LEARNING EFFECTIVE BDD VARIABLE ORDERS FOR BDD-BASED PROGRAM ANALYSIS

Michael Carbin

May 2006

Abstract

Software reliability and security are in jeopardy. As software has become ubiquitous and its capabilities have become more complex, code quality has been sacrificed in the race for the next ”killer app.” In response, program analysis researchers have mounted a revolution; they have developed new tools and methods, underpinned by traditional compilation techniques, in order to save software from its downward spiral. However, because these tools and analyses have also become more sophisticated, they too have suffered from scalability, reliability and complexity issues.

Just as program analysis researchers have set out to solve the problems of software developers, we have set out to solve the problems of program analysis researchers. The bddbddb (Binary Decision Diagram-Based Deductive DataBase) system has recently made possible many advanced, context-sensitive program analyses. Such analyses can be expressed in bddbddb as Datalog queries, which are quantifiably easier to write than a traditional implementation. The bddbddb system's unique compilation mechanisms also yield analyses of exceptional performance. The key to this performance is the use of Binary Decision Diagrams (BDDs), a compact representation that exploits repeated patterns in data, to represent the principle elements of an analysis. However, finding efficient BDD representations (i.e., BDD variable orders) for a particular analysis is nearly impossible to accomplish manually and, in some cases, is without a solution given an analysis and its formulation.

This thesis presents an algorithm that helps automate the discovery of efficient BDD representations. Our technique reformulates the search for BDD variable orders as an active learning process over the space of BDD variable orders and their execution times. This technique revolves around an iterative process of carefully sampling new orders and then reducing the search space by extracting features from high performance orders. The dominant features of the sampled variable orders can then be used to generate new orders, refine existing orders, or even revise analysis formulations. The variable orders generated by our algorithm outperform those obtained after months of manual exploration. And, more importantly, our results make bddbddb a viable and valuable tool for program analysis researchers in their quest for better quality code.

Acknowledgments

I would like to thank all those who contributed both directly and indirectly to this thesis. In particular, I thank both Professor Monica Lam and John Whaley for their many hours of thinking, writing, and coding. Some of the writing and figures for this thesis (i.e., in Chapter 2 and Chapter 4) are derived from work we've done for various submissions and other publications[53]. I also thank them for sparking my interest in further pursuing research.

Above all, I would like to thank my friends and family; they have dealt with my many late nights of hacking over the years and without them, I would be lost in a sea of code with no means for relief.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software bugs cost an estimated $59.9 billion a year in the US alone[36]. As code size and complexity have increased, the traditional manual methods for code reasoning, testing, and verification have proven inadequate. The growing chasm between the rate of increase of code complexity and that of code quality has given birth to the field of program analysis. Program analysis, the study of computer programs for the discovery of program properties, comprises both academic and industrial projects in the pursuit of better code quality. However, the main challenge that we in this field face is not the difficulty in designing cutting edge algorithms, but, rather, the inherent trade-offs between analysis scalability, quality/applicablity, and implementation complexity. The bddbddb (Binary Decision Diagram Based Deductive DataBase) program analysis framework, which John Whaley, Professor Monica Lam, I, and others have built, aims to address these problems[24, 50, 53].

The bddbddb system provides a flexible program analysis framework aimed at experts and non-experts alike. Program analyses are represented by a declarative Datalog query. This specification consists of sets of relations over input program elements and a sequence of rules by which to combine them. These rules dictate the manner by which new, more interesting properties are formed. The bddbddb system then automatically compiles this specification down to an optimized, low-level Binary Decision Diagram (BDD) based implementation, which then executes over a given input program.

We have found these Datalog queries to be smaller and much easier to write than their equivalent BDD-based manual implementations in Java or C++. For example, 3450 lines of Java for one of our context-sensitive points-to analyses shrinks to 33 lines of Datalog with the help of bddbddb. Datalog provides analysis writers with a high-level specification language that pushes the complex, intertwined issues of analysis performance and implementation down into bddbddb. This abstraction puts the matter of optimized BDD manipulation into the hands of BDD experts and, thus, makes analyses much easier to get right due to their concision and simplicity.

As to generality, we, and others, have successfully used this tool to drive a number of analyses, including both Java and C context-sensitive pointer analyses[2, 50] (both of which were largely unfeasible previously), SQL-injection detection[24], static race detection[35], finite tree automata analysis[18], and reflection-aware call graph construction[29].

The performance and scalability of our framework follows from our use of BDDs. The BDD data structure can compact a large data set by exploiting the commonalities in the data. With BDDs, we have scaled context-sensitive analyses to large programs even though such analyses typically succumb to an exponential blowup in the data representation. While it is possible for BDDs to represent context-sensitive analyses compactly, it is hard to find such representations. The BDD representation choice (i.e., the BDD variable order) alone can make the difference between an analysis that terminates in a few minutes versus one that does not terminate at all. A considerable amount of manual search and reformulation was needed to make our context-sensitive Java points-to analysis efficient. Our first formulation of a BDD-based algorithm failed due to memory exhaustion even on small programs. It took nearly six months to obtain an algorithm that handled medium-sized programs in three hours, but it still failed to terminate when handling large programs. Our experience taught us that the manual process of finding BDD representations is tedious, unintuitive, and sometimes unfeasible given an analysis or input program. Thus, for bddbddb and, even more generally, BDD-based program analysis to be of any practical use, the search for BDD representations must be automated.

Contributions This thesis aims to make the following contributions to the field:

1. Insights into using BDDs for program analysis. Through the use of bddbddb, we have amassed considerable experience in both developing and optimizing BDD-based programs. This thesis shares many of those insights, and may thus interest those who use BDDs for program analysis.

2. A machine learning technique that automates the discovery of BDD representations for BDD-based program analyses.
3. An effective demonstration of how machine learning and artificial intelligence techniques can be adapted to suit the needs of programming analysis researchers.

Thesis Organization The rest of this thesis is organized as follows: Chapter 2 provides an overview of the process by which bddbddb compiles and executes program analyses. Chapter 3 provides introductory background information on BDD representations and some previous approaches to the problem. Chapter 4 details the active learning algorithm that we use to automatically find BDD representations. Chapter 5 presents the experimental results of our algorithm when used on a variety of program analyses. Chapter 6 discusses related work from both the BDD and machine learning communities. Finally, Chapter 7 presents our conclusions.

Chapter 2

The bddbddb System

This section presents a brief overview of the bddbddb system. Since bddbddb itself is not the main focus of this thesis, readers are encouraged to consult other publications for more in-depth discussions of binary decision diagrams and the implementation of the bddbddb system[24, 53, 50].

2.1 Binary Decision Diagrams

A BDD is a directed acyclic graph (DAG) with a single root node and two terminal nodes, representing the constants one and zero. Each non-terminal node t in the DAG represents an evaluation of an decision variable and has exactly two outgoing edges: a high edge and a low edge. t represents the boolean expression $(t_{var} \wedge t_{high}) \vee (\neg t_{var} \wedge t_{low})$, where t_{var} is a decision variable, t_{high} is the expression represented by the node along the high edge of t, and t_{low} is the expression represented by the node along the low edge of t. The root node represents the boolean expression describing f.

To evaluate f for a specific input, one simply starts at the root node and, for each node, follows the high edge if the corresponding decision variable is true, and the low edge if the decision variable is false. The value of the terminal node that we reach is the value of function f for that input. Figure 2.1(a) gives an example of a BDD implementing the function $b_1b_2 + b_3b_4$. Each non-terminal node t is labeled with a decision variable, and a solid line indicates a high edge while a dashed line indicates a low edge.

