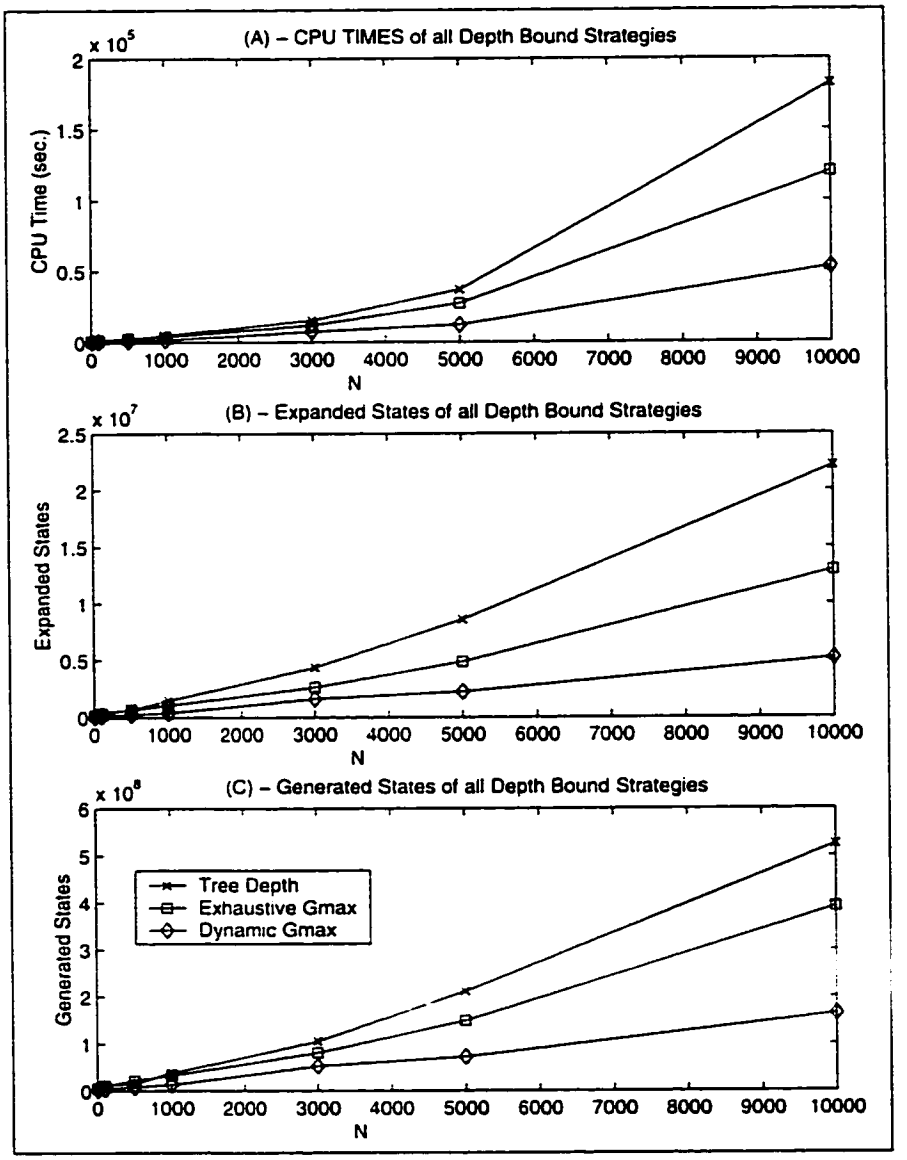

Figure 5.14: The *Solar Flare* Results (40 bits,$Relaxation\_Factor$= 3.0)

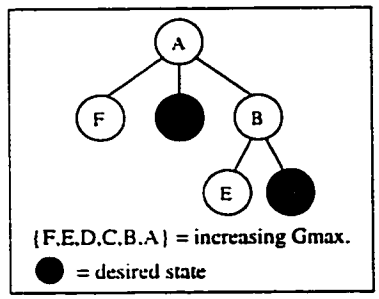

Figure 5.15: A Search Space Example

To illustrate this idea, consider the search space in Figure 5.15. Let the states $\{F, E, D, C, B, A\}$ be sorted by the increasing order of their G_{max} values, i.e. $G_{max}(B) \leq G_{max}(A)$ etc. Also, let the set of desired N states be found in the subtrees rooted at the states $\{D, C\}$. In this situation, when the search bound is set to $G_{max}(D)$, then, in a given DFS pass the search algorithm explores the states $\{A, B, D, C\}$ respectively until it reaches the set of N desired states. However, if the search bound is set to $G_{max}(E)$, then the search algorithm explores the states $\{A, B, D, E, C\}$ because DFS must explore all subtrees before it backtracks to explore the next candidate state. This is a consequence of eliminating the open list of candidate states used in the Best-First Search. This difference in setting the search bound causes the search, in this case, to explore additional subtrees of E which may not contain any desired states. The $Dynamic\_ID_{G_{max}}$ sets the search bound based on the $Relaxation\_Factor$ which, in this example may be set to $G_{max}(D)$ using an appropriate value of the $Relaxation\_Factor$. The $Exhaustive\_ID_{G_{max}}$ algorithm, however, drops the search bound to the lowest observed G_{max} value in the most recently completed DFS. Thus, the $Exhaustive\_ID_{G_{max}}$ may be forced to set the search bound so that it may generate and explore more states in the search space. The vari-

able performance of the $Dynamic\_ID_{G_{max}}$ shows that some values of the *Relaxation_Factor* are more suitable for the iterative deepening. This illustrates the benefits of using the *Relaxation_Factor* as a parameter to control the iterative deepening search so it explores the search space more efficiently. Our observations on the performance of these three search algorithms also hold for a fixed *Relaxation_Factor* = 3.0 and a varying N shown in Figure 5.14.