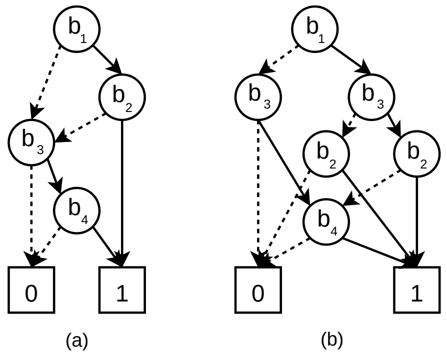

Figure 2.1: BDDs representing the function $b_1b_2+b_3b_4$. The left figure (a) uses the BDD decision variable order b_1, b_2, b_3, b_4, while the right figure (b) uses the BDD decision variable order b_1, b_3, b_2, b_4.

bddbddb specifically uses a variant of BDDs called *ordered binary decision diagrams*, or OBDDs[8]. In an "ordered" BDD, the sequence of variables evaluated along any path in the DAG is guaranteed to respect a given total *decision variable order*. The choice of decision variable order can significantly affect the number of nodes required in a BDD. The BDD in Figure 2.1(a) uses variable order b_1, b_2, b_3, b_4, while the BDD in Figure 2.1(b) represents the same function, with variable order b_1, b_3, b_2, b_4. In the worst case, the size of a BDD can be exponentially larger when using a different variable order. Finding the optimal variable order is an NP-complete problem[6]; thus, we use machine learning techniques to automatically discover effective variable orders.

2.2 BDDs and Program Analysis

Binary decision diagrams (BDDs) were originally invented for hardware verification to efficiently store a large number of states that share many commonalities[8]. Recently, it has been shown that BDDs can also be used to efficiently solve various problems in program analysis[2, 4, 50, 27, 53, 24].

The bddbddb system translates high-level program analysis specifications, written in Datalog, into BDD-based implementations. This allows one to specify and use advanced program analyses without having to worry about the complexities of manipulating BDD data structures.

2.3 Specifying Program Analysis with Datalog

Many program analyses can be succinctly and declaratively specified in a logic programming language called Datalog. A Datalog program consists of a set of domains, a set of relations, and a set of rules. A domain corresponds to a universe of possible values. A relation stores values from one or more domains. A rule specifies how to update a relation based on the values in other relations.

To make the discussion more concrete, we present an example of an actual analysis written in Datalog. Algorithm 1 is an actual Datalog program that implements inclusion-based pointer analysis for Java. It is similar to the analysis of Berndl et al[4].

Algorithm 1 Context-insensitive points-to analysis with a precomputed call graph.

DOMAINS

V	262144
H	65536
F	16384

RELATIONS

input	vP_0	$(\mathit{variable} : \mathrm{V}, \mathit{heap} : \mathrm{H})$
input	store	$(\mathit{base} : \mathrm{V}, \mathit{field} : \mathrm{F}, \mathit{source} : \mathrm{V})$
input	load	$(\mathit{base} : \mathrm{V}, \mathit{field} : \mathrm{F}, \mathit{dest} : \mathrm{V})$
input	assign	$(\mathit{dest} : \mathrm{V}, \mathit{source} : \mathrm{V})$
output	vP	$(\mathit{variable} : \mathrm{V}, \mathit{heap} : \mathrm{H})$
output	hP	$(\mathit{base} : \mathrm{H}, \mathit{field} : \mathrm{F}, \mathit{target} : \mathrm{H})$

RULES

$$vP(v, h) \quad :- \quad vP_0(v, h). \tag{2.1}$$

$$vP(v_1, h) \quad :- \quad \mathit{assign}(v_1, v_2), vP(v_2, h). \tag{2.2}$$

$$hP(h_1, f, h_2) \quad :- \quad \mathit{store}(v_1, f, v_2), vP(v_1, h_1), vP(v_2, h_2). \tag{2.3}$$

$$vP(v_2, h_2) \quad :- \quad \mathit{load}(v_1, f, v_2), vP(v_1, h_1), hP(h_1, f, h_2). \tag{2.4}$$

□

There are three domains: V is the domain of variables in the program, H is the domain of heap objects, and F is the domain of fields. The numbers after the domain specifiers are the maximum number of elements within a domain. There are six relations: vP_0 is the set of initial points-to relations, $store$ is the set of store instructions in the program, $load$ is the set of load instructions, $assign$ is the set of assignment instructions, vP is the set of points-to relations from variables to heap objects, and hP is the set of points-to relations between heap objects. Each of the relations has some number of attributes, each of which has a domain. For example, the vP relation has attribute $variable$ with domain V and attribute $heap$ with domain H.

The analysis consists of four rules. The semantics of Datalog are similar to Prolog: If there exists an assignment to variables that makes the right hand side true, then the left hand side is also made true. Rule (2.1) incorporates the initial variable points-to relations into vP. Rule (2.2) finds the transitive closure over assignment edges. If v_2 is assigned to v_1 and variable v_2 can point to object h, then v_1 can also point to h. Rule (2.3) models the effect of store instructions on heap objects. Given a statement "$\texttt{v}_1\texttt{.f = v}_2\texttt{;}$", if $\texttt{v}_1$ can point to h_1 and $\texttt{v}_2$ can point to h_2, then $h_1.\texttt{f}$ can point to h_2. Rule (2.4) resolves load instructions. Given a statement "$\texttt{v}_2\texttt{ = v}_1\texttt{.f;}$", if $\texttt{v}_1$ can point to h_1 and $h_1.\texttt{f}$ can point to h_2, then $\texttt{v}_2$ can point to h_2.

2.4 From Datalog to BDDs

In bddbddb, relations are represented using BDDs as follows. Each element d in an n-element domain D is represented as an integer between 0 and $n-1$ using $log_2(n)$ bits. A relation $R : D_1 \times \ldots \times D_n$ is represented as a boolean function $f : D_1 \times \ldots \times D_n \rightarrow \{0,1\}$ such that $(d_1, \ldots, d_n) \in R$ iff $f(d_1, \ldots, d_n) = 1$, and $(d_1, \ldots, d_n) \notin R$ iff $f(d_1, \ldots, d_n) = 0$. For example, the vP relation is represented using a BDD with 34 decision variables: 18 decision variables to represent the $variable$ attribute and 16 decision variables to represent the $heap$ attribute.

Each rule is translated into a sequence of relational algebra operations, such as join, project, rename, etc. Each of the relational algebra operations is then translated into its corresponding BDD operation. For example, a relational join corresponds to a BDD "and" operation. While this translation may seem straightforward, we apply a number of both novel and traditional optimizations to the program at both the relational algebra and BDD operation levels. These optimizations yield a sizable performance gain over naïve translation[53].

bddbddb finds a fixpoint solution by successively applying each rule until the relations converge. The execution time of a BDD operation depends on the size of the input and output BDDs, and not on the number of elements in the relation. As we demonstrated in Figure 2.1, the size of the representation can vary greatly with the order of the decision variables. Therefore, a good variable order is essential to an efficient BDD-based analysis.

Chapter 3

Background

3.1 Finding BDD Variable Orders

Finding the optimal order that yields an optimally small BDD for a given function is an NP-complete problem[6]. While much progress has been on this problem in the way of static heuristics[11, 9] and dynamic reordering techniques[43], existing approaches have not proved effective in general[42].

Using BDDs to implement a relational database does provide a few important rules of thumb on how variables need to be ordered. However, the run time depends greatly on the characteristics of the inputs, the performance of an operation can vary significantly during the course of single run, and furthermore, the order that yields the smallest BDD representation does not necessarily have the best performance (due to BDD cache effects). Thus, we often find our primitive rules insufficient for whole program reasoning. By exhaustively exploring a subset of the possible variable orders for a small program, we found that there exists a small number of nearly best variable orders that dramatically outperform the rest. Moreover, the features of these best orders are not immediately intuitive. When given the best order for an analysis that we have worked on for months, we can only begin to postulate *ex post facto* why the order works.

3.2 Learning BDDs

More promising approaches have arisen via sampling[30] and machine learning[14] methods. Grumberg et al. studied the problem of using BDDs in hardware verification[20]. They randomly sampled a number of variable orders, measured the size of the resulting BDD, and then used machine learning techniques to derive the dominant characteristics of good performing orders. From these characteristics they could then generate orders that were comparable to those found using a static technique.

Such an approach, however, would not produce a desirable answer in our case. First, because there are only a very small number of "nearly-best" orders among the huge search space, random sampling may not be sufficiently representative. Second, it may take an unreasonably long time to find the run time for each variable order, especially if they are randomly sampled. Third, an approach that relies only on the total runtime of a program would fail to discover the dynamic behavior of BDD evaluation and, further, miss opportunities for sharing information among similar parts of the program.

3.3 Actively Learning BDDs

Our approach is to formulate the search for dominant program characteristics as an *active learning* problem. Rather than passively learning from a *labeled* set of independent and identically distributed samples, an active learner takes an active role in selecting the data to be labeled. The careful selection of inputs can greatly reduce the number of instances needed to induce a classifier with an arbitrarily high accuracy[1]. In our case, labeling a data set involves profiling the run time of the BDD operations with respect to a given variable order. Active learning is formulated precisely for problems like ours where labeling is costly; we can improve upon random sampling by choosing to label points that provide us with the most information.

We can further adapt active learning to capture whole-program relationships by extending the basic technique with *multi-view decomposition*. By constructing multiple views of the program at various levels of locality, we can exploit whole-program similarity and, thus, expose more program behavior. We can also adapt active learning techniques to our specific problem of optimization. The challenge in variable order optimization is that we need to find a specific and small class of data points in a very large data space, namely those that give the best execution times. Because we are not entirely interested in the classification of the rest of the space (unlike researchers in field of machine learning) we have the opportunity to make this potentially expensive learning process run faster by biasing our search towards areas that may yield good results.

Chapter 4

Active Learning Algorithm

This section provides a description of the BDD variable order problem space, a general formulation of our variable order learning algorithm, and details on the integration and implementation of our algorithm in the bddbddb framework.

4.1 The Problem Space

A relation in an analysis typically has from 2 to 8 attributes, each of which may be allocated up to 60 decision variables. It is not uncommon for a BDD in a program analysis to use about 200 decision variables. Theoretically, there are $n!$ possible orders for each relation where n bits are used. Fortunately, we can reduce the choice of variable orders by exploiting the high-level semantics of relations.

Because bits within a domain represent the same type of data, they are likely to be correlated, and therefore we do not consider permutations of bits within a domain. In any case, elements within a domain can be deliberately numbered so that similar elements will have similar numberings [50]. Therefore, each attribute in a relation can be considered as a unit. For each pair of attributes a_1 and a_2 in a relation, orderings between the pair are reduced to three choices: decision variables for a_1 precede those of a_2, written $a_1 < a_2$, decision variables for a_2 precede those of a_1, written $a_1 > a_2$, decision variables for a_1 are interleaved with those of a_2, written $a_1 \sim a_2$.

The number of possible variable orders in this new model is given by the sequence of ordered Bell numbers[49], which grows rapidly. The number of orderings for 1 through 8 variables are 1, 3, 13, 75, 541, 4683, 47293, 545835.

4.2 Problem Characteristics

As discussed in Section 1, we have performed exhaustive searches over subsets of the search space of BDD variable orders. Our observations from this experience, as outlined below, led us to the conclusion that active machine learning is the correct approach to solving this problem.

The problem is complicated; simple heuristics do not work well. We have spent several months studying the efficient variable orders discovered by our exhaustive search in order to define general heuristics. We were unable to find any such heuristics. However, We were able to find some small insights; for example, if two variables are required to be equivalent, then interleaving their bit representation is effective. Also, the orders used for large input BDDs can be used as a guide. However, these small insights do not lead to the efficient variable orders that we found.

Some features characterize the problem well. The relative order of a particular subset of all variables typically decides the execution time of the ”nearly best” answers. This order can be represented as a set of pair-wise ordering constraints or, as we will refer to them, features. However, one must be careful during algorithm design as some of the features may be correlated.

Presence of critical rules. The performance of an analysis is often dominated by a small subset of the rules. While some optimization decisions can be considered on a rule-by-rule basis, there may still be different order preferences among a set of rules. Ideally, a relation used by two different rules should have the same order so as to avoid potentially expensive reorderings. Therefore, information from rules that operate on the same relations/domains should be shared.

Dynamically varying behavior. Different applications of the same rule typically favor the same variable ordering; however, that is not necessarily the case. In the iterative process of resolution, the nature of the inputs for a rule may change and, thus, the representation may require a different variable order.

The search space is not smooth. Because certain features can have a significant impact on performance and these features are often correlated, the search space is not smooth. The best variable order may be very similar to one which blows up; the difference may be merely one or two features.

Trials are expensive. It is not unusual for a rule application to take two minutes to execute even with a relatively good variable order. Good orders can outperform bad ones by many orders of magnitude.

Feasibility is not guaranteed. There are problems for which there is no order that yields a compact answer. Sometimes, it is possible to manually improve the Datalog query of an analysis by avoiding the generation of large intermediate sets of data. However, there is still no guarantee as to whether the analysis will finish.

In sum, the BDD variable order problem is not easily conquered by simple heuristics. Those few that we have found do not produce accurate generalizations. In our experience, profiling has been the only technique to yield results. However, knowing what to profile is difficult to discern manually as the space of variable orders is so large. Exhaustive exploration has, until now, proven to be our only reliable resource. But, as our program analyses have become larger and more complex, this approach has become prohibitively expensive.

4.3 Approach and Formulation

Our solution to the BDD variable order problem adopts an iterative *sample-learn-extract* approach. For a particular set of variables, we sample a subset of the variable order space and label each instance with a quality metric. From the set of labeled orders we employ machine learning algorithms to learn a classification system that maps arbitrary variable orders to predictions of quality. We then extract the relevant features from our classification system. By iterating the algorithm and summing our samples across iterations, we can incrementally create better, more general classification systems. This entire process is tightly integrated into bddbddb.

4.3.1 Sampling

Random sampling is a common sampling technique for applications such as ours. However, we improve on random sampling by incorporating previously learned information into the sampling decision. By uncertainty sampling the space, we sample those orders for which we have the least information; their labeling and addition to the sample set yields the largest gain in information. Furthermore, this technique produces higher quality classification systems with fewer data samples [12, 25].

4.3.2 Learning

A wide variety of algorithms for constructing classification systems exist. Through the empirical evaluation of several classification algorithms, we have found decision trees to be well suited for our problem as they are computationally inexpensive to create, comprehensible in structure, and give quality classifications for our domain. Decision trees represent a class of algorithms that recursively partition data by feature values to make classifications. We use an ensemble of perceptually different decision trees to both increase classification quality and capture whole program relationships.

4.3.3 Extraction

Feature extraction for our domain is non-trivial. Due to possibly inconsistent requirements different rules, dynamically changing inputs, and the inherent fuzziness of the BDD variable order space, greedy heuristics do not work well for collecting consistent sets of features. To address these issues, feature extraction is modeled by a Fuzzy-Constraint Satisfaction Problem (FCSP). Fuzzy constraints associate a real-valued notion of satisfaction to a constraint. For example, in addition to the traditional boolean notions of a fully satisfied and a fully violated constraint, fuzzy constraints can be partially satisfied. This representation allows us to encode fuzzy variable order preference information and, thus, generate a number of ranked solutions.

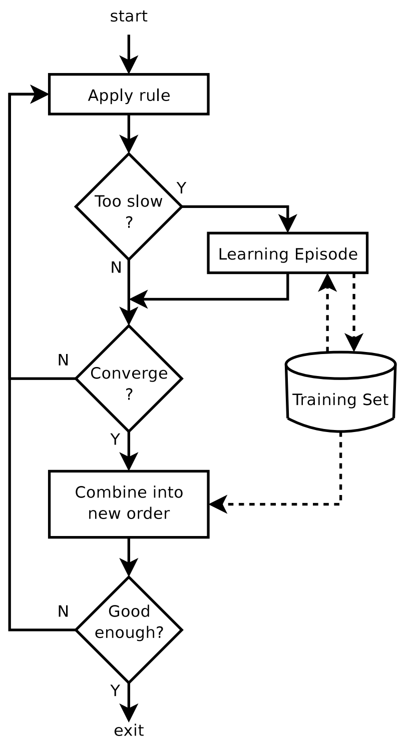

Figure 4.1: The learning algorithm embedded in Datalog resolution.