The *Flight Errors* Results

The second of our large datasets is the *Flight Errors* dataset. These experimental results support the observations and conclusions we have made so far. The $Conservative\_ID_{G_{max}}$ algorithm is included in the first run of this experiment to confirm its poor performance on at least one of the large datasets. Figure 5.16 on page 124 shows the performance of all four search algorithms when $N = 500$ is fixed and the *Relaxation_Factor* is varied. The $Conservative\_ID_{G_{max}}$ performs the worst of all four in CPU time and number of generated states. However, the $ID_{TreeDepth}$ expands the most number of states of all four algorithms. The poor performance of the $ID_{TreeDepth}$ in the expanded states measure highlights the deficiencies of DFS it suffers from and the difference between generating the search space using the actual tree depth approach and using the G_{max} values. The $ID_{TreeDepth}$ generates fewer states than the $Conservative\_ID_{G_{max}}$, therefore, it compensates for expanding many more states when the CPU time is measured. However, both the $Conservative\_ID_{G_{max}}$ and the $ID_{TreeDepth}$ are consistently outperformed by the other two G_{max}-based algorithms. The interesting performance comes from the $Exhaustive\_ID_{G_{max}}$ algorithm which seems to perform reasonably well on this dataset which suggests that

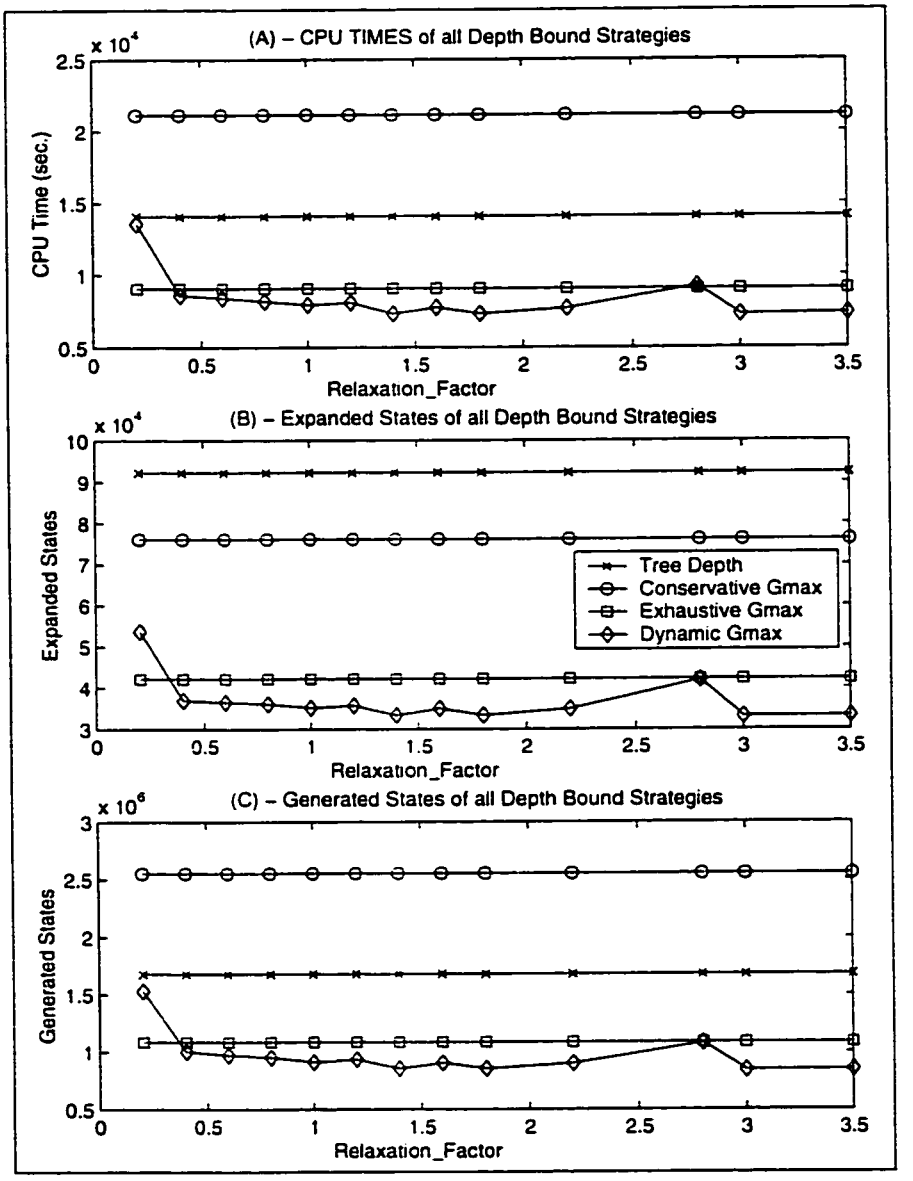

Figure 5.16: The *Flight Errors* Results (30 bits,$N = 500$)