4.3.4 Integration

Figure 4.1 shows how we integrated machine learning into bddbddb. The system starts by applying a Datalog rule to infer new relations. If the application of the rule takes longer than a specified threshold, we invoke the active learning algorithm. The active learner uses a training set of data that persists across different invocations of bddbddb. If the learner performs any new trials, they are in turn added back into the training set. The system continues applying rules until the relations converge. At that point, we use the data in the training set to infer and report the information discovered. From this report we can take a number of actions, such as refining or generating an order. We can then evaluate the new variable order or refine it further by repeating the entire process until we obtain a satisfactory variable order. Repetition ensures that all relevant rules are analyzed as a new, learned order may run slower on rules that previously executed quickly.

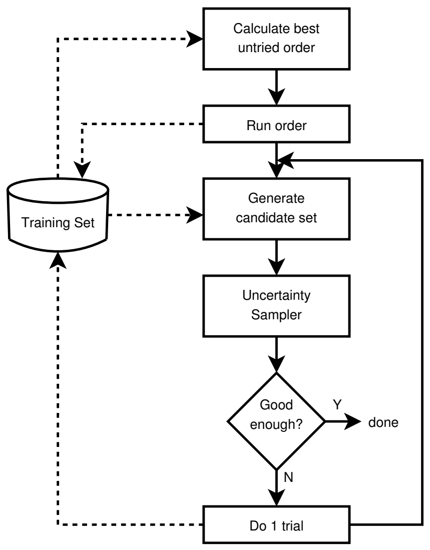

Figure 4.2: Steps of a learning episode. This corresponds to the contents of the Active Learner box in Figure 4.1.

4.4 Algorithm Details

Figure 4.2 shows the steps of one invocation of the learning algorithm, called a learning episode. First, we build a candidate generator (described in Section 4.4.2) from the training set (Section 4.4.1) to predict the best untried order. This order serves as a baseline measurement, allowing us to abort rule applications for which sampled orders take significantly longer than our baseline. This is important as rule applications with poorly performing sampled orders can take exponentially longer to complete. Moreover, the quality of this order also gives us an online indication of the quality of our classification system. We then generate a candidate set of orders from which an uncertainty sampler (Section 4.4.3) selects an order, or set of orders, to add to the trial set. If the trial set is non-empty, these orders are tried, the results are incorporated into the training set, and the process repeats; otherwise, we end the learning episode.

Episode	Rule	Update	Operation	Order	Time	Class
1	2	1	1	$v_1 < v_2 < h$	5ms	1
				$v_2 < v_1 \sim h$	15ms	1
				$h < v_2 < v_1$	50ms	2
2	3	1	1	$v_1 < f < h_2 < v_2 < h_1$	∞	1
				$v_2 < h_1 < h_2 < f < v_1$	1000ms	1
3	2	2	3	$v_2 \sim v_1 < h$	10ms	2
				$h \sim v_2 < v_1$	50ms	2

Table 4.1: The grouping of the data entries in our training set. Each episode is tied to a particular rule, rule application, and operation within the rule. Within each episode, the order along with its run time is stored. The class column is used only for the example presented by Figure 4.3.

4.4.1 Data Sets

The *training set* consists of the learning episodes we have executed thus far. Table 4.1 depicts the association of each learning episode with a particular rule, rule update, and BDD operation within that rule, along with the set of trials that it encompasses. Each trial of the episode is paired with its execution time. If there are no previous trials at the beginning of a new episode for a particular rule, rule update, and operation, we bootstrap the training set by trying a set of randomly generated orders.

The *candidate set* is the reduced space of orders that we would like to consider. As mentioned earlier, the candidate generator creates this set by generating partial orders that exhibit features that are either unseen or known to be good. The candidate generator then populates the candidate set with the enumeration of all total orders that follow from these partial orders.

In the vein of active learning, the *trial set* is the subset of orders in the candidate set that have been actively selected for labeling. Our adoption of uncertainty sampling dictates that we add to this set those orders that we know the least about. We estimate this criterion by find those orders for which the probability of being within the top class of orders is closest to 50%. If no orders are sufficiently close (given by a parameter), this set is empty.

4.4.2 Candidate Generator

A classifier can be loosely described as a function that accepts a vector of features and returns a classification. For our problem, as discussed in Section 4.1, we use pairwise ordering constraints as "features": given two elements, one can either occur before ($<$), after ($>$), or be interleaved ($\sim$) with the other. Features in a given variable ordering should form a partial order. A classification is an element in a set of *performance classes*. This set of performance classes is determined by the discretization of run-times into a finite number of classes.

Multi-View Decomposition

Although each learning episode focuses on just one rule, information learned from other rules can also be relevant because these rules may operate on the same relations or the same domains. We use *multi-view decomposition* to integrate information from all these different sources. Therefore, the candidate generator is actually composed of the following three classifiers induced from three translations of the training set.

1. For each rule, we have a "variable" classifier to gauge the utility of an order on the operations of just that particular rule. The features for these instances are the pairwise orderings of the rule's variables. Thus, for Rule (2.2) in Algorithm 1, these features are the pairwise orderings of v_1, v_2, h.
2. The "attribute" classifier incorporates global information about the quality of orders over the relational attributes used by a rule. For a classifier of this type, we translate variables to their relation attributes. Thus, for Rule (2.2), the features used are the pairwise orderings of $vP_{variable}$, vP_{heap}, $assign_{dest}$, $assign_{source}$. Translations that lead to contradictions are dropped.
3. The "domain" classifier makes reasonable, but possibly imprecise, decisions about the quality of an order as it relates to all rule applications that have used a related set of domains. This classifier is best quantified as a binary decision between truly poor orders and ones that stand a reasonable chance of termination. A classifier of this type for Rule (2.2) uses the pairwise orderings of the domains of its variables:

(V,V) and (V,H). As in the attribute classifier, translations that lead to contradictions are dropped.

Classifier Induction

Each feature vector is paired with the performance, measured in run time, of the variable order from which it was derived. We have investigated machine independent metrics for order performance to allow for portable datasets, but we have found these metrics to be difficult to extract and imprecise in nature. To compare trials over the span of different rule applications, trials are normalized against the best observed run time for a rule application. To discretize the run time results, we use an equal frequency binning algorithm to extract relative performance classes from our continuously scored trials. In our experience, this approach is preferable to an equal width or clustering approach as we gain the explicit interval refinement of clustering with the lower overhead of equal width binning.

An appropriate number of bins must be chosen to reflect the degree of precision expected. The “rule,” “attribute,” and “domain” classifiers are progressively less precise as each incorporates more, possibly tangential or incomplete, information. Thus, the number of performance classes for these three classifiers are $\sqrt{n}$, $\sqrt[4]{n}$, and 2, respectively, where n is the number of trials seen. The domain classifier incorporates so much conflicting information that we have found it useful only for distinguishing between acceptable and extremely poor orders. Hence, the domain classifier has only two bins.

Classifying an instance with a decision tree begins by querying the value of the attribute designated at the root node of the tree. This value is then used to select the next node to test against. This procedure continues recursively until it reaches a leaf node. The classification recorded for this leaf node is returned as the classification for the instance. The value of this classification can vary with application. For discrete applications, the classification of the majority of training set instances that occupy the node is often returned. Also, as shown in Figure 4.3, a node may have no successor for a particular attribute value. In this case, a value denoting no classification is returned. We later exploit this fact to encourage the algorithm to explore unseen features.