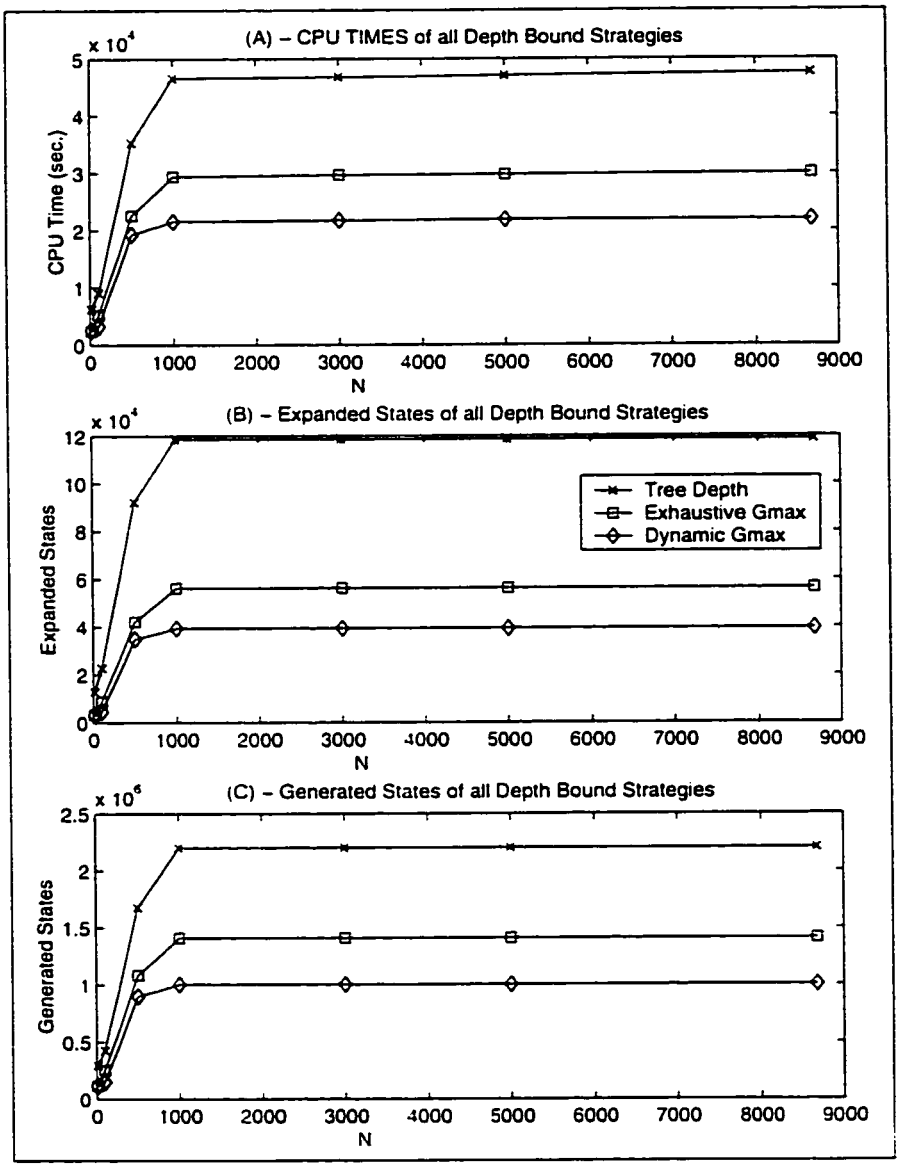

Figure 5.17: The *Flight Errors* Results (30 bits.$Relaxation\_Factor = 2.2$)

using the G_{max} values to generate the search space provides iterative deepening with a more suitable *heuristic branching factor*. However, the fact that the $Conservative\_ID_{G_{max}}$ performs so poorly indicates that must explore many values of G_{max} in each of the DFS passes.

In the second run of this dataset, presented in Figure 5.17, we excluded the $Conservative\_ID_{G_{max}}$ due to its poor performance. The figure shows the performance of $ID_{TreeDepth}$, $Exhaustive\_ID_{G_{max}}$, and $Dynamic\_ID_{G_{max}}$ when the $Relaxation\_Factor = 2.2$ is fixed and N is varied. As the number of sought after states N increases to 1000, all three algorithms reach a point where their performance remains constant. This observation indicates that the maximum number of states whose $G > 0$ has been reached, or that all such states are found in the same level of the search tree, thus no additional cost of exploration is required to reach desired states. This is illustrated by the flat performance curve. However, in this dataset, the $Dynamic\_ID_{G_{max}}$ algorithm, once again, outperforms all the other algorithms with the $Exhaustive\_ID_{G_{max}}$ outperforming the $ID_{TreeDepth}$. This observation confirms our claim that using the G_{max}-based algorithms exploring many G_{max} values in each DFS perform better than the depth-based algorithm. In addition, using the dynamic approach to relax the search bound improves the performance of iterative deepening even better.