There many extensions to this basic algorithm that are tailored for different notions of classification accuracy and speed. Our classifiers are induced using Quinlan’s simple

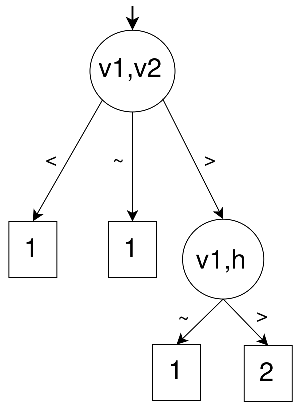

Figure 4.3: An example decision tree induced from the training set data in Table 4.1 on rule 2.

ID3 algorithm[40]. ID3 partitions a dataset on the attribute with the highest information gain (IG). Information gain computes the estimated reduction in entropy resulting from the partitioning of the dataset on a particular attribute. By giving priority to attributes with a higher information gain, we aim to produce short trees with high degrees of generalization. Information gain can be computed as follows:

$$I(S) = \sum_{c \in S}^{n} -p_c \log p_c$$

$$I(A,S) = \sum_{v \in A} \frac{|S_v|}{|S|} * I(S_v)$$

$$IG(A,S) = I(S) - I(A,S)$$

$I(S)$ is the information (or entropy) of a set of instances (S) where p_c is the proportion of instances of classification c in S. $I(A,S)$, the information of a set of instances (S) given an attribute (A) partition, is the weighted sum of information for each subset of instances grouped by attribute value (v). Thus, $IG(A,S)$ is the information gained by a partition of instance set S on attribute A.

For each node in the tree, the information gain of each attribute is computed. The set of instances is then partitioned by the attribute with the highest expected information gain. The algorithm then recurses on each subset until no further information can be gained by partitioning.

Candidate Set Generation

For each rule, the candidate generator is used to generate both the best untried order for baseline calibration and the candidate set from which the uncertainty sampler chooses the trial set. As mentioned before, we wish to run the fastest order first in order to establish a baseline execution time for this operation. If subsequent trials begin to take substantially longer than our baseline, we abort the computation and assign it a large value for execution time.

To find the best predicted order we combine the information from the variable, attribute, and domain classifiers as follows: A performance class is assigned a score equal to the mean run time of the members of that class. The scores from each classifier are weighted by a factor related to the history of its accuracy. Correct classifications increase the classifier weight, incorrect classifications decrease it. We use an exponential decay function to diminish the penalties for mispredictions that occurred in the past. This provides a feedback mechanism that automatically adjusts the weights based on the actual problem; depending on the problem, certain classifiers may be more or less accurate than others.

Because there are typically far too many orders to iterate over for an operation, we extract the features that lead to the best order directly from the classifiers. This problem is complicated by the fact that we have three separate classifiers and that the progression of scores in the performance classes of the classifiers is non-linear. We solve this by first computing the scores from every possible combination of classes and then sorting the combinations by their scores. We use the sorted combinations of classes to extract the features from each classifier and combine them, skipping combinations that lead to constraints that cannot be combined.

The features collected so far represent a set of constraints. A simple procedure can enumerate all the total orderings that satisfy these constraints. From this set of total orderings we can select the best predicted untried order or the top n candidates to populate the candidate set. To include in the candidate set feature combinations that have not been evaluated before, nodes with no instances in the decision trees (i.e., leaf nodes that return no classification), are boosted to have the highest performance class.

Since we bootstrap the process by a set of randomly generated orders (Section 4.4.1), the initial order is never considered unless it is independently generated by this process.

4.4.3 Uncertainty Sampler

The goal behind our selection of variable orderings is to maximize information gain by trying orderings that carry the greatest uncertainty. While this may seem orthogonal to our goal of optimization, it has been shown that relevance sampling[47] (selecting the input classified as the best) produces poor classifiers compared to those produced by uncertainty methods[26].

For our problem, we are interested in knowing if a given variable ordering falls in the top performing class. We refer to the top performance class as the positive class and to the rest as the negative class. An instance is given probability 1 if it definitely belongs to the positive class and 0 if it definitely does not. Thus a probability of .5 represents maximum uncertainty.

Probability Estimate Trees (PETs) have been shown to be useful Class Probability Estimators (CPEs). A primitive PET constructed from a decision tree returns the distribution of instance classifications at a leaf node, rather than a single classification. This distribution is then interpreted as the probability that an instance with the given features is in a particular class. However, this is a poor measure of uncertainty in the case where the number of instances at a node is small. We smooth probability estimation for relatively unpopulated nodes by computing $\frac{k_c+1}{n+C}$, where k_c is the number of instances with label c in the population of the leaf node, n is the total number of instances at that node, and C is the total number of classes. In our case, since we are distinguishing between a positive and a negative performance class, $c = 2$. Thus, the probabilities of nodes with fewer instances are shifted toward .5. This technique is known as Laplace correction.

A Bagged Probability Estimate Tree (BPET) constructs k trees, where each is induced from a *bootstrap sample*[15] of the training set. A bootstrap sample is a random sample with replacement. The class probability estimate for the BPET is the average of the probability estimates over all k randomly permuted trees. Just as in multi-view decomposition, an ensemble of perceptually different classifiers often leads to more robust classifications and, hence, more accurate class probability estimates.

We use Laplace correction and BPETs to create three estimators—one each for rules, attributes, and domains—from the training set. The probability estimate for a particular variable ordering is computed by combining the probability estimates of each estimator. The score attributed to a given variable order from the candidate set can be computed as the distance from a chosen centroid by using a root-mean-squared computation:

$$\sqrt{\frac{(E_r - R)^2 + (E_a - A)^2 + (E_d - D)^2}{3}},$$

where $P_r, P_a,$ and P_d are the class probability estimates returned by our rule, attribute and domain classifiers, and $R, A,$ and D are the centroids of focus for our search. As uncertainty sampling dictates, to choose the order that has the most uncertainty, we would set R, A, and D to be .5, and find the order with the smallest score. Thus, a score of 0, given these centroids, would indicate that our all three of our estimators predicted a class probability of .5. However, we have found that by adjusting these centroids appropriately, we can direct the focus of our search: a lower centroid biases the search toward more exploration, while a high centroid directs the search to areas of higher confidence. Since our domain classifier data is best suited for determining extremely poor orders, we bias its centroid toward searching orders it deems relatively good. As a result, the initial trials for newly encountered rules typically follow the optimal global scheme.

4.5 User feedback

At the end of the learning process, we can present a variety of outputs for the user to query. For each rule on which a learning episode has executed, we can retell the top n-best executed orders. Moreover, for any rule that has variable, attribute, or domain information associated with it (including those not directly considered during learning) we can generate the top n-best predicted orders or the constraints associated with those orders as in Section 4.4.2. Another more useful utility is the generation of global constraints that can be used to generate an entirely new order or to improve an existing order.

Global constraint generation is complicated by the fact that we have a distributed record of runs; our trials span across rules of varying runtime and hence, importance. Also, dominant performance features for a collection of rules are not necessarily consistent. As a result, we have constructed this problem as a Fuzzy Constraint Satisfaction Problem (FCSP) [44].

A FCSP can be defined by the tuple (V, D, C, J), where V is a set of variables, D is a list of domains for the variables, and C is a set of fuzzy constraints. J is a function that computes the *joint satisfaction* of a partial assignment of values from D to variables in V over the fuzzy constraints in C. For our purposes, the set of variables is the set of physical BDD domains. D consists of a single integer domain of values between 1 and $|V|$ that is shared among all variables. Thus, one can consider a variable assigned a value as an assignment of a domain to a fixed position in a total ordering of all domains. This allows us to express precedence constraints and interleaved constraints if we consider domains assigned to the same position as interleaved.

A fuzzy or soft constraint is a mapping of tuples of values for its input variables to a local degree of satisfaction, measured as a floating point value on the interval [0,1] (where 0 denotes full violation and 1 denotes full satisfaction). A solution to a FCSP thus consists of a complete assignment of values from D to all variables of V and an estimate of the solution's global or joint satisfaction of the constraints. Ruttkay and others have defined several means for determining the joint satisfaction of a set of constraints. For instance, joint satisfaction can be computed as the minimum local satisfaction among all constraints, the multiplicative product of the local satisfactions of all constraints, or the average satisfaction of all local satisfactions. We have chosen to maximize the average of the local satisfactions as it seems the most applicable to our problem.