The *Congressional Voting* Results

The final large dataset of this experiment is the *Congressional Voting* dataset. These experimental results are also consistent with the conclusion that the $Dynamic\_ID_{G_{max}}$ and the $Exhaustive\_ID_{G_{max}}$ algorithms always outperform the $ID_{TreeDepth}$. Figure 5.18 on page 127 shows that for a fixed $N = 10$ and a changing $Relaxation\_Factor$, the performance

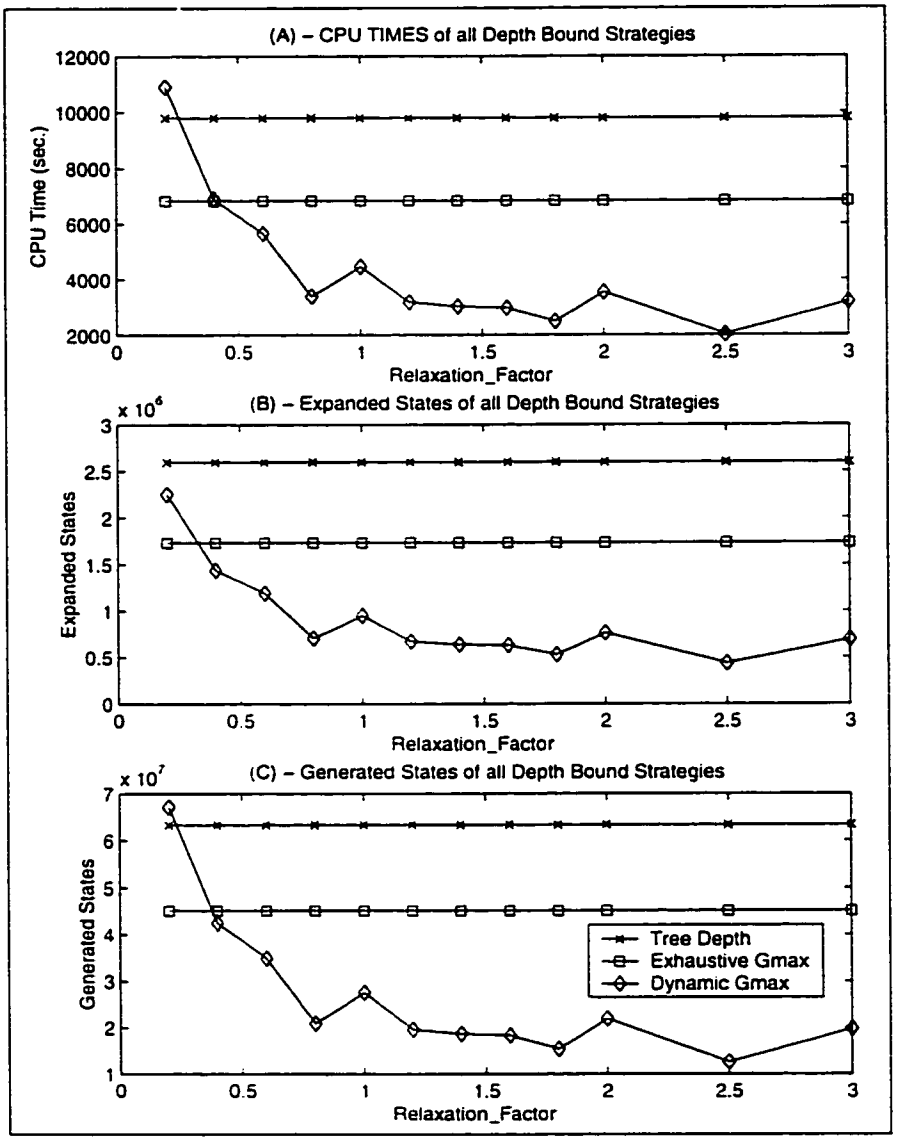

Figure 5.18: The *Congressional Voting* Results (34 bits,$N = 10$)

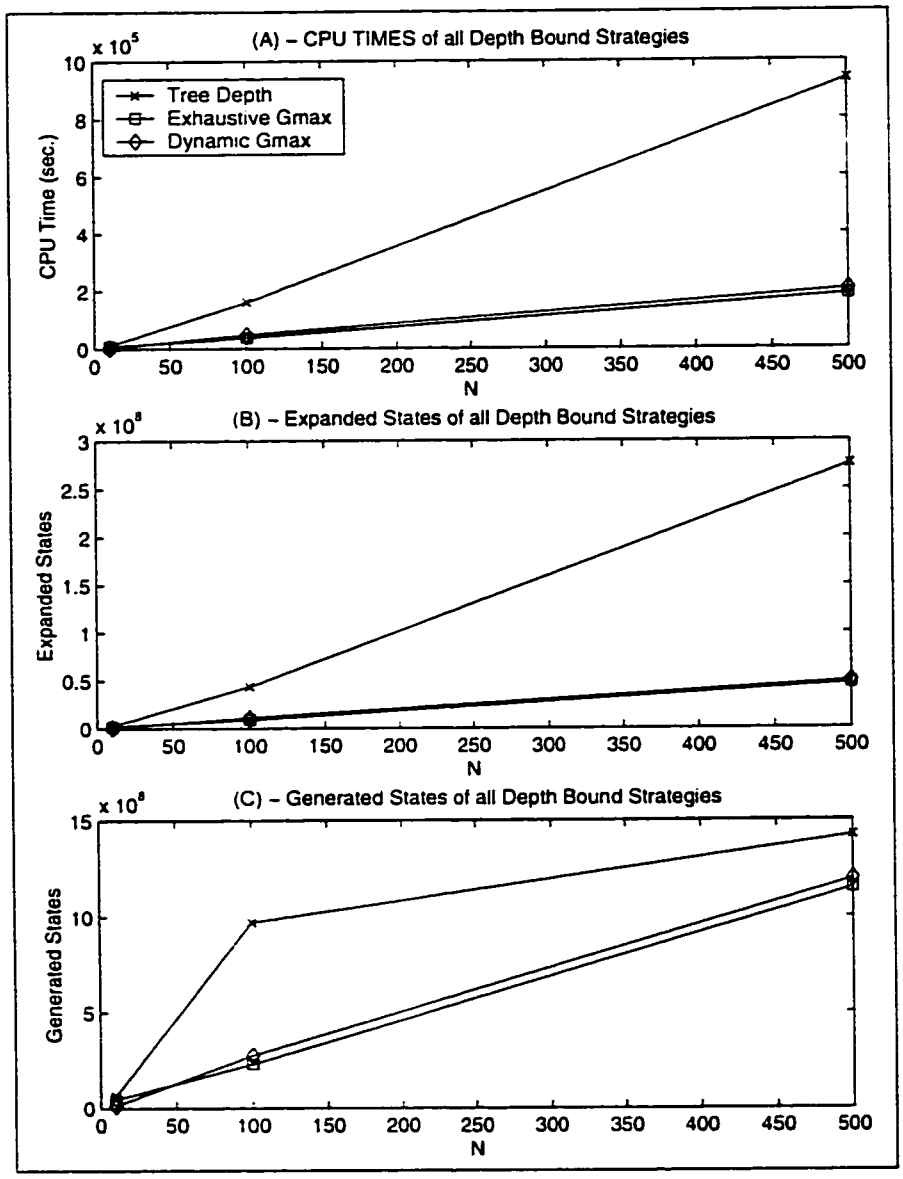

Figure 5.19: The *Congressional Voting* Results (34 bits,$Relaxation\_Factor$= 2.5)

of the $ID_{TreeDepth}$ is significantly worse than the G_{max}-based algorithms. Very early on, the $Dynamic\_ID_{G_{max}}$ succeeds in improving its performance to almost a factor of 5 to that of the $ID_{TreeDepth}$ and a factor of 2 to that of the $Exhaustive\_ID_{G_{max}}$. However, consider the performance of the $Dynamic\_ID_{G_{max}}$ and the $Exhaustive\_ID_{G_{max}}$ algorithms when the $Relaxation\_Factor = 2.5$ is fixed and N is varied. As shown in Figure 5.19, both search algorithms seem to perform very closely and consistently better than the $ID_{TreeDepth}$ algorithm. This close performance may indicate a low number of fringe states that survive pruning. Thus both of the algorithms explore similar shape search trees. The $ID_{TreeDepth}$, once again, exhibits a bigger gap in the states expanded to the other two algorithms than in the generated states. In addition, the $ID_{TreeDepth}$ suffers from an explosion in the number of states it generates when $N = 100$, when $N = 500$ the gap to the other two algorithms in the number of generated states decreases which is consistent with a combinatorial increase followed by a decrease in the number of states to generate. This confirms the deficiency of using the search tree depth (based on the state generation method) in reducing the search space to focus on the desired states especially when the G_{max} estimate of the G upper bounds fails to prune and guide the search effectively. The $ID_{TreeDepth}$ algorithms, again, suffers from both deficiencies of generating a large search space and deficiencies related to using the DFS to explore an explosive search space.

5.5.2 Conclusions

In this experiment, we scaled up the size of the dataset used to compare the performance of the four rule mining algorithms. We increased the number of attributes and values which effectively increases the size of the

search space exponentially. The aim is to investigate the performance differences between the MSDD-based iterative deepening, $ID_{TreeDepth}$, and the three G_{max}-based strategies of $ID_{G_{max}}$ search algorithm, namely, the $Conservative\_ID_{G_{max}}$, the $Exhaustive\_ID_{G_{max}}$, and the $Dynamic\_ID_{G_{max}}$. The experiment concludes with many encouraging results consistent with our previous experimental results. First, our approach of generating and searching a search space based on the G_{max} values is more effective than using the search tree depth based on the state generation method. However, in order to benefit from this speed up, the G_{max}-based iterative deepening algorithm must search multiple G_{max} values in each DFS pass. This is shown by the consistent poor performance of the $Conservative\_ID_{G_{max}}$ algorithm and the better performance of the $Exhaustive\_ID_{G_{max}}$ and the $Dynamic\_ID_{G_{max}}$ algorithms. The $ID_{TreeDepth}$ algorithm suffers from the deficiency of generating a larger search space in addition to the deficiency of using the DFS technique. This was illustrated by the consistent poor performance of the $ID_{TreeDepth}$ algorithm, particularly by consistently expanding more states than the other two algorithms. Second, we have illustrated that the $Exhaustive\_ID_{G_{max}}$ may possibly suffer from deficiencies of using the DFS technique when compared to the $Dynamic\_ID_{G_{max}}$ algorithm due to the elimination of the open list of candidate states, a deficiency that hampers the performance of the $ID_{TreeDepth}$ algorithm as well. Finally, this experiment shows that using the $Dynamic\_ID_{G_{max}}$ algorithm with a suitable $Relaxation\_Factor$ may improve the performance by almost 50% when compared to the $Exhaustive\_ID_{G_{max}}$.