Several heuristic methods have been proposed for solving FCSPS [21]. Of these, a branch and bound approach is the most straightforward. This technique relies on the fact that the joint satisfaction of a partial assignment serves as an upper bound for any further extension of the assignment. This allows us to prune sections of the search tree. The algorithm proceeds as follows: it first checks whether the current joint satisfaction is lower than the joint satisfaction of the best solution thus far. If so, further exploration of this subtree is avoided. Next, if the current assignment is a complete assignment, it is recorded as a solution and execution returns to the previous level. Otherwise, the algorithm chooses a variable from the set of variables that have yet to be assigned. For each value in this variable's domain, the assignment is extended and the algorithm proceeds recursively. It is common practice for the ”most constrained” variable to be selected and the values to be iterated in the ”least constraining” order so as to maximize the likelihood of finding a solution early. These ordering heuristics are derived from perturbations of the joint satisfaction.

By giving priority to rules with longer execution times, the fuzzy constraints for this algorithm can be generated to simulate weighted voting. For each rule, sets of constraints that lead to high performing orders are generated as described in Section 4.4.2. Each individual constraint is given a score computed as its parent constraint set's performance multiplied by the total execution time of the rule. The votes for a pair of BDD domains and its different orientations (before, after, or interleaved) are totaled globally and a fuzzy constraint for the pair of domains is created. This fuzzy constraint maps a physical assignment of the two domains to an associated orientation and returns the proportion of votes for that orientation as its local satisfaction.

The physical assignment outputted by the FCSP can then be enumerated to generate the selected ordering constraints. These constraints are then presented to the user, and either a new order is generated or, in the case when small or local learning was performed, an existing order can be made consistent.

Chapter 5

Experimental Results

This section presents the experimental results we obtained by using our system to discover variable orders for various BDD program analyses.

5.1 Methodology

To evaluate the effectiveness of our technique, we used the active learning algorithm to discover variable orders for a variety of analyses. We then evaluated the quality of the resulting orders.

Name	Description	Rules	Relations	Domains	Orders
j_pa	Java context-insensitive pointer analysis	8	20	12	2.8×10^{10}
j_pacs	Java context-sensitive pointer analysis	12	25	14	2.3×10^{14}
j_paos	Java object-sensitive pointer analysis	9	20	15	5.3×10^{15}
sqlinject	SQL injection query for Java	20	19	17	3.9×10^{18}
c_pa	C context-insensitive pointer analysis	222	85	13	5.3×10^{11}
c_pacs	C context-sensitive pointer analysis	249	112	14	1.0×10^{13}

Figure 5.1: Information about the analyses that we used to evaluate our BDD order finding algorithm. The four columns of numbers are the number of rules, relations, and domains in the input Datalog file, and the number of possible domain orders.

Figure 5.1 shows the list of analyses that we used to evaluate the effectiveness of our algorithm. Many analyses written for bddbddb are simple and execute quickly, so we focused on the ones we have found to be the most time-consuming because they require the computation of a fixed-point solution. These analyses are the ones that can benefit the most from improving the variable order.

Although, from their descriptions, some of these analyses may appear to compute similar information, the listed analyses have vastly different performance characteristics. The extra context information in the context-sensitive analyses completely dwarfs the rest of the analyses as compared to their context-insensitive counterparts. Likewise, the C analyses are very different from the Java analyses. The C analyses are far more complex, having to deal with internal pointers, pointer arithmetic, pointers to stack variables, etc.; this extra complexity is evidenced by the fact that the C analyses have 20 times as many Datalog rules as the Java versions.

Of all the analyses, the context-sensitive C pointer analysis poses the greatest difficulty for our algorithm because the high number of rules and relations lead to an extremely large search space. Good variable orders within the space are rare, and optimal orders for one rule often conflict with those for other rules.

We performed all experiments on an AMD Opteron 150 with 4 GB of RAM running RHEL v3. The bddbddb system uses the JavaBDD library[52], which is based on the BuDDy BDD library[28]. For all experiments, we used an initial node table size of 10M and a cache size of 2M. We used our hand-tuned order to generate the input relations for learning (as stated in Section 4.4.2, this has no effect on the learning process). The initial relations for the Java analyses were generated by the Joeq compiler infrastructure[51]; the C analyses used the SUIF compiler[54]. We trained the Java analyses using `joeq` as an input program; for the C analyses, we trained on `enscript`.

We used the following parameter values:

- minimum run time for an operation to be considered: 100ms/10ms[1]
- minimum number of orders to evaluate per operation: 10
- top proportion of performance classes to add to candidate set: 1/2
- number of candidate orders for evaluation: 500
- location of centroid (see Section 4.4.3): (0.5,0.5,1)
- cutoff score for deciding whether a trial should be performed: 0.25

[1]For the C analyses, we used a value of 10ms. For all others, we used 100ms.

- delay past best time before killing a trial: 10s

Both of the C analyses have a large number of rules that execute many times very quickly. We found that with a minimum time threshold of 100ms, too few rules were considered during the learning process; this led to poor performance as the learning algorithm did not have any information about the missing rules. As a result, we lowered the threshold for these two analyses to 10ms.

5.2 Results

Name	Random	Sifting	Hand-tuned	Active Learning	Episodes	Trials	Unique Orders	Learning time
j_pa	379s	107s	11s	9s	132	636	97	38m
j_pacs	∞	∞	211s	209s	1006	2554	1294	4h 14m
j_paos	∞	∞	53s	51s	160	1301	819	1h 57m
sqlinject	∞	∞	39s	33s	213	1450	1101	2h 15m
c_pa	30s	1570s	19s	4s	238	1901	1714	3h 7m
c_pacs	∞	∞	220s	65s	516	2784	2601	6h 47m

Figure 5.2: The results of our learning algorithm. The first four columns of numbers compare the speed of a random order, an order generated with a sifting algorithm, our best hand-tuned order, and the order output by the algorithm. ∞ means that the analysis did not complete because it ran out of memory. The next four columns give statistics on the performance of the learning algorithm.

Name	Description	Classes	Methods	Bytecodes
sshdaemon	SSH daemon	485	2053	115K
azureus	Java bittorrent client	498	2714	167K
jgraph	graph-theory objects and algorithms	1041	5753	337K
umldot	makes UML class diagrams from Java code	1189	6505	362K
jxplorer	ldap browser	1927	10702	645K

Figure 5.3: Brief program size information detailing the number of classes, methods, and bytecodes for the input programs benchmarked in Figure 5.4.

Figure 5.2 contains the results of our learning algorithm. We compared the execution times to those of a randomly-generated order, an order generated by a sifting algorithm,

Name	j_pa		j_pacs		j_paos	
	Hand-tuned	Learned	Hand-tuned	Learned	Hand-tuned	Learned
sshdaemon	5s	4s	41s	39s	9s	8s
azureus	5s	4s	48s	47s	9s	8s
jgraph	17s	12s	312s	300s	48s	45s
umldot	19s	13s	216s	214s	78s	73s
jxplorer	38s	32s	581s	574s	26s	25s

Figure 5.4: A comparison of the run times of our hand-tuned and generated orders for the j_pa, j_pacs and j_paos analyses on several different input programs.

and to our best known, hand-tuned variable order. The random order was very poor and was able to complete in only two of the cases. The sifting algorithm, a traditional dynamic BDD variable order optimization technique [43], also performed poorly.

The hand-tuned variable orders were the best orders we knew about before running our learning algorithm. The fact that our algorithm was in all cases able to find an order that was competitive with or soundly beat our "best" hand-tuned variable order without any kind of seeding or user direction is heartening. This seems to indicate that our algorithm is not getting stuck in local minima and is effective in finding a good order.