Chapter 6

Conclusion & Future Work

This chapter presents a final summary of our research work, followed by concluding statements based on the studies and results conducted in this thesis. This chapter also provides a discussion of several key points for future work.

6.1 Research Summary

In this thesis, we addressed the problem of mining relationship rules from datasets which record transactions involving items. The objective is to discover the strongest N relationship rules from a database. In chapter 2, we provided basic knowledge in the field of knowledge discovery and data mining. Mining relationship rules from transaction data involves many tasks and is subject to many interpretations. We introduced the idea of association rules which interpret the relationship expressed by the rule as an "Association" indicating how items "co-occur" together in data transactions. We also reviewed several other relationship rule interpretation methods, such as "Correlation" rules measured by the χ^2 test and "De-

pendency" rules measured by the G measure of independence. Finally, we reviewed methods of generating and evaluating relationship rules and illustrated how the problem of discovering relationship rules from data can be reduced into a state space exploration problem for which there exist several methods. In chapter 3, we established evaluation criterion to help evaluate our proposed search algorithm. In addition, we reviewed several existing methods to explore state spaces in search of a desired set of states. There are generic and problem specific search methods. In both cases, we illustrated how these strategies work and discussed their strengths and weaknesses. Finally, we presented our implementation of the MSDD algorithm ($ID_{TreeDepth}$) which we use to compare against our search algorithm ($ID_{G_{max}}$). In chapter 4, we presented our newly developed algorithm, $ID_{G_{max}}$ with a discussion of how our algorithm works using iterative deepening with a search bound based on an upper bound estimation of the quality of states in the search space. We also proposed three different variations of how to relax the search bound for iterative deepening, the conservative, the dynamic, and the exhaustive search bound strategies. In chapter 5, we presented the experimental results where we described the objectives of the experiments, and the properties of datasets used, discussed the experimental results, and finally, drew conclusions based on the observations.

6.2 Conclusions

Our conclusions for this thesis work are mainly based on the experimental results we obtained. The $ID_{G_{max}}$ algorithm is a search algorithm we designed to explore a tree-based search space. Given a tree state space,

a state evaluation measure G, and a heuristic upper bound on the state evaluation G_{max}, the $ID_{G_{max}}$ generates a search space based on the G_{max} values and explores portions of this state space in search of the N states that have the highest G score. The $ID_{G_{max}}$ uses the technique of iterative deepening to explore the search space in a non-increasing order of the G_{max} values. Our experiments illustrated the following:

1. Using a heuristic upper bound (G_{max}) to generate the search space may produce a search space more focused on the desired quality of states than generating the search space, in the case of $ID_{TreeDepth}$, based on the depth of states as they are specialized in the search tree. The G_{max}-based *heuristic branching factor* is more suitable for the iterative deepening search technique provided that the search explores multiple G_{max} values in each of the DFS pass. The performance of the $Conservative\_ID_{G_{max}}$ strategy was consistently poor. In addition, the number of generated states by the G_{max}-based algorithms was, almost always, less than that generated by $ID_{TreeDepth}$.

2. Using the above heuristic approach as a search bound for iterative deepening performs better than using the tree depth. The performance of $ID_{G_{max}}$ algorithm exploring multiple G_{max} values in each DFS was shown to be consistently better than that of $ID_{TreeDepth}$ algorithm on small and large datasets where $ID_{TreeDepth}$ expanded more states and consumed more CPU time most of the time.

3. Our experimental results showed that the conservative strategy for relaxing the search bound is inefficient. However, the exhaustive strategy performs well when compared to the existing depth bound strategy ($ID_{TreeDepth}$). Moreover, the dynamic relaxation of the search bound

can perform worse, the same, or better than the exhaustive strategy based on the chosen value of the *Relaxation_Factor*. The main conclusion is that, given an appropriate *Relaxation_Factor*, the dynamic relaxation of the search bound can outperform all other strategies with a significant speed up.

4. Having applied the $ID_{G_{max}}$ search algorithm to the problem of mining relationship rules introduces a novel and an efficient rule mining algorithm which we evaluated, according to criterion discussed in chapter 3, to perform significantly better than the existing $ID_{TreeDepth}$.

6.3 Future Work

In this thesis, we applied $ID_{G_{max}}$ search algorithm to the problem of mining relationship rules from transaction data. Further research may be conducted in two main areas. The first is to study methods of how to generate and evaluate desired relationship rules from transaction data efficiently. An improvement in this area may produce a significant reduction in the size of the search space, thus, adding more efficiency to the search algorithm. The second area is to improve the efficiency of the heuristic search algorithm itself. In the following sections, we discuss both of these areas, and present some key points which may provide some solutions to significant problems.

6.3.1 Future Work in Generating & Evaluating Rules

The problem of discovering any rules from a set of transaction data depends on many important factors. Such important factors may be the size of the space of all possible rules, and the effectiveness of establishing a rank between desired and undesired rules, i.e. evaluation of such rules. Therefore,

performing either of these two tasks more effectively and consistently can improve the performance of the data mining algorithm significantly. For instance:

1. Recall, the desired rules are rules which express a relationship between itemsets. To generate the space of all possible such relationships, we need to determine itemsets that could possibly be involved in such relationships. First, we may construct the rules (in this thesis work, we enumerated all possible rules between all possible itemsets which produce an exponentially large search space, $3^{\#ofbits}$), then, we use heuristic methods to prune and reduce the size of the search space. Many researchers [30, 5] devised methods to reduce the number of items involved in the relationship first, then generated the space of all possible rules on the smaller set of items. This can reduce the size of the problem significantly. Such methods can easily be added to the $ID_{G_{max}}$ search algorithm, simply because $ID_{G_{max}}$ is independent of how states are generated and evaluated in the search space.

2. From an implementation point of view, evaluating a rule, currently, involves performing a pass on all dataset records for every rule evaluated. When the dataset contains a large number of records, this task becomes costly. Therefore, for very large datasets, it is beneficial to investigate the use of more efficient methods to evaluate the rules without performing many passes on the entire dataset records. There exists several methods recently derived to solve this problem. Such as, estimation [2], the smart use of data structures [5], smart counting of rules [13], sampling [39], or using a database to support the rule mining process [18].

6.3.2 Future Work in State Exploration Algorithm - $ID_{G_{max}}$

The following identifies a list of ideas which may increase the efficiency of $ID_{G_{max}}$ algorithm:

1. At present, the search bound for a depth-first search pass of $ID_{G_{max}}$ iterative deepening is always set before starting the DFS pass and remains unchanged for the duration of the DFS pass. Perhaps, if the algorithm attempts to alter the search bound while performing the DFS, the algorithm might become more sensitive to the structure of the search tree. Hence, it may be more capable of adjusting its search bound to avoid a possible explosion in the state space or to extract useful rules early on the search which can increase the effectiveness of pruning. This idea may help reduce the number of times states are being re-explored by iterative deepening simply by reducing the number of DFS passes required.

2. The current implementation of $ID_{G_{max}}$ restarts a new DFS pass for every new search bound set. In a new DFS pass, $ID_{G_{max}}$ explores previously explored states plus new states found in the current search bound. An idea may be, is to store important information in memory about the most recently completed DFS pass so that the subsequent DFS pass will explore only new states found in the new search bound. This idea attempts to improve the efficiency of the search algorithm by eliminating (or reducing) the need to explore states which have already been expanded.

3. The idea of using a *Relaxation_Factor* to control the relaxation of the iterative deepening search bound was shown, in our experimental results, to produce a significant speed up in execution time. This

leads to the question as to how can we determine the most suitable factor to mine a given dataset?. Moreover, if it is possible to compute such a factor, we may be tempted to use machine learning techniques to learn such factors from previously mined datasets, these questions remain open.

4. Our $ID_{G_{max}}$ search algorithm employs a fixed strategy of iterative deepening. Oates et al., in [31], presented a search algorithm (MSDD) which replaces the search strategy (iterative deepening) with another during the the execution of the search algorithm. It may be worth investigating the effectiveness of altering the search strategy during the search when certain characteristics of the search space become more apparent.

5. Our main objective in the development of the $ID_{G_{max}}$ algorithm is to reduce the memory required to solve the problem of mining rules from transaction data. However, given that in these days, memory is widely available in significant quantities, the question remains, how can $ID_{G_{max}}$ benefit most from having additional memory.

6. Chapter 3 presented existing search space exploration techniques. One such technique is the Depth-First Branch and Bound ($DFBBS$) reviewed in section 3.4.7 in which we presented a possible implementation. In the future work, we may implement and evaluate the DFBBS against the different search bound strategies used by $ID_{G_{max}}$.

In conclusion, when searching for a desired set of states in a tree-based search space, using the iterative deepening technique with a search bound based on a heuristic evaluation of the quality of desired states to iteratively explore multiple values of this heuristic is shown to be effective and fruitful.

Bibliography

[1] *IBM Intelligent Miner User's Guide.*, volume Version 1, Release 1. International Business Machines., 1994.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining associations between sets of items in massive databases. In *ACM-SIGMOD Int. Conf. on Management of Data*, pages 207–216, 1993.

[3] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. *Advances in Knowledge Discovery and Data Mining. AAAI Press.* pages 307–328, 1996.

[4] R. Agrawal, M. Mehta, J. Shafer, and R. Srikant. The quest data mining system. In *The First Int. Conf. on Knowledge Discovery and Data Mining*, pages 244–249. AAAI Press., 1996.

[5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *The Twentieth VLDB Int. Conf.*, pages 487–499, 1994.

[6] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Research Report RJ 9839, IBM Almaden Research Center, San Jose, California*, 1994.

[7] A. Agresti. *Categorical Data Analysis.* John Wiley & Sons, New York 1990.

[8] K. Ali, S. Manganaris, and R. srikant. Partial classification using association rules. In *The Third Int. Conf. on Knowledge Discovery and Data Mining*, pages 115–118, 1997.

[9] R. J. Bayardo. Brute-force mining of high-confidence classification rules. In *The Third Int. Conf. on Knowledge Discovery and Data Mining*, pages 188–197, 1997.

[10] R. J. Bayardo and R. Agrawal. Mining the most interesting rules. In *The Fifth Int. Conf. on Knowledge Discovery and Data Mining*, pages 145–154, 1999.

[11] R. J. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining in large, dense databases. In *The Fifteenth Int. Conf. on Data Engineering*, pages 188–197, 1999.

[12] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *ACM-SIGMOD Int. Conf. on The Management of Data.*, pages 265–276, 1997.

[13] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *ACM-SIGMOD Int. Conf. on The Management of Data.*, pages 255–264, 1997.