In the cases of j_pa and j_pacs, we had actually run a mostly-exhaustive search of all possible variable orders to obtain our hand-tuned orders. These searches took many days to complete. When our learning algorithm did not return the order we had found by exhaustive search, we were initially dismayed. However, after we tried the orders from the learning algorithm we found that one of them was actually *faster* than the supposed "best"! Upon closer investigation, we realized that we had run the exhaustive search on an older machine with a smaller operation cache size and an older version of the BDD library. The combination of a newer machine, a newer version of the BDD library, and a larger operation cache size changed the best order for the analysis.

Figure 5.4 shows the generalization of the generated orders over the other input programs listed in Figure 5.3. The algorithm proves to generate general orders as these orders show performance gains in all instances.

Chapter 6

Related Work

Finding BDD variable orders. BDD variable order optimization is a well-studied problem. Over the years, researchers have proposed static heuristics, dynamic reordering techniques and, more recently, search and sample techniques.

Static approaches seek to develop heuristics by which BDD orders can be statically synthesized by structural inspection of the boolean function itself (typically a circuit description). Fujita and others have proposed techniques that inspect properties such as variable depth and pair-wise variable distance [17, 11]. While static techniques are computationally inexpensive to apply, they do not produce high quality results for many applications. In our case, the primitive properties that these methods inspect are not readily producible for our Datalog programs. Moreover, these methods would fail to capture the dynamic properties of our problem.

Dynamic reordering techniques apply variable order modifications to the BDD as it is manipulated in real-time. Sifting, the canonical dynamic technique, continually swaps adjacent BDD variables until a reasonable order is found [43]. Others have developed more advanced modification algorithms, such as variable grouping[37], for the sifting framework. While these methods have found many practical applications, they are extremely expensive in both memory consumption and computation time and often get stuck in local minima. In addition, these methods fail to provide an intuitive decomposition of variable orders into dominant features.

Recent work has focused on sampling-based search and learning processes. Bollig improved on previous efforts to develop a simulated annealing solution [5, 33]. Other researchers have also used used genetic algorithms[14] and scatter search[23] to find good variable orders. While these approaches can generate large sets of variable orders from which one could extract similarities, they fail to explicitly provide any intuitive abstraction of the problem. Moreover, these algorithms are tailored for optimizing one BDD as opposed to the dynamic multi-BDD environment we must deal with. However, their use static heuristics to bootstrap the search process and dynamic reordering techniques to define sample neighborhoods could complement our learning approach.

Machine learning BDD variable orders. Grumberg et al. has tackled the BDD variable order problem with machine learning[20]. Their framework is inherently different from ours, as they lack the high-level notion of domains and, instead, seek to order individual BDD variables. Thus, we can search more effectively, as their search space is factorial in the number of individual BDD variables, whereas ours is exponential in the number of domains. Their implementation also differs significantly: while they mention active selection of training data as a possibility, their results are based on the selection of relatively arbitrary orders. Also, they use low-level BDD properties (e.g. , variable connectedness) as features in their machine learning algorithms, while we have chosen the pairwise ordering of domains. Moreover, they evaluate the performance of an order based on the size of the BDD it creates, while we have noted that smaller BDDs do not necessarily mean faster BDD operations as cache behavior plays an important role.

Probability Estimate Trees. A number of extensions to Probability Estimate Trees (PETs) have been suggested to increase their usefulness as Class Probability Estimators (CPEs). Laplace correction and m-estimators[10, 39, 55] can smooth the jagged probability curves of standard PETs. Bootstrap aggregating (Bagging)[7], has also been shown to increase the effectiveness of PETs[3, 39, 46]. Our approach employs both Laplace correction and bagging. In addition to Laplace correction, m-estimation, and bagging, researchers have proposed probability-focused decision tree induction algorithms, splitting criteria[38], and tree growth conditions[32, 55] aimed at increasing PET performance. However, some approaches appear better suited for large data sets than small data sets such as ours.

Ensemble classification. Ensemble classification proposes that through the combination of an ensemble of perceptually different domain experts one can increase the accuracy of classification. Bootstrap Aggregating (Bagging) increases performance through the vote of several classifiers, each built upon different bootstrap samples of the original data[7]. Boosting, in its various incarnations, combines the vote of trees that are sequentially generated from reweighted data[16]. Furthermore, rather than alter the data representation, Ho et al. have found that a combination of classifiers induced by stochastic algorithms can yield gains in classification performance[22].

Bagging, where trees can be substituted for any classifier, has been shown to work well with unstable classifiers—classifiers in which small variations in data produce large variations in prediction[3]. Given the unstable nature of decision trees and the nature of our domain, where the lack of a single feature can mean the difference between a good order and one of exceptionally poor performance, we believe these methods match our domain well. Moreover, Zhou et al. have observed that bagging performs better than similar algorithms on problems with fewer data points[56], another characteristic of our domain. Moreover, many other researchers have achieved positive results with bagged decision trees[3, 7, 13, 31, 41].

Active learning. Active learning and its cousin active classifiers[19] have found a niche application in settings where a premium is placed on the proper selection of inputs as oftentimes data collection is expensive. Thus, the means by which new instances are selected have received a variety of treatments. Sueng and Freund have proposed that the information gain of a candidate instance can be quantified by the level of disagreement among an ensemble or committee of classifiers[48]. Saar-Tsechansky and Provost have put forth the notion that the potential gain for an instance is related to the variance in probability estimates over an ensemble of class probability estimators[45]. Their results have shown that variance-based methods can substantially outperform uncertainty sampling when paired with weighted sampling: new instances are selected from a distribution closely related to the variance of probability estimates across the entire space of candidate instances. However, in our empirical evaluation, we found that the sampling decisions made by variance-based methods do not intuitively follow from the information learned thus far. Since observation of the automated learning process can be a valuable tool for reasoning about an analysis, we have instead used uncertainty sampling. Furthermore, we believe we suffer no substantial loss of learning quality as Saar-Tsechansky and Provost also found weighted uncertainty sampling to be competitive with their variance-based method[46].

Multi-view active learning. Muslea et al. have explored the area of multi-view active classification with their co-testing framework, wherein multiple classifiers are constructed from disjoint sets of features and instances selected based on the degree and confidence of misclassification among these views[34]. They found that the strengths of multi-view approaches lie in their ability to identify rarely occurring, but still relevant, instances from sparse data sets. The BDD variable order space exhibits this exact property and, thus, believe that a multi-view approach, such as ours, is crucial to the success of a learning algorithm in this domain.

Chapter 7

Conclusions

This thesis presents an active learning algorithm that automates BDD variable order discovery for BDD-based program analyses. There are only a small number of variable orders that deliver ”good” performance and their performance can be orders of magnitude better than that of other, unoptimized orders. However, searching, labeling, and classifying the BDD variable order space is expensive.

Our algorithm uses a number of techniques to make the search effective. To maximize the information gained by each variable order sample, we use uncertainty sampling to find the variable orders that we are the most uncertain about. We take samples on a rule-by-rule basis and then assemble global perspectives with multi-view classification techniques. We then further bias the search towards using features that are known to be effective in order to prevent the algorithm from investing a significant portion of time investigating undesirable areas of the search space.

Our experimental results lead us to conclude that machine learning is well-suited for the BDD variable order problem. The orders we discovered are competitive and, in some cases, better than the best variable orders obtained either by hand or through multi-day, exhaustive searches. Moreover, machine learning, unlike other methods, can concisely extract the core performance features of the problem, giving analysis writers an essential aid for writing analyses in bddbddb. With the help of this work, bddbddb represents a breakthrough in the area of program analysis. While Datalog and BDDs have each been independently used in program analysis, no other tool has married the two in a way that provides a concise, automatically optimizing program analysis framework to aid researchers in the pursuit of code quality.

Bibliography

[1] Dana Angluin. Queries and concept learning. *Mach. Learn.*, 2(4):319–342, 1988.

[2] Dzintars Avots, Michael Dalton, V. B. Livshits, and Monica S. Lam. Using C pointer analysis to improve software security. In *Proceedings of the 27th International Conference on Software Engineering*, May 2005.