[14] P. Clark and P. Boswell. Rule induction with cn2: Some recent improvements. In *The Fifth European Conf. on Machine Learning*, pages 151–163, 1991.

[15] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining*. AAAI Press / The MIT Press, 1996.

[16] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data mining using two-dimensional optimized association rules: Scheme, algorithms, and visualization. In *ACM-SIGMOD Int. Conf. on The Management of Data*, pages 13–23, 1996.

[17] J. Han, J. Y. Chiang, S. Chee, J. Chen, and Q. Chen. Dbminer: A system for data mining in relational databases and data warehouses. In *The Second Int. Conf. on Knowledge Discovery and Data Mining*, pages 250–255, 1996.

[18] M. Holsheimer, M. Kerten, H. Mannila, and H. Toivonen. A perspective on databases and data mining. In *The First Int. Conf. on Knowledge Discovery and Data Mining*, pages 150–155. AAAI Press., 1995.

[19] M. Kamber, J. Han, and J. Y. Chiang. Metarule-guided mining of multi-dimensional association rules. In *The Third Int. Conf. on Knowledge Discovery and Data Mining*, pages 207–210, 1997.

[20] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *The Third Int. Conf. on Information and Knowledge Management.*, pages 401–407, 1994.

[21] R. E. Korf. Depth-first iterative deepening: An optimal admissible tree search. In *Artificial Intelligence*, volume 27, pages 97–109, 1985.

[22] R. E. Korf. Optimal path-finding algorithms. In L. Kanal and V. Kumar, editors. *Search in Artificial Intelligence*, pages 223–267. Springer-Verlag, 1988.

[23] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *The Third Int. Conf. on Knowledge Discovery and Data Mining*, pages 115–118, 1997.

[24] N. Megiddo and R. Srikant. Discovering predictive association rules. In *The Fourth Int. Conf. on Knowledge Discovery and Data Mining*, pages 274–278, 1998.

[25] W. Mendenhall, R. Scheaffer, and D. Wackerly. *Mathematical Statistics with Applications*. Duxbury Press, 3rd edition, 1986.

[26] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., 1994.

[27] Y. Morimoto, T. Fukuda, H. Matsuzawa, T. Tokuyama, and K. Yoda. Algorithms for mining association rules for binary segmentation of huge categorical databases. In *The Twenty-fourth VLDB Int. Conf.*, pages 380–391, 1998.

[28] S. Morishita. On classification and regression. In *The First Int. Conf. on Discovery Science - Lecture Notes in Artificial Intelligence.*, volume 1532, pages 40–57, 1998.

[29] S. Morishita and A. Nakaya. Parallel branch-and-bound graph search for correlated association rules. In *ACM SIGKDD Workshop on Large-Scale Parallel KDD Systems*, 1999.

[30] T. Oates and P. R. Cohen. Searching for structure in multiple streams of data. In *The Thirteenth Int. Conf. on Machine Learning*, pages 346–354, 1996.

[31] T. Oates, M. D. Schmill, and P. R. Cohen. Efficient mining of statistical dependencies. In *The Sixteenth Int. Joint Conf. on Artificial Intelligence.*, pages 794–799. 1999.

[32] B. P. Patrick, M. Almulla, and M. M. Newborn. An upper bound on the complexity of iterative-deepening-a*. In *Annals of Mathematics and Artificial Intelligence*, volume 5, pages 167–278, 1992.

[33] G. Piatetsky-Shapiro and W. J. Frawley. *Knowledge Discovery in Databases.* AAAI Press / The MIT Press, 1991.

[34] R. Rastogi and K. Shim. Mining optimized association rules with categorical and numeric attributes. In *The Fourteenth Int. Conf. on Data Engineering.*, pages 503–512, 1998.

[35] C. Silverstein, S. Brin, R. Motwani, and J. Ullman. Scalable techniques for mining causal structures. In *Data Mining and Knowledge Discovery*, pages 163–192. 2000.

[36] R. Srikant and R. Agrawal. Mining generalized association rules. In *The Twenty-first Int. Conf. on Very Large Databases*, pages 407–419, 1995.

[37] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *The Third Int. Conf. on Knowledge Discovery and Data Mining*, pages 67–73. 1997.

[38] M. E. Stickel and Tyson W. M. An analysis of consecutively bounded depth-first search with applications in automated deduction. In *The Ninth Int. Joint Conf. on Artificial Intelligence*, pages 1073–1075, 1985.

[39] H. Toivonen. Sampling large databases for association rules. In *Twenty Second VLDB Int. Conf. on Very Large Databases*, pages 134–145. Morgan Kaufmann, 1996.

[40] H. Toivonen, M. Klemettienen, K. Hatonen, and H. Mannila. Pruning and grouping discovered association rules. In *MLnet Workshop on Statistics, Machine Learning, and Discovery in Databases*, pages 47–52, 1995.

[41] N. R. Vempaty, V. Kumar, and R. E. Korf. Depth-first versus best-first search. In *The 9th National Conf. on Artificial Intelligence*, pages 434–440, 1991.

[42] G. I. Webb. Opus: An efficient admissible algorithm for unordered search. *Journal of Artificial Intelligence Research*, 3:431–465, 1995.

[43] T. D. Wickens. *Multiway Contingency Tables Analysis for The Social Sciences*. Lawrence Erlbaum Associates., 1989.