[3] Eric Bauer and Ron Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Mach. Learn.*, 36(1-2):105–139, 1999.

[4] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 103–114, June 2003.

[5] B. Bollig, M. Lobbig, and I. Wegener. Simulated annealing to improve variable orderings for obdds, 1995.

[6] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, September 1996.

[7] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[8] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[9] Kenneth M. Butler, Don E. Ross, Rohit Kapur, and M. Ray Mercer. Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pages 417–420, New York, NY, USA, 1991. ACM Press.

[10] B. Cestnik. Estimating probabilities: A crucial task in machine learning. In *Proc. of the 9th ECAI*, pages 147–149, Stockholm, Sweden, 1990.

[11] Pi-Yu Chung, Ibrahim N. Hajj, and Janak H. Patel. Efficient variable ordering heuristics for shared ROBDD. In *ISCAS*, pages 1690–1693, 1993.

[12] David Cohn, Les Atlas, and Richard Ladner. Improving generalization with active learning. *Mach. Learn.*, 15(2):201–221, 1994.

[13] Thomas G. Dietterich. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Mach. Learn.*, 40(2):139–157, 2000.

[14] Rolf Drechsler and Nicole Göckel amd Bernd Becker. Learning heuristics for OBDD minimization by evolutionary algorithms. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature – PPSN IV*, pages 730–739, Berlin, 1996. Springer.

[15] B. Efron and J. T. Robert. *An Introduction to the Bootstrap*. Chapman and Hall, 1993.

[16] Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In *International Conference on Machine Learning*, pages 148–156, 1996.

[17] M. Fujita, H. Fujisawa, and Y. Matsunaga. Variable ordering algorithms for ordered binary decision diagrams and their evaluation. In *IEEE Transactions on Computer-Aided Design*, volume 12, pages 6–12, January 1993.

[18] John P. Gallagher, Kim S. Henriksen, and Gourinath Banda. Techniques for scaling up analyses based on pre-interpretations. In *Proceedings of the 21st International Conference of Logic Programming*, pages 280–296, 2005.

[19] Russell Greiner, Adam J. Grove, and Dan Roth. Learning cost-sensitive active classifiers. *Artif. Intell.*, 139(2):137–174, 2002.

[20] Orna Grumberg, Shlomi Livne, and Shaul Markovitch. Learning to order BDD variables in verification. *Journal of Artificial Intelligence Research*, 18:83–116, 2003.

[21] Hans W. Guesgen and Anne Philpott. Heuristics for solving fuzzy constraint satisfaction problems. In *ANNES '95: Proceedings of the 2nd New Zealand Two-Stream International Conference on Artificial Neural Networks and Expert Systems*, page 132, Washington, DC, USA, 1995. IEEE Computer Society.

[22] T. K. Ho. Random decision forests. In *Proc. of the 3rd Int'l Conference on Document Analysis and Recognition*, pages 278–282, Montreal, Canada, August 1995.

[23] William N.N. Hung and Xiaoyu Song. Bdd variable ordering by scatter search. In *ICCD '01: Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, page 368, Washington, DC, USA, 2001. IEEE Computer Society.

[24] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, June 2005.

[25] David D. Lewis and Jason Catlett. Heterogeneous uncertainty sampling for supervised learning. In William W. Cohen and Haym Hirsh, editors, *Proceedings of ICML-94, 11th International Conference on Machine Learning*, pages 148–156, New Brunswick, US, 1994. Morgan Kaufmann Publishers, San Francisco, US.

[26] David D. Lewis and William A. Gale. A sequential algorithm for training text classifiers. In W. Bruce Croft and Cornelis J. van Rijsbergen, editors, *Proceedings of SIGIR-94, 17th ACM International Conference on Research and Development in Information Retrieval*, pages 3–12, Dublin, IE, 1994. Springer Verlag, Heidelberg, DE.

[27] Ondřej Lhoták and Laurie Hendren. Jedd: A BDD-based relational extension of Java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.

[28] Jorn Lind-Nielsen. BuDDy, a binary decision diagram package. http://www.itu.dk/research/buddy/, 2004.

[29] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In *Proceedings of Programming Languages and Systems: Third Asian Symposium*, November 2005.

[30] Yuan Lu, Jawahar Jain, Edmund M. Clarke, and Masahiro Fujita. Efficient variable ordering using aBDD based sampling. In *Design Automation Conference*, pages 687–692, 2000.

[31] Richard Maclin and David Opitz. An empirical evaluation of bagging and boosting. In *AAAI/IAAI*, pages 546–551, 1997.

[32] D. Margineantu and T. Dietterich. Improved class probability estimates from decision tree models, 2002.

[33] M. Ray Mercer, Rohit Kapur, and Don E. Ross. Functional approaches to generating orderings for efficient symbolic representations. In *Design Automation Conference, 1992. Proceedings., 29th ACM/IEEE*, pages 624–627, June 1992.

[34] Ion Muslea, Steven Minton, and Craig A. Knoblock. Selective sampling with redundant views. In *AAAI/IAAI*, pages 621–626, 2000.

[35] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *In Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006.

[36] National Institute of Standards & Technology. The economic impacts of inadequate infrastructure for software testing. May 2002.

[37] Shipra Panda and Fabio Somenzi. Who are the variables in your neighborhood. In *ICCAD '95: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 74–77, Washington, DC, USA, 1995. IEEE Computer Society.

[38] Csar Ferri Peter. Improving the auc of probabilistic estimation trees, 2003.

[39] Foster Provost and Pedro Domingos. Tree induction for probability-based ranking. *Mach. Learn.*, 52(3):199–215, 2003.

[40] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, 1986.

[41] J.R. Quinlan. Bagging, boosting, and c4.5. In *Proceedings of the 13th National Conference on Artificial Intelligence*, pages 725–730, 1996.

[42] R. Ranjan, A. Aziz, R. Brayton, B. Plessier, and C. Pixley. Efficient bdd algorithms for fsm synthesis and verification. 1995.

[43] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93*, pages 42–47, 1993.

[44] Zsofi Ruttkay. Fuzzy constraint satisfaction. In *Proceedings 1st IEEE Conference on Evolutionary Computing*, pages 542–547, Orlando, 1994.

[45] Maytal Saar-Tsechansky and Foster Provost. Active learning for class probability estimation and ranking. In *Proceedings of the 17th Intl. Joint Conf. on Machine Learning*, pages 911–920, 2001.

[46] Maytal Saar-Tsechansky and Foster Provost. Active sampling for class probability estimation and ranking. *Mach. Learn.*, 54(2):153–178, 2004.

[47] Gerard Salton and Chris Buckley. Improving retrieval performance by relevance feedback. Technical report, Cornell University, 1988.

[48] H. S. Seung, M. Opper, and H. Sompolinsky. Query by committee. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 287–294. ACM Press, 1992.

[49] N. J. A. Sloane. The on-line encyclopedia of integer sequences: A000670. http://www.research.att.com/as/sequences, 2003.

[50] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.

[51] John Whaley. Joeq: A virtual machine and compiler infrastructure. In *Proceedings of the SIGPLAN Workshop on Interpreters, Virtual Machines, and Emulators*, pages 58–66, June 2003.

[52] John Whaley. JavaBDD library, 2004. http://javabdd.sourceforge.net.

[53] John Whaley, Dzintars Avots, Michael Carbin, and Monica Lam. Using datalog with binary decision diagrams for program analysis. In *Proceedings of Programming Languages and Systems: Third Asian Symposium*, November 2005.

[54] Robert P. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[55] Bianca Zadrozny and Charles Elkan. Obtaining calibrated probability estimates from decision trees and naive Bayesian classifiers. In *Proc. 18th International Conf. on Machine Learning*, pages 609–616. Morgan Kaufmann, San Francisco, CA, 2001.

[56] Z. H. Zhou, D. Wei, G. Li, and H. Dai. On the size of training set and the benefit from ensemble. In *Proceedings of the 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04)*, pages 298–307. LNAI, 2004.